Hochschule Worms, Fachbereich Informatik

Studiengang: Angewandte Informatik

Bachelorarbeit

Anwendung der Prozeduralen Generierung zum Erstellen einer Fortschrittsanzeige in Form wachsender Abbilder realer Baumarten in 2D

Jonas Steinbach

Abgabe der Arbeit: 5. Juni 2017

Betreut durch:

Prof. Dr. Alexander Wiebel, Hochschule Worms

Zusammenfassung

Diese Arbeit beschreibt die Probleme und Überlegungen beim Anwenden der Prozeduralen Generierung zum Erstellen wachsender Abbilder realer Baumarten in 2D. Bisher werden die Abbilder nur manuell erstellt, was teuer und zeitaufwändig ist. Um das Erstellen zu automatisieren und beschleunigen wird das Anwenden der Prozeduralen Generierung in Erwähnung gezogen. Dabei wird ein kurzer Überblick über die Prozedurale Generierung mit Fokus auf das Erstellen von Bäumen gegeben, welcher dann durch Beschreibung und Erklärung von L-Systemen vertieft wird. Anschließend wird ein Programm beschrieben, dass das die Prozeduralen Generierung in Form von L-Systemen zum Erzeugen von Bäumen anwendet. Dabei werden Probleme beim Transformieren von Punkten im zweidimensionalen Raum gelöst. Zum Schluss werden die mit dem Programm erzeugten Grafiken vorgestellt und diskutiert, worauf eine Zusammenfassung und ein Ausblick auf Verbesserungen des Programms und der Generation gegeben wird.

Abstract

This work describes the problems and considerations for the application of procedural generation to the generation of growing images of real tree species in 2D. So far, the images are only created manually, which is expensive and time-consuming. To automate and accelerate the generation, the usage of procedural generation is considered. For this, a short overview over procedural generation with focus on generation of trees is given, which then deepens into the description and explanation of l-systems. Afterwards, a program that applies the procedural generation in form of l-systems for the generation of trees is described. Thereby, problems with the transformation of points in the two-dimensional space are solved. At the end, the created graphical images are being presented and discussed, after which a summary and outlook of the improvement of the program and the generation are given.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codebeispielverzeichnis

Symbolverzeichnis

Symbol	Bedeutung
V	Alphabet eines L-Systems
V^*	Menge aller Wörter über V
V^+	Menge aller nicht-leerer Wörter über V
ω	Axiom eines L-Systems
P	Menge an Regeln eines L-Systems
p_n	Regel eines L-Systems
a	Zeichen oder *predecessor* eines L-Systems
χ	Wort oder *successor* eines L-Systems

Abkürzungsverzeichnis

PG	Prozedurale Generierung
PCG	Procedural Content Generation
AVD	Android Virtual Device
API	Application Programming Interface
PX	Pixel
DPI	Dots per Inch

1 Einleitung

1.1 Vorstellung des Kontexts der Arbeit

Die vorliegende Arbeit behandelt das Anwenden der Prozeduralen Generierung zum Erzeugen und Darstellen von wachsenden Abbildern von Bäumen und ist in das Feld der Computergrafik einzuordnen [Sta+14, S.119]. Als Computergrafik wird allgemein das Erzeugen, Verändern, Manipulieren, Analysieren und Interagieren mit grafischen Repräsentationen von Objekten und Daten mit Computern bezeichnet [Dai08], wobei der Fokus dieser Arbeit aber hauptsächlich auf dem Erstellen von Grafiken liegt.
Entstanden ist die Arbeit im Rahmen der Entwicklung einer Bildungs-App „uRnature“ [UDA17b] für das Umwelt- und Bildungs-Unternehmen UDATA GmbH in Neustadt an der Weinstraße [UDA17a]. Gefördert wurde die App durch den Waldklimafonds der Bundesministerien für Landwirtschaft sowie Umwelt (BMEL und BMUB), Förderkennzeichen 28WC506201 [Wal].
Die App behandelt den Klimawandel in deutschen Wäldern und besteht aus ortsunabhängigen Mini-Spielen sowie ortsbasierte Missionen. Dabei soll dem Anwender durch interaktives Veranschaulichen vor allem Handlungs- und Faktenwissen zum Klimawandel vermittelt werden. Für die Inhalte der App ist es wichtig, verschiedene einfache Baumgrafiken, also Abbilder von realen Baumarten zu erstellen. Bisher werden diese noch zeitaufwändig von Hand erstellt. Der Einsatz von prozeduralen Verfahren soll das Erstellen dieser Grafiken beschleunigen (siehe Zielsetzung/Motivation). Weiterhin erhofft sich das Unternehmen durch den Einsatz von prozeduralen Techniken ein einfacheres Modifizieren und Darstellen von Bäumen in verschiedenen Entwicklungsstadien. Dies ist zum Beispiel für die interaktive Veranschaulichung relevant und eröffnet eine didaktisch neue Art der Vermittlung der Natur an junge Menschen [DL01, S.65].

1.2 Zielsetzung/Motivation

Wie in 1.1 beschrieben werden die für die App benötigten Baum-Grafiken bisher nur von Hand erstellt. Dies ist zeitintensiv und teuer, da für das sachgemäße Erstellen von Grafiken eine Person mit Grafikausbildung beschäftigt werden muss [STN14]. Weiterhin ist es für die interaktiven Inhalte der App oft erforderlich, schnell viele verschiedene Varianten eines Baumes oder einen einzelnen Baum in verschiedenen Wachstumsstadien zu zeigen, was einen weiteren Mehraufwand für die Person mit Grafikausbildung bedeutet. Auch zeigt sich dabei das manuelle Erstellen der Baum-Grafiken als „Flaschenhals“ für die Entwicklung, da die Entwickler eventuell auf relevante Grafiken warten müssen.

Ein weiteres Problem beim Verwenden manuell erstellter Grafiken ist, dass diese unflexibel sind und sich zur Laufzeit nicht verändern lassen: Bei einem der geplanten Spiele soll der Anwender sich einen eigenen Baum aus vorgegebenen Baum-Bestandteilen wie Blättern, Stamm und Früchten erzeugen lassen. Dabei werden für die verschiedenen Bestandteile Grafiken erzeugt und wie bei einem Schiebepuzzle übereinander gelegt um einen eigenen Baum zu bilden. Beim Wechseln zwischen Laub- und Nadelbäumen passiert es aber, dass die Blätter von Laubbäumen eine Krone bilden, während Nadelbäume ihre Blätter seitlich, am Stamm ausgerichtet, tragen. Folglich müssen für Laub- und für Nadelbäume verschiedene Varianten der Blatt-Grafik angefertigt werden. Auch müssen die Baum-Bestandteile als Grafiken genau übereinander passen, um ein problemloses Austauschen und Aneinanderlegen zu erlauben. Dabei können aber irreguläre Wuchsformen der Bäume, wie zum Beispiel die Zwiesel, nicht beachtet werden, und führen jeweils zu weiteren Varianten an Grafiken.
Weiterhin entsteht bei Verwendung von vielen verschiedenen Grafiken in der App ein Speicher-Problem, da mobile Endgeräte im Gegensatz zu Desktop-Computern nur über eine limitierte, schwer erweiterbare Speicherkapazität verfügen (So kann das neue Smartphone von Samsung nur mit max. zwei SD-Karten erweitert werden [Sam]):
Für das geplante Spiel mit fünf Bäumen à drei Bestandteilen werden mindestens 30 Grafiken benötigt. Damit diese Grafiken auf verschiedenen mobilen Endgeräten fehlerfrei dargestellt werden können, müssen von ihnen für die verschiedenen Display-Varianten jeweils alternative Versionen erstellt werden [And17i]. Um die vier häufigsten Display-Varianten (siehe 4.2) abzudecken werden also mindestens 120 Grafiken gebraucht. Dabei werden alle erstellten Grafiken in der installierten App als Bild-Ressourcen heruntergeladen und gespeichert. Die maximale Größe einer App im Google Playstore ohne Erweiterungsdatei beträgt zur Zeit (25. Mai 2017) 100 MB [And17e]. Wenn die erstellten Grafiken nach ihrer Optimierung im Schnitt 10kB groß sind, verbrauchen die Grafiken der fünf Bäume zusammen 1,2 MB, also mehr als 1% des verfügbaren Speichers. Laut Google vermeiden Benutzer oft das Herunterladen von Apps die zu groß wirken [And17h], es besteht also auch ein finanzielles Interesse an der Reduzierung der Größe der App.

Aus dieser für die Entwicklung ökonomisch als auch logistisch inadäquaten manuellen Erzeugung der benötigten Baum-Grafiken formuliert sich der Bedarf nach einer schnelleren, günstigeren, einfacheren und effizienteren Methode.
Als potentielle Lösung für die zuvor genannten Probleme bietet sich die Prozedurale Generierung an [And17h]. Dabei werden die benötigten Baum-Grafiken zur Laufzeit im Programm erzeugt und angepasst, womit die zuvor formulierten Probleme umgangen werden. Aus diesem

Lösungsansatz formulieren sich die folgenden Fragen:

Können die benötigten Baum-Grafiken prozedural generiert werden?
Können die benötigten Baum-Grafiken ohne Eingaben des Benutzers generiert werden?
Können die benötigten Baum-Grafiken in verschiedenen Varianten erzeugt werden?
Können die benötigten Baum-Grafiken in verschiedenen Wachstumsstadien dargestellt werden?

Diese Fragen sollen in Form einer Anwendung, die verschiedene prozedural generierte Baum-Grafiken als Abbilder realer Bäume in verschiedenen Stadien des Wachstums anzeigt, geklärt werden. Weiterhin soll die Anwendung als Fortschrittsanzeige fungieren, da dies eine weitere geplante Funktionalität ist: Der Benutzer erspielt sich in den verschiedenen Mini-Spielen Punkte. Hat er genügend Punkte gesammelt, wird ein echter Baum gepflanzt. Dabei wird sein Fortschritt zum echten Baum in der App durch das Wachstum eines virtuellen Baums symbolisiert.

Die resultierende zusammenfassende Forschungsfrage lautet:

Wie kann man die Prozedurale Generierung zum Erstellen einer Fortschrittsanzeige in Form wachsender Abbilder realer Baumarten anwenden?

1.3 Aufgabenstellung

Um das Umfeld der Arbeit einzugrenzen und die in 1.2 formulierte Aufgabenstellung zu präzisieren, ist es erforderlich die Arbeit in folgende Aufgaben und Nicht-Aufgaben zu unterscheiden:

Es ist Aufgabe

- die Prozedurale Generierung zum Erzeugen von Abbildern realer Baumarten anzuwenden.
- die erzeugten Abbilder in 2D darzustellen.
- die erzeugten Abbilder in verschiedenen Wachstumsstadien in Form einer Fortschrittsanzeige anzuzeigen.

Es ist NICHT Aufgabe

- einen neuen prozeduralen Algorithmus zu entwickeln.
- einen bestehenden prozeduralen Algorithmus zu verbessern.
- Prozedurale Algorithmen zu vergleichen.

- den effizientesten/besten Algorithmus zu finden.
- Baumstrukturen zu vergleichen.
- das fachlich-korrekte Erzeugen realer Baumarten sicherzustellen.
- das fachlich-korrekte Darstellen realer Baumarten sicherzustellen.
- Bäume zu modellieren, sondern Abbilder von Bäumen zu erzeugen und darzustellen.

1.4 Aufbau der Arbeit

Diese setzt grundlegende Kenntnisse der Informatik und Computergrafik voraus. Weiterhin wird empfohlen, mit Java und Android vertraut zu sein, da in dieser Arbeit das Erstellen einer App für Android beschrieben wird.
Um die Arbeit in den richtigen Kontext einordnen zu können, werden in den ersten beiden Kapiteln die Aufgabenstellung so wie die notwendigen Grundlagen der Prozeduralen Generierung vorgestellt, bevor diese Grundlagen im dritten Kapitel vertieft werden. Dabei wird die Theorie und Anwendung von L-Systemen erklärt.
Im vierten Kapitel wird dann die Entstehung und Funktionsweise des Programms beschrieben, worauf eine Vorstellung und Diskussion der Ergebnisse folgt.
Zum Schluss wird das Erarbeitete zusammengefasst und ein Ausblick über weitere Verbesserungen und Problemstellungen gegeben.

2 Prozedurale Generierung

Die Prozedurale Generierung (PG) bildet ein Kernelement dieser Arbeit. Um die nachfolgenden Kapitel besser zu verstehen, ist es wichtig, Begriff und Verfahren der Prozeduralen Generierung zuvor zu klären.

2.1 Was ist Prozedurale Generierung?

Bei der Prozeduralen Generierung werden Inhalte nicht vom Menschen entworfen, sondern „prozedural“, d.h. durch eine vorgegebene (programmatische) Prozedur erzeugt [STN14] [Wik17]. Dabei ist die wichtigste Eigenschaft der Prozeduralen Generierung dass sie ein Element, sei es eine Geometrie, Textur oder ein Effekt, nicht als statischen Block an Daten sondern in Form einer Sequenz an Generierungs-Anweisungen erzeugt. Diese Anweisungen können dann aufgerufen werden um Instanzen des Elements zu erstellen. Auch kann die Beschreibung parametrisiert werden um unterschiedliche Instanzen zu erzeugen [KM06, S.2]. Ein bekanntes Beispiel für Prozedurale Generierung ist das Spiel .kkrieger, ein funktionierender Shooter mit einer Dateigröße von nur 96kb [the04]

2.2 Procedural Content Generation

Neben dem Begriff „procedural generation“ (PG) wird auch häufig der Begriff „procedural content generation“ (PCG) verwendet. Im Gegensatz zu der in 2.1 definierten prozeduralen Generierung bezeichnet PCG nur das Anwenden der Prozeduralen Generierung zum Erstellen von Computerspiel-Inhalten (game content), wobei als Inhalt das Meiste was in Spielen vorkommt zählt, sei es Level, Karte, Spielregeln, Texturen, Geschichten, Items, Quests, Musik, Waffen, Fahrzeuge oder Charaktere [STN14].
So wird „procedural content generation“ auch als das algorithmische Generieren von Spielinhalten mit limitierter oder indirekter Eingabe des Anwenders definiert, also als Computer-Programm das Spielinhalte alleine oder zusammen mit menschlichen Einfluss generiert [Tog+11] [STN14]. In vielen Fällen sind die beiden Begriffe PG und PCG austauschbar und bilden ein Kontinuum [PCG14].
Im weiteren Verlauf dieser Arbeit wird hauptsächlich der Begriff Prozedurale Generierung verwendet, obwohl die erzeugten Grafiken auch in den Bereich der Prozeduralen Content Generierung fallen.

2.3 Anwendungsgebiete

Die Prozedurale Generierung ist vielseitig und kommt in vielen verschiedenen Anwendungsgebieten, wie zum Beispiel in Filmen und Computer-Spielen, beim Erstellen von Werbung oder beim Visualisieren von Architektur zum Einsatz [GK08, S.1]. So wurden zum Beispiel die Bäume im Film „Der Herr der Ringe 2“ sowie in den Computer-Spielen „Witcher3“, „Destiny“ und „Far Cry 4“ prozedural generiert [AP03] [Spe].

2.4 Prozedurale Generierung speziell für Pflanzen und Bäume

Zum Modellieren und Erzeugen der Bäume werden L-Systeme, ein in den letzten Jahren als Bedeutend hervorgegangener Formalismus zum Modellieren der Entwicklung von Pflanzen und Bäume verwendet [Bou+12, S.1].
L-Systeme sind ein in der Computergrafik häufig verwendeter Ansatz zum Erstellen komplexer botanischer Bäume und stellen die Entwicklung von Pflanzen über einen Zeitraum dar [Pru+01, S.28] [GK08, S.1]. Sie sind in der Lage eine große Menge verschiedener und komplexer Bäumen aus einer geringen Menge an Parametern zu erzeugen [GK08, S.1] [Sta+14, S.118].

3 L-Systeme

Im vorherigen Kapitel wurde ein Überblick über die Prozedurale Generierung geschaffen und dabei L-Systeme in 2.4 für das in 1.2 und 1.3 beschriebene Vorhaben (dem prozeduralen Erzeugen von wachsenden Abbildern realer Baumarten) gewählt. In diesem werden L-Systeme mit Fokus auf das Erstellen von Pflanzen und Bäumen näher beschrieben.

3.1 Was sind L-Systeme?

Lindenmayer-Systeme sind ein ursprünglich 1968 von dem Biologen Aristid Lindenmayer in [Lin68] eingeführter und nach ihm benannter mathematischer Formalismus zum Beschreiben und Modellieren von einfachen mehrzelligen Organismen (wie z.B. Algen und Bakterien) [PH89] [PL90] [HP93, S.14] [Pru+95, S.1] [Pru+96, S.1] [Pru+97, S.1]. Aufgrund seiner nahen Verwandschaft mit abstrakten Automaten und formalen Sprachen zog dieser als L-System bekannt gewordene Formalismus schnell die direkte Aufmerksamkeit der theoretischen Informatik auf sich, was zur Folge hatte, dass L-Systeme ausgeweitet und zum Modellieren von Pflanzen und komplexen Verzweigungs-Strukturen angewandt wurden [PL90] [Pru+95, S.2] [Pru+96, S.1].
1984 stellte Alvy Ray Smith dann in [Alv84] die Verbindung zwischen L-Systemen und Fraktalen her und machte damit das Erzeugen von komplexen Pflanzen als Computergrafiken möglich [PH89]. Weiterhin beobachtete Smith dabei das Phänomen der *data-base amplification*, dem bei L-Systemen inhärenten Erstellen von komplexen Strukturen aus kompakten Datensätzen, welches den Eckpfeiler der Anwendung von L-Systemen bei der Bildersynthese bildet [Pru+96, S.1].
Nach der Einbindung von geometrischen Eigenschaften in die L-Systeme wurden diese detailliert genug, realistische Pflanzen zu beschreiben und in der Computergrafik abzubilden [PL90, S.1]. Weiterhin können L-Systeme beispielsweise zur Generierung von *tilings*, der Reproduktion von geometrischer Kunst aus Ost-Indien und der Synthese von Musik-Noten auf Grundlage von Fraktal-Interpretation angewandt werden [PH89].

3.2 Funktionsweise von L-Systemen

Das Kernprinzip von L-Systemen ist das Ersetzen [PL90, S.1]. Dabei werden die zu beschreibenden Organismen und Pflanzen in einzelne Module (z.B. Verzweigung, Spitze, Blüte oder Ast) unterteilt und in jedem Entwicklungsschritt alle parallel durch entsprechende Nachfolge-Module ersetzt, mit dem Ziel die Entwicklung der Pflanze als Ganzes zu beschreiben [Pru+95, S.3].

Die Module entstammen alle einem endlichen Alphabet aus Modul-Typen, was das Beschreiben des Verhaltens einer beliebig langen Konfiguration an Modulen durch das Benutzen eines endlichen Sets von *Ersetzungs-Regeln* oder *productions* ermöglicht [Pru+96, S.3]. Im Normalfall werden für L-Systeme *strings* verwendet, da diese die am häufigsten untersuchten und am besten verstandenen Ersetzungs-Systeme sind [PL90, S.2].
Eine ausführliche Beschreibung von L-Systemen und ihren Anwendungen zur Modellierung von Pflanzen findet sich in [PL90], an die sich auch die nachfolgenden Abschnitte anlehnen.

3.3 Aufbau von L-Systemem

L-Systeme bestehen aus verschiedenen Komponenten, die hier im Einzelnen erklärt werden.

3.3.1 Alphabet

Das Alphabet enthält alle Zeichen des Systems und bildet die Grundlage der Beschreibung.
Formal wird das Alphabet eines L-Systems als V bezeichnet, wobei V^* die Menge aller Wörter über V und V^+ die Menge aller nicht-leerer Wörter über V ist [PL90, S.4].

Beispiel: Ein D0L-System (die einfachste Art von L-Systemen, siehe 3.4.1) besteht aus den Buchstaben *a* und *b*, die wiederholt in einem Wort (*string*) vorkommen können. Dabei ist jeder Buchstabe mit einer Produktionsregel (siehe 3.3.3) verbunden[PL90, S.3].

3.3.2 Axiom

Das Grund- oder Startwort des L-Systems wird *axiom* genannt. Es bildet die ursprüngliche Beschreibung des Objekts (der Pflanze) und ist zugleich Ausgangsposition für den Ersetzungsprozess [PL90, S.3].
Formal wird das Startwort *axiom* als nicht-leeres Wort $\omega \in V^+$ geschrieben [PL90, S.4].

Beispiel: Das folgende D0L-System beginnt mit einem einzelnen Buchstaben *b*.

$$\omega \quad : \quad \text{b}$$

3.3.3 Regeln

Um ein Wort zu entwickeln, werden Produktionsregeln, auch *productions* genannt, benutzt, die Instruktionen für das Ersetzen enthalten. Sind keine Regeln für ein Zeichen definiert, wird das Zeichen mit sich selbst ersetzt [HP93, S.15].
Formal wird eine Produktionsregel *production* (Annahme: D0L-System) als $(a, \chi) \in P$ geschrieben, wobei $P \subset V \times V^*$ eine endliche Menge an *productions* ist. Dabei wird das Zeichen a als Vorgänger *predecessor* und das Wort χ als Nachfolger *successor* der *production* bezeichnet [PL90, S.4].

Beispiel: Das in 3.3.1 genannte D0L-System besitzt die Regel $a \rightarrow ab$, die aus jedem a ein ab und die Regel $b \rightarrow a$, die aus jedem b wieder ein a macht [PL90, S.3].

$$p_1 \ : \ \text{a} \rightarrow \text{ab}$$
$$p_2 \ : \ \text{b} \rightarrow \text{a}$$

3.3.4 Zusammenfassung

Zusammen bilden die in 3.3.1, 3.3.2 und 3.3.3 beschriebenen Beispiel-Komponenten das folgende D0L-System:

$$\begin{aligned} \omega &: \text{b} \\ p_1 &: \text{a} \rightarrow \text{ab} \\ p_2 &: \text{b} \rightarrow \text{a} \end{aligned}$$

Eine zeitliche Anwendung erzeugt die in 1 abgebildete Beschreibung.

3.4 Arten von L-Systemen

Es gibt verschiedene Arten von L-Systemen, die unterschiedlich komplex sind und für verschiedene biologische Anwendungsgebiete ausgelegt sind.

3.4.1 Kontextfreie L-Systeme

0L-Systeme bilden die einfachste Art von L-Systemen, da ihre Entwicklung nur anhand ihrer Abstammung gesteuert wird; das Ersetzen jedes einzelne Zeichens (siehe 3.3.1) hängt nur von

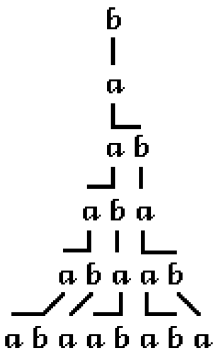


Abbildung 1: Zeitlicher Verlauf eines D0L-Systems. Bildquelle: [PL90, S.4]

der jeweiligen Regel ab. Es werden keine Informationen zwischen den Zeichen ausgetauscht, das L-System ist also kontextfrei oder null-seitig. Ein 0L-System ist ein geordnetes Triplet $G = \langle V, \omega, P \rangle$ [HP93, S.15].
Falls es für jedes Zeichen $a \in V$ genau ein (!) Wort $\chi \in V^*$ gibt, so dass $a \to \chi$ gilt, dann ist das 0L-System deterministisch und wird als D0L-System bezeichnet [HP93, S.15].

3.4.2 Kontextsensitive L-Systeme

Im Gegensatz zu den in 3.4.1 beschriebenen kontextfreien L-Systemen werden in kontextsensitiven L-Systemen zwischen den einzelnen Zeichen Informationen ausgetauscht, um die anzuwendenden Produktionsregeln zu bestimmen und die Interaktion zwischen einzelnen Pflanzen-Teilen zu simulieren. Dabei wird zwischen Ein- und Zweiseitigen Systemen 1L- und 2L unterschieden: Bei einem 2L-System werden die Produktionsregeln nur angewendet, wenn das aktuelle Zeichen a beidseitig von der richtigen Kombination aus Zeichen umgeben ist. Die dazugehörige Produktionsregel lautet: $a_l < a > a_r \to \chi$. In diesem Fall wird a als *strict predecessor* bezeichnet und kann nur ersetzt werden wenn es zwischen a_l und a_r steht. Ein 1L-System hingegen hat einen einseitigen Kontext und wird in der Form $a_l < a \to \chi$ oder $a > a_r \to \chi$ notiert, wobei hier nur jeweils das linke oder rechte Zeichen betrachtet wird[PL90, S.30].

3.4.3 Stochastische L-Systeme

Die zuvor genannten Systeme erzeugen unter den selben Bedingungen immer die selbe Pflanze. Um dies zu vermeiden und Varianz innerhalb eines Modells zu schaffen, wird das System randomisiert. Dafür werden –ausgehend von einem 0L-System (3.4.1)– für ein Zeichen a mehrere mögliche Wörter χ definiert, wobei die Summe aller Wahrscheinlichkeiten 1 ist, so dass ein *sto-*

chastisches 0L-System ein Quadrupel $G_\pi = \langle V, \omega, P, \pi \rangle$ mit der *Wahrscheinlichkeitsverteilung* $\pi : P \to [0, 1]$ bildet [PL90, S.28].

Beispiel:

$$\begin{aligned} \omega &: \text{a} \\ p_1 &: \text{a} \xrightarrow{.33} \text{a} \\ p_2 &: \text{a} \xrightarrow{.33} \text{b} \\ p_3 &: \text{a} \xrightarrow{.34} \text{ab} \end{aligned}$$

Falls es nur eine Möglichkeit für den *successor* gibt, wird dies in folgender Form ausgedrückt:

$$p : * \to$$

3.4.4 Parametrische L-Systeme

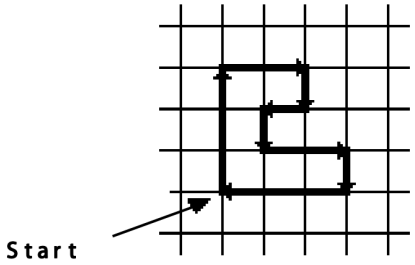
Parametrische L-Systeme erweitern die Möglichkeiten der Modellierung, vor allem bezüglich der zeitlichen Veränderung, indem numerische Parameter zu den bereits existierenden Zeichen hinzugefügt werden. Ein Zeichen $A \in V$ mit den zugehörigen Parametern $a_1, a_2, ..., a_n \in \Re$ wird $A(a_1, a_2, ..., a_n)$ geschrieben [PL90, S.41].

Beispiel:

$$\begin{aligned} \omega &: \text{b}(10) \\ p_1 &: \text{a}(x, y) \to \text{c}(x)\text{b}(y) \\ p_2 &: \text{b}(x) \to \text{a}(x * 3, x) \\ p_3 &: \text{c}(x) \to \text{b}(x) \end{aligned}$$

3.5 Interpretation von L-Systemen

Die graphische Auswertung von L-Systemen erfolgt hier mit Hilfe einer sog. Schildkröte oder *turtle*. Dafür wird jedem Zeichen des L-Systems eine entsprechende Zeichenanweisung zugewiesen. Die Schildkröte interpretiert und befolgt diese Anweisungen Schritt-für-Schritt, wobei sie sich auf einer zweidimensionalen Leinwand (dem Bildschirm) fortbewegt [PL90, S.6ff]. Für das L-System bedeutet das, dass Zeichen zweifach belegt sind: Einmal für das L-System (3.3) und einmal für die Schildkröten-Interpretation [PL90, S.46].



FFF-FF-F-F+F+FF-F-FFF

Abbildung 2: Einfache Schildkrötengrafik mit Winkel $\delta = 90°$ und Länge $d = 1$. Bildquelle: [PL90, S.7]

3.5.1 Einfache Schildkröte

Eine einfache Schildkröte oder *turtle* hat die Eigenschaften Längeneinheit d, Winkel δ und Position (x, y) und kann die folgenden Anweisungen befolgen [PL90, S.7]:

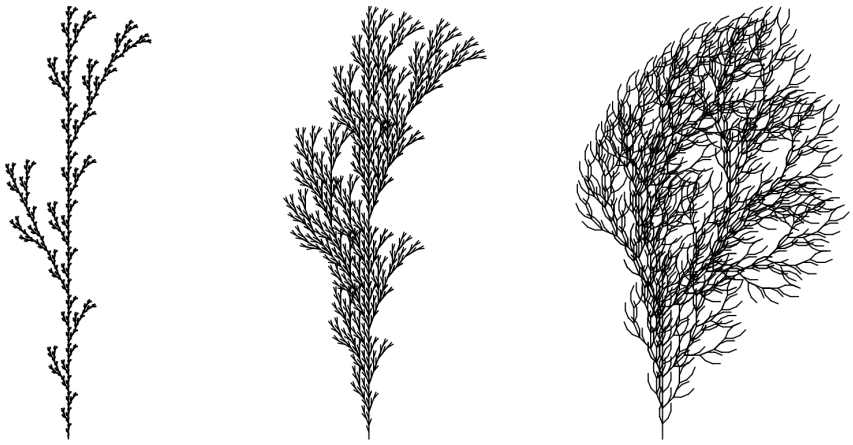
- F Bewege dich um d vorwärts und zeichne eine Linie zwischen Anfangs- und Endpunkt
- f Bewege dich vorwärts ohne zu zeichnen.
- $+$ Drehe dich um δ nach rechts.
- $-$ Drehe dich um δ nach links.

Eine einfache Schildkröten-Grafik mit einem Winkel $\delta = 90°$ und einer Länge $d = 1$ könnte zum Beispiel wie in 2 aussehen. Die Schildkröte zeigt beim Start nach oben und befolgt die vorgegebenen Zeichenanweisungen der Reihe nach. Es ist zu sehen, dass sich die Schildkröte beim Erreichen eines Zeichen $+$ oder $-$ nur neu ausrichtet, aber nicht bewegt.

3.5.2 Springende Schildkröte

Zum Modellieren von verzweigten Strukturen ist es nötig, die einfache Schildkröte (3.5.1) mit einer Sprung-Mechanik zu erweitern. Dabei wird für eine Verzweigung die aktuelle Position der Schildkröte zusammen mit ihren momentanen Informationen auf einem *Last-In-First-Out-Stack* gespeichert. Ist die Verzweigung abgeschlossen, springt die Schildkröte zur letzen gespeicherten Position zurück. Die Anweisung zum Speichern und Laden der Schildkröten-Positionen lauten [PL90, S.24]:

- $[$ Speichere die aktuellen Informationen der Schildkröte auf dem *stack*
- $]$ Lade die neuen Informationen vom *stack* in die Schildkröte



a
n=5, δ=25.7°
F
F →F[+F]F[-F]F

b
n=5, δ=20°
F
F →F[+F]F[-F][F]

c
n=4, δ=22.5°
F
F →FF-[-F+F+F]+
[+F-F-F]

Abbildung 3: Mit einem 0L-System generierte Pflanzen. *n* ist die Anzahl der Durchläufe, δ der verwendete Winkel. Bildquelle: [PL90, S.25]

Mit diesen Schildkröten-Anweisungen ist es möglich, komplexe Pflanzen darzustellen (3).

3.5.3 Dreidimensionale Schildkröte

Um fortgeschrittene Grafiken im dreidimensionalen Raum darzustellen, wird die Schildkröte mit weiteren Informationen in Form von Position (x, y, z) und Orientierung in Form von Richtungsvektoren, die die Ausrichtung der Schildkröte spezifizieren, ausgestattet. Die Vektoren geben Auskunft über die Richtung $\overrightarrow{H}$, in die die Schildkröte gerade zeigt, sowie über die Richtung zu ihrer Linken $\overrightarrow{L}$ und nach Oben $\overrightarrow{U}$. Zum Steuern müssen die Anweisungen zum Drehen der Schildkröte anhand der neuen Informationen angepasst werden (siehe 4) [PL90, S.18f]:

- $+$ Drehe dich um δ nach links.
- $-$ Drehe dich um δ nach rechts.
- $\&$ Kippe um δ nach unten.
- $\wedge$ Kippe um δ nach oben.
- $\backslash$ Rotiere um δ nach links.
- $/$ Rotiere um δ nach rechts.
- $|$ Wende um 180°

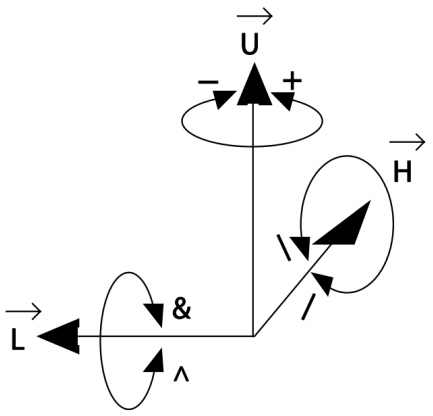


Abbildung 4: Orientierung der Schildkröte im dreidimensionalen Raum. Bildquelle: [PL90, S.19]

3.5.4 Parametrisierte Schildkröte

Die letzte (für das in der Arbeit beschriebene Programm) relevante Modifizierung der Schildkröte erweitert diese für die Benutzung von in 3.4.4 beschriebenen Parametern. Dabei werden den Zeichen-Anweisungen Parameter in Klammern nachgestellt. Hat eine Anweisung keine Parameter, wird die Anweisung wie eine nicht-parametrische Anweisung aus 3.5.3 mit Standard-Werten behandelt. Die wichtigsten parametrisierten Zeichen [PL90, S.46] lauten:

- $F(a)$ Bewege dich um a vorwärts und zeichne eine Linie zwischen Anfangs- und Endpunkt
- $f(a)$ Bewege dich um a vorwärts ohne zu zeichnen.
- $+(a)$ Drehe dich um a Grad um die Z-Achse.
- $\&(a)$ Drehe dich um a Grad um die Y-Achse.
- $/(a)$ Drehe dich um a Grad um die X-Achse.

3.5.5 Spezielle Befehle

Neben den in den vorherigen Unterabschnitten vorgestellten Zeichen-Anweisungen gibt es noch spezielle Anweisungen, die für die in 3.6 vorgestellten Systeme relevant sind [PL90, S.57]:

- $!(w)$ Setze die Breite der zu zeichnenden Linie auf w.
- $\$$ Drehe dich, bis du wieder in einer horizontalen Position bist.

Eine vollständige Liste aller verwendeten Zeichen-Anweisungen findet sich im A 1.

3.6 Modellierung von Bäumen mit L-Systemen

Mit Hilfe der in 3.5 vorgestellten Schildkröten-Interpretation ist es möglich Bäume durch L-Systemen zu erzeugen und darzustellen [PL90, S.51-61]. Dabei sind besonders die Systeme nach Honda ([Hon71], entnommen [PL90, S.55]) [PL90, S.57f] und von Lindenmayer und Prusinkiewicz [PL90, S.58, S.60f] für diese Arbeit interessant, da sie die in 1.3 gestellten Vorgaben des automatischen Erstellen von Abbildern von Bäumen in verschiedenen Stadien des Wachstums ohne Interaktion mit dem Anwender erfüllen.

3.6.1 Modell nach Honda

Das in 5 abgebildete L-System nach Honda erzeugt *monopodiale* Baum-Strukturen und bildet klare Spitzen. Die Zweige verästeln sich, wobei die einzelnen Zweige immer dünner werden. Das L-System besitzt drei Produktionsregeln. Die Regel p_1 stellt das generelle Wachstum nach oben dar. Die Zweige und ihre Entwicklung werden in den Regeln p_2 und p_3 beschrieben, die sich immer weiter in kleinere Äste aufteilen.

Eine genauere Beschreibung findet sich in [PL90, S.55ff].

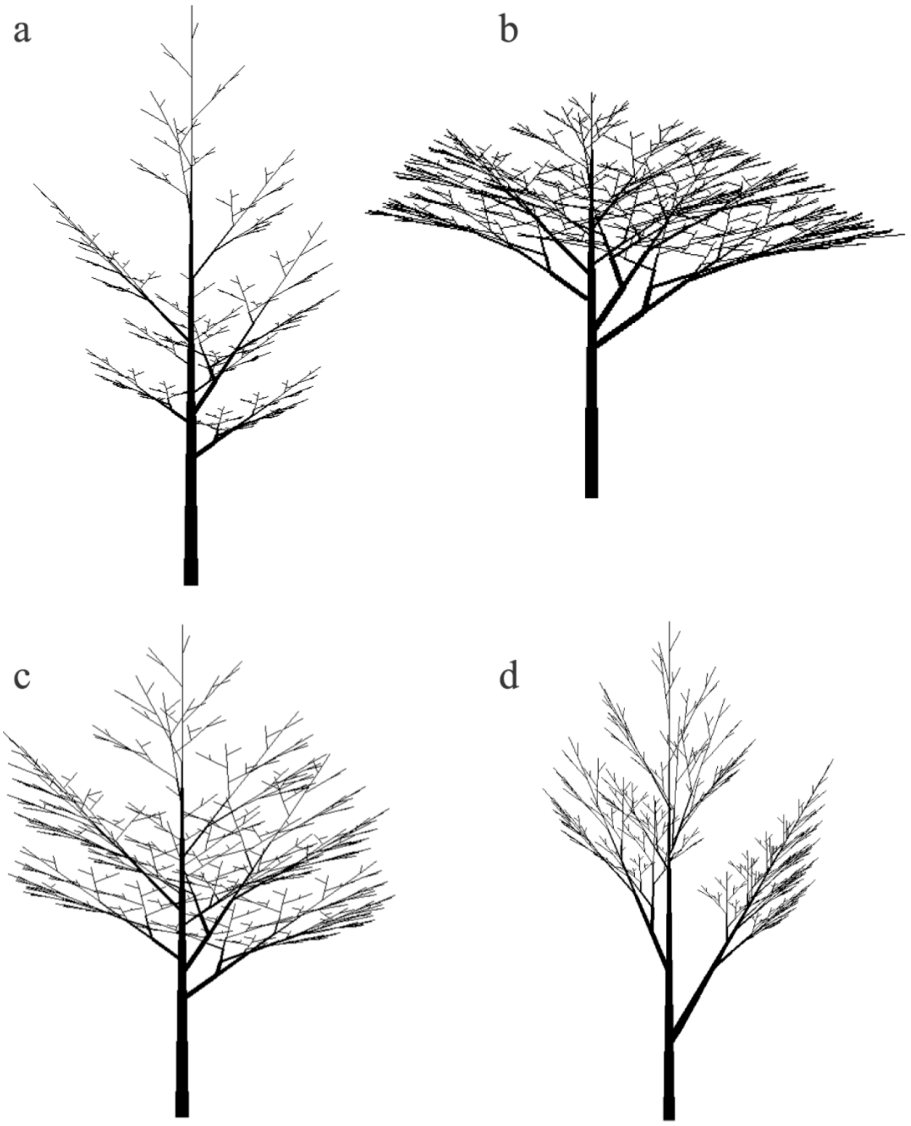
3.6.2 Modell von Lindenmayer und Prusinkiewicz

Das in 6 abgebildete L-System nach Lindemayer und Prusinkiewicz erzeugt Baum-Strukturen durch Wiederholungen, wobei durch Veränderung der Parameter verschiedene Baum-Strukturen erzeugt werden können.
Das L-System besitzt drei Produktionsregeln. Die Regel p_1 stellt das Wiederholen durch Erzeugen von neuen, gleich-großen Abschnitten dar, während die Regeln p_2 und p_3 die Länge und Dicke der vorherigen Abschnitte erhöht, was zur Folge hat, dass die ältesten Abschnitte am längsten sind.
Zum Schluss werden die erzeugten Zweige mit einem vorgegebenen *tropism vector* leicht rotiert, um das Auswirken von physischer Kraft auf den Baum zu simulieren. [PL90, S.58].

Eine genauere Beschreibung findet sich in [PL90, S.58, S.60f].

3.6.3 Wurzeln

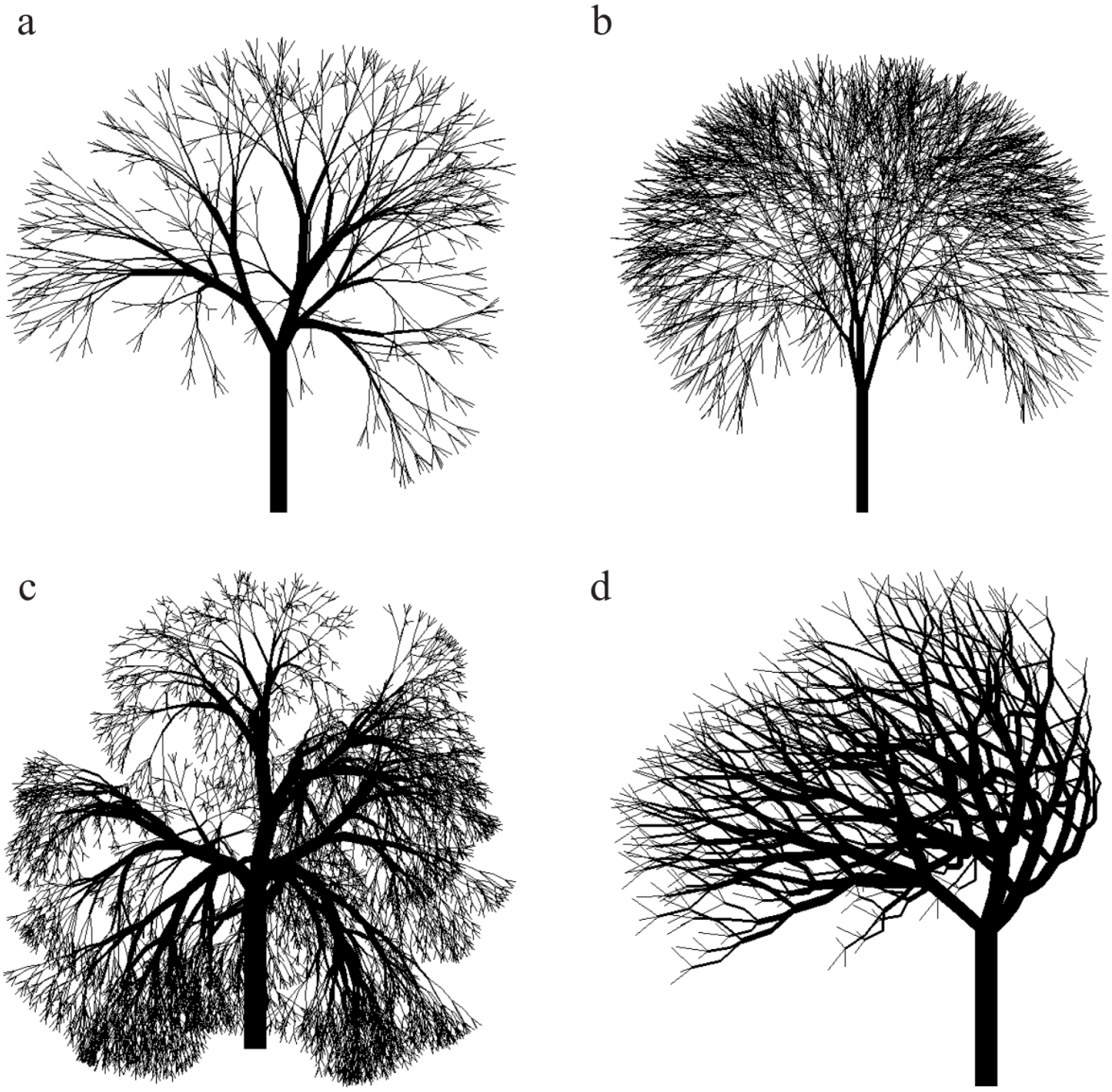
Es fällt auf, dass die erzeugten Bäume keine Wurzeln besitzen, da die in diesem Abschnitt beschriebenen L-Systeme nur die Verzweigung und Struktur der Bäume über dem Boden




```
n = 10
#define r1 0.9     /* contraction ratio for the trunk */
#define r2 0.6     /* contraction ratio for branches */
#define a0 45      /* branching angle from the trunk */
#define a2 45      /* branching angle for lateral axes */
#define d 137.5    /* divergence angle */
#define wr 0.707   /* width decrease rate */

ω : A(1,10)
p1: A(l,w) : * → !(w)F(l)[&(a0)B(l*r2,w*wr)]/(d)A(l*r1,w*wr)
p2: B(l,w) : * → !(w)F(l)[-(a2)$C(l*r2,w*wr)]C(l*r1,w*wr)
p3: C(l,w) : * → !(w)F(l)[+(a2)$B(l*r2,w*wr)]B(l*r1,w*wr)
```

Abbildung 5: Baum-Strukturen nach Honda mit Beschreibung L-System. Bildquelle: [PL90, S.56]




```
#define d1 94.74          /* divergence angle 1 */
#define d2 132.63         /* divergence angle 2 */
#define a 18.95           /* branching angle */
#define lr 1.109          /* elongation rate */
#define vr 1.732          /* width increase rate */

ω  :  !(1)F(200)/(45)A
p1 :  A : *     → !(vr)F(50)[&(a)F(50)A]/(d1)
                  [&(a)F(50)A]/(d2)[&(a)F(50)A]
p2 :  F(l) : *  → F(l*lr)
p3 :  !(w) : *  → !(w*vr)
```

Abbildung 6: Baum-Strukturen nach Lindemayer und Prusinkiewicz mit Beschreibung L-System. Bildquelle: [PL90, S.60]

modellieren. Eine Erweiterung des Systems zum Modellieren von Wurzeln ist aber möglich [PJM94, S.8] [MP96, S.9].

3.7 Weitere Möglichkeiten der Modellierung

Neben dem in diesem Kapitel beschriebenen regelbasierten Vorgehen gibt es noch weitere Möglichkeiten der Modellierung von Bäumen, die aber für das gedachte Anwendungsgebiet zu komplex und rechenintensiv (Anwendung auf einem mobilen Endgerät) oder zu speziell sind. Im allgemeinen ist das Modellieren von Bäumen sehr anspruchsvoll, da viele verschiedene Aspekte von Bäumen, wie zum Beispiel Wachstumsbedingungen (Sonneneinstrahlung und Boden), mechanische Belastungen (Wind und Wetter), chemische Vorgänge (Photosynthese) und das Verhältnis von Bäumen zu ihrer Umgebung betrachtet werden müssen[PJM94] [MP96, S.2] [BL10, S. 1342].
Generell kann die Modellierung von Bäumen durch Computergrafiken in drei Methoden unterteilt werden, die alle das Ziel haben, eine plausible Baum-Geometrie zu erzeugen [Che+08, S.1f] [Est+15, S.3]:

Regel-basiert	Der Baum wird durch das systematische Anwenden von Regeln erzeugt, zum Beispiel durch L-Systeme. Ein häufiges Problem der regel-basierten Generierung sind die für den Anwender unintuitiven und undurchsichtigen Regeln, deren korrektes Anwenden häufig biologischen Vorwissens bedarf.
Sketch-basiert	Der Baum wird vom Benutzer kurz skizziert, anschließend wird der Baum anhand dieser Skizze generiert. Im Gegensatz zu den Regel-basierten sind diese System intuitiver, benötigen aber auch mehr Input.
Bild-basiert	Der Baum wird aus vorgegebenen Beispiel-Bildern synthetisiert. Diese Systeme sind am komplexesten und haben die höchsten Anforderungen, da sie auch die vorgegebenen Bilder analysieren und aus ihnen ein Modell erstellen müssen.

4 Programm

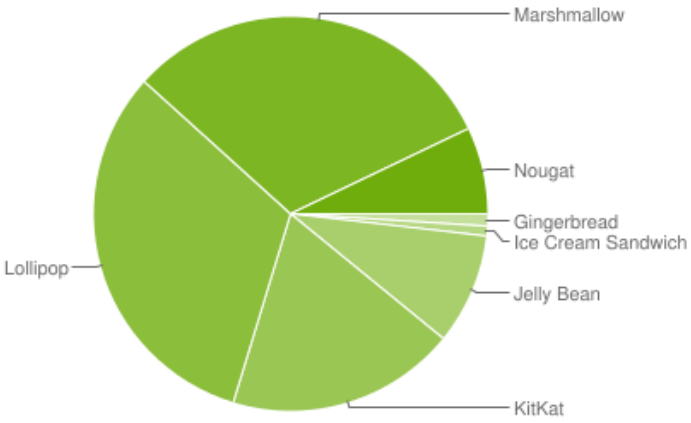
In diesem Kapitel wird das Anwenden der Prozeduralen Generierung zum Erstellen einer Fortschrittsanzeige in Form wachsender Abbilder realer Bäume in Android beschrieben.

4.1 Anforderungen

Vor dem Beginn der Programmierung werden zuerst die Anforderungen des zu erstellenden Programmes festgehalten, die sich aus der in 1.2 und 1.3 beschriebenen Zielsetzung ergeben:

Das Programm

- Muss in Android API 16+ (Jellybean) funktionieren.
- Soll möglichst nur mit Android-internen Mitteln erzeugt werden (kein OpenGL, keine Bibliotheken) um die Applikation problemlos in die bestehende App (Beschrieben in 1.1) einbinden zu können.
- Muss reale Bäume abbilden/darstellen können.
- Muss Bäume von Grund auf wachsen lassen (Jung $\rightarrow$ Alt, Stamm $\rightarrow$ Krone); Bäume können nicht einfach nur erscheinen.
- Muss die Bäume überall auf dem Bildschirm wachsen lassen können (entlang der X-Achse).
- Muss Bäume in 2D darstellen.
- Soll das Erzeugen der Bäume animieren.
- Soll Bäume modulartig erzeugen, so dass „Teile“ der Bäume austauschbar sind und verschiedene Baumarten erzeugt werden können.
- Soll wiederverwendbar sein (Erzeugen des Baumes in verschiedenen Aktivitäten innerhalb der App).
- Soll automatisch Bäume erzeugen; der Benutzer soll nur die grundlegendsten Steuerungsmöglichkeiten haben (auswählen verschiedener Baumarten).



Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	1.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.8%
4.1.x	Jelly Bean	16	3.2%
4.2.x		17	4.6%
4.3		18	1.3%
4.4	KitKat	19	18.8%
5.0	Lollipop	21	8.7%
5.1		22	23.3%
6.0	Marshmallow	23	31.2%
7.0	Nougat	24	6.6%
7.1		25	0.5%

Abbildung 7: Verteilung der Android-Versionen. Stand 2. Mai 2017. Bildquelle: [And17f]

4.2 Technische Spezifikationen

In diesem Abschnitt werden die technischen Spezifikationen der Entwicklung festgehalten, um ein erfolgreiches Testen und Nachvollziehen der vorgestellten Anwendung zu ermöglichen.

Die benutzte Entwicklungsumgebung ist AndroidStudio 2.3.2 [And17g], die verwendete Programmiersprache Java.

Das Programm wurde auf mehreren Geräten mit unterschiedlichen Betriebssystemen und Bildschirm-Größen getestet, die sich aus der aktuellen (Stand 1.6.17) Verteilung der Android-Geräte ergeben haben [And17f]. Die Android-Versionen 4, 5, 6 und 7 decken zusammen 98.2% aller Android-Versionen ab, wobei mehr als die Hälfte aller Geräte auf den Versionen 5 (32%) und 6 (31.2%) läuft. Android Version 4 (27.9%) bildet das untere Ende der unterstützen Geräte, und Version (7.1%) das obere Ende (7).
Der größte Teil der Android-Geräte (89.3%) verfügt über einen Bildschirm regulärer Größe (kein Tablet, keine Smartwatch). Von diesen haben die meisten (87%) eine DPI-Verteilung (Dots per Inch) von hdpi ($\tilde{2}$40dpi) oder größer (xhdpi $\tilde{3}$20dp, xxhdpi $\tilde{4}$80dpi) [And17i] (8).
Die verwendeten Test-Geräte (**??**) entsprechen dem Zufolge der Verteilung der Android-Version und Bildschirm-Konfigurationen.

**Das Gerät ist ein AVD (Android Virtual Device): ein virtuelles, auf dem Computer simuliertes Gerät*

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	1.0%						1.0%
Normal		2.1%	0.2%	34.6%	34.8%	17.6%	89.3%
Large	0.1%	3.5%	1.8%	0.4%	0.4%		6.2%
Xlarge		2.4%		0.5%	0.6%		3.5%
Total	1.1%	8.0%	2.0%	35.5%	35.8%	17.6%	

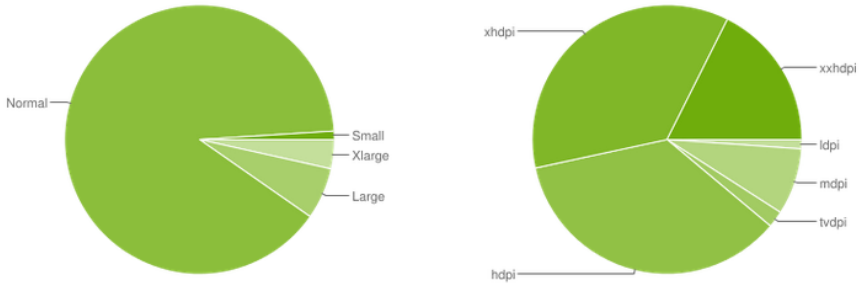


Abbildung 8: Verteilung der Bildschirm-Größen. Stand 2. Mai 2017. Bildquelle: [And17f]

Geräte-Name	Android-Version	Bildschirmdiagonale	Bildschirmdaten
HTC One (m8)	Android 6.0 Marshmallow (API 23)	5.0“	1080x1920px *hdpi*
HTC One (m8)	Android 7.1 Nougat (API 25)	5.0“	1080x1920px *hdpi*
Nexus 2*	Android 5.1 Lollipop (API 22)	4.7“	768x1280px *xdpi*
HVGA Slider (ADP1)*	Android 4.1 Jellybean (API 16)	3.2“	320x480px *mdpi*

Tabelle 1: Verwendete Test-Geräte.

Grundsätzlich gilt beim Programmieren für Android, dass die erstellten Anwendungen auf verschiedenen mobilen Endgeräten mit unterschiedlichen Betriebssystem-Versionen und Bildschirmkonfigurationen (also auf Geräten die sich in Bildschirm-Auflösung, physikalischer (Bildschirm-)Größe und Bildschirm-Pixel-Dichte unterscheiden) angezeigt und betrachtet werden. Dabei kann es sein, dass sich zwischen verschiedenen Geräten Unterschiede bei der Anzeige auftun.

Das Programm ist optimiert für Geräte mit einer Auflösung von 1920x1080px.

4.3 Zeichnen in Android

Es gibt verschiedene Möglichkeiten mit Android Grafiken zu erstellen. Für das Programm sind die folgenden Methoden auf Grund ihrer hohen Steuerbarkeit von Interesse:

4.3.1 Surface-View und Canvas

Um in Android eigene Grafiken zu erstellen, müssen diese direkt auf ein Canvas (eine Zeichenoberfläche, vergleichbar mit einer Leinwand) gezeichnet werden [And17b]. Dabei wird zwischen dem direkten Zeichen auf dem Hauptthread in der *view*, dem Grundbaustein des *User Interface* [And], und dem Zeichnen auf einer speziellen, in einen sekundären Thread ausgelagerten, *surfaceview* unterschieden [And17a]. Da das direkte Zeichen in der *view* hauptsächlich für langsame Zeichenprozesse ohne hohe Leistungsansprüche ist, die geplanten Spiele aber komplex sind und zum Teil hohe Leistungsanforderungen haben, wird stattdessen das Zeichnen in der *surfaceview* verwendet.
Ein alternativer Ansatz wäre, die Baum-Grafiken als Bitmaps zu erstellen und dann in der *view* anzuzeigen [Staa]. Diese Alternative wurde aber nicht weiter verfolgt, da das Zeichnen auf dem Canvas die von Android empfohlene Methode zum Erstellen von Grafiken ist.
Ein Problem beim Zeichnen auf dem Canvas ist, dass das Canvas mit der *surfaceview* geteilt wird und für jeden Zeichenvorgang komplett (!) neu befüllt werden muss:

„The content of the Surface is never preserved between unlockCanvas() and lockCanvas(), for this reason, every pixel within the Surface area must be written.[And17d]“

Es ist also nicht möglich das Bild (den Baum) schrittweise zu zeichnen. Stattdessen wird immer eine komplette Beschreibung des Baumes gespeichert und auf das Canvas angewendet.

Auch passt sich die *surfaceview* nicht automatisch der Bildschirmperspektive an. Die Anwendung muss also bei jedem Programm-Start die aktuelle Höhe und Breite der *view* ermitteln und sich daran anpassen.

4.3.2 OpenGL

Eine mögliche Alternative zu den Android-eigenen Zeichenmethoden ist OpenGL ES, eine „cross-platform graphics API“ die für hochperformante Grafiken ausgelegt ist [And16].
Android unterstützt OpenGL, aber nicht auf allen Geräten gleich, was unter anderem an unterschiedlichen Implementationen der OpenGL Grafik-Pipeline verschiedener Geräte-Hersteller

liegt [And16].
OpenGL ES Version 1.1 und 2.0 wird laut Google auf allen Geräten unterstützt, Version 3.0 auf 61.7% und Version 3.1 nur auf 18.2% aller Geräte [And17f]. Dabei unterscheiden sich die Versionen in Leistung, Programmieraufwand, Kontrolle und Unterstützung von Texturen [And16].
Eine Erweiterung der Anwendung zur Benutzung von OpenGL ist also möglich, aber nicht gefordert (siehe Anforderungen).

4.4 Aufbau des Programms

Das Programm besteht aus einer Zeichenumgebung und einem Baum-Objekt.

Das Baum-Objekt enthält alle für das L-System relevanten Informationen wie Beschreibung, Startwort und Produktionsregeln. Weiterhin speichert das Baum-Objekt eine Chronik aller vorherigen Beschreibungen und die aktuelle Generation. Das Baum-Objekt verwendet ein L-System zum Entwickeln seiner aktuellen Beschreibung (siehe 3).
Die Zeichenumgebung befindet sich in der Hauptaktivität des Programmes und kann die aktuelle Beschreibung des Baum-Objektes interpretieren und in eine geometrische Beschreibung in Form von Linien umwandeln, die dann von einer Schildkröte auf ein Canvas abgebildet und in der *surfaceview* dem Benutzer angezeigt wird.
Weiterhin enthält das Hauptprogramm (die Zeichenumgebung) neben der Hauptansicht zwei ausklappbare Seitenmenüs, über die verschiedenen Debug-Einstellung (z.B. zur Kontrolle des Zeichen-Mechanismus und der Animation) sowie vordefinierte Baum-Objekte zum Zeichnen gewählt werden können.

4.5 Beschreibung der Bäume

Ein Baum-Objekt wird im Programm als neues Baum-Objekt erstellt und dann mit für das L-System relevanten Inhalten gefüllt. Dabei verwendet das Programm das in 3.6.1 beschriebene System nach Honda.

4.5.1 StringObject

Wie in 3.2 beschrieben, basieren L-Systeme häufig auf *strings*. In der Anwendung wird an Stelle von *strings* ein *string*-basiertes Objekt, *StringObject* genannt, verwendet, da die Interpretation reiner Strings in Verbindung mit Parametern zu Performance-Problemen führt:
Eine reine *string*-Anweisung $F()$ besteht aus mindestens drei Zeichen ('F', '(' und ')'). Bei jeder Verwendung des Zeichen 'F' (zum Entwickeln der Beschreibung oder für die graphische Interpreation) müssen dafür in einer Subroutine die aktuellen Parameter abgefragt werden. Weiterhin müssen die ausgelesenen Parameter in jeweils weiteren Subroutinen in Zahlen des passenden Types (*int*, *float* oder *double*) umgewandelt werden. Dies ist aufwändig, langsam und fehleranfällig.
Ein *StringObject* hingegen besteht aus einem *symbol* genannten *string* für das Zeichen, so wie einem oder mehreren dazugehörigen Parametern die gleich im richtigen Typ abgespeichert sind. Wird das Zeichen verwendet, können das *symbol* und die zugehörigen Parameter direkt ohne Umwandlung separat abgerufen werden. Die zuvor erwähnten Subroutinen fallen damit weg.

Um die Handhabung mehrerer *StringObjects* zu verbessern, werden *StringObjects* im Programm immer in Listen zusammengefasst.

Beispiel: Das in 5 (Baumstrukturen nach Honda) beschriebene *axiom*

$$\omega \quad : \quad A(1, 10)$$

wird im Programm als *StringObject* mit Parametern (1, 10) definiert, zu einer Liste hinzugefügt und dann als Start-Beschreibung des Baum-Objekts gesetzt (1).

```
ArrayList<StringObject> description = new ArrayList<StringObject>();
description.add(new StringObject("A", new Parameter(1), new Parameter(10)))
    ;

// create the tree with the start description
Tree tree = new Tree(description);
```

Code 1: Definition des Baum-Objekts mit Startwort

4.5.2 Rule

Neben der Startbeschreibung verfügt das Baum-Objekt auch noch über Produktionsregeln. Diese werden ebenfalls in Form einer Liste aus *StringObjects* ausgedrückt.

Beispiel: Die Produktionsregel p_1 aus 5 lautet

$$A(l,w) \quad :* \rightarrow \quad !(w)F(l)[\&(a_0)B(l*r_2, w*w_r)]/(d)A(l*r_1, w*w_r).$$

Im Programm wird diese Regel als Liste mit *StringObjects* wie in 2 gezeigt initialisiert und dem Baum hinzugefügt.

```
ArrayList<StringObject> ruleA = new ArrayList<>();
ruleA.add(new StringObject("!"));
ruleA.add(new StringObject("F"));
ruleA.add(new StringObject("["));
ruleA.add(new StringObject("&", new Parameter(a0)));
ruleA.add(new StringObject("B", new Parameter(0,r2), new Parameter(0,wr)));
ruleA.add(new StringObject("]"));
ruleA.add(new StringObject("/", new Parameter(d)));
ruleA.add(new StringObject("A", new Parameter(0,r1), new Parameter(0,wr)));

tree.addRule("A", ruleA);
```

Code 2: Definition einer Produktionsregel

Dabei fällt auf, dass die Parameter der *StringObjects* zum Teil mit mehreren Werten erstellt werden. Neben dem eigentlichen Parameter-Wert erhält der Parameter auch einen Multiplikator. Dieser ist standardmäßig 1, bewirkt also keine Veränderung. In einigen Fällen der Ersetzung (zum Beispiel bei $B(l*r_2, w*w_r)$ in der Produktionsregel) muss das Zeichen aber multipliziert werden (Beschrieben in Evolution der Bäume).

4.6 Evolution der Bäume

Um die Beschreibung des Baum-Objekts zu entwickeln wird diese gemäß der Produktionsregeln (wie in 3.3 beschrieben) schrittweise ersetzt. Dafür wird durch alle Zeichen der Beschreibung iteriert, wobei jedes Zeichen auf eine Übereinstimmung mit den in den Regeln festgehaltenen *predecessors* geprüft wird (3).

```
boolean match = false;
// check if any replacement rule matches
for (Rule rule : rules){
        // symbol matches rule predecessor
        if (stringObject.getSymbol().equals(rule.getPredecessor())){
```

```
                // replace the symbol with its successor and return the new
                    word
                ArrayList<StringObject> replacement = replace(stringObject,
                    rule.getSuccessor());

                // add the returned word to the interims_description
                for (StringObject replacementObject : replacement)
                        interims_description.add(replacementObject);

                match = true;
                break;  // exit the rule-match loop
        }
}
// no rule match
if (!match){
        // just append the current StringObject
        interims_description.add(stringObject);
```

Code 3: Vergleich der Regeln mit der Beschreibung

Stimmen Zeichen und *predecessor* überein, wird das Zeichen mit dem *successor* ersetzt. Dafür wird das Wort (die Beschreibung) des *succesor* Zeichen für Zeichen in eine neue Liste gefüllt, wobei die Parameter jedes neuen Zeichens entsprechend seiner Regel-definierten Besonderheiten entwickelt wird (4).

Das Zeichen B wird z.B. an Hand der Parameter des *predecessors* multipliziert (5 und 5).

```
private ArrayList<StringObject> replace(StringObject predecessor, ArrayList
   <StringObject> successor_word){
        for (StringObject successor : successor_word){
                switch (successor.getSymbol()) {

                        case "B":
                                new_word.add( new StringObject("B",
                                   predecessor.getParameters()[0].
                                   getMultipliedParam(), predecessor.
                                   getParameters()[1].getMultipliedParam())
                                   );
                                break;

                        //more cases

                        default:
```

```
                                    new_word.add(successor);
                                    break;
```

Code 4: Ersetzen des Zeichens

```
private float
            param,
            mult = 1;   // default multiplikator is 1

Parameter getMultipliedParam(){
        return new Parameter(param * mult, mult);
}
```

Code 5: Multiplikation eines Parameters

Zum Schluss wird das ersetzte Wort als neue Beschreibung des Baum-Objects gesetzt.

4.7 Schildkröten-Interpretation der Bäume

Um die Beschreibung des Baum-Objekts in eine für die Zeichenumgebung verwertbare Form zu bringen, muss sie gemäß 3.5 interpretiert werden. Dafür wird die Schildkröte anhand der aktuelle Beschreibung des Baum-Objekts bewegt, so dass am Ende eine Linien-Repräsentation des Baum-Objekts entsteht. Vor dem Erzeugen einer neuen Interpretation wird die Schildkröte wider auf ihre Startposition zurückgesetzt. (6).

```
List<Line> interpretTreeDescription(ArrayList<StringObject> desc){
        clear(); // reset data
        for (StringObject symbol : desc){
                switch (symbol.getSymbol()){

                        case "F":
                                forward();
                                break;

                        // more cases

                }
        }
        return lines;
}
```

Code 6: Interpretation der Baum-Beschreibung.

Im Folgenden werden die Eigenschaften der Schildkröte und die Interpretation der einzelnen Zeichen als Schildkröten-Anweisung beschrieben.

4.7.1 Startwerte der Schildkröte

Der Startpunkt der Schildkröte und die Länge der Linien werden beim Start des Programms dynamisch basierend auf der Größe des Bildschirms festgelegt. Grundsätzlich befindet sich der Startpunkt aber immer in der Mitte des Bildschirms am unteren Rand, wie in den Anforderungen festgelegt.

4.7.2 Zeichnen (F)

Das Zeichen F bewegt die Schildkröte um die Länge des aktuellen Parameters in Richtung ihrer Richtungswinkel. Dabei wird eine Linie zwischen dem Start- und dem Endpunkt gezeichnet.

Eine genaue Beschreibung findet sich in Zeichnen der Bäume.

4.7.3 Veränderung der Richtungswinkel ($+ - \& /$)

Die Orientierung der Schildkröte im dreidimensionalen Raum (x,y,z) wird durch drei Richtungswinkel bestimmt. Diese werden im Programm *rotationTurn*, *rotationRoll* und *rotationPitch* genannt.
$+$ und $-$ drehen, $\&$ nickt und $/$ rollt die Schildkröte an Hand des Parameters des aktuellen Symbols: Ist der Parameter positiv, wird sein Wert zu dem Wert des betreffenden Winkels addiert, andernfalls abgezogen.

```
// + :  Turn left if parameter is positive and right if parameter is
   negative.
case "+":
        setRotationTurn(symbol.getParameter());
        break;

private void setRotationTurn(float r){
        rotationTurn = rotationTurn + r;
}
```

Code 7: Ändern eines Richtungswinkels

4.7.4 Speichern und Laden der Position [ ]

Die Position der Schildkröte wird mit [gespeichert und mit] geladen.

Zum Speichern wird die aktuelle Position der Schildkröte, ihre Orientierung und ihre Informationen (Dicke der Linie) auf einem *stack* gespeichert (8).

```
private void savePoint(){
        // get current position
        Point3 current = HEAD;
        // save turtle position, orientation and data
        savePoints.add(current);
        saveRotationTurn.add(rotationTurn);
        saveRotationPitch.add(rotationPitch);
        saveRotationRoll.add(rotationRoll);
        saveBrushWidth.add(strokeW);
}
```

Code 8: Speichern der Position

Zum Laden werden die zuletzt gespeicherten Informationen der Schildkröte wieder vom *stack* in die Schildkröte gelesen, wobei der letzte Eintrag des *stacks* entfernt wird. (9).

```
private void loadPoint(){
        // get last saved point
        Point3 loadedPoint = savePoints.get(savePoints.size()-1);
        // remove the last saved point (pop)
        savePoints.remove(savePoints.size()-1);
        // set loaded coordinates as new HEAD
        HEAD = loadedPoint;

        rotationTurn = saveRotationTurn.get(saveRotationTurn.size()-1);
        saveRotationTurn.remove(saveRotationTurn.size()-1);
        // .. load other informations the same way..
}
```

Code 9: Laden der Position

4.7.5 Ändern der Liniendicke !

Das Symbol ! setzt die Dicke der zu zeichnenden Linie auf den aktuellen Wert des Symbol-Parameters.

4.8 Zeichnen der Bäume

Bäume werden in Form von Linien auf das Canvas gezeichnet.
Der Startpunkt der ersten Linie beginnt in der Mitte des Bildschirms am unteren Rand und wird ROOT genannt. Seine Position wird automatisch an Hand der Bildschirmgröße generiert und ändert sich während aller Durchläufe des L-Systems nicht.
Linien haben einen Start- und einen Endpunkt. Beim Erzeugen einer neuen Linie (Schildkröten-Interpretation des Zeichen F) wird ihr Startpunkt auf den Endpunkt der vorherigen Linie gelegt, während ein temporärer Endpunkt auf der Y-Achse direkt über dem Startpunkt der neuen Linie erzeugt wird, so dass die Länge der Linie dem Wert des aktuellen Parameters entspricht.
Anschließend wird der temporäre Endpunkt entsprechend der Orientierung der Schildkröte transformiert, bis alle drei Winkel abgehandelt und ein finaler Endpunkt ermittelt ist (Beschreibung der Transformation in 4.8.1 und 4.8.2). Zwischen diesem und dem Startpunkt der Schildkröte wird dann eine Linie gezeichnet, deren Endpunkt als neue Position der Schildkröte gesetzt wird. Zum Schluss wird die neue Linie der aktuellen geometrischen Beschreibung hinzugefügt (10).

```
private void forward(float len){
        PointF start = HEAD;
        // line moves from bottom to top, root is at Ymax, therefore lines
           move from Ymax to Ymin (0)
        PointF end_untransformed = new PointF(start.x, start.y - (int)len);
        // turn
        PointF end_turned = end_untransformed;
            end_turned = pointTransformations.turn(start, end_untransformed
               , rotationTurn);
                // pitch and roll...

        // point HEAD to new end point
        HEAD = end_rolled;
        Line line = new Line(start, end_rolled, p, strokeW * strokeWMod,
           branch_generation);
        lines.add(line);
}
```

Code 10: Zeichenanweisung der Schildkröte.

In der Zeichenumgebung wird die geometrische Beschreibung dann auf das Canvas gebracht. Dafür muss das Canvas von der *surfaceview* angefordert und wie in 4.3.1 spezifiziert von allen fehlerhaften Pixeln gesäubert werden. Anschließend werden die Linien der Beschreibung

nacheinander in einer Schleife auf das Canvas gezeichnet (11).

```
private void draw(List<Line> lineList){
        // get the canvas from the surfaceview
        Canvas canvas = surfaceView.getHolder().lockCanvas();
        // clear the canvas white (to remove left-over pixels)
        canvas.drawRGB(255,255,255);

        int line_number = 0;
        for (Line line : lineList){
                canvas.drawLine(
                                line.getStart().x,
                                line.getStart().y,
                                line.getEnd().x,
                                line.getEnd().y,
                                line.getPaint());
        }
        // post the canvas to the surfaceview
        surfaceView.getHolder().unlockCanvasAndPost(canvas);
}
```

Code 11: Zeichnen der Linien auf dem Canvas der *surfaceview*.

Das Anwenden der Transformationen ist in den folgenden Unterabschnitten beschrieben. Dabei wird zwischen dem Erstellen der Linien in 2D und 3D unterschieden.

4.8.1 Zeichnen in 2D

Standardmäßig verfügt das Android-Canvas über zwei Dimensionen [And17b] [And17c], es ist also nur das Rotieren der Linie um die Z-Achse möglich 12.
Für das richtige Darstellen des L-Systems sind aber auch das Rotieren um die Y- und das Nicken um die X-Achse erforderlich (siehe Dreidimensionale Schildkröte).

```
PointF turn(PointF angleP, PointF p, float degrees) {
        Matrix transform = new Matrix();
        // This rotates around the previous Point by degrees
        transform.setRotate(degrees, angleP.x, angleP.y);

        // Create new float[] to hold the rotated coordinates
        float[] pts = new float[2];
        // Initialize the array with our Coordinate
```

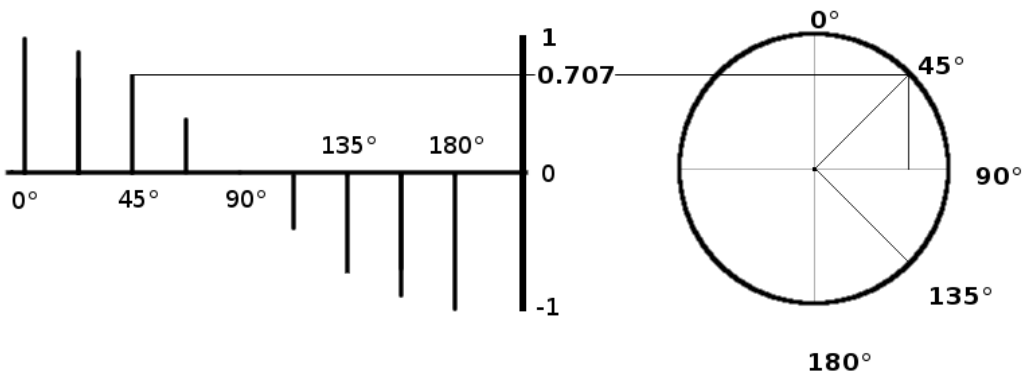


Abbildung 9: Bestimmung des Skalierungsfaktors aus dem Winkel

```
        pts[0] = p.x;
        pts[1] = p.y;
        // Use the Matrix to map the points
        transform.mapPoints(pts);
        // Now, create a new Point from our new coordinates
        PointF newPoint = new PointF(pts[0], pts[1]);
        return newPoint;
}
```

Code 12: Rotieren der Linie um die Z-Achse mit Hilfe einer Rotations-Matrix in Android

Um die erforderten Transformationen erzeugen zu können muss die Linie anhand des entsprechenden Winkels skaliert werden. Dafür wird ein Skalaierungsfaktor aus dem Sinus (zum Rollen) oder dem Kosinus des Winkels (zum Nicken) errechnet (9), der das Drehen um die X- oder Y-Achse simuliert (13).
Der Gedanke hinter Simulation ist, dass der Betrachter eine Rotation um die X- oder Y-Achse als Skalierung wahrnimmt. Schaut der Betrachter frontal auf die Linie, sieht er sie nur bei einer Rotation von 0° in voller Größe. Wird sie z.B. um 45° um die X-Achse rotiert, schrumpft die Linie auf das 0.707 Fache ihrer originalen Größe (10).
Bei einem Winkel von 180° erscheint die Linie umgekehrt, sie wird also in Richtung -1 gezeichnet.

```
PointF turnRoll(PointF angleP, PointF p, float degrees) {
        float scale = (float) Math.sin(degrees * Math.PI / 180); //roll
        //float scale = (float) Math.cos(degrees * Math.PI / 180); //pitch
```

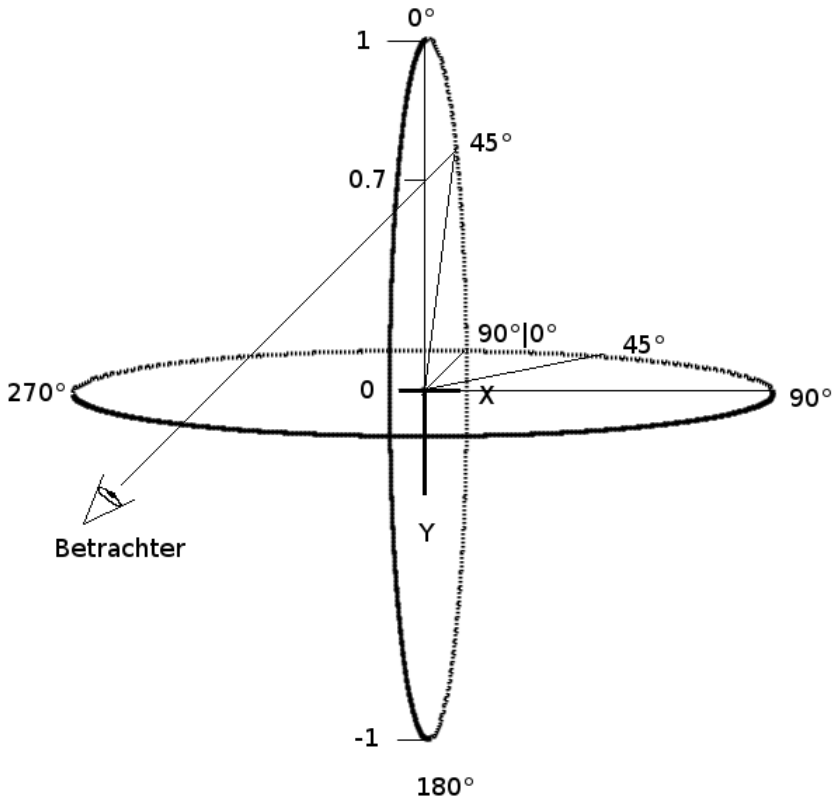


Abbildung 10: Transformation der Linie durch Skalieren

```
        Matrix transform = new Matrix();
        // This rotates around the previous Point by degrees
        transform.setScale(scale, scale, angleP.x, angleP.y);

        // Create new float[] to hold the rotated coordinates
        float[] pts = new float[2];
        // Initialize the array with our Coordinate
        pts[0] = p.x;
        pts[1] = p.y;

        // Use the Matrix to map the points
        transform.mapPoints(pts);

        // Now, create a new Point from our new coordinates
        PointF newPoint = new PointF(pts[0], p.y);      //roll
        // PointF newPoint = new PointF(p.x, pts[1]);//pitch
        return newPoint;
}
```

Code 13: Skalieren der Linie um die X und Y-Achse mit Hilfe einer Skalierungs-Matrix in Android

Der Code für die gezeigten und benutzen Matrix-Operationen in Android basiert auf [Stab].

4.8.2 Zeichnen in 3D

Um das Transformieren zu erleichtern, werden die Punkte der Linien als eigene 3D-Punkte definiert. Diese 3D-Punkte können dann regulär wie in [PL90, S.19] beschrieben mit Rotationsmatrizen um die Z (16), Y(15) und X-Achse (14) rotiert werden.

```
double newX = Math.cos(degree) * x - Math.sin(degree) * y;
double newY = Math.sin(degree) * x + Math.cos(degree) * y;

this.x = newX;
this.y = newY;
```

Code 14: Rotieren der Linie um die Z-Achse mit Hilfe einer Rotations-Matrix in 3D

```
double newX = Math.cos(degree) * x + Math.sin(degree) * z;
double newZ = -Math.sin(degree) * x + Math.cos(degree)* z;

this.x = newX;
this.z = newZ;
```

Code 15: Rotieren der Linie um die Y-Achse mit Hilfe einer Rotations-Matrix in 3D

```
double newY = Math.cos(degree) * y - Math.sin(degree) * z;
double newZ = Math.sin(degree) * y + Math.cos(degree) * z;

this.y = newY;
this.z = newZ;
```

Code 16: Rotieren der Linie um die X-Achse mit Hilfe einer Rotations-Matrix in 3D

Zum Anzeigen der Punkte auf dem zweidimensionalen Canvas werden sie dann in Form einer Parallel-Perspektive in 2D umgewandelt indem die Z-Koordinate entfernt wird.

4.8.3 Blätter

Zum Zeichnen der Blätter wird am Ende jeder Linie ein grüner Punkt gezeichnet.

4.8.4 Performance

Die Ausführungsgeschwindigkeit beim Erzeugen höherer Generationen (Generation neun oder höher) ist zum Teil sehr langsam. Eine Verbesserung der Baum-Entwicklung durch Benutzen von *threads* ist geplant.

4.9 Wachstum und Animation

Die Benutzung von L-Systemen macht das in den Anforderungen geforderte Wachsen und Animieren der Bäume sehr einfach: Es müssen dem Benutzer nur die erzeugten geometrischen Beschreibungen in chronologischer Reihenfolge angezeigt werden.

4.9.1 Wachstum

Um das Wachstum weiter zu verdeutlichen werden die Bäume mit einer Wachstumskomponente ausgestattet, die die Länge der zu zeichnenden Linien an Hand der aktuellen Generation des Baums festlegt. Dabei wird die Länge der Linien der geometrischen Beschreibung des Baumes in jeder Generation erhöht, so dass ein Baum der zehnten Generation die zehnfache Linienlänge eines Baums der ersten Generation besitzt. Ein Beispiel dafür ist in 11 zu sehen.

4.9.2 Animation

Das Wachstum der Bäume wird im Programm nur durch das zuvor genannte chronologische Aneinanderreihen der erzeugten Abbilder animiert, was zu einem sprunghaften Wachstum führt, da der Zustand des Systems nur in festgelegten Interwallen bekannt ist [PHM93, S.1] [DL01, S.60]. Eine Verbesserung der Animation ist aber möglich, z.B. durch das Mischen von relgebasierten und algorithmischen Elementen zum Erzeugen eines „Kurzfilms“ [DL01, S.62].

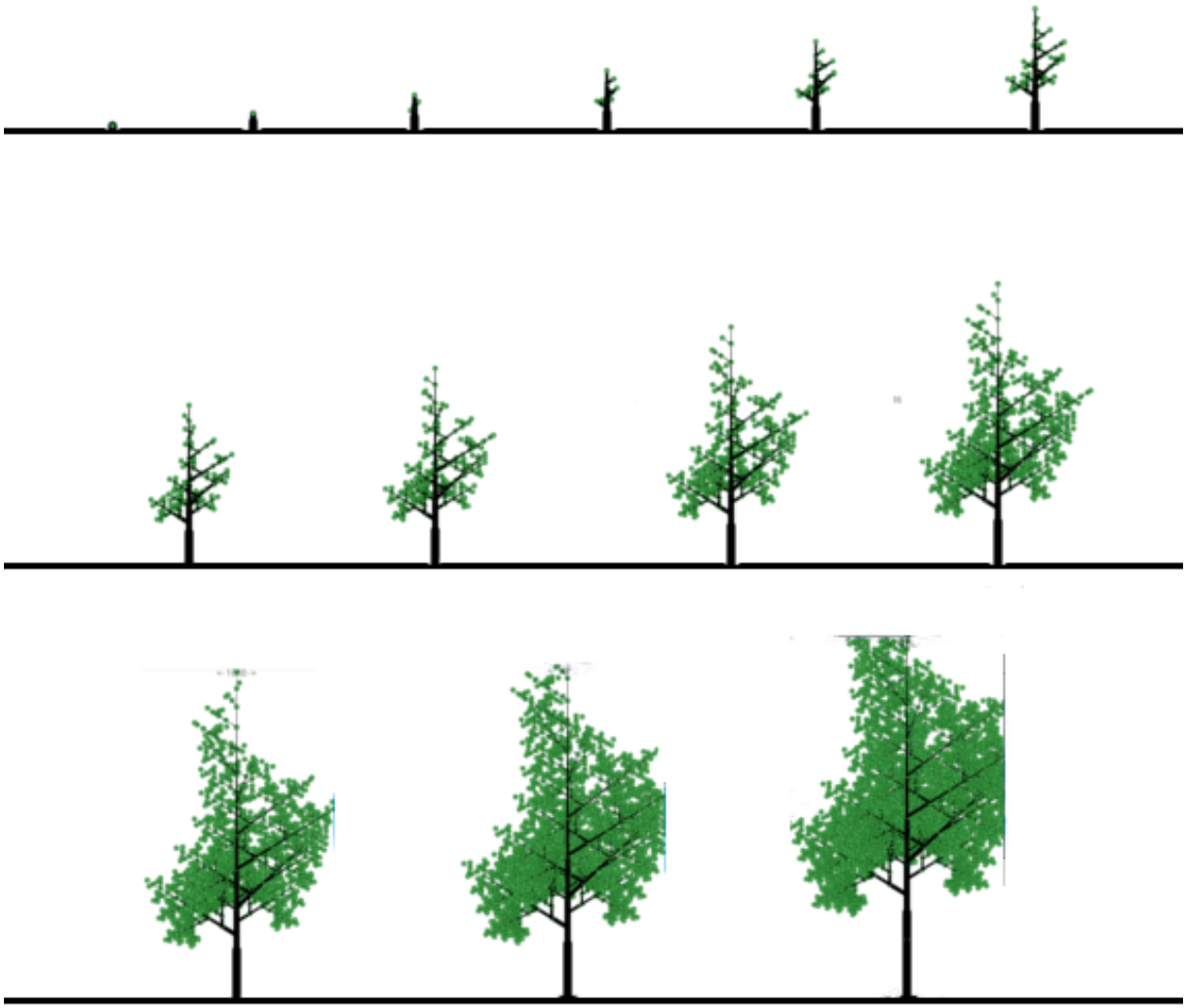


Abbildung 11: Wachstum eines erzeugten Baumes durch Verlängerung der Linien

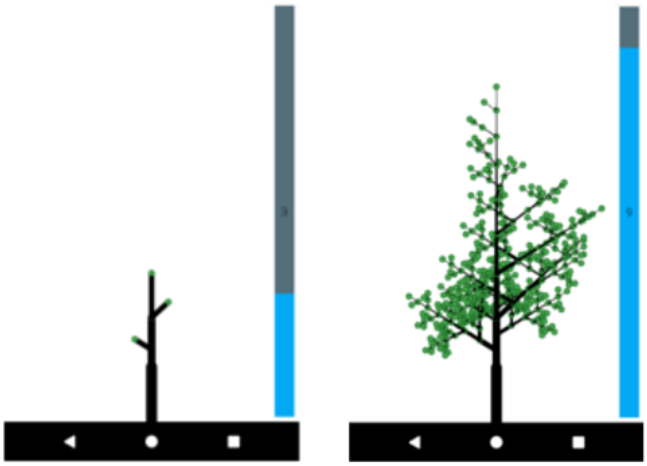


Abbildung 12: Wachstum der Bäume als Fortschrittsanzeige

4.10 Fortschrittsanzeige

Um das Wachstum der Bäume als Fortschrittsanzeige zu veranschaulichen, wird in der Hauptansicht neben den Bäumen eine vertikale *progressbar* in Form einer sich füllenden Linie angezeigt [Stac]. Diese ist entsprechend den zehn möglichen Baum-Generationen in zehn Schritte unterteilt, die zusammen dem Fortschritt von 0% bis 100% darstellen (12, abgebildet ist das Wachstum für 30% und 90%).
Im Moment ist es aber auch möglich, den Baum in mehr als zehn Generation zu generieren (11 bildet das Wachstum eines Baums bis in die 13. Generation ab).

5 Ergebnis

In dieser Arbeit ist ein Programm entstanden, dass mit Hilfe von L-Systemen Abbilder von Bäumen erstellen kann. Das Programm erzeugt Bäume durch zwei verschiedene Methoden der grafischen Interpretation. Die durch die zweidimensionale Methode (4.8.1) erstellten Bilder sind in 13 zu sehen, die durch die dreidimensionale Methode (4.8.2) erstellten Bilder in 14.

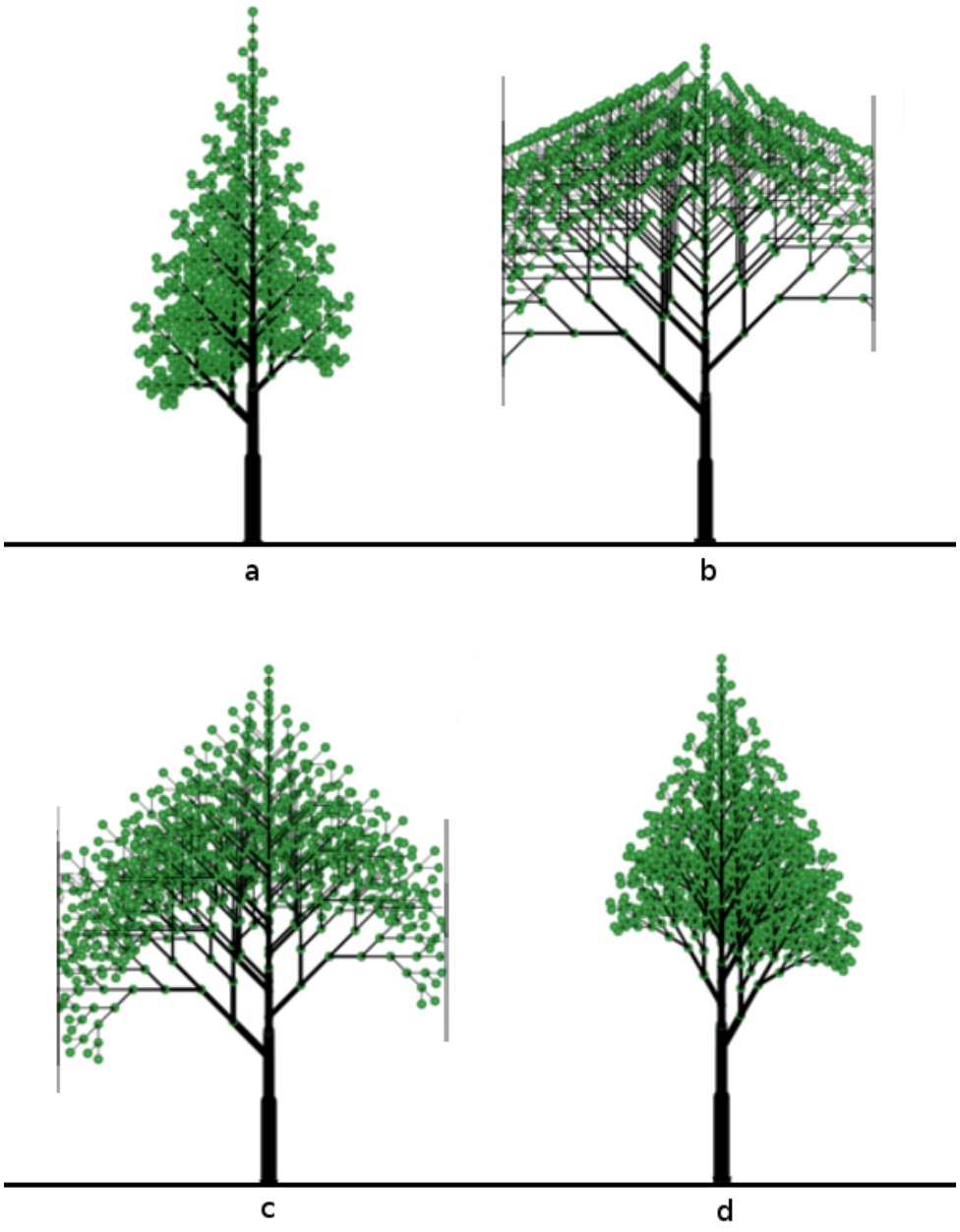


Abbildung 13: In 2D gezeichnete Bäume. Einige der Bäume stoßen an den Rand des Bildschirms.

Die erzeugten Grafiken werden im folgenden Kapitel diskutiert.

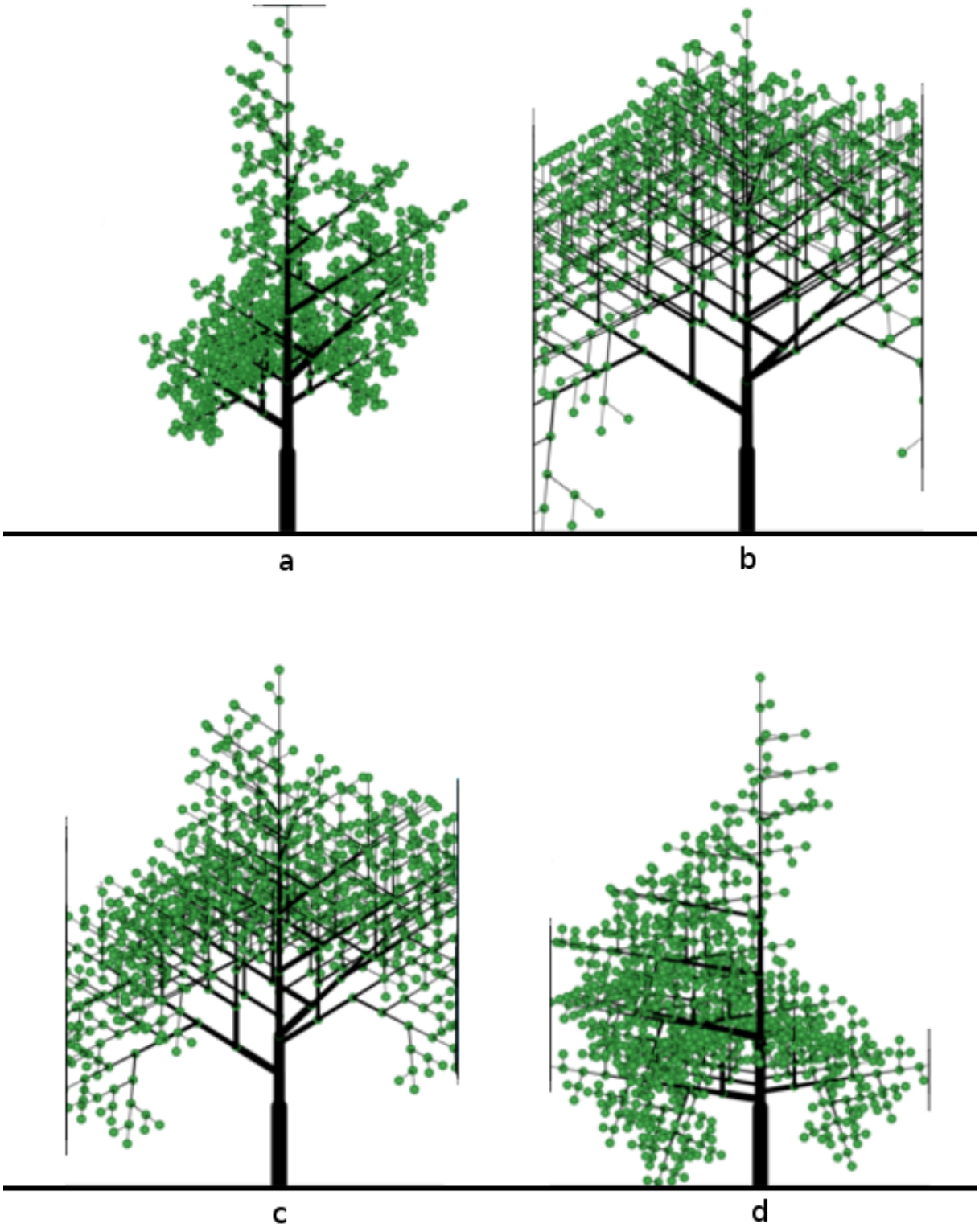


Abbildung 14: In 3D gezeichnete Bäume. Einige der Bäume stoßen an den Rand des Bildschirms.

6 Diskussion

Die in 5 präsentierten Grafiken sehen zum Teil noch sehr geometrisch aus. Insbesondere die durch die zweidimensionale Methode erzeugten Grafiken weisen viele anorganische Winkel und Wiederholungen auf, wo bei der Baum b aus 13 besonders auffällt.
Eine mögliche Ursache für diese Wiederholungen ist die Interpretation des Rollen und Nickens um die X- und die Y-Achse als Skalierung. Wie in 4.8.1 beschrieben, kann eine mögliche Lösung das erneute Anwenden der Transformationen sein.

Die mit der dreidiemnsionalen Methode erstellten Bäume (**??**) sehen organischer aus, die Böume b und c aus der Abbildung weisen aber ebenfalls viele geometrische Wiederholungen auf.
Eine möglicher Grund dafür kann das falsche Anwenden der Rotationsmatrizen sein.

Die erstellten Baumgrafiken können aber trotz ihrer Probleme als Schattenbilder realer Baumarten interpretiert werden: So weisen die Bäume a und d aus 13 eine den Zypressen ähnelnde Wuchsform auf (Vergleiche mit 15a), während Baum c aus 14 einem Obstbaum ähnelt (Vergleiche mit 15b).

Es ist also möglich Abbilder von Bäumen zu zeichnen. Dabei muss aber zwischen den Anwendungsgebieten unterschieden werden: Das Programm eignet sich in der aktuellen Version nicht, detailreiche Bäume zu erstellen. Es kann aber schnell viele verschiedene Varianten eines oder mehrerer Bäume erstellen, die Unterstützend, also als Hilfs- oder Hintergrundgrafiken verwendet werden. Um detaillierte Bäume zu zeichnen ist weiterhin eine Person mit Grafikausbildung gefordert.

(a) Rauhborkige Arizona-Zypresse. Bildquelle: [Phi80, S.110]

(b) Oregon-Wildapfel. Bildquelle: [Phi80, S.138]

Abbildung 15: Verschiedene Baumschattenbilder. Bildquelle: [Phi80]

7 Zusammenfassung

In dieser Arbeit wurde ein Programm zum Erzeugen wachsender Abbilder realer Baumarten in Form einer Forschungsanzeige vorgestellt. Das Programm erfüllt alle gestellten Anforderungen:

Das Programm

- funtkioniert in Android API 16+ (Jellybean) (4.2). Es wurde auf verschiedenen Test-Geräten auf Basis der Allgemeinen Android-Geräte-Konfigurationen getestet.
- kann Bäume ohne Verwendung externer Mittel (OpenGL, Bibliotheken) erzeugen (4.8). Das Programm benutzt nur die Android-internen Mittel. Dabei wurden Probleme beim Zeichnen der Bäume im dreidimensionalen Raum gelöst.
- kann Bäume abbilden und darstellen (5). Die erzeugten Bäume können als Baumschattenbilder realer Baumarten interpretiert werden.
- lässt Bäume von Grund auf wachsen (4.8). Um das Wachstum zu veranschaulichen wird die Länge der Linien der geometrischen Interpretation auf Basis der Generation des erzeugten Baumes verlängert oder verkürzt (4.9.1).
- kann die Bäume überall auf dem Bildschirm wachsen lassen können. Dazu muss nur der Startpunkt des Systems verändert werden (4.8).
- kann Bäume in 2D darstellen. Die erzeugten Bäume werden auf einem zweidimensionalen Canvas gezeichnet (4.8).
- animiert das Erzeugen der Bäume (4.9.2).
- erzeugt die Bäume modulartig. Durch Verwenden verschiedener Farben, Definitionen und Produktionsregeln kann der Baum innerhalb der Anwendung ausgetauscht und verändert werden (4.8.3 und 4.5.2).
- ist wiederverwendbar. Die *activity* (Die Haupt-Methode) kann in beliebige Android-Projekten eingebunden werden.
- erzeugt Bäume automatisch. Der Benutzer kann in einem Seiten-Menü zwischen verschiedenen Bäumen wählen (4.4).

8 Ausblick

Das erstellte Programm kann Abbilder von Bäumen generieren und Anzeigen, ist aber noch erweiterbar.
So ist die Generierung höherer Baum-Generationen noch sehr langsam. Auch wird im Moment nur eines der beiden erwähnten L-Systeme zur Generation verwendet, das Andere ist unbenutzt.
Weiterhin können gezeigten Abbilder der Bäume verbessert werden. Dazu gibt es verschiedene Möglichkeiten:
Eine Möglichkeit ist, das L-System mit einem Sketch- oder Bild-basiertes System zu unterstützen oder zu ersetzen(Vergleiche 3.7). Auch kann die Chance auf das Erzeugen eines natürlich-aussehenden Baums durch Benutzen eines vordefinierten Baum-Modells erhöht werden [Che+08, S.6].
Eine weitere Möglichkeit ist, dass verwendete L-System zum Erzeugen spezifischer Baumarten und Bäume anzupassen, da die mit dem verwendeten L-Systemen erzeugten Abbilder eher generisch wirken [PL90, S.62]. Dies ist aber schwierig, da regel-basierter Systeme spezialisiertes biologisches und bio-mechanisches Wissen voraussetzten und auch kleine Änderungen der Regeln große Auswirkungen auf den Aufbau der Bäume haben können [Che+08, S.2].
An Stelle der Veränderung des kompletten Regelsatzes zum Erzeugen spezifischer Baumarten können Bäume statt dessen aus verschiedenen L-Systemen zusammengesetzt werden: Wenn ein erzeugter Baum eine für Kiefern-typische Krone aufweist, er aber keine geeigneten Äste besitzt, können diese durch ein anderes L-System erzeugte und mit dem Stamm des ersten Baums verknüpft werden. Dies ist weniger aufwändig und vielseitiger als das Erstellen eines neuen L-Systems für eine spezielle Baumart, erfordert aber das Verwenden mehrerer L-Systeme zum Erzeugen der verschiedenen Baumteile. Durch Verwendung eines solchen Baukastensystems ist auch das einfache Generieren biologischer Besonderheiten wie der Zwiesel bei Bäumen möglich.

Ein anderer Ansatz zum Verbessern der erzeugten Abbilder basiert auf der Modifikation der Zeichenumgebung:
Um die in der Natur vorkommenden Krümmungen von Ästen und Zweigen, die ein wichtiges Merkmal der Spezies sind, nachzuahmen, können zum Zeichnen der Linien Polygonzüge verwendet werden [Pru+01, S.29]. Dabei werden die Linien aus mehreren Punkten zusammengestellt, um krumme und gebogene Äste und Zweige zu erstellen. Auch kann das Zeichnen der Blätter verbessert werden, so dass diese nicht mehr nur am Endpunkt sondern auch seitlich des Zweiges in gleichmäßigen Abständen gezeichnet werden. Weiterhin werden die Äste im Programm bisher nur mit schwarzer Farbe gemalt, Bäume haben aber meistens eine braune Rinde.
Neben dem Verwenden von Farbe ist auch ein texturieren der Blätter und Rinde des Baums

vorstellbar.

Literaturverzeichnis

[And16] Android Developers. *OpenGL ES — Android Developers*. 11.10.2016. URL: https://developer.android.com/guide/topics/graphics/opengl.html (besucht am 01. 06. 2017).

[And17a] Android Developers. *Canvas and Drawables — Android Developers*. 8.02.2017. URL: https://developer.android.com/guide/topics/graphics/2d-graphics.html#draw-with-canvas (besucht am 01. 06. 2017).

[And17b] Android Developers. *Canvas — Android Developers*. 17.05.2017. URL: https://developer.android.com/reference/android/graphics/Canvas.html (besucht am 02. 06. 2017).

[And17c] Android Developers. *PointF — Android Developers*. 17.05.2017. URL: https://developer.android.com/reference/android/graphics/PointF.html (besucht am 02. 06. 2017).

[And17d] Android Developers. *SurfaceHolder — Android Developers*. 17.05.2017. URL: https://developer.android.com/reference/android/view/SurfaceHolder.html (besucht am 01. 06. 2017).

[Lin68] A. Lindenmayer. „Mathematical models for cellular interactions in development. II. Simple and branching filaments with two-sided inputs“. In: *Journal of theoretical biology* 18 (3 1968), S. 300–315. URL: http://www0.cs.ucl.ac.uk/staff/p.bentley/teaching/L6_reading/lsystems.pdf (besucht am 28. 05. 2017).

[Hon71] Hisao Honda. „Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body“. In: *Journal of theoretical biology* 31.2 (1971), S. 331–338.

[Phi80] Roger Philips. *Das grosse Kosmosbuch der Baeume*. Franck, 1980.

[Alv84] Alvy Ray Smith. „Plants, Fractals, and Formal Languages“. In: *ACM SIGGRAPH Computer Graphics* 18 (3 1984), S. 1–10. ISSN: 00978930. URL: http://alvyray.com/Papers/CG/PlantsFractalsandFormalLanguages.pdf (besucht am 28. 05. 2017).

[PH89] Przemyslaw Prusinkiewicz und James Hanan. *Lindenmayer systems, fractals, and plants*. Springer-Verlag New York, Inc., 1989.

[PL90] Przemyslaw Prusinkiewicz und Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 1990. URL: http://algorithmicbotany.org/.

[HP93] James Hanan und Przemyslaw Prusinkiewicz. *Parametric L-systems and their application to the modelling and visualization of plants*. Citeseer, 1993.

[PHM93] Przemyslaw Prusinkiewicz, Mark S Hammel und Eric Mjolsness. „Animation of plant development". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM. 1993, S. 351–360.

[PJM94] Przemyslaw Prusinkiewicz, Mark James und Radomír Měch. „Synthetic topiary". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. the 21st annual conference. (Not Known). Hrsg. von Dino Schweitzer. ACM Special Interest Group on Computer Graphics and Interactive Techniques. New York, NY: ACM, 1994, S. 351–358. ISBN: 0897916670. DOI: 10.1145/192161.192254. URL: http://algorithmicbotany.org/papers/topiary.sig94.pdf (besucht am 28. 05. 2017).

[Pru+95] Przemyslaw Prusinkiewicz u. a. „The artificial life of plants". In: *Artificial life for graphics, animation, and virtual reality* 7 (1995), S. 1–1.

[MP96] Radomír Měch und Przemyslaw Prusinkiewicz. „Visual models of plants interacting with their environment". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. the 23rd annual conference. (Not Known). Hrsg. von John Fujii. ACM Special Interest Group on Computer Graphics and Interactive Techniques. New York, NY: ACM, 1996, S. 397–410. ISBN: 0897917464. DOI: 10.1145/237170.237279. (Besucht am 28. 05. 2017).

[Pru+96] Przemyslaw Prusinkiewicz u. a. „L-systems: from the theory to visual models of plants". In: *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences* 3 (1996), S. 1–32.

[Pru+97] Przemyslaw Prusinkiewicz u. a. „Visual models of plant development". In: *Handbook of formal languages*. Springer, 1997, S. 535–597.

[DL01] Oliver Deussen und Bernd Lintermann. „Computerpflanzen". In: *Spektrum der Wissenschaft* 2 (2001), S. 58–65.

[Pru+01] Przemyslaw Prusinkiewicz u. a. „The use of positional information in the modeling of plants". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, S. 289–300.

[AP03] Matt Aitken und Martin Preston. „Grove: a production-optimised foliage generator for The Lord of the Rings: The Two Towers“. In: *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM. 2003, S. 37–38.

[the04] theprodukkt. *.kkrieger*. 2004. URL: https://web.archive.org/web/20120204065754/http://www.theprodukkt.com/faqmaking (besucht am 26. 05. 2017).

[KM06] George Kelly und Hugh McCabe. „A survey of procedural techniques for city generation“. In: *ITB Journal* 14 (2006), S. 87–130. URL: https://pdfs.semanticscholar.org/1e9c/1349b91d9352148a4567d172df7a965c180c.pdf.

[Che+08] Xuejin Chen u. a. *Sketch-based tree modeling using Markov random field*. Bd. 5. ACM, 2008. ISBN: 1450318312. URL: http://dl.acm.org/citation.cfm?id=1409062.

[Dai08] John Daintith. „A dictionary of computing“. eng. In: *Oxford reference online premium* (2008). URL: http://www.oxfordreference.com/.

[GK08] Björn Ganster und Reinhard Klein. „1-2-tree: Semantic Modeling and Editing of Trees“. In: *VMV* (2008), S. 51–60.

[BL10] Adrian Bejan und Sylvie Lorente. „The constructal law of design and evolution in nature“. eng. In: *Philosophical transactions of the Royal Society of London. Series B, Biological sciences* 365 (1545 2010). Journal Article Research Support, U.S. Gov't, Non-P.H.S., S. 1335–1347. ISSN: 0962-8436. DOI: 10.1098/rstb.2009.0302. eprint: 20368252.

[Tog+11] Julian Togelius u. a. „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.

[Bou+12] Frédéric Boudon u. a. „L-Py. An L-system simulation framework for modeling plant architecture development based on a dynamic language“. In: *Frontiers in plant science* 3 (2012), S. 76. URL: http://journal.frontiersin.org/article/10.3389/fpls.2012.00076/full.

[PCG14] PCGwiki. *Procedural Generation - Procedural Content Generation Wiki*. 2014. URL: http://pcg.wikidot.com/pcg-algorithm:procedural-generation (besucht am 25. 05. 2017).

[STN14] Noor Shaker, Julian Togelius und M Nelson. *Procedural Content Generation In Games*. 2014.

[Sta+14] Ondrej Stava u. a. „Inverse procedural modelling of trees“. In: *Computer Graphics Forum*. Bd. 33. 6. Wiley Online Library. 2014, S. 118–131.

[Est+15] Rolando Estrada u. a. „Tree topology estimation“. In: *IEEE transactions on pattern analysis and machine intelligence* 37 (8 2015), S. 1688–1701. ISSN: 0162-8828.

[And17e] Android Developers. *APK Expansion Files — Android Developers*. 2017. URL: https://developer.android.com/google/play/expansion-files.html (besucht am 23. 05. 2017).

[And17f] Android Developers. *Dashboards — Android Developers*. 2017. URL: https://developer.android.com/about/dashboards/index.html#Screens (besucht am 22. 05. 2017).

[And17g] Android Developers. *Download Android Studio and SDK Tools — Android Studio*. 2017. URL: https://developer.android.com/studio/index.html (besucht am 01. 06. 2017).

[And17h] Android Developers. *Reduce APK Size — Android Developers*. 2017. URL: https://developer.android.com/topic/performance/reduce-apk-size.html (besucht am 22. 05. 2017).

[And17i] Android Developers. *Supporting Multiple Screens — Android Developers*. 2017. URL: https://developer.android.com/guide/practices/screens_support.html (besucht am 23. 05. 2017).

[UDA17a] UDATA. *UDATA — Wir stehen für Umwelt und Bildung*. 2017. URL: http://udata.de/ (besucht am 25. 05. 2017).

[UDA17b] UDATA. *uRnature - die App zum WaldKlima-Projekt*. 2017. URL: http://urnature.info/ (besucht am 25. 05. 2017).

[Wik17] Wiktionary. *procedural - Wiktionary*. 2017. URL: https://en.wiktionary.org/wiki/procedural (besucht am 26. 05. 2017).

[And] Android Developers. *View — Android Developers*. URL: https://developer.android.com/reference/android/view/View.html (besucht am 01. 06. 2017).

[Sam] Samsung. *Galaxy S8+ 64GB (Unlocked) Phones - SM-G955UZKAXAA — Samsung US*. URL: http://www.samsung.com/us/mobile/phones/galaxy-s/galaxy-s8-plus-64gb--unlocked--sm-g955uzkaxaa/#specs (besucht am 03. 06. 2017).

[Spe] SpeedTree. *Tree Models & Animated Forest FAQs — SpeedTree*. URL: http://www.speedtree.com/tree-models-faq.php (besucht am 03. 06. 2017).

[Staa] Stack Overflow. *android - Create a new bitmap and draw new pixels into it - Stack Overflow*. URL: https://stackoverflow.com/questions/10180449/create-a-new-bitmap-and-draw-new-pixels-into-it (besucht am 01. 06. 2017).

[Stab] Stack Overflow. *android - Get new position of coordinate after rotation with Matrix - Stack Overflow*. URL: https://stackoverflow.com/questions/7795028/get-new-position-of-coordinate-after-rotation-with-matrix (besucht am 02. 06. 2017).

[Stac] Stack Overflow. *Android - set a ProgressBar to be a vertical bar instead of horizontal? - Stack Overflow*. URL: https://stackoverflow.com/questions/3926395/android-set-a-progressbar-to-be-a-vertical-bar-instead-of-horizontal/40538069#40538069 (besucht am 02. 06. 2017).

[Wal] Waldklimafonds. In: (). URL: https://www.waldklimafonds.de/fileadmin/SITE_MASTER/content/Dokumente/Projektbeschreibung/062_WaldKlimaApp.pdf (besucht am 23. 05. 2017).

Anhang A 1. Zeichenerklärung L-System

Erklärung aller von der Schildkröte im Programm interpretierten Zeichen.

F	Bewege dich um Standard-Distanz d vorwärts und zeichne eine Linie zwischen Anfangs- und Endpunkt
$F(a)$	Bewege dich um a vorwärts und zeichne eine Linie zwischen Anfangs- und Endpunkt
[	Speichere die aktuellen Informationen der Schildkröte auf dem *stack*
]	Lade die neuen Informationen vom *stack* in die Schildkröte
$+$	Drehe dich um Standardwinkel δ nach rechts.
$-$	Drehe dich um Standardwinkel δ nach links.
$+(a)$	Drehe um a nach links.
$\&$	Kippe um Standardwinkel δ nach unten.
$\wedge$	Kippe um Standardwinkel δ nach oben.
$\&(a)$	Kippe um a nach unten.
$\backslash$	Rotiere um Standardwinkel δ nach links.
$/$	Rotiere um Standardwinkel δ nach rechts.
$/(a)$	Rotiere um a nach rechts.
\$	Drehe dich bis du wieder in einer horzontalen Position bist.
$!(w)$	Setze die Breite der zu zeichnenden Linie auf w.

Anhang A 2. Ehrenwörtliche Erklärung

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Seminararbeit/Bachelorarbeit/Masterarbeit mit dem Titel

_____

_____

selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie nicht an anderer Stelle als Prüfungsarbeit vorgelegt habe.

_____

Ort

_____ _____

Datum Unterschrift