



Brandenburgische Technische Universität Cottbus (BTU)

Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln

Diplomarbeit am Lehrstuhl für Praktische Informatik / Grafische Systeme

Prof. Dr. Winfried Kurth

Institut für Informatik,

Informations- und Medientechnik

Brandenburgische Technische Universität Cottbus (BTU)

von

cand. Dipl. Inform.

Michael Henke

Betreuer:

Prof. Dr. Winfried Kurth

Dipl. Phys. Ole Kniemeyer

Cottbus, den 21. Dezember 2006

erweiterte und überarbeitete Version

13. August 2007



Lehrstuhl für Praktische Informatik /
Grafische Systeme

Erklärung

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass ich die von mir eingereichte Diplomarbeit

Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln

eigenhändig, selbstständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst habe. Sollten sonstige Hilfsmittel zum Einsatz gekommen sein, so wurden diese explizit erwähnt. Aus der angegebenen Literatur übernommene Textpassagen wurden entsprechend gekennzeichnet. Weiterhin wurde diese Arbeit keiner anderen Prüfungsbehörde, die für die Vergabe eines akademischen Grades zuständig ist, vorgelegt.

Cottbus, den 21. Dezember 2006

MICHAEL HENKE

Danksagung

Diese Diplomarbeit entstand von Juni bis Dezember 2006 an der Brandenburgische Technische Universität Cottbus am Lehrstuhl für Praktische Informatik / Grafische Systeme.

An dieser Stelle möchte ich mich bei allen Personen für die vielseitige Unterstützung während meiner Diplomarbeit bedanken. Einen besonderen Dank verdienen die folgenden Personen:

- Prof. Dr. Winfried Kurth, für eine interessante Aufgabenstellung und die Unterstützung beim Erstellen dieser Arbeit.
- Dipl. Phys. Ole Kniemeyer, für die Betreuung der Diplomarbeit und seine zahlreichen Anregungen und Erklärungen rund um die Aufgabe und die GroIMP-Software.
- Guido Richter möchte ich für die freundliche Unterstützung beim Korrekturlesen danken.

Inhaltsverzeichnis

Aufgabenstellung	1
1 Einleitung	3
1.1 Motivation	3
1.2 Vorgehen	4
1.3 Gliederung der Arbeit	4
2 3D-Pflanzenmodellierung	7
2.1 Zelluläre Automaten	7
2.2 Ein erstes kontinuierliches Modell	8
2.3 Lindenmayer-Systeme (L-Systeme)	9
2.3.1 Textersetzungssysteme	9
2.3.2 Grafische Interpretation	10
2.3.3 Geklammerte L-Systeme	12
2.3.4 Sonstige Erweiterungen der L-Systeme	13
2.3.5 Grenzen von Lindenmayer-Systemen	15
2.4 Prozedurales Modellieren	17
2.5 Partikelsysteme	19
2.6 Ein fraktales Baummodell	19
2.7 Meristem-orientierte Pflanzenmodellierung	20
2.8 Bäume aus Strängen (Strang-Systeme)	22
2.9 Iterierte Funktionssysteme (IFS)	23
2.10 Objektinstanzierung	25
2.11 CSG-basierte Modellierung	26
2.12 Einordnung der Verfahren	27
3 Regelbasierte Objekterzeugung	29
3.1 Komponententypen	30
3.1.1 Geometrieerzeugende Komponenten	31
3.1.2 Multiplizierende Komponenten	32
3.1.3 Globale Modellierung	35
3.2 Kombination von Komponenten	36

3.3	Beispiel	36
4	Integration der regelbasierten Objekterzeugung in GroIMP	39
4.1	GroIMP und XL	39
4.2	Instanzierungsregeln	41
5	Bausteinkatalog	45
5.1	Attributtypen	45
5.2	Baustein-Attribute	49
5.3	Kombination von Komponenten	50
5.4	Horn	51
5.5	Tree	56
5.6	Wreath	61
5.7	Hydra	61
5.8	PhiBall	63
5.9	Variation	65
5.10	BlockColor	67
5.11	BlockScale	68
5.12	Arrange	70
5.12.1	Geometrische Anordnungen	76
5.12.2	Wahrscheinlichkeitsanordnungen	78
5.12.3	Halbtonverfahren	82
5.12.4	Zusätzliche Verfahren	87
6	Level of Detail (LOD)	91
6.1	LOD-Modellierung in GroIMP	93
6.2	Umgesetzte LOD-Methoden	97
7	Beispielanwendungen	101
7.1	Der Modellierungsprozess	101
7.2	Beispiel Gerbera	103
7.3	Beispiel Import eines Xfrog-Modells: beta1.xfr	108
7.4	Arrange-Beispiele	109
7.4.1	Arrange-Demo	109
7.4.2	Arrange-Anwendung: Waldmodellierung	111
7.5	Modellierung eines Baumes	118
7.6	Kleines Farn-Modell	123
7.7	Kombination von Instanzierungsregeln mit Graph-Grammatiken	127
7.8	Weitere Beispielmodelle	127
8	Diskussion und Ausblick	131

8.1	Diskussion	131
8.2	Ausblick und weitergehende Aufgaben	133
A	Implementierungsstruktur	139
B	Funktionen	141
C	Funktionsparser	147
C.1	Grammatik	147
C.2	Beschreibung der Funktionen	149
D	Xfrog FileParser	153
D.1	Grammatik	154
	Literaturverzeichnis	155

Abbildungsverzeichnis

2.1	Diskrete Verzweigungsmuster nach Ulam	8
2.2	Kontinuierliche Verzweigungsstrukturen nach Cohen (Bild: D. Cohen) . . .	9
2.3	Turtle-Interpretation der Kochkurve bei $n = 1$	11
2.4	Grafische Interpretation der Kochkurve in verschiedenen Ableitungsebenen	12
2.5	Entwicklung eines Blattes in verschiedenen Ableitungen	15
2.6	L-System Rotbuche nach Kurth (wgrogra v. 2.4: bu4.lsy)	16
2.7	Grafische Interpretation Rotbuche	16
2.8	Verzweigungsstruktur nach Honda (Bild: Honda)	18
2.9	Strukturierte Partikelsysteme nach Reeves und Blau (Bilder: W. Reeves) .	19
2.10	Fraktales Baummodell (Bild: Oppenheimer)	21
2.11	Simulation von Knospung (Bild: De Reffye)	21
2.12	Baummodell aus Strängen (Bild: M. Holton)	23
2.13	Erzeugung des Sierpinski-Dreiecks (Tiefe 0, 1, 2, 3, 6, 9)	23
2.14	Generierung eines Farnblattes (Bild: M. Barnsley)	25
2.15	Körperkonstruktion mittels CSG-Graphen	26
2.16	Verschiedene Konstruktionsvarianten eines Körpers	27
3.1	Geometrie-Erzeugung (Bild: O. Deussen)	30
3.2	Übersicht über die Xfrog-Komponententypen	30
3.3	Definition verschiedener Blattgeometrien (Bild: O. Deussen)	31
3.4	Verschiedene Parametervariationen einer Baumkomponente (Bild: O. Deussen)	32
3.5	Orientierung der multiplizierenden Komponenten (Bild: O. Deussen)	32
3.6	Phyllotaxis-Anwendungen	33
3.7	Anordnung nach dem Goldenen Schnitt nach Vogle	34
3.8	Verteilung nach dem Goldenen Schnitt auf einem Kugelausschnitt	34
3.9	Beispiel: Sonnenblume (Bild: Deussen)	37
4.1	Unterschiedliche Arbeitsweisen in der Pflanzenstrukturmodellierung (nach Kurth)	40
4.2	Beispiel Graphersetzung (nach Kniemeyer)	41
4.3	Instanzierter Baum	42

4.4	Grafische Resultat der Anwendung der Wreath-Instanzierung auf ein Cone-Objekt	43
5.1	Funktionsfeld und Komplexfeld	46
5.2	Funktionsinterpolation zwischen zwei Werten in einem Komplexfeld; Funktionswert1=3, Funktionswert2=7.5	47
5.3	Random-Feld des Arrange-Bausteins	48
5.4	LocationParameter-Feld-Aufbau	50
5.5	Auswirkung von child- und multiply-Kanten	51
5.6	Auswirkung der Rotations- und Translationsattribute (Bilder: Deussen)	53
5.7	Verschiedene Range-Einstellungen bei Horn	54
5.8	Verschiedene Zweigmanipulationen des Horn-Bausteins (Bilder: Deussen)	55
5.9	Beispiel für Horn-Objekt	56
5.10	Verschiedene Arrangement-Einstellungen des Tree-Bausteins	59
5.11	Tree Branches Distribution-Berechnung	60
5.12	Verschiedene Neigungseinstellungen des Hydra-Bausteins	63
5.13	PhiBall-Manipulationen	65
5.14	Verschiedene Influence-Einstellungen bei PhiBall	66
5.15	Verschiedene Type-Einstellungen des Variations-Bausteins	67
5.16	Beispiel für die Bausteine BlockColor und BlockScale	69
5.17	Beispiel für verschiedene Terrains	70
5.18	Berechnung Dichtefeld: Ausgangssituation	73
5.19	Berechnung Dichtefeld: Clipping-Zwischenschritt	73
5.20	Berechnung Dichtefeld: Wahrscheinlichkeit einer Belegung	74
5.21	Wirkung des <i>densityI</i> -Attributs auf die Objektpositionierung	75
5.22	Attributfeld der geometrischen Anordnungen	76
5.23	Zugrunde liegende Beispielverteilung	80
5.24	Auswertung der Verteilung	80
5.25	Attributfeld der Wahrscheinlichkeitsanordnungen	80
5.26	Unterteilung der Grundfläche bei verschiedenen <i>hierarchic</i> -Einstellungen im Arrange-Baustein, <i>Level</i> = 1	82
5.27	Attributfeld der Halbtonverfahren	82
5.28	Anwendung des Lloyd-Algorithmus auf eine Punktmenge (Bilder: Deussen)	89
5.29	Beispielkachelung: <i>Number</i> = 5, <i>Level</i> = 4	90
6.1	Verschiedene Detailstufen einer Kugel	92
6.2	LOD-Demonstration für eine einfache baumähnliche Struktur nach Beaudoin	93
6.3	LOD-Darstellung eines Zweiges	93
6.4	GroIMP-LOD-Menü	94
6.5	Verschiedene Detailstufen eines Horn-Objekts	96

6.6	Zustandekommen der Detailstufeneinstellung	97
6.7	<i>scale</i> -LOD-Methode an einem Horn-Beispiel demonstriert	98
7.1	Texturen der einzelnen Modellteile	103
7.2	Aufbau des Gerberamodells	104
7.3	Beispielübersicht der Gerberamodelle	106
7.4	Vergleich: Xfrog-Modell vs. nach GroIMP importiertes Xfrog-Modell	108
7.5	p-Graph zum Arrange-Beispiel	110
7.6	Die dem Arrange-Beispiel zugrunde liegenden Felder	110
7.7	Das Arrange-Beispiel mit den Zwischenstufen	112
7.8	Aufgliederung der Standortparameter	113
7.9	Das Standortparameterfeld <i>LocationParameter field</i>	113
7.10	Auswirkung der Standortparameter auf das Wachstum	114
7.11	Graph der Längen- und Durchmesserfunktion	115
7.12	Ergebnis der Waldmodellierung	116
7.13	Das Kronenmodell in Gitternetzdarstellung	116
7.14	Anordnungsvarianten im Waldmodell	118
7.15	Verschiedene Standortparameterfelder	118
7.16	Ergebnisse der Waldmodellierungen mit verschiedenen Standortparameterfeldern	119
7.17	Fertiges Modell: <i>TreeEx.gsz</i>	120
7.18	Struktur und Geometrie des Baummodells	121
7.19	Verwendete Texturen	121
7.20	Skelett des fertigen Modells	123
7.21	Texturen der einzelnen Modellteile	124
7.22	Der Ast eines Farnwedels	124
7.23	Gegenüberstellung	126
7.24	Beispielmodelle	128
7.25	Fortsetzung Beispielmodelle 1	129
7.26	Fortsetzung Beispielmodelle 2	130
8.1	Beispiel für Behaarung auf Pflanzenmodellen nach Prusinkiewicz [47]	134
8.2	Xfrog-Deformationskomponenten	135
8.3	Verschiedene <i>Segments</i> -Einstellungen beim Horn-Objekt	135
8.4	Beispielanordnung verschiedener Pflanzenspezies	136

Tabellenverzeichnis

2.1	Interpretation der Turtle-Befehle in 2D	11
2.3	Beispiele von Turtle-Interpretationen geklammerter L-Systeme	12
2.5	Definition der affinen Abbildungen	23
5.1	Tabellarisch erfasste Daten	79
5.3	Übersicht der bereitgestellten Wahrscheinlichkeitsfunktionen	83

Aufgabenstellung

Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln

Die am Lehrstuhl Grafische Systeme in Entwicklung befindliche, in Java implementierte 3D-Plattform GroIMP enthält mit der integrierten Programmiersprache XL einen Formalismus zur Modellierung (nicht nur) pflanzlicher Strukturen. Die Programmiersprache XL erlaubt es, Instanzierungsregeln zu spezifizieren, welche Baumstrukturen (im Sinne der Informatik) auf Anfrage (etwa bei 3D-Zeichen) aus einer kompakten Beschreibung erzeugen. Die Beschreibung kann Bezug auf einen Graphen nehmen, hiermit kann die Funktionsweise der regelbasierten Objekterzeugung von Xfrog erreicht werden. Es existieren aber noch keine Implementationen solcher an Xfrog angelehnten Funktionen. In der Diplomarbeit sollen diese geschaffen werden. Des Weiteren sollen Level-Of-Detail-Techniken in den Instanzierungsregeln integriert werden, welche als möglichst glatte Funktionen des Abstandes zur Kamera verschiedene Auflösungsstufen des Modells instanzieren. Eine interessante Möglichkeit hierfür bietet ein Ansatz von B. Lintermann, der die Selbstähnlichkeit von Pflanzen ausnutzt.

Die Implementation der Instanzierungsklassen soll idealerweise mithilfe der XL-Instanzierungsregeln erfolgen. Sie kann aber auch direkt in Java erfolgen, wenn dieses etwa aus Gründen der Laufzeit erforderlich sein sollte. Es sollen Äquivalente zu den Xfrog-Komponenten Horn, Baum, Hydra, Wreath und Phiball geschaffen werden. Eine Blattkomponente ist nicht unbedingt erforderlich, da sie eine Terminalstellung im Instanzierungsgraphen besitzt und an solchen Stellen auch die existierenden GroIMP-Klassen genutzt werden können. Für die geometrienerzeugenden Komponenten Horn und Baum sind GroIMPs NURBS-Funktionen zu verwenden. Neben den Xfrog-Funktionen sollen weitere sinnvolle Komponenten entworfen und implementiert werden. Dazu soll wenigstens eine Komponente zum Platzieren mehrerer Pflanzen-Instanzen anhand einer stochastischen Verteilung gehören. Diese Komponente soll Terraindaten von GroIMP berücksichtigen (Höhe, Bodenbeschaffenheit); diese Daten sollen auch die erzeugten Pflanzen beeinflussen. Dazu müssen die pflanzenerzeugenden Komponenten auf einen Satz von Attributen zur Bodenbeschaffenheit zurückgreifen können und durch diesen gesteuert werden.

Ein Import von Xfrog-Dateien ist wünschenswert. Diese besitzen ein hierarchisches Klartext-Format und sind somit gut zu entschlüsseln.

Die Leistungsfähigkeit des implementierten Baukastens soll anhand der Nachbildung einiger Standardbeispiele von Xfrog demonstriert werden. Diese sollen auf www.grogra.de verfügbar gemacht werden.

1 Einleitung

Das Ziel der hier vorliegenden Diplomarbeit ist es, einen Baukasten zur 3D-Pflanzenvisualisierung zu entwerfen und zu implementieren, und ihn als Teil in die bereits bestehende Modellierungs- und Simulationssoftware GroIMP zu integrieren. Das Prinzip des Baukastens ist die Nutzung der später näher zu erläuternden Instanzierungsregeln.

1.1 Motivation

Fast überall in unserer Umgebung sehen wir Pflanzen, sie gehören zu unserem Leben. So kommt kaum eine Präsentation eines Architekten oder Landschaftsplaners, der die Pläne seiner neuesten Entwürfe vorstellt, heute ohne Bäume, Stäucher oder Blumen aus. Sie verleihen ihren Modellen erst das Leben, das es ihrem Auftraggeber ermöglicht, sich vorzustellen, wie sich der Bau in die Umwelt integriert. Vor allem Entwicklungssimulationen, wie sich beispielsweise der Vorgarten eines Einfamilienhauses in 10 oder 15 Jahren verändert, wenn man die eine oder andere Baumart pflanzt, helfen Entscheidungen zu treffen. Planungsalternativen können visuell abgewogen werden. Neue Computerspiele, welche Wert auf ausgefeilte Grafiken legen, die sich am besten nicht wiederholen, sich anpassen, auf Eingriffe reagieren und so natürlich wie möglich aussehen sollen, verzichten heute kaum auf computergenerierte Pflanzen.

Diese einführenden Beispiele zeigen, wie umfangreich die Aufgabenpalette und die Anwendungsmöglichkeiten für 3D-Pflanzenmodelle sind. Zusammenfassend lassen sich die Aufgaben folgendermaßen unterteilen:

- Erzeugung von Pflanzen jeder Art
- Wachstumssimulation einzelner Individuen
- Simulation von Populationen

Dabei werden eine ganze Reihe weiterer Anforderungen an diese Modelle gestellt. Eine der Hauptanforderungen ist meist eine möglichst genaue Simulation der Natur, im Äußeren wie im Inneren. Wobei hier unter "Äußerem" das rein optische Erscheinungsbild und unter "Innerem" u. a. der Stoffwechselkreislauf, Zellteilungen, Wasser-, Hormon- und Nährstofftransporte und die Interaktion mit der Umwelt zu verstehen sind. Ebenfalls zu

den inneren Anforderungen zählt die lokale Sensitivität, die z. B. dafür sorgt, dass sich die Äste eines Baumes unter ihrem eigenen Gewicht neigen, wenn sie wachsen. Die globale Sensitivität, die Einflüsse wie Sonneneinstrahlung und Wind umfasst, fällt ebenfalls unter die inneren Anforderungen.

In dieser Arbeit wird vor allem die Erzeugung von Pflanzen untersucht. Dazu wird mit den Instanzierungsregeln ein intuitives Verfahren vorgestellt, mit dem in wenigen Schritten bereits eindrucksvolle Modelle erzeugt werden können. Ohne weiteres können ebenfalls nicht pflanzliche Strukturen mit diesem Verfahren erzeugt werden.

1.2 Vorgehen

Zur Erfüllung der Aufgabenstellung wurde in einem ersten Schritt die Software Xfrog von Deussen und Lintermann der Firma greenworks¹, im speziellen die vorhandenen Bausteine, genau analysiert. Parallel dazu wurde versucht, die gewonnenen Erkenntnisse in mathematische Modelle umzusetzen und sie anschließend in XL² zu modellieren. Der nächste Schritt war es, die XL-Bausteine in Java zu implementieren und neue, in Xfrog nicht vorhandene, Bausteine zu entwickeln und zu implementieren.

In einem weiteren Schritt wurde ein Level of Detail-Konzept entwickelt und in die Bausteine integriert. Abschließend wird die Leistungsfähigkeit und die Funktionsweise an einigen ausgewählten Beispielen demonstriert.

1.3 Gliederung der Arbeit

Nach diesem einleitenden Kapitel erfolgt in Kapitel zwei eine Übersicht über geläufige Verfahren zur 3D-Pflanzenmodellierung. Aus den geschilderten Vor-/Nachteilen wird die Motivation für die der Arbeit zugrunde liegenden Instanzierungsregeln erarbeitet.

Das dritte Kapitel stellt das in dieser Diplomarbeit verwendete Verfahren ausführlich vor. Der Entwurf der Bausteine wird diskutiert und die Arbeitsweise des Verfahrens veranschaulicht.

Die Einordnung des Verfahrens in die bestehende Modellierungs- und Simulationssoftware GroIMP und in die ihr zugrunde liegende Programmiersprache XL wird in Kapitel 4 vorgenommen.

In Kapitel 5 werden die implementierten Bausteine mit all ihren Attributen ausführlich beschrieben und anhand kleiner Beispiele veranschaulicht.

¹siehe dazu www.greenworks.de greenworks organic-software

²”extended L-system language”, in die Software GroIMP integrierte Programmiersprache

Das sechste Kapitel stellt verschiedene Verfahren der Detailstufensteuerung (Level of Detail, kurz LOD) vor und beschreibt die in den Bausteinen implementierte LOD-Methode anschließend.

Im siebenten Kapitel sollen einige Beispiele das Vorgehen bei der Modellierung und die Mächtigkeit der implementierten Bausteine demonstrieren.

Das letzte Kapitel fasst die Ergebnisse der Arbeit kurz zusammen, diskutiert sie und gibt einen Ausblick auf weiterführende Arbeiten.

Im Anhang A wird die Implementierungsstruktur des Bausteinkatalogs erläutert. Eine Beschreibung der mathematischen Funktionen sowie des integrierten Funktionsparsers sind als Anhänge hinzugefügt. In Anhang D ist der Xfrog FileParser beschrieben, der es ermöglicht, Xfrog-Dateien (**.xfr*) zu öffnen und die Modelle nach GroIMP zu importieren.

2 3D-Pflanzenmodellierung

Im folgenden Kapitel werden kurz eine Reihe geläufiger Verfahren zur 3D-Pflanzenmodellierung vorgestellt sowie ihre jeweiligen Vor- und Nachteile erläutert.

Botaniker und Informatiker beschäftigen sich seit über 40 Jahren mit der synthetischen Erzeugung von Bildern pflanzenähnlicher Objekte. Bereits 1966 entstanden, basierend auf zellulären Automaten, Verfahren zur Erzeugung einfacher Verzweigungsstrukturen. Dan Cohen veröffentlichte im Jahre 1967 ein erstes kontinuierlich wachsendes Pflanzenmodell mit relativ realistisch aussehender Verzweigungsstruktur.

Die im Jahre 1968 von Aristid Lindenmayer eingeführten Textersetzungssysteme waren ein weiterer bedeutender Schritt. Das erste parametrisierbare Modell zur Herstellung dreidimensionaler Baumstrukturen wurde 1971 von Hashimoto Honda veröffentlicht.

Aber erst Anfang der achtziger Jahre mit wachsender Rechner- und Grafikleistung gewannen die Modelle eine breitere Aufmerksamkeit außerhalb der Wissenschaft. Sie starteten ihren heute nicht mehr weg zu denkenden Einzug z. B. in die Welt der Computeranimationen und der Computerspiele.

Auf eine Einteilung der Verfahren, wie z. B. von Prusinkiewicz [45] in struktur- und raumorientierte Modelle, oder wie von Deussen [11] in prozedurale und regelbasierte Modellierung, wird am Ende des Kapitels kurz eingegangen.

Die hier getroffene Auswahl orientiert sich vor allem an Deussen [11], sie ist unvollständig, versucht aber eine chronologische Reihenfolge einzuhalten.

2.1 Zelluläre Automaten

Aufbauend auf die von von Neumann in den fünfziger Jahren eingeführten zellulären Automaten [42] konstruierte Ulam [54] einfache Verzweigungsmuster.

Konzept von von Neumann:

1. Für die Pflanzenmodellierung wird der zwei- oder dreidimensionale Raum in Zellen gleicher Form und Größe unterteilt.
2. Zuweisung eines Zustandes an jede Zelle
3. Iteratives Verändern der Zustände in Abhängigkeit vom Zustand der Nachbarzellen

Beispiel: Ein 2D-Raum wird in Quadrate unterteilt, die Regel für die Zustandsänderung der Zellen lautet: "Schalte alle Nachbarn einer angeschalteten Zelle an, falls nur 1 Nachbar angeschaltet ist." Das entstehende Muster der ersten drei Iterationen und das Ausgangssystem sind in Abbildung 2.1 zu sehen.

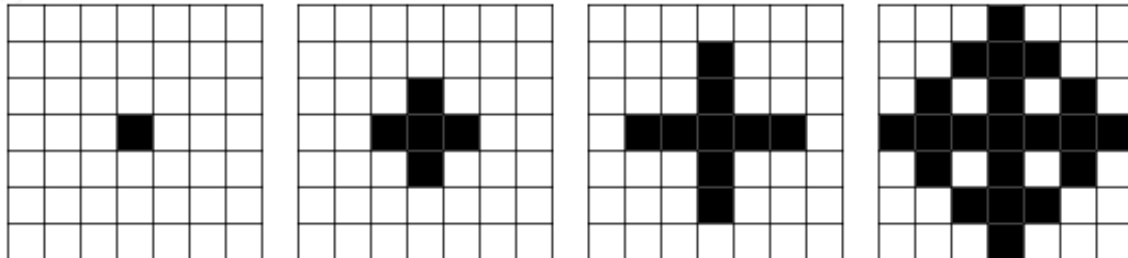


Abbildung 2.1: Diskrete Verzweigungsmuster nach Ulam

Neben der Aufteilung in Würfel können Dreiecke, Tetraeder oder andere Grundobjekte verwendet werden. Auf die sich daraus ergebenden festen Nachbarschaften lassen sich jeweils verschiedene Nachbarschaftsbeziehungen anwenden. So gibt es bei quadratischer Aufteilung im Zweidimensionalen z. B. die von Neumann-Nachbarschaft, die die direkten vier Nachbarn beachtet, und die Moore-Nachbarschaft, bei der alle acht Nachbarn einer Zelle beachtet werden.

2.2 Ein erstes kontinuierliches Modell

Der Botaniker Dan Cohen implementierte erste prozedurale Verfahren zur Modellierung von Verzweigungsstrukturen [7]. Pro Verzweigungsmuster wird ein Fortran-Programm mit einer Menge einfacher Wachstumsvorschriften implementiert. Drei einfache Regeln bestimmen das Wachstum und die Verzweigungsbildung in einer Struktur [11]:

- Das Wachstum findet nur an den Spitzen der Äste statt, d. h. der biologische Mechanismus des Sprosses wird direkt nachgebildet.
- Stärke und Winkel des Wuchses werden durch die aktuelle Richtung, ein lokales Dichtefeld, dessen Gradienten sowie die Resistenz der Struktur gegen Winkelveränderungen bestimmt.
- Die Verzweigungstendenz wird durch ein probabilistisches Maß bestimmt, das neben einem generellen Wert von der Entfernung zur letzten Verzweigung und vom lokalen Dichtefeld abhängt.

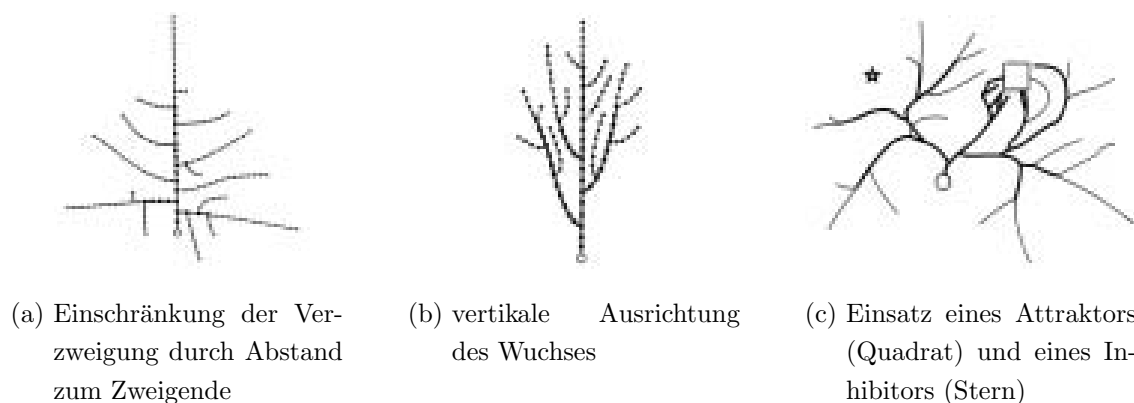


Abbildung 2.2: Kontinuierliche Verzweigungsstrukturen nach Cohen (Bild: D. Cohen)

Der Modellaufbau erfolgt durch eine Folge diskreter Abschnitte, die nach obigen Regeln ausgerichtet werden. Die Auswirkungen, die sich durch die Anwendung unterschiedlicher Regeln ergeben können, sind in Abbildung 2.2 zu sehen.

”Der Ansatz umfasst auch die Änderung von Parameterwerten über die Verzweigungsordnung hinweg, dies erlaubt überhaupt erst eine visuelle Differenzierung in Verzweigungsordnungen und wird in vielen der nachfolgenden Arbeiten eingesetzt.” [11] Mit dieser Arbeit wies Cohen bereits auf die Mächtigkeit von Regelsystemen und auf die Rolle, die diese Verzweigungsstrukturen für die Botanik spielen können, hin.

2.3 Lindenmayer-Systeme (L-Systeme)

Die von Aristid Lindenmayer (1025-1989) eingeführten und später nach ihm benannten Lindenmayer-Systeme (L-Systeme) stellten im Jahre 1968 einen weiteren wesentlichen Schritt in der Pflanzenmodellierung dar [36]. Der Botaniker an der Universität Utrecht suchte eine Methode zur Beschreibung von einfachen Zellorganismen, im speziellen von fadenförmigen Algen.

Mittels eines grammatikalischen Modells, einigen Erweiterungen und der Verwendung einer grafischen Interpretation können L-Systeme dazu verwendet werden, höhere Pflanzen und deren Strukturen darzustellen.

2.3.1 Textersetzungssysteme

L-Systeme sind parallele Textersetzungssysteme. Ein Ersetzungssystem basiert auf der Definition von komplexen Objekten durch die Substitution einfacher Objekte mit Hilfe von Ersetzungsregeln. Dabei wird ein Startwort mittels der Anwendung von Ersetzungsregeln

in ein neues Wort transformiert. Auf das so erzeugte Wort werden erneut die Ersetzungsregeln angewendet. Dies geschieht über mehrere Iterationen hinweg, bis die gewünschte Iterationstiefe erreicht wird.

Beispiel: Startwort = F ; Ersetzungsregel $F \Rightarrow F + F - F + F$

Die Klammerungen dienen nur zur Verdeutlichung, welche Symbole ersetzt wurden, und haben mit dem Ersetzungssystem nichts zu tun.

Iteration	Wort
0	F
1	$F + F - F + F$
2	$(F + F - F + F) + (F + F - F + F) - (F + F - F + F) + (F + F - F + F)$
3	$((F + F - F + F) + (F + F - F + F) - (F + F - F + F) + (F + F - F + F)) +$ $((F + F - F + F) + (F + F - F + F) - (F + F - F + F) + (F + F - F + F)) -$ $((F + F - F + F) + (F + F - F + F) - (F + F - F + F) + (F + F - F + F)) +$ $((F + F - F + F) + (F + F - F + F) - (F + F - F + F) + (F + F - F + F))$
4	...

L-Systeme: Formalismus

Spezielle Grammatiken der Form $L = (\Sigma, \alpha, IP)$

Σ das Alphabet, eine nicht leere, endliche Menge von Zeichen

α ein Startwort aus Σ^+

IP eine Menge von Produktionsregeln der Form: Symbol \Rightarrow Folge von Symbolen,

$$IP \in \Sigma \times \Sigma^*$$

Wobei die Anwendung der Produktionsregeln IP , beim Übergang vom Zeitpunkt t zum Zeitpunkt $t + 1$, parallel auf alle Symbole der Zeichenkette erfolgt. Symbole, auf die keine Regeln anwendbar sind, werden unverändert übernommen.

2.3.2 Grafische Interpretation

Eine mögliche grafische Interpretation der durch die Ersetzungssysteme erzeugten Zeichenketten ist die der so genannten Turtle-Grafiken.

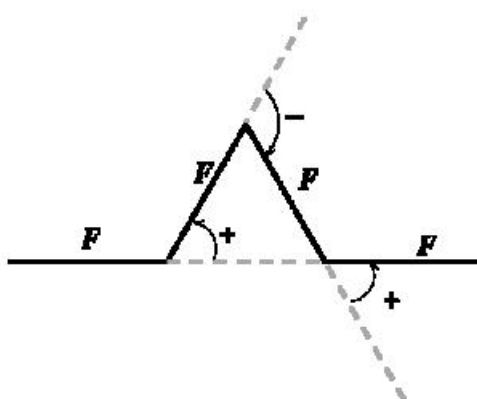
Die Turtle, eine Art virtueller Zeichenstift, wird durch das Tripel (x, y, α) in 2D bzw. durch das 4-Tupel (x, y, z, α) in 3D beschrieben, wobei die kartesischen Koordinaten (x, y, z) die Position und der Winkel α die Blickrichtung der Turtle angeben. Die Turtle interpretiert Grundsymbole wie in Tabelle 2.1 gezeigt (für eine vollständige Übersicht der Turtle-Befehle siehe [37]).

Tabelle 2.1: Interpretation der Turtle-Befehle in 2D

F	Bewegt die Turtle um einen Schritt der Länge d und zeichnet eine Linie.	
f	Bewegt die Turtle um einen Schritt der Länge d und zeichnet keine Linie.	
+	Dreht die Blickrichtung der Turtle um den Winkel δ nach rechts.	
-	Dreht die Blickrichtung der Turtle um den Winkel δ nach links.	

Interpretiert man das in Kapitel 2.3.1 Textersetzungs-systeme gezeigte Beispiel, bei gegebenem Winkel $\delta = 60^\circ$, so erhält man die bekannte Koch-Kurve (Schneeflocken-Kurve) [31]. Die im ersten Ersetzungsschritt erzeugte Zeichenkette $F + F - -F + F$ wird folgendermaßen interpretiert:

Ziehe einen Strich der Länge d , ändere die Blickrichtung der Turtle um 60° im Uhrzeigersinn. Ziehe wieder einen Strich. Ändere die Blickrichtung der Turtle um 120° gegen den Uhrzeigersinn und ziehe dann wieder einen Strich. Ändere nun die Blickrichtung der Turtle um 60° im Uhrzeigersinn und ziehe den letzten Strich. Wichtig ist hierbei, die Winkeländerungen immer aus der aktuellen Blickrichtung der Turtle zu betrachten. Es ergibt sich so das in Abbildung 2.3 gezeigte Bild.

Abbildung 2.3: Turtle-Interpretation der Kochkurve bei $n = 1$

Die Interpretationen der weiteren Ableitungsschritte ergeben folgende Bilder 2.4:

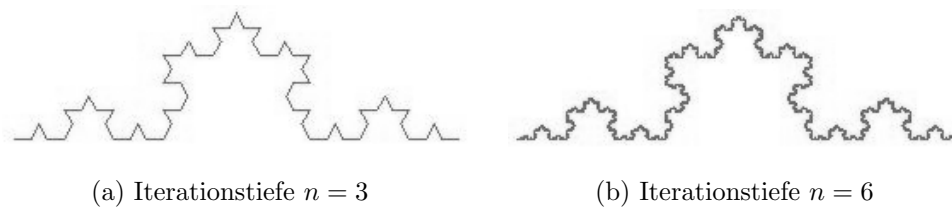


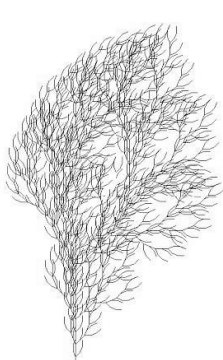
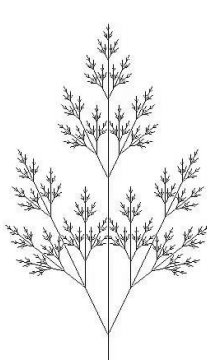
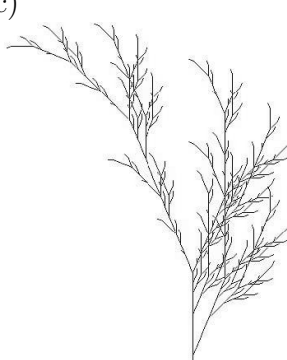
Abbildung 2.4: Grafische Interpretation der Kochkurve in verschiedenen Ableitungsebenen

2.3.3 Geklammerte L-Systeme

Eine weitere wichtige Erweiterung der bisher vorgestellten L-Systeme, mit denen bis jetzt nur Linienzüge erzeugt werden können, ist die Einführung eines Stack. Das Alphabet wird um die Symbole [und] erweitert, wobei [den aktuellen Zustand der Turtle auf einem Stack speichert und] das obere Element vom Stack holt und es zum neuen Zustand der Turtle macht.

Diese Erweiterung ermöglicht Verzweigungsstrukturen, mit denen einfache pflanzenähnliche Modelle erzeugt werden können, wie in Tabelle 2.3 gezeigt.

Tabelle 2.3: Beispiele von Turtle-Interpretationen geklammerter L-Systeme

<p>a)</p> 	<p>b)</p> 	<p>c)</p> 
<p>n=4 $\delta = 22.5^\circ$ $\alpha = F$ $F \Rightarrow FF - [-F + F + F]$ $+ [+F - F - F]$</p>	<p>n=7 $\delta = 25.7^\circ$ $\alpha = X$ $X \Rightarrow F[+X][-X]FX$ $F \Rightarrow FF$</p>	<p>n=5 $\delta = 22.5^\circ$ $\alpha = X$ $X \Rightarrow F - [[X] + X] + F[+FX] - X$ $F \Rightarrow FF$</p>

2.3.4 Sonstige Erweiterungen der L-Systeme

Die folgenden Erweiterungen werden der Vollständigkeit halber erwähnt, auf eine ausführliche Erläuterung wird an dieser Stelle verzichtet und auf die umfangreiche Literatur verwiesen. Besonders zu empfehlen ist das Standardwerk von Prusinkiewicz und Lindenmayer "The Algorithmic Beauty of Plants" [37].

So lässt sich mittels **stochastischer L-Systeme** der Determinismus der bisherigen L-Systeme überwinden. Alle Pflanzen, die mit demselben deterministischen L-System erzeugt wurden, sind identisch. Um eine in der Natur nicht vorkommende Gleichmäßigkeit zu vermeiden, müssen zwischen den einzelnen Pflanzen Abweichungen eingeführt werden, welche Einzelheiten verändern, ohne die allgemeine Struktur der Pflanze zu beeinflussen. Dies geschieht, indem jede Produktionsregel mit einem Wahrscheinlichkeitswert belegt wird, welcher über ihre Anwendung entscheidet.

Ein weiteres Problem der bisherigen L-Systeme liegt darin, dass die Längen aller vorkommenden Linien als ein ganzzahliges Vielfaches der Basislänge d modelliert werden müssen. Gleiches gilt für den Basiswinkel δ . Diese Einschränkung begrenzt die Möglichkeiten dessen, was darstellbar ist, erheblich. Man versuche nur, ein rechtwinkliges, gleichschenkliges Dreieck darzustellen. Es wird nicht gelingen, da z. B. bei einer Länge der Katheten von 1 die Hypotenuse die Länge $\sqrt{2}$ hat [37]. Abhilfe schaffen so genannte **parametrische L-Systeme**, wie sie von Prusinkiewicz [37, 46] formuliert wurden. Bei ihnen ist es möglich, den Produktionsregeln Parameter anzuhängen, die dann ihrerseits die Turtle beeinflussen.

Kontextsensitive L-Systeme ermöglichen eine Einbeziehung der Umwelt. Man unterscheidet globale Sensitivität, wie z. B. hinsichtlich Sonneneinstrahlung, Bewuchsdichte und Eingriffe durch Schädlinge, und lokale Sensitivität, wie z. B. Transport von Nährstoffen oder Hormonen, oder bei der Neigung von Ästen unter ihrem Eigengewicht. Auf diese Weise ist es möglich, einen Informationsfluss innerhalb einer Pflanze zu modellieren. Wachstumsprozesse einzelner Pflanzenteile, wie etwa das sukzessive Aufblühen von Blüten in einem großen Blütenstand oder aber das Verdorren von Teilbäumen eines größeren Baumes, lassen sich somit simulieren.

Eine weitere von Prusinkiewicz und Lindenmayer in [37] eingeführte Erweiterung, die später in anderer Art noch von Interesse wird (s. Kapitel 4), ist die der Angabe von Polygonen, auch "**contour tracing**" genannt. Mit ihr ist es möglich, ganze Pflanzenorgane wie Blütenblätter und Blätter in einem Schritt direkt zu definieren. Die Umrisse eines Objekts werden über einen geschlossenen planaren Polygonzug definiert, der keinerlei Auswirkung auf die Ableitungen hat. Der Interpretierer muss dafür in die Lage versetzt werden, gefüllte Polygone darstellen zu können. Prusinkiewicz führt dazu weitere neue Symbole ein, die folgendermaßen interpretiert werden:

- { Startet ein neues Polygon, indem es ein ggf. vorhandenes Polygon auf den Polygonstack legt und ein neues leeres Polygon erzeugt.
- . Die aktuelle Position der Turtle wird als weiterer Eckpunkt des Polygons gespeichert
- G Wird wie F verwendet, nur dass der Endpunkt nicht zum Polygon hinzugefügt wird.
- } Schließt und zeichnet das aktuelle Polygon mit den angegebenen Punkten, danach wird das nächste Polygon vom Polygonstack geholt und so zum aktuellen gemacht.

Dadurch, dass während der Ableitung des Resultatwortes Ersetzungen vorgenommen werden können, kann eine Entwicklung, z. B. das Wachstum von Blättern, simuliert werden. Zur Veranschaulichung diene folgendes L-System [37], mit $\delta = 9.0^\circ$:

$$\begin{aligned} \alpha & : [A][B] \\ p_1 & : A \rightarrow [+A\{.\}.C.] \\ p_2 & : B \rightarrow [-B\{.\}.C.] \\ p_3 & : GC \end{aligned}$$

Die Abbildung 2.5 zeigt die grafischen Interpretationen unterschiedlicher Ableitungstiefen. Dabei wurden die Polygone ohne Füllung dargestellt, um die Struktur besser zu veranschaulichen.

Aufbauend auf diese Erweiterung wurde 1994 von Kurth eine weitere zusätzliche Regelmenge, die sog. "**Interpretationsregeln**", eingeführt. In erster Linie dienen sie zum direkten Zeichnen von Objekten und stellen somit eine Vorstufe der Turtle-Interpretation da, deren Anwendung erst unmittelbar vor der Interpretation durch die Turtle erfolgt. Interpretationsregeln können durchaus Auswirkungen auf Symbole rechts von ihrer Anwendungsstelle haben. Wird beispielsweise innerhalb einer Interpretationsregel ein Durchmesser definiert, so steuert dieser Wert den Durchmesser aller nachfolgenden Objekte, bis ein neuer Durchmesser festgelegt wird.

Dieses Verfahren ermöglicht es, Strukturen, wie z. B. Knospen und Blätter "fest" vorzugeben, so dass sie nicht über einen aufwändigen Ableitungsprozess erzeugt werden müssen. Unter dem Begriff Homomorphismen benutzen Fournier und Andrieu [18] den Ansatz der Interpretationsregeln für geometrische Spezifikationen in einem Mais-Modell.

Zur besseren Veranschaulichung sei folgendes Beispiel nach Kurth [33] gegeben. Es erlaubt eine Simulation der Zweigentwicklung bei Rotbuchen. Die Interpretationsregeln sind in dieser Notation mittels ## gekennzeichnet, so entstehen die Knospen (bud-Regel) und die Blätter (leaf-Regel) durch Interpretationsregeln. In diesen Regeln wird ein Polygonzug definiert, dessen grafische Interpretation in Abbildung 2.7a und Abbildung 2.7b zu sehen

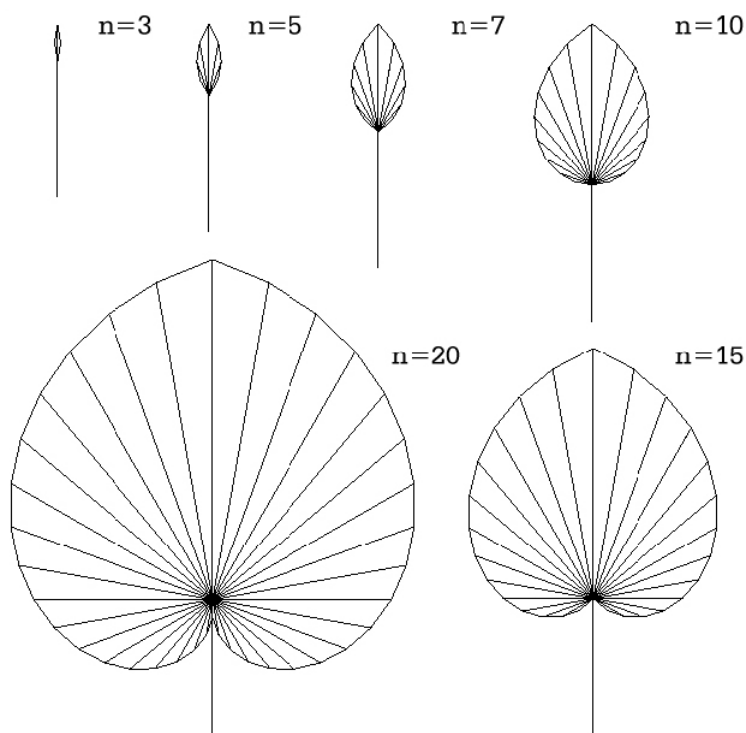


Abbildung 2.5: Entwicklung eines Blattes in verschiedenen Ableitungen

ist. Abweichend vom obigen Formalismus von Prusinkiewicz erfolgt dies durch Speichern von Positionen unter einem Index (S-Befehl) und späteres Verbinden zweier gespeicherter Positionen (C-Befehl, "connect").

2.3.5 Grenzen von Lindenmayer-Systemen

Lindenmayer-Systeme stellen einen mächtigen Formalismus zur Modellierung von Pflanzen und pflanzenähnlicher Strukturen dar, mit dem sich sehr realistische Ergebnisse erzielen lassen. "Der Formalismus von L-Systemen ist einfach und hat seinen ästhetischen Reiz" [11]. Der Modellierer ist aufgrund der Vielzahl von Erweiterungen in der Lage, fast alle Pflanzenarten nachzubilden zu können.

Ein Problem ist, dass bei L-Systemen die Generierung rein theoretisch erfolgt. D. h., dass durch die Erstellung eines teilweise recht komplexen Regelwerkes eine indirekte Darstellung erzeugt wird. "Problematisch ist aber nach wie vor der Modellieraspekt. [...] ist die Herstellung einer vorgegebenen Pflanze mit einem L-System ein schwieriger Prozess." [11] Es setzt viel Erfahrung voraus, eine vorgegebene Pflanze über ein L-System darzustellen. Bereits geringe Änderungen lassen manchmal eine komplett andere Gesamtgestalt entstehen.

Bei der Modellierung herrscht das Prinzip von Versuch und Irrtum vor. Es ist somit eine

```

\var l1 table 8 9 11 14 17 16 14, /* leaf description: */
\var l2 table 40 44 44 40 31 20 9, /* lengths inside leaf blade */
\var l3 table 10 11 12 15 17 15 13,
\var l4 table 24 35 38 35 27 17 7,
\var i index,
\angle 5, /* zigzag growth */
\var x1 normal 40 5, % /* branching angle */
\const bl 35, /* length conversion factor internode/leaf */
\const minlf 15, /* minimal leaf length */
\const cf 0.2, /* coefficient in leaf length estimation */
\const secgr 1.6, /* strength of secondary growth */
\var len length,
\axiom veg 1 2, /* vegetative production: bud, leaf */
\axiom shoot 1-8, /* half-annual steps of shoot development */

veg # bud,
bud # leaf,
leaf # ,

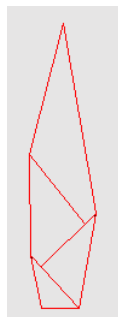
bud ## P12 [RU-90 F1 S(0)] RU90 F1 S(1) RU-132 F3.8 S(2) f-0.8 RU88
F4 S(3) f-0.8 RU-84 F4.7 S(4) RU52 F7.2 S(5) C(0,2) C(2,4)
C(5,3) C(3,1),

leaf ## P2 D1 F12 D0 [ S(0) [ RU58 f(15) S(1) ] [ RU53 f(30) S(2) ]
&(7) < F(l1(i)) [ RU(55-4*i) F(l2(i)) S(i+3) ] > F11 S(10) ]
[ RU-45 f(15) S(18) ]
&(7) < f(l3(i)) [ RU(4*i-55) F(l4(i)) S(17-i) ] >
&(18) < C(i, i+1) > C(18, 0),

shoot # L(bl) D+1.2 m(1, 9), /* growth starts with 9 internodes */
(j<2) m(j, n) # m(j+1, n),
(j=2 && n>1) m(j, n) # L*0.68
&(n-1) < - a(0) RH180 - [ RU(x1) ang m(1, if(i<(n-1)/2,i/2,i-1)) ] L*1.1 >
- L*0.5 a(0) RH180 - L*1.7 m(1, n), /* main rule for shoot growth */
(j=2 && n<=1) m(j, n) # L*0.4 a(0) L*2 m(1, 1),
ang # RU10,
a(t) # a(t+0.5),
a(t) ## D1+(secgr * t) F, /* internode */
m(j, n) ## L1(len/bl + 1/(cf*len+1/minlf)) D11 0(veg, j),

```

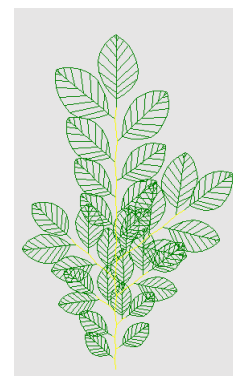
Abbildung 2.6: L-System Rotbuche nach Kurth (wgrogra v. 2.4: bu4.lsy)



(a) erste Ableitung



(b) zweite Ableitung



(c) sechste Ableitung

Abbildung 2.7: Grafische Interpretation Rotbuche

schnelle Visualisierung von Modellen notwendig, um die Auswirkung von Änderungen der Regeln und Parameterwerte zu evaluieren. Daraus folgt ebenfalls, dass eine Generierung fast nur von Hand möglich ist, da es sehr auf die Evaluierung durch den Modellierer ankommt.

Aufgrund der linearen Datenstruktur können Netzwerke (wie etwa Genregelungsnetzwerke, Interaktionsnetzwerke mit der Umwelt oder zelluläre Automaten) nicht abgebildet werden. Dies resultiert daraus, dass Ersetzungsregeln darauf angelegt sind, wachsende Strukturen zu erzeugen, wohingegen bei Netzwerken die Strukturen oftmals festliegen. Bei ihnen ändern sich lediglich die Parameter der Objekte (z. B. Konzentrationen bei physiologischen Prozessen wie Transportvorgängen, Stoffwechsel, Zustandsänderungen eines zellulären Automaten).

Da das Wachstum ausschließlich durch Ersetzung erfolgt, ist so der Informationsfluss zwischen den Pflanzenteilen nur ungenügend simulierbar. Prusinkiewicz (2004) räumt ein, dass die Einbeziehung genetischer Mechanismen in bisherige "virtuelle Pflanzen" (auf L-System-Basis) zur Zeit ein offenes Problem darstellt.

Dass eine Generierung mit L-Systemen langwieriger ist, liegt an der Tatsache, dass gleichzeitig die Entwicklung der Pflanze betrachtet wird, die in keinem Fall trivial ist. Der Regelkanon bei solch komplexen Pflanzen wird schnell umfangreich und unübersichtlich. Vor allem hat der Grafikmechanismus der Turtle gravierende Nachteile, er ist bei komplexen Geometrien sehr umständlich. Texturierte Oberflächen wie bei polygonalen Modellen sind hier z. B. nicht möglich, ebenso gibt es keine Möglichkeit der einfachen Vervielfältigung von Objekten.

2.4 Prozedurales Modellieren

Als ein Art "Gegenbewegung" zu den regelbasierten Systemen stellten Honda sowie Fischer ihr prozedurales Modell vor [24–26]. Mit ihm war es zum ersten Mal möglich, dreidimensionale Baumskelette zu erzeugen. Die Bilder erhält man durch Projektion der 3D-Daten auf eine Betrachtungsebene, s. Abbildung 2.8.

Wie in der Natur zu beobachten, nimmt die Verzweigung in diskreten Schritten mit jeder Verzweigungsordnung zu. Dabei geschieht die Verzweigung binär. Die Länge der verzweigten Segmente (Kindsegmente) ist über Verhältnswerte auf die Länge des Vatersegments bezogen. Der Verzweigungswinkel zwischen den beiden Kindsegmenten ist konstant, sie liegen immer in der selben Ebene wie das Vatersegment. Da die zu dieser Zeit vorhandene Hardware nur wenige Möglichkeiten bei der grafischen Wiedergabe unterstützt, werden die

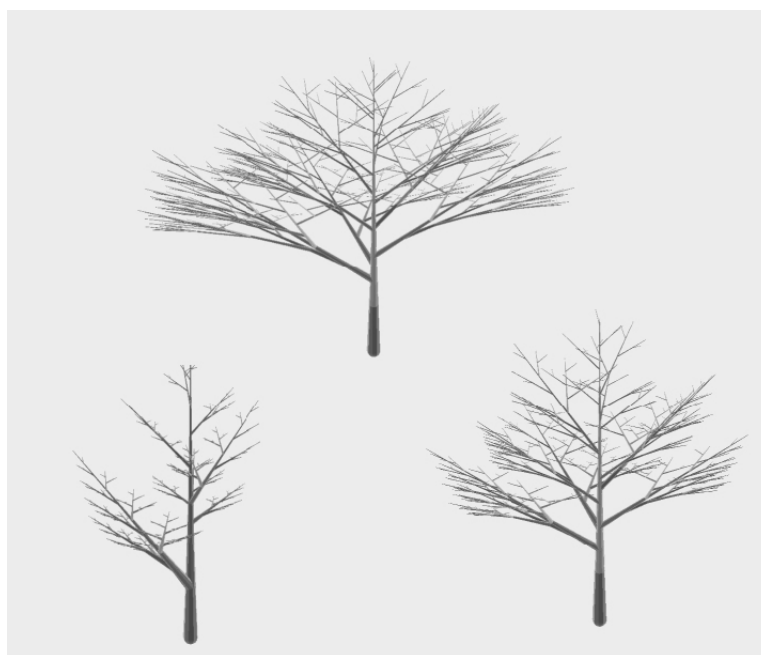


Abbildung 2.8: Verzweigungsstruktur nach Honda (Bild: Honda)

Internodien¹ aus einfachen Liniensegmenten gezeichnet, ihre Dicke wird nicht beachtet. Der Aufbau der Modelle ist relativ einfach. Es entstehen realistisch wirkende Baumskelette, die durch ihre Dreidimensionalität beliebig am Bildschirm gedreht werden können.

In den 80er Jahren wurde dieses Verfahren von Aono und Kunii [1] erweitert. Das Hauptaugenmerk liegt auf der Modellierung von Verzweigungen, die durch Bifurkation zu realistischeren Ergebnissen führen soll. Der Begriff Bifurkation stammt ursprünglich aus der Geomorphologie und bezeichnet die Gabelung eines Wasserlaufs und Verteilung des Wassers auf zwei getrennte Flussgebiete. Die kritischen Punkte an denen eine Verzweigung stattfindet werden über Bifurkations-Diagramme bestimmt. Unter konstant bleibendem Verzweigungswinkel nehmen die Länge und der Durchmesser der sich verzweigenden Äste mit konstantem Faktor ab. Neu ist, dass die Kindsegmente und das Vatersegment nicht mehr in der selben Ebene liegen. Die beiden verzweigenden Äste liegen in der Ebene, die durch den Vater und seinen maximalen Gradienten aufgespannt wird. Mit der Einführung von Attraktoren und Inhibitoren wird versucht, den Einfluss von Wind nachzubilden. Weitere Verzweigungsarten werden eingeführt und die Verzweigungswinkel in Abhängigkeit von ihrem Entstehungsalter verändert. Das Ergebnis dieser Erweiterungen ist ein breites Spektrum beeindruckend natürlich wirkender Modelle.

In diesem Zusammenhang kritisieren Aono und Kunii die zum gleichen Zeitpunkt populär werdenden L-Systeme. Sie seien zu unhandlich und nicht in der Lage, ihre eigenen Modelle

¹[lateinisch] Botanik: durch Knoten (Nodien) voneinander gesonderte Sprossabschnitte (z.B. bei Gräsern und Bäumen)

nachzubilden. Dies wurde von Prusinkiewicz und Lindenmayer insbesondere durch die Verwendung parametrischer L-Systeme in [37] widerlegt, die ähnliche Verzweigungsformen erzeugen.

2.5 Partikelsysteme

Die von Reeves und Blau [49] eingeführten Partikelsysteme waren zunächst für den Einsatz im Film gedacht. Reeves, angestellt bei Lucasfilm, benötigte eine schnelle, relativ realistisch wirkende Methode, um Wälder und Wiesen zu simulieren. Dabei kam es weniger auf botanische Korrektheit an, mehr sollte ein realistisches Gesamtbild erzeugt werden.

Das Verfahren benutzt kleine Partikel als Objekte, die zum großen Teil stochastisch bestimmt werden. Über einen rekursiven Algorithmus werden Äste erzeugt und an den Stamm angehängt, was die grobe Form des Baumes vorgibt. Die daran angefügten Blätter sind kleine Kugeln mit Farbe und Ausrichtung.



Abbildung 2.9: Strukturierte Partikelsysteme nach Reeves und Blau (Bilder: W. Reeves)

Dieses primitive Verzweigungsverfahren erzeugt sehr regelmäßige Strukturen, was einen nachträglichen Einbau von zufallsgesteuerten Unregelmäßigkeiten nötig werden lässt. Dennoch man mit Hilfe der Farbgebung sehr realistische Ergebnisse erzielen.

Ein Nachteil ist, dass das Rendering bei großer Partikelzahl ein nichttriviales Problem darstellt.

2.6 Ein fraktales Baummodell

Mit der Entdeckung der Fraktale und inspiriert durch die Arbeiten von Mandelbrot [40,41], befassen sich Mitte der achtziger Jahre eine Reihe von Computergrafikern mit rekursiven

Erzeugungsprozeduren für natürliche Objekte. Mittels rekursiver Funktionen werden Verzweigungen erzeugt, die eine gewisse Selbstähnlichkeit besitzen. Es lassen sich so sehr detaillierte Modellierungen, erstmals mit gebogenen Teilstücken, konstruieren. Der Schwerpunkt des Verfahrens von Oppenheimer [43] liegt auf einer guten visuellen Darstellung und auf einer schnellen Erzeugung.

Der rekursive Ansatz bringt es mit sich, dass es zu einer recht starken Selbstähnlichkeit kommen kann. Um diesen Effekt etwas zu relativieren, werden Zufallsparameter eingebaut. Die weiteren von Oppenheimer [43] verwendeten Parameter sind:

- Verzweigungswinkel
- Verhältnis der Größe von Vater- und Kindzweigen
- Grad der Verjüngung entlang Stamm und Ästen
- Anzahl der Zweige pro Stammsegment
- Deviationswinkel

Die Rekursion kann beliebig fortgesetzt werden, um immer feinere Verästelungen entstehen zu lassen, wird aber in der Praxis beschränkt.

Der Stamm und die Äste werden als generalisierte Zylinder über die Verbindung der einzelnen Segmente modelliert. Zur Erzeugung der realistisch aussehenden Rinde wird eine horizontal verlaufende Sägezahnfunktion, der ein Brownsches Rauschen hinzugefügt wird, verwendet.

$$Rinde(x, y) = Sägezahn(N * (x + R * noise(x, y)))$$

noise(x, y) : periodisch in x- und y-Richtung verlaufendes Rauschen

In Abbildung 2.10 ist ein Ergebnis der rekursiven Prozedur zu sehen. Oppenheimer behauptet, Bäume dieser Komplexität zur damaligen Zeit bereits in Echtzeit dargestellt zu haben.

2.7 Meristem-orientierte Pflanzenmodellierung

De Reffye u. a. m. stellen einen botanisch fundierten Ansatz von Wachstumsregeln vor [50]. Ausgehend vom Meristem, teilungsaktives Gewebe in den Knospen, wird in diskreten Zeitschritten Wachstum simuliert. Von Knoten zu Knoten werden so entlang eines Sprosses Internodium an Internodium gesetzt. Innerhalb der Äste ist eine Ordnung festgelegt. Diese Ast-Ordnung bestimmt hierbei die Wachstumsgeschwindigkeit. Äste mit höherer Ordnung

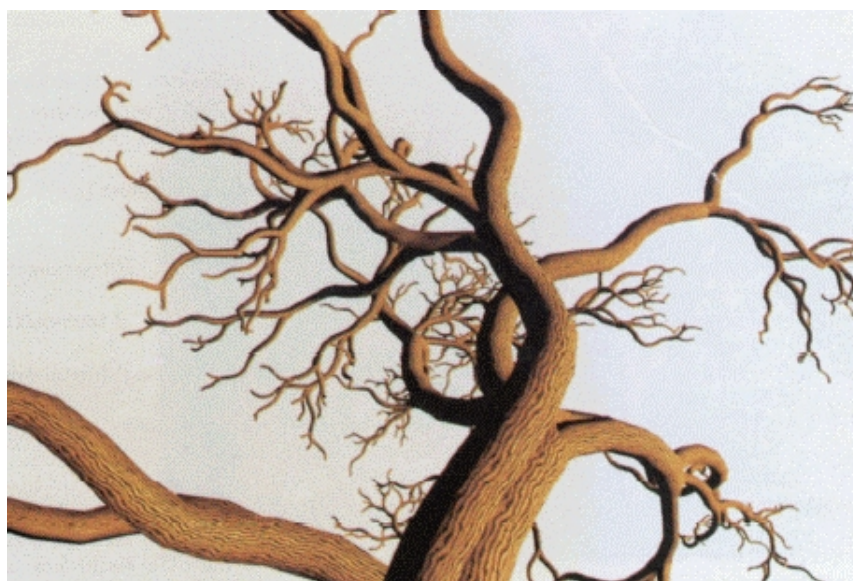
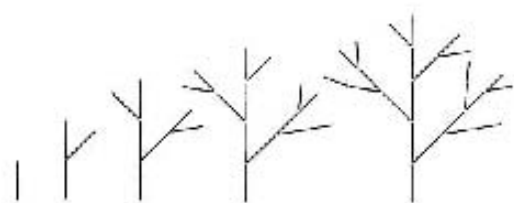


Abbildung 2.10: Fraktales Baummodell (Bild: Oppenheimer)

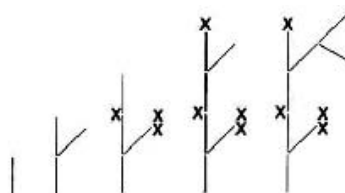
können mit größerer Geschwindigkeit wachsen als solche mit niedriger Ordnung. Gesteuert über mehrere Wahrscheinlichkeiten, die jede Knospe in sich trägt, kann ein Spross sich verzweigen, ruhen oder absterben.

Es handelt sich hierbei um ein zufallsgesteuertes Modell, wobei der Zufall viel stärkere Auswirkungen hat, da weitreichende strukturelle Eigenschaften verändert werden können.

Abbildung 2.11 zeigt die Simulation eines Wachstumsvorganges über Knospung für zwei unterschiedliche Parametersätze. Die Wahrscheinlichkeit für das Ruhen einer Knospe ist jeweils gleich null.



(a) Wahrscheinlichkeit des Absterbens: 0



(b) Wahrscheinlichkeit des Absterbens: $\neq 0$

Abbildung 2.11: Simulation von Knospung (Bild: De Reffye)

Um beispielsweise einen Baum zu modellieren, müssen demnach folgende Parameter vorgegeben sein: Alter des Baumes, Wachstumsgeschwindigkeiten der Äste verschiedener Ordnungen, Anzahl möglicher Knospen pro Knoten in Abhängigkeit von der Ordnung, sowie die Wahrscheinlichkeiten für Absterben, Aussetzen, Verzweigen. Der Pseudocode für die diskrete Simulation lässt sich wie folgt formulieren:

```

FOR (t = 1 : tmax) { // jedes Zeitsignal
  FOR (jede noch lebende Knospe) {
    IF (!(Knospe.stirbt) AND !(Knospe.wartet)) {
      Generiere_Internodium();
      Generiere_Blatt();
    }
    FOR (in Frage kommende Knospen) {
      IF (Knospe.verzweigt) {
        Bilde_Verzweigung();
      }
    }
  }
}

```

Die kommerzielle Software (siehe www.bionatics.com) nutzt das Verfahren und stellt dem Benutzer eine umfangreiche Bibliothek von Pflanzenalgorithmen zur Verfügung. Das Verfahren ermöglicht anatomisch korrektes Wachstum, obwohl die Autoren dies in [50] nicht durchführen. Animationen werden ermöglicht, wobei der Algorithmus in der vorliegenden Form auf diskrete Zeitschritte festgelegt ist und daher durch geeignete Interpolationsmethoden ergänzt werden muss, um gleichmäßige Animationen zu erreichen.

2.8 Bäume aus Strängen (Strang-Systeme)

Bereits Leonardo da Vinci vermutete, dass für Bäume der Querschnitt eines Astes gleich der Summe der Querschnitte seiner Kinder ist.

$$\text{Querschnitt}_{\text{Vater}} = \sum_{i=1}^{\#Kinder} \text{Querschnitt}_{\text{Kind}_i}$$

Diese Eigenschaft, zusammen mit dem inneren Aufbau von Bäumen, brachte da Vinci darauf, Baumstrukturen aus Strängen aufzubauen. Ein anfängliches Bündel von Strängen teilt sich, von der Wurzel ausgehend, in jeder Astgabelung bis hin zu Einzelsträngen in den Blättern auf. Der jeweilige Astquerschnitt ergibt sich aus der aktuellen Anzahl der Stränge, die ihn bilden.

Erst später erwies sich dieser Schätzwert als sehr präzise, was Matthew Holton [23] für seine Modelle nutzte. Bei diesem Verfahren, s. Abbildung 2.12, bestimmt die Anzahl der Stränge nicht nur die Stärke der Äste, sondern auch deren Länge und die Anzahl der Blätter sowie den Verzweigungswinkel.



Abbildung 2.12: Baummodell aus Strängen (Bild: M. Holton)

2.9 Iterierte Funktionssysteme (IFS)

Iterierte Funktionssysteme stellen ein einfaches Verfahren, mit dem eine gute Bildqualität erzeugt werden kann, dar. Der IFS-Code ist sehr kompakt und leicht editierbar, aber aufgrund der Gestalt (lange Listen von Matrizen) sehr unanschaulich. Bereits kleine Änderungen im Code wirken sich überall im Bild aus, so dass eine Vorhersage des Ergebnisses nicht möglich ist.

Das Verfahren bezieht keine physikalischen oder organischen Strukturprinzipien in die Modellierung ein. Das Verfahren arbeitet nicht mit Geometriedaten, sondern mit Punktmenge in der Ebene. Das Resultat des Verfahrens ist wiederum eine Punktmenge, es gibt entsprechend keine Oberfläche, wie bei den anderen Verfahren.

Beispiel: Sierpinski-Dreieck, Deterministischer Algorithmus:



Abbildung 2.13: Erzeugung des Sierpinski-Dreiecks (Tiefe 0, 1, 2, 3, 6, 9)

In Abbildung 2.13 sind die Ergebnisse der Anwendung der Abbildungen aus Tabelle 2.5 in den ersten Iterationstiefen zu sehen.

Tabelle 2.5: Definition der affinen Abbildungen

w_i	a	b	c	d	e	f	p
1	0.5	0	0	0.5	1	1	0.33
2	0.5	0	0	0.5	1	50	0.33
3	0.5	0	0	0.5	50	50	0.34

IFS-Formalismus

Ein iteriertes Funktionssystem (auf \mathbb{R}^2) ist:

- endliche Menge von kontrahierenden Abbildungen
 n Abbildungen $M_j \in \mathbb{R}^2 \rightarrow \mathbb{R}^2$, $M = \{M_1, M_2, \dots, M_n\}$
- n Wahrscheinlichkeiten $P = \{p_1, p_2, \dots, p_n\}$ $\sum_{j=1}^n p_j = 1$
- Startpunkt z_0 bzw. Menge von Startpunkten Z_0

Affine Abbildung M_j :

$$M_j(z_i) = w_i * z_i + p_i = M_j(x, y) = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

Grundlegende Funktionsweise:

Auf jeden Punkt der Startmenge Z_0 wird entsprechend der Wahrscheinlichkeiten eine Abbildung angewendet, wodurch der der Punkt verschoben wird. Das entscheidende bei der Hintereinanderschaltung mehrerer Abbildungen M_j sind die erstaunlich komplexen Muster der nach häufiger Iteration erhaltenen Punktmenge, die so genannte Attraktormenge (kurz: der Attraktor). Enthält die Startmenge nur einen einzigen Punkt, bildet die Spur, die der Punkt beim verschieben hinterlässt, das erzeugte Bild.

Drei unterschiedliche Arbeitsweisen:

- Deterministischer Algorithmus: Breitenabstieg
- Stochastischer Algorithmus: Tiefenabstieg, zufällig für Startpunkte
- Adaptiver Algorithmus: Breitensuche mit adaptivem Abschneiden der Teilbäume je nach Gesamtkontraktion

Barnsley und Demko [4, 10] stellten diese Methode 1984 vor. Barnsley [3] stellt eine Reihe Anwendungen der IFS vor, so den folgenden, nach ihm benannten Barnsley-Farn. In Abbildung 2.14 ist das hohe Maß an Selbstähnlichkeit, das der Figur zu Grunde liegt, sehr deutlich zu sehen. Es ist sogar möglich, die fraktale Dimension von IFS-Attraktormengen zu berechnen [32].

Von Prusinkiewicz und Lindenmayer wurde in [37] ein Verfahren vorgestellt, mit dem es möglich ist, L-Systeme in RIFS umzuwandeln. RIFS (recurrent IFS) sind eine Erweiterung der IFS, die Barnsley in [3] vorstellte. Bei den RIFS darf nicht mehr jede Transformation auf jede beliebige folgen, indes werden Paare von aufeinander folgenden Transformationen spezifiziert.

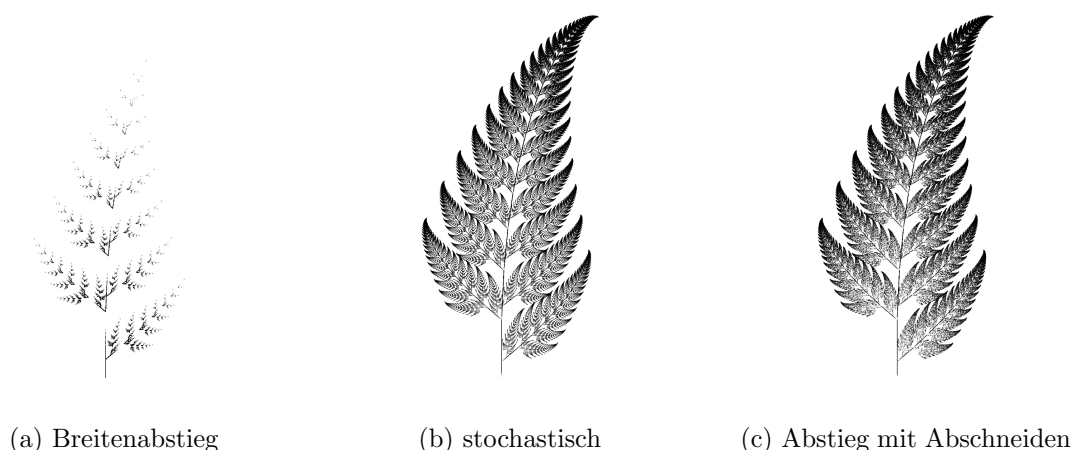


Abbildung 2.14: Generierung eines Farnblattes (Bild: M. Barnsley)

2.10 Objektinstanzierung

Objektinstanzierung ist ein weiterer Ansatz der 3D-Objekterzeugung. Er basiert auf der Eigenschaft, dass sich viele Verzweigungsstrukturen als baumartige Datenstrukturen repräsentieren lassen. So z. B. die fraktalen Baummodelle oder die L-Systeme, denen ebenfalls ein Baum, diesmal in linearisierter Form als Text, zugrunde liegt. Auch iterierte Funktionssysteme lassen sich als Bäume repräsentieren.

Nun können diese Datenstrukturen, je nach Komplexität des modellierten Objektes, sehr groß werden, so dass es sinnvoll ist, geeignete Kompressionstechniken einzuführen. Eine bestehende Möglichkeit ist es, einzelne Primitive prototypisch zu erzeugen und sie später, mittels relativen Positionsangaben, zu platzieren. Dies erlaubt es, Teilbäume die mehrfach vorkommen, durch einen einzelnen zu ersetzen. Es entsteht so aus dem Baum ein gerichteter azyklischer Graph. Dieses von Sutherland in seinem System "Sketchpad" [52] eingeführte Verfahren wird Objektinstanzierung genannt.

Des Weiteren ist es nicht nur möglich, Instanzierungen ausschließlich innerhalb einer Verzweigungsstruktur vorzunehmen, man kann mit dieser Methode ganze Objekte vervielfältigen. Anstatt in einer Wiese jede Pflanze neu zu definieren, wird eine kleine Menge von Prototypen erzeugt und an viele Stellen kopiert. Diese starke Speicherplatzersparnis macht große Szenen überhaupt erst handhabbar.

Prusinkiewicz und Lindenmayer zeigen in [37], dass sowohl eine Teilmenge der L-Systeme als auch iterierte Funktionssysteme effizient durch azyklische Graphen repräsentiert werden können.

Verwendung findet das Verfahren heute z. B. in VRML, wo mit dem ähnlichen Szenengraphen-Ansatz eine Instanzierung von geometrischen Primitiven vorgenommen wird.

2.11 CSG-basierte Modellierung

Eine gängige Methode der Objektbeschreibung ist die des CSG-Graphen (Constructive Solid Geometry). Dabei werden die Objekte durch regularisierte Mengenoperationen aus vorgegebenen Grundobjekten (Kugel, Quader, Zylinder, Kegel, usw.) gebildet. Die Verwendung regularisierter Operationen (op^*) sichert, dass das Ergebnis immer ein Volumen ist.

$$A \text{ op}^* B = \text{abgeschlossene H\u00fclle}(\text{Inneres}(A \text{ op} B))$$

Da die verwendeten Mengenoperationen (\cap , \cup , \setminus , usw.) einwertig oder zweiwertig sind, kann der Ausdruck in Form eines Bin\u00e4rbaumes (CSG-Tree) dargestellt werden, bei dem die Knoten Operationen und die Bl\u00e4tter Primitive sind. Zur Veranschaulichung s. Abbildung 2.15. In 2.15a sind die Primitive dargestellt, aus denen der K\u00f6rper K in 2.15b gebildet wird.

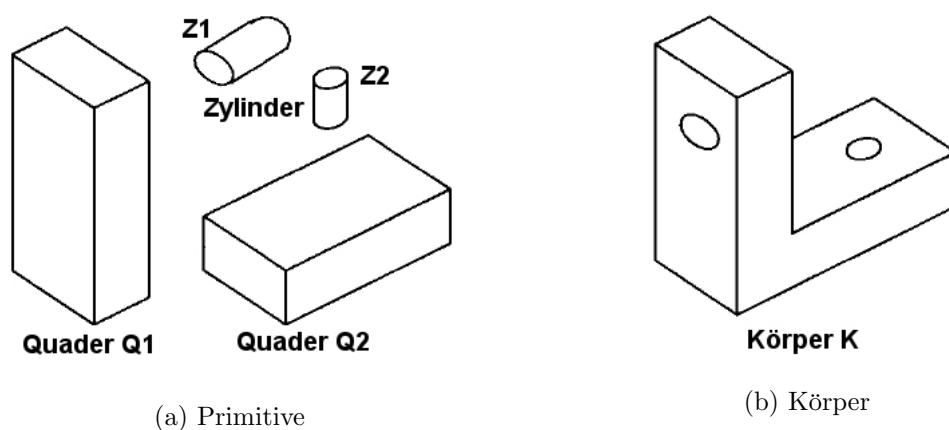


Abbildung 2.15: K\u00f6rperkonstruktion mittels CSG-Graphen

Ein sich aus dem Konstruktionsprinzip ergebendes Problem ist, dass die Konstruktion eines Objektes nicht eindeutig ist. Dies zeigt Abbildung 2.16, in der verschiedene Konstruktionsvarianten des K\u00f6rpers 2.15b zu sehen sind.

Ein weiteres Problem stellt sich bei der Berechnung der Oberfl\u00e4che des endg\u00fcltigen Objektes, da in einem CSG-Baum die Geometrie von Kanten und Fl\u00e4chen nicht im Baum abgespeichert ist. Die Berechnung muss auf der Ebene der Primitive vorgenommen werden, da nur sie die Geometriewerte enthalten. Der Baum wird dazu rekursiv bis zu den Primitiven durchlaufen, wo dann die Berechnung f\u00fcr das jeweilige Primitiv durchgef\u00fchrt wird. Das Teilergebnis wird sukzessive zusammengesetzt, bis das Ergebnis f\u00fcr das Gesamtobjekt vorliegt. Genau diese aufw\u00e4ndige Operation begrenzt die Einsatzgebiete der CSG-B\u00e4ume sehr.

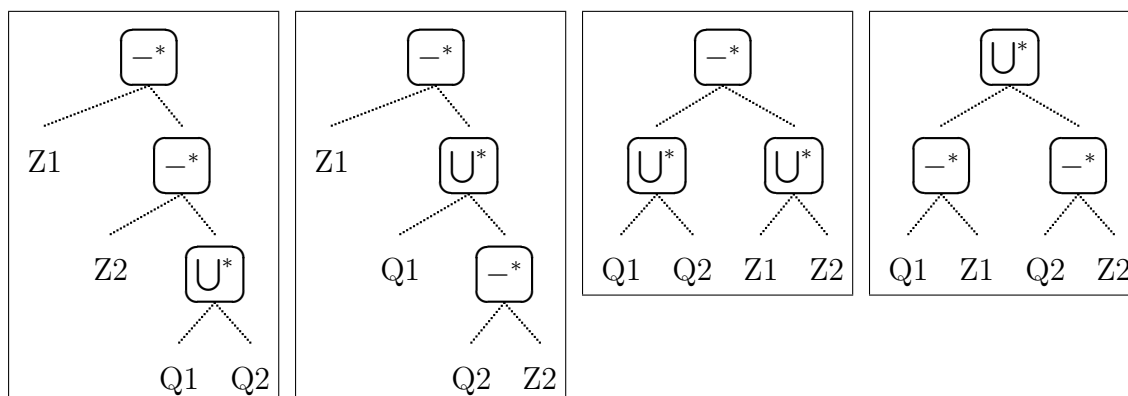


Abbildung 2.16: Verschiedene Konstruktionsvarianten eines Körpers

Eine Erweiterung des CSG-Verfahrens wurde von Gervautz und Traxler [20] vorgestellt. Indem sie zyklische CSG-Graphen zulassen, können beliebig feine Objektbeschreibungen erzeugt werden.

2.12 Einordnung der Verfahren

Es gibt eine Reihe von Einteilungen, in die sich die im vorangegangenen Kapitel vorgestellten Verfahren einteilen lassen. Eine Einteilung ist z. B. die in prozedurale und regelbasierte Modellierung. So gehören die L-Systeme, die Iterierten Funktionssysteme und die CSG-basierte Modellierung zu der regelbasierten Modellierung, alle anderen zu der prozeduralen Modellierung. Beide Verfahren haben ihre Vor- und Nachteile.

L-Systeme sind, wie bereits beschrieben, äußerst kompakt, ihnen liegt ein einfacher Formalismus zugrunde, und die genannten Erweiterungen machen sie zu einem sehr mächtigen Werkzeug. Bereits in [51] wurde gezeigt, dass L-Systeme prinzipiell in der Lage sind, alle 23 Baumarchitekturen von Hallé, Oldemann und Tomlinson [21] zu erzeugen. Der Modellieraspekt ist aber nach wie vor problematisch. Der Modellierer muss über einen reichen Erfahrungsschatz verfügen, um eine Pflanze von Grund auf über ein L-System darzustellen. Schon kleinste Änderungen bewirken manchmal eine komplette Änderung der Gesamtgestalt. Jede Parameteränderung kann erst durch die komplette Modellerzeugung mitsamt Geometrierzeugung überprüft werden, was den Prozess zusätzlich verlangsamt.

Prozedurale Methoden hingegen lassen sich einfach und intuitiv handhaben. Mit einem Algorithmus ist aber meist nur eine sehr eingeschränkte Menge von Pflanzen modellierbar.

Die Frage, die sich nun stellt, ist die, wie sich die beiden Verfahren so kombinieren lassen, dass sich die jeweiligen Vorteile ergänzen. Ziel ist es, die Mächtigkeit der regelbasierten Verfahren mit der Intuitivität des prozeduralen Ansatzes zu kombinieren. Des Weiteren

soll es möglich sein, fast alle Arten von Pflanzen mittels einer konsistenten Beschreibung zu erzeugen. Was den Modellierungsprozess anbelangt, ist ein interaktives System mit direkter Rückkopplung erstrebenswert. Ein solches Verfahren wird im nachfolgenden Kapitel vorgestellt.

3 Regelbasierte Objekterzeugung

Die regelbasierte Objekterzeugung, von Oliver Deussen und Bernd Lintermann entwickelt [11], basiert auf einer Verbindung von regelbasierter mit prozeduraler Modellierung. Eine Umsetzung des Verfahrens wurde von ihnen in der Software Xfrog¹ [12, 14, 38, 39] realisiert. Das Verfahren ist botanisch nur schwach fundiert, da es u. a. über keinerlei Möglichkeiten zur Wachstumssimulation verfügt, dennoch liefert es grafisch sehr zufrieden stellende Ergebnisse.

Das Regelsystem wird durch einen gerichteten Graphen², den von den Autoren so genannten p-Graph, beschrieben, wobei die Knoten des p-Graphen durch verschiedene Komponenten gebildet werden; seine Kanten beschreiben Ersetzungsregeln bzw. Ersetzungshängigkeiten. Die prozedurale Beschreibung steckt in den Komponenten, die mittels eines umfangreichen Katalogs von Parametern individuell editiert werden können. Ein Objekt wird somit durch die Graphstruktur und einen Parametersatz, für jede Komponente, beschrieben.

Ebenso ist die Generierung der Endgeometrie in zwei Schritte unterteilt. Ausgehend vom p-Graph wird in einem ersten Schritt der sog. i-Baum erzeugt, ein temporärer Baum, der nötig ist, da strukturelle Informationen durch den p-Graphen und durch die Komponenten repräsentiert werden. Der i-Baum wird aus Instanzen der Komponenten im p-Graphen zusammengesetzt. Im Unterschied zu den Komponenten (den Baustein-Prototypen) werden in den Instanzen verschiedene Parameter, wie deren lokale Koordinatensysteme, von den Vätern weitergegeben. Enthält der p-Graph eine Multiplikationskomponente, so werden im i-Baum mehrere Instanzen der nachfolgenden Objekte erzeugt, in Abhängigkeit von den Parametern des Multiplikators. Im anschließenden Schritt wird der i-Baum traversiert und ausgehend von der Wurzel die Geometrie erzeugt. Das nachfolgende Beispiel 3.1 soll dies verdeutlichen.

A, B, C und D sind nicht näher spezifizierte Baustein-Prototypen. A ist die Wurzel des p-Graphen in Abbildung 3.1a und B eine geometrieerzeugende Komponente, die einen Stiel erzeugt. C ist ein Multiplikator-Baustein, der selbst keine Geometrie erzeugt, jedoch seine Nachfolger dreimal produziert. Komponentenprototyp D erzeugt, mit einer Rekursionstiefe von drei, einen kleinen Zweig. Der entsprechend dieser Vorgaben erzeugte i-Baum

¹siehe dazu www.greenworks.de greenworks organic-software

²In der Softwareumsetzung Xfrog sind Rekursionen generell nicht zugelassen.

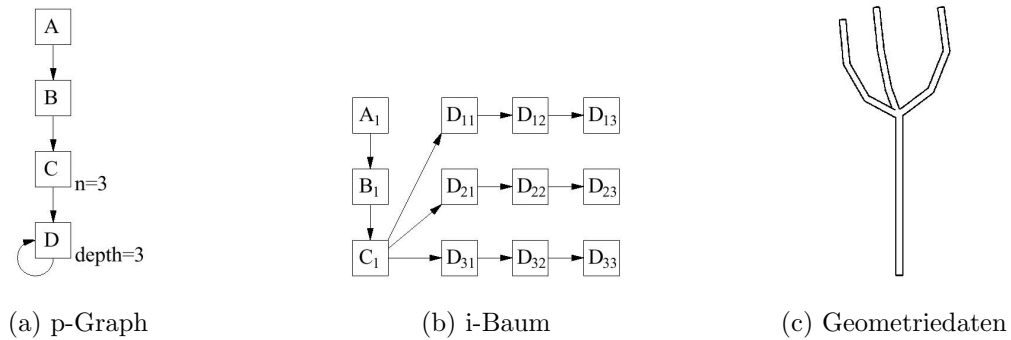


Abbildung 3.1: Geometrie-Erzeugung (Bild: O. Deussen)

wird in Abbildung 3.1b dargestellt. Die Instanzen jeder Komponente X werden mit X_i beschrieben. Dabei ist zu beachten, dass der Baustein-Prototyp D neunmal instanziiert wird: dreimal rekursiv, und das Ganze wird durch den Multiplikator C verdreifacht, wobei jeder dieser Teilbäume im i -Baum einen Ast der Geometrie aus Abbildung 3.1c erzeugt.

3.1 Komponententypen

Deussen teilt in [11] die Komponenten in drei Gruppen ein. Die Gruppe der *Geometrieerzeugenden Komponenten* dient zur Erzeugung der grafischen Objekte. Mit den sog. *multiplizierenden Komponenten* ist es möglich, die direkt nachfolgende Struktur zu vervielfältigen und sie entsprechend anzuordnen. Die letzte Gruppe ist die der *Komponenten zur globalen Modellierung*, mit denen beispielsweise die Welt zur Modellierung von Tropicamen³ beschrieben wird.

Abbildung 3.2 zeigt die bei Deussen [11] bzw. in Xfrog zur Verfügung stehenden Komponenten.

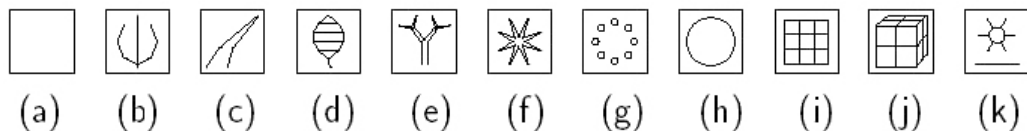


Abbildung 3.2: Übersicht über die Xfrog-Komponententypen

Geometrieerzeugung	a) Simple, b) Revo, c) Horn, d) Leaf
Multiplizierende Komponenten	e) Tree, f) Hydra, g) Wreath, h) Phiball
Globale Modellierung	i) FFD, j) Hyperpatch, k) World

³[griechisch] der, Krümmungsbewegung festsitzender Organismen zu einer Reizquelle hin oder von ihr weg

Genau genommen müsste man noch eine weitere Gruppe hinzufügen, die der sowohl geometrieezeugenden als auch multiplizierenden Komponenten, zu der die Horn- und die Treekomponente zählen würden.

3.1.1 Geometrieezeugende Komponenten

Die einfachsten Typen von Komponenten sind die geometrieezeugenden Komponenten. Sie erzeugen die grafischen Objekte, die später in Polygone aufgeteilt und gerendert werden.

Simple Mit dieser Komponente werden grundlegende geometrische Primitive, wie Kugel, Zylinder oder Quader, erzeugt. Es ist möglich, sie mit einer Farbe oder Textur zu versehen, sowie sie beliebig in Lage und Größe zu transformieren.

Horn Die Hornkomponente wird verwendet, um alle Arten von Stielen, Ästen oder Stämmen zu erzeugen, kann aber durchaus zur Herstellung anderer Objekte genutzt werden [53]. Die erzeugte Geometrie wird verschiedentlich als generalisierter Zylinder bezeichnet: Ein erweitertes Zylindermodell, gebildet aus einer Fläche, z. B. einem Viereck, welcher im Raum entlang einer Bahn verschoben wird. Das durch die Verschiebung der Fläche eingeschlossene Volumen bildet den generalisierten Zylinder. Ferner ist es möglich, mit der Hornkomponente nachfolgende Objekte zu vervielfältigen. Die so erzeugten Objekte werden entlang der Hauptachse der Hornkomponente nach jedem Segment angefügt.

Blatt Für alle Arten von Blättern und Blütenblättern ist die Blattkomponente gedacht. Die Oberfläche ergibt sich aus einer Folge von Flächenprimitiven, die anschließend trianguliert werden. Die Umrandung wird durch eine polygonale Kurve bestimmt. Ein breites Spektrum an Blättern kann so erzeugt werden, s. Abbildung 3.3. In der Praxis ist es aus Gründen der Performance besser, die Form möglichst einfach zu halten und sie mit einer Textur in Form einer Fotografie eines realen Blattes zu versehen.

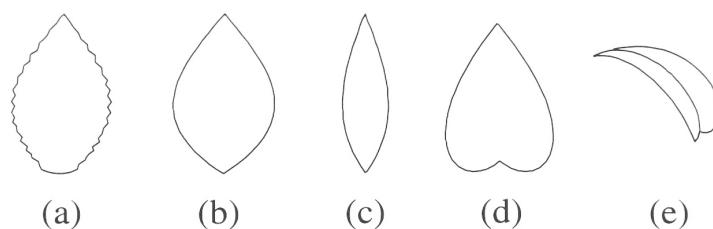


Abbildung 3.3: Definition verschiedener Blattgeometrien (Bild: O. Deussen)

3.1.2 Multiplizierende Komponenten

Um von einem Objekt mehrere Instanzen zu erzeugen und sie entsprechend anzuordnen, werden multiplizierende Komponenten verwendet. In Xfrog erzeugen die Komponenten bis auf die Treekomponente keine eigene Geometrie.

Tree Durch die Treekomponente wird, ähnlich wie bei der Hornkomponente, eine kegelähnliche Geometrie erzeugt, an deren Hauptachse die nachfolgenden Objekte (Äste) angeordnet werden. Ausgehend von der Natur gibt es mehrere Möglichkeiten, die Äste zu beeinflussen. Siehe dazu Abbildung 3.4: (a) Standardform, (b) vertikaler Winkel, (c) Verzweigungsdichte, (d) horizontaler Winkel zwischen Ästen, (e) Größe (f), Dicke entlang der Sprossachse, (g) Knotzigkeit

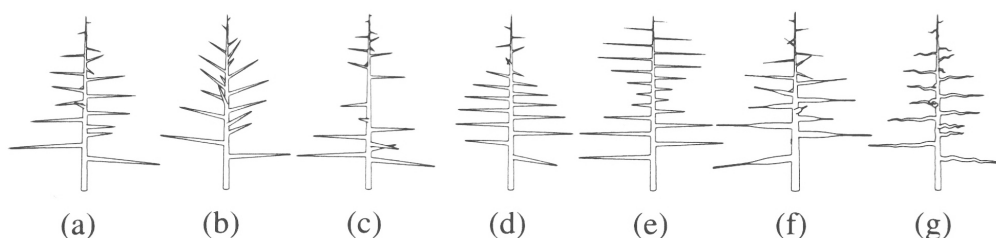


Abbildung 3.4: Verschiedene Parametervariationen einer Baumkomponente (Bild: O. Deussen)

Hydra Beim Vervielfältigen einer Komponente mittels einer Hydra werden n erzeugte Instanzen sternförmig rund um die Hauptachse angeordnet, s. Abbildung 3.5a. Die Öffnungswinkel sind frei wählbar.

Wreath Die Wreathkomponente ordnet, ähnlich wie die Hydrakomponente, die erzeugten Instanzen rund um die Hauptachse an. Im Unterschied zur Hydra ist die Ausrichtung der neuen Objekte parallel zur Hauptachse, s. Abbildung 3.5b. Über einen weiteren Parameter ist es möglich, den Abstand von den Hauptachse zu bestimmen.

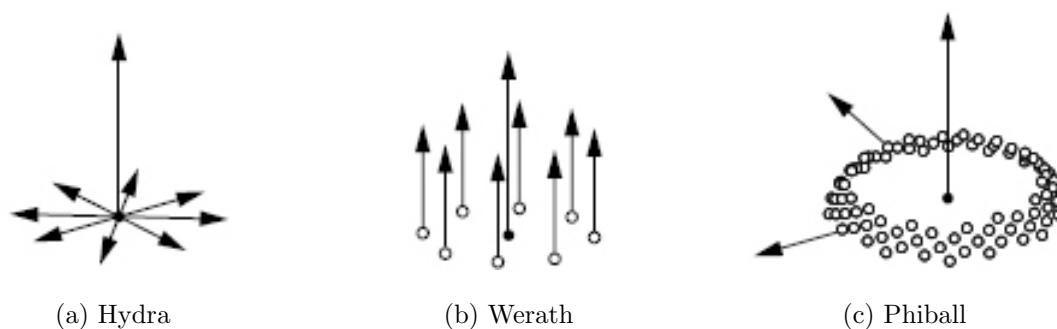


Abbildung 3.5: Orientierung der multiplizierenden Komponenten (Bild: O. Deussen)

Phiball Mittels der Phiballkomponente werden Objekte nach der Regel des goldenen Schnittes auf einer Kugeloberfläche angeordnet, s. Abbildung 3.5c. Dieses in der Natur weit verbreitete Phänomen findet man z. B. bei der Anordnung von Samen, Blättern und Fruchständen. Abbildung 3.6 zeigt zwei Anwendungen in der Pflanzenmodellierung.



(a) Zapfen eines Nadelbaumes (Bild: [37])



(b) Kakteen (Bild: [19])

Abbildung 3.6: Phyllotaxis-Anwendungen

Eine einfache Berechnungsvorschrift für die ebene Anordnung wurde bereits 1979 von Vogel [56] vorgestellt. Dabei wird die Position des i -ten Objektes in Polarkoordinaten durch den Winkel α_i und den Radius r_i beschrieben:

$$r_i = c * \sqrt{i} \quad \text{und} \quad \alpha_i = i * \phi,$$

wobei c eine positive Konstante ist und für ϕ gilt:

$$\phi = \frac{360^\circ}{\tau^2} = 137,5077\dots^\circ \quad \text{mit} \quad \tau = \frac{\sqrt{5} + 1}{2}$$

Das Teilungsverhältnis des goldenen Schnittes ist durch die Konstante τ gegeben, die hier im Quadrat steht, da bei dichter Packung die Anzahl der Platzierungen auf einer Kreisscheibe proportional zu deren Oberfläche und damit zu τ^2 ist. Um eine gleichmäßige Positionierung zu gewährleisten, muss bei fortschreitender Platzierung zwangsläufig ein Wachstum mit der Quadratwurzel des Radius erfolgen.

Abbildung 3.7a zeigt die sich aus diesem Formalismus ergebende Anordnungsreihenfolge, in Abbildung 3.7b sind drei Variationen des Divergenzwinkels ϕ zu sehen, bei denen gut zu erkennen ist, dass bereits kleine Änderungen zu deutlicher Streifenbildung führen.

Die analytische Lösung nach Deussen [11], für die Verteilung von Objekten auf einer Kugel mit dem Goldenem-Schnitt-Algorithmus (s. Abbildung 3.8), geht zunächst

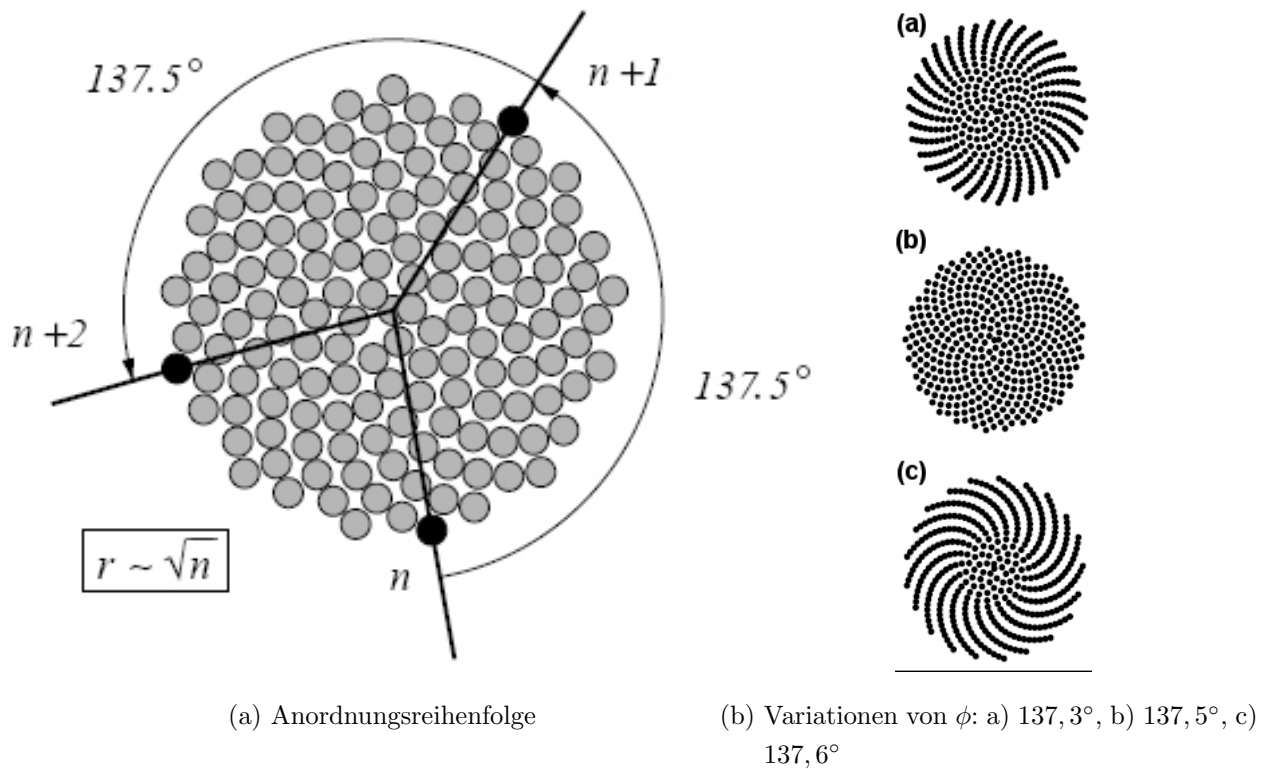


Abbildung 3.7: Anordnung nach dem Goldenen Schnitt nach Vogle

von der Annahme aus, dass alle Objekte gleich groß sind. Für die Kugel wird eine Rotationsachse z definiert, die identisch mit der Zentralachse der Vorgängerkomponente ist.

Es werden nun N Objekte um diese z -Achse mit Winkel $\Phi_i = i * d\Phi$ für das i -te Objekt und $d\Phi = 2\pi/((1 + \sqrt{5})/2)$ als Goldener Schnitt platziert. Der Winkel $d\Phi \approx 137,5^\circ$ ist hierbei der Fibonacci-Winkel.

Die Rotation θ_i der Objekte erfolgt um eine zur z -Achse lotrecht stehenden Achse. Die Berechnung von θ_i erfolgt in mehreren Schritten:

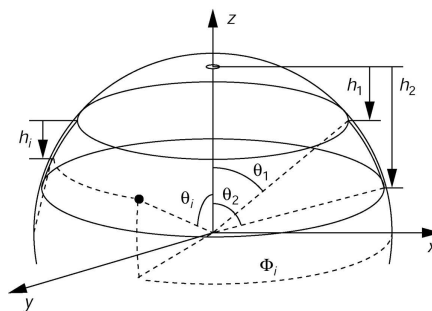


Abbildung 3.8: Verteilung nach dem Goldenen Schnitt auf einem Kugelausschnitt

Zunächst wird die Höhe der Fläche auf der Kugel berechnet, die Platz für N Objekte mit je einer Grundfläche von A_E bietet. Auf einer Kugel mit Radius R ist die Fläche $S(\theta_1, \theta_2)$, $0 \leq \theta_1 < \theta_2 \leq \pi$ gleich $A = 2\pi Rh$, mit $h = R(\cos(\theta_1) - \cos(\theta_2))$.

Um N Objekte mit einer Gesamtfläche von $A_N = NA_E$ zu platzieren, wird eine Kugel mit Radius

$$R = \sqrt{\frac{A_N}{2\pi(\cos(\theta_1) - \cos(\theta_2))}}$$

benötigt. Um θ_i innerhalb der i -ten Iteration zu berechnen, wird die Fläche, die von i Objekten $A_i = iA_E$ abgedeckt wird, als Funktion der Höhe des entsprechenden Kugelbereiches dargestellt.

Es wird angenommen, dass die Fläche komplett mit den Objekten gefüllt werden kann.

$$A_i = 2\pi Rh_i \quad \text{mit} \quad h_i = \frac{A_i}{2\pi R} = \frac{A_i h}{A_N}$$

Nun gilt:

$$\begin{aligned} \theta_i &= \arcsin\left(\frac{R - (h_1 + h_i)}{R}\right) \\ &= \arcsin\left(\frac{R - R(1 - \cos(\theta_1)) - \frac{A_i h}{A_N}}{R}\right) \\ &= \arcsin\left(\cos(\theta_1) - \frac{A_i}{A_N}(\cos(\theta_1) - \cos(\theta_2))\right) \end{aligned}$$

Wenn die einzelnen Objekte unterschiedlicher Größe sind, sei

$$A_N = \sum_{j=1}^N a_j \quad \text{und} \quad A_i = \sum_{j=1}^i a_j$$

wobei a_j die Fläche des Objekts j ist. Für Objekte mit konstanter Größe gilt:

$$\theta_i = \arcsin\left(\cos(\theta_1) - \frac{i}{N}(\cos(\theta_1) - \cos(\theta_2))\right)$$

Mit θ_i und Φ_i können nun die Objekte auf den Kugeln referenziert werden.

3.1.3 Globale Modellierung

In diese Gruppe gehören bei Deussen [11] die FFD-, Hyperpatch- und die Weltkomponente. Mit ihnen ist es möglich, wie der Name bereits sagt, den globalen Kontext der Objekte zu beeinflussen.

FFD- und Hyperpatchkomponente Mit beiden Komponenten wird eine Freiform-Deformation (FFD) definiert, die die globale Gestalt einer Pflanze beeinflusst. Bei

der FFD werden vom Benutzer Funktionen $D(x)$, $D(y)$ und $D(z)$ definiert, wohingegen die Hyperpatchkomponente mit dreidimensionalen Bézierfunktionen vom Grad eins bis drei arbeitet. Sie werden beispielsweise eingesetzt, um den Einfluss von Wind zu simulieren.

Weltkomponente Mittels der Weltkomponente ist es möglich, die Ausrichtung der Blatt- und Treekomponente zu beeinflussen. Die beiden hier simulierten Effekte sind der Foto- und der Gravotropismus, die eine Anziehung der Komponente hin zum Licht bzw. zum Boden bewirken.

3.2 Kombination von Komponenten

Die vom Benutzer durch den p-Graphen vorgegebene Struktur einer Pflanze ist das Resultat einer Kombination von Komponenten. Hierbei werden bei Deussen [11] drei Arten von Verbindungen (Kantentypen) zugelassen:

Child-Link Die Standardverbindung, sie platziert die Geometrie der Komponente relativ zur Geometrie der Vaterkomponente.

Branch/Rib-Link Dieser Kantentyp wird zwischen den Verzweigungen einer Treekomponente oder den Rippen einer Hornkomponente benutzt; werden andere Komponenten so verbunden, so wird sie als Child-Link interpretiert.

Leaf-Link Ist die Vaterkomponente Teil einer Rekursion, wird die Geometrie der Kindkomponente nur nach Abbruch der Rekursion erzeugt.

Es ist möglich, jeden Komponententyp mit jedem über beliebige Kanten zu verbinden. In der Softwareumsetzung Xfrog spielen aber nur die Child- und Branch/Rip-Link-Kanten eine Rolle, da Rekursionen generell nicht zugelassen sind.

3.3 Beispiel

Am Beispiel der Modellierung einer Sonnenblume soll demonstriert werden, wie Pflanzen mit der Xfrog-Software erzeugt werden, siehe [13]. Eine gute Herangehensweise ist die Konstruktion einer Pflanze in mehreren Schritten. Das in einem Teilschritt erzeugte und überprüfte "Pflanzenmodul" wird anschließend entsprechend seiner Natur hierarchisch mit anderen "Modulen" verbunden.

Für die Blütenblätter und Blätter werden Fotos realer Pflanzen benötigt. Mit einem transparenten Hintergrund ist es später möglich, sie einfach als Textur der Objekte zu verwenden und somit einen möglichst realistischen Eindruck zu erzeugen.

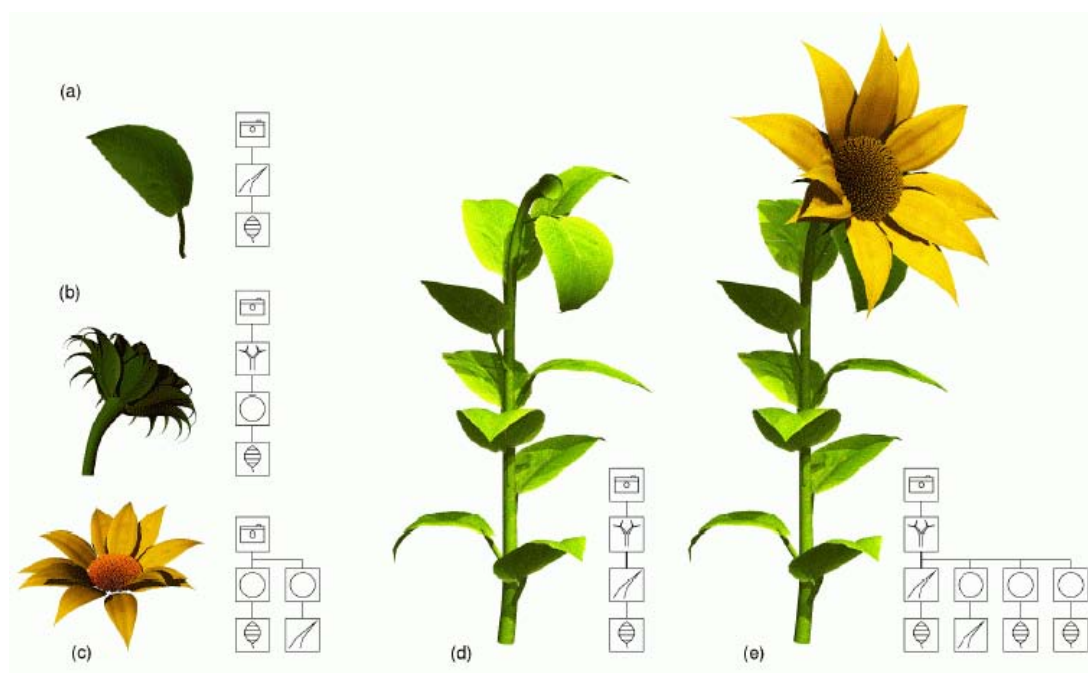


Abbildung 3.9: Beispiel: Sonnenblume (Bild: Deussen)

Das Ergebnis des ersten Schrittes in der Konstruktion einer Sonnenblume, das Blatt, ist in Abbildung 3.9 zu sehen. Ausgehend von der Wurzel ist an einer Hornkomponente eine Blattkomponente angefügt. Beide Komponenten werden über Parameter so lange beeinflusst, bis das Resultat möglichst natürlich wirkt.

Die Sonnenblume wird aus einem Stiel, im Beispiel eine Treekomponente, an die die aus 3.9(a) stammenden Blätter angehängt werden (3.9(d)), und einem Kopfteil gebildet. Der Kopf wird aus mehreren multiplizierenden Komponenten und Blattkomponenten gebildet, s. Abbildung 3.9(b) und (c).

Im letzten Schritt werden, wie in Abbildung 3.9(e) zu sehen, die Einzelteile zusammengesetzt, und nach weiteren Parameteranpassungen ist die Sonnenblume modelliert.

Für weitere Beispiele zur globalen Modellierung oder zur Animation sei an dieser Stelle auf die Publikationen von Deussen und Lintermann [11, 38, 39] verwiesen. Ein kompletter Pflanzenkatalog der kommerziellen Pflanzenmodelle, der mittlerweile über 1000 Pflanzen verschiedenster Arten und Gattungen enthält, kann auf dem Internetportal www.greenworks.de bezogen werden.

4 Integration der regelbasierten Objekterzeugung in GroIMP

Das in dem vorangegangenen Kapitel vorgestellte Konzept der regelbasierten Objekterzeugung wurde in dieser Diplomarbeit mittels Instanzierungsregeln umgesetzt und in die bereits bestehende Modellierungs- und Simulationssoftware GroIMP integriert.

Dieses Kapitel beschreibt die in GroIMP integrierte Programmiersprache XL, sowie das dahinter stehende Konzept der Relationalen Wachstumsgrammatiken (RGG). Im zweiten Abschnitt wird die Verwendung der Instanzierungsregeln erläutert und die Einbindung in XL beschrieben.

4.1 GroIMP und XL

Mit der Software GroIMP (Growth-grammar related Interactive Modelling Platform) wurde am Lehrstuhl Praktische Informatik / Grafische Systeme der BTU Cottbus eine auf die Bedürfnisse der Pflanzenmodellierung zugeschnittene Modellierungsplattform geschaffen. Mit XL (extended L-system language) [30] ist eine verständliche und somit relativ leicht zu erlernende, regelbasierte "Metasprache" integriert, die es erlaubt, eine Modellspezifikation schnell und einfach zu ändern und leicht zu validieren, was wiederum ihre Wartbarkeit und Wiederverwendbarkeit enorm erhöht. Die so erreichte Transparenz ermöglicht eine gute Vergleichbarkeit und Kombinierbarkeit der Modelle.

XL [30] ist eine auf "Relationalen Wachstumsgrammatiken" (RGG) basierende Programmiersprache, die als Erweiterung von Java implementiert wurde und so eine Verbindung zum prozeduralen und zum objektorientierten Programmierparadigma herstellt. Die relationalen Wachstumsgrammatiken gehören zu den Graph-Grammatiken [29], sie basieren auf dem etablierten regelorientierten Modellierungs-Formalismus der L-Systeme (s. Kapitel 2.3), erweitern aber die zugrunde liegenden Datenstrukturen von Zeichenketten auf Graphen und die Zeichenkettenersetzung auf Graph-Ersetzung. Somit ist es möglich, dreidimensionale Strukturen von Pflanzen auf direktere Weise zu generieren als mit L-Systemen.

Wie in Abbildung 4.1 zu sehen, entfällt der Interpretationsschritt durch die Turtle bei den Graph-Grammatiken, da sie direkt auf 3D-Strukturen arbeiten. Diese 3D-Strukturen bilden die Knoten im Graph, die mittels gerichteter Kanten verbunden werden, wobei unterschiedliche Kantentypen verwendet werden können (z. B. a und b in Abbildung 4.2).

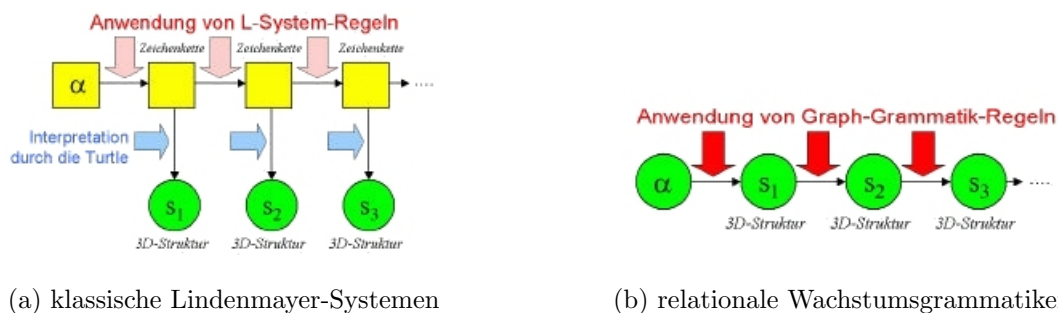


Abbildung 4.1: Unterschiedliche Arbeitsweisen in der Pflanzenstrukturmodellierung (nach Kurth)

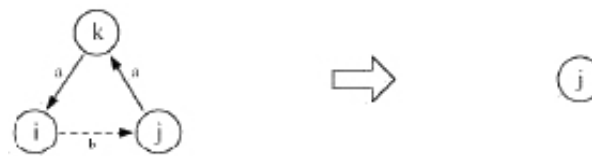
Mathematisch handelt es sich um verschiedene Relationen auf der Knotenmenge – daher der Name ”relationale Wachstumsgrammatiken”. Eine Ersetzungsregel in textueller Form, wie diese:

$$i - b - > j - a - > k - a - > i \quad ==> \quad j\{P\};$$

würde in grafischer Form der Abbildung 4.2a entsprechen, wobei P eine optionale Folge von XL-Anweisungen ist. Abbildung 4.2b zeigt eine Anwendung der Regel auf einen Beispielgraphen. Der linke Teil einer Regel beschreibt den Teil eines Graphen, der durch den rechten Regelteil zu ersetzen ist. Eine Menge solcher RGG-Regeln wird relationale Wachstumsgrammatik genannt. Die klassischen L-Systeme sind als Spezialfall enthalten. Des Weiteren ist es möglich, durch die Angabe von Kontextbedingungen die Anwendung der Regeln zu steuern, sowie pro Regel ein Stück prozeduralen Code (P) zu definieren, der bei jeder Regelanwendung ausgeführt wird.

Als Knoten können beliebige Java-Klassen verwendet werden, was die Anbindung von in Java implementierten (Teil-)Modellen an XL-Modelle vereinfacht. XL enthält selbst wesentliche Konstrukte von Java, die für die Modellierung bereit stehen.

Die vollständige XL-Sprachspezifikation von Kniemeyer [30] sowie eine Reihe von Beispielen stehen unter der Webadresse www.grogra.de bereit. Ebenfalls an dieser Stelle wird die Modellierungs- und Simulationssoftware GroIMP beschrieben und bereitgestellt.



(a) RGG-Regel



(b) Anwendung der RGG-Regel auf einen Graphen

Abbildung 4.2: Beispiel Graphersetzung (nach Kniemeyer)

4.2 Instanzierungsregeln

Basierend auf den Prinzip der Interpretationsregeln, die von Kurth im Zusammenhang mit der L-System-Software GROGRA eingeführt wurden, mit denen ganze Pflanzenorgane durch Polygonzüge definiert werden können (s. Kapitel 2.3.4), ist es mit Hilfe der Instanzierungsregeln möglich, vollständige Objekte zu erzeugen, ohne die Struktur erst über einen zusätzlichen Ableitungsprozess zu generieren.

Deussen [11] spricht in diesem Zusammenhang von Objektinstanzierung (Kapitel 2.10) und erweiterte dieses Konzept zu der im Kapitel 3 beschriebenen regelbasierten Objekterzeugung.

Der Begriff Instanzierungsregeln stammt von Kniemeyer, er basiert auf dem aus der objektorientierten Programmierung bekannten Begriff Instanzierung, was bereits darauf hindeutet, dass hier mit Instanzen gearbeitet wird.

Instanzierungsregeln sind den Interpretationsregeln ähnlich, aber nicht deckungsgleich. Interpretationsregeln können beispielsweise Auswirkungen auf Symbole rechts von ihrer Anwendungsstelle haben, wohingegen Instanzierungsregeln reine "Zeichenanweisungen" sind. Mit Instanzierungsregeln kann man wiederum rekursive Strukturen definieren.

Mit ihnen besteht eine komfortable Möglichkeit, komplexe Objekte zu modellieren, die dann beispielsweise in bestehende Simulationen eingebunden werden können.

Instanzierungsregeln werden innerhalb von Java-Klassen deklariert. Ihr Aufbau richtet sich dabei nach folgender Syntax (s. [30]):

```
InstancingMethodDeclaration ::= Modifiers void Identifier '('
    FormalParameterList ')' [ Throws ] '==>' { Instantiation } ';' ;
```

Man kann sie somit an dem Pfeil '==>' identifizieren, ihr Returntyp ist void.

Die folgenden Codezeilen, eingebunden in eine RGG-Datei, demonstrieren die Verwendung einer Instanzierungsregel in XL. Das Beispiel erzeugt ein kleinen Baum 4.3, mit einer Rekursionstiefe von 4.

```

module Baum(int depth, float length) ==>
    F(length)
    if (depth > 0) (
        [RÜ( 30) RH(90) Baum(depth - 1, length*0.8)]
        [RU(-30) RH(90) Baum(depth - 1, length*0.8)]
    );

protected void init() [
    Axiom ==> Baum(4, 1);
]

```

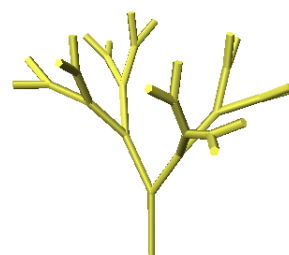


Abbildung 4.3: Instanzierter Baum

Das Konzept der Instanzierungsregeln wurde mit den in Kapitel 5 vorgestellten Bausteinen umgesetzt, implementiert und steht fortan als Spracherweiterung von XL bereit. Aus Performancegründen wurden die Bausteine nicht direkt als Instanzierungsregeln implementiert, sondern als eigenständige Java-Klassen. Eine Umsetzung als reine Instanzierungsregeln wäre durchaus möglich, wie das folgende Beispiel einer Wreath-Instanzierung zeigt.

Eine Wreath-Komponente vervielfältigt alle mit ihr verbundenen Strukturen und ordnet sie in einem Ring um den Mittelpunkt an. Der Radius des Rings sowie die Anzahl der erzeugten Instanzen kann dabei vom Benutzer frei definiert werden. Zur Erhöhung der Übersichtlichkeit wurden in dieser Implementierung nur vier Attribute des Wreath-Baustein verwendet.

number Anzahl der erzeugten Instanzen
radiusX X-Radius
radiusY Y-Radius
scale uniformer Skalierungsfaktor

Diese Parameter werden in Zeile 2 bis 5 eingebunden. Die anschließende Schleife enthält zwei Teile: einen Java-Teil (Zeile 12-19) mit einigen Berechnungen und den Graph-Grammatik-Teil (Zeile 19-23) der die Geometrie erzeugt. Über eine XL-Graphabfrage in Zeile 22 wird das erste mit der Wreath-Instanz verbundene Objekt abgefragt. Damit ein Objekt vervielfältigt wird, muss es mittels einer "multiply"-Kante mit der Wreath-Instanz verbunden sein.

```

1 module Wreath(
2     int number,
3     float radiusX,
4     float radiusY,
5     float scale
6 ) extends Point ==>
7 {
8     float delta = (float)(2 * PI / number);
9 }
10 for (int i : (0:number))
11 (
12     {
13         float w = i * delta;
14         float x = radiusX * (float) Math.cos(w);
15         float y = radiusY * (float) Math.sin(w);
16         float r =(float)Math.sqrt(x*x+y*y);
17         float phi = (float) Math.toDegrees(Math.atan2(y, x));
18     }
19 [
20     RH(phi) RU(90) M(r) RU(-90)
21     RH(-phi) Scale(scale)
22     getFirst(multiply)
23 ]
24 );

```

Wird die Instanzierungsregel wie folgt aufgerufen, erzeugt sie eine elliptische Verteilung von 15 Kegelobjekten, wie in Abbildung 4.4 zu sehen.

```
Axiom ==> Wreath(15, 6, 4, 1) -multiply-> Cone;
```

Die elliptische Form kommt von den unterschiedlichen Radien für die X- und Y-Achse, hier sechs für die X- und vier für die Y-Achse. Der Skalierungsfaktor ist in diesem Beispiel für alle Instanzen eins.

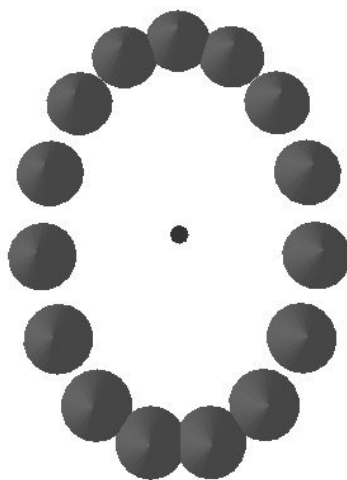


Abbildung 4.4: Grafische Resultat der Anwendung der Wreath-Instanzierung auf ein Cone-Objekt

5 Bausteinkatalog

In diesem Kapitel werden die von mir implementierten Bausteine genau vorgestellt und beschrieben. Es werden all ihre Attribute erläutert, sowie die dahinter stehenden mathematischen Funktionen erklärt. Der komplette Bausteinkatalog steht mit allen benötigten Komponenten im Paket 3D-Construction Set (3D-CS) der GroIMP-Software zur Verfügung.

Im Rahmen dieser Diplomarbeit wurden die aus Xfrog bekannten Horn-, Tree-, Hydra-, Wreath- und Phiball-Komponenten um die Arrange-, die Variation- sowie die BlockScale- und BlockColor-Komponente erweitert. Bei den bekannten Komponenten wurden die Namen, die Definitionsbereiche sowie die Initialwerte der Attribute übernommen.

5.1 Attributtypen

Die meisten Bausteine lassen sich mittels einer Vielzahl an Attributen parametrisieren, wobei jedes Attribut von einem speziellen Typ ist.

Der entsprechende Attributtyp wird in den nachfolgenden Beschreibungen bei jedem Baustein für jedes Attribut angegeben. Alle Attributbeschreibungen werden im Nachfolgenden immer dieser Form entsprechen:

Attributname *Attributtyp (Wertefeld)*

Das Wertefeld kann, wie in den Typbeschreibungen angegeben, variieren.

Attributtypen:

Integer *Wertefeld = (Initialwert, Minimalwert, Maximalwert)*

In einem Attributfeld dieses Typs können ganze Zahlen eingegeben werden. Werte, die die Intervallgrenzen nicht einhalten, werden für die Berechnung auf den jeweiligen Grenzwert gesetzt.

Float *Wertefeld = (Initialwert, Minimalwert, Maximalwert)*

Analog dem *Integer*-Feld, aber diesmal mit Gleitkommazahlen.

CheckBox *Wertefeld = (Initialwert)*

Erlaubt ein Zu- bzw. Abschalten einer Eigenschaft.

Textfeld Wertefeld = (Initialwert)

In einem Textfeld können beliebige textuelle Eingaben gemacht werden.

Funktionsfeld Wertefeld = (Initialwert, Minimalwert, Maximalwert)

Ein Funktionsfeld, s. Abbildung 5.1a, ist ein reellzahliges Eingabefeld, mit dem es möglich ist, umfangreiche mathematische Funktionen zu definieren. Es unterteilt sich in den Namen des Feldes, gefolgt von dem Identifikator "function" und dem eigentlichen Eingabefeld der Funktion. Der Parameter p in der zweiten Zeile ist eine reelle Zahl im Intervall -1 bis 5, der als Variable in den Funktionen eingesetzt werden kann.

Als Funktionen können z. B. die Winkel-, Wurzel- oder Logarithmenfunktionen eingesetzt werden, natürlich auch nur einfache Zahlenwerte, wie in diesen Beispielen:

$$\sin(x); 3 * \text{sqrt}(x) + \text{pi}; (\text{rnd}(x) * 10)\%2; (x + x * \text{random}) * p; \text{ usw.}$$

In Anhang C ist die dem Funktionsparser zugrunde liegende Grammatik, eine vollständige Funktionsbeschreibung aller zur Verfügung stehenden Funktionen sowie die Beschreibung der bereitstehenden Variablen und Konstanten zu finden.

Ergeben sich bei der Funktionsauswertung unzulässige Werte außerhalb der Intervallgrenzen, so werden diese auf den überschrittenen Grenzwert gesetzt.

(a) Funktionsfeld

(b) Komplexfeld

Abbildung 5.1: Funktionsfeld und Komplexfeld

Komplexfeld Wertefeld = (Initialwert, Minimalwert, Maximalwert, Funktion),

wenn beide Funktionsfelder gleich initialisiert werden, sonst ist das Wertefeld folgendermaßen aufgebaut:

$$\text{Wertefeld} = (\text{Wertefeld Funktion1}, \text{Wertefeld Funktion2}, \text{Funktion})$$

Ein Komplexfeld setzt sich aus zwei Funktionsfeldern und einer Funktion (FloatToFloat-Feld) zusammen, s. Abbildung 5.1b. Eine Liste aller vordefinierten Funktionen ist in Anhang B angegeben.

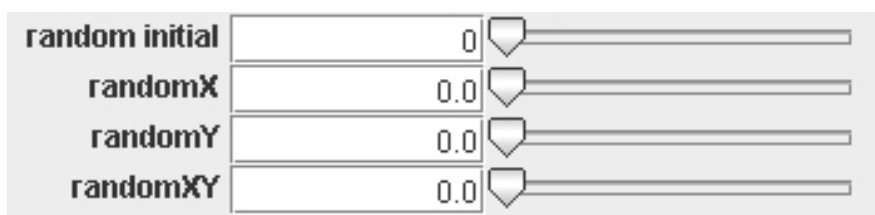


Abbildung 5.3: Random-Feld des Arrange-Bausteins

Die in diesem Feld zusammengefassten Attribute sind:

initial *Integer* (0, 0, 50)

Der mittels dieses Attributs eingegebene Wert wird dem verwendeten Zufalls-generator als Seed-Wert bei der Instanzierung übergeben. Auf diese Art ist es möglich, "verschiedene" Zufallsgeneratoren zu erzeugen, die aber jedesmal die selben Zufallszahlen generieren, um die selben Anordnungen zu erzeugen.

randomX *Float* (0, 0, 20)

Dieser Wert legt den maximalen Betrag fest, um den eine Instanz entlang der X-Achse verschoben werden kann.

$$x = x + \text{random}(-\text{randomX}, \text{randomX})$$

Da bei bereits bei kleinen Zufallsmanipulationen der Punkte das Gesamtbilds stark verändert wird, ist ein Maximalwert von 20 ausreichend.

randomY *Float* (0, 0, 20)

Analog *randomX*, aber auf die Y-Achse bezogen.

randomXY *Float* (0, 0, 20)

Mit *randomXY* wird der maximale Betrag festgelegt, um den eine Instanz sowohl der X- als auch der Y-Achse entlang verschoben werden kann.

LODFeld *Wertefeld = (Initialwert)*

Das **Level of Detail** Feld ist ein Sammelfeld, in dem die jeweiligen Attribute zur Detailstufensteuerung zusammengefasst sind. Das Kapitel 6 erläutert allgemeine Verfahren, die umgesetzten Methoden werden im Unterkapitel 6.2 beschrieben.

LocationParameterBase *Wertefeld = (Initialwert)*

LocationParameterBase ist eine Zusammenfassung mehrerer Attribute, die die Manipulation der drei freien locationParameter-Werte *n1*, *n2* und *n3* erlauben, s. Abbildung 5.4.

ChannelMap *Wertefeld = (Initialwert)*

Eine ChannelMap ist ein umfangreiches Attribut, mit dem Bilder geladen und erzeugt werden können.

Color3f Wertefeld = (*Initialwert: Rot, Grün, Blau; Minimalwert: Rot, Grün, Blau; Maximalwert: Rot, Grün, Blau*)

Mit einem Attribut diesen Typs wird eine Farbe definiert.

5.2 Baustein-Attribute

Der Bausteinkatalog umfasst momentan acht Komponenten. Die Attribute, die die Bausteine gemeinsam haben, werden hier zusammengefasst erläutert, um sie nicht bei jedem Baustein erneut beschreiben zu müssen.

Name *Textfeld (leer)*

Erlaubt die Angabe eines Namens für einen Baustein.

Layer *Integer (0, 0, 15)*

Der Parameter legt die Schicht fest, in der die Instanz gezeichnet wird. Mit diesem Mechanismus ist es z. B. möglich, alle Blätter eines Baummodells in die selbe Schicht einzuordnen und dann die Schicht auszublenden, so dass nur noch das Baumskelett zu sehen ist.

locationParameter *LocationParameterBase ()*

Mit den Attributen in einem LocationParameter-Feld ist es möglich, drei freie Variablen zu steuern, die z. B. für den internen Informationsfluss verwendet werden können. Die LocationParameter-Werte stehen in jedem Baustein für jede erzeugte Instanz zur Verfügung und werden vom Vater auf alle vervielfältigten Instanzen weitergegeben. Abbildung 5.4 zeigt den Aufbau von einem LocationParameter-Feld. Im aktLocationParameter-Feld werden die aktuellen vom Vater an das Objekt übergebenen Werte dargestellt, wobei immer nur die Werte der ersten vom Vater erzeugten Instanz ausgegeben werden, unabhängig davon, wie viele Instanzen vom Vaterbaustein erzeugt werden. Es handelt sich hierbei um ein reines Ausgabefeld, das jegliche Änderungen seitens des Benutzers zwar zulässt, aber ignoriert.

Mittels der drei anschließenden Attribute setLocationParameter(s1, s2, s3) können die LocationParameter-Werte für alle nachfolgenden Objekte, die mit diesem Baustein verbunden sind, gesetzt werden.

Die deltaLocationParameter-Felder ermöglichen es, für jeden einzelnen LocationParameter-Wert einen prozentualen Wert zu definieren, über den die Änderung zur nächsten Generation festgelegt wird. Die Änderung kann minus bis plus 100 Prozent betragen, die neuen Werte ergeben sich wie folgt:

$$\begin{aligned} \text{newLocationParameterValues} &= \text{aktLocationParameterValues} + \\ &\quad \text{aktLocationParameterValues} * \text{deltaLocationParameter} \end{aligned}$$

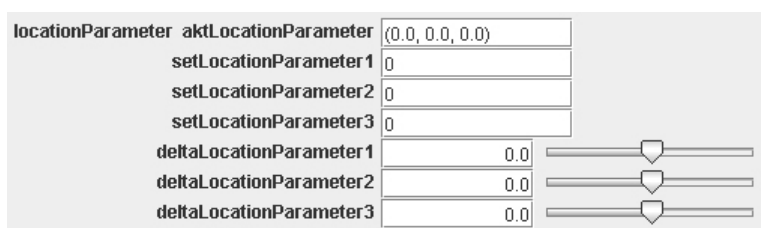


Abbildung 5.4: LocationParameter-Feld-Aufbau

Der BlockScale-, BlockColor- und Arrange-Baustein haben dieses Attribut nicht.

initAll *CheckBox* ()

Durch Betätigen der initAll-CheckBox werden alle Attribute des Bausteins wieder initialisiert, alle bis dahin vorgenommenen Änderungen gehen verloren. Der BlockScale- und BlockColor-Baustein haben dieses Attribut nicht.

5.3 Kombination von Komponenten

Zur Konstruktion von Strukturen, die aus mehr als einem Baustein bestehen, ist es nötig, diese miteinander zu verbinden. Der resultierende p-Graph (s. Kapitel 3) wird im Graph-Fenster von GroIMP dargestellt. Dem Benutzer stehen in der BlockConst-Klasse drei verschiedene Kantentypen zur Verfügung.

multiply Die multiply-Kante ist die häufigste Verbindung zwischen den Bausteinen. Objekte, die so mit einem Baustein verbunden sind, werden entsprechend dem Baustein vervielfältigt.

child Dieser Kantentyp wird nur von Horn- und Tree-Bausteinen benutzt. Objekte, die mittels einer child-Kante mit einem dieser Baustein verbunden sind, werden in Wuchsrichtung, sozusagen an die Spitze, des Baustein gezeichnet, s. Abbildung 5.5b. Alle anderen Bausteine, von denen eine solche Kante ausgeht, behandeln sie wie eine multiply-Kante.

next Next-Kanten wirken sich nur auf Variations-Bausteine aus, andere Bausteine werden durch sie nicht beeinflusst.

Wird die Klasse BlockConst in eine RGG-Datei statisch importiert,

```
import static de.grogra.blocks.BlockConst.*; (1)
```

so stehen die Kantentypen in dieser Art zur Verfügung:

```
-multiply->    -child->    -next->
```

Andernfalls müssen sie folgendermaßen angesprochen werden:

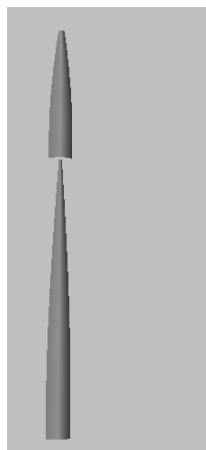
```
-BlockConst.multiply-> -BlockConst.child-> -BlockConst.next->
```

In allen nachfolgenden Beispielen wird davon ausgegangen, dass die BlockConst-Klasse wie in (1) angegeben importiert wurde.

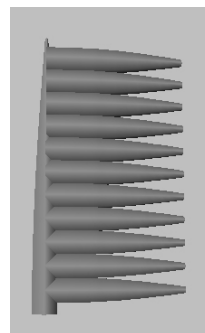
Zur besseren Veranschaulichung der Funktionsweise der child- und multiply-Kante in Verbindung mit Horn- oder Tree-Bausteinen seien die beiden nachfolgenden XL-Codezeilen gegeben.

```
Axiom ==> Horn -child-> Horn(10).(setLength(0.5f));
Axiom ==> Horn -multiply-> Horn(10).(setLength(0.5f));
```

In beiden Fällen wird ein Horn-Baustein mit einem weiteren Horn-Baustein verbunden. Im ersten Fall wird die child-Kante verwendet, was dazu führt, dass die erzeugte Instanz in Wuchsrichtung des Vaterobjekts an dessen oberem Ende gezeichnet wird, das Resultat ist in Abbildung 5.5a zu sehen. Die vervielfältigten Instanzen werden rechtwinklig zur Z-Achse der Vaterinstanz gezeichnet (Abbildung 5.5b) wenn wie in der zweiten Zeile eine multiply-Kante verwendet wird.



(a) child-Kante



(b) multiply-Kante

Abbildung 5.5: Auswirkung von child- und multiply-Kanten

5.4 Horn

Der Horn-Baustein stellt einen verallgemeinerten (generalisierten) Zylinder dar, der in der Grundeinstellung aus 20 übereinander angeordneten Zylindern besteht, die aufgrund der Range-Einstellung konisch zulaufen und somit dem Objekt eine "hornähnliche" Form geben. Ein generalisierter Zylinder ist ein erweitertes Zylindermodell, gebildet aus einer

Grundfläche, z. B. einem Viereck, welche im Raum entlang einer Bahn verschoben wird. Das durch die Verschiebung der Fläche eingeschlossene Volumen bildet den generalisierten Zylinder.

Wie bereits in 5.3 beschrieben, werden Objekte in Abhängigkeit vom verwendeten Kanten-typ, mit dem sie verbunden sind, unterschiedlich von dem Horn-Baustein vervielfältigt. Die über eine child-Kante angehängten Objekte werden senkrecht auf die Spitze des Horns gezeichnet, wohingegen Objekte, die über eine multiply-Kante mit dem Horn verbunden sind, rechtwinklig zur Hauptachse des Horn, von unten beginnend, an jedem Segmentknoten angebracht werden, s. Abbildung 5.5a und 5.5b.

Attribute:

Segments *Funktionsfeld* (20, 1, 100)

Regelt die Anzahl der Segmente und damit die Anzahl, wie oft ein angehängtes Objekte vervielfältigt wird, wenn es über eine multiply-Kante verknüpft ist.

numberPerSegment *Funktionsfeld* (1, 1, 10)

Bestimmt die Anzahl der vervielfältigten Instanzen, die pro Segment erzeugt werden, wobei die Verteilung gleichmäßig um die Hauptachse erfolgt.

Length *Funktionsfeld* (1, 0.001, 10)

Bestimmt die Gesamtlänge des Horn-Objekts. Der angegebene Wert wird intern mit 20 multipliziert und anschließend durch die Anzahl der Segmente geteilt, um so die Länge eines Segments zu bestimmen.

rotEndAngle *Funktionsfeld* (11.6, 0, $6 * \pi$)

Summe der absoluten Winkel, im Bogenmaß, um den alle Segmente des aktuellen Horn-Bausteins zusammen maximal gedreht werden. Das i -te Segment wird um den Winkel $i * \alpha$ gedreht, wobei sich α , das hier vorerst frei von der Rotationsrichtung betrachtet wird, folgendermaßen ergibt.

$$\begin{aligned} \text{rotEndAngle} &= 1 * \alpha + 2 * \alpha + 3 * \alpha + \dots + \text{segments} * \alpha \\ &= (1 + 2 + 3 + \dots + \text{segments}) * \alpha = \sum_{i=1}^{\text{segments}} i \\ \alpha &= \text{rotEndAngle} / \sum_{i=1}^{\text{segments}} i \end{aligned}$$

Der Ausgangswert von 11.6° ($\approx 3.69 * \pi$) entspricht etwas mehr als eineinhalb Umdrehungen für das letzte Segment in Bezug zur Ausgangsrichtung.

Dieser Maximalwinkel ist für alle Rotationsparameter der selbe.

rotX *Komplexfeld* (0, -1, 1 : Id)

Erlaubt eine Rotation um die X-Achse, s. Abbildung 5.6a, wobei der Wert angibt,

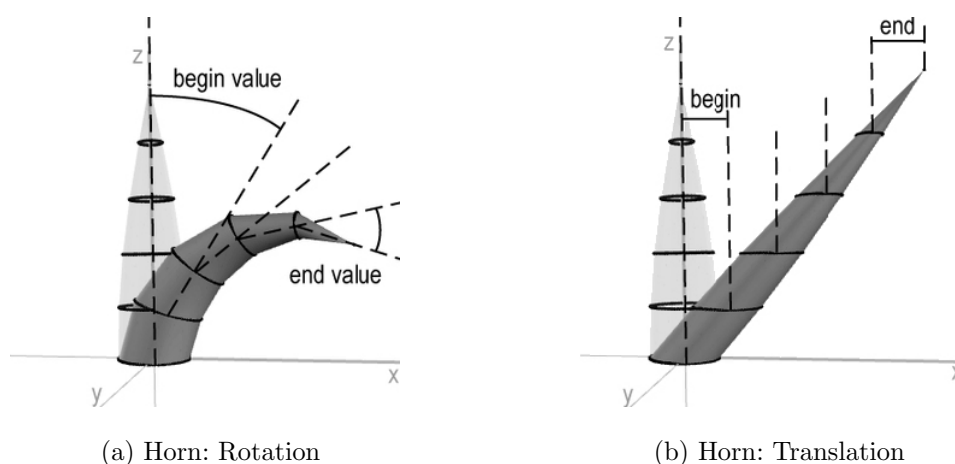


Abbildung 5.6: Auswirkung der Rotations- und Translationsattribute (Bilder: Deussen)

wie viel Prozent von α (s. voriges Attribut) letztendlich in die Berechnung der X-Rotation eingehen.

$$\alpha_X = \alpha * rotX$$

Dadurch, dass der Wertebereich von minus bis plus eins reicht, ist eine Rotation in beide Richtungen möglich.

rotY *Komplexfeld* (0, -1, 1 : Id)

Analog zu *rotX*, nur in bezug zur Y-Achse.

rotZ *Komplexfeld* (0, -1, 1 : Id)

Analog zu *rotX*, nur in bezug zur Z-Achse.

transX *Komplexfeld* (0, -20, 20 : Id)

Erlaubt die Definition einer Translation entlang der X-Achse, um die die Segmente in beide Richtungen verschoben werden können, s. Abbildung 5.6b.

transY *Komplexfeld* (0, -20, 20 : Id)

Analog zu *transX*, nur in Bezug zur Y-Achse.

transZ *Komplexfeld* (0, $-\pi$, π : Id)

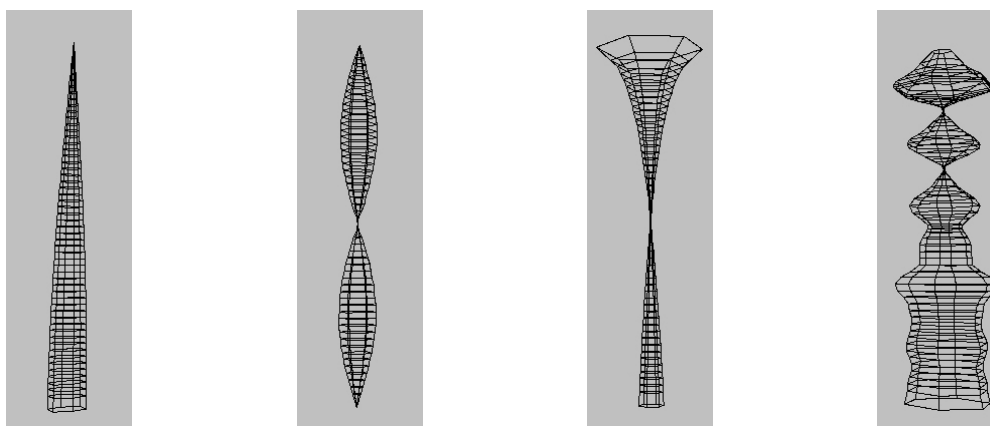
Analog zu *transX*, nur in Bezug zur Z-Achse.

useShape *CheckBox* (*false*)

Bestimmt, ob das *Shape*-Attribut oder das *range*-Attribut die Außenlinie des Horn-Objekts bestimmt. Ist *useShape* aktiviert, wird *Shape* verwendet, sonst *range*.

Shape *FloatToFloat* ("*Functions/Tree/shape*")

Die hier definierte Funktion bestimmt die Außenlinie des Horn-Objekts, indem es die einzelnen Segmente entsprechend skaliert. *Shape* wird nur verwendet, wenn durch das *useShape*-Attribut aktiviert.



- (a) Initialeinstellung: $\text{range1}=\pi/2$, $\text{range2}=0$, Funktion= Cos
- (b) $\text{range1}=\pi$, $\text{range2}=-\pi$, Funktion= Sin
- (c) $\text{range1}=0.05$, $\text{range2}=2$, Funktion= Log
- (d) $\text{range1}=3$, $\text{range2}=3$, Funktion= Rnd

Abbildung 5.7: Verschiedene Range-Einstellungen bei Horn

range *Komplexfeld* $(0, -\pi, \pi; \pi/2, -\pi, \pi : \text{Cos})$

Ist das *useShape*-Attribut deaktiviert, bestimmt *range* den Durchmesser der Segmente und somit die Außenlinie des Horn-Objekts. In der Grundeinstellung ist das Horn-Objekt ein konisch zulaufender Zylinder, s. Abbildung 5.7a:

Dieses XL-Beispiel generiert vier Horn-Instanzen, zwischen die mit der Regel B ein kleiner Abstand eingefügt wurde.

```
B ==> RU(90) M(10) RU(-90);
Axiom ==>
Horn B
Horn.(setRange1("pi"),setRange2("-pi"),setRangeMode(new Sin())) B
Horn.(setRange1(0.05),setRange2(2),setRangeMode(new Log())) B
Horn.(setRange1(3),setRange2(3),setRangeMode(new Rnd()));
```

Die Auswirkungen verschiedener Range-Einstellungen sind in Abbildung 5.7 zu sehen. Die auf das angegebene Intervall angewendeten Funktionen lauten der Reihe nach Kosinus, Sinus, Logarithmus und Random.

top *CheckBox* (*true*)

Gibt an, ob das letzte Segment spitz zuläuft oder entsprechend der *range*- bzw. *Shape*-Einstellung offen bleibt.

scale *Komplexfeld* $(1, 0, 5 : \text{Id})$

Der über diesen Parameter definierte Skalierungsfaktor wirkt sich nur auf die vom

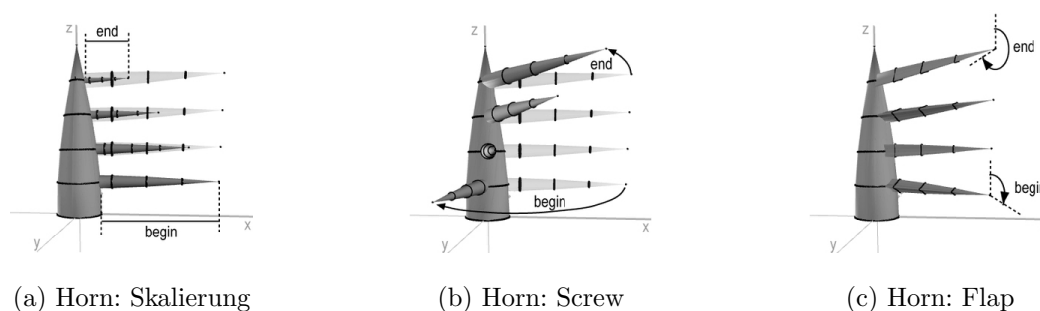


Abbildung 5.8: Verschiedene Zweigmanipulationen des Horn-Bausteins (Bilder: Deussen)

Horn vervielfältigten Instanzen aus. Die erzeugten Objekte werden wie in Abbildung 5.8a skaliert.

steps *Komplexfeld* (1, 0, 200 : *Id*)

In der Initialeinstellung werden die von Horn erzeugten Instanzen nach jedem Segment der Vaterinstanz gezeichnet, mit *steps* kann diese Anordnung gesteuert werden.

screw *Komplexfeld* (0, -50, 50 : *Id*)

Screw beschreibt einen Winkel, um den die vervielfältigten Instanzen um die Hauptachse der Vaterinstanz herum gedreht werden, s. Abbildung 5.8b.

flap *Komplexfeld* (0, -7, 7 : *Id*)

Mit flap wird ein Winkel definiert, um den die vervielfältigten Instanzen um ihre eigene Hauptachse herum gedreht werden, Abbildung 5.8c.

Profile *BSplineKurve* (*RegularPolygon*)

Dieses Attribut beschreibt die Grundfläche, die einem Horn-Objekt zugrunde liegt. Damit ist es z. B. möglich, dreieckige, runde oder beliebige selbst definierte Formen als Grundfläche festzulegen.

Geometry *CheckBox* (*true*)

Mit diesem Attribut kann die Oberfläche des Horn-Objekts ein- bzw. ausgeblendet werden. Der Benutzer kann so die Geometrie des Horn-Bausteins abschalten.

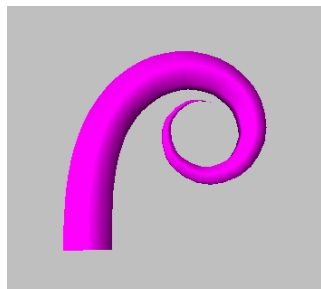
lod *LODFeld*

Die implementierten Methoden werden im Unterkapitel 6.2 beschrieben.

Im folgenden XL-Beispiel wird ein Horn-Objekt angelegt, dem über ein Phong-Objekt eine Farbe zugeordnet wird. Innerhalb des Phong-Objekts wird nur der Diffuse- und Specular-Teil gesetzt, der Ambient-, Emissive- und Transparency-Teil stehen ebenfalls zur Verfügung.

```
Axiom ==> Horn.(setRotX1(0.75), setMaterial(
```

```
new Phong().(setDiffuse(new RGBColor(1, 0, 1)),
            setSpecular(new RGBColor(0, 0, 1) ));
```



(a) mit Farbe



(b) mit (Rinden-)Textur

Abbildung 5.9: Beispiel für Horn-Objekt

Zusätzlich wurde das Horn-Objekt mit der Angabe `setRotX1(0.75)` um die X-Achse gedreht. Im OpenGL-Modus der GroIMP-Software entspricht das Ergebnis dem in Abbildung 5.9a dargestellten Objekt. Statt einer Farbe können auf diesem Weg beliebige Texturen als Farbe gesetzt werden. Abbildung 5.9b zeigt ein Horn-Objekt mit einer Rinden-Textur, die als diffuse Farbe gesetzt wurde.

5.5 Tree

Der Tree-Baustein ist ein speziell für die Konstruktion von Bäumen gedachtes Verzweigungselement. Wie der Horn-Baustein erzeugt er in der Grundeinstellung eine eigene Geometrie, entlang der die vervielfältigten Instanzen (im Folgenden auch Zweige genannt) angeordnet werden. Der Tree-Baustein ist der komplexeste der Bausteine, er stellt ein Reihe der Natur entlehnte Attribute bereit, die den modellierten Baummodellen ein möglichst realistisches Aussehen geben.

Attribute:

Stem Length *Funktionsfeld* (20, 0, 200)

Bestimmt die Gesamtlänge des Tree-Objekt. Der angegebene Wert wird der Verteilung der Segmente entsprechend auf diese verteilt.

Trunk Scale *Funktionsfeld* (1, 0, 2)

Skalierungsfaktor, der nicht nur die Länge der Segmente beeinflusst, es werden ebenfalls die von dem Segment ausgehenden Instanzen mit diesem Faktor skaliert.

Spline *CheckBox* (*false*)

Ist dieses Feld aktiviert, so wird die Hauptachse des Tree-Bausteins durch die mittels

des *Trajectory*-Attributs beschriebene Kurve bestimmt, sonst folgt die Hauptachse der Z-Achse.

Trajectory *BSplineKurve* (" *Functions/Tree/trajectory* ")

Mittels dieser Kurve wird die Hauptachse des Tree-Bausteins definiert, wenn das *Spline*-Attribut auf *true* steht.

Shape *FloatToFloat* (" *Functions/Tree/shape* ")

Die hier definierte Funktion bestimmt die Außenlinie des Stamms, indem es die einzelnen Segmente entsprechend skaliert.

Top *CheckBox* (*true*)

Gibt an, ob das letzte Segment spitz zuläuft oder entsprechend der *Shape*-Einstellung offen bleibt.

Crookedness Amount *Funktionsfeld* (2, 0, 10)

Mittels diesen Attribut wird ein Faktor definiert, der angibt, wie stark der Betrag der Unregelmäßigkeit ist, der letztlich dem Tree-Objekt als Irregularität hinzugefügt wird.

Crookedness Intensity *FloatToFloat* (" *Functions/Tree/intensity* ")

Definiert, in welchem Maße eine Unregelmäßigkeit dem Tree-Objekt als Irregularität hinzugefügt wird. Es gibt also eine interne Kurve *crookedness* (genau genommen drei Funktionen, pro Achse eine),

$$\begin{aligned} CrookednessValue_{xyz} &= CrookednessAmount * \\ CrookednessIntensity.evaluateFloat(distribution[i]) &* \\ crookedness_{xyz}(distribution[i]) & \end{aligned}$$

die entsprechend der Zweigposition ausgewertet wird. Das Resultat wird nun noch mit dem Wert von *CrookednessIntensity* an der Stelle des *i*-ten Zweiges und dem Faktor *CrookednessAmount* multipliziert.

Deviate *FloatToFloat* (" *Functions/Tree/deviate* ")

Die *Deviate*-Funktion beschreibt ebenfalls eine Irregularität, die dem Tree-Objekt hinzugefügt werden kann. Die interne Kurve (eigentlich drei Funktionen) wird an der Position des *i*-ten Zweiges ausgewertet und mit dem Resultat der *Deviate*-Funktion an der selben Zweigposition multipliziert.

$$\begin{aligned} deviateValue_{xyz} &= Deviate.evaluateFloat(distribution[i]) * \\ deviate_{xyz}(distribution[i]) & \end{aligned}$$

Das Gesamtergebnis wird, wie das der *CrookednessIntensity*-Funktion, als Rotationen um die X-, Y- und Z-Achse auf das *i*-te Segment angewendet.

Screw *FloatToFloat* ("Functions/Tree/screw")

Bestimmt den Rotationswinkel, um den die Zweige um die Hauptachse (Z-Achse) des Tree-Objekts gedreht werden. Bei den Winkelangaben handelt es sich hier um relative Winkel, d. h. der i -te Winkel $screwValue_i$ berechnet sich aus der Summe der vorangegangenen Winkel:

$$screwValue_i = 180^\circ * \sum_{j=0}^i Screw.evaluateFloat(x_j) \text{ mit} \\ x_i = i / (BranchesNumberI - 1)$$

Die Zweige verteilen sich hierbei gleichmäßig auf den Definitionsbereich der *Screw*-Funktion, deren Definitionsbereich das Intervall von null bis eins ist, woraus sich die obige Berechnung des x_i ergibt. Um den Winkel zu erhalten, wird die Summe mit 180° multipliziert.

Branches Number *Funktionsfeld* (10, 2, 100)

Bestimmt die Anzahl der vervielfältigten Instanzen. Somit lässt sich z. B. die Anzahl der Äste bzw. Zweige eines Baumes bestimmen.

Arrangement *Auswahlfeld* (*perpendicular*)

Dieses Attribut bestimmt die Art der Anordnung der vervielfältigten Instanzen um die Hauptachse des Tree-Objekts herum. Es stehen mit *perpendicular* (senkrecht) und *lateral* (seitlich) zwei Hauptkategorien, die jeweils frei, abwechselnd und paarweise verwendet werden können, sechs Voreinstellungen, die häufig in der Natur vorkommen, zur Auswahl.

perpendicular Arrangiert die Zweige senkrecht zur Hauptachse des Vaterobjekts in freier Anordnung, s. Abbildung 5.10a.

lateral Arrangiert die Zweige parallel zur Hauptachse des Vaterobjekts in freier Anordnung, s. Abbildung 5.10e.

alternate perpendicular Arrangiert die Zweige senkrecht zur Hauptachse des Vaterobjekts abwechselnd auf beiden Seiten, s. Abbildung 5.10b.

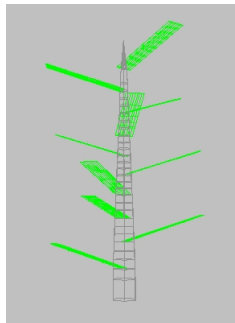
alternate lateral Arrangiert die Zweige parallel zur Hauptachse des Vaterobjekts abwechselnd auf beiden Seiten, s. Abbildung 5.10f.

pair perpendicular Arrangiert die Zweige senkrecht zur Hauptachse des Vaterobjekts paarweise auf beiden Seiten, s. Abbildung 5.10c.

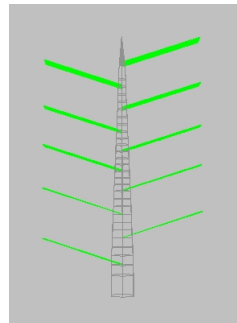
pair lateral Arrangiert die Zweige parallel zur Hauptachse des Vaterobjekts abwechselnd auf beiden Seiten, s. Abbildung 5.10g.

one-sided perpendicular Arrangiert die Zweige senkrecht zur Hauptachse des Vaterobjekts einseitig, s. Abbildung 5.10d.

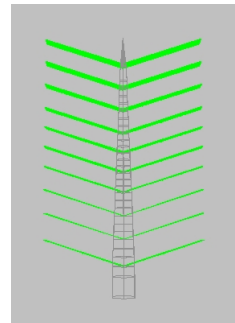
one-sided lateral Arrangiert die Zweige parallel zur Hauptachse des Vaterobjekts einseitig, s. Abbildung 5.10h.



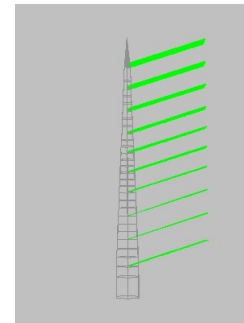
(a) freie senkrechte Anordnung



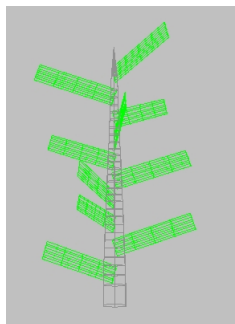
(b) abwechselnd senkrechte Anordnung



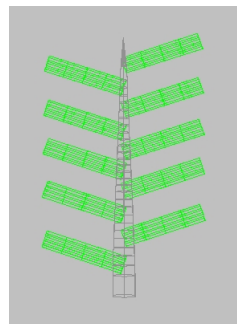
(c) paarweise senkrechte Anordnung



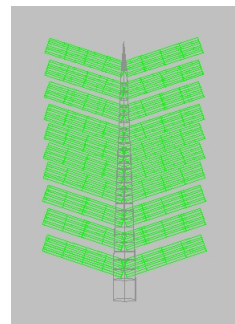
(d) einseitig senkrechte Anordnung



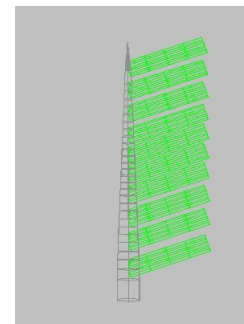
(e) freie laterale Anordnung



(f) abwechselnd laterale Anordnung



(g) paarweise laterale Anordnung

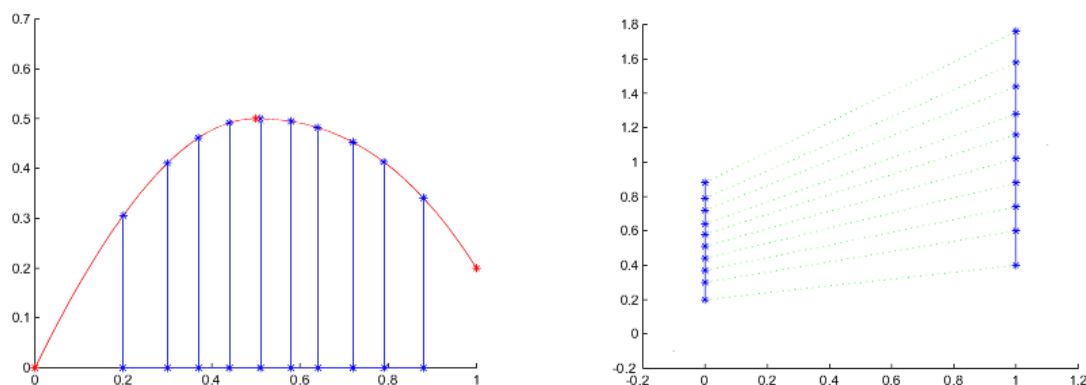


(h) einseitig laterale Anordnung

Abbildung 5.10: Verschiedene Arrangement-Einstellungen des Tree-Bausteins

Branches Distribution *FloatToFloat* ("Functions/Tree/distribution")

Diese Funktion bestimmt die Verteilung der Zweige entlang der Hauptachse des Tree-Bausteins. Die Berechnung wird nun an der Initialfunktion (s. Abbildung 5.11) demonstriert. Zuerst wird das Integral I unter der Funktion bestimmt, im zweiten Schritt wird die so ermittelte Fläche durch die Anzahl der Zweige *BranchesNumber* plus eins geteilt, um so die Fläche pro Zweig d_i zu ermitteln. Der letzte Schritt besteht darin, die Gesamtfläche der Reihe nach, beginnend bei Null, in *BranchesNumber* viele Teile der Größe d_i einzuteilen, so dass sich das Bild 5.11a ergibt. Die X-Werte der so ermittelten Punkte bilden die relativen Positionen der Zweige des Tree-Objekts, nachfolgend mit *distribution* bezeichnet. Um die Endposition wie in 5.11b zu berechnen, werden die Werte noch mit der Länge des Tree-Objekts multipliziert (hier beispielsweise mit 2).

(a) Integralunterteilung in d_i große Bereiche

(b) Skalierung der ermittelten X-Werte

Abbildung 5.11: Tree Branches Distribution-Berechnung

Branches GrowthScale *FloatToFloat* ("Functions/Tree/growthscale")

Dieses Attribut bestimmt einen Skalierungsfaktor, der sich nur in vervielfältigten Tree-Bausteinen, somit erst in der nächsten Ebene, auswirkt. Der i -te Faktor ergibt sich aus der Auswertung der *BranchesGrowthScale*-Funktion an der i -ten Position.

$$branchesGrowthScale_i = BranchesGrowthScale.evaluateFloat(distribution[i])$$

Branches GeometricScale *FloatToFloat* ("Functions/Tree/geomscale")

Definiert einen Skalierungsfaktor, mit dem die Zweige skaliert werden. Der i -te Faktor wird entsprechend dieser Formel berechnet:

$$branchesGeometricScale_i = BranchesGeometricScale.evaluateFloat(distribution[i])$$

Dabei wird die *BranchesGeometricScale*-Funktion an der i -ten Position ($distribution[i]$) ausgewertet.

Branches Angle *FloatToFloat* ("Functions/Tree/angle")

Gibt den Neigungswinkel zwischen der Hauptachse (Z-Achse) und den Zweigen an. Um den i -ten Winkel zu erhalten, wird die *BranchesAngle*-Funktion an der i -ten Position ($distribution[i]$) ausgewertet und das Ergebnis mit 180° multipliziert.

$$branchAngle_i = 180^\circ * BranchesAngle.evaluateFloat(distribution[i])$$

Somit ist es beispielsweise möglich, einen Zweig mit 0° Neigungswinkel in Wachsrichtung und mit 180° , entsprechend einem Funktionswert von eins, ihn entgegengesetzt zeigen zu lassen.

Branches Dense *FloatToFloat ("Functions/Tree/dense")*

Bestimmt die Dichte der Zweige in der nachfolgenden Zweigebene bei vervielfältigten Tree-Objekten. Um den i -ten Faktor zu bestimmen, wird die *BranchesDense*-Funktion an der i -ten Position (*distribution[i]*) der Zweigverteilung entsprechend ausgewertet. Dieser Wert wird in die nächste Ebene weitergegeben und mit der Anzahl der Zweige *BranchesNumber* multipliziert.

Profile *BSplineKurve (RegularPolygon)*

Dieses Attribut beschreibt die Grundfläche, die einem Tree-Objekt zugrunde liegt. Damit ist es z. B. möglich, dreieckige, runde oder beliebige selbst definierte Formen als Grundfläche festzulegen.

Geometry *CheckBox (true)*

Mit diesem Attribut kann die Oberfläche des Tree-Objekts ein- bzw. ausgeblendet werden. Der Benutzer kann so die Geometrie des Tree-Bausteins abschalten.

lod *LODFeld*

Die implementierten Methoden werden im Unterkapitel 6.2 beschrieben.

In diesem XL-Code-Beispiel vervielfältigt ein Tree-Objekt Blätter, sog. leaf-Objekte, die von XL bereitgestellt werden. Die Länge und Breite kann dabei als Parameter an das leaf-Objekt übergeben werden.

```
Axiom ==> Tree -multiply-> leaf (3, 1);
```

Das Resultat des Beispiels ist eine astähnliche Struktur bestehend aus einem Zweig mit Blättern, zu sehen in Abbildung 5.10a.

5.6 Wreath

Das Wreath-Objekt, wie es in Xfrog v.3.5 noch vorhanden ist, wurde nicht extra implementiert, da der Hydra-Baustein nach Einfügen eines Neigungsparameters seine Funktionalitäten vollständig übernimmt. Ein Hydra-Baustein mit den Einstellungen Radius=0 und Neigungswinkel=0 entspricht somit einen Wreath-Baustein.

Dies geschah aufgrund der in Xfrog v.4.0 vorgenommenen Zusammenfassung beider Bausteine.

5.7 Hydra

Der Hydra-Baustein vervielfältigt jede mit ihm verbundene Struktur und ordnet sie, in der Grundeinstellung, in einem Kreis an. Die Erweiterung der Attribute mit einem Neigungs-

attribut ermöglicht es, dass der Hydra-Baustein die Funktionalität des Wreath-Bausteins abdeckt und daraufhin der Wreath-Baustein entfallen kann. Durch die Einführung einer Trajektorie, die die Kurve, auf der die vervielfältigten Instanzen angeordnet werden, festlegt, ist es jetzt möglich, auf eine Reihe vordefinierter Kurven zuzugreifen, die entsprechend parametrisierbar sind. Es können auch selbst definierte Kurven geladen werden und so die Objekte frei platziert werden. Dies ist eine erheblicher Erweiterung gegenüber der Xfrog-Hydra-Komponente, bei der die Anordnung immer in einem Kreis variablen Radius erfolgt. Somit hat der Modellierer mehr Freiheiten.

Attribute:

number *Funktionsfeld* (10, 1, 100)

Bestimmt die Anzahl von Kopien, die vom zu vervielfältigenden Objekt erzeugt werden.

trajectory *BSplineKurve* (*Circle*(3))

Diese Option legt die Kurve fest, auf der die vervielfältigten Instanzen gleichabständig positioniert werden. *Circle*(3) beschreibt einen Kreis mit Radius 3 als Initialkurve.

twist *Komplexfeld* (0, $-\pi$, π : *Id*)

Definiert eine Rotation der vervielfältigten Objekte um die Y-Achse.

spin *Komplexfeld* (0, $-\pi$, π : *Id*)

Definiert eine Rotation der vervielfältigten Objekte um die Z-Achse.

scale *Komplexfeld* (0, $-\pi/2$, $\pi/2$: *Cos*)

Dieser Parameter erlaubt es, einen Skalierungsfaktor für alle vervielfältigten Instanzen zu bestimmen.

slopeFunction *FloatToFloat* (*Functions/Hydra/slopeHydra*)

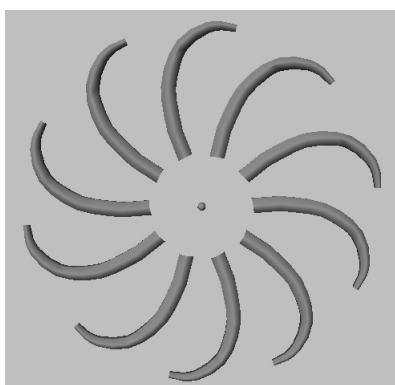
Definiert einen Neigungswinkel, bei dem die erzeugten Instanzen sowohl um die Y-Achse als auch um die Z-Achse gedreht werden, s. Abbildung 5.12.

lod *LODFeld*

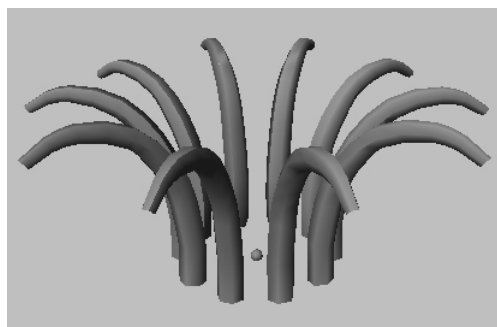
Die implementierten Methoden werden im Unterkapitel 6.2 beschrieben.

Die Abbildung 5.12 zeigt Horn-Bausteine, die durch einen Hydra-Baustein vervielfältigt werden. In Abbildung 5.12a ist die Neigung des Hydra-Bausteins auf Hydra eingestellt, in 5.12b auf Wreath. Der XL-Code, mit dem die Abbildungen erzeugt wurden, lautet:

```
Axiom ==> Hydra(10).(setRadius(6)) -multiply->
          Horn(10).(setRotX1(0.25f));
```



(a) Hydra-Einstellung



(b) Wreath-Einstellung

Abbildung 5.12: Verschiedene Neigungseinstellungen des Hydra-Bausteins

5.8 PhiBall

Der PhiBall-Baustein ist ein Vervielfältigungsobjekt, das alle mit ihm verbundenen Strukturen vervielfältigt und die erzeugten Instanzen nach dem Prinzip des Goldenen Schnitts (Kapitel 3.1.2) auf der Oberfläche eines Ellipsoids anordnet. Als Erweiterung zu Xfrog v3.5, wo nur eine Kugel als Grundkörper zur Verfügung steht, ist es jetzt möglich, die X-, Y- und Z-Radien bis zu einem maximalen Wert von 50 frei zu definieren, womit ein für die meisten Anwendungen ausreichend großes Objekt erzeugt werden kann. Ebenfalls neu ist, dass der PhiBall-Baustein in dieser Implementierung eine optional zuschaltbare eigene Geometrie erzeugen kann.

Attribute:

number *Funktionsfeld* (50, 1, 500)

Definiert die Anzahl der vervielfältigten Instanzen.

radiusX *Funktionsfeld* (0.5, 0, 50)

Gibt den X-Radius des Ellipsoids an.

radiusY *Funktionsfeld* (0.5, 0, 50)

Gibt den Y-Radius des Ellipsoids an.

radiusZ *Funktionsfeld* (0.5, 0, 50)

Gibt den Z-Radius des Ellipsoids an.

fan *Komplexfeld* (0, 0, π ; π , 0, π : *Id*)

Gibt den Winkel, im Bogenmaß, ausgehend vom Mittelpunkt des PhiBalls zur Z-Achse an, in dem die erste bzw. die letzte generierte Instanz gezeichnet wird. In der Grundeinstellung wird eine Kugel gebildet. Das Beispiel in Abbildung 5.13a zeigt eine Art Gürtel, der mit der entsprechenden Fan-Einstellung erzeugt wird.

angle *Komplexfeld* (1, -10, 10 : *Phi*)

Mit *angle* wird der Winkel bestimmt, in den die Objekte auf der Oberfläche angeordnet werden. Die Standardeinstellung ist *phi*, die dem Fibonacci-Winkel entspricht, so dass die Gesamtanordnung einer Verteilung nach dem Goldenen-Schnitt entspricht. Die entsprechenden Berechnungen werden in Abschnitt 3.1.2 auf Seite 33 erläutert.

trans *Komplexfeld* (0, -10, 10 : *Id*)

Bestimmt einen Translationswert entlang der Z-Achse des PhiBalls, um den die Instanzen verschoben werden. Dies erlaubt eine Verzerrung einer Kugel, wie in Abbildung 5.13b zu sehen.

scale *Komplexfeld* (0, $-\pi$, π : *Cos*)

Mit diesem Parameter wird ein Skalierungsfaktor für jede vervielfältigte Instanz bestimmt.

influence *Funktionsfeld* (1, 0, 1)

Dieser Parameter bestimmt bei skalierten Instanzen die Position in Abhängigkeit von ihrer Größe, wobei Größe hier als Fläche, die sie auf der Kugel einnehmen, zu verstehen ist. Da die Größe der Instanzen zum Zeitpunkt der Positionsberechnung nicht bekannt ist, wird der Skalierungsfaktor als Ersatz für die Größe betrachtet und für die Berechnungen herangezogen. Die genaue Berechnungsvorschrift wird in Abschnitt 3.1.2 erläutert.

Mit der Einstellung Null wird die Größe der Objekte bei der Platzierung nicht beachtet, wohingegen sie bei Eins zu 100% mit in die Positionsberechnung eingeht, dann wird der Abstand zwischen kleinen Objekten verkleinert, bei großen hingegen vergrößert, um eine gleichmäßige Nutzung der Fläche zu erzielen.

In Abbildung 5.14 sind zwei Beispiele gegeben, die den Effekt verdeutlichen.

Geometry *CheckBox* (*false*)

Mit diesem Attribut kann die Oberfläche des Ellipsoids, auf dem die Instanzen angeordnet werden, ein- bzw. ausgeblendet werden. Der PhiBall-Baustein kann somit eine eigene Geometrie erzeugen.

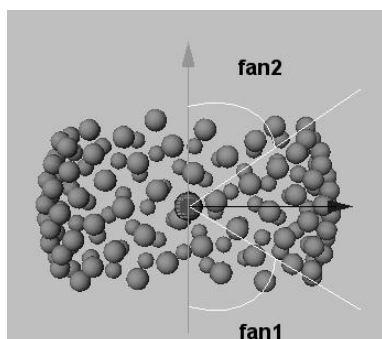
Diese Erweiterung kam aufgrund der Analyse einiger Xfrog-Modelle zustande, bei denen auffiel, dass bei der Modellierung von Blüten oftmals ein Kugelobjekt mit einem PhiBall-Objekt kombiniert wurde, um so dem PhiBall einen "Körper" zu geben, damit die Blütenblätter nicht frei in der Luft hängen. Dieser "Körper" kann nun direkt zugeschaltet werden, das zusätzliche Kugelobjekt entfällt somit.

lod *LODFeld*

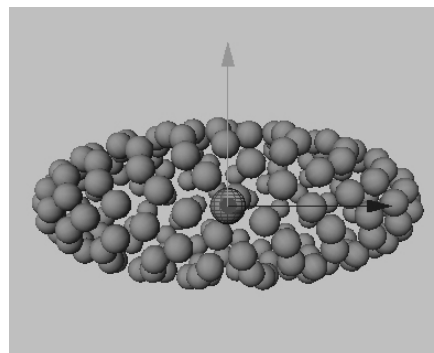
Die implementierten Methoden werden im Unterkapitel 6.2 beschrieben.

Der folgende XL-Code erzeugt eine Kugel mit einem Radius von 5, auf der 150 kleine Kugeln verteilt werden.


```
Axiom ==> PhiBall(150).(setRadius(5)) -multiply-> Sphere(0.5);
```



(a) fan2=1; fan1=2.1



(b) trans2=3; trans1=-3

Abbildung 5.13: PhiBall-Manipulationen

5.9 Variation

Der Variations-Baustein erlaubt eine Variation, auf verschiedene Arten, zwischen einem Vervielfältigungsobjekt und einer Menge von Objekten. Dazu muss das Vervielfältigungsobjekt mit einem Variations-Baustein verbunden werden und alle Objekte, die variiert werden sollen, müssen mittels next-Kanten als Kinder mit dem Variations-Baustein verknüpft werden. Diese können dann regelmäßig oder zufällig abwechselnd vervielfältigt werden, oder aber es kann eine bestimmte Ausnahmeposition definiert werden.

Attribute:

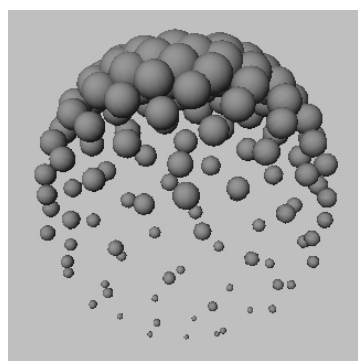
Type *Auswahlfeld (random)*

Mit diesem Parameter kann die Art der Anordnung, in der die vervielfältigten Objekte positioniert werden festgelegt, werden. Dabei stehen folgende vier Verfahren zur Auswahl:

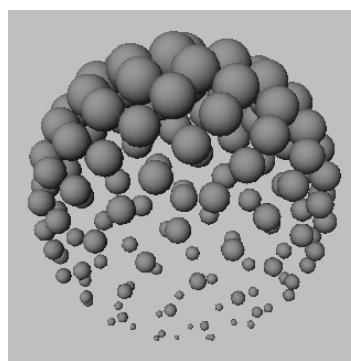
random Erzeugt eine zufällige Reihenfolge, wobei es durchaus vorkommen kann, dass eine Objektart nicht ausgewählt wird, s. Abbildung 5.15a.

sequential Erzeugt eine fortlaufende Anordnung, beachtet dabei die Reihenfolge, in der die Objekte mit der Variation verbunden sind, s. Abbildung 5.15b.

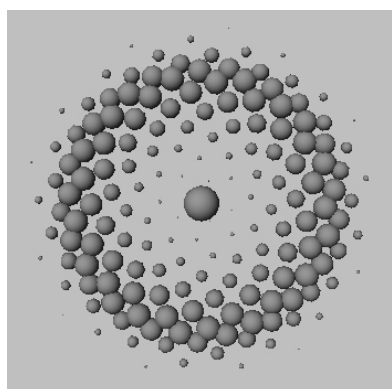
distribute Versucht eine Verteilung der Objekte in gleichen Teilen zu erzeugen, wobei bei nicht glatt aufgehender Rechnung das letzte Objekt zur Regulation verwendet wird, s. Abbildung 5.15c.



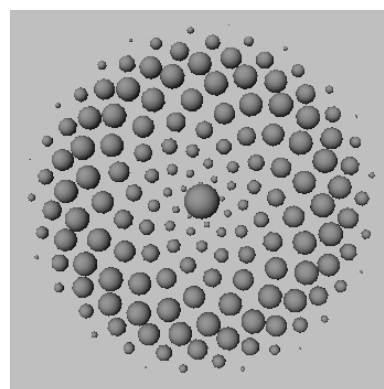
(a) Seitenansicht einer Kugel: Influence=0



(b) Seitenansicht einer Kugel: Influence=1



(c) Draufsicht auf eine Kreisscheibe: Influence=0



(d) Draufsicht auf eine Kreisscheibe: Influence=1

Abbildung 5.14: Verschiedene Influence-Einstellungen bei PhiBall

exception In dieser Einstellung wird bis auf eine Position, die über das *exceptionValue*-Attribut definierte Positionsnummer, nur das erste Unterobjekt verwendet, nur im Ausnahmefall wird das zweite benutzt, s. Abbildung 5.15d.

seed *Integer* (*zufall*, 0, *x*)

Mit diesem Longinteger-Wert wird der Zufallsgenerator instanziiert. Durch Setzen des selben Wertes wird erreicht, dass der Zufallsgenerator die selben Zufallszahlen generiert und somit das gleiche Ergebnisbild der Variation mit der Type-Einstellung *random* erzeugt wird. Das Attribut wird somit nur in der Type-Einstellung *random* beachtet.

exceptionValue *Integer* (0, 0, *x*)

Dieses Attribut wird nur in der Type-Einstellung *exception* benutzt. Das über diese Positionsnummer angegebene Objekt wird als Ausnahme angesehen. Ist die Nummer größer als die Anzahl der von dem Vervielfältigungsobjekt zu erzeugenden Objekte, wird das entsprechende "Restobjekt", nach einer Division mit Rest, ersetzt.

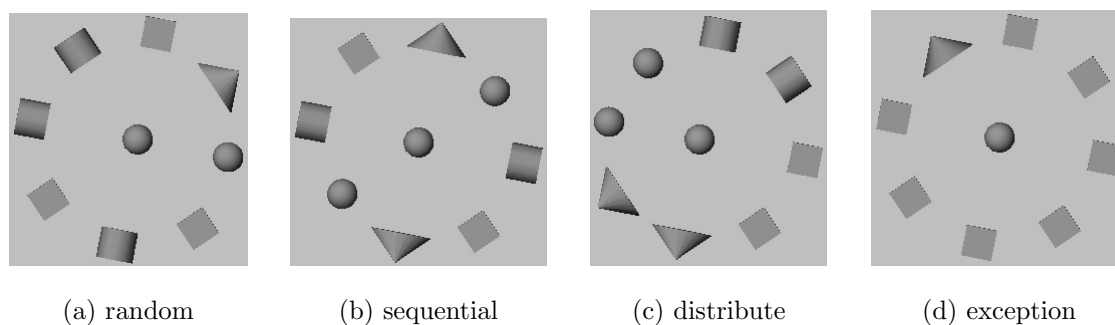


Abbildung 5.15: Verschiedene Type-Einstellungen des Variations-Bausteins

Sollen beispielsweise 4 Primitive (Kugeln, Kegel, Würfel und Zylinder) in einem Kreis angeordnet und mittels eines Variations-Bausteins in ihrer Reihenfolge zufällig variiert werden, würde dies in XL so aussehen:

```
Axiom ==> Hydra(8) -multiply->
    Variation() -next-> Sphere(0.5) -next-> Cone()
    -next-> Box() -next-> Cylinder(0.5);
```

Das Hydra-Objekt erzeugt 8 Instanzen, die dazugehörige grafische Ausgabe ist in Abbildung 5.15 zu sehen.

5.10 BlockColor

Mit dem BlockColor-Baustein ist es möglich, eine Farbe zu definieren und zu setzen. Die aktuell gesetzte Farbe wirkt sich als Baumattribut auf alle nachfolgenden Objekte des von diesem Knoten ausgehenden Teilbaumes aus, bis eine neue Farbe gesetzt wird. Zu Verfügung stehen drei Funktionsfelder, mit denen der Rot-, Grün- und Blauanteil im Intervall von 0 bis 255 angegeben werden kann.

Der Grund für die Implementierung dieses Bausteins ist der dass es sich bei den Attributfeldern um Funktionsfelder handelt, in denen die internen Variablen (siehe Anhang C) zur Verfügung stehen und so in Abhängigkeit von ihnen die Farbe definiert werden kann. Attribute:

- addR** *Funktionsfeld* (0,0,255)
Definiert den Rotanteil der Farbe.
- addG** *Funktionsfeld* (0,0,255)
Definiert den Grünanteil der Farbe.
- addB** *Funktionsfeld* (0,0,255)
Definiert den Blauanteil der Farbe.

Das Beispiel in Abbildung 5.16a wurde durch diesen XL-Code erzeugt:

```
Axiom ==> Horn(45).(setRange1(0)) -multiply->
           BlockColor().(setAddR("h*15"), setAddB("h*10"))
           Horn(10).(setLength(0.5), setRange1(0));
```

Die Farbe der Seitenarme wird über die Höhe innerhalb des Objektes berechnet. Der Rotanteil wird mit 15, der Blauanteil mit 10 multipliziert, der Grünanteil bleibt konstant auf Null.

Folgenden XL-Notationen ermöglichen eine bequeme Eingabe der Farbwerte (*colorRGB* steht für einen gemeinsamen Farbwert aller Kanäle, *colorX* bestimmt einen einzelnen Kanal):

```
BlockColor(colorRGB)
BlockColor(colorR, colorG, colorB)
BlockColor().(setColor(colorRGB))
BlockColor().(setColor(faktorR, faktorG, faktorB))
BlockColor().(addR(faktorR), addG(faktorG), addB(faktorB))
```

Color kann hier entweder für eine Funktion stehen und wird dann durch Anführungszeichen begrenzt, oder für einen Zahlenwert, bei dem die Anführungszeichen entfallen können, aber nicht müssen.

5.11 BlockScale

Der BlockScale-Baustein kann in Verbindung mit allen Bausteinen verwendet werden, um diese zu skalieren. So ist es beispielsweise möglich, als Wurzel eines Teilgraphen ganze Objektteile gleichmäßig zu skalieren, wobei die Faktoren für jede Achse separat definiert werden können.

Der entscheidende Unterschied zum standard *Scale*-Objekt liegt darin, dass es sich bei den Attributfeldern um Funktionsfelder handelt, in denen die internen Variablen (siehe Anhang C) zur Verfügung stehen und somit eine Skalierung in Abhängigkeit von diesen Variablen möglich ist. Der maximale Skalierungsfaktor ist auf 25 begrenzt, was einer Vergrößerung von 2500 Prozent entspricht und somit für die meisten Anwendungen ausreichen sollte. Attribute:

scaleX *Funktionsfeld* (1, 0, 25)

Legt den Skalierungsfaktor der X-Achse fest.

scaleY *Funktionsfeld* (1, 0, 25)

Legt den Skalierungsfaktor der Y-Achse fest.

scaleZ *Funktionsfeld* (1, 0, 25)

Legt den Skalierungsfaktor der Z-Achse fest.

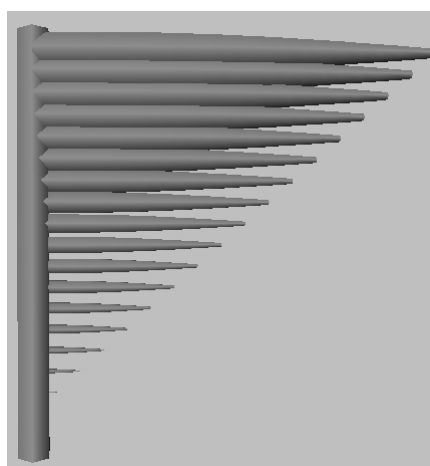
Dieses XL-Beispiel

```
Axiom ==> Horn(30).(setLength(1.2), setRange1(0)) -multiply->
           BlockScale().(setScale("i/15")) Horn(10);
```

erzeugt ein Horn-Objekt, an das mit einem BlockScale-Baustein skalierte Horn-Objekte angefügt wurden. Die Skalierung ist in diesem Fall von der Positionsnummer i abhängig, die durch 15 geteilt wird, um einen deutlichen Effekt zu erzielen. Die Methode *setScale* setzt für alle Achsen den selben Skalierungsfaktor. Das Resultat dieser Konstruktion ist in Abbildung 5.16b zu sehen.



(a) BlockColor



(b) BlockScale

Abbildung 5.16: Beispiel für die Bausteine BlockColor und BlockScale

Die folgenden XL-Notationen sind möglich, um die Faktoren zu setzen (*factorXYZ* definiert einen gemeinsamen Skalierungsfaktor für alle Achsen, *factorX* legt den Faktor für eine Achse fest):

```
BlockScale(factorXYZ)
```

```
BlockScale(factorX, factorY, factorZ)
```

```
BlockScale().(setScale(factorXYZ))
```

```
BlockScale().(setScale(factorX, factorY, factorZ))
```

```
BlockScale().(setScaleX(factorX), setScaleY(factorY), setScaleZ(factorZ))
```

Factor kann hier entweder eine Funktion sein und wird dann durch Anführungszeichen begrenzt, oder ein Zahlenwert, bei dem die Anführungszeichen entfallen können, aber nicht müssen.

5.12 Arrange

Da bei großen Mengen zu verteilerter Objekte nur in den seltensten Fällen eine individuelle Positionierung erfolgen kann, müssen effiziente Verfahren zur Modellierung ganzer Populationen gefunden werden. Der Arrange-Baustein bietet eine Vielzahl solcher Anordnungsmöglichkeiten.

Neben der Anordnung von Objekten auf einer planaren Grundfläche kann der Benutzer durch die Angabe eines Höhenfelds Terraindaten definieren, die die Grundfläche entsprechend deformieren, wie in Abbildung 5.17 dargestellt.

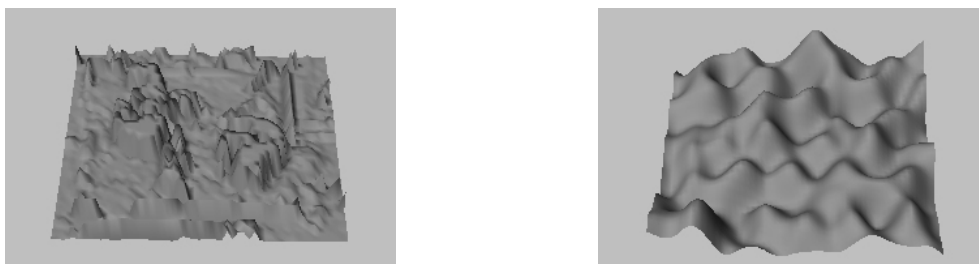


Abbildung 5.17: Beispiel für verschiedene Terrains

Der Arrange-Baustein stellt, als neuer Baustein, eine Erweiterung des von Xfrog vorgegebenen Bausteinkatalogs dar, der die Einsatzmöglichkeiten erheblich erhöht.

Attribute:

arrangeMethod *Auswahlfeld (Geometric)*

Dieses Feld ermöglicht die Auswahl der grundlegenden Anordnungsmethode. Dabei kann aus diesen Methoden gewählt werden:

Geometrische Anordnungen Ordnen die Objekte nach rein geometrischen Gesichtspunkten an, z. B. alle in Linien oder in Kreisen, s. Kapitel 5.12.1.

Wahrscheinlichkeitsanordnungen Ordnen die Objekte nach Wahrscheinlichkeitsverteilungen an, z. B. Poisson- oder Normal-Verteilung, s. Kapitel 5.12.2.

Halbtonverfahren Ordnen die Objekte auf einen vorgegebenen Dichtefeld an. Dabei wird das Dichtefeld als Vorlage genutzt und mittels Halbtonverfahren die Positionen bestimmt, s. Kapitel 5.12.3.

Zusätzliche Verfahren Sammlung von Verfahren, die sich nicht in die obigen Klassen einordnen lassen, z. B. Kachelungen oder iterative Verfahren wie Voronoi-Lloyd, s. Kapitel 5.12.4.

number *Textfeld (leer)*

Hierbei handelt es sich um ein reines Ausgabefeld, das jegliche Eingaben ignoriert. In ihm wird die Anzahl der im vorletzten Aktualisierungsschritt erzeugten Instanzen angezeigt. Das (leider) nur die Anzahl der vorletzten Aktualisierung und nicht die der aktuellen angezeigt wird, liegt daran, dass während des aktuellen Instanzierungsdurchgangs aus technischen Gründen keine Aktualisierung der Attribute vorgenommen werden kann.

random *RandomBase (leer)*

Dieses Feld fasst Attribute zusammen, mit denen die Positionen der erzeugten Instanzen zufällig verändert werden können. Siehe Kapitel 5.1 für eine genaue Beschreibung.

fitToRaster *CheckBox (false)*

Ist dieses Feld aktiviert, werden die erzeugten Instanzen auf die nächstliegenden Gitterlinien des Rasters gesetzt, sonst wird ihre Position nicht verändert.

scaleFunction *FloatToFloat (Functions/Arrange/scale)*

Dieses Feld erlaubt es, für jede vervielfältigte Instanz einen Skalierungsfaktor zu definieren. Der i -te Faktor $scale_i$ berechnet sich nach dieser Vorschrift:

$$scale_i = 1 + (MAX_SCALE * scaleFunction.evaluateFloat(x_i) - MAX_SCALE / 2),$$

wobei MAX_SCALE eine mit dem Wert 2 vereinbarte Konstante ist. Somit ist abhängig vom Ergebnis der Funktionsauswertung der $scaleFunction$ -Funktion, eine Skalierung von minus bis plus 100% Prozent möglich.

Der Wert von x_i ergibt sich aus der gleichmäßigen Verteilung von n Objekten auf dem Intervall $[0, 1]$, wie folgt:

$$x_i = i * 1/n = i/n$$

spinFunction *FloatToFloat (Functions/Arrange/spin)*

Definiert eine Rotation der erzeugten Instanzen um ihre eigene Hauptachse, die Z-Achse. Um den Winkel $spin_i$ zu bestimmen, um den das i -te Objekt gedreht werden soll, wird die $spinFunction$ -Funktion an der Stelle x_i ausgewertet.

$$spin_i = 360^\circ * spinFunction.evaluateFloat(x_i)$$

slopeFunction *FloatToFloat (Functions/Arrange/slope)*

Definiert einen Neigungswinkel um die Y-Achse des Objektes, um den die vervielfältigten Objekte geneigt werden. Um den Winkel $slope_i$ zu bestimmen, um den das

i -te Objekt gedreht werden soll, wird die *slopeFunction*-Funktion an der Stelle x_i ausgewertet.

$$\text{slope}_i = 360^\circ * \text{slopeFunction.evaluateFloat}(x_i)$$

densityMin *Float* (0, 0, 100)

Legt die Minstdichte fest, bei der eine Instanz noch erzeugt werden kann.

densityMax *Float* (100, 0, 100)

Legt die Maximaldichte fest, bei der eine Instanz erzeugt wird.

useDensityI *CheckBox* (*true*)

Mit diesem Feld wird festgelegt, ob zwischen der Mindest- und Maximaldichte Normalverteilt werden soll. Ist das Feld nicht aktiviert, werden die Originalwerte aus dem Dichtefeld verwendet.

Die Normalverteilung erfolgt dabei mit Betonung auf den optimalen Dichtewert *densityOpt*, der sich aus den Grenzwerten *densityMin* und *densityMax* folgendermaßen berechnet:

$$\text{densityOpt} = \text{densityMin} + \text{abs}(\text{densityMax} - \text{densityMin})/2$$

Die Standardabweichung σ beträgt 10.

densityI *Float* (0, 0, 1)

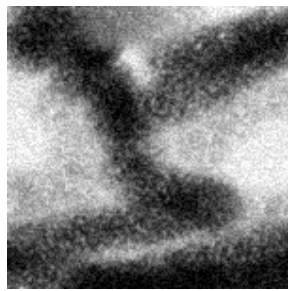
Bestimmt, wenn mittels *useDensityI* aktiviert, wie stark die Normalverteilung zwischen der Mindest- und Maximaldichte ist. Bei Null wird keine Normalverteilung vorgenommen. Die Wahrscheinlichkeit für einen Punkt der Grundfläche, dessen Wert innerhalb der Intervallgrenzen liegt, mit einem Objekt belegt zu werden, wenn dieser ausgewählt wird, liegt bei 100%. Steht *densityI* auf Eins, so wird mittelwertbetont interpoliert.

density *ChannelMap* (*Graytone*)

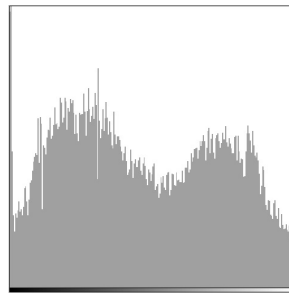
Erlaubt die Definition eines Dichtefeldes oder einer Verteilungsmaske, die über die Grundfläche gelegt wird. Dies ermöglicht die Positionierung von Objekten entsprechend dieser Vorgabe.

Die Berechnung des endgültigen Dichtefeldes geschieht in mehreren Schritten. Nachdem der Benutzer ein Graustufenbild geöffnet oder erzeugt hat (Abbildung 5.18a), wird es zuerst in eine Matrix geladen und die Farbwerte in Prozente umgerechnet. Lädt der Benutzer an dieser Stelle ein Farbbild, so wird nur der Blaukanal beachtet.

Im zweiten Schritt wird das Ausgangsbild gemäß dem Minstdichtewert *densityMin* und Maximaldichtewert *densityMax* beschnitten, d. h. alle Werte, die nicht zwischen dem Minimal- und Maximalwert liegen, werden entfernt, d. h. ihre Wahrscheinlichkeit, mit einem Objekt belegt zu werden, wenn ausgewählt, wird auf Null gesetzt.



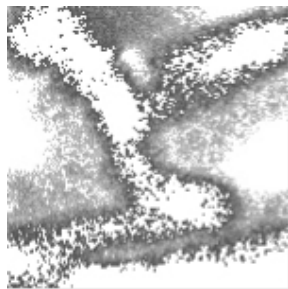
(a) Ausgangsbild



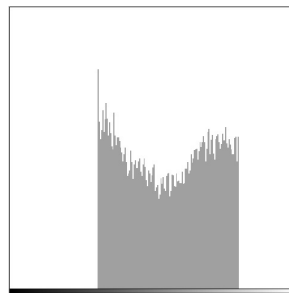
(b) dazugehöriges Histogramm

Abbildung 5.18: Berechnung Dichtefeld: Ausgangssituation

In diesem Beispiel ist $densityMin = 30$ und $densityMax = 80$, entsprechend der Farbwerte 77 als Minimalwert und einem Maximalwert von 205. Das Resultat ist in Abbildung 5.19a zu sehen.



(a) beschnittenes Ausgangsbild



(b) dazugehöriges Histogramm

Abbildung 5.19: Berechnung Dichtefeld: Clipping-Zwischenschritt

Das Dichtefeld zwischen 30% und 80% ist somit definiert, die erzeugten Instanzen werden nur innerhalb der Grenzen positioniert.

Entscheidend ist die Wahrscheinlichkeit, mit der ein Punkt der Grundfläche innerhalb der Intervallgrenzen belegt wird, wenn er ausgewählt wird. Die dicke schwarze Linie in den Bildern der Abbildung 5.20 gibt die Wahrscheinlichkeiten bei unterschiedlichen Einstellungen an. Sie liegt zwischen Null und Eins, entsprechend 0 und 100 Prozent.

Ist $useDensityI$ nicht aktiviert, wird die Wahrscheinlichkeit eines Punktes p_i gleich seinem prozentualen Farbwert gesetzt. $W_{p_i} = Farbe(p_i)/MAX_FARBWERT$, s. Abbildung 5.20a. Ist $useDensityI$ aktiviert, so hängt die Wahrscheinlichkeit von der $densityI$ -Einstellung ab. Die Positionierung erfolgt an jeder Stelle innerhalb der Grenzen mit einer Wahrscheinlichkeit von 100% (5.20b), wenn $densityI$ auf Null steht. Es wird also nicht interpoliert, und somit liegt eine Art Schwarz-Weiß-Sicht vor, in der ein Objekt entweder auf einen Punkt platziert werden darf oder nicht.

Steht *densityI* hingegen auf Eins, wird mittelwertbetont interpoliert, Abbildung 5.20c. Bei allen Werten dazwischen wird zu *densityI* * 100 Prozent interpoliert. Der Mittelwert, der das Optimum *densityOpt* bestimmt, ergibt sich wie folgt:

$$densityOpt = densityMin + abs(densityMax - densityMin)/2$$

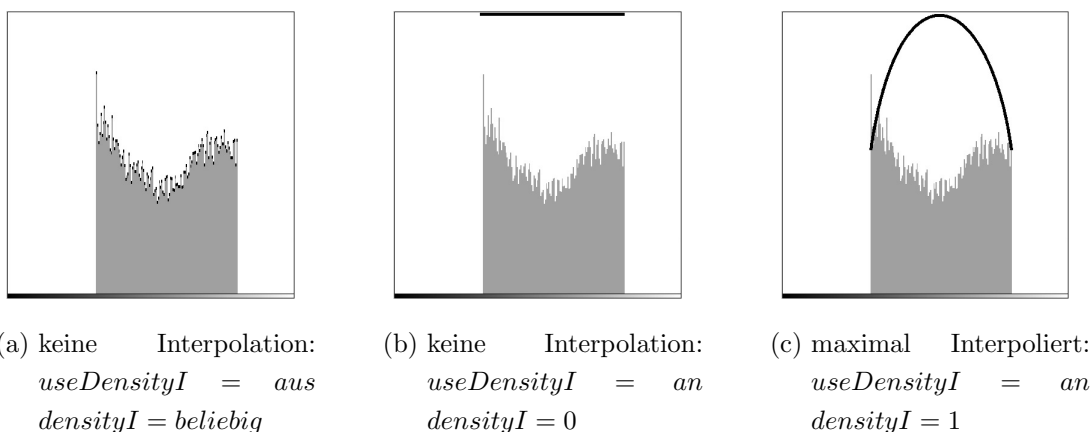
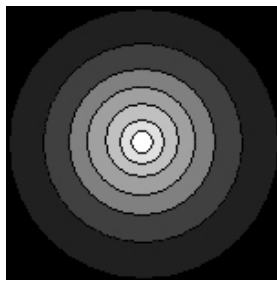


Abbildung 5.20: Berechnung Dichtefeld: Wahrscheinlichkeit einer Belegung

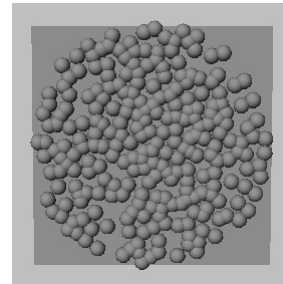
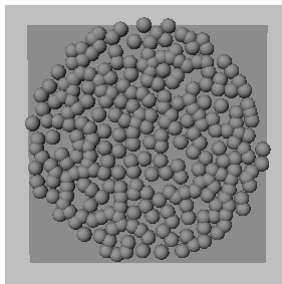
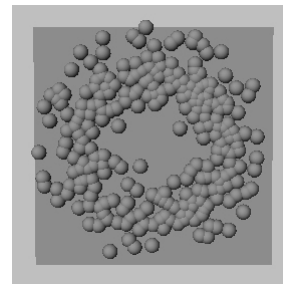
Die Auswirkung auf die Verteilung von Objekten ist in Abbildung 5.21 deutlich zu erkennen. Die zufällige Verteilung von 250 Kugelobjekten auf einer Grundfläche, der die in Abbildung 5.21a dargestellte Dichteverteilung zu Grunde liegt, führt bei einer Dichtebegrenzung von *densityMin* = 20 und *densityMax* = 100 Prozent und der *densityI* = 0 Einstellung zu der in Abbildung 5.21c gezeigten Anordnung. Wird das *densityI*-Attribut auf 1 gesetzt, wird zwischen *densityMin* und *densityMax* eine mittelwertbetonte Verteilung vorgenommen, also mit dem Schwerpunkt auf 60, was zu den ausgefransten Rändern innen und außen und der erhöhten Objektkonzentration dazwischen in Abbildung 5.21d führt. Ist *useDensityI* nicht aktiviert, ergibt sich die Abbildung 5.21b, in der eine erhöhte Objektkonzentration um den Mittelpunkt herum erkennbar ist.

Eine einfaches Anwendungsbeispiel aus der Natur ist die Beeinflussung der Pflanzenausbreitung entsprechend der Bodenfeuchtigkeit. Weiß man über eine Pflanzenart, welche Mindest- und Maximalfeuchtigkeit sie für ihre Entwicklung benötigt, so können diese Feuchtigkeitswerte als Grenzen gesetzt werden. Mit einem entsprechenden Feuchtigkeitsbild einer Grundfläche ist es nun möglich, die erzeugten Instanzen nur in den für diese Pflanzenart günstigen Bereichen zu positionieren.

Der Wahrscheinlichkeitswert liegt für jeden Punkt der Grundfläche vor und wird an die ggf. auf diesen Punkt platzierte Instanz weitergegeben.



(a) Ausgangsbild

(b) Verteilung wenn *useDensityI* = *aus*(c) Verteilung bei *useDensityI* = *an* und *densityI* = 0(d) Verteilung bei *useDensityI* = *an* und *densityI* = 1Abbildung 5.21: Wirkung des *densityI*-Attributs auf die Objektpositionierung

locationParameterMin *Color3f* (0, 0, 0; 0, 0, 0; 255, 255, 255)

Bestimmt die Minimalwerte der drei LocationParameter-Werte. Der Rotkanal steht für den ersten, der Grünkanal für den zweiten und der Blaukanal für den dritten LocationParameter-Wert.

locationParameterMax *Color3f* (255, 255, 255; 0, 0, 0; 255, 255, 255)

Bestimmt die Maximalwerte der drei LocationParameter-Felder. Der Rotkanal steht für den ersten, der Grünkanal für den zweiten und der Blaukanal für den dritten LocationParameter-Wert.

locationParameter1 Function *FloatToFloat* (*Id*)

Die hiermit definierte Funktion wird auf die *LocationParameter1*-Werte angewendet.

locationParameter2 Function *FloatToFloat* (*Id*)

Die hiermit definierte Funktion wird auf die *LocationParameter2*-Werte angewendet.

locationParameter3 Function *FloatToFloat* (*Id*)

Die hiermit definierte Funktion wird auf die *LocationParameter3*-Werte angewendet.

locationParameter *ChannelMap (Graytone)*

Dieses Attribut definiert die Grundlage der drei internen LocationParameter-Variablen, der Rotkanal wird als *LocationParameter1*, der Grünkanal als *LocationParameter2* und der Blaukanal als *LocationParameter3* interpretiert. Die LocationParameter-Werte liegen für jeden Punkt der Grundfläche vor und werden an die ggf. auf diesen Punkt platzierte Instanz weitergegeben, s. Kapitel 5.2, dabei hat das Feld keine Auswirkung auf die Platzierung der Instanzen.

Mit dieser Technik können Standortparameter wie der Nährstoff- oder Wassergehalt im Boden an die erzeugten Instanzen übertragen werden und so diese beeinflussen. Da die LocationParameter-Werte in jedem Baustein zur Verfügung stehen und auch von jedem weitergegeben werden, können so Stoffflüsse simuliert werden.

lod *LODFeld*

Die implementierten Methoden werden im Unterkapitel 6.2 beschrieben.

5.12.1 Geometrische Anordnungen

In dieser Einstellung werden die Objekte nach einfachen geometrischen Anordnungsverfahren verteilt.

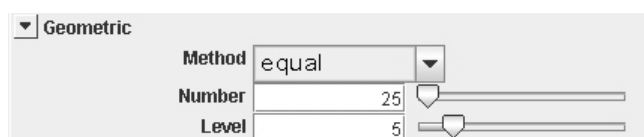


Abbildung 5.22: Attributfeld der geometrischen Anordnungen

Der Benutzer wählt die Methode aus, legt die Anzahl der vervielfältigten Objekte fest und kann so sehr komfortabel zum Beispiel ein Feld einer Plantage modellieren.

Ist ein Dichtefeld definiert, so werden die vervielfältigten Objekte erst so platziert, als wäre kein Dichtefeld vorhanden, im Anschluss werden die einzelnen Objektpositionen mit dem entsprechenden Positionen im Dichtefeld überprüft, ob ein Objekt an dieser Position zulässig ist. Ist dies nicht der Fall, so wird das Objekt an dieser Stelle nicht erzeugt. So ist der Effekt zu erklären, dass man weniger Instanzen erzeugt bekommt, als im *Number*-Attribut angegeben ist.

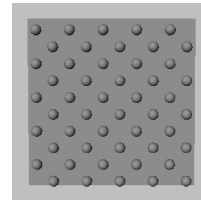
Attribute:

Method *Auswahlfeld (equal)*

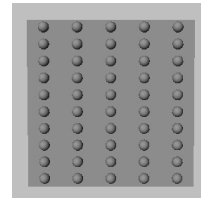
Bestimmt die Anordnungsmethode, nach der die Objekte verteilt werden. Die nachfolgenden Beispiele wurden alle mit *Number* = 50 und *Level* = 5 erzeugt.

equal

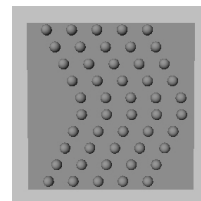
Ordnet die Objekte versetzt an, so dass sie die Fläche gleichmäßig füllen. Über *Level* wird die Anzahl der "Doppellinien", bei der jeder zweite Punkt versetzt ist, festgelegt.

**line**

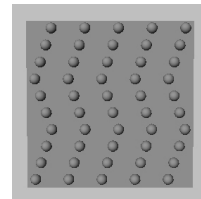
Ordnet die Objekte in parallelen Linien an. *Level* bestimmt die Anzahl der Linien.

**triangle**

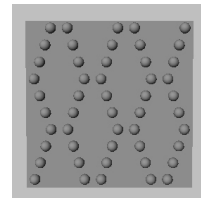
Ordnet die Objekte in parallelen Linien an, wobei die Linie in der Mitte geknickt ist. Die Anzahl der Linien wird mittels *Level* festgelegt.

**staggering**

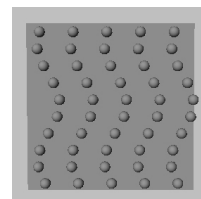
Ordnet die Objekte in parallelen *Level*-vielen Zickzacklinien an.

**staggering 2**

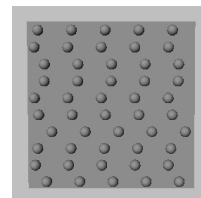
Ordnet die Objekte in Zickzacklinien an, wobei jede zweite entgegengesetzt ist. Das *Level*-Attribut bestimmt die Anzahl der Linien.

**wave**

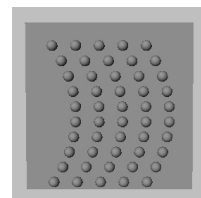
Ordnet die Objekte in parallelen Wellenlinien an, wobei die Anzahl der Linien durch das *Level*-Attribut festgelegt wird.

**double wave**

Ordnet die Objekte in parallelen Wellenlinien an, wobei diesmal der absolute Betrag der Sinusfunktion addiert wird. Durch das *Level*-Attribut wird die Anzahl der Linien definiert.

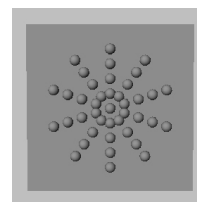
**parabola**

Ordnet die Objekte in parallelen großen Bögen an. Die Anzahl der Linien wird durch das *Level*-Attribut angegeben.

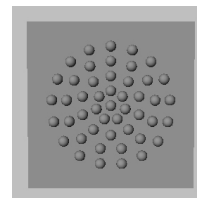


circle

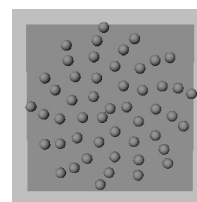
Ordnet die Objekte in konzentrischen Kreisen mit jeweils gleicher Objektzahl pro Kreis an. Die Anzahl der Kreise wird durch das *Level*-Attribut angegeben.

**circle2**

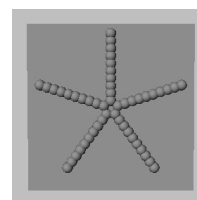
Ordnet die Objekte in konzentrischen Kreisen mit jeweils proportional zunehmender Objektzahl pro Kreis an. Mit *Level* wird die Anzahl der Kreise definiert.

**phyllotaxis**

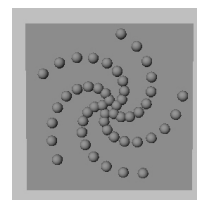
Ordnet die Objekte entsprechend der Verteilung des Goldenen Schnittes an, s. Kapitel 3.1.2. Mit *Level* kann der Winkel verändert werden.

**ray**

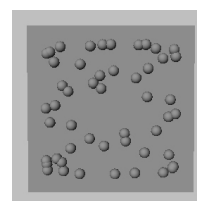
Ordnet die Objekte ausgehend vom Mittelpunkt in *Level*-vielen Strahlen an.

**spiral**

Ordnet die Objekte ausgehend vom Mittelpunkt in einer Spirale an, wobei die Anzahl der Spiralarme über das *Level*-Attribut bestimmt werden kann.

**lissajou**

Ordnet die Objekte in Lissajous-Figuren an, mit *Level* wird die Figur geändert.



Number *Integer* (25, 2, 500)

Legt die Anzahl der vervielfältigten Instanzen fest.

Level *Integer* (5, 1, 25)

Wirkt sich wie in den Methoden angegeben aus.

5.12.2 Wahrscheinlichkeitsanordnungen

Diese Einstellung ermöglicht die Verteilung von Objekten entsprechend einer Wahrscheinlichkeitsverteilung, wobei aus einer umfangreichen Liste an Wahrscheinlichkeitsfunktionen gewählt werden kann.

Zuerst einmal ein kleiner Ausflug in die Statistik. Die Aufgabe besteht darin, herauszufinden, welcher Verteilung die Anordnung von bestimmten Objekten auf einem bestimmten Gebiet entspricht. In Abbildung 5.23 werden die Objekte durch rote Punkte repräsentiert. Biologen zum Beispiel stehen häufig vor diese Aufgabe, wenn sie die Verteilung von speziellen Baumarten, das Auftreten von Krankheitsherden oder die Anordnung von Vogelgelegen untersuchen. Vereinfachend wird im Folgenden davon ausgegangen, dass es sich um eine ebene, rechteckige Grundfläche handelt. Diese wird zuerst in eine große Zahl von Segmenten, etwa Quadrate oder Rechtecke gleicher Fläche, unterteilt. Man zählt anschließend die Anzahl der betrachteten Objekte pro Segment. Die empirischen gewonnenen Daten können wie in Tabelle 5.1 dargestellt werden.

Tabelle 5.1: Tabellarisch erfasste Daten

Objekte pro Segment	Anzahl Segmente	Objekte
0	8	0
1	13	13
2	9	18
3	5	15
4	1	4
Σ	36	50

In diesem Fall werden die Segmente null- bis viermal von Objekten belegt, die Tabelle hat demzufolge fünf Zeilen. Eine Zeile beschreibt in der ersten Spalte das Ereignis bzw. das untersuchte Merkmal, die zweite Spalte gibt die Häufigkeit, wie oft das Ereignis auftrat, an und in der dritten Spalte steht die Gesamtzahl der Objekte, die diesem Ereignis entsprechend zugeordnet werden.

Um eine Unabhängigkeit von der Anzahl der Objekte zu erreichen, werden für die Visualisierung der empirischen Daten die absoluten Häufigkeiten normiert, wobei zu beachten ist:

- Die relativen Häufigkeiten h_i sind wie folgt definiert:

$$h_i = \frac{\text{absolute Häufigkeit}}{\text{Zahl der Beobachtungen}} = \frac{n_i}{n}$$

- Die Summe der absoluten Häufigkeiten ist gleich der Zahl der Beobachtungen,

$$\sum_{i=1}^k n_i = n \quad (\text{i=Laufindex, k=Grenze})$$

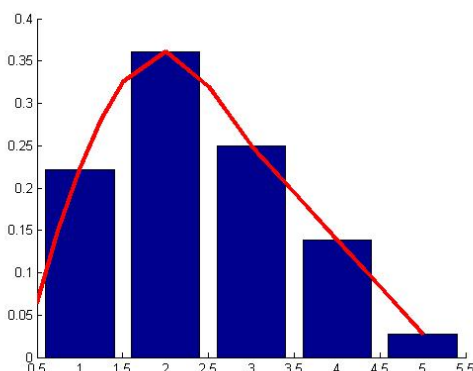
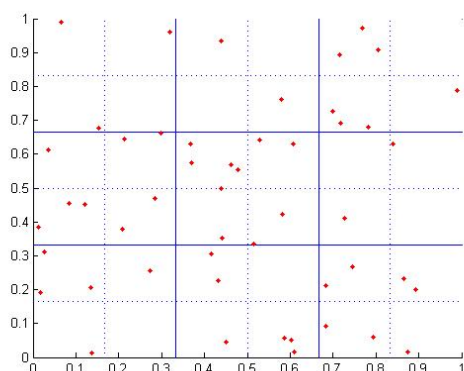


Abbildung 5.23: Zugrunde liegende Beispielspielverteilung

Abbildung 5.24: Auswertung der Verteilung

- Die Summe der relativen Häufigkeiten ist 1,

$$\sum_{i=1}^k h_i = \frac{1}{n} \sum_{i=1}^k n_i = 1$$

Nun ist es möglich, die normierten Häufigkeiten in einem Diagramm darzustellen, z. B. in einem Balkendiagramm, oder die Werte zu interpolieren, beides in Abbildung 5.24 dargestellt. Die interpolierte Funktion wird in einem anschließenden Schritt hinsichtlich der Wahrscheinlichkeitsverteilung, der sie näherungsweise entspricht, untersucht, womit die Aufgabe gelöst ist.

Bei der Berechnung der Wahrscheinlichkeitsanordnungen wird hier der entgegengesetzte Weg gegangen. Zuerst wird vom Benutzer eine Wahrscheinlichkeitsfunktion ausgewählt, danach kann mit dem *Level*-Attribut die Anzahl der Segmente bestimmt werden. Für

Abbildung 5.25: Attributfeld der Wahrscheinlichkeitsanordnungen

jedes Segment wird anschließend eine Zufallszahl z_i entsprechend der selektierten Wahrscheinlichkeitsfunktion ermittelt. Die so gewonnene Zufallszahl wird gleichgesetzt mit der Anzahl der Objekte pro Segment. Die z_i Objekte werden zufällig auf dem Segment verteilt unter der Beachtung des *minRadius*-Attribut, mit dem ein Mindestabstand zwischen den erzeugten Instanzen festgelegt wird.

Attribute:

Method *Auswahlfeld (normal)*

Legt die Wahrscheinlichkeitsfunktion fest, nach der die Objekte verteilt werden. In Tabelle 5.3 sind die verfügbaren Wahrscheinlichkeitsfunktionen aufgelistet.

Level *Integer (5, 1, 15)*

Bestimmt die Anzahl der Unterteilungen der Grundfläche. Bei $Level = 1$ wird die Grundfläche einmal entlang der X- und einmal entlang der Y-Achse unterteilt, s. Abbildung 5.26a. Level gibt damit die Anzahl der Schnitte pro Achse an. Die Anzahl der Segmente ergibt sich entsprechend aus:

$$anzSegmente = (Level + 1) * (Level + 1) = Level^2 + 2 * Level + 1$$

minRadius *Float (0.5, 0, 5)*

Definiert einen Mindestabstand zwischen den erzeugten Instanzen, der nicht unterschritten werden darf.

Wird mehrfach (10 mal) vergebens versucht, ein Objekt zu platzieren, wobei es daran scheitert, dass in jeder Umgebung der neu gewählten Position sich bereits mindestens ein Objekt befindet, das Feld somit recht dicht besetzt ist, wird der Mindestradius in 5%-Schritten verringert, bis eine Positionierung möglich ist, spätestens bei $minRadius = 0$.

hierarchic *CheckBox (false)*

Dieses Attribut beeinflusst das Verfahren, wie mit der selektierten Methode die Objekte verteilt werden. In der Initialeinstellung *false* und mit $Level = 1$ wird die Grundfläche wie in Abbildung 5.26a zu sehen aufgeteilt, ist das *hierarchic*-Feld hingegen aktiviert, wird die Grundfläche im ersten Schritt mit $Level + 1$ -vielen Schnitten pro Achse unterteilt und danach werden die entstandenen Segmente abermals unterteilt, nur diesmal mit $Level$ -vielen Schnitten. Auf diese Weise wird eine Hierarchie erzeugt, bei der die Segmente der Unterebene einen um eins niedrigere Level-Unterteilung haben.

$$\begin{aligned} anzSegmente &= (Level + 2) * (Level + 2) * (Level + 1) * (Level + 1) \\ &= (Level^2 + 4 * Level + 4) * (Level^2 + 2 * Level + 1) \\ &= (Level^4 + 6 * Level^3 + 13 * Level^2 + 12 * Level + 4) \end{aligned}$$

Somit wird die Grundfläche, wie in Abbildung 5.26b dargestellt, in 36 Segmente unterteilt.

Die Liste der Wahrscheinlichkeitsfunktionen in Tabelle 5.3 und deren Implementierung stammen vom *Institute for Computing Systems Architecture School of Informatics, University of Edinburgh* aus dem *eduni.distributions*-Paket. Die Zusammenstellung ist sehr

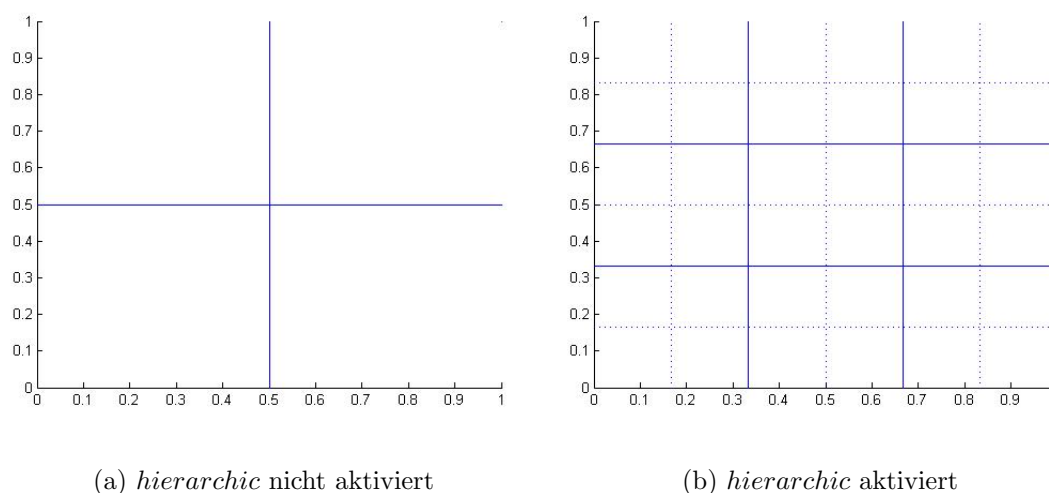


Abbildung 5.26: Unterteilung der Grundfläche bei verschiedenen *hierarchic*-Einstellungen im Arrange-Baustein, *Level* = 1

umfangreich, wobei keine Auswahl hinsichtlich ihrer Nützlichkeit bei der Modellierung getroffen wurde. Die für die Modellierung von Ökosystemen oft benutzten Normal-, Poisson- oder Binomialverteilung sind selbstverständlich enthalten.

5.12.3 Halbtonverfahren

Die Anordnungsverfahren, die hier zusammengefasst werden, arbeiten alle auf einem vorgegebenen Dichtefeld, das die Positionen der vervielfältigten Instanzen bestimmt. Halbtonverfahren, auch Rasterungs-Algorithmen genannt, stammen aus der digitalen Bildverarbeitung, wo mit ihrer Hilfe die Anzahl von Farben/Grautönen in einem Bild herabgesetzt wird, ohne allzu große Qualitätseinbußen hinnehmen zu müssen. Analog werden diese Verfahren bei der Ausgabe von Bildern auf Geräten mit niedrigerer Farb-/Grauwerttiefe angewandt, so zum Beispiel auf Monochrom-Monitoren, Druckern, aber auch zur Reduktion des Speicherbedarfs bei der Archivierung von Bildern.

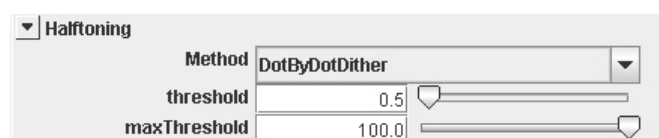


Abbildung 5.27: Attributfeld der Halbtonverfahren

Das der Berechnung zugrunde liegende Graustufenbild ist das über das *density*-Attribut definierte Dichtefeld. Es wird, entsprechend dem vom Benutzer ausgewählten Verfahren, in ein Halbtonbild (Binärbild) überführt, dessen weiße Pixel die Positionen der neuen

Tabelle 5.3: Übersicht der bereitgestellten Wahrscheinlichkeitsfunktionen

Name	Parameter	Algorithmus
Bernoulli	prob	<i>return random <= prob?1 : 0;</i>
Beta	shape_a, shape_b	(s. Implementierung)
Betaprime	shape_a, shape_b	<i>return 1/(beta(shape_a, shape_b)) - 1;</i>
Binomial	prob, trials	<i>return $\sum_{i=0}^{trials} \text{bernoulli}(prob)$;</i>
Cauchy	median, scale	<i>return median + scale/tan($\pi * random$);</i>
Chisquare	deg_freedom	<i>return $\sum_{i=0}^{deg_freedom} \text{normal2}(0, 1)$;</i>
Erlang	scale, shape	<i>product = $\prod_{i=0}^{shape} random$;</i> <i>return - scale * log(product);</i>
F	num_deg_freedom, den_deg_freedom	<i>return (chisquare(num_deg_freedom)/ num_deg_freedom)/ (chisquare(den_deg_freedom)/ den_deg_freedom);</i>
Gamma	scale, shape	(s. Implementierung)
Geometric	prob	<i>return (long)ceil(log(random)/log(1 - prob));</i>
Invgamma	scale, shape	<i>return 1/gamma(scale, shape);</i>
Logistic	location, scale	<i>return location - scale * log((1/random) - 1);</i>
Lognormal	mean, variance	<i>return lognormal2(mean, $\sqrt{variance}$);</i>
Lognormal2	mean, variance	<i>return exp(mean + std_dev * normal2(0, 1));</i>
Negexp	mean	<i>return - mean * log(random);</i>
Normal	mean, variance	<i>return normal2(mean, $\sqrt{variance}$);</i>
Normal2	mean, std_dev	<i>return mean + std_dev * cos(2 * π * random) * $\sqrt{-2 * \log(random)}$;</i>
Pareto	scale, shape	<i>return scale/pow(random, 1/shape);</i>
Pascal	prob, successes	<i>return $\sum_{i=0}^{successes} \text{geometric}(prob)$;</i>
Poisson	mean, deg_freedom	<i>long x = -1;</i> <i>double m = exp(-mean), product = 1;</i> <i>do {</i> <i> x ++;</i> <i> product *= random;</i> <i>} while(m < product);</i> <i>return x;</i>
Uniform	min, max	<i>return (max - min) * random + min;</i>
Weibull	scale, shape	<i>return scale * pow(log(random), 1/shape);</i>

Instanzen bestimmen. Daraus folgt, dass bei diesen Verfahren die Anzahl der zu erzeugenden Instanzen nicht angegeben werden kann. Sie kann nur nachherig im *Number*-Feld überprüft werden. Die einzige Methode, die Anzahl zu beeinflussen, liegt in der Variation der Schwellenattribute (*threshold* und *maxThreshold*).

Einteilung der implementierten Verfahren:

Threshold-Diffusion - Schwellenwertverfahren Das Threshold-Verfahren weist jedem Grauwert eines Pixels entweder schwarz oder weiß zu. Dabei bedient es sich eines fixen Schwellenwertes. Ist die Graustufe kleiner als der Schwellenwert, so wird der Punkt schwarz gefärbt, ansonsten weiß.

Vertreter: DotByDotDither, RandomDither und AverageDither

Error-Diffusion - Fehlerverteilungsverfahren Durch das Setzen eines Bildpunktes auf schwarz oder weiß beim konstanten und festen Schwellwert wird pro Bildpunkt ein Fehler bezüglich des Grauwertes gemacht. Berücksichtigt man bei der Erzeugung eines Binärbildes diesen Fehler $F_{xy} = Alt_{xy} - Neu_{xy}$ und verteilt ihn auf die Nachbarpunkte, so kann man diesen Fehler kompensieren.

Vertreter: HilbertDither, FloydSteinberg, Stucki6Dither, Stucki12Dither und Jarvis12Dither

Ordered-Dither - Verfahren mit variablem Schwellwert Beim Ordered-Dither-Verfahren handelt es sich beim Schwellenwert nicht mehr um einen einzelnen Zahlenwert, sondern um eine Matrix. Der Schwellwert für jeden Bildpunkt wird dann durch den Wert an der korrespondierenden Stelle in der Matrix bestimmt.

Vertreter: Bayer, Spiral, Classical und Line screen

Dieses Positionierungsverfahren ist vor allem dann von Nutzen, wenn die natürliche Grundfläche durch den Eingriff des Menschen verändert wurde, wie dies in Parkanlagen, Gärten oder bei landwirtschaftlich genutzten Flächen der Fall ist, und sich Gebiete mit verschiedenen Platzierungsdichten ergeben.

Attribute:

Method *Auswahlfeld (DotByDotDither)*

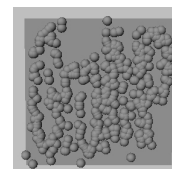
Legt das Halbtonverfahren fest, wobei die eingebauten Algorithmen eine Auswahl von gängigen Rasterungsverfahren widerspiegeln. Nachfolgend werden sie kurz erklärt. Tiefere Einblicke in die verschiedenen Verfahren bieten die Bücher von R. Ulichney [55], H. R. Kang [28] und D. L. Lau / G. R. Arce [35].

Im Folgenden entspricht schwarz dem Wert 0 und weiß *maxThreshold*, der Schwellenwert ist über *threshold* definiert. Das jeweilige Beispielbild ist in unterschiedlichen Einstellungen auf dieser Grundlage entstanden:



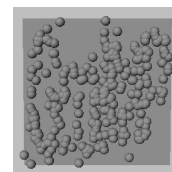
DotByDotDither

Die einfachste Methode zur Überführung eines Grautonbildes in ein Halbtonbild. Das Verfahren weist jedem Grauwert eines Pixels entweder schwarz oder weiß zu. Dabei bedient es sich eines konstanten Schwellenwertes *threshold*. Ist die Graustufe kleiner als der Schwellenwert, so wird der Punkt schwarz gefärbt, ansonsten weiß. Alle Punkte, die am Ende den Wert *maxThreshold* haben, werden als Positionen interpretiert.



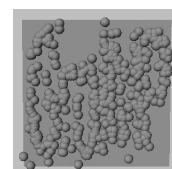
RandomDither

Dieser Algorithmus verwendet den Grauwert eines Bildpunktes als Wahrscheinlichkeit, dass dieser Bildpunkt im gerasterten Bild weiß erscheint. Weiße Pixel ergeben also mit 100% Wahrscheinlichkeit einen weißen Punkt, ein 50%-Grau ergibt zu 50% einen weißen Rasterpunkt. Es wird somit jeder Bildpunkt mit einem Zufallswert zwischen null und *threshold* als Schwellenwert verglichen.



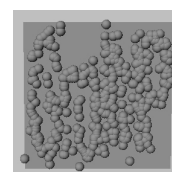
AverageDither

Das Verfahren arbeitet analog dem DotByDotDither, nur dass der Schwellenwert der Durchschnittswert aller Bildpunkte ist. Das *threshold*-Attribut wird nicht beachtet.



HilbertDither

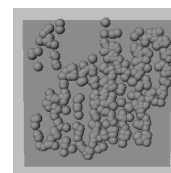
Das auf der Idee von Peano und D. Hilbert beruhende Verfahren wurde zuerst 1981 beschrieben, s. [2]. Es arbeitet ebenfalls bildpunktweise und vergleicht jeden Bildpunkt mit dem Schwellenwert. Der gemachte Fehler wird auf den nächsten Bildpunkt übertragen. Entscheidend ist, wie die Reihenfolge, in der die Punkte bearbeitet werden, bestimmt wird. Dazu wird über das Bild die sog. Hilbert-Kurve gelegt. Sie verläuft über das gesamte Bild und berührt jeden Bildpunkt genau einmal. Ihr scheinbar zufällig verlaufender Pfad sorgt für eine Fehlerverteilung, die ein harmonisches Resultat erzeugt, in dem kaum störende Muster auftreten.



FloydSteinberg

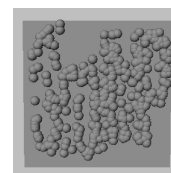
Jeder Bildpunkt wird wie im Schwellenwertverfahren gemäß einem Schwellenwert schwarz oder weiß gerastert. Der Fehler, der dabei gemacht wird, wird auf die umliegenden, noch nicht gerasterten Punkte verteilt. Die Verteilung bestimmt eine Matrix, die sog. Threshold-Matrix.

$$\text{Matrix:} \begin{pmatrix} 7 & 3 \\ 5 & 1 \end{pmatrix} / 16$$

**Stucki6Dither**

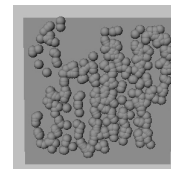
Arbeitet analog dem FloydSteinberg-Verfahren, nur mit der folgenden Threshold-Matrix.

$$\text{Matrix:} \begin{pmatrix} 8 & 2 & 2 \\ 8 & 2 & 2 \end{pmatrix} / 24$$

**Stucki12Dither**

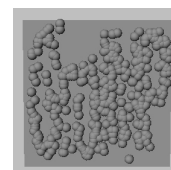
Arbeitet analog dem FloydSteinberg-Verfahren, nur mit der folgenden Threshold-Matrix.

$$\text{Matrix:} \begin{pmatrix} 8 & 4 & 2 & 4 \\ 8 & 4 & 2 & 1 \\ 2 & 4 & 2 & 1 \end{pmatrix} / 42$$

**Jarvis12Dither**

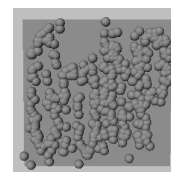
Arbeitet analog dem FloydSteinberg-Verfahren, nur mit der folgenden Threshold-Matrix.

$$\text{Matrix:} \begin{pmatrix} 7 & 5 & 3 & 5 \\ 7 & 5 & 3 & 1 \\ 3 & 5 & 3 & 1 \end{pmatrix} / 48$$

**Ordered-Dither Bayer 4x4**

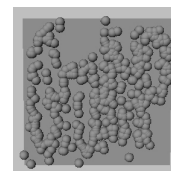
Das Bayer-Dither-Verfahren stammt von B. E. Bayer von 1973 [5]. Es beruht auf einer Threshold-Matrix, die für jeden Rasterpunkt den Grauwert verwaltet, ab dem dieser Punkt weiß dargestellt wird.

$$\text{Matrix:} \begin{pmatrix} 1 & 9 & 3 & 11 \\ 13 & 5 & 15 & 7 \\ 4 & 12 & 2 & 10 \\ 16 & 8 & 14 & 6 \end{pmatrix} / 16$$

**Ordered-Dither Spiral 4x4**

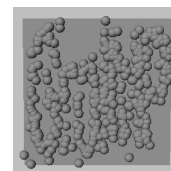
Arbeitet analog dem BayerDither, nur dass die Threshold-Matrix die Fehler spiralförmig verteilt.

$$\text{Matrix:} \begin{pmatrix} 10 & 9 & 8 & 7 \\ 11 & 16 & 15 & 6 \\ 12 & 13 & 14 & 5 \\ 1 & 2 & 3 & 4 \end{pmatrix} / 16$$

**Ordered-Dither Classical 4x4**

Arbeitet analog dem BayerDither, nur dass die Threshold-Matrix die Fehler in einer großen Spirale verteilt.

$$\text{Matrix:} \begin{pmatrix} 4 & 8 & 10 & 1 \\ 11 & 15 & 14 & 5 \\ 7 & 16 & 13 & 9 \\ 3 & 12 & 6 & 2 \end{pmatrix} / 16$$

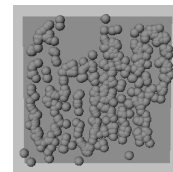


Ordered-Dither Line screen 4x4

Arbeitet analog dem BayerDither, nur dass die Threshold-Matrix die Fehler linienbetont verteilt.

Matrix:

$$\begin{pmatrix} 5 & 7 & 8 & 6 \\ 13 & 15 & 16 & 14 \\ 9 & 11 & 12 & 10 \\ 1 & 3 & 4 & 2 \end{pmatrix} / 16$$



threshold *Float* (0.5, 0, 15)

Legt für die Schwellenwertverfahren den konkreten Schwellenwert fest.

maxThreshold *Float* (100, 0, 100)

Definiert den maximalen Farbwert. Punkte, die diesen Wert haben, werden als Position interpretiert.

5.12.4 Zusätzliche Verfahren

Bei den Verfahren in diesem Abschnitt handelt es sich um Verfahren, die keiner der vorangegangenen Klassen zugeordnet werden können, oder sie sind unter einem eigenständigen Namen und entsprechender Implementierung bekannt, die nicht zu den anderen Verfahren passt.

Attribute:

Method *Auswahlfeld* (*DotByDotDither*)

Bestimmt die Anordnungsmethode.

DartThrowing Dieses Verfahren erzeugt eine einfache Zufallsverteilung unter der Berücksichtigung des Mindestabstands zwischen den Instanzen. So wird einfach und effizient eine Poisson-disk-Verteilung simuliert. Der Algorithmus arbeitet aber nur solange effizient, wie die Objektdichte gering ist. Stehen nur noch wenige Positionen für weitere Punkte zur Verfügung, so sinkt die Effizienz des Algorithmus rapide, da das Suchen nach freien Punkten eine zunehmend höhere Anzahl an Versuchen benötigt.

Attribute:

Number *Integer* (25, 1, 500)

Legt die Anzahl der vervielfältigten Instanzen fest.

minRadius *Float* (0.5, 0, 10)

Definiert einen Mindestabstand zwischen den erzeugten Instanzen, der nicht unterschritten werden darf.

Wird mehrfach (10 mal) vergebens versucht, ein Objekt zu platzieren, wobei es daran scheitert, dass in jeder Umgebung der neu gewählten Position sich bereits mindestens ein Objekt befindet, das Feld somit recht dicht

besetzt ist, wird der Mindestradius in 5%-Schritten verringert, bis eine Positionierung möglich ist, spätestens bei $minRadius = 0$.

Voronoi Lloyd In dieser Einstellung werden Objekte nicht entsprechend des im *density*-Attribut definierten Dichtefeldes verteilt. Die hier erzeugte Verteilung ist das Resultat eines Optimierungsprozess, bei dem punktförmige Objekte aus einer zufälligen oder regelmäßigen Verteilung über Relaxierung auf der Basis von Voronoi-Diagrammen in ein so genanntes Schwerpunkt-Voronoi-Diagramm (zentroidale Voronoi-Tesselierung) überführt werden. Die Anordnung der Punktobjekte entspricht dann einer Poisson-Disc-Verteilung [15, 22].

Ein Voronoi-Diagramm zerlegt eine gegebene Grundfläche bei ebenfalls gegebenen n Punkten in n Regionen. Jedem dieser Punkte – genannt Voronoi-Punkte – ist genau eine Region zugeordnet. Eine Region beschreibt die Menge an Punkten um einen Voronoi-Punkt, für die gilt, dass sie ihrem Voronoi-Punkt näher sind als allen anderen Voronoi-Punkten. Schwerpunkt-Voronoi-Diagramme sind Voronoi-Diagramme, bei denen die Voronoi-Punkte gleichzeitig Schwerpunkt ihrer Region sind.

Allgemein gesehen stellt sich das Problem als Optimierungsaufgabe, bei der eine Menge von Punkten p_1, \dots, p_n so über eine Fläche verteilt werden sollen, dass die Fläche in möglichst gleich große Teilflächen zerlegt wird. Die durch Optimierung zu minimierende Kostenfunktion lautet nach [11]:

$$F(p_1, \dots, p_n) = \sum_{i=1}^n \int_{v_i} f(\sqrt{\|p - p_i\|^2}) \phi(p) dx$$

Die Gewichtsfunktion $\phi(p)$ charakterisiert das lokale Gewicht einer Region und steuert so die Punktdichte dieser Region. Die einfache Kostenfunktion f berechnet den euklidischen Abstand zwischen zwei Punkten.

Als befriedigende Lösungsmöglichkeit dieser lokalen Optimierungsaufgabe erwies sich der Lloyd'sche-Algorithmus, mit dem durch die wiederholte Anwendung folgender Schritte:

1. Berechne Schwerpunkte der Voronoi-Regionen
2. Berechne Voronoi-Diagramm aus den Schwerpunkten

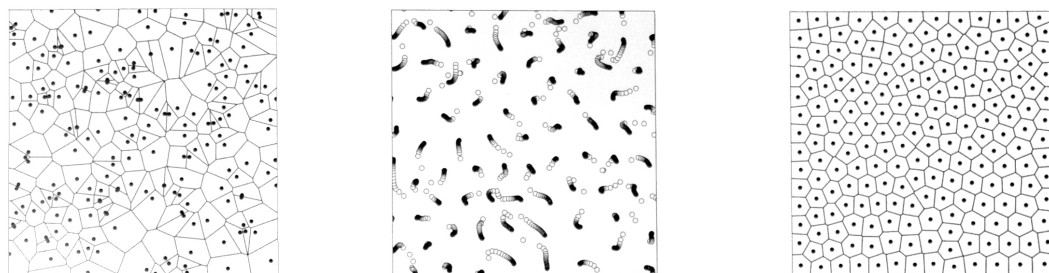
eine gute Näherung an das Schwerpunkt-Voronoi-Diagramm erreicht wird.

Eigenschaften des Lloyd-Algorithmus:

- Determinierte Berechnung
- Konvergenz gegen Schwerpunkt-Voronoi-Diagramm bei konvexer Grundmenge (eine Konvergenz des Verfahrens für den allgemeinen Fall ist analytisch noch nicht bewiesen [16].)

- Hohe Konvergenzgeschwindigkeit, niedrige Iterationsgeschwindigkeit
- Schwierige Berechnung von Voronoi-Diagramm, Polygonschnitten und Schwerpunkten

Dieser Prozess kann beliebig lang fortgesetzt werden und endet in einem hexagonalen Gittermuster. In der Praxis wird der Prozess bei Erreichen der gewünschten Verteilung abgebrochen.



(a) Ausgangspunkte mit Voronoi-Gebieten mit (b) Bewegung der Voronoi-Punkte, $densityI = 0$ (c) Endlage nach 20 Iterationen, $densityI = 1$

Abbildung 5.28: Anwendung des Lloyd-Algorithmus auf eine Punktmenge (Bilder: Deussen)

Abbildung 5.28 zeigt die Anwendung des Lloyd-Algorithmus auf eine Punktmenge mit 30 Punkten. In der Ausgangssituation 5.28a sind die Punkte zufällig verteilt, die Voronoi-Gebiete haben eine sehr unregelmäßige Form. Die Bewegung der Voronoi-Punkte in 5.28b zeigt den Optimierungsprozess, der nach 20 Iterationen in fast gleichmäßigen Fünf- und Sechsecken endet (5.28c).

Das Verfahren kann auf allgemeine Objekte erweitert werden, mehr dazu in [22]. Der hier implementierte Algorithmus geht auf [48] zurück.

Attribute:

Number *Integer* (25, 1, 500)

Legt die Anzahl der vervielfältigten Instanzen fest.

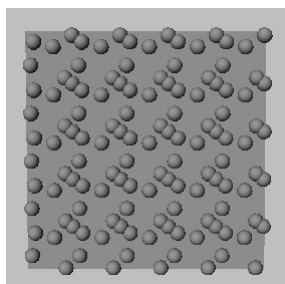
anzIterationen *Integer* (40, 2, 150)

Bestimmt die Anzahl der Optimierungsschritte.

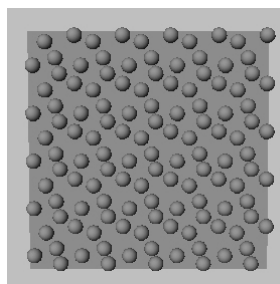
Kachelungen Mit diesem Verfahren können einfache, periodische Kachelungen erzeugt werden. Ein Kachelprototyp mit *Number*-vielen Instanzen wird im ersten Schritt erzeugt und anschließend lückenlos auf die Grundfläche übertragen. Leider ist das Auge äußerst kritisch, was das Erkennen periodischer Muster betrifft, so dass das Ergebnis sehr unharmonisch erscheint. Entsprechende Erweiterungen mit mehreren verschiedenen Kacheln, die nichtperiodisch verteilt

werden, sind in Deussen [11] zu finden.

Attribute:



(a) *checkBorders* = 0, inaktiv



(b) *checkBorders* = 1, aktiv

Abbildung 5.29: Beispielkachelung: *Number* = 5, *Level* = 4

Number *Integer* (5, 1, 500)

Legt die Anzahl der vervielfältigten Instanzen pro Kachel fest.

minRadius *Float* (0.5, 0, 10)

Definiert einen Mindestabstand zwischen den erzeugten Instanzen, der nicht unterschritten werden darf.

Wird mehrfach (10 mal) vergebens versucht, ein Objekt zu platzieren, wobei es daran scheitert, dass in jeder Umgebung der neu gewählten Position sich bereits mindestens ein Objekt befindet, das Feld somit recht dicht besetzt ist, wird der Mindestradius in 5%-Schritten verringert, bis eine Positionierung möglich ist, spätestens bei *minRadius* = 0.

Level *Integer* (1, 1, 15)

Bestimmt die Anzahl der Unterteilungen der Grundfläche. Bei *Level* = 1 wird die Grundfläche einmal entlang der X- und einmal entlang der Y-Achse unterteilt, s. Abbildung 5.26a. *Level* gibt damit die Anzahl der Schnitte pro Achse an. Die Anzahl der Segmente ergibt sich entsprechend aus:

$$\text{anzSegmente} = (\text{Level} + 1) * (\text{Level} + 1) = \text{Level}^2 + 2 * \text{Level} + 1$$

checkBorders *Integer* (*false*)

Ist dieses Attribut aktiviert, werden die Randbedingungen der Kachelgrenzen beachtet, so dass, wenn die Kacheln nebeneinander positioniert werden, die erzeugten Instanzen den Mindestabstand einhalten (5.29b). Anderenfalls kann es vorkommen, dass an den Kachelgrenzen Instanzen den Mindestabstand unterschreiten (5.29a).

6 Level of Detail (LOD)

Um bei Szenen, in denen viele Objekte vorhanden sind, keine Performance-Verluste hinnehmen zu müssen, sind entsprechende Techniken nötig, die die Szene vereinfachen, möglichst ohne dass es der Betrachter merkt. Solche Methoden werden **Level of Detail-Methoden** (kurz LOD) genannt.

Das Ziel bei der Darstellung großer Szenen besteht aus einer realistischen Visualisierung, die möglichst in Echtzeit erfolgen soll. Dem entgegen stehen die geometrische Komplexität und die ggf. aufwändige Beleuchtungsrechnung. Dabei ist lediglich die geometrische Komplexität der darzustellenden Modelle ein Performance-Problem, welches hier bearbeitet werden kann.

Hilfreich ist die begrenzte Ortsauflösung des menschlichen Auges, denn es ist offensichtlich, dass mit zunehmender Distanz immer weniger Details gesehen werden können. Daraus resultiert, dass Details einer Szene ab einer bestimmten Distanz zum Betrachter nicht mehr wahrnehmbar sind und somit entfallen können, was zu einer beachtlichen Reduktion der Komplexität führt.

Ein Objekt, dessen visuelle Größe in einem Bild nur wenige Pixel beträgt, weil es etwa im Hintergrund steht, muss nicht in seiner vollständigen Auflösung dargestellt werden. Die geometrische Größe sollte somit entsprechend des Abstands zum Betrachter an die visuelle Größe angepasst werden. Die Übergänge zwischen den verschiedenen Detaillierungsgraden sind automatisch und sollten möglichst fließend sein. Der große zeitliche Aufwand, Modelle für jeden Detaillierungsgrad erstellen zu müssen, entfällt, wenn sich die Modelle selbst anpassen können.

Der Katalog an LOD-Methoden ist so umfangreich wie die Objekte, auf die sie angewendet werden. So müssen beispielsweise Häuser anders behandelt werden als Pflanzen oder gar Landschaften. Grundlegend wird zwischen statischen und dynamischen Verfahren unterschieden.

Bei den statischen Verfahren werden verschiedene Detailstufen eines Modells vorab erzeugt, zwischen denen mit zunehmender Entfernung zur virtuellen Kamera vom komplizierten hin zum einfachen Modell überblendet wird. D.h. beim Übergang von zwei Modellen werden diese übereinander gezeichnet, und mittels Alpha-Blending wird die

Transparenz des neuen Modells erhöht, die des anderen hingegen verringert.

Die dynamischen Verfahren passen die Detailstufen des Modells kontinuierlich je nach projizierter Größe an, dabei beinhalten die erzeugten Modelle selbst die benötigten Methoden zur Vergrößerung bzw. Verfeinerung.

In der Praxis kommt oft eine Kombination beider Verfahren zur Anwendung. Nachfolgend werden gängige dynamische Methoden vorgestellt, von denen eine in dieser Arbeit umgesetzt wurde, sie wird in Abschnitt 6.2 genau erläutert.

Die grundlegenden Ansätze der dynamischen Methoden funktionieren meist recht ähnlich. Zum einen wird versucht, Teile zu entfernen, zum anderen werden Teile vereinfacht. Eine Vereinfachung besteht meist in der Verringerung der Polygonzahl, wie in Abbildung 6.1 an einem Kugelobjekt demonstriert.

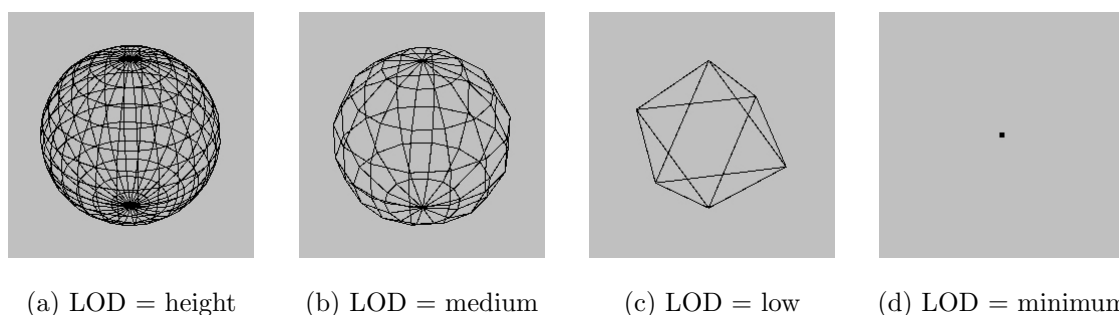


Abbildung 6.1: Verschiedene Detailstufen einer Kugel

Das Entfernen ganzer Teile kann ebenfalls auf verschiedene Arten realisiert werden. So können zum Beispiel rückwärtige Äste eines Baums entfernt werden, da sie ohnehin nicht gesehen werden können, oder es kann die Anzahl der Äste variiert werden. Bei Pflanzen entstehen oft große durchsichtige Lücken, so dass es schnell zu einer anderen Kontur kommt, was wiederum durch entsprechende Skalierungen der restlichen Äste kompensiert werden kann. Dieses auf Lintermann [11] zurückgehende Verfahren wird im anschließenden Abschnitt noch ausführlich beschrieben.

Andere Verfahren, wie das von Beaudoin und Keyser [6] vorgestellte, arbeiten auf der Struktur des erzeugten Objekts und befassen sich mit der Vereinfachung der Aststruktur. Ausgehend von der Länge eines Astes und der Orientierung der Kindäste relativ zum Vaterast wird die Richtung und Länge der vereinfachten Aststruktur berechnet. Abbildung 6.2 demonstriert verschiedene Auflösungsstufen dieses Verfahrens.

Eine weitere Methode ist das Zusammenfassen von Teilen und deren Ersetzung durch ein Bild der Ursprungsszene. So können z.B., wie in Abbildung 6.3 zu sehen, die einzelnen Blätter an einem Zweig, für die jeweils eine eigene Textur geladen werden muss, durch eine

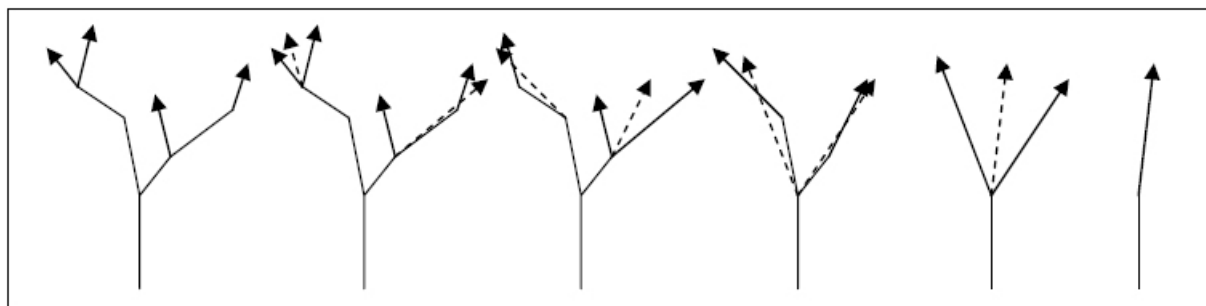
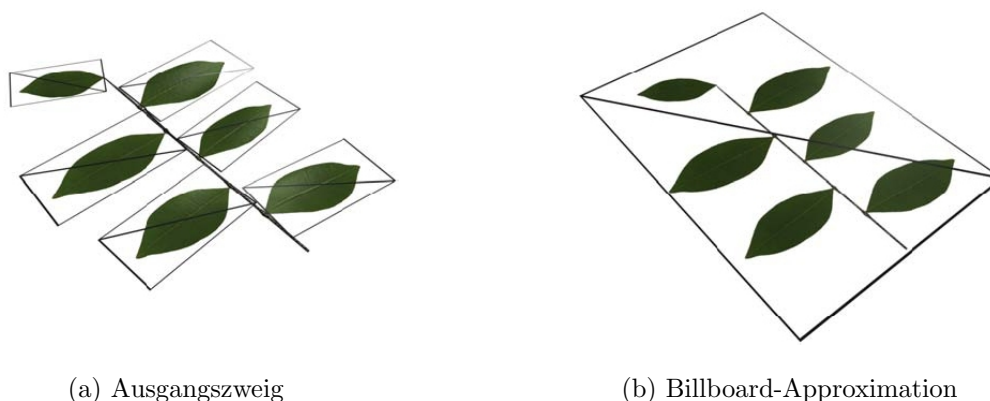


Abbildung 6.2: LOD-Demonstration für eine einfache baumähnliche Struktur nach Beaudoin



(a) Ausgangszweig

(b) Billboard-Approximation

Abbildung 6.3: LOD-Darstellung eines Zweiges

einzigste "große" Textur ersetzt werden. Dies erfordert selbstverständlich spezielle Texturen, die die Charakteristik der kombinierten Blätter repräsentieren.

Aus dieser Idee folgt ein grundsätzliches Verfahren, bei dem Pflanzen durch eine Menge von texturierten Ebenen, den so genannten Billboards, dargestellt werden. Werden die Texturen "geschickt" gewählt, so lässt sich kaum ein Unterschied zur ursprünglichen Szene ausmachen. Entscheidend ist allerdings das Verfahren, mit dem die benutzten Billboards bestimmt werden. Einige interessante Ansätze werden bei Décoret [9] vorgeschlagen, das Grundprinzip ist meist dasselbe. Dabei "suchen" iterative Clusteringansätze nach ähnlichen Dreiecken, die zusammengefasst und durch ein neues Oberflächenelement ersetzt werden. Die Definition von Ähnlichkeit variiert jedoch bei den einzelnen Verfahren.

6.1 LOD-Modellierung in GroIMP

In GroIMP stehen pro Darstellungsart (Wireframe (AWT)/OpenGL) derzeit jeweils vier LOD-Stufen zur Auswahl (6.4). Die Auswirkungen auf ein Kugelobjekt sind in 6.1 dargestellt.

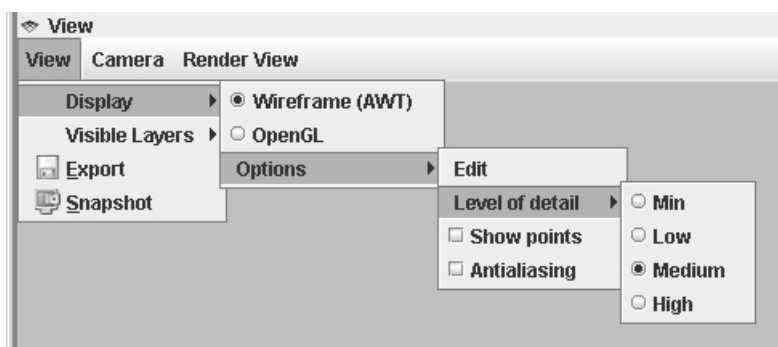


Abbildung 6.4: GroIMP-LOD-Menü

Die Detailstufe LOD_{local} , in der die Primitivobjekte dargestellt werden, ergibt sich aus der globalen LOD-Stufe, die das Maximum angibt, und der Entfernung, die das Objekt zum Benutzer hat.

$$LOD_{local} = LOD_{global} * Abstand,$$

wobei der *Abstand* prozentual angibt, wie viel der aktuell maximal zulässigen Detailstufe LOD_{global} , "verwendet" werden darf.

Der *Abstand* kann leider nicht direkt berechnet werden. Intern wird er mit Hilfe der Anzahl der Pixel, die für eine Kante eines Würfels der Seitenlänge Eins benötigt werden, approximiert. Zur Veranschaulichung: Ein Würfel, der nah am Benutzer ist, erscheint größer, benötigt somit mehr Pixel zur Darstellung als einer, der weit weg ist. Ausgehend von der geometrischen Größe des Objekts in Pixeln im Verhältnis zur aktuellen Darstellungsgröße des Objektes in Pixeln ergibt sich ein Wert zwischen Null und Eins.

Der globale LOD-Wert LOD_{global} wird entsprechend den zur Verfügung stehenden Ressourcen automatisch angepasst. Bei Ressourcenknappheit wird der globale LOD-Wert verringert, bis er bei ausreichenden Ressourcen schrittweise auf den vom Benutzer eingestellten Wert erhöht werden kann.

Die Detailstufensteuerung der Bausteine hat weitere Parameter, mit denen für eine Objektklasse eigene Detailstufen eingestellt werden können. Der aktuelle LOD-Wert entsprechend einer speziellen LOD-Methode $aktLOD_{methode}$ ergibt sich wie folgt:

Der globale LOD-Wert wird bestimmt und das Verhältnis zum maximalen globalen LOD-Wert LOD_{MAX} berechnet. Das Resultat wird mit $LOD_{global}P$ bezeichnet, wobei das P im Folgenden einen prozentualen Wert identifiziert.

$$LOD_{global}P = (LOD_{global} + 1) / (MAX_LOD_{global} + 1) \quad (6.1)$$

Um eine glatte Funktion des Abstandes zur virtuellen Kamera zu erreichen, wird der Abstand eines Objektes, wie oben beschrieben, aus dem Verhältnis einer Strecke der Länge

Eins in Pixeln an der aktuellen Objektposition *lengthOfOne* zur Länge dieser Strecke in Pixeln an der kürzest möglichen Entfernung zur Kamera bestimmt.

$$Abstand_{object}P = lengthOfOne / MAX_PIXEL \quad (6.2)$$

Dies beschreibt vorerst nur das reine Größenverhältnis, beachtet werden muss des Weiteren die geometrische Größe des Ausgangsobjekts, was in den Bausteinen unterschiedlich realisiert wurde. So wird im PhiBall-Baustein der größte Radius, im Tree-Baustein der *trunkScale*-Wert, im Horn-Baustein der *length*-Wert und im Arrange-Baustein die maximale Ausdehnung der Grundfläche als Referenz verwendet.

Bis hier ergibt sich der lokale LOD-Wert in Abhängigkeit vom globalen LOD-Wert und dem Abstand.

$$LOD_{local}P = Abstand_{object}P * LOD_{global}P \quad (6.3)$$

Als nächstes kommen die lokalen Einstellungen der Bausteine ins Spiel. Die fünf vielfältigsten Bausteine Horn, Tree, Hydra, PhiBall und Arrange verfügen jeweils über einen eigenen LOD-Block, in dem verschiedene LOD-Methoden zusammengefasst sind. Die meisten Methoden können über Regler von Null bis $2 * LOD_MAX$, dem doppelten des maximalen globalen LOD-Wertes von GroIMP, gesteuert werden.

$$LOD_{methode}P = (Wert_{LOD-Methode} + 1) / (2 * LOD_MAX + 1) \quad (6.4)$$

Somit ergibt sich der endgültige LOD-Wert einer speziellen LOD-Methode aus dem lokalen LOD-Wert $LOD_{local}P$ multipliziert mit dem LOD-Wert der Methode $LOD_{methode}P$:

$$aktLOD_{methode}P = LOD_{methode}P * LOD_{local}P, \quad (6.5)$$

wobei die Berechnung des aktuellen LOD-Wert einer Methode $aktLOD_{methode}P$ sich von Methode zu Methode unterscheidet.

Im letzten Schritt wird der so ermittelte Wert auf die der Methode entsprechende Eigenschaft angewendet, um einen endgültigen Wert zu berechnen.

Zur Veranschaulichung soll folgendes konkretes Beispiel dienen. Ausgehend von 3, dem maximalen globalen LOD-Wert LOD_MAX , der momentan in GroIMP möglich ist, und einer angenommenen aktuellen Einstellung von "medium" (s. Abbildung 6.4), was einem Wert von zwei entspricht, ergibt sich nach Formel 6.1 für $LOD_{global}P$ ein Wert von 0.75. Global ist somit ein Detailierungsgrad von maximal 75 Prozent zugelassen.

Ist das Objekt nicht allzu weit entfernt, so dass eine Linie der Länge Eins noch mit 10 Pixeln dargestellt wird, wohingegen für die selbe Linie in der kürzesten Distanz 50 Pixel benötigt werden, ergibt sich als Abstandswert aus Formel 6.2 für $Abstand_{object}P$ der Wert von 0.2, entsprechend 20 Prozent der Maximalgröße.

Bei einem maximalen Detailierungsgrad von 75 und der nur noch 20-prozentigen Größe

folgt für das Objekt nach Formel 6.3 ein lokaler LOD-Wert $LOD_{local}P$ von 0.15. Das Objekt kann somit an dieser Position nur noch mit Detailierungsgrad von 15 Prozent dargestellt werden.

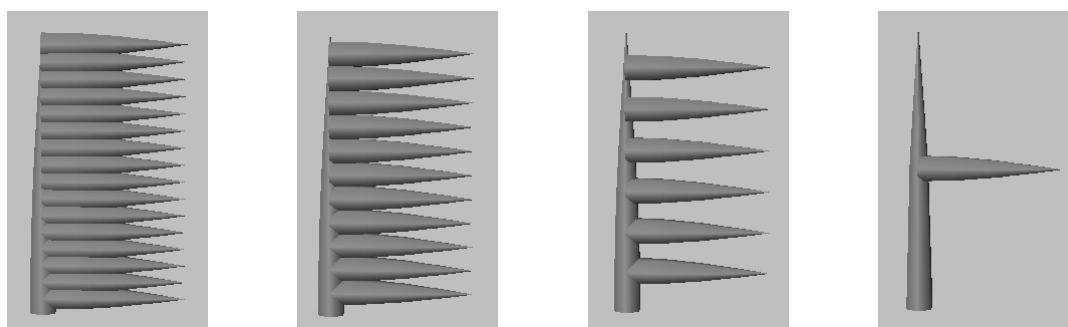
Der Horn-Baustein verfügt beispielsweise über eine LOD-Methode, mit der die Anzahl der vervielfältigten Instanzen gesteuert werden kann. Bei geringer Detailstufe bzw. entsprechend dem Abstand zum Betrachter wird die Anzahl der "Äste" verringert. Der Grad der Verringerung wird von der *Horn-segments*-LOD-Methode bestimmt und sei hier maximal, woraus gemäß Formel 6.3 für $LOD_{methode}P = 1$ folgt.

Der endgültige LOD-Wert der *Horn-segments*-LOD-Methode $aktLOD_{methode}P$ beträgt somit 0.15.

Die dem zulässigen Detailierungsgrad entsprechende Anzahl der erzeugten Instanzen LOD_{Anzahl} ergibt sich letztlich aus der Anzahl, die vom Horn-Objekt erzeugt werden soll ($sollAnzahl$) multipliziert mit dem aktuellen LOD-Wert der "segments"-LOD-Methode.

$$LOD_{Anzahl} = sollAnzahl * aktLOD_{methode}P$$

Unter diesen Einstellungen werden somit maximal 15 Prozent der $sollAnzahl$ an Objekten von dem Horn-Objekt vervielfältigt.



(a) segments-LOD = 6 (b) segments-LOD = 4 (c) segments-LOD = 2 (d) segments-LOD = 0

Abbildung 6.5: Verschiedene Detailstufen eines Horn-Objekts

In Abbildung 6.5 sind verschiedenen Einstellungen zu sehen. Variiert wurde nur der Wert der *Horn-segments*-LOD-Methode, der Abstand und der globale LOD-Wert bleiben unverändert, die Verringerung der Anzahl geschieht linear.

Ausgehend von den vier globalen Detailstufen (6.6), die momentan in GroIMP implementiert sind, den stufenlosen Abstands-LOD-Wert und den in den Bausteinen implementierten LOD-Methoden ergeben sich somit eine Vielzahl verschiedener Detailstufeneinstellungen, die so für fließende Übergänge zwischen den Auflösungsstufen des instanziierten Modells sorgen.

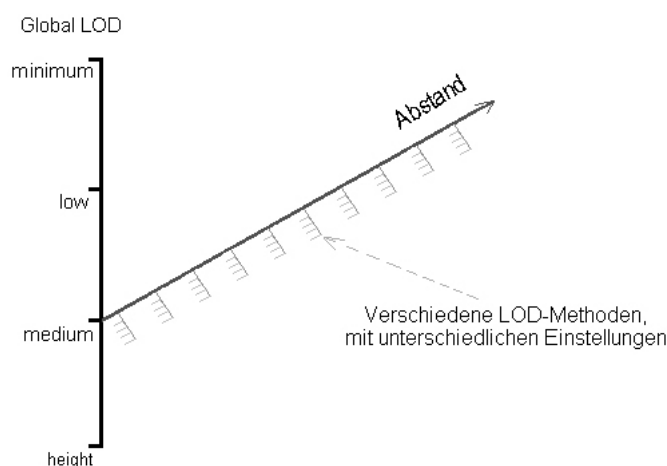


Abbildung 6.6: Zustandekommen der Detailstufeneinstellung

6.2 Umgesetzte LOD-Methoden

Die Bausteine Horn, Tree, Hydra, PhiBall und Arrange verfügen jeweils über einen eigenen LOD-Block, in dem verschiedene LOD-Methoden zusammengefasst sind.

Die hier verwendeten Methoden basieren auf einem von Lintermann [11] entwickelten Verfahren, das die Tatsache ausnutzt, dass die Strukturen pflanzenähnlicher Objekte, im speziellen von Bäumen, fraktalen Charakter haben. Die daraus resultierende Selbstähnlichkeit der erzeugten Modelle, d. h. Einzelteile ähneln der Gesamtgestalt, ermöglichen es, ein Modell durch ein kleineres Modell, in dem Teile entfernt und andere dafür vergrößert werden, zu approximieren. Dieses Verfahren wird als "Drop and Resize" bezeichnet.

Allgemein stehen in den Bausteinen somit Funktionen zur Verfügung, mit denen die Anzahl der vervielfältigten Instanzen sowie ihre Skalierung beeinflusst werden kann. In dieser Umsetzung sind keine Verfahren implementiert, die versuchen, möglichst optimal Objekte zu löschen bzw. zu skalieren, um so ein harmonisches Gesamtbild zu erzeugen.

Die Berechnung des endgültigen Detailierungsgrads, in dem ein Objekt dargestellt wird, ergibt sich nach der im vorangegangenen Abschnitt beschriebenen Berechnungsvorschrift. Nachfolgend die Erläuterungen der implementierten LOD-Methoden, die die Bausteine gemeinsam haben:

useLOD Über dieses Attribut lassen sich sämtliche LOD-Methoden zu- bzw. abschalten, in der Grundeinstellung sind die LOD-Methoden aktiviert.

number Dieses Attribut steuert, wie stark die Anzahl der erzeugten Instanzen variiert wird. Abbildung 6.5 zeigt diesen Effekt an einem Horn-Objekt, das weitere Horn-

Objekte erzeugt. Je höher der eingestellte Wert, desto höher der Detailierungsgrad, je mehr Instanzen werden letztlich erzeugt.

Der Name dieses Attributs ist in den Bausteinen unterschiedlich, bezieht sich aber immer auf die Anzahl der vervielfältigten Instanzen. Die verschiedenen Namen rühren von den Originalnamen in Xfrog her, an die die Bausteine angelehnt sind. (Horn: *segments*; Tree: *branchesNumber*; Hydra, PhiBall, Arrange: *number*)

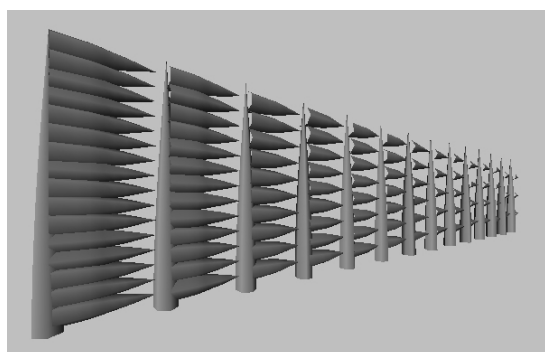
numberMode Bei diesem Attribut handelt es sich um ein FloatToFloat-Feld (5.1), mit dem Funktionen, wie in Anhang B beschrieben, definiert werden.

Die vom Benutzer festgelegte Funktion wird auf den Abstandswert angewendet. Somit kann der Modellierer die Rate, in der die erzeugten Instanzen verringert werden, entsprechend einer Funktion definieren.

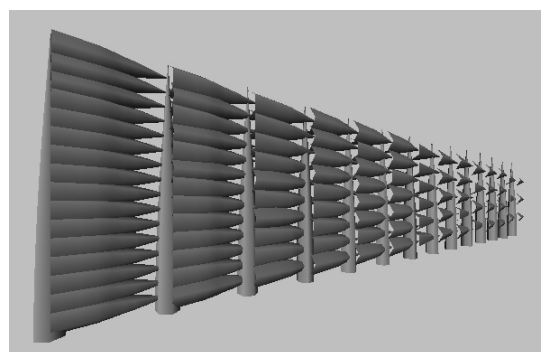
minNumber Mit diesem Attribut wird eine Mindestanzahl vervielfältigter Instanzen definiert, die von diesem Baustein erzeugt wird, auch wenn es laut der *number*-Einstellung weniger Instanzen wären. Initialisiert wird es mit Null.

scale Mittels *scale* wird festgelegt, wie stark die erzeugten Instanzen vergrößert werden, je weiter sie von der virtuellen Kamera entfernt sind. So können beispielsweise bei einer Objektreduzierung mit dem *number*-Attribut entstehende Lücken kaschiert werden.

Das Beispiel in Abbildung 6.7 zeigt ein Horn-Objekt, das wiederum Horn-Objekte vervielfältigt. Die Anzahl der Äste wird automatisch entsprechend der *number*-LOD-Methode und der Entfernung der Vaterinstanz zur virtuellen Kamera berechnet, wobei mit zunehmender Entfernung die Anzahl der Äste verringert wird. Die *number*-LOD-Einstellung ist in beiden Fällen maximal, nur die *scale*-LOD-Einstellung wurde von Null in Abbildung 6.7a bis maximal in Abbildung 6.7b verändert. Im direkten Vergleich, vor allem der hinteren Instanzen, ist der "lückenfüllende" Effekt deutlich zu sehen.



(a) *scale* = 0



(b) *scale* = *maximal*

Abbildung 6.7: *scale*-LOD-Methode an einem Horn-Beispiel demonstriert

scaleMode Bei diesem Attribut handelt es sich um ein FloatToFloat-Feld (5.1), mit dem Funktionen, wie in Anhang B beschrieben, definiert werden.

Die vom Benutzer festgelegte Funktion wird auf den Abstandswert angewendet und erlaubt somit beispielsweise quadratische oder logarithmische Skalierungen.

Der Maximalwert der *number*-Methode sowie der *scale*-Methode wird nach folgender Vorschrift bestimmt: $2 * (MAX_LOD_{global} - 1)$, bei den vier zur Zeit in GroIMP implementierten LOD-Stufen ergibt sich ein Maximalwert von sechs. Somit sind die LOD-Methoden der Bausteine ohne Änderungen am Quellcode auf zukünftige Erweiterungen der LOD-Stufen in GroIMP vorbereitet.

Der Tree- und der Horn-Baustein verfügen jeweils über eine weitere LOD-Methode, die *profile*-LOD-Methode.

profile Diese Methode ist nur an- bzw. ausschaltbar. Ist sie aktiviert, wird das Objekt durch das Verschieben der mit dem *Profile*-Attribut definierten Grundfläche gebildet, andernfalls werden einfache Zylinderobjekte, die wesentlich einfacher zu berechnen sind, verwendet.

(Randbemerkung: Von der ursprünglichen Idee, das Profil automatisch zu deaktivieren, wenn der globale LOD-Wert minimal ist, wurde abgesehen, da es immer wieder zu dem unerwünschten Effekt kommt, dass die Darstellungsart umschaltet, wenn man die Szene im 3D-Betrachter bewegt. Die Erklärung des Phänomens liegt darin begründet, dass der globale LOD direkt von den zur Verfügung stehenden Ressourcen abhängt. Sind diese knapp, wird der LOD entsprechend verringert, u. U. bis zum Minimalwert, woraufhin das Profil deaktiviert wird. Stehen Augenblicke später wieder ausreichend Ressourcen bereit, kann das Profil nicht einfach wieder aktiviert werden, da das System nicht zwischen automatischem und vom Benutzer deaktiviertem Profil unterscheiden kann.)

In dem BlockScale-, BlockColor- und im Variations-Baustein wurden keine LOD-Methoden integriert, da ihr begrenzter Funktionsumfang keine Variationen hinsichtlich der Detailstufenmanipulation ermöglicht.

7 Beispielanwendungen

Mit den in Kapitel 5 beschriebenen Bausteinen ist es möglich, ein breites Spektrum an nicht nur pflanzlichen Strukturen zu modellieren. Dieses Kapitel widmet sich einigen Beispielen, die einen kleinen Eindruck der Mächtigkeit dieser Modellierungsmethode vermitteln und die Vorgehensweise bei der Modellierung demonstrieren.

Die hier vorgestellten Modelle haben keinen Anspruch auf botanische Korrektheit, sie sollen lediglich in nachvollziehbaren Schritten den Weg von der Aufgabe bis zum fertigen Modell beschreiben. Die beschriebenen Beispiele versuchen, den verschiedenen Bausteinen mit ihren jeweiligen Eigenschaften gerecht zu werden und ihre Einsatzmöglichkeiten zu demonstrieren.

Für die Beispiele werden elementare Kenntnisse der Programmiersprache XL sowie im Umgang mit GroIMP vorausgesetzt. Einführungen und Beispiele sind bei Kniemeyer [29, 30] und im Internet unter www.grogra.de zu finden. Die XL-Sprachspezifikation steht unter www.grogra.de/xlspec bereit. Die Quellen der im Anschluss beschriebenen Beispiele sind ebenfalls im Netz unter www.grogra.de zu beziehen.

Wie in Kapitel 3 beschrieben, wird ein Modell durch zwei Teile beschrieben: Einerseits durch die Struktur, die über den p-Graph definiert wird, andererseits über einen Parametersatz, der die einzelnen Bausteine steuert. Der p-Graph wird in Form von XL-Code aufgebaut, wobei es bereits an dieser Stelle möglich ist, weite Teile der Parameter an die Bausteine zu übergeben.

Der letzte Abschnitt in diesem Kapitel 7.8 zeigt weitere Modelle, deren Aufbau aber nicht beschrieben wird. Sie sollen nur einen kleinen Ausblick auf die Vielfalt der Anwendungsmöglichkeiten geben.

7.1 Der Modellierungsprozess

Der Modellierungsprozess pflanzlicher Strukturen beinhaltet immer wiederkehrende Arbeitsschritte, die sich zu einem Leitfaden zusammenfassen lassen. Der Leitfaden beschreibt nacheinander die einzelnen Schritte, wie man von der Aufgabenstellung grundlegend zu einem Modell kommt.

Das Hauptaugenmerk bei der Modellierung liegt auf der Nachbildung der Struktur sowie in der Imitation des optischen Erscheinungsbilds einer Vorlage. Die botanische Korrektheit spielt eine eher untergeordnete Rolle, für viele Anwendungen genügt vielmehr eine möglichst überzeugende Illusion.

1. Analyse der grundlegenden Struktur der zu modellierenden Pflanze und gleichzeitiges Identifizieren von Bestandteilen, die einzeln modelliert werden können.
2. Erstellen aller benötigten Texturen. Die Texturen werden meist aus Bildern realer Beispiele ausgeschnitten und wenn nötig mit einem transparenten Hintergrund gespeichert.
3. Weitestgehend unabhängig voneinander werden die einzelnen Bestandteile modelliert.
4. Zusammensetzen der Einzelteile und Abstimmen u. a. der Proportionen aufeinander. Ein günstiges Vorgehen, dem das Zusammensetzen folgen kann, ist das sog. Bottom-Up-Prinzip: Ausgehend von kleinsten Teilen, die zu größeren verbunden werden, bis zur Gesamtstruktur.
5. Abschließende Feinabstimmung der Parameter, bis das gewünschte Ergebnis erzielt ist.

Der Bottom-Up-Prinzip hat den Nachteil, dass die einzelnen Bestandteile fast unabhängig voneinander modelliert werden und anschließend aufeinander abgestimmt werden müssen. Das Anpassen der Einzelteile aneinander kann wiederum recht aufwändig werden und u. U. weitreichende Korrekturen nach sich ziehen, da es grade bei miteinander verbundenen Tree-Bausteinen Abhängigkeiten zwischen den Attributen gibt, die nur schwer zu überblicken sind. Der Bottom-Up-Ansatz bietet sich somit nur für kleinere Strukturen an. Oftmals besser ist der Top-Down-Modellierungsprozess, bei dem ausgehend von der Hauptachse, meist den Stamm oder Stängel bzw. dem entsprechenden Wurzelobjekt, im p-Graph nach und nach die weiteren Teile hinzugefügt werden. Hauptvorteil ist, dass die Hauptstruktur weitestgehend modelliert ist, und nur noch die neu hinzugefügten Teile angepasst werden müssen. Die Auswirkungen der Abhängigkeiten zwischen den Tree-Bausteinen können direkt während der Modellierung integriert werden.

Der Modellierungsprozess in GroIMP unterteilt sich in zwei Schritte. In einem neuen RGG-Projekt wird mittels XL-Code die Struktur des Modells festgelegt und ggf. die ersten Parameter definiert. Nach dem Speichern des Projekts wird die Geometrie von GroIMP aufgebaut. Im integrierten Graph-Fenster wird der p-Graph der erzeugten Struktur angezeigt. Der zweite Modellierungsschritt besteht darin, die Bausteine so zu parametrisieren, bis das gewünschte Ergebnis erreicht ist. Dazu können im Graph-Fenster die Baustein-Knoten selektiert werden und ihre Attribute in Attribut-Fenster geändert werden.

Während der Entwicklungsphase eines Modells ist es ratsam, die Darstellungsart von GroIMP auf Wireframe (AWT) zu setzen. Die so eingesparten Berechnungen erlauben einen wesentlich schnelleren Modellaufbau, was letztlich die Modellierungsphase verkürzt. Beachte: Alle vorgenommenen Attribut-Einstellungen gehen verloren, nachdem die Struktur des Modells im XL-Code abgespeichert wird, was nach einer Änderung nötig ist. Wird hingegen das Projekt abgespeichert, werden die Attribute mitgespeichert.

7.2 Beispiel Gerbera

Dieses Beispiel beschreibt, wie der Blütenstand einer Gerbera modelliert werden kann. Ausgehend von dem Bild einer Gerbera in Abbildung 7.3a kann die Grundstruktur in den Stängel und den Blütenkopf unterteilt werden. Der Kopf wiederum unterteilt sich in mehrere Ringe kreisförmig angeordneter Blütenblätter.

Die Texturen in Abbildung 7.1 wurden, bis auf die der Sprossachse, der Abbildung 7.3a entnommen.

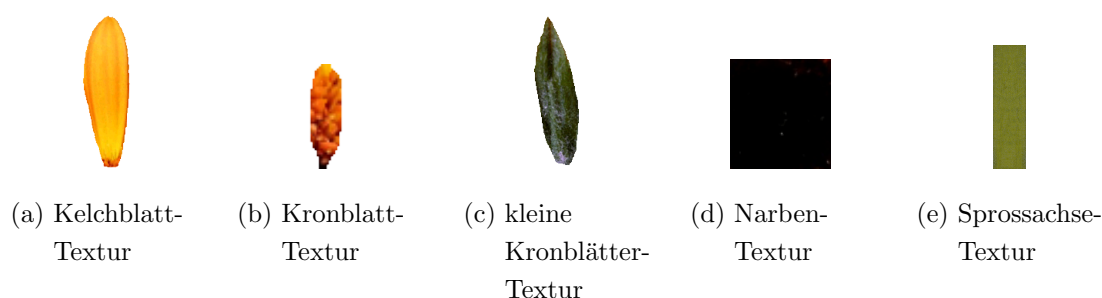
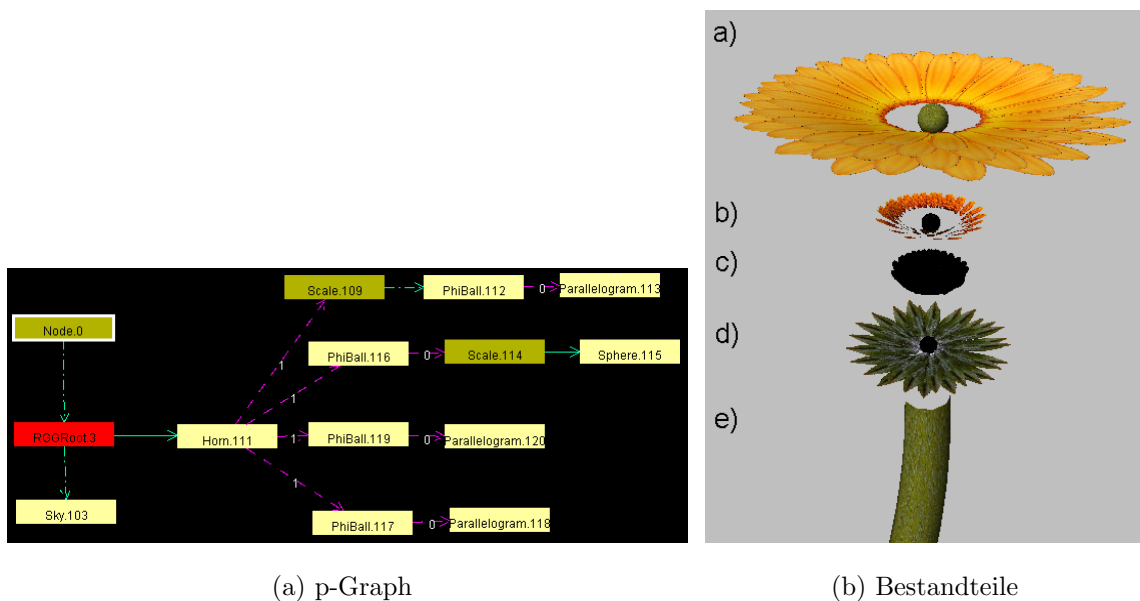


Abbildung 7.1: Texturen der einzelnen Modellteile

Das Modellieren der einzelnen Ringe kann grundsätzlich in beliebiger Reihenfolge vorgenommen werden. Der einzige übergreifende Punkt, auf den bereits bei der Modellierung geachtet werden kann, ist der Radius der Ringe, der aufeinander abgestimmt werden kann. Bei dieser Modellierung wird die Blüte in vier Ringe unterteilt, die übereinander gelegt den Blütenkopf bilden. Abbildung 7.2b zeigt die fertig modellierten Bestandteile, in die der Blütenkopf zerlegt wird. Die Teile 7.2b.a und 7.2b.b beschreiben die Ringe der Kelchblätter, Abbildung 7.2b.c zeigt die Narbe, und in Abbildung 7.2b.d ist der Ring der Kronblätter dargestellt. Der Stängel ist in Abbildung 7.2b.e zu sehen.

Bei den Kugelobjekten im Mittelpunkt der Ringe handelt es sich um "Anker" mit denen die Objekt im View-Fenster ausgewählt werden können, in der gerenderten Darstellung werden sie nicht dargestellt.



(a) p-Graph

(b) Bestandteile

Abbildung 7.2: Aufbau des Gerberamodells

Da das Vorgehen bei der Modellierung der Ringe a, b und d nahezu identisch ist, wird hier exemplarisch die Anordnung der großen Kelchblätter (7.1a bzw. 7.2b.a) erläutert.

Die Blütenblätter werden mit den von GroIMP bereitgestellten leaf-Bausteinen modelliert. Dabei handelt es sich um ein Parallelogramm, dessen Größe über den Konstruktor angegeben werden kann. Ihnen werden die entsprechenden Texturen zugewiesen, um so effizient den Eindruck eines Blattes zu erhalten, ohne die Form explizit durch eine Angabe der Umrisse definieren zu müssen. Alle Texturen müssen vorab erzeugt bzw. geladen werden. Die Methode *myleaf1a* erzeugt ein solches leaf-Objekt und gibt es zurück.

```
const ShaderRef leafmat1a = shader ("Lambert");
```

```
Parallelogram myleaf1a () {
    TMatrix4d m = new TMatrix4d ();
    m.rotZ(Math.PI/2);
    return leaf (5.0f, 2f).(setMaterial(leafmat1a), setTransform(m));
}
```

Um die Blütenblätter anschließend kreisförmig um einen zentralen Punkt anzuordnen, wird ein PhiBall-Baustein verwendet. Die Anzahl der zu erzeugenden Instanzen kann dem PhiBall direkt als Parameter übergeben werden. Für das Gerberamodell erwies sich 75 als gute Wahl. Sind die LoD-Methoden des Bausteins aktiviert, entfällt es sich die Anzahl der zu erzeugenden Instanzen etwas höher zu setzen, da sie durch den Eingriff der LoD-Methoden wieder reduzierte wird. In der Grundeinstellung erzeugt der PhiBall

eine Gleichverteilung auf einer Kugeloberfläche. Der Radius des PhiBalls wird relativ klein gewählt und der der Z-Achse wird zusätzlich noch verringert, um den Neigungswinkel der Kelchblätter zu erhöhen.

Die Kelchblätter einer Gerbera befinden sich nur in einem schmalen Streifen, um die Mitte herum. Möchte man diesen Effekt mit dem PhiBall-Baustein nachbilden, so muss der "Öffnungswinkel", auf dem die Objekte verteilt werden, mit den *Fan*-Attributen entsprechend begrenzt werden. Mit der *Scale*-Einstellung wird die Skalierung der erzeugten Blätter erreicht, so dass die inneren Blätter kleiner als die äußeren sind. Über eine *multiply*-Kante wird der PhiBall-Baustein mit dem leaf-Baustein verbunden, damit dieser vervielfältigt wird.

Diese XL-Code-Zeilen erzeugen den beschriebenen Ring von Kelchblättern, wie in Abbildung 7.2b.a zu sehen.

```
Ring1 ==> PhiBall(75).(setRadius(1.5), setRadiusZ(0.6), setFan1(1.5),
    setFan2(1.35), setScale1(0.8)) -multiply-> myleaf1a;
```

Die Narbe (7.2b.c) wird in Grundzügen ähnlich modelliert. Bei ihr werden leicht gestreckte Kugelobjekte von einem PhiBall-Baustein verteilt, um den Eindruck der vielen kleinen Einzelnarben zu erzeugen. Die Anordnung erfolgt hier jedoch über den gesamten oberen Teil des PhiBall-Objekts. Um dem PhiBall-Objekt mehr "Masse" zu geben und eine geschlossene Oberfläche zu erzeugen, wird mit der *setGeometry(true)*-Anweisung der Körper eingeschaltet. Das PhiBall-Objekt erzeugt somit eine eigene Geometrie, dies verringert die Anzahl der zu vervielfältigenden Instanzen, um eine lückenlose Füllung zu erreichen.

```
Ring2 ==> PhiBall(100).(setMaterial(mat1),setRadius(1.8),setRadiusZ(0.73),
    setFan1(0.7), setTrans1(0.3), setGeometry(true)) -multiply->
    Scale(0.2, 0.2, 0.5) Sphere;
```

Der letzte fehlende Bestandteil ist der Stängel. Er wird durch ein Horn-Baustein modelliert. Der Durchmesser wird mittels der *Range*-Attribute so bestimmt, dass ein nahezu gleich starker Stiel erzeugt wird. Mit den Rotationsangaben erzielt man ein leicht geschwungenes Objekt.

```
Stiel ==> Horn(15).(setMaterial(stemmat),setLength(0.375),setRotX2(0.05),
    setRotX1(-0.05), setRotZ1(0.05),setRange2(1.25),setRange1(1.4),
    setScale1(0.06));
```

Jetzt sind die Einzelteile nur noch zu kombinieren und aufeinander abzustimmen. Damit das Horn-Objekt die Ringe nur einmal erzeugt und sie am oberen Ende platziert, werden

sie über *child*-Kanten mit dem Horn verbunden. Damit alle Teile an die selbe Position gezeichnet werden, müssen sie in eckige Klammern gefasst werden, näheres dazu im Kapitel *Geklammerte L-Systeme 2.3.3*.

```
Blume ==> Stiel [-child-> Scale(1.5) [Ring1]]
           [-child-> Ring2]
           [-child-> Ring3]
           [-child-> Ring4];
```

Die so erzeugte Struktur ist im dazugehörigen p-Graph (s. Abbildung 7.2a) deutlich wieder zu finden.

Abbildung 7.3 zeigt im Direktvergleich das Bild der Ausgangsgerbera (7.3a), ein mittels Xfrog erzeugtes Modell (7.3b) und das in diesem Beispiel erzeugte GroIMP-Modell in Abbildung 7.3c.



(a) natürliche Gerbera

(b) Xfrog-Modell

(c) GroIMP-Modell

Abbildung 7.3: Beispielübersicht der Gerberamodelle

Da die großen Kelchblätter nur mittels zweidimensionaler leaf-Bausteine modelliert wurden und der PhiBall nur einen Prototyp eines Kelchblattes vervielfältigt, wirkt das GroIMP-Modell noch etwas flach und zu gleichmäßig, somit noch leicht unnatürlich, was aber für die Demonstration des Modellierungsprozesses nicht weiter stören soll.

Der hier demonstrierte Modellaufbau lässt sich auf fast alle Korbblüter übertragen, so dass durch kleine Änderungen an diesem Modell eine Vielzahl von Blüten modelliert werden kann, wie zum Beispiel Margariten oder Sonnenblumen.

Der vollständige XL-Quellcode der modellierten Gerbera aus *Gerbera.gsz*:

```

import de.grogra.imp3d.objects.*;
import de.grogra.blocks.*;
import static de.grogra.blocks.BlockConst.*;

module Blume;
module Stiel;
module Ring1;
module Ring2;
module Ring3;
module Ring4;

const ShaderRef leafmat1a = shader ("Lambert");
const ShaderRef leafmat1b = shader ("Lambert 2");
const ShaderRef leafmat = shader ("Lambert 3");
const ShaderRef stemmat = shader ("Lambert 5");
const ShaderRef mat1 = shader ("Lambert 4");

Parallelogram myleaf () {
    TMatrix4d m = new TMatrix4d ();
    m.rotZ(Math.PI/2);
    return leaf (4f, 1.4f).(setMaterial(leafmat), setTransform(m));
}
Parallelogram myleaf1a () {
    TMatrix4d m = new TMatrix4d ();
    m.rotZ(Math.PI/2);
    return leaf (5.0f, 2f).(setMaterial(leafmat1a), setTransform(m));
}
Parallelogram myleaf1b () {
    TMatrix4d m = new TMatrix4d ();
    m.rotZ(Math.PI/2);
    return leaf (1.1f, 0.5f).(setMaterial(leafmat1b), setTransform(m));
}

public void derive() [
    Blume ==> Stiel [-child-> Scale(1.5) [Ring1]]
                    [-child-> Ring2]
                    [-child-> Ring3]
                    [-child-> Ring4];

    Stiel ==> Horn(15).(setMaterial(stemmat),setLength(0.375),setRotX2(0.05),
        setRotX1(-0.05), setRotZ1(0.05),setRange2(1.25),setRange1(1.4),
        setScale1(0.06));
    Ring1 ==> PhiBall(75).(setRadius(1.5),setRadiusZ(0.6),setFan1(1.5),setFan2(1.35),
        setScale1(0.8)) -multiply-> myleaf1a;
    Ring2 ==> PhiBall(100).(setMaterial(mat1),setRadius(1.8),setRadiusZ(0.73),
        setFan1(0.7),setTrans1(0.3),setGeometry(true)) -multiply->
        Scale(0.2, 0.2, 0.5) Sphere;
    Ring3 ==> PhiBall(75).(setRadius(1.9),setRadiusZ(0.5),setFan1(0.8),setFan2(1.2),
        setScale1(1.2)) -multiply-> myleaf1b;
    Ring4 ==> PhiBall(100).(setRadius(0.5),setFan1(1.5),setFan2(1.4),setTrans1(-1.0))
        -multiply-> myleaf;
]

protected void init() {
    for (apply()) init2 ();
    for (apply()) derive ();
}

public void init2() [
    Axiom ==> [Sky.(setShader(new RGBAShader(0,0,0)))] Blume;
]

```

7.3 Beispiel Import eines Xfrog-Modells: beta1.xfr

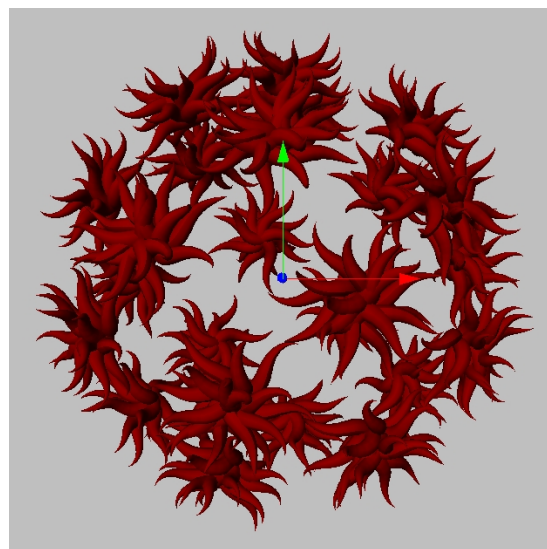
Mit dem ebenfalls im Rahmen dieser Diplomarbeit implementierten XFrog-FileParser (Anhang D), mit dem Xfrog-Modelle geladen und nach GroIMP importiert werden können, wird das folgende Beispiel erstellt.

Importiert wird eines der bei der Demoversion mitgelieferten Beispiele eines abstrakten Objekts, das in `../Greenworks/Xfrog 3.5/Models/Abstracts/beta1.xfr` zu finden ist.

Im GroIMP-Modell wurden die *useLOD*-Attribute des zweiten PhiBall- sowie des Horn-Bausteins deaktiviert, um das Resultat nicht durch unbeabsichtigte Eingriffe seitens der LOD-Methoden zu verändern.



(a) Xfrog-Modell



(b) GroIMP-Modell

Abbildung 7.4: Vergleich: Xfrog-Modell vs. nach GroIMP importiertes Xfrog-Modell

Aus Gründen der Performance wurde das *useLOD*-Attribut des ersten PhiBall-Bausteins nicht deaktiviert. Im zweiten PhiBall-Baustein und im Horn-Baustein wurde das *useLOD*-Attribut deaktiviert. Die Anzahl der vervielfältigten Instanzen im zweiten PhiBall wurde von 30 auf 20 herab gesetzt, was zu den leicht unterschiedlichen Geometrien in Abbildung 7.4 führt.

Zum Importieren eines Xfrog-Modells nach GroIMP wählt man nach dem Start von GroIMP aus dem *File*-Menü den Eintrag *Open*. Über den *Datei öffnen*-Dialog kann anschließend das Xfrog-Modell bzw. die entsprechende xfr-Datei ausgewählt werden. Das Modell wird daraufhin importiert. Das hier beschriebene Beispiel ist unter *beta1.gsz* zu finden.

7.4 Arrange-Beispiele

Die nächsten zwei Beispiele beschreiben den Umgang mit dem Arrange-Baustein. Das erste demonstriert den rein technischen Ablauf mit einem fiktiven "Wald", wohingegen das zweite Beispiel sich auf eine konkrete Waldsimulation bezieht. Die Einsatzmöglichkeiten des Arrange-Bausteins in der Forstwirtschaft, der Ökologie oder allgemein in der Biologie sind nahezu grenzenlos.

7.4.1 Arrange-Demo

Der folgende XL-Code erzeugt ein Arrange-Feld, auf dem 50 Horn-Objekte gleichmäßig verteilt werden. In der Grundeinstellung werden die erzeugten Instanzen in versetzten Reihen angeordnet, wie sie beispielsweise auf Obstplantagen zu finden sind.

```
import de.grogra.blocks.*;
import static de.grogra.blocks.BlockConst.*;

protected void init() {
    for (apply()) init2 ();
}

public void init2() [
    Axiom ==> Arrange(50) -multiply->
        BlockScale("0.01+n1", "0.01+n1", "0.5")
        BlockColor("10", "n2*256", "10")
        Horn(5).(setLength("0.05+n3*0.2"));
]
```

Der erzeugte p-Graph (7.5) zeigt die Struktur des Beispiels. Ausgehend von der Wurzel *Node* und dem RGG-Knoten folgt der Arrange-Knoten. Die von ihm vervielfältigte Struktur besteht aus einem BlockScale-, BlockColor- und einem Horn-Knoten.

In dieser Form erzeugen die Code-Zeilen aus *ArrangeTest.gsz* die Abbildung 7.7a. Mit dem Arrange-Baustein stehen interessante Manipulationsmöglichkeiten zu Verfügung, von denen einige nachfolgend vorgestellt werden.

Die erste, die hier betrachtet wird, ist das *Height field*. Mit ihm kann ein Höhenfeld definiert und der Grundfläche zugewiesen werden. Für dieses Beispiel wird Abbildung 7.6a als Vorlage für das Höhenfeld benutzt. Wird das Höhenfeld gesetzt, resultiert die Abbildung 7.7b. Der Arrange-Baustein gibt die Höhe der Position, an der ein Objekt

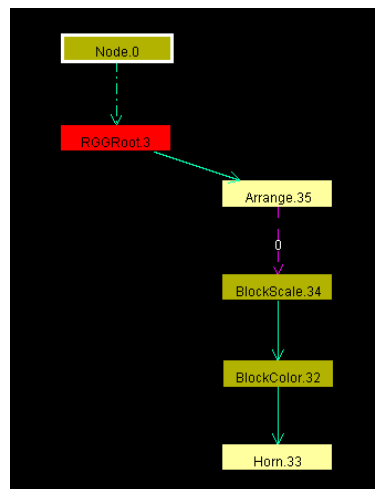


Abbildung 7.5: p-Graph zum Arrange-Beispiel

platziert wird, an dieses weiter, so dass der Wert in Funktionsfeldern (Kapitel 5.1) für Manipulationen genutzt werden kann.

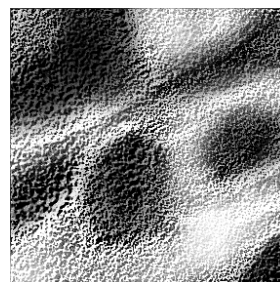
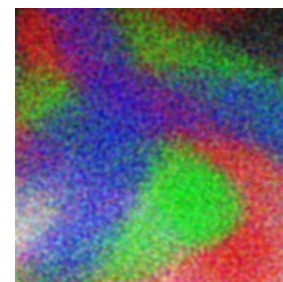
(a) *Height field*(b) *Density field*(c) *LocationParameter field*

Abbildung 7.6: Die dem Arrange-Beispiel zugrunde liegenden Felder

Mit den *Density field* kann ein Dichtefeld definiert werden, das nachträglich die Anordnung der Objekte bestimmt. Die Arrange-Attribute *densityMin* und *densityMax* definieren die Grenzen, innerhalb der Objekte platziert werden dürfen. Mit den Einstellungen $densityMin = 10$, $densityMax = 87$ und dem Dichtefeld (7.6b) wird, alleinig auf die Grundeinstellung angewendet, die Abbildung 7.7c erzeugt. Als Dichtefeld wird in diesem Beispiel das Negativbild des Höhenfelds benutzt, so wird auf einfache Weise erreicht, dass Objekte nur in den flachen Regionen der Grundfläche platziert werden können.

Die komplexeste Manipulationsmöglichkeit bietet das LocationParameter-Feld. Mit ihm können für jeden Punkt der Grundfläche drei Werte bestimmt werden, die, wie der Höhenwert, vom Arrange-Baustein an die erzeugten Instanzen weiter gegeben werden. Die Definition der Werte geschieht über ein Bild mit maximal 256 Farben, wobei jeder Farbkanal für einen LocationParameter-Wert steht. Rot steht für *LocationParameter1*, Grün für

LocationParameter2 und Blau für *LocationParameter3*. Abbildung 7.7d entsteht, wenn als *LocationParameter*-Feld Abbildung 7.6c definiert wird.

Die gleichzeitige Anwendung aller drei Felder erzeugt Abbildung 7.7e. Die Länge der Horn-Objekte wird von dem dritten *LocationParameter*-Wert bestimmt, der entsprechend skaliert mit `setLength("0.05 + n3 * 0.2")` gesetzt wird.

Die `BlockScale("0.01 + n1", "0.01 + n1", "0.5")`-Zeile im XL-Code skaliert die Horn-Objekte entlang der X- und Y-Achse in Abhängigkeit von dem *LocationParameter1*-Werten und bestimmt somit indirekt den Durchmesser. Die Z-Achse wird mit den Faktor 0.5 skaliert, um eine 50%-ige Verkleinerung zu erreichen.

Mit `BlockColor("10", "n2 * 256", "10")` wird die Farbe der Horn-Objekte definiert. Sie ergibt sich direkt aus dem *LocationParameter2*-Wert, der mit 256 multipliziert auf den Grünkanal gelegt wird. Gewisse Übereinstimmungen zwischen dem *LocationParameter field* 7.6c und Abbildung 7.7d sind durchaus zu erkennen.

7.4.2 Arrange-Anwendung: Waldmodellierung

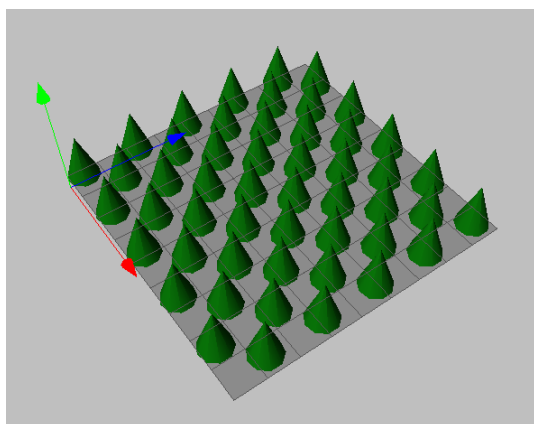
Das vorangegangene Arrange-Beispiel ist ein einfaches Beispiel, das nur den Umgang mit dem Arrange-Baustein demonstriert.

In dieser Waldmodellierung werden die Auswirkungen verschiedener Standortparameter auf das Wachstum von fiktiven Bäumen modelliert. Das hier erstellte Modell orientiert sich an Pretzsch [44] und an den Herleitungen der Zuwachsraten des Wachstumsmodells in der Simulationssoftware SILVA¹.

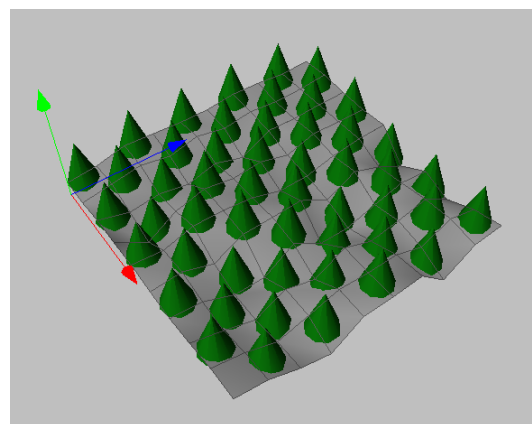
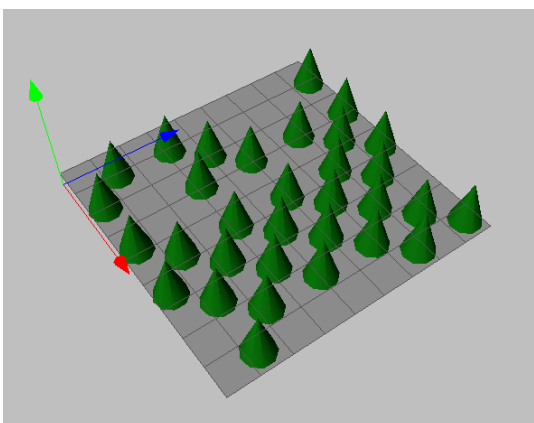
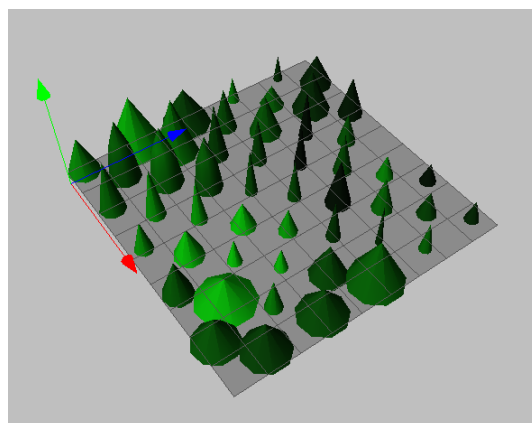
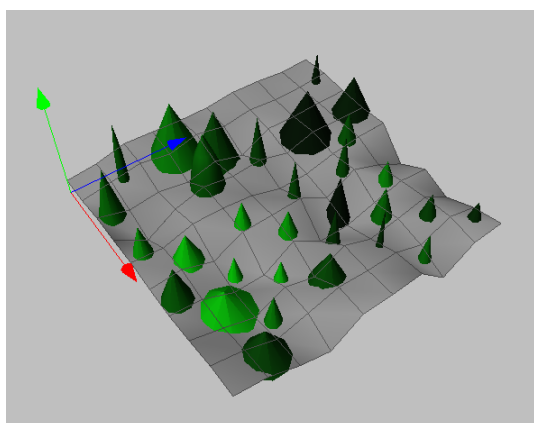
Ein Standort wird bei Pretzsch [44] durch neun Standortparameter beschrieben, diese sind:

- Nährstoffversorgung
- NO_X -Gehalt der Luft
- CO_2 -Gehalt der Luft
- Länge der Vegetationszeit (Anzahl von Tagen mit Temperaturen über 10°)
- Jahrestemperaturamplitude
- mittlere Temperatur in der Vergetationszeit
- Ariditätsindex nach De Martonne
- Niederschlagssumme in der Vergetationszeit
- Bodenfrische

¹Lehrstuhl für Waldwachstumskunde der Technischen Universität München; Prof. Dr. H. Pretzsch und Dr. P. Biber



(a) Grundfläche

(b) *Height field* angewendet(c) *Density field* angewendet(d) *LocationParameter field* angewendet

(e) Alle Felder angewendet

Abbildung 7.7: Das Arrange-Beispiel mit den Zwischenstufen

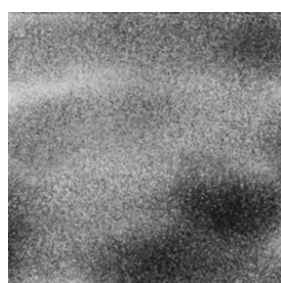
Für die hier erstellte Beispielmodellierung werden nur die folgenden drei Standortparameter betrachtet:

Wasserversorgung: Beschreibt die Bodenfeuchtigkeit.

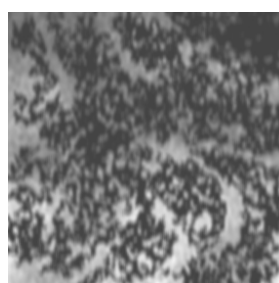
pH-Wert: Gibt den Säuerungsgrad des Bodens an, welcher stark mit dem Nährstoffgehalt korreliert.

Durchschnittstemperatur: Definiert die mittlere Temperatur in der Vegetationszeit.

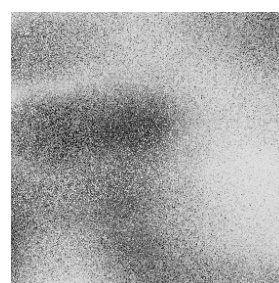
Die Standortparameter sind, wie der Name schon sagt, positionsabhängig und beziehen sich auf die für die Aufnahme durch eine Pflanze an dieser Position zur Verfügung stehenden Ressourcen zu einem konkreten Zeitpunkt. In der Vorbereitung werden sie in drei einzelnen Grauwertbildern (7.8), die die Konzentrationen angeben, erzeugt.



(a) Wasserversorgung



(b) pH-Wertverteilung



(c) Durchschnittstemperatur

Abbildung 7.8: Aufgliederung der Standortparameter

RGB-kombiniert ergeben die Standortparameter das in Abbildung 7.9 dargestellte Standortparameterfeld *LocationParameter field*. In gängigen Bildbearbeitungsprogrammen ist diese Funktion unter "RGB-Kanäle kombinieren" zu finden.

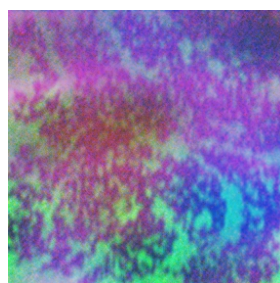
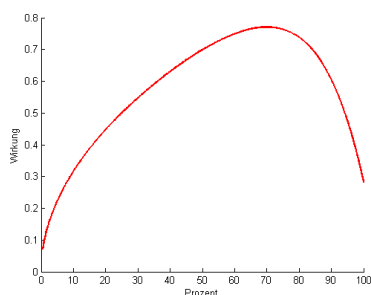


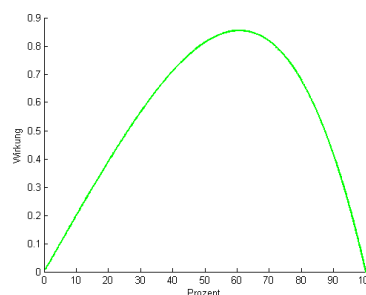
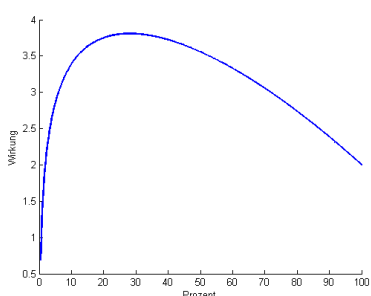
Abbildung 7.9: Das Standortparameterfeld *LocationParameter field*

Jeder dieser Standortparameter wirkt sich verschieden auf das Einzelwachstum eines Individuums aus. Die Auswirkung kann über folgende Funktionen (7.10) beschrieben werden:



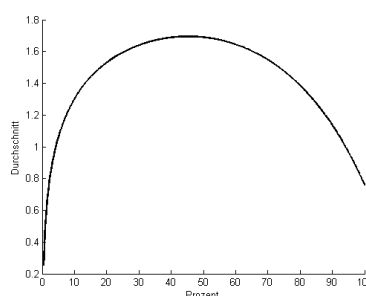
(a) Wasserversorgung:

$$r_1(x) = 1 + x^3 - \exp(x^3) + \sqrt{x}$$

(b) pH-Wert: $r_2(x) = -x^4 - x^3 + 2 * x$ 

(c) Durchschnittstemperatur:

$$r_3(x) = 6 + \log(x) - x * (x + 3)$$



(d) Durchschnittskurve:

$$r_{123}(x) = (r_1(x) + r_2(x) + r_3(x))/3$$

Abbildung 7.10: Auswirkung der Standortparameter auf das Wachstum

Die hier benutzten Funktionen sind stark vereinfacht, ähneln aber grundlegend den für Simulationen benutzten Funktionen.

In diesem Beispiel wird das Höhenwachstum l sowie der Stammdurchmesser d modelliert. Die Funktionen in 7.11 beschreiben Durchschnittswerte eines fiktiven 25-jährigen Bestandes unter den oben definierten Standortbedingungen.

Die Werte der Variablen x_1 und x_2 in den Wachstumsfunktionen ergeben sich aus Kombinationen der Standortparameter, die sich meist unterschiedlich auswirken. So fördert beispielsweise der eine Parameter das Laubwachstum, ein anderer hingegen hemmt das Längenwachstum. Dieses Beispiel geht von einer Durchschnittskurve $x_1 = x_2 = (r_1 + r_2 + r_3)/3$ aus, s. Abbildung 7.10d.

Der Stammdurchmesser wird durch den Durchschnitt der Standortparameter (7.10d) an seiner Position, auf die Durchmesserfunktion (7.11b) angewendet wird, bestimmt. Analog berechnet sich die Stammlänge.

Die eigentliche Modellierung beginnt im ersten Teil mit der Erstellung eines neuen RGG-Projekts, in dem mit diesem XL-Code ein einfaches Arrange-Objekt erzeugt wird, das 50 Horn-Objekte vervielfältigt.



(a) Länge: $l(x_1) = 0.05 * \sin(x_1) * \exp(x_1)$

(b) Durchmesser: $d(x_2) = 0.15 * \exp(x_2) * \log(1.1 * x_2)$

Abbildung 7.11: Graph der Längen- und Durchmesserfunktion

Axiom ==> Arrange(50) -multiply-> Horn(4);

Die Anzahl der Segmente im Horn-Objekt kann mit vier auf einen geringen Wert gesetzt werden, da in dieser Modellierung kein feingliedriges Objekt benötigt wird.

Im anschließenden zweiten Teil werden schrittweise die Parameter der Bausteine gesetzt, um das Modell in den Details zu definieren.

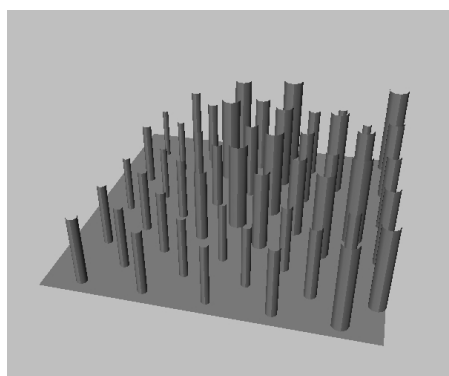
Geänderte Attribute im Arrange-Baustein:

1. Zur Vergrößerung der Grundfläche werden die *Rectangular X* und *Y Width* auf 20 verdoppelt. Analog wird mit den Grid-Feld-Attributen *Grid Size X* und *Y* verfahren.
2. Laden des Standortparameterfelds auf das *locationParameter*-Attribut.
3. Definieren der drei Standortparameter-Funktionen aus Abbildung 7.10 auf die *locationParameter(1/2/3)Function*-Attribute.
4. Das *useLOD*-Attribute deaktivieren.

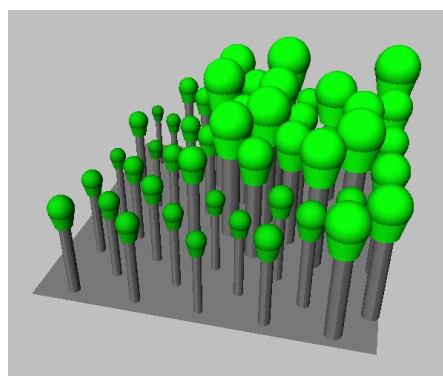
Geänderte Attribute im Horn-Baustein:

1. Setzen der Längenfunktion aus Abbildung 7.11a.
2. Definieren der Range-Attribute, um den Durchmesser der Horn-Objekte zu bestimmen. Durch das Gleichsetzen von *range1* und *range2* in Verbindung mit *rangeMode = Identity* Einstellung erreicht man einen gleich dicken Stamm mit dem Durchmesser von *range1*. Abbildung 7.11b zeigt die benutzte Durchmesserfunktion, die *range1* und *range2* zugewiesen wird.
3. Das *useLOD*-Attribute deaktivieren.

Zu diesem Zeitpunkt sieht die erzeugte Geometrie wie in Abbildung 7.12a aus.



(a) ohne Kronenmodell



(b) mit Kronenmodell

Abbildung 7.12: Ergebnis der Waldmodellierung

Als Zusatz wird nun ein stark vereinfachtes Kronenmodell entwickelt und in das bestehende Modell integriert.

Ein Kegelstumpf und eine Kugel bilden die schematische Krone eines Laubbaums, bei der die Größe, die Höhe sowie der maximale Durchmesser aufeinander abgestimmt sind.

Die folgenden Code-Zeilen erzeugen ein solches Kronenmodell:

```
Crown ==> Frustum(2.25, 1, 1.5) Sphere(1.5);
```

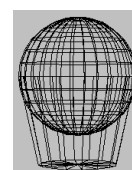


Abbildung 7.13: Das Kronenmodell in Gitternetzdarstellung

Der Kegelstumpf hat einen unteren Radius von eins, so dass die Krone durch einfaches Skalieren mit dem `BlockScale`-Baustein, der den Stammdurchmesser übergeben bekommt, auf die passende Größe gebracht werden kann. Die Anweisung `BlockColor("10", "h1 * 45", "10")` färbt die Kronen in Abhängigkeit von der lokalen Höhe, in der die Krone platziert wird.

```
Crown ==>
  BlockScale("0.2*exp((n1+n2+n3)/3)*log(1.1*(n1+n2+n3)/3)")
  BlockColor("10", "h1*45", "10")
  Frustum(2.25, 1, 1.5) Sphere(1.5);
```

Die Krone wird durch eine *child*-Kante mit dem Horn verbunden. Der vollständige XL-Code für das Waldmodell lautet somit (*WaldModell.gsz*):

```
import de.grogra.blocks.*;
```

```
import static de.grogra.blocks.BlockConst.*;

module Crown;

protected void init() {
    for (apply()) init2 ();
}

public void init2() [
    Axiom ==> Arrange(50) -multiply-> Horn(4) -child-> Crown;

    Crown ==>
        BlockScale("0.2*exp((n1+n2+n3)/3)*log(1.1*(n1+n2+n3)/3)")
        BlockColor("10", "h1*45", "10")
        Frustum(2.25, 1, 1.5) Sphere(1.5);
]
```

Das Endergebnis der Modellierung ist in Abbildung 7.12b zu sehen.

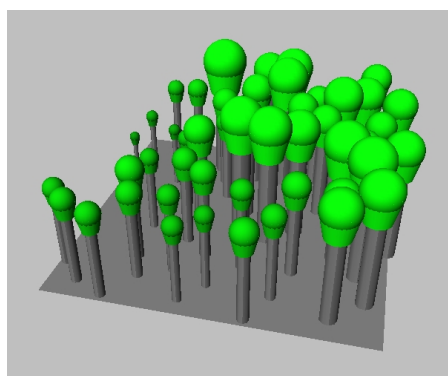
Ein in diesem Modell nicht betrachteter Punkt ist das Dichtefeld. Die Konkurrenz, in der Bäume zueinander stehen, beeinflusst das Baumwachstum jedoch erheblich. So wirkt sich der Abstand zwischen den Bäumen u. a. auf den maximalen Kronendurchmesser, die Wuchshöhe und somit auf den Stammdurchmesser aus. Möchte man diesen Aspekt in das Modell integrieren, so ist dies durch Einfügen eines Faktors in die Formeln der Abbildung 7.11a möglich.

Ein weiterer Punkt, der in dieser Modellierung nicht beachtet wurde, ist die Anordnung der Instanzen. Der Arrange-Baustein bietet eine Vielzahl vordefinierter Anordnungsmethoden, Abbildung 7.14 zeigt zwei mögliche Beispiele.

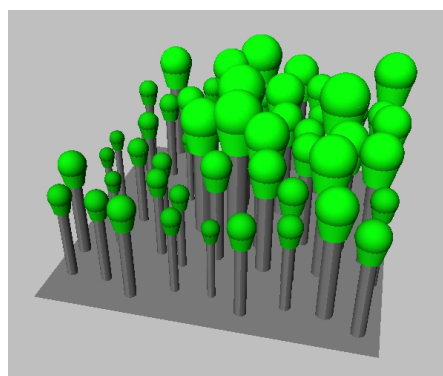
Mit dem erzeugten Modell lassen sich jetzt auf einfache Weise die Auswirkungen verschiedener Standortparameter-Konzentrationen simulieren. Dazu bedarf es jeweils nur eines neuen *LocationParameter fields* (7.9), das die gewünschten Standortparameter repräsentiert.

Im ersten Beispiel werden die Konzentrationen des pH-Werts und der Durchschnittstemperatur aus der bisherigen Modellierung beibehalten, einzig die Wasserversorgung wird überall konstant auf 50% gesetzt. Das daraus resultierende *LocationParameter field* ist in Abbildung 7.15a zu sehen.

Als Auswirkungen auf das Waldmodell lässt sich in Abbildung 7.16a ein leicht stärkeres Wachstum der Bäume erkennen.

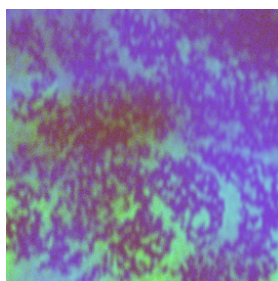


(a) Dart Throwing, Mindestradius = 1.8

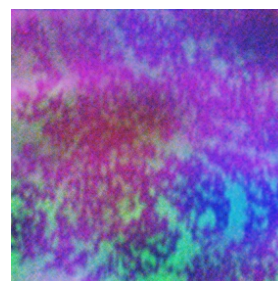


(b) Kachelung, Mindestradius = 1.8

Abbildung 7.14: Anordnungsvarianten im Waldmodell



(a) Gleichmäßige Wasserversorgung



(b) geringerer pH-Wert bei erhöhter Durchschnittstemperatur

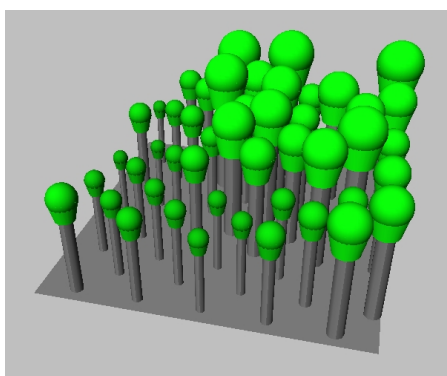
Abbildung 7.15: Verschiedene Standortparameterfelder

Das zweite Beispiel geht von einem um 20% geringeren pH-Wert und von einer 10%igen Erhöhung der Durchschnittstemperatur aus (7.15b). Wie in Abbildung 7.16b zu sehen, wirkt sich diese Kombination deutlich nachteiliger auf das Wachstum aus.

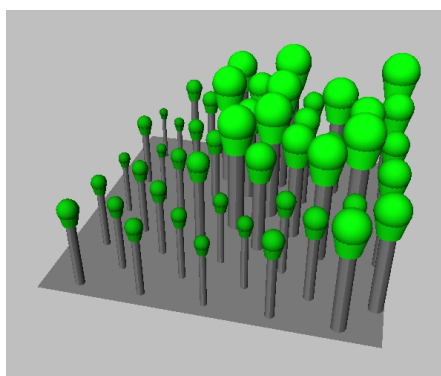
Mit diesem Verfahren können sehr bequem in einem kurzen Zeitraum eine Vielzahl von Standortparameter-Kombinationen untersucht werden, um so beispielsweise optimale Düngerkombinationen zu ermitteln.

7.5 Modellierung eines Baumes

In diesem Beispiel soll ein kleiner Baum modelliert werden. Das Resultat dieser Modellierung ist in Abbildung 7.17 zu sehen. Der Baum besteht aus vier Tree-Bausteinen, die hintereinander gehängt werden, es ergeben sich somit drei Verzweigungsebenen. Das letzte Tree-Objekt vervielfältigt die Blätter. Mit diesem XL-Code wird eine solche Struktur erzeugt (*TreeEx.gsz*):



(a) Erstes Beispiel



(b) Zweites Beispiel

Abbildung 7.16: Ergebnisse der Waldmodellierungen mit verschiedenen Standortparameterfeldern

```

1 import de.grogra.blocks.*;
2
3 protected void init() {
4     for (apply()) init2 ();
5 }
6
7 public void init2() [
8 {
9     LightNode light1 = new LightNode();
10    light1.setTransform(5,5,15); // position
11    PointLight pLight1 = new PointLight();
12    pLight1.getColor().set(1,1,1); // color
13    light1.setLight(pLight1);
14 }
15 Axiom ==> [Sky.(setShader(new RGBAShader(0,0,0)))] [light1]
16     Tree.(setName("Trunk"),setLayer(0),setMaterial(shader("Lambert")))
17     -BlockConst.multiply->
18     Tree.(setName("Branch1"), setLayer(1)) -BlockConst.multiply->
19     Tree.(setName("Branch2"), setLayer(2)) -BlockConst.multiply->
20     Tree.(setName("Branch3"), setLayer(3)) -BlockConst.multiply->
21     leaf (5f, 5f).(setName("Leaf"), setLayer(4),
22         setMaterial(shader ("Lambert 2")));
22 ]

```

Die Zeilen 9 bis 13 erzeugen ein Licht-Objekt, das in Zeile 15 zusammen mit einem schwarzen Himmel-Objekt in den Graph eingefügt wird. Beide Objekte werden ausschließlich für das abschließenden Rendern der Szene benötigt. Für Informationen über ihren Aufbau und die Verwendung wird auf die XL-Sprachspezifikation unter www.grogra.de/xlspec verwiesen.

Die *setName*-Anweisung weist den Bausteinen Namen zu, so dass sie im Graph-Fenster 7.18a leichter zu identifizieren sind. Mit der *setLayer*-Anweisung werden den Bausteinen verschiedene Darstellungsebenen zugewiesen, um später in der Modellansicht beispielsweise die Blätter einfach ausblenden zu können und so nur das Baumskelett darzustellen.

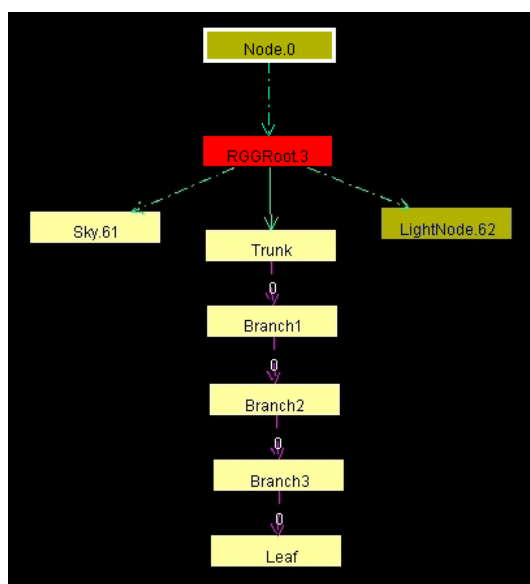


Abbildung 7.17: Fertiges Modell: *TreeEx.gsz*

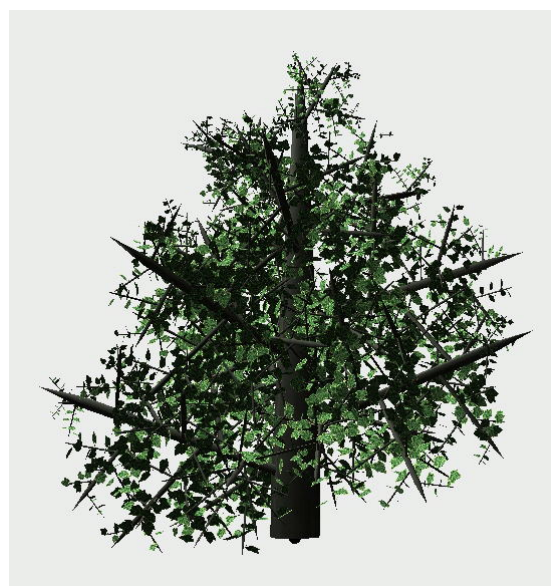
Der p-Graph der erzeugten Struktur sowie ihre Geometrie sind in Abbildung 7.18a zu sehen.

Um auf den Stamm und die Äste eine Rinden-Textur 7.19a zu legen, wird sie dem *Trunk-Tree* zugewiesen und von ihm ausgehend automatisch auf alle angehängten Objekte übertragen. Dieser Übertragungsmechanismus wird fortgesetzt, bis einem Objekt eine andere Textur zugewiesen wird. Zum Modellieren der Blätter wird den leaf-Objekten eine Blatt-Textur 7.19b zugewiesen, bei der der Hintergrund transparent ist, damit der Blattumriss die Bildgrenze bildet.

Im zweiten Modellierungsschritt werden die Attribute der Bausteine gesetzt. Das Baum-Modell ist einfach gehalten. Es wurde versucht, weitestgehend auf die Initialeinstellungen der Bausteine zurückzugreifen und nur wenige Parameter zu setzen. Die verwendeten

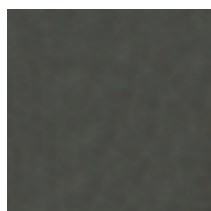


(a) Anordnung der Baustein-Knoten im p-Graph



(b) Ausgangszustand der Geometrie

Abbildung 7.18: Struktur und Geometrie des Baummodells



(a) Textur der Rinde



(b) Textur der Blätter

Abbildung 7.19: Verwendete Texturen

Funktionen wurden dabei nach Möglichkeit wiederverwendet.

Geänderte Attribute im *Trunk*-Tree-Baustein:

Spline: Das *Spline*-Attribut wurde aktiviert, damit die Trajektorie des Tree-Objekts der *Trajectory*-Kurve folgt.

Trajectory: Die Trajektorie wurde neu definiert, dazu wurden die Parameter in eine Datei gespeichert und anschließend geladen. Die hier verwendete Kurve gibt der Hauptachse des Baums einen leichten Schwung und lässt den Stamm somit natürlich wirken.

Shape: Um den Stamm schlanker zu modellieren, wurde die Stamm-Außenlinie neu definiert.

Branches Number: Die Anzahl der Äste wurde auf 16 erhöht, um dem Baum mehr Fülle zu geben.

Branches Distribution: Damit die Äste im oberen Baumbereich konzentriert werden, wurde die Verteilungsfunktion wie unten angegeben definiert: Der *type* wurde auf 2 gesetzt, um eine Hermite-Interpolation der Punkte zu erhalten.

Branches Angle: Das *BranchesAngle*-Attribut wurde auf *custom* gesetzt und 0.25 eingegeben, was einen Winkel von 45° Grad entspricht, damit die Äste leicht steil nach oben zeigen.

Trajectory-Kurve:

x	y	z
0.0	0.0	0.00
0.1	-0.1	2.00
0.0	0.3	4.00
0.2	0.2	6.00
0.0	0.3	8.00
0.2	0.0	10.00
0.0	0.2	12.00
-0.2	-0.1	14.00
0.0	0.0	16.00
-0.1	0.1	18.00
0.0	0.0	20.00

Shape-Funktion:

x	y
0.0	0.5
0.25	0.4
0.5	0.3
0.75	0.2
1.0	0.01

Distribution-Funktion:

x	y
0.0	0.0
0.3	0.0
0.5	0.6
0.75	0.75
1.0	0.0

Geänderte Attribute im *Branch1*-Tree-Baustein:

Spline, Trajectory und Shape: Wurden analog dem *Trunk*-Tree-Baustein geändert.

Crookedness Intensity: Wurde auf *custom* gestellt, um eine starke Irregularität im Zweigverlauf zu erzeugen.

LoD useLOD: Wurde deaktiviert, damit die Anzahl der erzeugten Zweige nicht durch die LOD-Methoden beeinflusst werden.

Geänderte Attribute im *Branch2*-Tree-Baustein:

Spline, Trajectory und Shape: Wurden analog dem *Trunk*-Tree-Baustein geändert.

Crookedness Intensity: Wurde auf *custom* gestellt und 0.1 eingegeben, um eine leichte Irregularität im Zweigverlauf zu erzeugen.

Branches Number: Die Anzahl der Äste wurde auf 8 verringert, damit nicht zu viele Objekte erzeugt werden und GroIMP beschleunigt wird.

LoD useLOD: Wurde deaktiviert, damit die Anzahl der erzeugten Zweige nicht durch die LOD-Methoden beeinflusst wird.

Geänderte Attribute im *Branch3*-Tree-Baustein:

Spline, Trajectory und Shape: Wurden analog dem *Trunk*-Tree-Baustein geändert.

Branches Number: Die Anzahl der Blätter wurde auf 15 erhöht, um eine geschlossene Laubdecke zu erzeugen.

Wie in Abbildung 7.17 zu sehen, reichen diese gut 20 Attribut-Änderungen aus, um ein einfaches, aber dennoch ansprechendes Modell zu erzeugen.

Mit den bereits erwähnten und hier verwendeten Layer-Einstellungen kann den Objekten eine Darstellungsebene zugewiesen werden. In GroIMP kann festgelegt werden, welche Ebenen angezeigt werden sollen. Standardmäßig werden alle Ebenen dargestellt. In Abbildung 7.20 wurde die 4-te Ebene ausgeblendet, in der im Modell die Blätter liegen, so dass nur noch das Baumskelett zu sehen ist.



Abbildung 7.20: Skelett des fertigen Modells

7.6 Kleines Farn-Modell

Dieses kleine Farn-Modell demonstriert einen Modellierungsprozess, bei dem das gesamte Modell in XL definiert wird. Zum Verständnis des Modells sind erweiterte Grundkenntnisse der XL-Programmierung nötig, auf die hier nicht eingegangen werden kann. Hilfestellungen zu XL stehen unter www.grogra.de/xlspec bereit.

In diesem Beispiel soll ein kleiner Farn modelliert werden. Aus der Strukturanalyse einer Farnpflanze ergibt sich der grundlegende Aufbau, der eine Pflanze in meist mehrere Wedel unterteilt, die wiederum entlang eines Stiels einzelne Äste vervielfältigen, von denen aus die Blätter seitlich abzweigen.

Die so ermittelte Struktur wird in den XL-Code-Zeilen 55 bis 68 erzeugt. Anders als zu den vorangegangenen Beispielen wurden in diesem Modell alle Parameter direkt an die Bausteine übergeben.

Die Definition der Parameter, der Kurven und Funktionen, geschieht in dem mit geschweiften Klammern begrenzten Bereich von Zeile 18 bis 53.

Mit dem Modul *myLeaf*, Zeile 13 bis 15, kann ein Blatt erzeugt werden. Ein Blatt ist in diesem Modell durch ein *leaf*-Objekt definiert, dem die entsprechende Farnblatt-Textur (7.21a) zugewiesen wurde.

Alle benutzten Texturen in Abbildung 7.21 wurden in Zeile 9, 10 und 11 definiert.

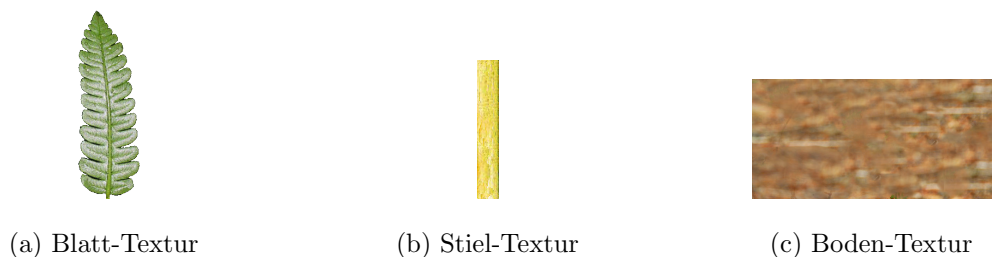


Abbildung 7.21: Texturen der einzelnen Modellteile

Der Aufbau der Farnpflanze ist wie beschrieben modular, dementsprechend wurde das Farn-Modell in verschiedene Einzelteile unterteilt. Die kleinste Einheit ist dabei ein Blatt (7.21a), wie es von dem Modul *myLeaf* erzeugt wird. Die nächst größere Struktur ist ein Ast, der mehrere Blätter seitlich entlang seiner Hauptachse verteilt. Abbildung 7.22 zeigt einen einzelnen Ast.

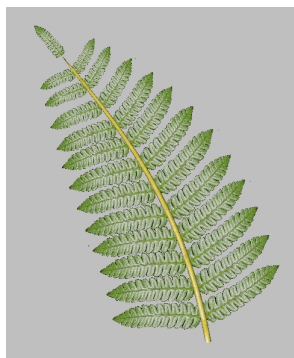


Abbildung 7.22: Der Ast eines Farnwedels

Der nächste Schritt ist es, aus den Ästen Farnwedel zu modellieren. Dazu werden die Äste wiederum von einem Tree-Baustein vervielfältigt, siehe Zeile 58.

```

1 import de.grogra.imp3d.objects.*;
2 import de.grogra.blocks.*;
3 import static de.grogra.blocks.BlockConst.*;
4
5 module FarnM;
6 module Stamm;
7 module Ast;
8

```

```

9 const ShaderRef leafmat = shader ("Lambert");
10 const ShaderRef stemmat = shader ("Lambert 2");
11 const ShaderRef groundmat = shader ("Lambert 3");
12
13 Parallelogram myLeaf () {
14     return leaf (3f, 1f).(setMaterial(leafmat));
15 }
16
17 public void derive() [
18 {
19     SplineFunction brScale = new SplineFunction(new Point2f[]
20         {new Point2f(0,1),new Point2f(0.5,0.75),new Point2f(1,0.2)},
21         SplineFunction.HERMITE);
22
23     SplineFunction distri = new SplineFunction(new Point2f[]
24         {new Point2f(0,0.8),new Point2f(0.75,0.7), new Point2f(1,0.95)},
25         SplineFunction.HERMITE);
26
27     //stamm
28     SplineFunction trunkShape = new SplineFunction(new Point2f[]
29         {new Point2f(0,0.1),new Point2f(0.5,0.07),new Point2f(1,0.03)},
30         SplineFunction.HERMITE);
31
32     SplineFunction distriS = new SplineFunction(new Point2f[]
33         {new Point2f(0,0),new Point2f(0.25,0.1), new Point2f(0.7,0.2),
34         new Point2f(0.9,0.35), new Point2f(1,0.5)},
35         SplineFunction.HERMITE);
36
37     SplineFunction angle = new SplineFunction(new Point2f[]
38         {new Point2f(0,0.5),new Point2f(0.5,0.5),new Point2f(1,0.5)},
39         SplineFunction.HERMITE);
40
41     SplineFunction brScale2 = new SplineFunction(new Point2f[]
42         {new Point2f(0,1),new Point2f(0.5,0.9),new Point2f(1,0.75)},
43         SplineFunction.HERMITE);
44
45     float[] data = {0,0,0, 1,0,5, 2,0,7, 5,0,9};
46     BSplineCurve trajectory = new BezierCurve (data, 3);
47
48     float[] data2 = {0,0,0, 0,1,5, 0,3,7, 0,4.5,9};
49     BSplineCurve trajectory2 = new BezierCurve (data2, 3);
50
51     TMatrix4d m = new TMatrix4d ();
52     m.rotZ(Math.PI/2);
53 }
54
55 FarnM ==> Hydra(3, false).(setRadius(1),
56     setSpin1(0.7), setSpin2(0.25), setScale1(1.57)) -multiply-> Stamm;
57
58 Stamm ==> Tree.(setMaterial(stemmat), setShape(trunkShape),
59     setBranchesGeometricScale(brScale2),
60     setArrangement(4), setBranchesDistribution(distriS),
61     setSpline(true), setTrajectory(trajectory), setBranchesAngle(angle))
62     [-multiply-> Scale(0.3) Ast] [-child-> Scale(0.1) Ast];
63
64 Ast ==> Scale(1.35) Tree.(setMaterial(stemmat), setShape(trunkShape),
65     setBranchesGeometricScale(brScale), setArrangement(5),
66     setBranchesDistribution(distri), setBranchesNumber(12),
67     setSpline(true), setTrajectory(trajectory2))
68     [-multiply-> myLeaf] [-child-> myLeaf.(setTransform(m))];
69 ]
70
71 protected void init() {
72     for (apply()) init2 ();
73     for (apply()) derive ();
74 }

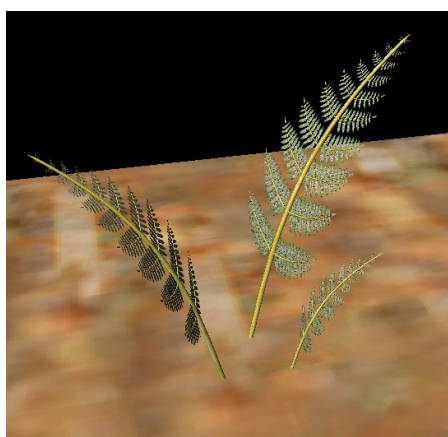
```

Im letzten Schritt, Zeile 55, wurden die Farnwedel von einem Hydra-Baustein dreimal kopiert. Die restlichen Parameter des Hydra-Bausteins sorgen für die gewünschte Skalierung, einen Radius von eins und eine leichte Rotation der erzeugten Instanzen.

Das Modell ist soweit vollständig. Der Block von Zeile 77 bis 85 definiert die Grundfläche, der eine dem Waldboden nachempfundene Textur 7.21c zugewiesen wurde. Die darauf folgenden fünf Zeilen erzeugen ein Licht-Objekt, das in der Zeile 93 zusammen mit einem schwarzen Hintergrund, der Grundfläche und dem Farn-Modell zum Gesamtmodell zusammengesetzt wird.

```
75 public void init2() [  
76 {  
77     RectangularHeightFieldMapping rhfm = new RectangularHeightFieldMapping();  
78     rhfm.setXwidth(30);  
79     rhfm.setYwidth(30);  
80     NetworkHeightField net = new NetworkHeightField();  
81     net.setMapping(rhfm);  
82     Patch ground = new Patch();  
83     ground.setMaterial(groundmat);  
84     ground.setGrid(net);  
85     ground.setTransform(-15,-15,0);  
86  
87     LightNode light1 = new LightNode();  
88     light1.setTransform(5,5,15);  
89     PointLight pLight1 = new PointLight();  
90     pLight1.getColor().set(1,1,1);  
91     light1.setLight(pLight1);  
92 }  
93     Axiom ==> [Sky.(setShader(new RGBAShader(0,0,0))] [ground] [light1] FarnM;  
94]
```

Abbildung 7.23 zeigt das fertige Farn-Modell und zum Vergleich eine kleine Farnpflanze aus der Natur.



(a) GroIMP-Farn-Modell



(b) Vergleichs-Farn aus der Natur

Abbildung 7.23: Gegenüberstellung

7.7 Kombination von Instanzierungsregeln mit Graph-Grammatiken

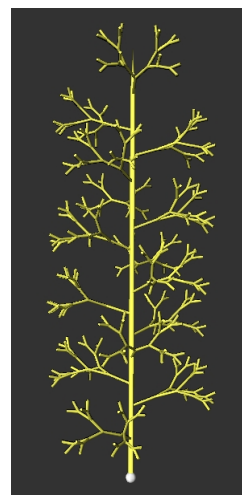
Einen sehr großen Zuwachs an Modellierungsfreiheit bietet die Möglichkeit, die Instanzierungsregeln mit Graph-Grammatiken zu kombinieren. Dabei ist die Kombination in beide Richtungen möglich. Einerseits können von den Bausteinen Strukturen, die über Graph-Grammatiken erzeugt wurden, vervielfältigt werden, zum Anderen ist es ebenfalls möglich, innerhalb von Graph-Grammatiken Instanzierungsstrukturen zu verwenden. In diesem Beispiel vervielfältigt ein Tree-Baustein eine kleine baumartige Struktur, die mittels der XL-Instanzierungsregel "Baum" erzeugt wurde, entlang seiner Hauptachse.

```
import de.grogra.blocks.*;
import static de.grogra.blocks.BlockConst.*;

module Baum(int depth, float x) ==>
  F(x)
  if (depth > 0) (
    [RU( 30) RH(90) Baum(depth - 1, x*0.8)]
    [RU(-30) RH(90) Baum(depth - 1, x*0.8)]
  );

protected void init() {
  for (apply()) init2 ();
}

protected void init2() [
{
  SplineFunction shape = new SplineFunction(
    new Point2f[] {
      new Point2f(0.0,0.1),
      new Point2f(0.5,0.1),
      new Point2f(1.0,0.1)}, SplineFunction.HERMITE);
}
  Axiom ==> Tree(15).(setShape(shape))
  -multiply-> Baum(4, 1);
]
```



Fast noch wichtiger für die Anwendung ist die Gegenrichtung, bei der mittels Graph-Grammatiken eine Simulation modelliert wird und die verwendeten Geometrien aus relativ einfach über Instanzierungsbausteine erzeugten Strukturen stammen. Abgeschlossene Pflanzenorgane, wie Blüten, oder vollständige Pflanzenmodelle müssen so nicht über einen aufwändigen Ableitungsprozess erzeugt werden.

7.8 Weitere Beispielmolelle

Anschließend ein paar weitestgehend unkommentierte Modelle, von denen nur die erzeugten Geometrien vorgestellt werden. Sie sollen als Anregung dienen und einen kleinen Eindruck vermitteln, über die Mächtigkeit der hier beschriebenen und umgesetzten Modellierungsmethode. Wie die Beispiele zeigen, sind den Anwendungsmöglichkeiten kaum Grenzen gesetzt.

Abbildung 7.24a zeigt ein Modell, das einer Blautanne nachempfunden ist. Es besteht aus vier hintereinander gehängten Tree-Bausteinen, von denen das Letzte die Textur eines Tannenzweiges vervielfältigt.

Das Modell einer Gerstenähre wird in Abbildung 7.24b dargestellt. Das eigentliche Korn wird mittels eines Rotationskörpers erzeugt, dem ein Horn-Objekt als Granne hinzugefügt wird.



(a) Tannen-Modell (*Tanne.gsz*)

(b) Gersten-Modell (*Gerste.gsz*)

Abbildung 7.24: Beispielmodelle

Das Modell einer Zimmerpflanze ist in Abbildung 7.25a zu sehen. Die Blattstiele bilden Horn-Objekte und die Blätter werden durch *leaf*-Bausteine erzeugt.

Abbildung 7.25b zeigt eine abstrakte Figur, bei der ein PhiBall-Baustein Horn-Objekte erzeugt.

Das interessante an dem Dalien-Modell in Abbildung 7.26a ist, dass die Blütenblätter nicht, wie in den vorangegangenen Beispielen, mit dem *leaf*-Baustein modelliert wurden. Eine geschwungenes Flächenstück bildet die Grundlage für die Textur und verleiht dem Blatt die nötige optische Tiefe, um das Modell plastisch wirken zu lassen.

Ein einfaches Modell einer Opuntie ist in Abbildung 7.26b zu sehen. Die einzelnen "Ohren" der Pflanze wurden durch PhiBall-Bausteine gebildet. Zwei entscheidende neue Funktionen des PhiBalls kamen dabei zum Einsatz: Zum einen die, dass die Radien jeder Achse frei definierbar sind, womit die flache Ellipsenform erzeugt wurde, zum anderen erzeugt der PhiBall seine eigene Geometrie, die den Körper bildet. Die Stacheln erzeugt ein Hydra-Baustein, der leicht geschwungene Horn-Objekte vervielfältigt.

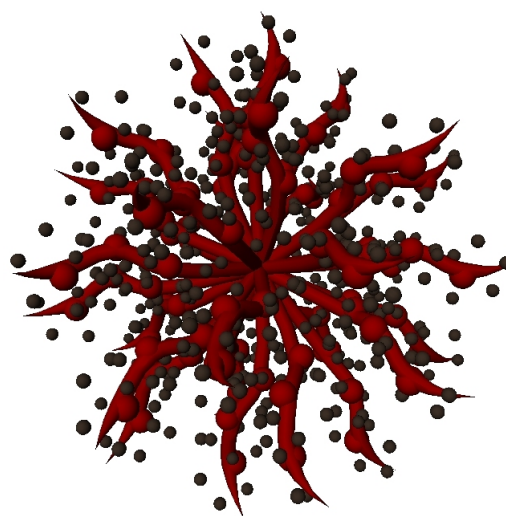
(a) Zimmerpflanze (*flowerT3.gsz*)(b) Abstrakte Figur (*abstract.gsz*)

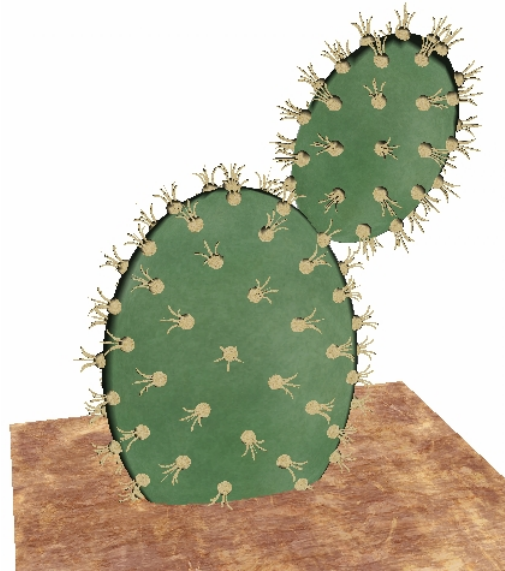
Abbildung 7.25: Fortsetzung Beispielmodelle 1

In Abbildung 7.26c wurde ein kleiner Blumenstrauß modelliert. Die einzelnen Blumen wurden mittels eines PhiBall-Bausteins verteilt. Der Pflanzenstängel wurde durch einen Tree-Baustein, der die Blätter vervielfältigt sowie den Blütenkopf an der Spitze platziert, modelliert.

Ein Strauch-Modell wurde in Abbildung 7.26d dargestellt. Es basiert auf einem importierten Xfrog-Modell, das etwas vereinfacht wurde. Drei miteinander verbundene Tree-Bausteine bilden die Struktur, von denen das letzte die Blätter erzeugt.



(a) Dalien-Modell (*Dalie.gsz*)



(b) Opuntien-Modell (*Kaktus.gsz*)



(c) Kleiner Blumenstrauß (*flowerT2.gsz*)



(d) Strauch-Modell (*bushTest.gsz*)

Abbildung 7.26: Fortsetzung Beispielm Modelle 2

8 Diskussion und Ausblick

In diesem Abschnitt werden die wichtigsten Ergebnisse der vorliegenden Arbeit zusammengefasst. Anschließend wird ein Ausblick auf zukünftige Arbeiten gegeben, die eine sinnvolle Erweiterung dieser Arbeit wären.

8.1 Diskussion

Gegenstand dieser Arbeit ist es, das Konzept der regelbasierten Objekterzeugung, wie es von Deussen mit der Xfrog-Software umgesetzt wurde, in Form von Instanzierungsregeln in die GroIMP-Software zu integrieren. Durch eine Reihe von Erweiterungen konnte die Mächtigkeit der Bausteine gesteigert werden, was den Freiheitsgrad der Modellierung und somit die Flexibilität im Umgang mit den Modellen erhöht.

Das Verfahren vereinigt den Ansatz der prozeduralen Modellierung mit dem der regelbasierten Modellierung, da im speziellen der L-Systeme. Ein Graph beschreibt den strukturellen Aufbau, während die Bausteine Informationen über einzelne Teile bzw. deren Erzeugung enthalten.

Mit dem hier umgesetzten Verfahren steht ein leistungsfähiges Werkzeug zur Verfügung, mit dem eine Vielzahl nicht nur pflanzlicher Strukturen modelliert werden können. "Mit dem Ansatz ist nichts modellierbar, was nicht auch in irgendeiner Weise mit parametrischen L-Systemen programmierbar wäre", so Deussen in [11].

Die erzeugten Computergrafiken sind mehr als nur schöne Bilder. Mit ihnen eröffnen sich neue Möglichkeiten beispielsweise in der Visualisierung ökologischer Daten. Sei es in der Landschaftsplanung, wo die Folgen von Eingriffen in Ökosysteme dargestellt werden: Entscheidungsträger werden so in die Lage versetzt, sich in einer geplanten virtuellen Landschaft umher zu bewegen und daraufhin die Planungsalternativen abzuwägen. Weitere Gebiete, die von diesen Entwicklungen profitieren könnten, sind u. a. die Architektur, Fahr- und Flugsimulatoren, computeranimierte Filme sowie Computerspiele.

Obwohl mit diesen Verfahren recht natürlich wirkende Pflanzenmodelle erstellt werden können, ist das Verfahren nicht so systematisch, wie man es sich wünschen würde. Der Modellierungsprozess beinhaltet ein erhebliches Maß an Probieren, bis der Parametersatz

so konfiguriert ist, dass ein erwünschtes Ergebnis erzeugt wird. Gewonnene Daten aus Vermessungen realer Pflanzen können nicht oder nur schwer in Parameter einer Simulation umgesetzt werden.

Durch die visuelle Rückmeldung, die GroIMP liefert, wird der vorhandene "Try and Error" Charakter im Modellierungsprozess jedoch gut kompensiert und die Arbeit erleichtert. Das soll aber nicht darüber hinweg täuschen, dass es ziemlich schwer ist, eine konkret vorgegebene Pflanze zu modellieren [8].

Das Verfahren ist biologisch nicht fundiert, und wie bereits beschrieben, lassen sich aus der Natur oder von einem Muster gewonnene Daten nur sehr schwer auf die Attribute der Bausteine übertragen. Eine Relation zwischen der Realität und dem erzeugten Modell kann fast nur nachträglich durch Messungen am Modell und Vergleichen mit dem Muster hergestellt werden. Es handelt sich somit in den seltensten Fällen um botanisch korrekte Modelle. Der Grad an Wirklichkeitstreue liegt einzig im Ermessen des Modellierers.

Leider ist die Laufzeiteffizienz von GroIMP und der implementierten Bausteinen sehr begrenzt. Dies liegt vor allem an zwei Punkten. Zum einen ist die Anzahl der Berechnungen, die in den Bausteinen ausgeführt werden müssen, recht hoch. Allein schon zur Berechnung der Attribute sind in den meisten Fällen Funktionsauswertungen nötig. Im Anschluss daran wird die Geometrie des Bausteins bestimmt sowie die Positionen und die Winkel, an denen die vervielfältigten Instanzen platziert werden, berechnet. Diese Berechnungen vervielfältigen sich mit jeder Verzweigungsebene, entsprechend um die Anzahl der zu erzeugenden Instanzen. Dies macht sich besonders bemerkbar, wenn Xfrog-Modelle geladen werden sollen. Meist bestehen Xfrog-Modelle aus wesentlich mehr als vier Verzweigungsebenen, von denen manche Komponenten durchaus 200 Instanzen erzeugen, was einen direkten Import verhindert. Als Lösungsmöglichkeit bleibt nur das "Abspecken" der Xfrog-Modelle, entweder direkt in Xfrog oder im nach XL konvertierten Code.

Der zweite Punkt, der die Arbeit mit den Modellen sehr langsam werden lässt, ist das in GroIMP verwendete Darstellungsverfahren, das die Modelle nach jeder Änderung im View-Fenster neu berechnet. Nach jedem Verschieben, Skalieren oder Drehen entsteht so eine Pause, die, selbst bei verhältnismäßig kleinen Modellen, durchaus bis zu 60 Sekunden oder länger dauern kann.

Anschließend und abschließend erfolgt ein Ausblick auf mögliche Fortsetzungen der Anstrengungen zu Verbesserung und Erweiterung der Modellierungs- und Simulationssoftware GroIMP.

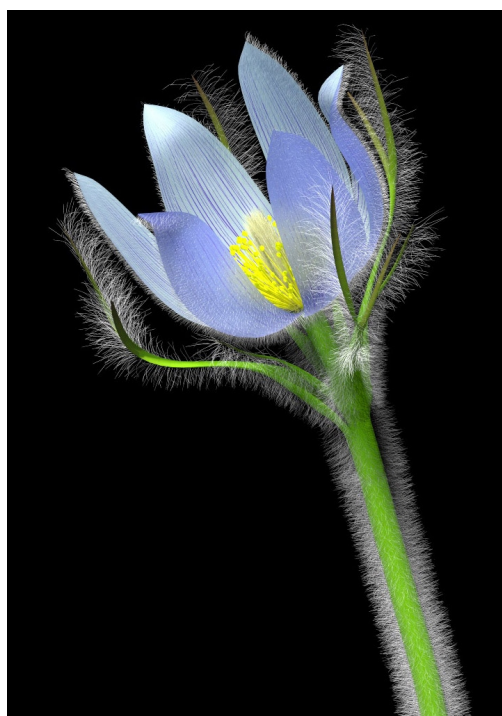
8.2 Ausblick und weitergehende Aufgaben

Aufgrund der Anzahl der implementierten Bausteine und ihrer Vielfältigkeit ergeben sich entsprechend viele weitere Ideen, wie diese Arbeit fortgeführt werden kann.

Anschließende Aufgaben sollten ein gesundes Mittelmaß zwischen den Erweiterungen der Bausteine hinsichtlich des Funktionsumfangs und der Vereinfachung der Bausteine haben, um die Effektivität der Modellierungsmethode und damit die Bedienbarkeit für den Benutzer weiterhin zu erhöhen.

Denkbare weitergehende Aufgaben an der GroIMP-Software sind im Zusammenhang mit Instanzierungsregeln:

- Ersetzen aller Attribute vom Typ *Komplexfeld*, wie in Abschnitt Attributtypen (5.1) beschrieben, durch Attribute vom *FloatToFloat* Typ. Werden die Werte so über Funktionen definiert, ist dies direkter und für den Benutzer intuitiver zu verstehen, als der Umweg über die Interpolation zwischen zwei Grenzwerten.
- Erweiterung des PhiBall-Bausteins zu einem "Phyllotaxis"-Baustein, bei dem die Phyllotaxisverteilung auf beliebige Rotationskörper angewendet werden kann.
- Zusammenlegen des Tree- und Horn-Bausteins, wie in Xfrog4 vorgenommen. Durch die von mir vorgenommene Erweiterung des Hydra-Bausteins, um die Möglichkeit, die Kurve mit dem *trajectory*-Attribut frei zu definieren, auf der die Instanzen verteilt werden, kann auch dieser Baustein integriert werden.
Die Funktionalität der drei Bausteine besteht grundlegend in der Anordnung von Objekten entlang einer Kurve und deren Manipulation. Die Integration in einen einzigen Baustein bringt deutlich mehr Flexibilität und reduziert die Anzahl verschiedener Objekte, mit denen die Nutzer umgehen müssen.
- Durch eine Manipulation der Objektflächen kann der Benutzer den Realitätsgrad der erzeugten Modelle erhöhen. Ein Beispiel dafür ist der Ansatz von Prusinkiewicz u. a. [47], die Behaarung auf Pflanzenoberflächen zu simulieren, s. Abbildung 8.1. Es ist durchaus ein Attribut, mit dem die Rauheit der Objektfläche variiert werden kann, um so den Eindruck von Rinde zu erzeugen, denkbar.
- Implementierung von Tropismen als eigenständigen Deformationsbaustein, der Phototropismus und Geotropismus zusammenfasst. Den Objekten kann somit eine Sensitivität geben werden, mit der die Auswirkungen von Umwelteinflüssen wie Licht, Wind oder Erdanziehung simuliert werden können. Siehe dazu die Xfrog-Komponenten (8.2) Hyper-Patch und Attractor.
- Implementierung eines Katalogs, aus dem der Benutzer vorgefertigte Strukturen laden kann, die dann als Grundlage eigener Modelle dienen können: Z. B. die 23



(a) wilder Krokus



(b) Mohnblume

Abbildung 8.1: Beispiel für Behaarung auf Pflanzenmodellen nach Prusinkiewicz [47]

verschiedenen Baumarchitekturen, die auf Hallé und Oldemann zurückgehen [21], grundlegende Blattformen sowie Blütenstrukturen würden sich als Prototypen anbieten. Will der Benutzer beispielsweise eine spezielle Kieferntyp modellieren, so kann er aus einem solchen Katalog die Grundstruktur eines Kieferntyps laden und diese dann nach Belieben modifizieren.

- Erhöhung der Beziehungen zwischen den kombinierten Bausteinen, so dass nicht mehr alle Attribute manuell eingegeben werden müssen und somit ggf. die Anzahl der Attribute reduziert werden kann - was letztlich die Arbeit wesentlich vereinfacht, gewünschte Ergebnisse könnten schneller erzielt werden.
Zur Zeit kann nur der Tree-Baustein eine solche "enge" Beziehung ausschließlich zu anderen Tree-Bausteinen eingehen. Dabei steuert das *BranchesGrowthScale*- und das *BranchesDense*-Attribut u. a. die Anzahl der erzeugten Instanzen des Tree-Bausteins in der zweiten Ebene.
- Erweiterung der GroIMP-Primitive Sphere, Box usw. um die Möglichkeit, die vom Arrange-Baustein bereitgestellten internen Variablen Kapitel C direkt in ihnen verwenden zu können: So z. B. den Radius einer Kugel von der Höhe ihrer Position abzuleiten. Ein entsprechender Aufruf könnte dann wie folgt aussehen: *Sphere("0.5*h")*, wobei *h* die interne Variable für die Höhe ist.

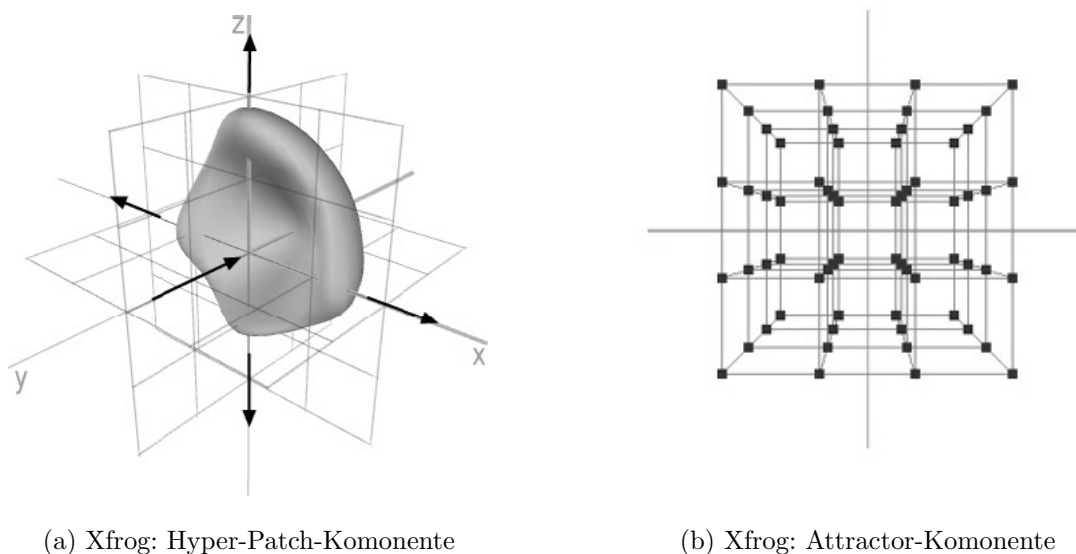
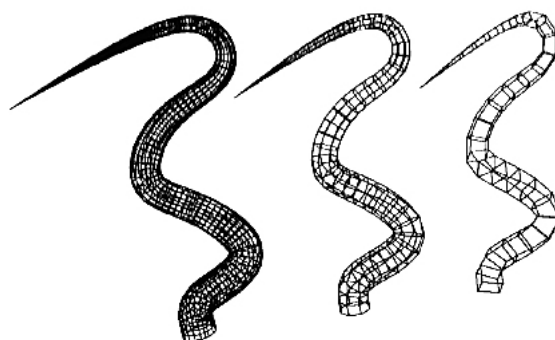


Abbildung 8.2: Xfrog-Deformationskomponenten

Erweitert man analog den Scale-, Color- und Rotate-Funktionsknoten von GroIMP, so können der BlockScale- und BlockColor-Baustein entfallen.

- Implementierung eines *Segments*-Attributs im Tree-Baustein sowie im Horn-Baustein, über den die Auflösung der erzeugten Instanz gesteuert werden kann (8.3), ohne aber die Anzahl der vervielfältigten Instanzen dadurch zu beeinflussen.

Abbildung 8.3: Verschiedene *Segments*-Einstellungen beim Horn-Objekt

- Ebenfalls denkbar ist eine Erweiterung des Arrange-Bausteins, hinsichtlich der Anordnung verschiedener Pflanzenspezies unter Beachtung jeweiliger Häufigkeits- bzw. Wahrscheinlichkeitsangaben oder eines Mindestabstands der Individuen einer Art zueinander. Auch die Objektgröße, sei es beispielsweise der Stamm- oder Kronendurchmesser von Bäumen, stellt einen entscheidenden Einflussparameter bei der Objektpositionierung da, der so beachtet werden könnte. Abbildung 8.4 zeigt eine

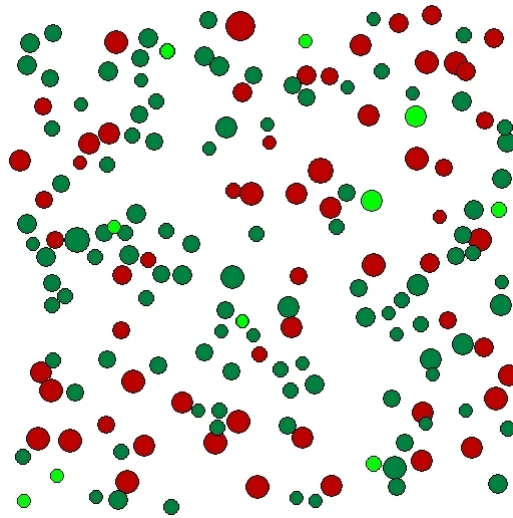


Abbildung 8.4: Beispielanordnung verschiedener Pflanzenspezies

solche Anordnung von drei verschiedenen Baumarten, die mit der bekannten Simulationssoftware SILVA¹ erzeugt wurde.

- Im Arrange-Baustein könnten durch die optionale Angabe von Häufungspunkten bei der Wahrscheinlichkeitsanordnung Konzentrationserhöhungen um einen Punkt bestimmt werden, um so beispielsweise eine kreisförmige Anordnung von Pilzen der zweiten Generation oder das Fallmuster von Blätter um einen Baum zu erzeugen.
- Erweiterung der Anordnungsmöglichkeiten im Arrange-Baustein bei den Kachelungen, hinsichtlich nichtperiodischer Verteilungen, variabler Kachelformen und verschiedener Kachelprototypen, wie bei Deussen [11] beschrieben.
- Eine Rückkopplung im Arrange-Baustein einbauen, mit der eine Art Nährstoffverbrauch zwischen den erzeugten Instanzen und dem *LocationParameter*-Feld simuliert werden kann, bei der im Einzugsgebiet von einem "großen Verbraucher" die Bodenparameter entsprechend stark sinken.

Im Weiteren kann das *LocationParameter*-Feld als Grundlage für die Anordnung der Objekte heran gezogen werden, was derzeit nur über das Dichtefeld möglich ist.

- Mit Xfrog können Animationen von Modellen erstellt werden. Interessant wäre ein Import der gesamten Animationssequenz. Dazu könnte in einem ersten Schritt der Xfrog-FileParser so erweitert werden, dass das zu ladende Model aus dem Animationszyklus ausgewählt werden kann, um anschließend die gesamte Animation zu importieren.

¹Lehrstuhl für Waldwachstumskunde der Technischen Universität München; Prof. Dr. H. Pretzsch und Dr. P. Biber

Auch ohne einen Import ist eine Animation der in Xfrog erzeugten Modelle eine interessante Aufgabe.

- Einbauen von flexibleren und stärker regelbasierten Verfahren (im Rahmen der Sprache XL) zur Spezifikation der Positions- und Dichtefelder, so dass der Benutzer von vorgefertigten Verteilungsmuster-Katalogen 5.12.1 unabhängig wird.
- Erweiterung des in GroIMP enthaltenen 2D-Subsystems (Graph-Fenster) um die Möglichkeit, die Struktur von Graphen interaktiv zu bearbeiten. Über Menüs können dann zu einem bestehenden Knoten erlaubte Nachbarknoten, verbunden über erlaubte Kanten, neu erzeugt werden, oder zwischen zwei ausgewählten Knoten können erlaubte Kanten erstellt werden. Mit einem solchen Struktureditor sollen insbesondere die Instanzierungsgraphen bearbeitet werden können.
- Zur Quantifizierung der Simulationen wie dem Waldmodell in Kapitel 7.4.2 auf Seite 117 fehlen Abfragemethoden, die auf den erzeugten Instanzen und nicht auf den Knoten im Graph arbeiten. So kann momentan die Summe der laufenden Holzmeter im Waldmodell, oder die Anzahl der Blätter in einem Baummodell nicht bestimmt werden, wie es für ein Objekt F im Graph durch die Abfragen: $count((* F *))$ und $sum((* F *).length)$ komfortabel möglich ist.

Eine Erweiterung der Level of Detail-Verfahren könnte in verschiedene Richtungen erfolgen:

- Zur weiteren Vereinfachung der Modelle ist es möglich, LOD-Verfahren zu implementieren, die auf der ganzen Modellstruktur arbeiten. Mit dem in Kapitel 6 beschriebenen Ansatz von Beaudoin und Keyser [6] kann so die Aststruktur vereinfacht werden.
- Erweiterung des implementierten Verfahrens um eine Methode, die die zu löschenden bzw. zu skalierenden Objekte optimal auswählt, so dass ein harmonisches Gesamtbild erzeugt wird, wie von Lintermann in [11] vorgeschlagen.
- Implementieren einer optionalen Koppelung der *number*-LOD-Methode und *scale*-LOD-Methode, um so die optimale Skalierung bzw. die optimale Objektanzahl automatisch zu bestimmen, um so den Nutzer von der Suche nach dem optimalen Verhältnis zu entlasten.

Um eine spürbare Erhöhung der Laufzeiteffizienz zu erreichen, wird man an Änderungen am Darstellungsverfahren von GroIMP nicht herum kommen. Durch geeignete Caching-Verfahren können viele Berechnungen eingespart werden, so braucht ein Modell nicht vollständig neu berechnet zu werden, wenn es nur gedreht oder verschoben wird.

Eine eher arbeitsintensive und langwierigere Aufgabe ist die Erstellung eines Pflanzenkatalogs, wie es ihn bei Xfrog oder andern Systemen zur Pflanzengenerierung gibt. Ein

solcher Katalog stellt eine Vielzahl vorgefertigter Modelle bereit, die es dem Benutzer ermöglichen, noch schneller seine Aufgaben zu erfüllen.

A Implementierungsstruktur

Der komplette Bausteinkatalog steht mit allen benötigten Komponenten im Paket 3D-Construction Set (3D-CS) der GroIMP-Software zur Verfügung.

Die vollständigen Java-Implementierungen der Bausteine sind im Quellcode-Verzeichnis *src/de/grogra/blocks* abgelegt. Die Klassennamen der Bausteine sind gleichlautend mit dem Baustein-Namen.

Für jeden Baustein existiert eine separate Datei, in der die zum Baustein gehörenden LOD-Methoden implementiert sind. Diese Dateien können durch den Bausteinnamen, gefolgt von dem Kürzel *LOD* identifiziert werden.

Beschreibung der Verzeichnisstruktur:

arrangeBlock In diesem Paket werden alle zum Arrange-Baustein gehörende Klassen zusammengefasst. Dazu gehören u. a. die Klassen für die Wahrscheinlichkeitsberechnungen, sowie die implementierten Halbtonverfahren.

functionParser Dieses Paket enthält den im Anhang C beschriebenen Funktionsparser mit den entsprechenden Funktionsimplementierungen.

Functions Dieses Verzeichnis enthält vier weitere Unterverzeichnisse, in denen für die Bausteine, dem Verzeichnisnamen entsprechend, benötigte Funktionen gespeichert sind. Dabei handelt es sich entweder um Kurven-Dateien: CPFPG Contour Format (*.con), oder um Funktionen, die über eine Datei im CPFPG Function Format (*.func) angegeben sind.

xFrogFileParser Die Implementierung des im Anhang C beschriebenen Xfrog File Parser befindet sich in diesem Paket.

B Funktionen

Dieses Kapitel beinhaltet eine Übersicht über alle vordefinierten Funktionen mit ihren jeweiligen Definitions- und Wertebereichen, sowie ihren Graphen. Diese Funktionen können in den in Abschnitt 5.1 beschriebenen Komplexfeldern eingesetzt werden.

1. **Menüeintrag:** From File

Beschreibung: Erlaubt es dem Benutzer, selbst definierte Funktionen über eine Datei im CPFG Function Format (*.func) einzubinden.

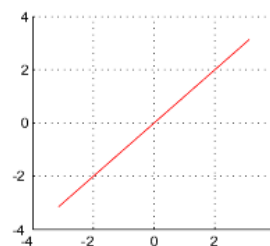
Menüeintrag: Identity

Funktion: $y=x$

2. **Beschreibung:** Identität

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $(-\infty, \infty)$



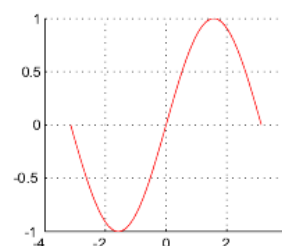
Menüeintrag: Sine

Funktion: $y=\sin(x)$

3. **Beschreibung:** Sinus von x

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $[-1,1]$



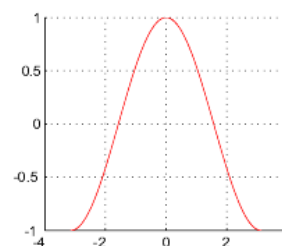
Menüeintrag: Cosine

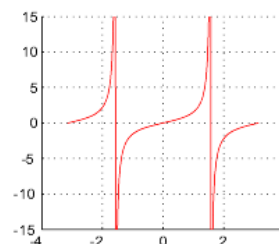
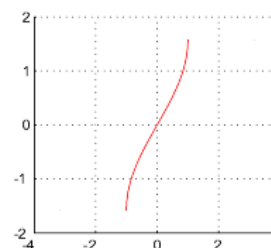
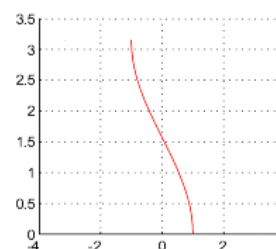
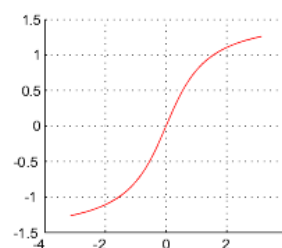
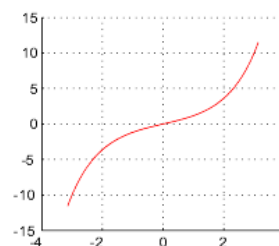
Funktion: $y=\cos(x)$

4. **Beschreibung:** Kosinus von x

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $[-1,1]$



Menüeintrag: TangentFunktion: $y=\tan(x)$ 5. Beschreibung: Tangens von x Definitionsbereich: $(-\infty, \infty)$ Wertebereich: $(-\infty, \infty)$ **Menüeintrag:** ArcsineFunktion: $y=\arcsin(x)$ 6. Beschreibung: Arkussinus von x Definitionsbereich: $[-1, 1]$ Wertebereich: $[-\pi/2, \pi/2]$ **Menüeintrag:** ArccosineFunktion: $y=\arccos(x)$ 7. Beschreibung: Arkuskosinus von x Definitionsbereich: $[-1, 1]$ Wertebereich: $[0, \pi]$ **Menüeintrag:** ArctangentFunktion: $y=\arctan(x)$ 8. Beschreibung: Arkustangens von x Definitionsbereich: $(-\infty, \infty)$ Wertebereich: $(-\pi/2, \pi/2)$ **Menüeintrag:** Hyperbolic sineFunktion: $y=\sinh(x)$ 9. Beschreibung: Sinushyperbolicus von x Definitionsbereich: $[-\pi, \pi]$ Wertebereich: $(-\infty, \infty)$ 

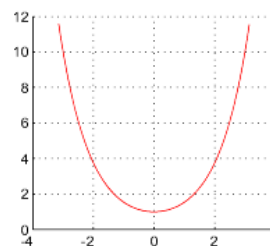
Menüeintrag: Hyperbolic cosine

Funktion: $y = \cosh(x)$

10. Beschreibung: Kosinushyperbolikus von x

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $[1, \infty)$



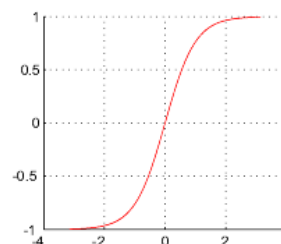
Menüeintrag: Hyperbolic tangent

Funktion: $y = \tanh(x)$

11. Beschreibung: Tangenshyperbolikus von x

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $[-1, 1]$



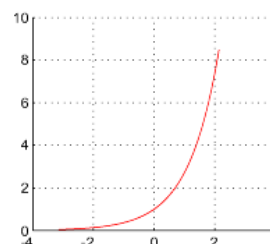
Menüeintrag: Exponential

Funktion: $y = \exp(x)$

12. Beschreibung: Exponential $y = e^x$

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $(0, \infty)$



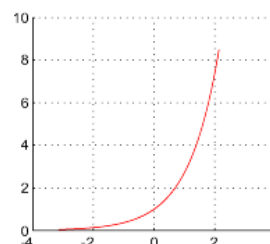
Menüeintrag: Exponential-1

Funktion: $y = \expm1(x)$

13. Beschreibung: Exponential $y = e^x - 1$

Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $(0, \infty)$



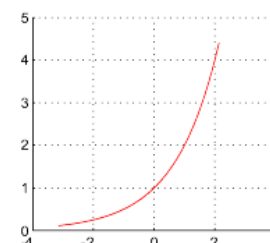
Menüeintrag: Exponential 2x

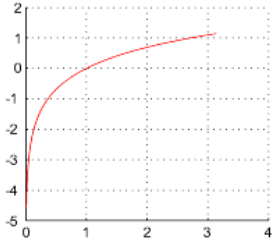
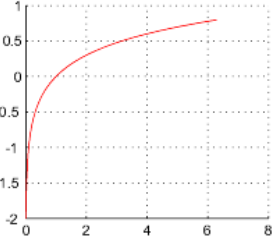
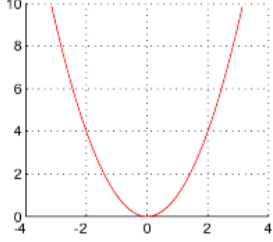
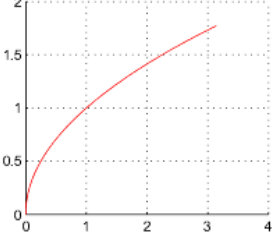
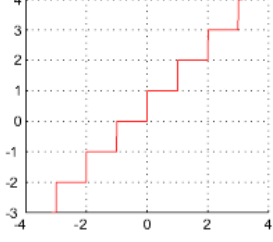
Funktion: $y = \text{pow}(2, x)$

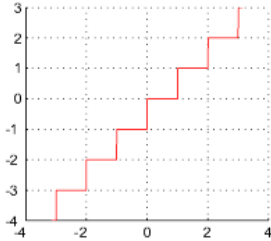
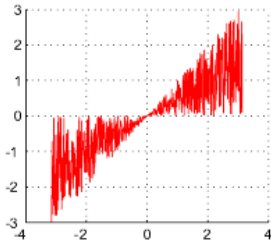
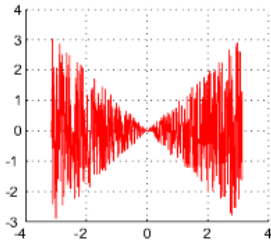
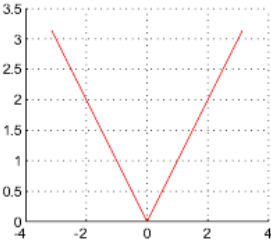
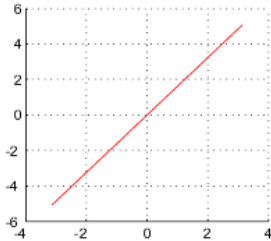
14. Beschreibung: Exponential $y = 2^x$

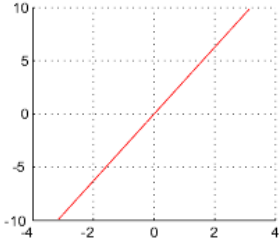
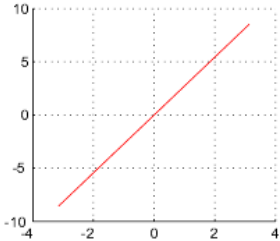
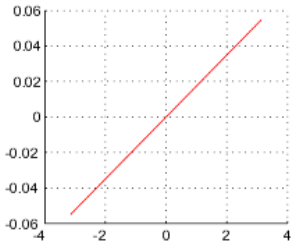
Definitionsbereich: $(-\infty, \infty)$

Wertebereich: $(0, \infty)$



15. **Menüeintrag:** Natural logarithm
 Funktion: $y=\log(x)$
 Beschreibung: Logarithmus zur Basis e von x
 Definitionsbereich: $[0,\infty)$
 Wertebereich: $[0,\infty)$
- 
16. **Menüeintrag:** Common logarithm
 Funktion: $y=\log_{10}(x)$
 Beschreibung: Logarithmus zur Basis 10 von x
 Definitionsbereich: $[0,\infty)$
 Wertebereich: $[0,\infty)$
- 
17. **Menüeintrag:** Square
 Funktion: $y=\text{sq}(x)$
 Beschreibung: Quadrat von x
 Definitionsbereich: $(-\infty,\infty)$
 Wertebereich: $[0,\infty)$
- 
18. **Menüeintrag:** Square root
 Funktion: $y=\text{sqrt}(x)$
 Beschreibung: Quadratwurzel von x
 Definitionsbereich: $[0,\infty)$
 Wertebereich: $[0,\infty)$
- 
19. **Menüeintrag:** Ceil
 Funktion: $y=\text{ceil}(x)$
 Beschreibung: Kleinste ganze Zahl von x abgerundet
 Definitionsbereich: $(-\infty,\infty)$
 Wertebereich: $(-\infty,\infty)$
- 

20. **Menüeintrag:** Floor
 Funktion: $y = \text{floor}(x)$
 Beschreibung: Größte ganze Zahl von x aufgerundet
 Definitionsbereich: $(-\infty, \infty)$
 Wertebereich: $(-\infty, \infty)$
- 
21. **Menüeintrag:** Rndabs
 Funktion: $y = \text{rndabs}(x)$
 Beschreibung: Zufallszahl zwischen $[0, x]$; $y = \text{Random} * x$
 Definitionsbereich: $(-\infty, \infty)$
 Wertebereich: $[-x, x]$
- 
22. **Menüeintrag:** Rnd
 Funktion: $y = \text{rnd}(x)$
 Beschreibung: Zufallszahl zwischen $[-x, x]$; $y = (2 * x * \text{Random}) - x$
 Definitionsbereich: $(-\infty, \infty)$
 Wertebereich: $[-x, x]$
- 
23. **Menüeintrag:** Absolute value
 Funktion: $y = \text{abs}(x)$
 Beschreibung: Absoluter Betrag von x
 Definitionsbereich: $(-\infty, \infty)$
 Wertebereich: $[0, \infty)$
- 
24. **Menüeintrag:** Phi
 Funktion: $y = x * \text{phi}$
 Beschreibung: x mal Goldener Schnitt
 Definitionsbereich: $(-\infty, \infty)$
 Wertebereich: $(-\infty, \infty)$
- 

25. **Menüeintrag:** Pi
 Funktion: $y=x*\text{PI}$
 Beschreibung: x mal Pi
 Definitionsbereich: $(-\infty,\infty)$
 Wertebereich: $(-\infty,\infty)$
- 
26. **Menüeintrag:** E
 Funktion: $y=x*\text{E}$
 Beschreibung: x mal Eulersche Zahl
 Definitionsbereich: $(-\infty,\infty)$
 Wertebereich: $(-\infty,\infty)$
- 
27. **Menüeintrag:** Rad
 Funktion: $y=x*\text{PI}/180$
 Beschreibung: Umrechnung x ins Gradmaß
 Definitionsbereich: $(-\infty,\infty)$
 Wertebereich: $(-\infty,\infty)$
- 
28. **Menüeintrag:** custom
 Beschreibung: Erlaubt es dem Benutzer, selbst definierte Funktionen direkt einzugeben. Die Funktionen haben der Spezifikation in Anhang C zu genügen.

C Funktionsparser

Bei vielen der Bausteine besteht die Möglichkeit, bei einigen ihrer Attribute, den sog. Funktionsfeldern, frei definierte Funktionen einzugeben. Hierbei bedeutet frei natürlich nicht beliebig, sondern nur frei in einen bestimmten Rahmen.

Dieser Rahmen, dem die eingegebenen Funktionen entsprechen müssen, damit sie als "gültige" Funktion vom Funktionsparser akzeptiert werden, wird in diesem Kapitel beschrieben.

Der Begriff Funktionsparser wird der Mächtigkeit der Funktionsfelder nicht gerecht. Mit ihnen ist es nicht nur möglich, auf eine umfangreiche Bibliothek von mathematischen Funktionen und Konstanten zuzugreifen, es stehen ebenfalls Variablen zur Verfügung, die mit Werten der erzeugten Instanzen belegt sind. Der Funktionsumfang des Funktionsparser wurde gegenüber den vordefinierten Funktionen aus Anhang B noch einmal deutlich erweitert.

Als weiteres sind logische Ausdrücke innerhalb einer IF-THEN-ELSE-Konstruktion erlaubt, so ist diese Eingabe:

```
(i%2==0)? 0 : 1.5
```

mit:

```
wenn i modulo 2 == 0, dann 1, sonst 1.5
```

zu übersetzen. Die Anwendungsmöglichkeiten sind entsprechend vielfältig, so kann dies z. B. dazu genutzt werden, jede zweite von einem Vervielfältigungsobjekt erzeugte Instanz anders zu behandeln, sie stärker zu skalieren oder weniger zu drehen.

C.1 Grammatik

Alle Funktionseingaben müssen dieser Grammatik entsprechen, damit sie vom Funktionsparser ausgewertet werden können. Die Grammatik ist in der Datei `3D - CS/src/de/grogra/blocks/functionParser/expr.cup` zu finden, sie entspricht einer LALR(1) Grammatik, die als Quelle für den Parsergenerator JavaCup¹ dient. Für eine

¹Frei erhältlich (als Java-Quellcode) unter: <http://www.cs.princeton.edu/~appel/modern/java/CUP/>

Beschreibung der Funktionsweise, sowie den Aufbau des zugrunde liegenden Dateiformats, in der die Grammatik spezifiziert wird, sei an dieser Stelle auf [27] verweisen.

```
expr ::= expr + term |
       expr - term |
       ( expression ) ? expr : term |
       term

term ::= term * factor |
       term / factor |
       term % factor |
       term ^ factor |
       factor

factor ::= NUMBER |
         ( expr ) |
         POW( expr, expr ) |
         KGV( expr, expr ) |
         GCD( expr, expr ) |
         NSQRT( expr, expr ) |
         + factor | - factor |
         ID( expr ) |
         SIN( expr ) | COS( expr ) | TAN( expr ) |
         ASIN( expr ) | ACOS( expr ) | ATAN( expr ) |
         SINH( expr ) | COSH( expr ) | TANH( expr ) |
         FAC( expr ) |
         EXP( expr ) | EXP2( expr ) | EXPM1( expr ) |
         LOG( expr ) | LOG10( expr ) | LOG1P( expr ) |
         SQR( expr ) | SQRT( expr ) |
         CEIL( expr ) | FLOOR( expr ) |
         ABS( expr ) |
         FRAC( expr ) | TRUNC( expr ) |
         ROUND( expr ) |
         RND( expr ) | RNDABS( expr ) |
         SIGN( expr ) |
         PHI( expr ) |
         RAD( expr ) | DEG( expr ) |
         PI | EULER | RANDOM |
         X | I | IO | P | D |
```

```
H | HL |
N1 | N2 | N3
```

```
expression ::= disjunktion |
              expression == disjunktion |
              expression != disjunktion |
              expr == expr |
              expr != expr

disjunktion ::= konjunktion |
              disjunktion || konjunktion

konjunktion ::= negation |
              konjunktion && negation

negation ::= atom |
            ! negation

atom ::= relation |
        TRUE | FALSE

relation ::= ( expression ) |
            expr < expr |
            expr <= expr |
            expr > expr |
            expr >= expr
```

C.2 Beschreibung der Funktionen

Alle erlaubten Funktionen werden in dieser Tabelle beschrieben.

$a + b$	Addition
$a - b$	Subtraktion
$a * b$	Multiplikation
a/b	Division
$a\%b$	Modulo
a^b	Potenz, Kurzschreibweise
$POW(a, n)$	Potenz, a hoch n

$KGV(a, b)$	kleinstes gemeinsames Vielfaches
$GCD(a, b)$	(g reatest c ommon d ivisor) größter gemeinsamer Teiler
$NSQRT(x, n)$	n-te Wurzel von x
$ID(x)$	berechnet die Identität von x
$SIN(x)$	berechnet den Sinus von x (x im Bogenmaß)
$COS(x)$	berechnet den Kosinus von x (x im Bogenmaß)
$TAN(x)$	berechnet den Tangens von x (x im Bogenmaß)
$ASIN(x)$	berechnet den Arkussinus von x (<i>Ergebnis</i> im Bogenmaß)
$ACOS(x)$	berechnet den Arkuskosinus von x (<i>Ergebnis</i> im Bogenmaß)
$ATAN(x)$	berechnet den Arkustangens von x (<i>Ergebnis</i> im Bogenmaß)
$SINH(x)$	berechnet den Sinushyperbolikus von x
$COSH(x)$	berechnet den Kosinushyperbolikus von x
$TANH(x)$	berechnet den Tangenshyperbolikus von x
$FAC(x)$	berechnet die Fakultät von x
$EXP(x)$	Exponentialfunktion von x
$EXP2(x)$	Exponentialfunktion 2^x
$EXPM1(x)$	Exponentialfunktion $e^x - 1$
$LOG(x)$	berechnet den natürlichen Logarithmus von x
$LOG10(x)$	berechnet den dekadischen Logarithmus von x
$LOG1P(x)$	berechnet den natürlichen Logarithmus von $1 + x$
$SQR(x)$	berechnet das Quadrat von x
$SQRT(x)$	berechnet die Wurzel von x
$CEIL(x)$	berechnet die kleinste ganze Zahl von x abgerundet
$FLOOR(x)$	berechnet die größte ganze Zahl von x aufgerundet
$ABS(x)$	berechnet den absoluten Betrag von x
$FRAC(x)$	berechnet die Nachkommastellen von x
$TRUNC(x)$	berechnet den ganzzahligen Anteil von x
$ROUND(x)$	rundet x auf die nächstliegende ganze Zahl
$RND(x)$	Zufallswert im Intervall zwischen $-x$ und x
$RNDABS(x)$	Zufallswert im Intervall zwischen Null und x
$SIGN(x)$	liefert das Vorzeichen von x
$PHI(x)$	multipliziert x mit Goldenem Schnitt $\frac{\sqrt{5}+1}{2}$
$RAD(x)$	Umrechnung von x ins Bogenmaß $RAD(x) = x * \pi/180$
$DEG(x)$	Umrechnung von x ins Gradmaß $DEG(x) = x * 180/\pi$
PI	Konstante: $Pi = 3,1415926535897932384626433\dots$
$EULER$	Konstante: Eulersche Zahl $e = 2,71828\dots$
$RANDOM$	Zufallszahl im Intervall $[0,1]$

x	Funktionsvariable
i	aktuelle Objekt-Id, fortlaufende Identifikationsnummer
$i0$	id des Vaterobjekt
p	Freie Variable in Funktionsfeldern.
d	Positionierungswahrscheinlichkeit, wird vom Arrange-Baustein gesetzt
h	Höhe des aktuellen Objektes über dem "Meeresspiegel" (Null)
hl	Höhe des aktuellen Objektes im Bezug auf das Vaterobjekt (lokal)
$n1$	erster (aktuell gesetzter) Standortparameter
$n2$	zweiter (aktuell gesetzter) Standortparameter
$n3$	dritter (aktuell gesetzter) Standortparameter

Die zur Verfügung stehenden logischen Operationen:

$(A)?B : C$	IF-THEN-ELSE-Konstruktion: wenn A wahr, dann B, sonst C
$a == b$	Test auf Gleichheit zweier logischer oder mathematischer Ausdrücke. true wenn a gleich b , sonst false
$a != b$	Test auf Ungleichheit zweier logischer oder mathematischer Ausdrücke. true wenn a ungleich b , sonst false
$a b$	Logisches Oder
$a&& b$	Logisches Und
$!a$	Logische Negation von a
$TRUE$	Logische Konstante: <i>true</i>
$FALSE$	Logische Konstante: <i>false</i>
$a < b$	Test auf "kleiner als" zweier mathematischer Ausdrücke. true, wenn a kleiner als b , sonst false
$a <= b$	Test auf "kleiner gleich" zweier mathematischer Ausdrücke. true, wenn a kleiner gleich b , sonst false
$a > b$	Test auf "größer als" zweier mathematischer Ausdrücke. true, wenn a größer als b , sonst false
$a >= b$	Test auf "größer gleich" zweier mathematischer Ausdrücke. true, wenn a größer gleich b , sonst false

D Xfrog FileParser

Der Xfrog FileParser ermöglicht es, Xfrog-Dateien¹ (*.*xfr*) zu öffnen, die enthaltenen Modellspezifikationen nach GroIMP zu importieren und sie so mittels Instanzierungsregeln darzustellen. Der Import kann direkt auf zwei verschiedene Wege erfolgen:

In ein bereits bestehendes GroIMP-Projekt :

Dies geschieht, indem über *Objects -> Insert File* eine *xfr*-Datei ausgewählt wird.

Als neues GroIMP-Projekt :

Dazu wird zuerst im Hauptmenü *File*, danach der Eintrag *Open* ausgewählt. Die im Dialog selektierte Datei wird in ein neu angelegtes Projekt importiert.

In ein GroIMP-Projekt können mehrere Xfrog-Modelle geladen und nebeneinander verwendet werden, um so z. B. verschiedene Baumarten oder unterschiedliche Gräser zu kombinieren.

Wenn in einem Xfrog-Modell der Wert eines Attributs über die Standardgrenzen hinausgeht, wird er in GroIMP-Bausteinen auf den entsprechenden Grenzwert gesetzt. Dies kann durchaus zu größeren Unterschieden in den erzeugten Geometrien führen, die vorerst nur durch Änderungen im Xfrog-Modell behoben werden können.

Ebenfalls zu unterschiedlichen Geometrien kommt es, wenn in den GroIMP-Bausteinen die LOD-Methoden aktiviert sind. Um das Resultat nicht durch unbeabsichtigte Eingriffe seitens der LOD-Methoden zu verändern, müssen in jedem Baustein die *useLOD*-Attribute deaktiviert werden.

Eine weitere indirekte Möglichkeit Xfrog-Modelle zu importieren, ist es sie in XL-Code umzuwandeln und diesen Code anschließend in ein RGG-Projekt einzubinden. Dazu wird als ersten in GroIMP ein neues RGG-Projekt angelegt und im Fenster *XL Console* das Programm *xFrogToXL* gestartet. Das Programm erwartet zwei Parameter. Der erste Parameter legt das Ausgabeziel fest, *-s* steht für *save* und mit der Option *-t* (für *text*) wird der XL-Code in der Console ausgegeben. Im "save"-Modus erzeugt *xFrogToXL* eine RGG-Datei mit dem Namen des Xfrog-Modells und dem *.rgg*-Suffix und speichert sie in dem Verzeichnis, in dem sich das Xfrog-Modell befindet, ab. Mit dem zweiten Parameter

¹siehe dazu www.greenworks.de greenworks organic-software

wird der vollständige Pfad und der Dateinamen des zu konvertierenden Xfrog-Modells angegeben.

Der Programmaufruf lautet entsprechend wie folgt:

```
de.grogra.blocks.xFrogToXL.main(new String[] {"-t", "C:\\name.xfr"})
```

D.1 Grammatik

Die Xfrog-Dateien besitzen ein hierarchisches Klartext-Format und konnten somit entschlüsselt werden. Die extrahierte Grammatik ist in der Datei *3D – CS/src/de/grogra/blocks/xFrogFileParser/expr.cup* zu finden, sie entspricht einer LALR(1) Grammatik, die als Quelle für den Parsergenerator JavaCup ² dient. In [27] werden eine ausführliche Beschreibung der Funktionsweise, sowie der Aufbau des zugrunde liegenden Dateiformats, in dem die Grammatik spezifiziert werden muss, gegeben.

Auf ein Abdrucken der Grammatik wird an dieser Stelle aufgrund der Länge von rund 1500 Zeilen verzichtet.

²Frei erhältlich (als Java-Quellcode) unter: <http://www.cs.princeton.edu/~appel/modern/java/CUP/>

Literaturverzeichnis

- [1] M. Aono und T. L. Kunii; Botanical Tree Image Generation; IEEE Computer Graphics and Applications, 4(5):10–34, 1984
- [2] T. Asano; Digital Halftoning Algorithm based on Random Space Filling Curve; IEEE Int. Conf. on Image Proc., Lausanne 1996
- [3] M. Barnsley; Fractals everywhere; Academic Press, 1988
- [4] M. Barnsley und S. Demko; Iterated function systems and the global construction of fractals; Proceedings of the Royal Society, Atlanta, Georgia, 1984. School of mathematics, Georgia Institut of Technology
- [5] B. E. Bayer; An Optimum Method For Two-Level Rendition Of Continous- Tone Pictures; International Conference on Communications, Vol. 1, 1973
- [6] J. Beaudoin and J. Keyser; Simulation Levels of Detail for Plant Motion; Texas A&M University
- [7] D. Cohen; Computer Simulation of Biological pattern generation processes; Nature, (216):246–248, 1967.
- [8] O. Deussen und B. Lintermann, (egl); Bildschirmbegrünung; C't magazin für computer technik; Ausgabe Nr. 11 vom 19.05.2003, S. 202-205
- [9] X. Décoret, F. Durand, F. X. Sillion, and J. Dorsey; Billboard clouds for extreme model simplification; volume 22, ACM Press, 2003
- [10] S. Demko, L. Hodges und B. Naylor; Construction of Fractal Objects with Iterated Function Systems; Computer Graphics (Proceedings of SIGGRAPH 1985), 19(3):271–278
- [11] Oliver Deussen; Computergenerierte Pflanzen: Technik und Design digitaler Pflanzenwelten; Springer Berlin 2003, ISBN 3-540-43606-5
- [12] O. Deussen und B. Lintermann; A Modelling Method and User Interface for Creating Plants; Graphics Interface '97, Morgan Kaufmann Publishers, May 1997, S. 189–198
- [13] O. Deussen und B. Lintermann; Interactive Modeling of Plants; Feature Article, IEEE Computer Graphics and Applications, 1999 S. 7-8

-
- [14] O. Deussen und T. Strothotte; Computer-Generated Pen-and-Ink Illustration of Trees; SIGGRAPH 2000 Conference Proceedings, Computer Graphics, 34(4):13–18
- [15] O. Deussen, S. Hiller, K. Van Overfeld und T. Strothotte; Floating Points: A Method for computing Stippel Drawings; D. Duke S. Coquillart, Hrsg., Proc. Eurographics 2000, Interlaken, Blackwell Publishers Ltd, 2000, S. C41.
- [16] Q. Du, V. Faber und M. Gunzburger; Centroidal Voronoi Tessellations; Siam Review, Vol. 41(4):637–676, 1999
- [17] J. D. Foley, A. van Dam, St. K. Feiner, J. F. Hughes; Computer Graphics; Addison-Wesley Publishing Company, 1993
- [18] D. Fournier und B. Andrieu; A 3D architectural and process-based model of maize development. Annals of Botany, 1998
- [19] D. R. Fowler, P. Prusinkiewicz und J. Battjes; A Collision-based Model of Spiral Phyllotaxis; Computer Graphics, 26, 2, (July 1992), ACM SIGGRAPH, New York, S. 361–368.
- [20] M. Gervautz und C. Traxler; Representation and realistic Rendering of Natural Phenomena with Cyclic CSG-Graphs; The Visual Computer, Vol. 12, 62–74, 1995
- [21] F. Hallé, R. Oldeman und P. Tomlinson; Tropical Trees and Forests; Springer-Verlag, 1978
- [22] S. Hiller, O. Deussen; Voronoi-Relaxierung allgemeiner Objekte; Technische Universität Dresden
- [23] M. Holton; Strands, Gravity and Botanical Tree Imagery; Computer Graphics Forum, 13(1):57–67, 1994
- [24] H. Honda; Description of the form of trees by the parameters of a tree-like body: effects of the branching angle and the branch length on the shape of the tree-like body; Journal of Theoretical Biology, 31:331–338, 1971.
- [25] J. Fisher und H. Honda; Computer simulation of branching pattern and geometry in Terminalia (Combretaceae), a tropical tree; Botanical Gazette, 138:377–384, 1977
- [26] J. Fisher und H. Honda; Branch geometry and effective leaf area: a study of Terminalia-branching pattern. 1 Theoretical ideas; American Journal of Botany, 66:633–644, 1979
- [27] S. E. Hudson; CUP User's Manual: Graphics Visualization and Usability Center; Georgia Institute of Technology Modified by Frank Flannery, C. Scott Ananian, Dan Wang with advice from Andrew W.; July 1999 (v0.10j)

-
- [28] H. R. Kang; Digital Color Halftoning; Bellingham, Washington: SPIE Optical Engineering Press. 1999
- [29] O. Kniemeyer; Rule-based modelling with the XL/GroIMP software; H. Schaub, F. Detje, U. Brüggemann (eds.), The Logic of Artificial Life. Proceedings of 6th GWAL, Bamberg 14.-16. 4. 2004, AKA Akademische Verlagsges. Berlin 2004, 56-65
- [30] O. Kniemeyer; Programming in XL: A brief introduction; BTU Cottbus, Institut für Informatik, Lehrstuhl Praktische Informatik / Grafische Systeme
- [31] H. Koch; Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes; Acta Mathematica 30, 1906
- [32] W. Kurth; Vorlesungsskript Computergrafik; BTU Cottbus, Institut für Informatik Vorlesung WS 2003/04
- [33] W. Kurth; Growth Grammar Interpreter GROGRA 2.4: A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modelling. Introduction and Reference Manual; Berichte des Forschungszentrums Waldökosysteme der Universität Göttingen, Ser. B, Band 38, 1994
- [34] W. Kurth, G. Buck-Sorlin, O. Kniemeyer; Relationale Wachstumsgrammatiken: Ein Formalismus zur Spezifikation multiskalierter Struktur-Funktions-Modelle von Pflanzen; 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 (2006), S. 36-45
- [35] Lau, Daniel L und Arce, R. Gonzalo; Modern Digital Halftoning; New York, Basel: Marcel Dekker, Inc. 2001
- [36] A. Lindenmayer; Mathematical models for cellular interaction in development; J. Theoret. Biology, 18:280–315, 1968
- [37] P. Prusinkiewicz und A. Lindenmayer; The Algorithmic Beauty of Plants; Springer-Verlag, New York, 1990
- [38] B. Lintermann und O. Deussen; A Modelling Method and User Interface for Creating Plants; Computer Graphics Forum, 17(1):73–82, 1998
- [39] B. Lintermann und O. Deussen; Interactive Modeling of Plants; IEEE Computer Graphics and Applications, 19(1):56–65, 1999
- [40] B. Mandelbrot; Fractals: Form, Chance and Dimension; W. Freeman and Co., San Francisco, 1977
- [41] B. Mandelbrot; The Fractal Geometry of Nature; W. Freeman and Co., San Francisco, 1983

- [42] J. von Neumann; Theory of self-reproducing automata; University of Illinois Press, Urbana, 1966
- [43] P. Oppenheimer; Real Time Design and Animation of Fractal Plants and Trees; Computer Graphics (SIGGRAPH 1986 Proceedings), Jahrgang 20, S.55–64, 1986
- [44] H. Pretzsch; Modellierung des Waldwachstums; Parey Buchverlag Berlin 2001; ISBN: 3-8263-3377-2, S. 199-210
- [45] P. Prusinkiewicz; Modelling and visualization of biological structures; Proceedings of Graphics Interface 1993, S. 128–137
- [46] P. Prusinkiewicz, M. Hammel, J. Hanan; Visual Models of plant development; G. Rozenberg and A. Salomaa, editors, Handbook of formal languages Springer-Verlag, 1996
- [47] M. Fuhrer, H. W. Jensen, P. Prusinkiewicz; Modeling Hairy Plants; Proceedings of Pacific Graphics 2004, 217-226
- [48] C. Raskob; Schwerpunkt-Voronoi-Diagramme; Diplomarbeit im Fachbereich Informatik, Lehrgebiet Praktische Informatik VI, FernUniversität Hagen, 2004
- [49] W. Reeves und R. Blau; Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems; Computer Graphics (SIGGRAPH '85 Proceedings), Jahrgang 19, 1985, S. 313–322
- [50] P. de Reffye, C. Edelin, J. Françon, M. Jaeger und C. Puech; Plant Models Faithful to Botanical Structure and Development; J. Dill, Hrsg., Computer Graphics (SIGGRAPH 1988 Proceedings), Jahrgang 22, S. 151–158. ACM SIGGRAPH, 1988
- [51] M. Shebell. Modelling branching plants using attribute L-Systems; Diplomarbeit, Worcester Polytechnic Institute, 1986.
- [52] I. Sutherland; Sketchpad – A Man-Machine Graphical Communication System; Proceedings of the Spring Joint Computer Conference, May 1963
- [53] S. Todd und W. Latham; Evolutionary Art and Computers; Academic Press, London, 1992
- [54] S. Ulam; Pattern of growth of figures: Mathematical aspects; G. Keps, Hrsg., Module, Proportion, Symmetry, Rhythm. Braziller, New York, 1966
- [55] R. Ulichney; Digital Halftoning; Cambridge: MIT Press. 1987
- [56] H. Vogel; A Better Way to Construct the Sunflower Head; Mathematical Biosciences, 44:179–189, 1979