

Bachelorarbeit
in der Angewandten Informatik
Nr. AI-2017-BA-027

Prozedurale Generierung von 3D-Bäumen

Martin Lesser
Matrikelnummer: 10272019
mlesser@gmx.net

Erfurt, 19. Oktober 2017

Prof. Dr. Jörg Sahm
Dipl.-Math. Anja Haußen

Thema der Arbeit

Prozedurale Generierung von 3D-Bäumen

Stichworte

prozedurale Generierung, L-System, Turtle-Grafik, OpenGL

Kurzfassung

Der Aufwand zur Entwicklung von Videospielen nimmt rasant zu. Jedes neue Spiel wird anhand seiner Vorgänger bemessen und soll dabei spektakulärer und umfangreicher werden. Um diesen hohen Ansprüchen gerecht zu werden, sucht man nach Möglichkeiten Inhalte einfacher und schneller zu generieren ohne dass diese an Qualität verlieren. Eine Methode dies zu erreichen ist die Technik der prozeduralen Generierung. In dieser Arbeit werden verschiedene Techniken der prozeduralen Synthese aufgezeigt und es wird speziell auf das L-System zur Generierung dreidimensionaler Bäumen eingegangen. Das Konzept von Aristid Lindenmayer und Przemyslaw Prusinkiewicz (1990) wird aufgegriffen, beleuchtet und umgesetzt, wobei der Fokus auf der Realisierung der Turtle-Grafik als dreidimensionales Polygon-Objekt liegt.

Title of the paper

Procedural generation of 3d-trees

Keywords

procedural generation, L-system, turtle graphics, OpenGL

Abstract

The effort to develop video games is increasing drastically. Every new game is measured according to its predecessors and is expected to become more extensive and exciting. In order to live up to these expectations the industry is looking for new ways to generate content faster and easier but without the loss of quality. One method to achieve this is procedural content generation. In this thesis diverse techniques of procedural synthesis are examined with special focus on L-systems to generate three-dimensional trees. The conception of Aristid Lindenmayer and Przemyslaw Prusinkiewicz (1990) will be explained, explored and implemented. The focus lies on the implementation of the turtle graphics as a three-dimensional object.

Inhaltsverzeichnis

Abbildungsverzeichnis	VI
Listings	VIII
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.3 Aufbau der Arbeit	2
2 Prozedurale Synthese	3
2.1 Definition	3
2.2 Methoden	4
2.2.1 Rauschen	4
2.2.2 Fraktale	5
2.2.3 Partikel-Systeme	6
2.2.4 Voronoi-Diagramm	6
2.2.5 Tiling	7
2.2.6 Künstliche Intelligenz	7
2.3 Zusammenfassung	8
3 Ersetzungssysteme	9
3.1 Thue-System	9
3.1.1 Einführung	9
3.1.2 Definition	10
3.2 Lindenmayer-System	10
3.2.1 Einführung	10
3.2.2 Definition	10
3.2.3 Turtle-Grafik	12
3.2.4 Verzweigte Strukturen	13
3.3 Zusammenfassung	13
4 Anwendungsgebiete und Beispiele	15
4.1 Videospiele	15
4.1.1 Level und Karten	15
4.1.2 Modelle	15
4.1.3 Texturen	16
4.1.4 Landschaften	16
4.1.5 Audio	17
4.1.6 Texte	17
4.2 Filme	18
4.3 Medizin	18

4.4 Zusammenfassung	19
5 Stand der Technik	20
5.1 SpeedTree	20
5.1.1 SpeedTree for Games	20
5.1.2 SpeedTree Modeler	21
5.2 PlantFactory	21
5.3 Weitere Systeme	22
5.3.1 Xfrog	22
5.3.2 Flora3D	22
5.3.3 Virtual Laboratory/L-Studio	22
5.4 Vergleich der Softwarelösungen	22
5.5 Fazit	23
6 Konzept	24
6.1 3D-Turtle	24
6.2 Mesh-Generierung	25
6.3 Optimierung	26
6.4 Blattgenerierung	26
6.5 Texturierung	27
6.6 Aufbau und Ablauf des Gesamtsystems	27
6.7 Zusammenfassung	28
7 Implementierung	30
7.1 PyOpenGL und PyGame	30
7.2 Einlesen der Grammatik-Datei	30
7.3 Text-Generator	31
7.3.1 Besondere Sprachkonstrukte	31
7.3.2 Nachverarbeitung	32
7.4 Mesh-Generator	32
7.4.1 Erzeugung des Skelett-Baums	32
7.4.2 Blattgenerierung	33
7.4.3 Erzeugung des Polygon-Modells	33
7.4.4 Rotation des Segments	33
7.5 Indize-Liste	34
7.6 Textur-Koordinaten	35
7.7 Scene-Renderer	35
7.8 Draw-Funktionen	36
7.9 Zusammenfassung	36
8 Ergebnisse	38
8.1 Steuerung, Speichern und Export	38
8.2 Grammatiken und Geometrien	38
8.2.1 Tree01-Grammatik	38
8.2.2 Tree02-Grammatik	40
8.2.3 Conifer-Grammatik	41
8.2.4 Bush-Grammatik	42
8.3 Variationen und Iterationsstufen	43

8.4 Bewertung	44
9 Fazit und Ausblick	45
9.1 Fazit	45
9.2 Ausblick	45
9.2.1 Grafisches Benutzer-Interface	45
9.2.2 Generierung von Level-Of-Details	45
9.2.3 Windanimation	45
9.2.4 Wachstumssimulation	46
Literaturverzeichnis	IX
Anhang	XIII
A Inhalt der DVD	XIV
A.1 Abbildungen der Ergebnisse	XIV
A.2 Abschlussarbeit	XIV
A.3 Kurzfassungen	XIV
A.4 Literaturquellen	XIV
A.5 Poster	XIV
A.6 Präsentation	XIV
A.7 Source-Code	XV
B Ausführungsanleitung	XVI
B.1 Python und Editor	XVI
B.2 PyGame	XVI
B.3 PyOpenGL	XVI
Selbstständigkeitserklärung	XVII

Abbildungsverzeichnis

2.1	Spieler erzeugt eine Kreatur im Spore-Editor (Quelle: Youtube [You13])	4
2.2	Zweidimensionaler Perlin Noise (Quelle: Wikipedia Foundation [wik05b])	5
2.3	Beispiele für Fraktale Strukturen in der Natur. Links: Romanesco (Quelle: itwissen.info [itw16]). Mitte: Farn (Quelle: clearingcomplexity.files.wordpress.com [cle11]). Rechts: Blumenkohl (Quelle: de.academic.ru [Aca]).	5
2.4	Abstrakte Fraktale. Links: Mandelbrot (Quelle: Wikipedia Foundation [wik05a]). Mitte: Julia Menge (Quelle: www.achim-und-kai.de [Ach]). Rechts: Koch-Schneeflocke (Quelle: Wikipedia Foundation [wik06]).	6
2.5	Partikelsystem, das Feuer simuliert (Quelle: Wikipedia Foundation [wik07]))	6
2.6	Generiertes Spielfeld der Siedler von Catan (Quelle: www.profeasy.de [pro])) . . .	7
3.1	Term-Baum (Entnommen aus Axel Thue (1910) [Thu10])	9
3.2	Kochsche Schneeflocke (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	11
3.3	Erzeugung mehrere Generationen eines L-Systems (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	11
3.4	Interpretation eines Turtle-Strings (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	12
3.5	Quadratische Koch-Insel (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	13
3.6	L-Systeme und ihre grafische Darstellung (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	14
4.1	Spiele mit Tile-Grafik. Links: Super Mario Bros mit Seitenansicht (Quelle: cdn.voxcdn.com [vox]). Mitte: Rogue mit Top-Down-Ansicht (Quelle: www.mobygames.com [mob]). Rechts: UFO: Enemy Unknown mit isometrischer Ansicht (Quelle: tweakpc.de [twe]).	15
4.2	CityEngine (Quelle: d2.alternativeto.net [alta])	16
4.3	Prozedurale Texturen. Links: Vase mit Perlin Noise Textur (Quelle: entnommen aus Ken Perlin (1985) [Per85]). Rechts: Stein-Fließen-Textur mittels Voronoi-Textur erstellt (Quelle: d2.alternativeto.net [altb])	16
4.4	Generierte Landschaft in No Man's Sky (Quelle: sm.ign.com [IGN])	17
4.5	Menschen-Massen-Simulation mit MASSIVE (Quelle: billdesowitz.com [bil]) . . .	18
5.1	Generation eines Baumes mit der Software SpeedTree (Quelle: ytmpg.com [yti]) .	21
5.2	Generation eines Baumes mit der Software PlantFactory (Quelle: zedspace.com [zed])	21
5.3	Vergleich verschiedener Softwarelösungen zur Generierung von 3D-Bäumen (Quelle: eigene Darstellung)	23

6.1	Rotationen der Turtle (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])	24
6.2	Skelett-Baum (Quelle: eigene Darstellung)	25
6.3	Berechnung der Punkte kreisförmig um einen Punkt des Skelett-Baums (Quelle: eigene Darstellung)	26
6.4	Geometrie eines Blatts (Quelle: eigene Darstellung)	27
6.5	Komponenten des Systems nach dem Model-View-Controller-Konzept geordnet (Quelle: eigene Darstellung)	28
6.6	Ablauf des Systems (Quelle: eigene Darstellung)	29
7.1	Zwei Segmente. Das äußere Segment (sechs Seiten) beinhaltet die grünen markierten Punkte und wird als <i>current_segment</i> bezeichnet. Das innere Segment (drei Seiten) beinhaltet die blauen Punkte und wird als <i>next_segment</i> bezeichnet. (Quelle: eigene Darstellung)	34
7.2	Die blauen Dreiecke werden als <i>triangles</i> , die roten Dreiecke als <i>inverse_triangles</i> bezeichnet (Quelle: eigene Darstellung)	35
7.3	Klassendiagramm des Gesamtsystems (Quelle: eigene Darstellung)	37
8.1	Skelett-Baum auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung) .	39
8.2	Drahtgitter-Modell auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung)	39
8.3	Texturiertes Modell auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung)	39
8.4	Drahtgitter-Modell auf Grundlage der Tree02-Grammatik (Quelle: eigene Darstellung)	40
8.5	Texturiertes Modell auf Grundlage der Tree02-Grammatik (Quelle: eigene Darstellung)	40
8.6	Drahtgitter-Modell auf Grundlage der Conifer-Grammatik (Quelle: eigene Darstellung)	41
8.7	Texturiertes Modell auf Grundlage der Conifer-Grammatik (Quelle: eigene Darstellung)	41
8.8	Drahtgitter-Modell auf Grundlage der Bush-Grammatik (Quelle: eigene Darstellung) .	42
8.9	Texturiertes Modell auf Grundlage der Bush-Grammatik (Quelle: eigene Darstellung)	42
8.10	Verschiedene Versionen (Abkürzung V) und Iterationsstufen (Abkürzung I) der Tree01-Grammatik (Quelle: eigene Darstellung)	43

Listings

7.1	Grammatik-Datei	30
7.2	Funktion zum Generieren eines Turtle-Strings	31
7.3	Algorithmus zur Berechnung der Segment-Scheitelpunkte	33
7.4	Algorithmus zur Berechnung der Indizes	34
7.5	Zuordnung der Textur-Koordinaten	35
8.1	Tree01-Grammatik	38
8.2	Tree01-Turtle-String	40
8.3	Tree02-Grammatik	40
8.4	Tree02-Turtle-String	41
8.5	Conifer-Grammatik	41
8.6	Conifer-Turtle-String	42
8.7	Bush-Grammatik	42
8.8	Bush-Turtle-String	43

1 Einleitung

1.1 Motivation

In den vergangenen Jahrzehnten ist die Zahl der Videospiele und deren Konsumenten stark gestiegen [biu17, 6]. Die Videospielindustrie ist stark gewachsen und professionell geworden. Gleichzeitig steigen die Erwartungen an aktuelle Videospiele zunehmend. Die Fans erwarten in der Regel, dass jedes neue Spiel in die Fußstapfen seiner Vorgänger tritt und somit großartiger und ausfeilter wird. Die Entwicklung heutiger Videospiele wird deshalb zunehmend aufwendiger und trifft dabei auf Schwierigkeiten in der Skalierbarkeit [HMVDVI13, 1]. Um den gewachsenen Anforderungen gerecht zu werden wird deshalb nach neuen Methoden gesucht die Inhalte einfacher, schneller und günstiger zu produzieren. Eine Möglichkeit dies zu erreichen ist Inhalte prozedural zu generieren.

Prozedurale Content Generierung (PCG), auch prozedurale Synthese genannt, bedeutet, dass Inhalte wie Modelle, Gelände, Texturen oder Sounds eines Videospiels mittels Algorithmen erzeugt werden und somit kein Entwickler dazu eingesetzt werden muss, um diese Inhalte „per Hand“ zu erschaffen. Hierbei besteht die Herausforderung darin die Algorithmen soweit heranzureifen, dass sie Ergebnisse mit ausreichender Qualität erzeugen. Aber auch Indie-Entwickler¹ nutzen die Methoden der prozeduralen Synthese, da diese aufgrund mangelnden Budgets und Personals mithilfe der PCG mehr Inhalte generieren können. Dies ermöglicht, dass selbst kleine Entwicklerteams den Spielern riesige Spielwelten anbieten können. Ein Beispiel ist der Indie-Entwickler „Hello Games“ [hel17], welcher mit nur 15 Entwicklern ein Videospiel namens „No Man's Sky“ [Hel16] erschaffen hat, in dem über 18 Trillionen² Planeten vom Spieler erkundbar sind und jeder Planet prozedural generierte Landschaften, Flora und Fauna bietet.

1.2 Zielsetzung

Ziel dieser Arbeit ist es aufzuzeigen wie man dreidimensionale Bäume mittels Lindenmayer-System³ generieren kann. Dabei soll gezeigt werden, dass ein solches System dazu geeignet ist Bäume aber auch andere Arten von Vegetation, unter Vorgabe bestimmter Parameter, zu erzeugen und dabei brauchbare Ergebnisse entstehen, die so in einem Videospiel verwendet werden können. Daraus ergeben sich folgende Zielvorgaben:

1. Deterministische Generierung von 3D-Modellen aus einer gegebenen Grammatik.
2. Die Bäume sollen Volumen besitzen und aus Polygonen bestehen.

¹Indie steht für Independent und bezeichnet in der Spielbranche Entwickler, die ohne Publisher und mit kleinem Budget Spiele produzieren.

²http://www.gamestar.de/artikel/no_mans_sky_3268828.html

³Das Lindenmayer-System ist ein Zeichenersetzungssystem, welches biologisches Wachstum imitiert. Das System wird in Abschnitt 3.2 erläutert.

3. Die Bäume sollen Blätter besitzen.
4. Die Modelle sollen optimiert sein, d. h. möglichst wenige Polygone enthalten.
5. Die erzeugten Modelle sollen rudimentär texturiert sein.

1.3 Aufbau der Arbeit

Im folgenden Kapitel wird auf das Konzept und die gängigen Methoden der prozeduralen Generierung eingegangen. Anschließend wird in Kapitel 3 das Lindenmayer-System vorgestellt, wobei auch geklärt wird wie man dieses mit der Turtle-Grafik verbinden kann, um verzweigte Strukturen zu erzeugen. In Kapitel 4 werden die Anwendungsgebiete der prozeduralen Generierung aufgezeigt und Beispiele gegeben, wo diese Verwendung finden. Die beiden professionellsten Softwarelösungen zur Generierung von 3D-Bäumen werden in Kapitel 5 beleuchtet. Außerdem stehen weitere Softwarelösungen im Fokus der Betrachtung, um den Stand der Technik in diesem Gebiet darzulegen. Die Konzeption eines Systems zur Generierung von 3D-Bäumen wird in Kapitel 7 erläutert. Darin werden der Aufbau und Ablauf des Systems erklärt, um aus einer Zeichenkette ein Modell zu erzeugen. Die Ergebnisse der praktischen Arbeit werden in Kapitel 8 gezeigt. Abschließend wird in Kapitel 9 ein Fazit aus dieser Arbeit gezogen und anschließend werden Möglichkeiten aufgezeigt, wie das System erweitert werden kann.

2 Prozedurale Synthese

Das Prinzip der prozeduralen Synthese besteht darin Inhalte mittels iterativer Prozeduren zu berechnen anstatt diese aus dem Speicher zu laden [KM06, 4]. Anhand der Parameter können die Prozeduren Inhalte erzeugen, die in Variation, Anzahl und Detailstufe beliebig variierbar sind. Dieses Kapitel klärt den Begriff der prozeduralen Generation und gibt einen Überblick über die am häufigsten verwendeten Techniken.

2.1 Definition

Prozedurale Synthese ist die Generierung von Inhalten (z.B. Texturen, Modellen, Level, Sound etc.) mittels Algorithmen, sodass ein Computer solche Inhalte mithilfe von Prozeduren erzeugen kann, anstatt diese von der Festplatte zu laden. Die Inhalte werden sonst von Entwicklern vorher manuell in mühsamer Arbeit erstellt. Hendrikx et al. [HMVDVI13] definieren PCG folgendermaßen:

„The main idea behind procedural content generation is that game content is not generated manually by human designers, but by computers executing a well-defined procedure.“ [HMVDVI13, 2]

Jedoch ist eine solche intuitive Definition nicht genau, denn Ansätze für prozedurale Algorithmen und deren erzeugten Inhalte sind sehr verschieden. So definieren Grafik-, Industrie- oder Videospieldesigner was man unter Inhalte versteht unterschiedlich und auch die Umsetzung der Algorithmen sind andere. Allerdings sind eindeutige Definitionen wohl nur in der Mathematik möglich [TKSY11, 1]. Dennoch versuchen Togelius et al. [TKSY11] klarzustellen was man unter PCG versteht bzw. nicht versteht. Dazu führen sie zwei Punkte auf, anhand derer sie ihren Standpunkt verdeutlichen. Als ersten Punkt weisen sie darauf hin, dass der User einen bestimmten Input dem PCG-Algorithmus übergibt. Allerdings wird dabei erwartet, dass daraufhin ein nicht trivialer Berechnungsaufwand erfolgt und das Ergebnis nicht genau vom User vorhergesehen wird. Als Gegenbeispiel verweisen Togelius et al. [TKSY11] auf das Videospiel „Spore“ [gam08b], bei dem der Spieler einen Editor bedienen kann, um eigene Kreaturen zu erschaffen. Allerdings erfolgt auf den Input des Spielers an den Editor unmittelbar das Ergebnis an genau der Stelle wo der Spieler die Änderung erreichen wollte (Siehe Abbildung 2.1). Deshalb wäre es falsch bei dieser Art von Content Generierung von prozedural zu sprechen. Man nennt diese Art offline Generierung, da der Spieler zuerst den Inhalt getrennt vom eigentlichen Spiel in einem Editor entwirft und speichert, um diese dann im eigentlichen Spiel zu laden und zu nutzen. Im Gegenzug erzeugt online Generierung Inhalte während des Spielens. Strategiespiele sind ein Beispiel dafür, da der Spieler Anweisungen gibt eine Siedlung zu errichten und das Spiel daraufhin Gebäude an bestimmten Plätzen platziert. Der Spieler kann bei Strategiespielen wie „Sim City“ [gam89] nicht genau bestimmen wie eine Stadt erbaut wird und sich entwickelt. Die Autoren argumentieren deshalb, dass auch hierbei nicht von PCG gesprochen werden kann, da



Abbildung 2.1: Spieler erzeugt eine Kreatur im Spore-Editor (Quelle: Youtube [You13])

der Spieler zumindest den Rahmen vorgibt wie das Spiel die Inhalte (z.B. Städtebau) realisieren soll. Allerdings ist die Erstellung der spielbaren Karten anhand von Parametern (z.B. Klimazone, Höhe, Landschaft, Ressourcen) in Strategiespielen eine Form von PCG. Als zweiten Punkt gehen die Autoren auf den Zufallscharakter von PCG ein und machen dabei klar, dass viele PCG-Systeme Pseudozufallszahlen nutzen, aber dies meist in einem beschränktem Rahmen, da die Ergebnisse nicht vollständig zufällig sein sollen (was zu schlechten Ergebnissen führen würde). Sie verdeutlichen, dass Zufall nicht zwingend Bestandteil von PCG sein muss. So ist das Weltraum-Simulationsspiel „Elite“ [gam84] ein Beispiel für deterministische PCG. Da das Spiel 1984 erschien und zu dieser Zeit die Hardwarebeschränkungen zu eng waren, um eine gigantische Weltraumsimulation mit tausenden von Planeten zu ermöglichen, musste man die Spielinhalte prozedural erzeugen. Das Spiel bestand somit aus einer Menge von Seeds, welche als Basis für die Generierung der Sonnensysteme dienten. Somit wurde PCG als Mittel zur Datenkomprimierung genutzt [TKSY11, 3]. Abschließend definieren Togelius et al. [TKSY11] den Begriff der prozeduralen Content Generierung genauer:

„We can therefore tentatively redefine PCG as the algorithmical creation of game content with limited or indirect user input.“ [TKSY11, 6]

Mit dieser Definition wird der Begriff eingegrenzt. Der Einfluss von User Input wird beschränkt, um somit Content Generierung wie sie in Editoren stattfinden auszuschließen. Weiterhin wird der Begriff „automatisch“ oder ähnliche Begriffe bewusst ausgelassen, um die Möglichkeit der zufälligen Generierung von Inhalten weder auszuschließen noch als obligatorischen Bestandteil von PCG festzulegen.

2.2 Methoden

2.2.1 Rauschen

Bei der Erzeugung von natürlichen Mustern, beispielsweise von Texturen für Holz oder Wolken, ist es wichtig diese natürlich aussehen zu lassen. Dazu setzt man Pseudozufallszahlen ein, allerdings ist es wichtig, dass die eingesetzten Funktionen zur Erzeugung dieser Pseudozufallszahlen deterministisch sind, d.h. bei gegebenen Parametern soll das Ergebnis immer gleich sein. So erreicht man, dass die erzeugten Texturen bei der Darstellung immer gleich aussehen

[Wal10, 8]. Funktionen, die diese Anforderung erfüllen, sind Rauschfunktionen. Die bekannteste ist die Perlin-Noise-Funktion, welche 1983 von Ken Perlin erfunden wurde [Per85]. Um aus Zufallszahlen kohärente Werte zu erzeugen, werden die Werte interpoliert z. B. mittels einer kubischen Interpolation, um so organische Verläufe, wie sie in Abbildung 2.2 zu sehen sind, zu erhalten [Wal10, 9]. Im Jahre 2001 entwickelte Ken Perlin eine verbesserte Version namens Simplex Noise [Sim]. Der Vorteil liegt vor allem in der besseren Rechenkomplexität, sodass die Berechnung für höhere Dimensionen weniger rechenintensiv ist [Sim, 1].

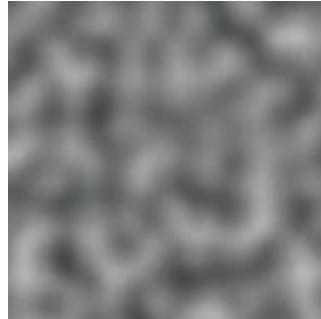
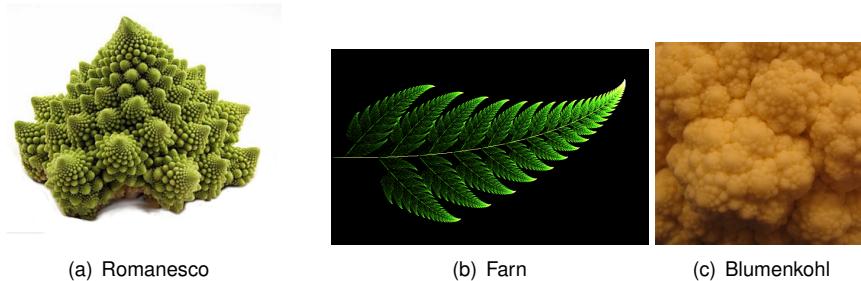


Abbildung 2.2: Zweidimensionaler Perlin Noise (Quelle: Wikipedia Foundation [wik05b])

2.2.2 Fraktale

In der Natur weisen viele Objekte fraktale Strukturen auf. Fraktal bedeutet Selbstähnlichkeit, sodass sich das Gesamtbild in seinen Einzelteilen wiederfindet. Der Begriff des Fraktales wurde 1975 von Benoît Mandelbrot geprägt [Man80]. Die klassischen Beispiele für Fraktale sind Romanesco, Blumenkohl oder Farne (siehe Abbildung 2.3). Aber auch viele weitere Naturerscheinungen weisen Selbstähnlichkeit auf z. B. Wolken, Küstenlinien oder Bäume. Wobei man zwischen statistischer und exakter Selbstähnlichkeit unterscheiden muss. Während Romanesco und Farn Beispiele für exakte Selbstähnlichkeit sind, da deren kleineren Bestandteile (fast) exakte Kopien des Größeren sind, bilden Bäume oder Wolken Beispiele für statistische Selbstähnlichkeit, da bei diesen die Statistik deren Geometrie selbstähnlich ist [Wal10, 10]. Es gibt aber auch abstraktere Fraktale, die mittels mathematischer Formeln erzeugt werden. Klassische Vertreter hierfür sind das Mandelbrot (nach dessen Erfinder Benoît Mandelbrot benannt), die Julia Menge oder die Koch-Schneeflocke (siehe Abbildung 2.4).



(a) Romanesco

(b) Farn

(c) Blumenkohl

Abbildung 2.3: Beispiele für Fraktale Strukturen in der Natur. Links: Romanesco (Quelle: it-wissen.info [itw16]). Mitte: Farn (Quelle: clearingcomplexity.files.wordpress.com [cle11]). Rechts: Blumenkohl (Quelle: de.academic.ru [Aca]).

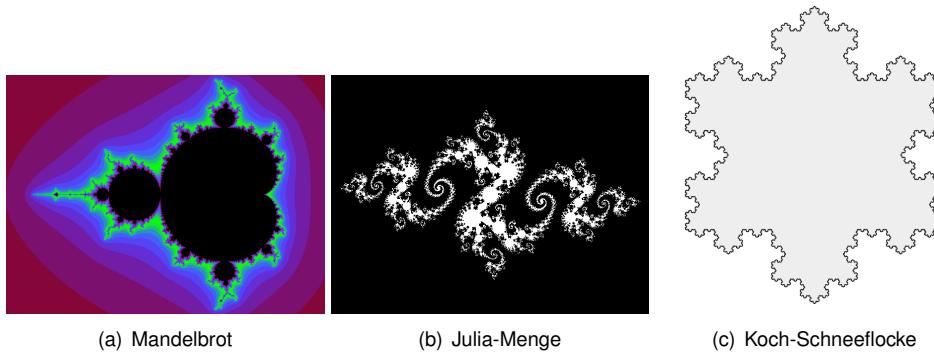


Abbildung 2.4: Abstrakte Fraktale. Links: Mandelbrot (Quelle: Wikipedia Foundation [wik05a]). Mitte: Julia Menge (Quelle: www.achim-und-kai.de [Ach]). Rechts: Koch-Schneeflocke (Quelle: Wikipedia Foundation [wik06]).

2.2.3 Partikel-Systeme

Im Jahr 1983 hat der Informatiker William Reeves eine Möglichkeit vorgestellt unscharfe Objekte wie Feuer oder Wolken darzustellen [Ree83]. Dies wurde erreicht, indem Oberflächen von Objekten nicht mittels Primitiven wie Dreiecken repräsentiert wurden, sondern als Wolke von einfachen Geometrien, den Partikeln (siehe Abbildung 2.5). Die Partikel sind nicht statisch, sondern verändern Form und Bewegung abhängig von der Zeit. Partikel durchleben einen Lebenszyklus, d. h. sie werden „geboren“, leben eine Zeit lang, wobei sie ihre Form verändern, sich bewegen und dann nach einer gewissen Zeit „sterben“.

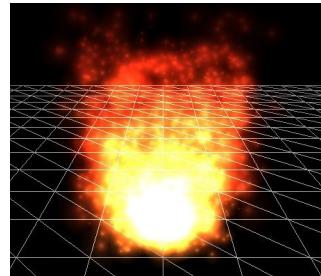


Abbildung 2.5: Partikelsystem, das Feuer simuliert (Quelle: Wikipedia Foundation [wik07]))

2.2.4 Voronoi-Diagramm

Bei einem Voronoi-Diagramm handelt es sich um die Zerlegung eines Raumes in einzelne Regionen, welche durch gegebene Punkte (auch Zentren genannt) definiert werden. Dabei enthält eine Region alle Punkte/Koordinaten, die dessen Zentrum am nächsten sind. Diese Diagramme wurden nach Georgi Feodosjewitsch Woronoi benannt, welcher ca. 1908 diese Art von Diagramm analysierte [Wor96, 6]. S. Worley kombinierte die Technik der Voronoi-Diagrammen mit Perlin-Noise und konnte zeigen wie man damit zelluläre fließenähnliche Texturen wie Haut, Eis, Gestein oder Krater generieren kann [Wor96, 1].

2.2.5 Tiling

Tiling ist die Aufteilung von Inhalten in einzelne Komponenten, welche miteinander beliebig verbunden werden können, um so ein Ganzes wie Levels, Karten oder Texturen zu erzeugen. Dies ist eine der ältesten Techniken der prozeduralen Generierung. Sie wird vorwiegend in Spielen eingesetzt. Beispielsweise bei dem Brettspiel „Die Siedler von Catan“ [Kla95], hier beschreiben die Tiles Landfelder wie Wald, Weideland, Ackerland usw. Die Tiles haben eine sechseckige Form und werden zufällig nebeneinander gelegt, um eine Karte zu „generieren“ (siehe Abbildung 2.6).



Abbildung 2.6: Generiertes Spielfeld der Siedler von Catan (Quelle: [www.profeasy.de \[pro\]](http://www.profeasy.de/pro/)))

2.2.6 Künstliche Intelligenz

Bisher wurden die Techniken der künstlichen Intelligenz (KI) größtenteils zur Evaluierung der Ergebnisse von prozeduraler Generierung eingesetzt. Künstliche Intelligenz wurde dabei so eingesetzt, dass die generierten Inhalte bewertet wurden und falls die Bewertung unter einem gewissen Schwellwert lag, wurde der Inhalt verworfen, neu generiert und wieder bewertet. Dies wiederholt sich solange bis der Inhalt akzeptiert wird [MJ15]. Vor allem Spielkarten werden somit bewertet, wobei nicht nur der Spaßfaktor als Bewertungsfaktor eine Rolle spielt, sondern auch die Lösbarkeit des Levels. Es gibt aber auch Bestrebungen KI einzusetzen, um Inhalte zu generieren. Die Software ANGELINA¹ von Michael Cook von der Universität Falmouth generiert Videospiele. Dabei entstehen Platformer² oder 3D-Labyrinthe. ANGELINA stellt dazu verschiedene Spielkomponenten wie Texturen, Modelle, Sounds, aber auch Spielregeln zusammen und bewertet die Kombination. Liegt die Bewertung unter einem Grenzwert, dann stellt ANGELINA eine neue Kombination zusammen und bewertet diese. Dieser Prozess wird solange wiederholt bis eine ausreichende Bewertung erzielt wird, um möglichst gute Spiele/Spielinhalte zu generieren [Hul15]. Neueste Forschungen versuchen maschinelles Lernen mit PCG zu verbinden. Man trainiert ein Modell des maschinellen Lernens mit Daten von existierenden Spielen und nutzt das trainierte Modell daraufhin, um neue Spiele oder einzelne Spielinhalte zu generieren [SSG⁺17, 1].

¹<http://www.gamesbyangelina.org/>

²Auch Jump 'n' Run genannt

2.3 Zusammenfassung

In diesem Kapitel wurde auf den Begriff der prozeduralen Generierung eingegangen und von anderen Methoden zur Erstellung von Inhalten abgegrenzt. Danach sind die meist verwendeten Methoden erklärt worden. Diese ermöglichen prozedurale Inhalte oder Spiele zu erstellen und sind die Grundlage für fortgeschrittenere Techniken.

3 Ersetzungssysteme

In diesem Kapitel werden zwei Ersetzungssysteme vorgestellt. Zuerst wird auf das Thue-System eingegangen und anhand dieses Systems wird der Begriff des Ersetzungssystems erklärt. Darauf aufbauend begründet sich das L-System, welches als zweites vorgestellt und definiert wird.

3.1 Thue-System

3.1.1 Einführung

Textersetzungssysteme gehören zu den regelbasierten Systemen, da auch diese auf Basis einer Regelmenge von einem Anfangszustand Regeln solange anwenden bis der gewünschte Endzustand erreicht ist. Das System geht auf Axel Thue (1863–1922) zurück [Tho10] und findet in der Informatik viele Anwendungen. Axel Thue beschrieb in seinem Werk „Die Lösung eines Spezialfalles eines generellen logischen Problems“ [Thu10] das Ersetzungssystem folgendermaßen:

„Es kann eintreffen, dass man aus einem beliebigen Begriffe [sic] A einer Begriffs-kategorie P und aus einem beliebigen Begriffe [sic] B einer Begriffskategorie Q durch ein gewisses Verfahren oder Operation θ einen Begriff C einer Begriffskategorie R bilden kann.“ [Thu10]

Mit Begriff meint Thue einen Term und mit Begriffskategorie den Typ. Er beschreibt dabei wie man Terme mithilfe anderer zu neuen Termen umformen kann, wobei dieser Prozess iteriert werden kann. Das Ergebnis lässt sich als Term-Baum darstellen (siehe Abbildung 3.1).

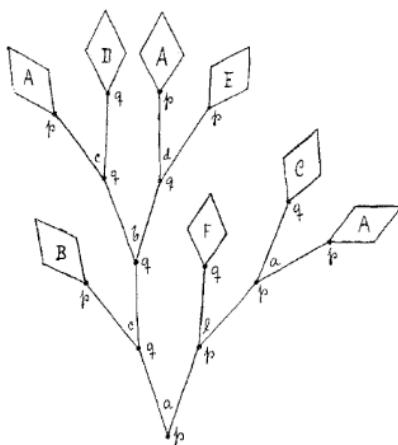


Abbildung 3.1: Term-Baum (Entnommen aus Axel Thue (1910) [Thu10])

Die atomaren Terme sind die Blätter (A-F), welche durch Funktionen (a-f) zu neuen Typen (p und q) gebildet werden. Man nennt die atomaren Terme auch Axiome und sie sind die Ausgangslage für die Term-Ersetzungen. Wenn ein solches System aus einem Alphabet, welches alle zulässigen Zeichen enthält, und einer Menge von Regeln besteht, so nennt sich dies Thue-System, sofern alle Regeln auch reflexiv angewandt werden dürfen. Trifft dies nicht zu und die Regeln gelten nur in eine Richtung, so spricht man von einem Semi-Thue-System.

3.1.2 Definition

Formal lässt sich ein Semi-Thue-System folgendermaßen beschreiben: Jede Regel in der Menge der Ersetzungsregeln S (Substitutionen) hat die Form $u \rightarrow v$. Wobei u und v über dem Alphabet Σ gebildet werden. Ein Semi-Thue-System ist dann als Tupel (Σ, S) definiert. Mit einem Axiom als Grundlage kann man Ableitungen unter Hilfenahme der Substitutionsregeln erzeugen. Mit dieser Arbeit im Jahr 1914 ebnete Axel Thue den Weg für Noam Chomsky, welcher 1956 die Chomsky Hierarchie definierte [Cho56]. Chomsky verallgemeinerte dazu das Semi-Thue-System zu einem allgemeinen Textersetzungssystem, wobei auch ganze Zeichenketten ersetzt werden dürfen, welche als Grammatik bezeichnet wird. Die Hierarchie klassifiziert dabei den Freiheitsgrad der Produktionsregeln verschiedener Grammatiken.

3.2 Lindenmayer-System

3.2.1 Einführung

Bei dem Lindenmayer-System handelt es sich um ein Ersetzungssystem, welches 1968 vom ungarischen Biologen Aristid Lindenmayer erfunden wurde [Lin68]. Dieses System, kurz L-System genannt, besteht aus einem Alphabet und einer Menge von Produktionsregeln. Ausgehend von einem Startsymbol oder Symbolgruppe werden die Symbole anhand der Produktionsregeln ersetzt. Allerdings findet die Ersetzung parallel quasi gleichzeitig statt [PL90, 1]. Dies steht im Kontrast zu den klassischen Chomsky Grammatiken, welche sequentiell arbeiten. Die Ersetzung findet parallel statt, da mit diesem System das Pflanzenwachstum nachempfunden werden soll [PL90, 3]. Da die Produktionsregeln mehrfach bei der Ersetzung angewendet werden, erzeugen L-Systeme fraktale Strukturen. Das klassische Beispiel hierfür ist die Koch-Kurve (siehe Abbildung 3.2). Die Erzeugung von diesem Fraktal wurde auch von Mandelbrot wie folgt beschrieben:

„One begins with two shapes, an initiator and a generator. The latter is an oriented broken line made up of N equal sides of length r . Thus each stage of the construction begins with a broken line and consists in replacing each straight interval with a copy of the generator, reduced and displaced so as to have the same end points as those of the interval being replaced.“ [PL90, 1]

3.2.2 Definition

Formal definieren lassen sich L-Systeme als Tripel: $G = (V, \omega, P)$, wobei V das Alphabet ist, also die Menge aller Zeichen, ω ist das Axiom, also das Startsymbol von dem ausgehend die

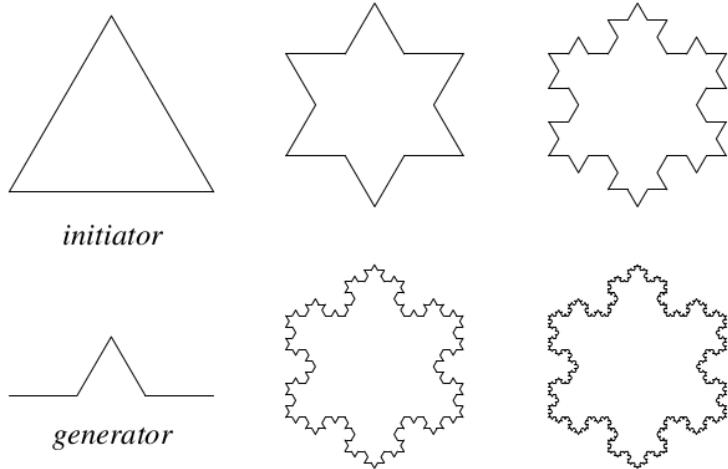


Abbildung 3.2: Kochsche Schneeflocke (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

Ersetzungen beginnen, P ist die Menge der Produktionsregeln, wobei eine Produktionsregel folgende Form aufweist: $a \rightarrow \chi$, a nennt man den Vorgänger und χ den Nachfolger. Wenn für alle Produktionsregeln einer Grammatik gilt, dass alle Vorgänger nur aus einem einzelnen Zeichen bestehen, dann ist die Grammatik kontextfrei, d.h. bei dem Ersetzen von Zeichen muss nicht auf den Kontext, also die Zeichen die davor oder dahinter stehen, geachtet werden. Wenn es außerdem für jedes Zeichen w genau eine Produktionsregel gibt mit $a \rightarrow \chi$ und $a = w$, dann ist die Grammatik deterministisch, bei der Ersetzung von Zeichen gibt es also nur genau eine Produktionsregel, die dieses Zeichen ersetzen kann.

Ein Beispiel soll die Funktionsweise von L-Systemen verdeutlichen: $G = (V, \omega, P)$, wobei $V = \{a, b\}$, $\omega = b$ und $P = \{a \rightarrow ab, b \rightarrow a\}$. Man beginnt mit dem Axiom b und ersetzt es zum ersten mal (Generation 1). Es gibt nur eine Regel, die auf b angewandt werden kann ($b \rightarrow a$). Durch Anwendung dieser Regel erhält man a . In der nächsten Generation wird a mit ab , aufgrund der Regel $a \rightarrow ab$, ersetzt. Als Ergebnis erhält man ab . In Generation 3 erfolgt nun erstmals das parallele Ersetzen: a wird mit ab und b mit a ersetzt. Das Ergebnis ist abb . Man kann nun beliebig viele weitere Generationen bilden. Dies ist in Abbildung 3.3 zu sehen.

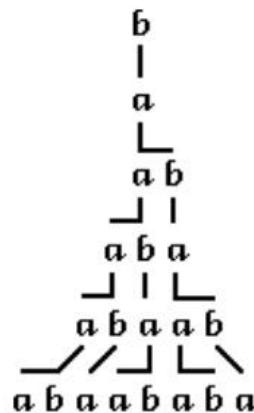


Abbildung 3.3: Erzeugung mehrere Generationen eines L-Systems (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

3.2.3 Turtle-Grafik

Bei der Turtle-Grafik handelt es sich um eine Bildbeschreibungssprache. Die Idee ist, dass man einem Roboter, der mit einem Stift ausgestattet ist, genaue Anweisungen gibt wie dieser sich zu bewegen hat und dabei mit dem Stift eine Linie malt während der Roboter sich bewegt. Die Bewegungsbefehle beschränken sich auf „nach links drehen“, „nach rechts drehen“, jeweils mit einem angegebenen Winkel, und „geradeaus fahren“ für eine gegebene Strecke. Dabei kann der Roboter den Stift auf das Papier drücken oder den Stift anheben, sodass kein Strich gemalt wird. Übersetzt man diese Idee nun auf ein Computerprogramm, so wird der Zustand dieses Roboters mittels des Tripels (x, y, α) definiert [PL90, 6]. Die Variablen x und y entsprechen den kartesischen Koordinaten des Roboters und α dem Richtungswinkel, in die der Roboter blickt. Weiterhin definiert man eine Konstante für den Winkel δ , der angibt um wie viel Grad sich der Roboter bei dem Befehl „nach links drehen“ oder „nach rechts drehen“ drehen soll. Des Weiteren wird eine Schrittweite d definiert, um festzulegen wie weit der Roboter bei dem Befehl „Gerade aus fahren“ fahren soll. Nun kann man dem Roboter eine Reihe von Befehlen geben z.B. „nach links drehen“, „fahren“, „nach links drehen“, „fahren“, „nach rechts drehen“, „fahren“. Da der Roboter während des Fahrens Linien zeichnet, entsteht ein Bild aus Linien. Eine solche Programmiersprache ist Logo [Bob67], welche beim Zeichnen des Bildes eine Schildkröte statt des Roboters als visuelle Repräsentation wählte. Deshalb bezeichnet man eine solche Grafik als Turtle-Grafik. Für den Begriff des Roboters wird im folgenden der Begriff der Turtle verwendet. Um das Schreiben von Befehlsketten an die Turtle zu verkürzen, hat man die Befehle kodiert. F steht für „fahre gerade aus“ um die Schrittweite d , $+$ bedeutet „nach rechts drehen“ um δ und $-$ bedeutet „nach links drehen“ um δ . Ein Beispiel wäre die Befehlskette $FFF - FF - F - F + F + FF - F - FFF$. In Abbildung 3.4 sieht man das Ergebnis dieser Befehlskette.

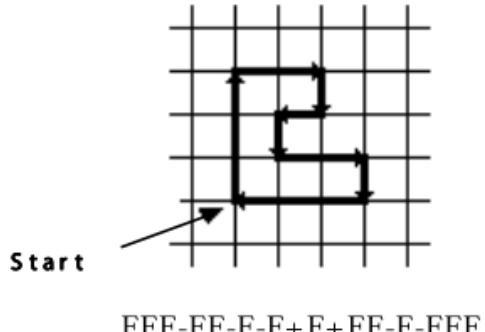


Abbildung 3.4: Interpretation eines Turtle-Strings (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

Die Befehle $(+, -, F)$ kann man als Alphabet für ein L-System nutzen, welches aus einem Axiom nach ein paar Generationen einen Turtle-String erzeugt, welchen man dann einem Turtle-Programm geben kann, das daraus eine Turtle-Grafik zeichnet. An dieser Stelle folgt ein Beispiel aus Prusinkiewicz und Lindenmayer (1990) [PL90, 8]:

$$\begin{aligned}\omega &: F - F - F - F \\ p &: F \rightarrow F - F + F + FF - F - F + F\end{aligned}$$

Dieses L-System besteht aus einer Produktionsregel. Da hier das F mit einer Zeichenkette

ersetzt wird und dies für jede Generation erneut geschieht, ist das Ergebnis selbstähnlich also fraktal. Dies kann man in Abbildung 3.5 erkennen.

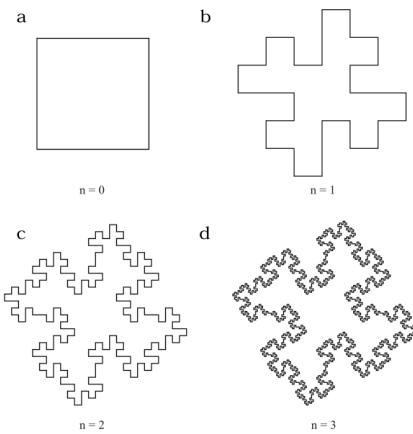


Abbildung 3.5: Quadratische Koch-Insel (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

3.2.4 Verzweigte Strukturen

Das bisherige L-System kann nur eine Linie zeichnen. Diese mag zwar eine komplexe Struktur aufweisen, verzweigte Strukturen wie man sie in der Natur bei Bäumen und Büschen beobachtet sind jedoch schwer umsetzbar. Um dies zu ermöglichen erweitert man das L-System um einen Stapspeicher¹ in Form von eckigen Klammern [PL90, 24]. Dabei hat die öffnende Klammer die Funktion den momentanen Standpunkt der Turtle, dessen Blickrichtung und weitere mögliche Daten, wie momentane Farbe oder Schrittweite, in den Stapspeicher abzulegen. Die schließende Klammer hat die Funktion die Daten wieder aus dem Stapspeicher herzustellen, auch wenn die Turtle in der Zwischenzeit sich weiter bewegt hat und die Daten (z. B. Standpunkt der Turtle) sich verändert haben. Dies ermöglicht Sprünge beim Zeichnen, welche zum Erzeugen von verzweigten Strukturen notwendig sind. Abbildung 3.6 zeigt verschiedene L-Systeme und dessen Turtle-Grafiken. Die Variable n ist die Anzahl der Generationen, also wie oft die Symbole (parallel) ersetzt worden sind.

3.3 Zusammenfassung

Dieses Kapitel hat das Konzept der Textersetzungssysteme und L-Systeme im Speziellen behandelt. Es wurde erklärt wie diese definiert sind und wie man sie mit Turtle-Grafiken verbinden kann, um grafische Darstellung zu erzielen. Diese Kombination eignet sich besonders gut zur Erzeugung verzweigter Strukturen wie man sie bei Bäumen und anderer Vegetation vorfindet.

¹Ein Stapspeicher ist in der Informatik eine dynamische Datenstruktur. Beim Entnehmen eines Elements aus diesem Speicher kann nur das oberste Element entnommen werden.

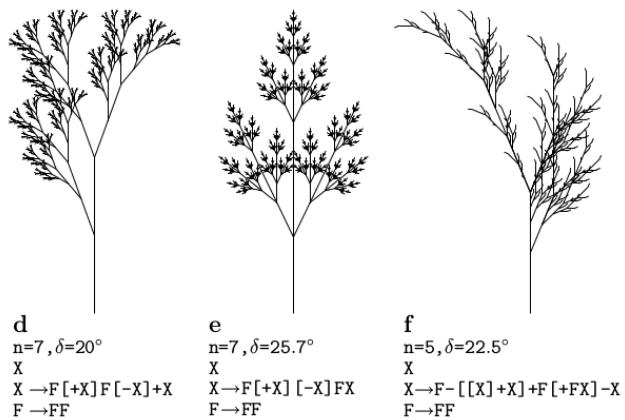


Abbildung 3.6: L-Systeme und ihre grafische Darstellung (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

4 Anwendungsgebiete und Beispiele

4.1 Videospiele

Die häufigste Anwendung von prozeduraler Generierung findet in Videospielen statt. Dessen Einsatz deckt ein breites Spektrum ab.

4.1.1 Level und Karten

Level und Karten von Videospielen werden u.a. mithilfe von Tiles erzeugt. Das Level wird dazu in ein Raster unterteilt und ein Tile in diesem Raster kann eine Wand, Boden, Charaktere, Gegner u.ä. repräsentieren. Die Sicht auf die Spielwelt kann variieren, wobei auch die Form des Rasters unterschiedlich sein kann. Bei Videospielen mit Seitenansicht wie „Super Mario Bros.“ [gam85] oder „Spelunky“ [gam08a] ist das Raster quadratisch, dies gilt auch für Top-Down-Ansichten wie bei „Rogue“ [gam80] oder „Dwarf Fortress“ [Tar06] (siehe Abbildung 4.1). Bei Spielen in Vogelperspektive wird jedoch eine isometrische Ansicht gewählt, sodass die Tiles rautenförmig sind, um somit den Eindruck von Perspektive zu erhalten. Man nennt dies auch 2.5D, um anzudeuten, dass es dreidimensional aussieht, aber dennoch zweidimensional gerendert¹ wird. Beispiele hierfür sind „Diablo“ [gam96] und „UFO: Enemy Unknown“ [gam94].

4.1.2 Modelle

Modelle für Vegetation oder Gebäude und Straßen können mittels L-Systemen generiert werden. Parish und Müller [PM01] haben gezeigt wie man mittels L-Systemen Straßen und Gebäude generieren kann und haben daraus eine Software namens „CityEngine“ [Pas08] ent-

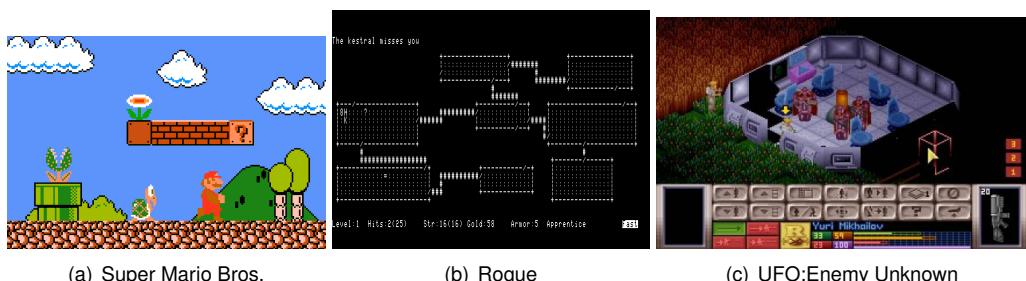


Abbildung 4.1: Spiele mit Tile-Grafik. Links: Super Mario Bros mit Seitenansicht (Quelle: cdn.vox-cdn.com [vox]). Mitte: Rogue mit Top-Down-Ansicht (Quelle: www.mobygames.com [mob]). Rechts: UFO: Enemy Unknown mit isometrischer Ansicht (Quelle: tweakpc.de [twe]).

¹Rendern bezeichnet die grafische Darstellung einer berechneten Szene.

wickelt (siehe Abbildung 4.2). „SpeedTree“ [IDV09] ist eine Middleware² zur Generierung von 3D-Bäumen. Die Engine³ wird von Spielentwicklern und der Filmindustrie lizenziert und in deren Produkten verwendet.



Abbildung 4.2: CityEngine (Quelle: d2.alternativeto.net [alta])

4.1.3 Texturen

Prozedural generierte Texturen können in beliebiger Auflösung erzeugt werden und benötigen kaum Speicher. Beispielsweise wurde die Rauschfunktion Perlin Noise zu diesem Zweck entwickelt und wurde in dem Film „Tron“ [tro82] verwendet, da der Primärspeicher zur damaligen Zeit sehr begrenzt war. Außerdem wollte man Texturen erzeugen, die nicht zu sauber aussahen. Voronoi-Diagramme helfen zelluläre Texturen, wie Haut, Zellen oder Fließen, zu erzeugen (siehe Abbildung 4.3).



Abbildung 4.3: Prozedurale Texturen. Links: Vase mit Perlin Noise Textur (Quelle: entnommen aus Ken Perlin (1985) [Per85]). Rechts: Stein-Fließen-Textur mittels Voronoi-Textur erstellt (Quelle: d2.alternativeto.net [altb])

4.1.4 Landschaften

Terrain ist in Videospielen allgegenwärtig. Vor allem in Open-World-Spielen⁴ sind riesige Landschaften ein wichtiges Spielelement. Um diese nicht per Hand zu erschaffen, werden verschiedene Techniken zur prozeduralen Generierung verwendet. Eine davon ist Heightmapping. Dabei wird jedem Punkt in der Welt ein Helligkeitswert zugeordnet. Je heller der Wert desto höher das Terrain und je dunkler desto niedriger [STN16, 59]. Um zwischen den Punkten auf der

²Middleware ist eine zusätzliche Schicht zwischen Betriebssystem und Anwendungen.

³Eine Engine ist ein im Hintergrund laufendes Programm, das den Kern eines Systems darstellt. Dieser dient als Grundlage, um z.B. Grafiken zu berechnen und anzuseigen.

⁴In Open-World-Spielen kann der Spieler die gesamte Spielwelt jederzeit bereisen und ist nicht durch Level beschränkt.

Heightmap einen organischen Verlauf zu erzielen, wird die Perlin-Noise-Funktion genutzt. Um die selbstähnliche Struktur von Landschaften zu erzeugen werden Fraktal-Funktionen eingesetzt [STN16, 63]. Es gibt viele weitere Techniken zur Erzeugung von Landschaften und oft wird eine Kombination von verschiedenen Methoden angewendet. Sean Murray zeigte 2017 in seiner Präsentation „Building Worlds Using Math(s)⁵ verschiedene Techniken zur Landschaftsgenerierung und erklärt dabei auch welche in seinem Spiel „No Man's Sky“ [Hel16] verwendet wurden (siehe Abbildung 4.4).



Abbildung 4.4: Generierte Landschaft in No Man's Sky (Quelle: sm.ign.com [IGN])

4.1.5 Audio

Audio ist in Spielen genauso wichtig wie das Visuelle. Sie kann dem Spieler Feedback geben, die Dramaturgie einer Szene unterstreichen und Emotionen übermitteln. Bisher wird Sound kaum prozedural in Spielen erzeugt. Jedoch könnte dies zu besseren Ergebnissen bei prozedural generierten Spielen führen, da die Variationen von Inhalten z.B. generierte Gegner auch unterschiedliche Sounds erhalten könnten. Andernfalls werden immer dieselben Geräusche verwendet, was der Vielfalt an generierten Inhalten nicht gerecht wird [HMVDI13, 4]. Jedoch gibt es Ansätze zur Realisierung von PCG in diesem Bereich. So unterteilt Andy Farnell in „An introduction to procedural audio and its application in computer games“ [Far07] die Erzeugung von generierten Sound in synthetischen, generativen, stochastischen und algorithmischen Sound. In seiner Arbeit zeigt Farnell wie diese erzeugt werden können.

4.1.6 Texte

Große Mengen generierten Inhalts bedürfen oft eigener Namen. Um einzigartige Namen für alle Entitäten z.B. Planeten zu vergeben, müssen die Namen ebenfalls generiert werden. Man nutzt dazu meist das Konzept der Markow Kette. Die Markow Kette ist ein stochastischer Prozess, der dazu dient Wahrscheinlichkeiten für zukünftige Ereignisse auf Basis vorangegangener zu berechnen. Dieses Modell benötigt dazu eine Menge von Daten als Basis zur Vorhersage. Man kann das Modell mit Namen füttern und dabei die Namen in drei Teile teilen, in Buchstabenketten für den Anfang, Mitte und Ende von Namen. Außerdem gibt man die jeweiligen Häufigkeiten der Buchstabenketten an. Mithilfe des Modells ist es dann möglich Namen zu generieren.

Auch Geschichten lassen sich prozedural erzeugen. Ein Ansatz dafür ist die Simulation der Ereignisse, die ein Protagonist in einer Welt erfährt. Man simuliert die Reaktion der einzelnen

⁵<https://www.gdcvault.com/play/1024514/Building-Worlds-Using>

Aktionen, die ein Protagonist verursacht und zählt zum Schluss die Aktionen und Reaktionen auf [STN16, 124]. Der Nachteil bei dieser Technik ist, dass dabei das was eine Story eigentlich ausmacht (beispielsweise der Spannungsbogen) zu kurz kommt.

Ein anderer Ansatz ist deshalb die Sicht des Erzählers einzunehmen und dessen Aktionen zu simulieren, um eine gute Geschichte zu erzählen. Dieser Ansatz nutzt die Technik der Planungssuche aus der künstlichen Intelligenz. Ziel hierbei ist es eine Sequenz von Aktionen zu finden, die das Planungsziel erfüllen [STN16, 125]. Das Planungsziel des Erzählers ist es eine gute Geschichte zu erzählen, darauf basierend wird nach einer Sequenz von Elementen gesucht, die dieses erfüllt. Nachteil hierbei ist, dass auf die Aktion des Spielers nicht eingegangen wird. Deshalb ist eine hybride Lösung aus beiden Ansätzen passender [STN16, 126].

4.2 Filme

Auch in der Filmindustrie wird PCG eingesetzt. Da in der Natur zwei Objekte, egal ob Vegetation, Tiere oder Steine, nie gleich aussehen und Künstler schwer tausende von einzigartigen solchen Objekten erschaffen können, verschafft man sich mittels PCG Abhilfe. Man verwendet dazu eine so genannte *imperfect factory*. Mit dieser kann ein Künstler auf Basis eines hand-geschaffenen Objekts beliebig viele weitere erzeugen, welche Abweichungen vom Original aufweisen. Für die Simulation von Menschenmengen greift man ebenfalls auf PCG zurück, da es sehr teuer wäre Statisten dafür einzustellen, zu unterrichten und auszustatten. Die Software MASSIVE wird zu diesem Zweck eingesetzt und hat beispielsweise in „Der Herr der Ringe“ (2001-2003) von Regisseur Peter Jackson die Schlachten der Armeen visualisiert (siehe Abbildung 4.5).

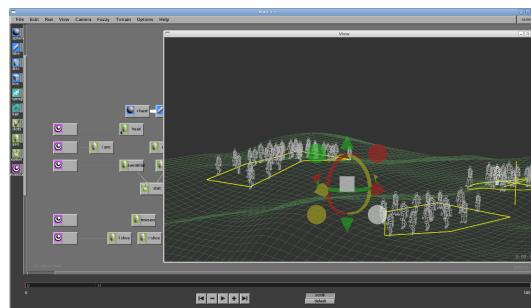


Abbildung 4.5: Menschen-Massen-Simulation mit MASSIVE (Quelle: billdesowitz.com [bil])

4.3 Medizin

In der Medizin könnte PCG ebenfalls eingesetzt werden. So zeigen Duffy und Wang [DW15] wie die Generierung von Patientendaten auf Basis von Krankheitssymptomen genutzt werden kann, um (angehenden) Ärzten beim Training der Diagnostik zu helfen.

4.4 Zusammenfassung

Die Anwendungsgebiete der prozeduralen Synthese erstrecken sich größtenteils über die Videospiel- und Filmindustrie. Die Generierung von graphischen Inhalten spielt dabei die größte Rolle. Aber auch die Generierung von Spielkonzepten und Story gewinnt zunehmend an Bedeutung. Außerdem dieser Gebiete findet PCG bisher leider wenig Anwendung. Die ersten Ansätze, wie sie in der Medizin zu sehen sind, wirken vielversprechend.

5 Stand der Technik

In diesem Kapitel werden die beiden professionellsten Softwarelösungen, die momentan State-Of-The-Art sind, vorgestellt. Es werden auch kleinere nennenswerte Programme präsentiert, die ähnlich gute Ergebnisse erzielen. Anschließend werden die Softwarelösungen miteinander verglichen. Zum Schluss wird ein Fazit aus dem Vergleich geschlossen.

5.1 SpeedTree

Die am meisten genutzte Software zur Generierung von Vegetation in Videospielen und Filmen ist die Middleware „Speedtree“ [IDV09]. Sie wurde von Interactive Data Visualization (IDV) entwickelt und war Anfangs nur ein Plugin für 3D Studio Max [Aut90a] und Maya [Aut90b]. Sie ermöglichte schon damals die Generierung von Pflanzen in verschiedenen Detailstufen. Im Jahr 2009 wurde die Plugin-Entwicklung von IDV eingestellt und es wurde fortan eine eigenständige Sofwarelösung entwickelt. IDV veröffentlichte den SpeedTree Modeler, ein Programm zur grafischen Entwicklung von Bäumen. Später folgten SpeedTree Cinema für Filmproduktionen und SpeedTree Games zur Unterstützung von Videospieleentwicklungen. SpeedTree ist somit eine Middleware und besteht aus einer Gruppe von Softwareprodukten. Sehr viele Videospiele nutzen die Middleware u.a. Battle Field, Call of Duty, Destiny, Horizon Zero Dawn, No Man's Sky. Auch für Filme wird SpeedTree immer häufiger genutzt z. B. Star Wars: The Force awakens, Avangers: Age of Ultron, Warcraft: The Beginning.¹

5.1.1 SpeedTree for Games

Die Suit für die Generierung von Bäumen für Videospiele nennt sich SpeedTree for Games. Sie enthält den Modeler, Compiler und die SDK². Die Suit ermöglicht die Generierung von Vegetation während der Laufzeit, also in Echtzeit. Die Erzeugten Geometrien besitzen wenige Polygone, nutzen Textur Atlanten³, haben Level-of-Detail-Stufen⁴ und werden in einem effizienten binär Format gespeichert.

¹<https://en.wikipedia.org/wiki/SpeedTree#Applications>

²Ein Software Development Kit (SDK) ist eine Sammlung von Programmierwerkzeugen und Programmbibliotheken, die zur Entwicklung von Software dient.

³Ein Textur Atlas ist eine große Textur, die viele kleine Texturen enthält.

⁴Level of detail (kurz LOD) ist eine Technik, bei der verschiedene Variationen eines Modells weniger Polygone enthalten, um Rechenkapazität zu sparen.

5.1.2 SpeedTree Modeler

Bei dieser Software handelt es sich um ein CAD-System⁵. Es ist also ein Modellierungs-Werkzeug, das für Windows entwickelt wurde. Sie ermöglicht sowohl das Generieren von Vegetation anhand verschiedener Parameter als auch die individuelle Anpassung von generierten Bäumen. Zu den einstellbaren Parametern gehören u.a. Astlänge, Verzweigungswinkel und Texturen für Rinde und Blätter des Baums.

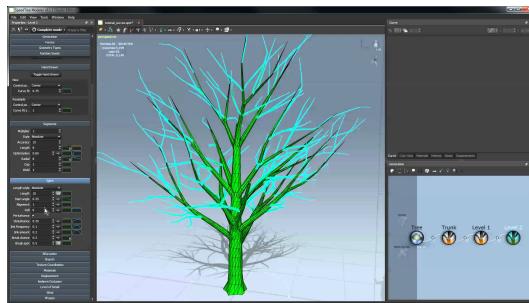


Abbildung 5.1: Generation eines Baumes mit der Software SpeedTree (Quelle: [ytmp3.com \[yti\]](#))

5.2 PlantFactory

PlantFactory ist eine Produktlinie von E-on Software, welche mit mehreren professionellen Produktlinien die Erstellung von natürlichen 3D-Szenen entwickelt. Dabei übernimmt PlantFactory die Generierung, Animierung und das Rendering von Vegetation. Weitere Features sind Wachstumssimulation (durch Parameter wie Wachstum, Dauer, Trieb-Entwicklung etc. wird biologisches Wachstum simuliert), Variationen (voreingestellte Variationen ermöglichen kleine Abwandlungen der Pflanzen) und Level-of-Detail-Generierung.

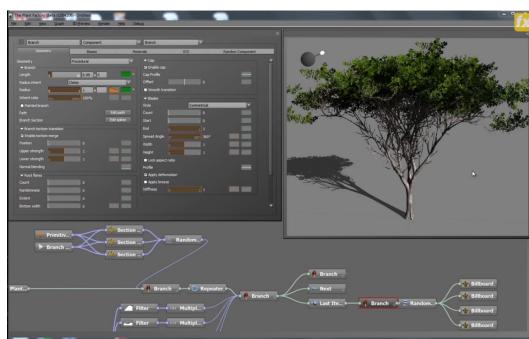


Abbildung 5.2: Generation eines Baumes mit der Software PlantFactory (Quelle: [zedspace.com \[zed\]](#))

⁵Computer-aided design (kurz CAD) ist Design mithilfe von Software.

5.3 Weitere Systeme

5.3.1 Xfrog

Mit Xfrog modelliert man Pflanzen, organische Formen oder abstrakte Objekte. Die Software generiert prozedural realistische, hoch detaillierte, voll texturierte dreidimensionale Bäume, Büsche und Blumen. Die Parameter erstrecken sich dabei über Wachstum, Anzahl der Zweige, Aststärke u.v.m. Jede Komponente kann animiert werden, dies ermöglicht Blumen blühen zu lassen oder Bäume vom Samen bis zum voll ausgewachsenen Baum zu simulieren. Eine Bibliothek stellt 600 Arten zur Verfügung. Die Modelle wurden alle manuell erstellt mit Hilfe von prozeduralen Hierarchien, welche botanischen Regeln unterliegen.

5.3.2 Flora3D

Dieses Programm ist ein Werkzeug für Künstler, Spielentwickler und 3D-Modellierer. Die Software erzeugt 3D-Modelle von Pflanzen, welche in verschiedenen Formaten exportiert werden können. Wobei die Ergebnisse für Spielszenen und Echtzeit-Rendering optimiert sind. Auch die Erzeugung von LODs gehört zum Repertoire dieser Software.

5.3.3 Virtual Laboratory/L-Studio

Professor Przemyslaw Prusinkiewicz, Co-Autor von „The Algorithmic Beauty of Plants“ [PL90], hat mit seiner Forschungsgruppe „Algorithmic Botany“ an der Universität Calgary eine eigene Software zur prozeduralen Generierung von Pflanzen entwickelt. Für Linux und Mac nennt sich diese vLab und für Windows L-Studio. Die Software dient als Grundlage für Forschungen der Gruppe um Professor Przemyslaw Prusinkiewicz.⁶

5.4 Vergleich der Softwarelösungen

Im Folgenden wurden die vorgestellten Softwareprodukte nach verschiedenen Kriterien untersucht. Dazu sind folgende Merkmale gewählt worden:

- Bedienung: Wie leicht lässt sich die Software bedienen? Ist die Benutzerführung intuitiv? Lassen sich Modelle einfach und nachvollziehbar erstellen?
- Konfiguration: Lassen sich die generierten Modelle individuell nachbearbeiten? Wie viele Parameter stehen zur Generierung zur Verfügung?
- Integration: In welche Umgebungen (z.B. Engines) lässt sich die Software integrieren?
- Optimierung: Sind die generierten Objekte für Echtzeit-Anwendungen optimiert?
- Animation: Lassen sich die Modelle animieren? Z.B. Windanimation.
- LOD: Kann man für die Modelle mittels der Software Level-Of-Details erzeugen?

⁶<http://algorithmicbotany.org/papers/>

- Qualität: Wie hoch ist die Qualität der Ergebnisse?
- Kosten: Wie viel kostet die Software?

Die Abbildung 5.3 zeigt den Vergleich der oben vorgestellten Softwarelösungen.

Produkt	Bedienung	Konfiguration	Integration	Optimierung	Animation	LOD	Qualität	Kosten
SpeedTree	intuitiv, schnell, einfach	individuelle Nachbearbeitung, viele Generierungsparameter	Unreal Engine, Unity und Lumbearyard	ja	Wind-simulation	dynamische LOD-Generierung	hoch	Games: 8500 € Cinema: 4300€ Non-Game: 770€
PlantFactory	intuitiv, schnell, einfach	individuelle Nachbearbeitung, viele Generierungsparameter	VUE, Carbon Scatter, LumenRT	ja	Windsimulation, Wachstums-simulation	dynamische LOD-Generierung, LOD-Regeln	hoch	Designer: 420€ Studio: 850€ Producer: 1700€
Xfrog	lange Einarbeitungszeit, komplexes Interface	individuelle Nachbearbeitung, viele Generierungsparameter	Unity, Vue	nein	Windsimulation, Wachstumssimulation	nein	Mittel	C4D/Maya: 220€ Windows: 160€
Flora3D	umständlich, langsam, schlicht	keine Nachbearbeitung, wenige Generierungsparameter	keine	ja	nein	manuell	Niedrig	51€
L-Studio	sehr schlicht, sehr wenige Optionen, Hohe Einarbeitungszeit	keine Nachbearbeitung, beliebig viele vom User erstellte Generierungsparameter	keine	nein	Wachstumssimulation	nein	Mittel	Kostenlos

Abbildung 5.3: Vergleich verschiedener Softwarelösungen zur Generierung von 3D-Bäumen
(Quelle: eigene Darstellung)

5.5 Fazit

SpeedTree und PlantFactory bieten die umfangreichsten Modellierungsoptionen an, sie sind einfach zu bedienen und erzeugen sehr gute Qualität. Sie sind die professionellsten Softwarelösungen in dem Bereich 3D-Vegetations-Modellierung und Generierung, allerdings sind sie sehr teuer, sodass sich diese Software nur große Unternehmen leisten können. Xfrog und Flora3D bieten einen kleineren Umfang an Features. Die Erstellung der Modelle ist schwieriger und es ist eine gewisse Einarbeitungszeit nötig. Die Ergebnisse sind vertretbar bis ausreichend. Dafür sind diese Softwarelösungen für kleinere Unternehmen erschwinglich. L-Studio ist schwer zu bedienen und bietet wenige Optionen. Dafür eignet sie sich sehr gut zum Erforschen von biologischen Wachstumsprozessen und ist außerdem kostenlos.

6 Konzept

Das Konzept, für das in dieser Arbeit vorgestellte System, wird in diesem Kapitel erarbeitet. Zur Erzeugung von 3D-Bäumen wird das Konzept des L-Systems, wie es in Abschnitt 3.2 erläutert wurde, gewählt, da sich dieses zum Erzeugen verzweigter Strukturen eignet. Das L-System muss dazu um Symbole erweitert werden, um mithilfe einer 3D-Turtle dreidimensionale Strukturen generieren zu können. Dabei entsteht zuerst nur das Skelett eines Baumes. Darauf aufbauend wird ein Mesh¹ bestehend aus Polygonen erzeugt und anschließend texturiert. Zuletzt zeige ich den Aufbau und Ablauf des Gesamtsystems.

6.1 3D-Turtle

Um Turtle-Grafiken in 3D zu zeichnen, muss das bestehende Turtle-Konzept erweitert werden. Die Blickrichtung der Turtle wird nun als Richtungsvektor (X, Y, Z) realisiert. Dieser Vektor zeigt initial in eine Richtung zum Beispiel nach oben $(0, 1, 0)$, angenommen Y zeigt nach oben. Die Turtle kann entlang seiner drei Achsen rotiert werden. Rotation um die eigene Achse (H), Rotation längst seiner Blickrichtung (L) und nach oben oder unten (U). Abbildung 6.1 veranschaulicht diese Rotationen. Die Rotationen werden mathematisch mittels Rotationsmatrizen (siehe Gle-

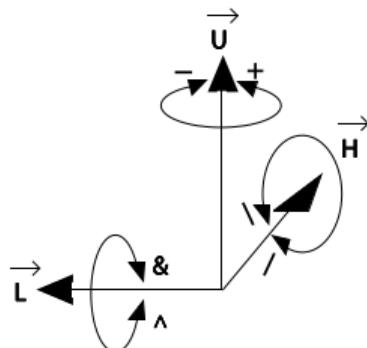


Abbildung 6.1: Rotationen der Turtle (Entnommen aus Prusinkiewicz und Lindenmayer (1990) [PL90])

¹Ein Mesh ist ein Netz aus Polygonen, wodurch ein Objekt modelliert wird.

chung 6.1) realisiert. Dazu wird der Richtungsvektor mit den Rotationsmatrizen multipliziert.

$$\begin{aligned}
 R_U(\alpha) &= \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 R_L(\alpha) &= \begin{bmatrix} \cos \alpha & 0 & -\sin \alpha \\ 0 & 1 & 0 \\ \sin \alpha & 0 & \cos \alpha \end{bmatrix} \\
 R_H(\alpha) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix}
 \end{aligned} \tag{6.1}$$

Das Alphabet des L-Systems wird erweitert um Zeichen für diese Rotationen.

- + Nach links drehen um den Winkel δ mittels Rotationsmatrix $R_L(\delta)$
- Nach rechts drehen um den Winkel δ mittels Rotationsmatrix $R_L(-\delta)$
- < Nach oben drehen um den Winkel δ mittels Rotationsmatrix $R_U(\delta)$
- > Nach unten drehen um den Winkel δ mittels Rotationsmatrix $R_U(-\delta)$
- ? Nach links drehen um die eigene Achse um den Winkel δ mittels Rotationsmatrix $R_H(\delta)$
- ! Nach rechts drehen um die eigene Achse um den Winkel δ mittels Rotationsmatrix $R_H(\delta)$

6.2 Mesh-Generierung

Da die 3D-Turtle aus einem Turtle-String, welcher durch ein L-System generiert wurde, nur einen Skelett-Baum zeichnet (siehe Abbildung 6.2), muss auf dieser Basis ein Mesh generiert werden. Dazu wird mittels der Gleichungen 6.2 und einem vorgegebenen Radius r um jeden

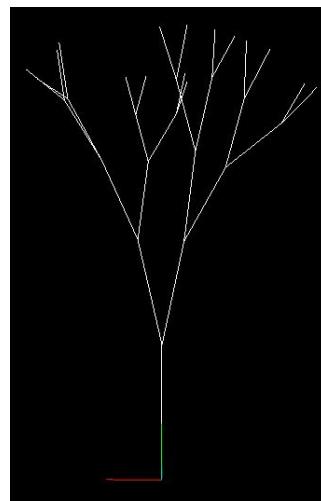


Abbildung 6.2: Skelett-Baum (Quelle: eigene Darstellung)

Punkt (x, y, z) des Skelett-Baums Scheitelpunkte (x^*, y^*, z^*) in einem „Kreis“ berechnet. Die Berechnung muss so oft durchgeführt werden wie viele Seiten der „Kreis“ haben soll. Abbildung 6.3 verdeutlicht diesen Sachverhalt. Sollen es beispielsweise sechs Seiten sein, so muss die Berechnung für sechs Scheitelpunkte stattfinden, wobei der Winkel δ beginnend mit 0° um 60°

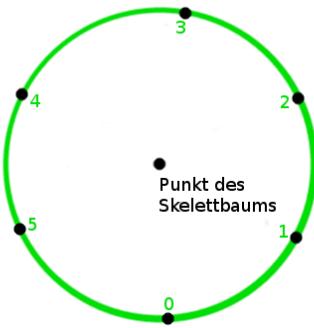


Abbildung 6.3: Berechnung der Punkte kreisförmig um einen Punkt des Skelett-Baums (Quelle: eigene Darstellung)

($360^\circ/6$) jedes mal erhöht werden muss. Der Radius verkürzt sich für jede weitere Kreisberechnung. Handelt es sich um die Spitze eines Astes findet keine Berechnung statt. Die Gleichungen 6.2 zeigen die Berechnung eines Scheitelpunkts.

$$\begin{aligned}x' &= x + r * \cos(\delta * \pi/180) \\y' &= y \\z' &= z + r * \sin(\delta * \pi/180)\end{aligned}\tag{6.2}$$

Alle Scheitelpunkte eines Kreises bilden zusammen ein Segment. Mehrere Segmente bilden zusammen einen Ast. Um aus den Scheitelpunkten Polygone zu rendern, muss der Engine die Reihenfolge der Scheitelpunkte übergeben werden. Jeweils drei Punkte ergeben ein Polygon. Diese Aufgabe ist nicht trivial, da die einzelnen Segmente unterschiedlich viele Scheitelpunkte besitzen können. Dies liegt an der Optimierung, die in Abschnitt 6.3 erklärt wird. Die Lösung des Problems wird in Abschnitt 7.5 erläutert. Durch das Rendern der Polygone besitzt ein generierter Baum Volumen, wodurch Anforderung 2 aus Abschnitt 1.2 (Die Bäume sollen Volumen besitzen und aus Polygonen bestehen) erfüllt wird.

6.3 Optimierung

Bei der Berechnung der kreisförmigen Segmente wie in Abschnitt 6.2 erläutert hängt der Winkel von der Anzahl der Seiten ab. Da aber die kleinen Äste weniger Seiten und somit weniger Polygone als der dicke Stamm und Äste besitzen sollen, muss die Anzahl der Seiten nicht als Konstante festgelegt sein, sondern muss sich anhand des Radius bei diesem Segment berechnen lassen. Durch diese Berücksichtigung wird Anforderung 4 aus Abschnitt 1.2 (Die Modelle sollen optimiert sein d.h. möglichst wenige Polygone besitzen) erfüllt.

6.4 Blattgenerierung

Um Blätter zu erzeugen wird das L-System um das Symbol B erweitert. Wird während der Turtle-String-Interpretation ein B gelesen, dann wird die Ebenengleichung in Parameterform

(siehe Gleichung 6.3) berechnet.

$$\vec{x} = \vec{p} + s\vec{u} + t\vec{v} \text{ mit } s, t \in \mathbb{R} \quad (6.3)$$

Der Vektor \vec{p} ist der sogenannte Stützvektor in unserem Fall der Ortsvektor der Astspitze an dem das Blatt gezeichnet werden soll. Die Vektoren \vec{u} und \vec{v} sind zwei Richtungsvektoren, in diesem Fall ist \vec{u} der Richtungsvektor des Astes und \vec{v} wird durch Rotation von \vec{u} um 90° mittels der Matrix $R_U(\delta)$ berechnet (wie in Abschnitt 6.1 beschrieben). Mit dieser Gleichung lässt sich jeder beliebige Punkt auf der Ebene berechnen, wenn man s und t vorgibt (s und t sind die x - und y -Koordinate auf dieser Ebene). Abbildung 6.4 zeigt das Beispiel für ein viereckiges Blatt, welches als Darstellung in der Spielentwicklung oft gewählt wird, da man eine Alpha-Textur² verwendet um die Blätter darzustellen. Damit wird Anforderung 3 (Die Bäume sollen Blätter besitzen) erfüllt.

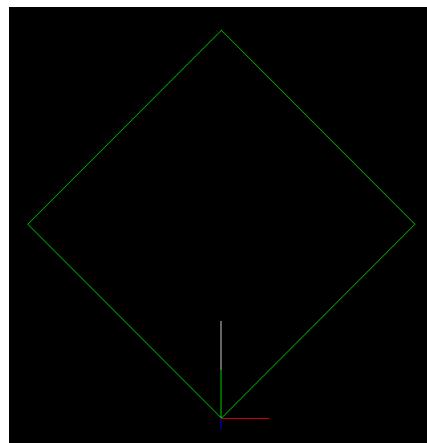


Abbildung 6.4: Geometrie eines Blatts (Quelle: eigene Darstellung)

6.5 Texturierung

Um das Modell zu texturieren muss jedem Scheitelpunkt die UV-Koordinaten der Textur zugeordnet und der Engine übergeben werden. Wenn man den drei Scheitelpunkten der Polygone dabei immer dieselben Koordinaten zuordnet (z.B. Scheitelpunkt 1 immer $(0,0)$, Scheitelpunkt 2 immer $(1,0)$, Scheitelpunkt 3 immer $(0,1)$), dann erreicht man damit eine rudimentäre Texturierung (Anforderung 5 Abschnitt 1.2: Die erzeugten Modelle sollen rudimentär texturiert sein.)

6.6 Aufbau und Ablauf des Gesamtsystems

Das System lässt sich in die Komponenten Interface, Grammatik, String-Generator, Mesh-Generator und Draw-Conroller aufteilen. Abbildung 6.5 verdeutlicht den Aufbau. Das Interface liest die Grammatik und alle zusätzlichen Informationen wie Stammdicke, Anzahl der Generationen etc. aus einer Text-Datei ein und speichert diese in das Modell der Grammatik. Der String-Generator erzeugt daraus einen Turtle-String, welches wiederum vom Mesh-Generator

²Eine Alpha-Textur besitzt für jedes Pixel einen Durchsichtigkeits-Wert. Dies ermöglicht durchsichtige Texturen.

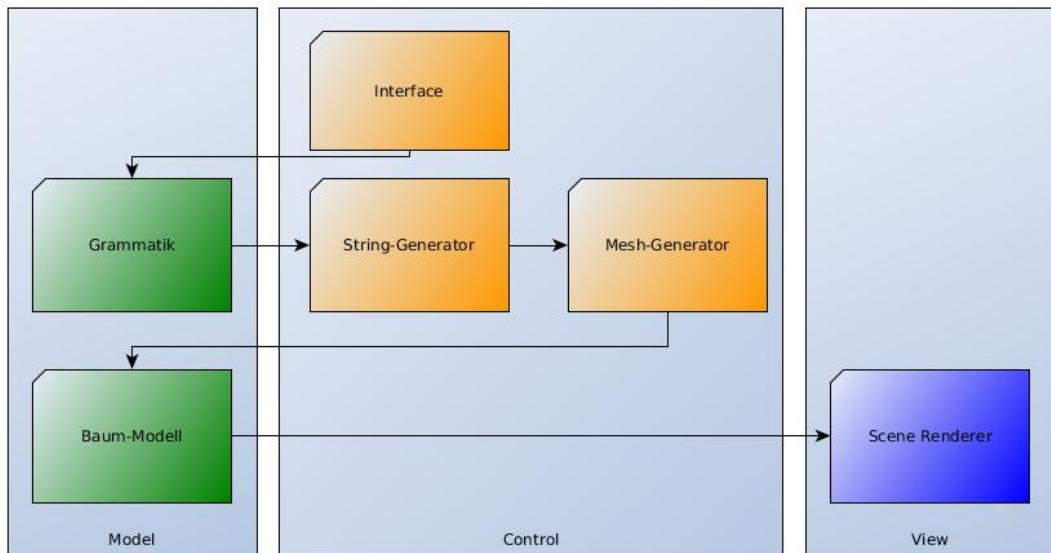


Abbildung 6.5: Komponenten des Systems nach dem Model-View-Controller-Konzept geordnet
(Quelle: eigene Darstellung)

verwendet wird, um zuerst den Skelett-Baum und darauf aufbauend den Baum mit Volumen und Polygonen zu erzeugen. Das erzeugte Modell wird an den Draw-Controller übergeben, welcher die Scheitelpunkte, die Indizes und Texturkoordinaten an die Engine übergibt. Abbildung 6.6 verdeutlicht diesen Ablauf.

6.7 Zusammenfassung

Die Erzeugung von 3D-Bäumen durch ein L-System und der Interpretation der Turtle bedarf einer Erweiterung um Symbole für die Rotationen nach unten bzw. oben sowie der seitlichen Rolle nach links oder rechts. Ein so erweitertes L-System erzeugt einen skelettartigen 3D-Baum. Dieser wird als Grundlage zur Generierung von Volumen benutzt, indem um alle Punkte des Skelettbaums kreisförmig Scheitelpunkte erzeugt werden. Diese wiederum werden zu Polygone zusammengefasst. Die Anzahl der Seiten pro Segment hängt von dessen Radius ab, um somit die Anzahl der Polygone zu optimieren. An jedem Astende wird ein Blatt mittels Ebenengleichung generiert. Abschließend werden jedem Scheitelpunkt auf sehr simple Weise Texturkoordinaten zugeordnet. Das Einlesen der Grammatik aus einer Datei erfolgt über die Interface-Komponente. Der String-Generator erzeugt auf dessen Grundlage einen 3D-Turtle-String, der durch den Mesh-Generator in eine 3D-Geometrie umgewandelt wird. Zuletzt werden die Daten des Models an eine Engine übergeben, die diese visualisiert.

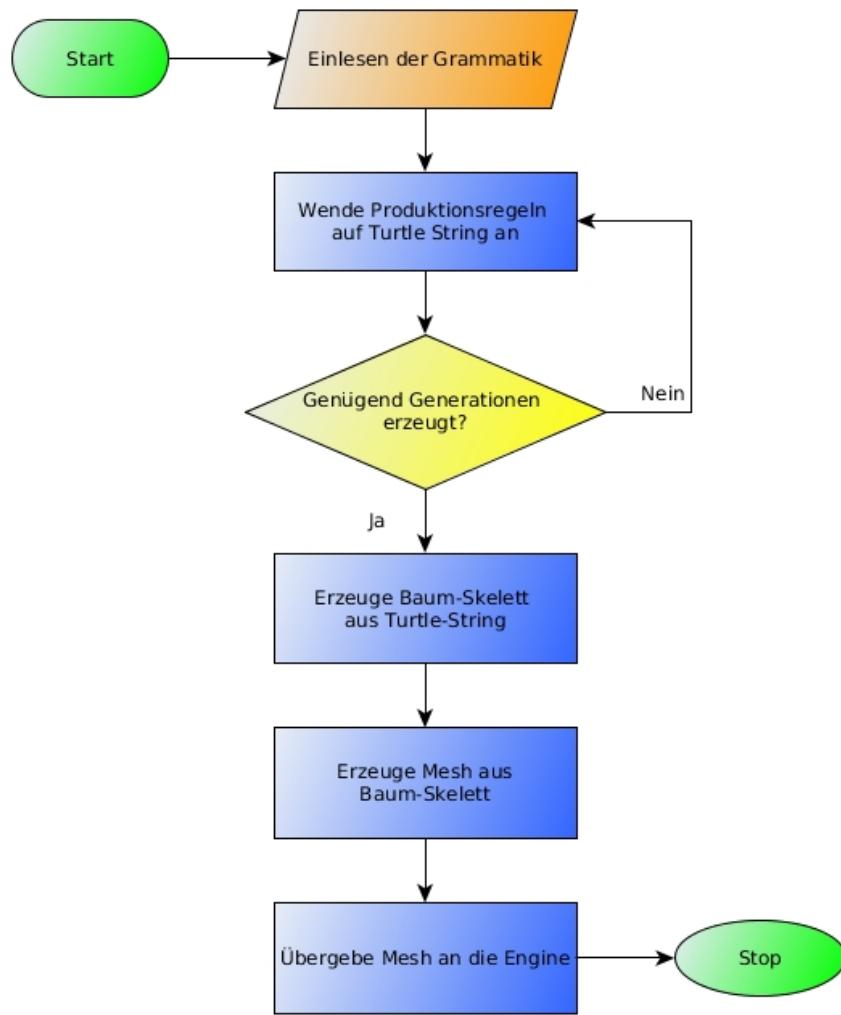


Abbildung 6.6: Ablauf des Systems (Quelle: eigene Darstellung)

7 Implementierung

Das Projekt wurde mit Python programmiert. Die Entscheidung dazu begründet sich darin, dass Python eine höhere Programmiersprache ist und sehr gut dazu geeignet ist String-Manipulationen durchzuführen. Auch die Arbeit von Allen Pike¹ weist darauf hin, dass sich Python zur Bearbeitung von Zeichenketten und zur Realisierung von L-Systemen dazu besser eignet.

7.1 PyOpenGL und PyGame

Da Python als Programmiersprache gewählt wurde, lag es nahe PyGame und PyOpenGL für die Visualisierung der Modelle einzusetzen. Beide sind Python-Programmbibliotheken für grafische Darstellungen. Bei PyGame steht die Spieleentwicklung (Spiel-Logik und Steuerung) im Vordergrund und PyOpenGL, als Bindeglied zu OpenGL, übernimmt die Darstellung von 3D-Szenen.

7.2 Einlesen der Grammatik-Datei

Zu Beginn des Programms wird ein Objekt der Klasse Interface erzeugt. Dieses liest die Datei ein, welche die Grammatik und weitere Informationen enthält.

Listing 7.1: Grammatik-Datei

```

1 axiom: t
2 angle_z: 10 - 20
3 angle_y: 10 - 20
4 angle_x: 10 - 20
5 rules: t -> F[b][b][o][o], o -> b, o -> x, b -> @rtB
6 branch_shortening: 0.8 - 0.9
7 generations: 10
8 base_length: 20
9 base_radius: 0.3
10 bark_texture: ../resources/bark_02_seamless.jpg
11 leaf_texture: ../resources/leafs02.png

```

Listing 7.1 zeigt ein Beispiel für solch eine Datei. Zu den Elementen der Grammatik gehört „axiom“ und „rules“. Axiom ist die Zeichenkette bzw. das Zeichen mit dem der Ersetzungsstring initial gefüllt wird. Rules enthält alle Regeln mit dem Schema $l \rightarrow r$, die einzelnen Regeln werden mit Komma getrennt. Die restlichen Zeilen enthalten weitere Informationen, die zur Generierung des 3D-Baums wichtig sind. „angle_z“ gibt den Freiheitsgrad von z -Rotationen für die Turtle an, dieser kann in diesem Beispiel zwischen 10 und 20 Grad variieren. Selbiges gilt für die y -Rotationen und x -Rotation. Diese Angaben fließen später in die Erzeugung des Turtle-Strings ein. „branch-shortening“ gibt die Variation der Ast-Verkürzung an. „generations“ bestimmt die

¹<http://www.allenpike.com/modeling-plants-with-l-systems/>

Anzahl der Generationen bei der Erzeugung des Turtle-Strings. „base_length“ bestimmt die anfängliche Länge des Stamms und der Äste. „base-radius“ gibt den Stamm-Radius für die Erzeugung des 3D-Baums an. „bark_texture“ gibt den Pfad zur Textur für den Stamm und den Ästen an, ebenso gibt „leaf_texture“ den Pfad für die Blatt-Textur an.

Die Daten für die Grammatik werden in ein Objekt der Klasse Grammar geschrieben. Genauso werden die restlichen Daten in ein Objekt der Klasse Settings gespeichert.

7.3 Text-Generator

Sind alle notwendigen Daten durch das Interface eingelesen, erfolgt daraufhin die Erzeugung des Turtle-Strings. Wichtige Grundlage ist das Objekt der Klasse Grammar. Ein Objekt der Klasse TextGenerator wird erzeugt und darauf die Methode generate_turtle_string_3d aufgerufen. Die Methode benötigt die Argumente Anzahl der zu generierenden Generationen, den Seed sowie die Grammatik und die Setting-Informationen. Der Seed ist wichtig, damit die Generierung deterministisch ist.

Listing 7.2: Funktion zum Generieren eines Turtle-Strings

```

1 def generate_turtle_string_3d(self, num_generations, seedNumber, grammar, settings):
2     seed(seedNumber)
3     string = grammar.axiom
4     for i in range(num_generations):
5         new_string = ""
6         for symbol in string:
7             if(symbol in grammar.variables):
8                 rules = grammar.search_rule_lhs(symbol)
9                 chosen_rule = rules[randint(0, len(rules)-1)]
10                new_string = new_string + chosen_rule.rhs
11            else:
12                new_string = new_string + symbol
13        string = new_string
14        string = self.pre_processing(string)
15        string = self.post_processing(string, settings)
16    return string

```

Listing 7.2 zeigt die Implementierung der Methode zur Generierung des Turtle-Strings. Der String wird zuerst mit dem Axiom der Grammatik befüllt. Danach folgt die Schleife zur Textersetzung, die so oft wiederholt wird wie die Generations-Variable aus Settings dies vorgibt. Eine Generation liest jedes Symbol des momentanen Strings und sucht nach einer Regel, welche dieses Symbol ersetzen kann. Falls mehrere mögliche Regeln gefunden werden, wird eine Regel zufällig ausgewählt. Die Auswahl ist anhand des Seeds deterministisch. Wurde eine Regel ausgewählt so wird diese angewendet. Zur Erzeugung der Generation wird ein neuer String verwendet, da somit kein umständliches Bearbeiten des alten Strings notwendig ist. Da für eine Generation alle Symbole abgearbeitet werden und ggf. ersetzt werden, findet die Ersetzung quasi parallel statt, dies ist charakteristisch für L-Systeme. Wurden genügend Generationen erzeugt, erhält man als Ergebnis den Turtle-String. Allerdings muss dieser noch von zwei Methoden nachverarbeitet werden.

7.3.1 Besondere Sprachkonstrukte

Grammatiken können mit speziellen Sprachkonstrukten erweitert werden. Die Methode *handle_special_constructs* behandelt solche Sprachkonstrukte. Darin findet die Behandlung eines disjunktiven Ausdrucks statt. Wird in dem erzeugten Turtle-String eine öffnende runde Klammer

gefunden so wird eines der Symbole in der Klammer zufällig gewählt und nur dieses bleibt im Turtle-String bestehen. Beispiel: „F+F(+-)F“ dieser String enthält das besagte Sprachkonstrukt und nur einer der beiden Symbole in den Klammern bleibt nach dem Durchlauf der Methode bestehen. Folglich ergibt sich am Ende der Turtle-String „F+F+F“ oder „F+F-F“ als Ergebnis.

7.3.2 Nachverarbeitung

An die Symbole für die Rotationen (+,-,<,>,?,!) und das Ast-Verkürzungssymbol @ müssen Werte angehangen werden, falls diese noch keine enthalten. Die Grammatik kann bereits solche Werte beinhalten, falls welche fehlen werden diese mittels der Methode *post_processing* angehangen. Die Werte werden aus den Settings-Informationen gewonnen. So wird nach einem +-Symbol ein Wert ausgewählt welcher in dem Bereich der *angle_z* Information aus Settings vorgegeben ist. Im Listing 7.1 ist eine Reichweite von 10 bis 20 angegeben, also kann ein +-Symbol nur einen Wert innerhalb dieses Bereichs angehangen werden. Z. B. „F+F+F“ könnte zu „F+12F+18F“ werden. Eine weitere Aufgabe der Methode besteht in dem Ersetzen des *r*-Symbols, welches generell für eine Rotation steht. Dabei wird ein *r* zu + oder –, < oder >, ? oder !. Außerdem werden auch hier die entsprechenden Werte angehangen. Beispielsweise kann der String „FrF“ zu „F-14>11?17F“ werden. Des Weiteren ersetzt die Funktion ein *B* mit *L*, woraus später ein Blatt generiert wird. Dies wird in dieser Methode getan, um sicher zu stellen, dass jedes Astende ein Blatt enthält. Falls die Generierung eines Astes nur mit einem *B* endet und danach keine weitere Generation stattfindet, wird kein Blatt erzeugt. Die geschweiften Klammern sind notwendig, um später die Generierung des Modells für Äste und Stamm von der Generierung der Blätter zu trennen.

7.4 Mesh-Generator

Wurde der Turtle-String erzeugt, erfolgt daraufhin die Generierung des 3D-Models daraus. Zuerst wird ein skelettartiger Baum erzeugt und aus diesem wiederum der 3D-Baum mit Volumen aus Polygonen bestehend. Anschließend werden daraus die Liste der Scheitelpunkte, die Indize-Liste und die Textur-Koordinaten für jeden Scheitelpunkt erzeugt. Diese Daten werden zur Visualisierung in OpenGL benötigt.

7.4.1 Erzeugung des Skelett-Baums

Die Methode *generate_vertex_model* arbeitet den Turtle-String Zeichen für Zeichen ab. Bestimmte Zeichen haben, wie in Abschnitt 6.1 erläutert, eine Bedeutung, welche durch eine 3D-Turtle interpretiert und ausgeführt werden. Bei einem *F* wird auf Basis des Orts- und Richtungsvektors der Turtle und der momentanen Schrittweite der neue Ortsvektor berechnet. Dieser Punkt wird in eine Liste gespeichert, diese repräsentiert einen Ast. Wird ein Rotationssymbol (+,-,<,>,?,!) gelesen wird mittels entsprechender Rotationsmatrix (siehe Gleichung 6.1) und dem dahinter stehenden Wert (dem Winkel) der neue Richtungsvektor berechnet. Werden daraufhin weitere *F* gelesen und entsprechende Punkte berechnet, werden diese der Ast-Liste hinzugefügt. Wird ein [gelesen so wird der momentane Ortsvektor, Richtungsvektor und Schrittweite

der Turtle auf jeweils einem Stack² abgelegt. Wird ein `]` gelesen werden diese Werte von ihren Stacks geholt und die momentanen Werte überschrieben. Dies ermöglicht Sprünge beim Zeichnen des Baumes. Außerdem wird die Ast-Liste abgeschlossen, dem Modell hinzugefügt und eine neue leere Ast-Liste erstellt. Somit weiß man welche Punkte einem Ast angehören. Ein `@` Symbol verkürzt die Schrittweite um den dahinter stehenden Faktor. Wird ein `L` gelesen dann wird ein Blatt erzeugt. Des Weiteren werden die Rotationswinkel in einer Liste gespeichert, diese wird für die Mesh-Generierung benötigt.

7.4.2 Blattgenerierung

Die Methode `create_leaf` erzeugt ein Blatt durch die in Abschnitt 6.4 beschriebene Ebenengleichung. Dazu werden vier Punkte in einem Quadrat berechnet, die abhängig von der Astlänge sind, um somit unterschiedlich große Blätter zu generieren.

7.4.3 Erzeugung des Polygon-Modells

Die Methode `generate_poly_model` arbeitet jeden Ast durch und betrachtet dabei jeden Punkt in der Liste eines Astes. Um jeden Punkt herum wird mittels des in Abschnitt 6.2 beschriebenen Verfahrens eine Menge von Punkten kreisförmig berechnet. Die Punkte bilden zusammen ein kreisförmiges Segment.

Listing 7.3: Algorithmus zur Berechnung der Segment-Scheitelpunkte

```

1 num_sides = 3 + local_radius / NUM_SIDES_FACTOR
2 delta = 360.0 / num_sides
3 while ankle < 360:
4     x = vertex[0] + local_radius * math.cos(ankle * math.pi / 180.0)
5     y = vertex[1]
6     z = vertex[2] + local_radius * math.sin(ankle * math.pi / 180.0)
7     point = (x, y, z)
8     segment.append(point)
9     ankle += delta
10    branch.append(segment)

```

Das Listing 7.3 zeigt die Berechnung eines Segments. Die Berechnung der Anzahl der Seiten (`num_sides`) ist abhängig vom Radius des Segments. Dadurch erhalten dicke Äste mehr Seiten und somit Polygone als dünne Äste. Dadurch wird eine Optimierung des Modells (wie in Abschnitt 6.3 erläutert) erreicht. `Vertex` ist der Punkt des Skelett-Baums, welcher gerade in der Schleife betrachtet wird. Handelt es sich bei dem momentanen Punkt um das Ende eines Astes (letztes Element in der Ast-Liste), so wird kein Segment berechnet und der Punkt an sich ist das Segment. Am Ende eines Schleifendurchlaufs wird das Segment einer Ast-Liste hinzugefügt.

7.4.4 Rotation des Segments

Die Berechnung des Segments, wie in Abschnitt 7.4.3 beschrieben, erzeugt nur Punkt in der Ebene. Damit diese dem Winkel des Skelett-Baums an dieser Stelle entsprechen wird das Segment um die Winkel rotiert, die in Abschnitt 7.4.1 in einer Liste gespeichert wurden.

²Der Stapelspeicher wird auch Stack genannt.

7.5 Indize-Liste

Die Scheitelpunkte des Modells sind nun berechnet. Zur visuellen Darstellung benötigt OpenGL jedoch die Reihenfolge der Scheitelpunkte, die jeweils ein Polygon bilden. Dies war eine besondere Herausforderung, da die Segmente eines Astes aufgrund der Optimierung, wie in Abschnitt 7.4.3 beschrieben, beliebig viele Scheitelpunkte besitzen können. Das momentane Segment eines Astes wird als *current_segment* bezeichnet und das nächste Segment als *next_segment*. Abbildung 7.1 verdeutlicht die Bezeichnungen. Des Weiteren wurden die Po-

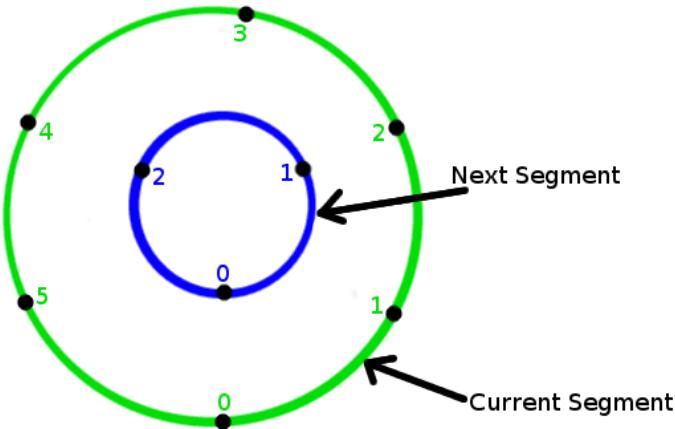


Abbildung 7.1: Zwei Segmente. Das äußere Segment (sechs Seiten) beinhaltet die grünen markierten Punkte und wird als *current_segment* bezeichnet. Das innere Segment (drei Seiten) beinhaltet die blauen Punkte und wird als *next_segment* bezeichnet. (Quelle: eigene Darstellung)

lygone in zwei Teilmengen unterschieden. Diese sind zum einen die Polygone, die „normal-herum“ gezeichnet werden (diese werden als *triangles* bezeichnet) und zum andern die Polygone, die „verkehrt-herum“ gezeichnet werden (diese werden als *inverse_triangles* bezeichnet). Abbildung 7.2 verdeutlicht diese Einteilung.

Listing 7.4: Algorithmus zur Berechnung der Indizes

```

1  for branch in model:
2      for segment_index in range(len(branch)-1):
3          current_segment = branch[segment_index]
4          next_segment = branch[segment_index + 1]
5          delta = float(len(current_segment)) / len(next_segment)
6          for cur_seg_index in range(len(current_segment)):
7              triangles.append(i + cur_seg_index)
8              triangles.append(i + (cur_seg_index + 1)%len(current_segment))
9              triangles.append(i + len(current_segment) + int(cur_seg_index/delta))
10         if len(next_segment) > 1:
11             for next_seg_index in range(len(next_segment)):
12                 inverse_triangles.append(i + len(current_segment) + next_seg_index)
13                 inverse_triangles.append(i + int(math.ceil((next_seg_index+1) * delta))%len(current_segment))
14                 inverse_triangles.append(i + len(current_segment) + (next_seg_index + 1)%len(next_segment))
15         i += len(current_segment)
16         i += 1

```

Listing 7.4 zeigt den Algorithmus für die Erzeugung der Indizes-Liste. Das Verhältnis der Anzahl Scheitelpunkte des *current_segment* und *next_segment* wird als *delta* bezeichnet und gibt die Anzahl der Scheitelpunkte an, die jedem Scheitelpunkt des *next_segment* der *current_segment* zugeordnet werden. Das Verhältnis muss außerdem um eins erhöht werden. In Abbildung 7.2

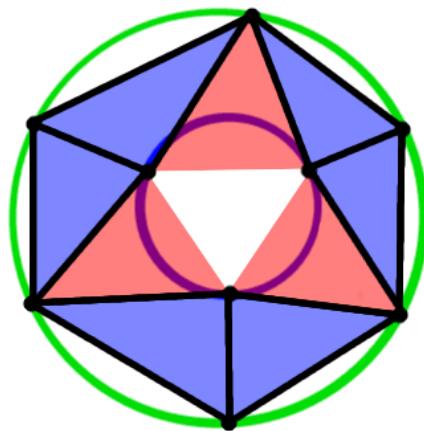


Abbildung 7.2: Die blauen Dreiecke werden als *triangles*, die roten Dreiecke als *inverses_triangles* bezeichnet (Quelle: eigene Darstellung)

werden beispielsweise jedem Scheitelpunkt des *next_segment* drei Scheitelpunkte des *current_segment* zugeordnet.

7.6 Textur-Koordinaten

Jedem Scheitelpunkt wird eine UV-Koordinate für die Textur zugeordnet und in einer Liste gespeichert. Die Zuordnung erfolgt rudimentär.

Listing 7.5: Zuordnung der Textur-Koordinaten

```

1  for i, index in enumerate(model_indices[0]):
2      if i % 3 == 0:
3          model_coord.append((0,0,w))
4      if i % 3 == 1:
5          model_coord.append((u,0,w))
6      if i % 3 == 2:
7          model_coord.append((0,v,w))

```

Listing 7.5 zeigt die simple Zuordnung der Koordinaten für jeden Scheitelpunkt der *triangles*-Polygone (siehe Abschnitt 7.5). Dabei wird jedem dritten Scheitelpunkt dieselbe Koordinate zugewiesen.

7.7 Scene-Renderer

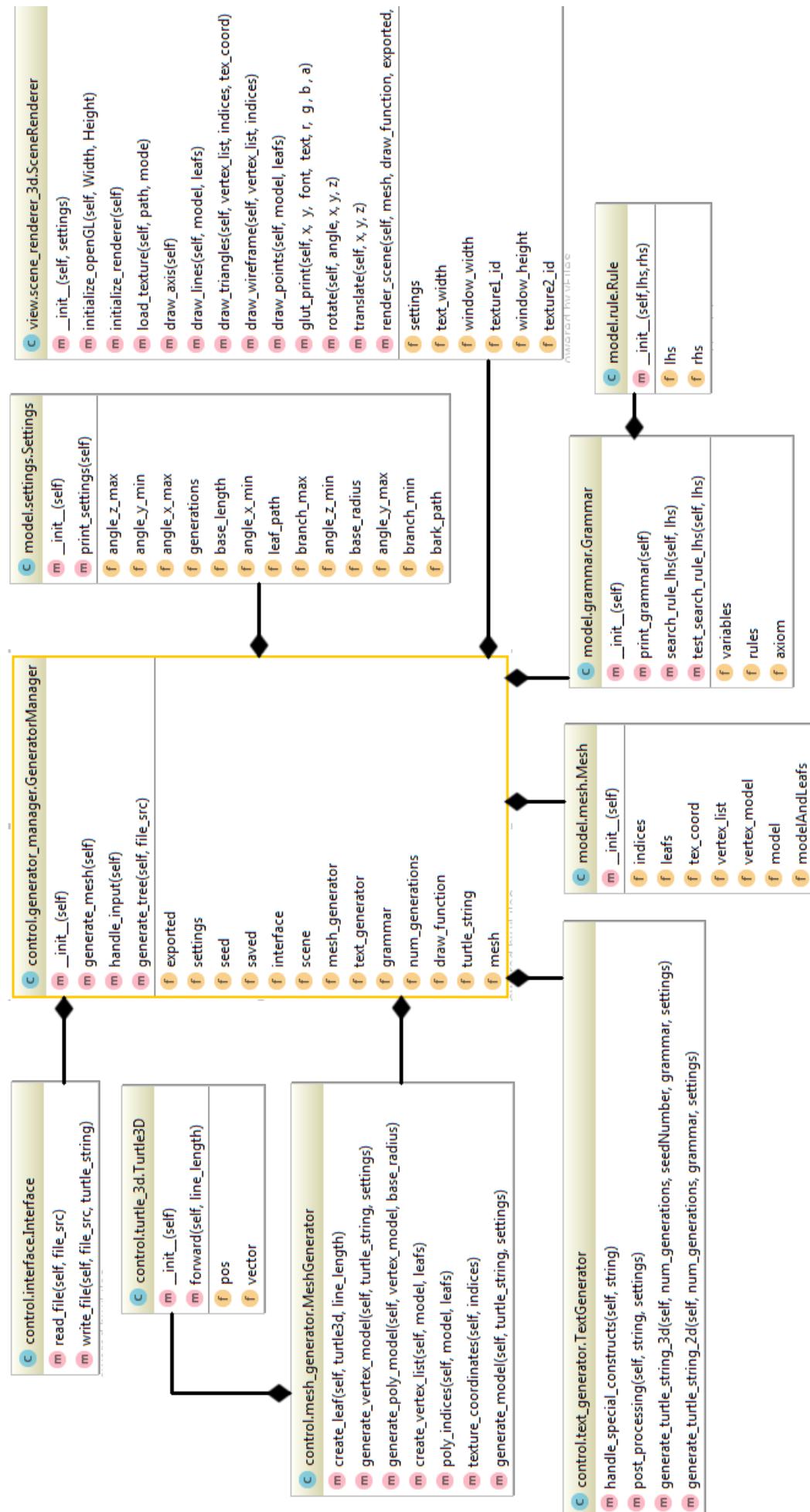
Zum Schluss wird dem Scene-Renderer das Mesh (inklusive Indizes, Scheitelpunkte und Textur-Koordinaten) übergeben und gerendert. Zuerst muss jedoch die Render-Umgebung und OpenGL initialisiert werden. Dabei wird ein Fenster erzeugt, die Texturen geladen und das Koordinatensystem verschoben, damit das Objekt, welches in der Mitte des Koordinatensystems gesetzt wird vor der Kamera liegt und gerendert werden kann.

7.8 Draw-Funktionen

Der User kann verschiedene Draw-Funktionen auswählen. Jede Funktion zeichnet andere Geometrien. Die *draw_points* zeichnet nur die Scheitelpunkte des Meshs, *draw_lines* zeichnet nur Linien, indem es zwei aufeinander folgende Punkte in der Indizes-Liste zu einer Linie verbindet, *draw_wireframe* zeichnet die Polygone des Meshs als Drahtgitter-Modell, indem es alle Punkte der Polygone verbindet und als Linie zeichnet. Die wichtigste Funktion ist *draw_triangles*, diese zeichnet die texturierten Polygone. Außerdem werden die Blätter mittels Alpha-Kanal in der Textur durchsichtig gerendert.

7.9 Zusammenfassung

In diesem Kapitel wurde erklärt wie das System zum Generieren von 3D-Bäumen implementiert wurde. Zuerst erfolgte die Erläuterung wie die Daten aus der Text-Datei eingelesen werden und in die Grammar- bzw. Settingsklasse gespeichert werden. Auf Basis dieser Daten wird mithilfe des Text-Generators der Turtle-String erzeugt, wobei der String zu Anfang mit dem Axiom befüllt wird und danach mehrmals abhängig von der Iterationstiefe (Anzahl der Generationen) der gesamte String mit den Regeln der Grammatik ersetzt wird. Danach erfolgt eine Nachverarbeitung des Strings, um Werte für die Winkel u. ä. einzusetzen. Der Turtle-String wird dann von der 3D-Turtle interpretiert. Dabei werden Punkte im dreidimensionalen Raum berechnet. Der dabei entstehende Skelett-Baum dient wiederum als Basis, um daraus einen 3D-Baum mit Volumen bestehend aus Polygonen zu erzeugen. Dies wird erreicht, indem um jeden Punkt des Skelett-Baums in einem Kreis Scheitelpunkte berechnet werden. Diese müssen am Ende in einer bestimmten Reihenfolge der Engine gegeben werden, welche aus jeweils drei Punkten ein Polygon visualisiert. Mithilfe von UV-Koordinaten kann die Engine die Polygone außerdem texturiert darstellen. Abbildung 7.3 zeigt das Klassendiagramm des Systems.



8 Ergebnisse

In diesem Kapitel wird die Steuerung des Programms erklärt und anschließend die Ergebnisse präsentiert. Dabei werden verschiedene Grammatiken und deren visuelle Darstellungen gezeigt. Die erzeugte Vegetation sind Beispiele und entsprechen keinen in der Natur vorfindbaren Spezies. Am Ende werden die Ergebnisse bewertet.

8.1 Steuerung, Speichern und Export

Mit Drücken der Enter-Taste kann der User des Programms zwischen den Draw-Funktionen wechseln. Die Pfeiltasten erlauben das Modell seitlich zu drehen bzw. nach oben oder unten zu verschieben. Das Mausrad kann genutzt werden, um das Modell heran- oder wegzuzoomen. Mittels Leertaste kann man einen neuen Turtle-String generieren und somit auch ein neues Modell erzeugen. Die Plus- und Minustaste kann benutzt werden, um die Iterationstiefe (Anzahl der Generationen) bei der Erzeugung des Turtle-Strings zu variieren. Mit s kann man das momentane Modell als Turtle-String in einer Text-Datei speichern und mittels e wird das Modell als Obj-Datei exportiert. Die Obj-Datei kann in einer Spielengine oder einem Modellierungseditor geladen werden.

8.2 Grammatiken und Geometrien

Es folgen Beispiele für die Erzeugung von Baum-Modellen aus verschiedenen Grammatiken mit dem System. Es wird jeweils die zugrundeliegende Grammatik, die Iterationstiefe (Anzahl der Generationen), die Polygonanzahl, der Turtle-String und die Bilder aufgelistet.

8.2.1 Tree01-Grammatik

Listing 8.1: Tree01-Grammatik

```

1 axiom: Fmnop
2 angle_z: 10 - 25
3 angle_y: 10 - 25
4 angle_x: 10 - 25
5 rules: m -> [+(><)AB], n -> [+<180(><)AB], o -> [+<90(><)AB], o -> x, p -> [+<270(><)AB], p -> x, A -> @F[rFXB]XB,
6 rules: X -> @rFX[rFXB]rFXB, X -> @rFXB]rFXB
7 branch_shortening: 0.6 - 0.8
8 generations: 3
9 base_length: 10
10 base_radius: 0.3
11 bark_texture: ../resources/bark_seamless.jpg
12 leaf_texture: ../resources/leafs.png

```

Iterationstiefe: 3

Polygonanzahl: 747

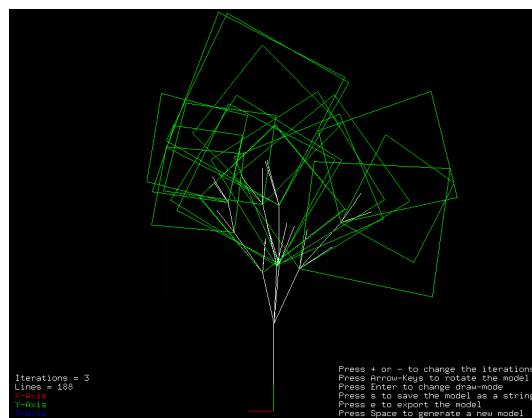


Abbildung 8.1: Skelett-Baum auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung)

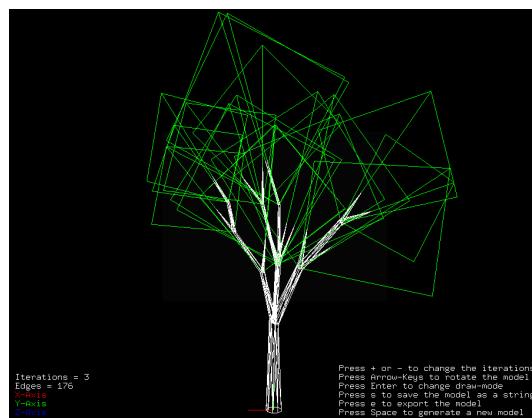


Abbildung 8.2: Drahtgitter-Modell auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung)

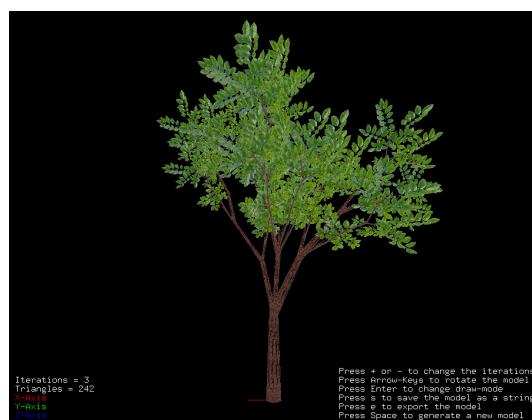


Abbildung 8.3: Texturiertes Modell auf Grundlage der Tree01-Grammatik (Quelle: eigene Darstellung)

Listing 8.2: Tree01-Turtle-String

```

1 F[+25>14@0.77F[+21>13?17F@0.66[+24<16?24FX{L}]-13>12!21FX{L}{L}]{@.79[-17<19!20FX{L}]+23>10!23FX{L}{L}}][+13<180<16@0.69F[-23>21?14F@0
.66[+19>13!22FX[-23>17?19FX{L}]-19>10!23FX{L}]-15>16!22FX{L}{L}]{@.76[+17<10!17FX{L}]+16<25!18FX{L}{L}}][+20<90>15@0.71F[-12>18!18
F@0.8]-18>14!25FX{L}]-16>14!18FX{L}{L}]{@.79[+24<13!23FX{L}]-16<25!20FX{L}{L}}][+15<270<10@0.78F[-12>17?14F@0.63[-23>11?14FX{L
}]]+14<16!23FX{L}{L}]{@.75[+16>16?17FX{L}]-13>14?14FX{L}{L}}

```

8.2.2 Tree02-Grammatik

Listing 8.3: Tree02-Grammatik

```

1 axiom: t
2 angle_z: 10 - 20
3 angle_y: 10 - 20
4 angle_x: 10 - 20
5 rules: t -> F[b][b][o][o], o -> b, o -> x, b -> @rlB
6 branch_shortening: 0.8 - 0.9
7 generations: 10
8 base_length: 20
9 base_radius: 0.3
10 bark_texture: ../resources/bark_02_seamless.jpg
11 leaf_texture: ../resources/leafs02.png

```

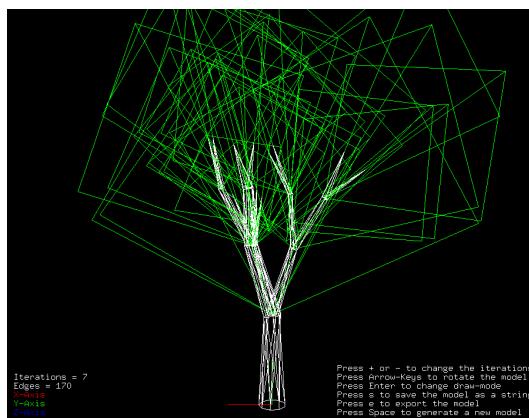


Abbildung 8.4: Drahtgitter-Modell auf Grundlage der Tree02-Grammatik (Quelle: eigene Darstellung)



Abbildung 8.5: Texturiertes Modell auf Grundlage der Tree02-Grammatik (Quelle: eigene Darstellung)

Iterationstiefe: 7

Polygonanzahl: 202

Listing 8.4: Tree02-Turtle-String

```

1 F[@0.89-18<15?11F[@0.8+13>19?15F[@0.82-13>17?18F[b][b][o][o]{L}][@0.87+12>19?18F[b][b][o][o]{L}][@0.82-14<11?18t{L}][x]{L}][@0.9-15>20?15F[
2 @0.84+18>16!15F[b][b][o][o]{L}][@0.83+15<14!15F[b][b][o][o]{L}][x][x]{L}][@0.85+17>14!12F[@0.84+20>15?11F[@0.9-10>16!19F[b
3 ][b][o][o]{L}][@0.81+20<11?16F[b][b][o][o]{L}][@0.87+18<11?19{L}][@0.86+15>14!20t{L}][@0.86-17>16!2F[@0.82-14>18!14F[b][b][o][
4 o]{L}][@0.8-12>11?17F[b][b][o][o]{L}][@0.84-20<19?12t{L}][@0.8+19>19?11t{L}][@0.8-15<12?17F[@0.84+13>18!13F[@
5 .83-18<18!10t{L}][@0.89-19>14?20t{L}][x][x]{L}][@0.8+13<18?20F[@0.88-12>12?19t{L}][@0.86-13<18?20t{L}][b][b]{L}][@0.84-16>19!19F[b][
6 b][o][o]{L}][@0.87+20>10?15F[b][b][o][o]{L}][L]

```

8.2.3 Conifer-Grammatik

Listing 8.5: Conifer-Grammatik

```

1 axiom: Ft
2 angle_z: 150 - 160
3 angle_y: 0 - 10
4 angle_x: 0 - 10
5 rules: t -> @0.7 f[@0.5+[<40(<>)a][<80(<>)a][<120(<>)a][<160(<>)a][<200(<>)a][<240(<>)a][<280(<>)a][<320(<>)a][<360(<>)a]]t, a -> @1.08@0.5F
6 , f -> F, c -> [@0.8rF][@0.8rF][@0.8rF]
7 branch_shortening: 0.1 - 0.3
8 generations: 8
9 base_length: 20
10 num_sides: 6
11 base_radius: 0.1
12 bark_texture: ../resources/conifer_bark.png
13 leaf_texture: ../resources/conifer.png

```

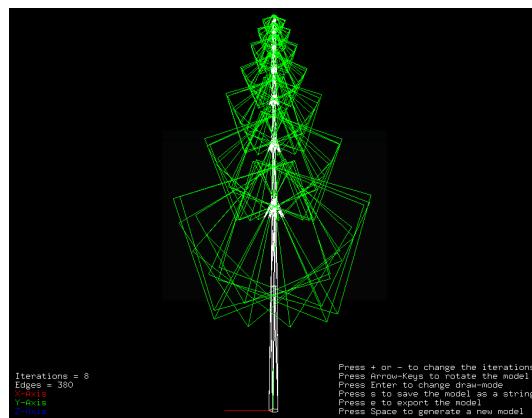


Abbildung 8.6: Drahtgitter-Modell auf Grundlage der Conifer-Grammatik (Quelle: eigene Darstellung)



Abbildung 8.7: Texturiertes Modell auf Grundlage der Conifer-Grammatik (Quelle: eigene Darstellung)

Iterationstiefe: 8

Polygonanzahl: 445

Listing 8.6: Conifer-Turtle-String

```

1 F@0.7F[@0.5+151[<40>4@1.0{L}@0.5F][<80<9@1.0{L}@0.5F][<120<1@1.0{L}@0.5F][<160<7@1.0{L}@0.5F][<200<7@1.0{L}@0.5F][<240<9@1.0{L}@0.5F
][<280<3@1.0{L}@0.5F][<320>0@1.0{L}@0.5F][<360>4@1.0{L}@0.5F]@0.7F[@0.5+154[<40<1@1.0{L}@0.5F][<80<5@1.0{L}@0.5F][<120>5@1.0{L}@0.5
F][<160>7@1.0{L}@0.5F][<200<4@1.0{L}@0.5F][<240>4@1.0{L}@0.5F][<280>4@1.0{L}@0.5F][<320>7@1.0{L}@0.5F][<360>4@1.0{L}@0.5F]]@0.7F[@0
.5+152[<40<3@1.0{L}@0.5F][<80<2@1.0{L}@0.5F]
```

8.2.4 Bush-Grammatik

Listing 8.7: Bush-Grammatik

```

1 axiom: mnop
2 angle_z: 10 - 25
3 angle_y: 10 - 25
4 angle_x: 10 - 25
5 rules: m -> [+(><)X], n -> [+<180(><)X], o -> [+<90(><)X], o -> x, p -> [+<270(><)X], p -> x, X -> @[{rFX@2.0B}]rFX@2.0B
6 branch_shortening: 0.5 - 0.7
7 generations: 3
8 base_length: 4
9 base_radius: 0.06
10 bark_texture: ../resources/bark_seamless.jpg
11 leaf_texture: ../resources/bush.png
```

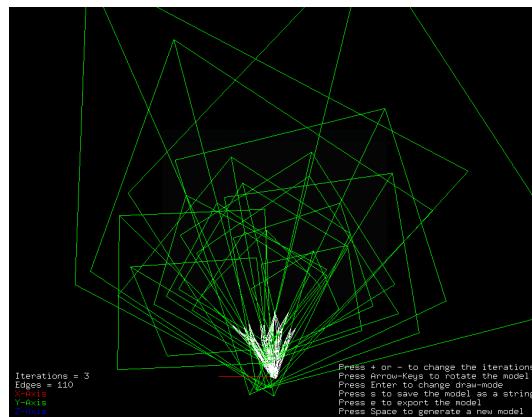


Abbildung 8.8: Drahtgitter-Modell auf Grundlage der Bush-Grammatik (Quelle: eigene Darstellung)

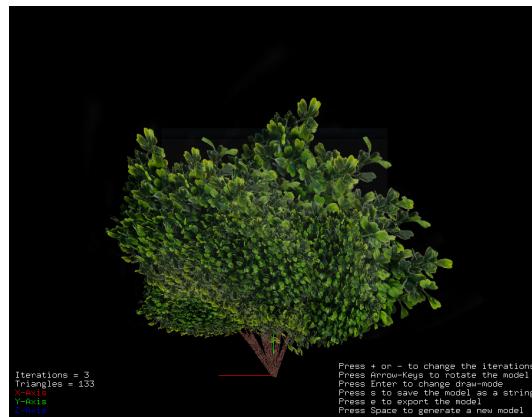


Abbildung 8.9: Texturiertes Modell auf Grundlage der Bush-Grammatik (Quelle: eigene Darstellung)

Iterationstiefe: 3

Polygonanzahl: 133

Listing 8.8: Bush-Turtle-String

```
[+21>19@0.61[−14<10?13F@0.56[−25>21113FX@2.0{L}]+18>14?18FX@2.0{L}]+25>13!16F@0.66[+15<15?18FX@2.0{L}]+23>13!10FX@2.0{L}@2.0{L}
]]+14<180>25@0.64[−17<19!4F@0.61[−15>10?23FX@2.0{L}]+14<10?20FX@2.0{L}]+14<14!11F@0.56[−21>23!3FX@2.0{L}]+14<16!15FX@2.0{L}
}@2.0{L}]+24<90<14@0.52[+19>25?16F@0.58[−14>14?16FX@2.0{L}]+25>11?23FX@2.0{L}]+20>16!10F@0.55[+11>19?10FX@2.0{L}]+11<14?13
FX@2.0{L}@2.0{L}]x
```

8.3 Variationen und Iterationsstufen

Die in Abschnitt 8.2 gezeigten Bäume sind jeweils nur ein Beispiel der zugrundeliegenden Grammatik. Es sind beliebig viele weitere Variationen erzeugbar, indem der Turtle-String zufällig anders generiert wird. In dem Programm erreicht man dies mittels Drücken der Enter-Taste. Auch die Iterationsstufen können für jede Variation geändert werden. Abbildung 8.10 zeigt eine Collage von verschiedenen Variationen und Iterationsstufen für die Tree01-Grammatik. Dies ermöglicht einem potenziellen Nutzer solange die Variation und Iterationsstufen zu wechseln bis ein gewünschtes Exemplar generiert wurde und dann gespeichert oder exportiert werden kann.



Abbildung 8.10: Verschiedene Versionen (Abkürzung V) und Iterationsstufen (Abkürzung I) der Tree01-Grammatik (Quelle: eigene Darstellung)

8.4 Bewertung

Das Programm erzeugt optisch annehmbare Baum-Modelle, welche in einem Videospiel verwendet werden könnten. Die Polygon-Anzahl ist vertretbar. Jedoch ist die Verzweigung teilweise unrealistisch und kantig. Dies liegt zum einen an der Grammatik, welche verbessert werden kann, zum anderen an der Technik wie das 3D-Modell erzeugt wird. Die Texturierung erfolgt nur rudimentär. Die Blätterzeugung funktioniert ausreichend gut, könnte aber auch verbessert werden, da teilweise zu viele Blätter erzeugt werden. Außerdem variiert die Blattgröße zu sehr, sodass auf niedrigen Iterationsstufen die Blätter zu groß werden, während auf höheren Iterationsstufen diese zu klein ausfallen.

9 Fazit und Ausblick

9.1 Fazit

Im Rahmen dieser Arbeit wurde die prozedurale Generierung von 3D-Bäumen mittels L-System vorgestellt. Die Arbeiten von Prusinkiewicz und Lindenmayer [PL90] dienten als Grundlage. Es wurde aufgezeigt wie man aus einer Zeichenkette ein dreidimensionales Modell generieren kann. Die erzeugten Ergebnisse (Vgl. Kapitel 8) zeigen, dass die Anforderungen aus Abschnitt 1.2 erfüllt wurden und somit gezeigt werden konnte, dass dieses System dazu geeignet ist verschiedene Arten von Bäumen und Büschen zu produzieren.

9.2 Ausblick

Das bestehende System kann um einige Techniken erweitert werden, sodass die Nutzbarkeit und die Vielfalt der Ergebnisse eine größere Bandbreite an Nutzungsmöglichkeiten liefern. Folgende Erweiterungen wären eine Bereicherung des Systems.

9.2.1 Grafisches Benutzer-Interface

Um den Umgang mit dem System zu erleichtern wäre ein grafisches Benutzer-Interface von großer Hilfe. So könnten die Regeln der Grammatik sowie weitere wichtige Daten direkt in der Nutzeroberfläche angepasst werden, wodurch die Änderungen sofort erkennbar sind. Auch das Einladen von anderen Grammatiken wäre während der Laufzeit möglich.

9.2.2 Generierung von Level-Of-Details

Die Generierung von LODs würde Spielentwicklern viel Arbeit abnehmen, da diese sonst einen Grafiker beauftragen müssten die verschiedenen Detailstufen eines Baums herauszuarbeiten. Die Generierung der LODs wäre leicht gemacht. Auf Basis des Skelett-Baums müsste das 3D-Mesh mit einer kleineren Seitenzahl pro Segment erzeugt werden. Außerdem können die kleineren Blätter entfernt werden.

9.2.3 Windanimation

Windanimation der Blätter und Äste gehört heute zum Repertoire der Bäume von Videospielen. Diese erhöht die Glaubwürdigkeit und Immersion der Bäume. Die Umsetzung dieser Aufgabe bedarf jedoch einer Überarbeitung der Software.

9.2.4 Wachstumssimulation

Strategiespiele und Sandbox-Simulationen verlangen oft die Simulation von Wachstum der Vegetation. Der Spieler oder NPCs pflanzen beispielsweise einen Baum und der Spieler sieht wie dieser im Lauf der Zeit heranwächst und im Winter seine Blätter verliert.

Literaturverzeichnis

- [Aca] ACADEMIC (Hrsg.): *Blumenkohl*. http://de.academic.ru/pictures/dewiki/66/Blumenkohl_fraktal.jpg, Abruf: 30. September. 2017
- [Ach] ACHIM UND KAI: *Julia Menge*. http://www.achim-und-kai.de/kai/fuc/fuc_jul.html, Abruf: 30. September. 2017
- [alta] ALTERNATIVETO: *CityEngine*. https://d2.alternativeto.net/dist/s/1754e2a5-3830-e011-a433-0200d897d049_2_full.png, Abruf: 12. Oktober. 2017
- [altb] ALTERNATIVETO: *Voronoi-Textur*. https://d2.alternativeto.net/dist/s/1754e2a5-3830-e011-a433-0200d897d049_2_full.png, Abruf: 12. Oktober. 2017
- [Aut90a] AUTODESK (Hrsg.): *3D Studio Max*. 1990
- [Aut90b] AUTODESK (Hrsg.): *Autodesk Maya*. 1990
- [bil] BILLDESOWITZ: *Simulation mit MASSIVE*. <http://billdesowitz.com/wp-content/uploads/2012/08/massive5.0.png>, Abruf: 12. Oktober. 2017
- [biu17] BUNDESVERBAND INTERAKTIVE UNTERHALTUNGSSOFTWARE (Hrsg.): *Jahres-report der Computer- und Videospielbranche in Deutschland 2017*. https://www.biu-online.de/wp-content/uploads/2017/09/BIU_Jahresreport_2017_interaktiv.pdf. Version: 2017, Abruf: 30. September. 2017
- [Bob67] DANIEL G. BOBROW AND WALLY FEURZEIG AND SEYMOUR PAPERT AND CYNTHIA SOLOMO (Hrsg.): *Logo*. 1967
- [Cho56] CHOMSKY, Noam: Three models for the description of language. In: *IRE Transactions on information theory* 2 (1956), Nr. 3, S. 113–124
- [cle11] CLEARING COMPLEXITY (Hrsg.): *Farn*. https://clearingcomplexity.files.wordpress.com/2011/11/fractal_ifs_barnsleys_fern.png?w=640&h=392&crop=1. Version: 2011, Abruf: 30. September. 2017
- [DW15] DUFFY, J. ; WANG, Z.: Application of Procedural Generation as a Medical Training Tool. In: *Int'l Conf. Health Informatics and Medical Systems*, 2015, S. 223
- [Far07] FARRELL, Andy: An introduction to procedural audio and its application in computer games. In: *Audio mostly conference* Bd. 23, 2007
- [gam80] EPYX (Hrsg.): *Rogue: Exploring the Dungeons of Doom*. 1980
- [gam84] ACORN SOFTWARE (Hrsg.): *Elite*. 1984
- [gam85] NINTENDO (Hrsg.): *Super Mario Bros.* 1985

- [gam89] MAXIS, ELECTRONIC ARTS (Hrsg.): *Sim City*. 1989
- [gam94] MICROPROSE (Hrsg.): *UFO: Enemy Unknown*. 1994
- [gam96] BLIZZARD ENTERTAINMENT (Hrsg.): *Diablo*. 1996
- [gam08a] MOSSMOUTH (Hrsg.): *Spelunky*. 2008
- [gam08b] EA GAMES (Hrsg.): *Spore*. 2008
- [Hel16] HELLO GAMES: *No Man's Sky*. 2016
- [hel17] <http://www.hellogames.org/>
- [HMVDVI13] HENDRIKX, Mark ; MEIJER, Sebastiaan ; VAN DER VELDEN, Joeri ; IOSUP, Alexandru: Procedural Content Generation for Games: A Survey. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9 (2013), Februar, Nr. 1, 1:1–1:22. <http://dx.doi.org/10.1145/2422956.2422957>. – DOI 10.1145/2422956.2422957. – ISSN 1551–6857
- [Hul15] HULICK, Kathryn: *Artificial Intelligence*. ESSENTIAL LIB, 2015. – ISBN 9781624039126
- [IDV09] INTERACTIVE DATA VISUALIZATION, INC. (Hrsg.): *SpeedTree*. 2009
- [IGN] IGN: *Landschaftsgenerierung in No Man's Sky*. http://sm.ign.com/ign_de/news/n/no-mans-sky-no-mans-sky-spieler-verbringt-30-stunden-auf-seinem-heimatplcpu2.jpg, Abruf: 12. Oktober. 2017
- [itw16] IT WISSEN (Hrsg.): *Romanesco*. <http://www.itwissen.info/lex-images/fraktal-modell-quadsoft-dotorg.png>. Version: 2016, Abruf: 30. September. 2017
- [Kla95] KLAUS TEUBER ; KOSMOS (Hrsg.): *Die Siedler von Catan*. 1995
- [KM06] KELLY, George ; McCABE, Hugh: A survey of procedural techniques for city generation. In: *The ITB Journal* 7 (2006), Nr. 2, S. 5
- [Lin68] LINDENMAYER, Aristid: Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. In: *Journal of theoretical biology* 18 (1968), Nr. 3, S. 280–299
- [Man80] MANDELBROT, Benoit B.: Fractal aspects of the iteration of $z \rightarrow \lambda z(1-z)$ for complex λ and z . In: *Annals of the New York Academy of Sciences* 357 (1980), Nr. 1, S. 249–259
- [MJ15] MOORE JR, Osler K.: Procedural Content Generation: Using AI to Generate Playable Content. (2015)
- [mob] MOBYGAMES: *Rogue*. <https://www.mobygames.com/images/shots/l/198799-rogue-trs-80-coco-screenshot-game-play-in-80-columns-hardware.gif>, Abruf: 12. Oktober. 2017

- [Pas08] PASCAL MUELLER, SIMON HAEGLER, ANDREAS ULMER, SIMON SCHUBIGER, MATTHIAS SPECHT, STEFAN MÜLLER ARISONA, BASIL WEBER (Hrsg.): *CityEngine*. 2008
- [Per85] PERLIN, Ken: An Image Synthesizer. In: *SIGGRAPH Comput. Graph.* 19 (1985), Juli, Nr. 3, 287–296. <http://dx.doi.org/10.1145/325165.325247>. – DOI 10.1145/325165.325247. – ISSN 0097–8930
- [PL90] PRUSINKIEWICZ, Przemyslaw ; LINDENMAYER, Aristid: *The algorithmic beauty of plants*. Springer Science & Business Media, 1990
- [PM01] PARISH, Yoav I. ; MÜLLER, Pascal: Procedural modeling of cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* ACM, 2001, S. 301–308
- [pro] PROFEASY: *Die Siedler von Catan.* [http://www.profeeasy.de/Siedler-Anleitung/assets/images/ohne_Nummernplaettchen_K3.jpg](http://www.profeasy.de/Siedler-Anleitung/assets/images/ohne_Nummernplaettchen_K3.jpg), Abruf: 12. Oktober. 2017
- [Ree83] REEVES, William T.: Particle systems—a technique for modeling a class of fuzzy objects. In: *ACM Transactions on Graphics (TOG)* 2 (1983), Nr. 2, S. 91–108
- [Sim] LINKÖPING UNIVERSITY (Hrsg.): *Simplex noise demystified*. <http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>, Abruf: 30. September. 2017
- [SSG⁺17] SUMMERVILLE, Adam ; SNODGRASS, Sam ; GUZDIAL, Matthew ; HOLMGÅRD, Christoffer ; HOOVER, Amy K. ; ISAKSEN, Aaron ; NEALEN, Andy ; TOGELIUS, Julian: Procedural Content Generation via Machine Learning (PCGML). In: *arXiv preprint arXiv:1702.00539* (2017)
- [STN16] SHAKER, Noor ; TOGELIUS, Julian ; NELSON, Mark J.: *Procedural Content Generation in Games*. Springer, 2016
- [Tar06] TARN ADAMS ; BAY 12 GAMES (Hrsg.): *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*. 2006
- [Tho10] THOMAS, W.: Procedural Content Generation via Machine Learning (PCGML). In: *Informatik Spektrum* 33 (2010), S. 504
- [Thu10] THUE, Axel: *Die Lösung eines Spezialfalles eines generellen logischen Problems, von Axel Thue...* J. Dybwad, 1910
- [TKSY11] TOGELIUS, Julian ; KASTBJERG, Emil ; SCHEDL, David ; YANNAKAKIS, Georgios N.: What is procedural content generation?: Mario on the borderline. In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* ACM, 2011, S. 3
- [tro82] DISNEY-STUDIOS (Hrsg.): *Tron*. 1982
- [twe] TWEAKPC: *UFO: Enemy Unknown*. http://www.tweakpc.de/gallery/data/563/medium/xcom_1.jpg, Abruf: 12. Oktober. 2017

- [vox] VOX-CDN: *Super Mario Bros.* [https://cdn.vox-cdn.com/thumbor/mQVeHmMDCRQdZXPIwdGgAaYhKQU=/0x0:2000x1500/1200x800/filters:focal\(840x590:1160x910\):no_upscale\(\)/cdn.vox-cdn.com/uploads/chorus_image/image/56594937/super-mario-bros.0.0.gif](https://cdn.vox-cdn.com/thumbor/mQVeHmMDCRQdZXPIwdGgAaYhKQU=/0x0:2000x1500/1200x800/filters:focal(840x590:1160x910):no_upscale()/cdn.vox-cdn.com/uploads/chorus_image/image/56594937/super-mario-bros.0.0.gif), Abruf: 12. Oktober. 2017
- [Wal10] WALLBAUM, Torben: *Procedural Generation. Natural Environments out of the Box.* 2010
- [wik05a] WIKIPEDIA FOUNDATION (Hrsg.): *Mandelbrot*. https://de.wikipedia.org/wiki/Fraktal#/media/File:Mandelbrot_set_with_coloured_environment.png. Version: 2005, Abruf: 30. September. 2017
- [wik05b] WIKIPEDIA FOUNDATION (Hrsg.): *Perlin Noise*. https://commons.wikimedia.org/wiki/File:Perlin_noise.jpg. Version: 2005, Abruf: 30. September. 2017
- [wik06] WIKIPEDIA FOUNDATION (Hrsg.): *Koch-Schneeflocke*. https://de.wikipedia.org/wiki/Fraktal#/media/File:Koch_Snowflake_7th_iteration.svg. Version: 2006, Abruf: 30. September. 2017
- [wik07] WIKIPEDIA FOUNDATION (Hrsg.): *Partikelsystem*. https://en.wikipedia.org/wiki/Particle_system#/media/File:Particle_sys_fire.jpg. Version: 2007, Abruf: 30. September. 2017
- [Wor96] WORLEY, Steven: A cellular texture basis function. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* ACM, 1996, S. 291–294
- [You13] ALEXEY ZEAL (Hrsg.): *Youtube-Video vom Spore-Editor*. <https://www.youtube.com/watch?v=WfV6G5qn7HE>. Version: 2013, Abruf: 30. September. 2017
- [yti] YTIMG: *SpeedTree*. <https://i.ytimg.com/vi/pFy8kCMdymM/maxresdefault.jpg>, Abruf: 12. Oktober. 2017
- [zed] ZEDSPACE: *PlantFactory*. <https://zedspace.files.wordpress.com/2013/02/screen-shot-2013-02-15-at-3-58-14-pm.png>, Abruf: 12. Oktober. 2017

Anhang A

Inhalt der DVD

A.1 Abbildungen der Ergebnisse

Pfad: /Abbildungen der Ergebnisse ... Abbildungen der Ergebnisse

A.2 Abschlussarbeit

Pfad: /Abschlussarbeit ... Originalfassung der Bachelorarbeit als TeX-Quelldateien

Pfad: /BachelorarbeitLesser2017.pdf ... Bachelorarbeit im PDF-Format

Pfad: /Abschlussarbeit/images ... Rohdaten der benutzten Abbildungen

A.3 Kurzfassungen

Pfad: /Kurzfassungen ... Kurzfassungen auf deutsch und englisch im Originalformat und als PDF

A.4 Literaturquellen

Pfad: /Literatur ... Kopien der benutzten Literatur als PDF-Dateien

A.5 Poster

Pfad: /Poster ... Posterentwurf in der Originalfassung

A.6 Präsentation

Pfad: /Präsentation ... Originalfassung des Vortrags als PowerPoint-Datei

A.7 Source-Code

Pfad: /Source-Code ... Dokumentierter Python-Source-Code

Anhang B

Ausführungsanleitung

B.1 Python und Editor

Um den Python-Source-Code ausführen zu können benötigt man Python 2.7.14:

Python-Installation

- Windows: <https://www.python.org/downloads/>
- Linux: sudo apt-get install python

Des Weiteren wird ein Editor, der Python unterstützt, benötigt. Es wird empfohlen den Editor Pycharm zu nutzen. Diesen gibt es in einer kostenlosen Version auf der Webseite <https://www.jetbrains.com/pycharm/>.

B.2 PyGame

Mittels des Paketverwaltungsprogramms von Python namens *pip* muss PyGame installiert werden:

PyGame-Installation

- Windows & Linux: pip install pygame

B.3 PyOpenGL

Außerdem muss PyOpenGL installiert werden:

PyOpenGL-Installation

- Windows & Linux: pip install PyOpenGL PyOpenGL_accelerate

Selbstständigkeitserklärung

Ich, Martin Lesser, versichere hiermit, dass ich die vorliegende Bachelorarbeit mit dem Thema

Prozedurale Generierung von 3D-Bäumen

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Erfurt, 19. Oktober 2017

Martin Lesser