Mediadesign Hochschule
für Design und Informatik

Bachelorarbeit im Fachbereich Informatik
zur Erlangung des Bachelor of Science

Über Algorithmen zur Städtegenerierung

Fabian Oyvin Kopp

Matrikelnummer: 104101503

Erstgutachter:
Prof. Dr. habil. Mike Scherfner

Zweitgutachter:
Prof. Don-Oliver Matthies

14. Februar 2019

Vorwort

Die vorliegende Bachelorarbeit „Über Algorithmen zur Städtegenerierung“ entstand im Rahmen meines Studiums an der Mediadesign Hochschule für Design und Informatik in Berlin.

Die Idee für dieses Thema entwickelte sich bereits während des Studiums, da mir auffiel, wie viel Ressourcen benötigt werden, um eine Spiele-Welt zu erschaffen. Diese Abschlussarbeit richtet sich insbesondere an Programmierer und Informatiker mit Grundkenntnissen im Bereich der prozeduralen Generierung, mit dem Interesse an der automatisierten Generierung von natürlich erscheinenden Städten. Für sie soll diese Arbeit eine Zusammenfassung der Grundlagen darstellen und ein Startpunkt für weitere Entwicklung bieten.

Zusammen mit meinem Betreuer, Prof. Dr. habil. Mike Scherfner, habe ich die Fragestellung für dieses Bachelorarbeit konkretisiert. Seine kritische Hinterfragung des Themas und seine Beratung für das Herangehen bei der Erstellung der Bachelorarbeit haben mir geholfen, das Thema konsistent zu halten und einen roten Faden zu finden. Daher möchte ich ihm meinen besonderen Dank ausdrücken.

Schließlich danke ich insbesondere meinen Eltern. Ihre moralische Unterstützung während des Schreibprozesses hat mir geholfen, diese Abschlussarbeit zu einem erfolgreichen Ende zu führen.

Berlin den 14. Februar 2019

Fabian Kopp

Inhaltsverzeichnis

1 Abstract **1**
- 1.1 Fragestellung und Zielsetzung 1
- 1.2 Einleitung 1

2 Prozedurale Generierung: Historische Entwicklung **3**

3 Prozedural generierte Städte: verschiedene Ansätze **6**
- 3.1 Auf Lindenmayer-Systeme basierender Ansatz 6
- 3.2 Auf Agenten basierender Ansatz 8
- 3.3 Auf Vorlagen basierender Ansatz 9
- 3.4 Auf Tensoren basierender Ansatz 10
- 3.5 Auf Entwicklung nach Zeit basierender Ansatz 11

4 Städtewachstum: Merkmale in der Realität **13**
- 4.1 Altertum 13
- 4.2 Mittelalter 14
- 4.3 Neuzeit 15
- 4.4 Gegenwart 16
- 4.5 Mögliche Merkmale von Städten in der Zukunft 16

5 Welche Methoden werden für das Produkt verwendet? **17**

6 Theoretische Grundlagen **19**
- 6.1 Lindenmayer-System 20
  - 6.1.1 Einfache Lindenmayer-Systeme 21
  - 6.1.2 Parametrische Lindenmayer-Systeme 22
- 6.2 Noise: Gitterbasierter Ansatz 22
  - 6.2.1 Gradient-Noise: Perlin-Noise 24
  - 6.2.2 Simplex-Noise 29

7 Methoden **30**
7.1 Straßen 30
7.2 Gebäude 34

8 Quellcode **36**

9 Fazit **46**

10 Ausblick **49**
10.1 Visualisierung 49
10.2 Erweiterung der Parameter 50
10.3 Erweiterung der Methoden 50

Kapitel 1

Abstract

1.1 Fragestellung und Zielsetzung

Das Ziel dieser Arbeit ist es, ein Programm zur prozeduralen Generierung einer Stadt zu erstellen. Die daraus entstandene Stadt, soll aus einem generierten Straßennetz und Gebäuden bestehen, die ein natürliches Erscheinungsbild wieder geben. Ein weiteres Ziel des Projektes ist es, zu ermitteln, ob ein Programm mit prozeduraler Generierung helfen kann, Personalressourcen einzusparen. Insbesondere ist es von Interesse, ob die benötigte Zeit für die Erschaffung einer Stadt mit Hilfe des Programms signifikant reduziert werden kann. Außerdem wird im Nachgang evaluiert, ob die aufgewendete Arbeitszeit zur Programmierung dieses Projektes im Vergleich zu der daraus gewonnenen Zeit in der Entwicklung einer Stadt zu rechtfertigen ist und damit ein Mehrwert geschaffen wird. Der Nachweis für die Machbarkeit, wird mit Hilfe eines selbst entwickelten Programms geführt.

In dieser Arbeit werden verschiedene prozedurale Techniken für die Generierung einer Stadt aufgezeigt. Auf Grundlage dieser theoretischen Basis wird ein eigenes Programm für die Erstellung einer natürlich erscheinenden Stadt entwickelt.

1.2 Einleitung

Die Erschaffung einer Spiele-Welt ist eine große Herausforderung bei der Produktion von neuen Videospielen. Insbesondere für Spiele-Welten mit weitläufigen Level-Grenzen und freiem Erkundungsraum, gestaltet sich die Erstellung als herausfordernd und komplex. Außerdem muss zurzeit ein hohes Maß an zeitlichen Ressourcen, als auch an geeignetem Personal aufgewendet

werden, damit die Spiele-Welten den Ansprüchen der Spieler gerecht werden. Diese Faktoren stellen besonders kleinere Firmen vor ein großes Problem, da ihre Ressourcen deutlich mehr eingeschränkt sind.

Verschiedene prozedurale Techniken werden schon seit längerem für diese Herausforderungen eingesetzt. Dies gilt für die Bereiche der Texturierung, der Spezialeffekte als auch für die Generierung organischer und natürlicher Objekte. Für die Grundlagen zur Erstellung einer Spiele-Welt werden vielfältige Vorlagen genommen. In den meisten Fällen werden Landkarten und Grundbuchpläne als Ausgangslage genutzt. Es können aber auch zwei-dimensionale Vektorfelder (Tensorfelder) oder zufällig generierte Umgebungen als Grundlage für die Generierung verwendet werden. Basierend auf diesen Grundlagen werden zum Beispiel Straßenverläufe und Gebäudeformen durch die unterschiedlichen Methoden der Städtegenerierung erschaffen. Das Ziel dieser Arbeit ist es, ein Programm zu entwickeln, welches auf der Basis der prozeduralen Generierung Städte mit scheinbar natürlichen Erscheinungsbildern erstellt. Es existieren bereits fortgeschrittene Programme, welche den Generierungsprozess in Bezug auf das Straßenwachstum optimiert haben.
Grundsätzlich besteht auf dem Markt der Videospiele der Bedarf, dass eine Spiele-Welt möglichst ohne großen Aufwand generiert werden kann. Insbesondere, weil die Spiele-Welten immer größer, komplexer und offen gestaltet werden sollen.

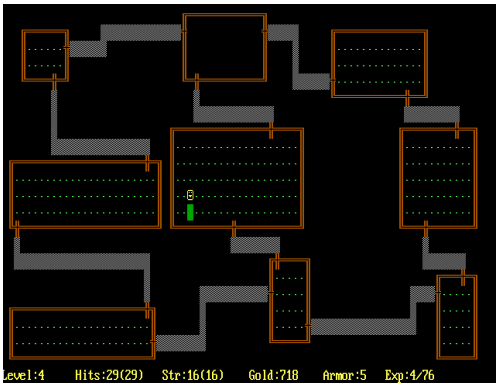
Kapitel 2

Prozedurale Generierung: Historische Entwicklung

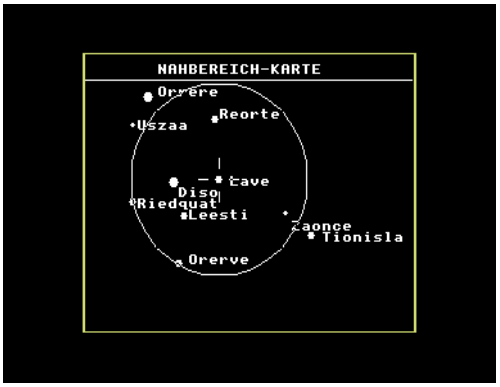
Dieses Kapitel befasst sich mit den bereits in der Spiele-Industrie genutzten prozeduralen Techniken im Bereich der Videospiele. Es wird erläutert, welche Techniken angewendet werden und welche Entwicklungsschritte die prozedurale Generierung genommen hat.

In der Geschichte von Videospielen wurde prozedurale Generierung von Inhalten schon lange zu Hilfe genommen und vielfältig eingesetzt. Insbesondere dann, wenn es darum ging große Spiele-Welten zu erschaffen. Die Generierung betrifft Texturen, Spezialeffekte und Sound, aber auch Objekte und/oder ganze Landschaften werden und wurden prozedural erschaffen. Der Vorteil der prozeduralen Generierung besteht unter anderem darin, dass aus einer geringeren Anzahl an Elementen eine sehr viel größere Menge an neuem Inhalt generiert werden kann. Durch diese Maßnahme kann Speicherplatz gespart werden. Außerdem kann man mit prozeduraler Generierung Zeit und Ressourcen einsparen. Diese eingesparten Ressourcen können stattdessen in anderen Bereichen verwendet und damit optimaler eingesetzt werden.

Bereits die Klassiker der Videospiele *Beneath Apple Manor* (1978) und das dem Genre namensgebende Spiel *Rogue* (1980) benutzten prozedurale Generierung zur Erstellung ihrer Spiele-Welten. Die prozedurale Generierung erlaubte es ihnen, komplexes Gameplay und ständig neue Herausforderungen für den Spieler zu erschaffen, ohne dass der Entwickler viel Zeit in die Erschaffung einer Spiele-Welt stecken muss [1]. Aber auch andere Genre nutzten bereits früher prozedurale Generierung. Die Raumkampf und Handelssimulation *Elite* (1984) nutzte nicht einfach nur prozedurale Generierung, die



(a) Level in *Rogue*



(b) Sonnensystem in *Elite*

Abbildung 2.1: Prozedural generierte Inhalte

Entwickler trieben es beim Entwurf des Spiels zum ersten Mal ins Extreme. Ursprünglich sollte das Spiel 2^{48} Galaxien in 256 Sonnensystemen beinhalten. Die finale Version hatte dann jedoch lediglich acht Galaxien. Der Grund für die Reduktion war der Publisher. Sie befürchteten, dass ein so gigantisches Universum bei den Spielern nicht auf Akzeptanz stoßen würde [2]. Außerdem waren die so geschaffenen Galaxien sehr repetitiv, weil sie nur auf eine sehr geringe Anzahl von Grundlagen zurückgriffen.

(a) Fraktale Berge in *Rescue on Fractalus*

(b) Screenshot aus *.kkrieger*

Abbildung 2.2: Prozedural generierte Inhalte

Im Bereich der rein grafischen prozeduralen Generierung wurden ebenfalls schon früh Fortschritte erzielt. Im Jahre 1985 wurde zum Beispiel für das Spiel *Rescue on Fractalus*, Fraktale zur visuellen Darstellung seiner zerklüfteten Berge genutzt. Ein weiteres Beispiel ist das Spiel *.kkrieger* (2004),

welches aus einer 96kB großen Microsoft Windows Anwendung mehrere hundert Megabyte an 3D-Modellen und Texturen durch prozedurale Generierung erstellt.

Die Methodik der prozeduralen Generierung ist jedoch nicht auf die Erstellung visueller Inhalte beschränkt. Auch Gameplay beeinflussende Elemente können von prozeduralen Systemen generiert werden. Action Rollenspiele und MMORPG (massive multiplayer online role playing games) nutzen diese Möglichkeit für ihre Belohnungssysteme. Diese Systeme können anhand von Bewertungskriterien eine entsprechende Belohnung generieren. Ist der Spieler zum Beispiel außergewöhnlich erfolgreich bei der Erfüllung seiner Aufgabe oder ist sein Level vergleichsweise niedrig im Vergleich zu der Herausforderung, dann wird seine Belohnung für den erfolgreichen Abschluss besser/höher ausfallen, als bei einem anderen Spieler. Diese Art von Belohnungssystem wird beispielsweise in der *Borderlands*-Serie für die erbeuteten Waffen verwendet [3].

Das bereits erwähnte Genre der Roguelikes erstellt bei jedem Spielstart neue Welten mit Hilfe der prozeduralen Generierung. Diese Methode ermöglicht es dem Entwickler, ohne großen Aufwand im Design, konstant neue Herausforderungen und Erfahrungen für den Spieler zu liefern. Diese Vielfalt ermöglicht es, den Spieler länger an ein Spiel zu binden. Nicht nur im Genre Roguelike wird der prozedurale Generierungsansatz verwendet, sondern er wird auch in vielen Openworld- und Survival-Spielen angewendet. Spiele wie *Dwarf Fortress* (2006) oder *No Man's Sky* (2016) nutzen die prozedurale Generierung nicht nur für die Erstellung einer Welt, sondern auch für die Generierung von Lebewesen, des Wetters, die historische Entwicklung der Welt und weiteren Inhalten [4, 5, 6]. Das dem Spiel zu Grunde liegende prozedurale System wird außerdem nach der Generierung der Spiele-Welt weiter verwendet. Dadurch erhält der Spieler den Eindruck, dass die Spiele-Welt sich weiter entwickelt und das Leben seinen natürlichen Gang nimmt.

Kapitel 3

Prozedural generierte Städte: verschiedene Ansätze

Wie die meisten Techniken der prozeduralen Generierung von Inhalten, sind auch die im folgenden beschriebenen Methoden halbautomatisch. Als Grundlage für die Erstellung der Städte werden in den meisten Fällen reale Datenkarten mit Informationen über zum Beispiel die Bevölkerungsdichte oder die Land-Wasser-Verteilung verwendet. Meistens ist es dem Nutzer möglich parametrische Daten, wie zum Beispiel die maximale Länge von Brücken oder die Schwellenwerte wann sich Nebenstraßen bilden, zu beeinflussen. In den folgenden fünf Punkten werden verschiedene Ansätze zur prozeduralen Generierung von Städten beschrieben. Die Beschreibungen umfassen die benötigten/möglichen Inputs des Users und eine kurze Zusammenfassung des zugrundeliegenden Algorithmus.

3.1 Auf Lindenmayer-Systeme basierender Ansatz

Der im folgenden beschriebene Ansatz für die prozedurale Generierung von Städten basiert auf der Arbeit von Parish und Müller (2001) [7].

Als Grundlage für die Generierung einer Stadt wird eine vergleichsweise große Anzahl von Datenkarten benötigt. Land-Wasserkarten und Höhenkarten werden für die Bestimmung von Hindernissen bei der Generierung genutzt und bilden damit die Grundlage für die prozedurale Generierung. Für die Bestimmung, wie sich die Straßen entwickeln und um welche Art von Straße es sich handeln soll, werden weitere Daten bezüglich der Bevölkerungsdichte und des Straßenmusters verwendet. In Kombination mit Zonenkarten und

einer Karte zur Bestimmung der maximalen Gebäudehöhe, werden nach der Generierung des Straßennetzes auch Gebäude erstellt. Durch dieses Zusammenspiel der Daten entsteht eine natürlich wirkende Stadt.

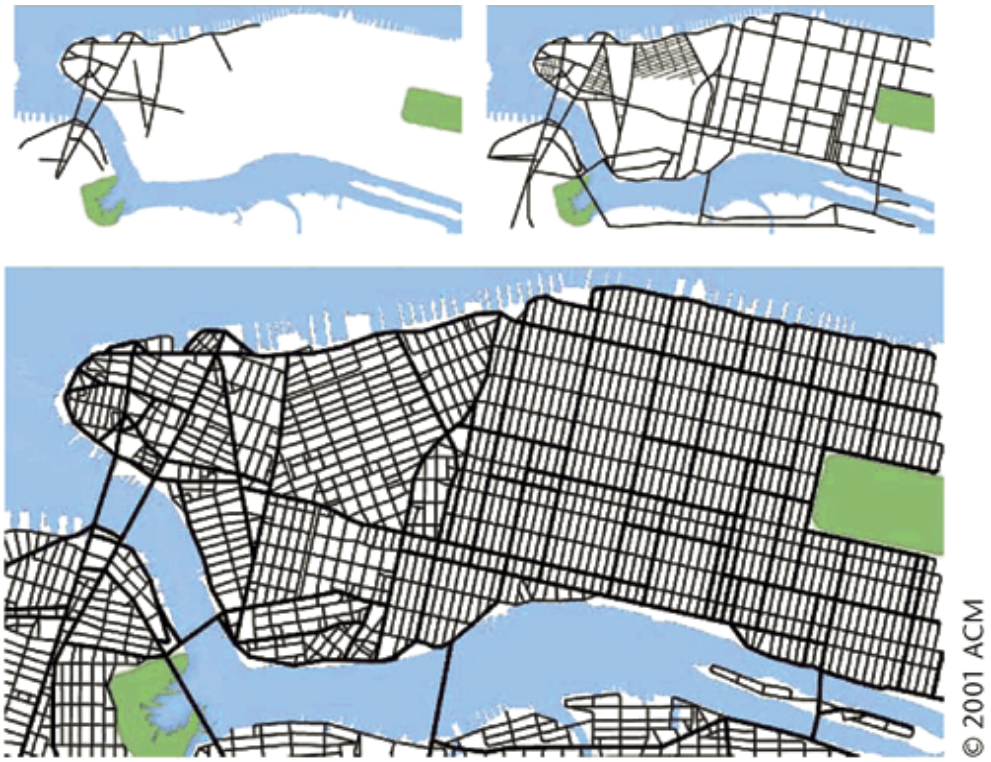


Abbildung 3.1: Prozedurale Straßennetzgenerierung auf Grundlagen des Stadtteils Manhatten (Parish und Müller (2001), Fig. 9)

Für die Generierung des Straßennetzes wird ein parametrisches Lindenmayer-System (L-System) verwendet. Dieses System erstellt in einem ersten Schritt einen neuen potentiellen Straßenabschnitt. Dieses Straßenbruchstück folgt globalen Zielen. Globale Ziele sind zum Beispiel, dass Zentren größter Bevölkerungsdichte mittels Hauptstraßen untereinander verbunden werden oder dass diese Straßen gewünschte Muster bilden. Im nächsten Schritt wird das neue Teilstück mit lokalen Gegebenheiten eingeschränkt. Diese lokalen Einschränkungen versuchen den Straßenverlauf nicht nur zu limitieren, sondern auch zu optimieren. Zu den Optimierungen gehört unter anderem, dass zum Beispiel lose Straßenenden in bereits bestehende Straßenkreuzungen integriert werden und die Anpassung der Parameter für den nächsten Schritt der Iteration. Falls jedoch ein Straßenbruchstück auf ein Hindernis trifft, wird

das Element insofern limitiert, dass zum Beispiel versucht wird durch eine Verkürzung oder mit Hilfe einer Deformation das Segment außerhalb der Problemzone enden zu lassen. Daraus resultiert zum Beispiel ein Straßenverlauf entlang der Küste. Falls sich aber das neue Straßensegment nicht anpassen lässt, wird es aus der Generierung entfernt.

3.2 Auf Agenten basierender Ansatz

Der hier beschriebene Ansatz der prozeduralen Generierung basiert auf der Arbeit von Lechner und Wilensky (2003) [8].

Dieser Agenten basierende Ansatz benötigt deutlich weniger Informationen für die Generierung der Städte als im zuvor beschriebenen Ansatz von Parish und Müller [7]. Es werden lediglich eine Höhenkarte und eine Start-Seed benötigt. Dennoch ist es dem Nutzer möglich, Einfluss auf die Entstehung der Stadt zu nehmen, indem lokale Modifikationen der vordefinierten Entstehungsregeln angegeben werden. Potentielle Modifikationen sind zum Beispiel wie eine spezifische Region aussehen soll oder welche Straßenmuster sich unter welchen Bedingungen bilden können.

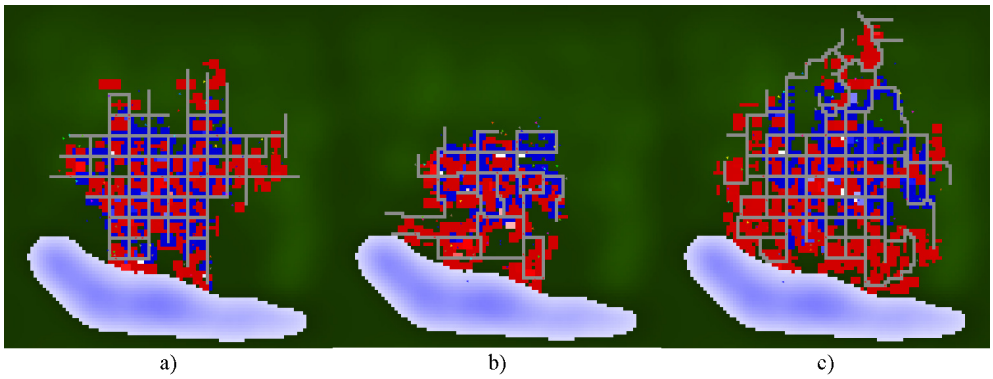


Abbildung 3.2: Beispiele unterschiedlicher Stadtstrukturen basierend auf unterschiedlichen Straßennetzwerken: a) Gittermuster, b) verzweigtes Muster, c) Kombination aus Gittermuster und verzweigtem Verhalten (Lechner und Wilensky (2003), Fig. 5)

Als Algorithmus für die Generierung der Straßen wird ein System von multiplen und gleichzeitig agierenden Agenten verwendet. Diese Agenten sind auf zwei unterschiedliche Arten definiert. Die erste Art sind Entwicklungsagenten, welche mit Hilfe der Höhenkarte versuchen, bisher nicht erreichbare

Gebiete mit dem bereits bestehenden Straßennetz zu verbinden. Die zweite Art der Agenten stellt einen Einwohner der Stadt dar, der sich den kürzesten Weg zwischen zwei zufällige gewählten Punkten auf dem vorhandenen Straßennetz heraussucht. Bei der Suche nach dem kürzesten Weg (Pathfinding), beschränkt sich der Agent aber nicht nur auf das bestehende Straßennetz. Falls diese zweite Art der Agenten beim Pathfinding zwischen diesen Punkten eine signifikante Verbesserung der Strecke durch die Generierung einer neue Straße findet, wird diese Straße dem Netzwerk hinzugefügt. Den Straßen wird in diesem Ansatz zusätzlich ein Einflussbereich zugewiesen, in welchem sich im Anschluss Gebäude entwickeln können.
Auch bei der Generierung von Gebäuden, gibt es weitere Agenten die diese Entwicklung bestimmen.Sie sind für die Konstruktion von zwei unterschiedlichen Arten der Gebäude zuständig (Wohnungen und Geschäfte). Die Generierung der Gebäude verläuft aber unabhängig von der Gebäudeart. Für die Errichtung von kommerziell genutzten Häusern werden bevorzugt Gebiete mit hohem Straßeneinfluss (z.B. hohe Straßendichte oder geringe Distanz zur Straße) gewählt. Wohnungen werden im Gegensatz dazu bevorzugt in Gebieten mit geringem Straßeneinfluss und weiter entfernt von Straßen errichtet.

Im Gegensatz zum Ansatz von Parish und Müller [7], ist das System an ein rechteckiges Raster gebunden und es wird nicht unterschieden, um welche Art der Straße es sich handelt. Durch die unterschiedlichen Arten der Agenten und indem den Straßen ein Einflussbereich gegeben wird, gelingt es Lechner und Wilensky dennoch ein realistisches Stadtbild zu generieren.

3.3 Auf Vorlagen basierender Ansatz

Die hier beschriebene Methode von Kelly und McCabe (2006) [9] greift den von Sun et al. [10] beschrieben Ansatz auf. Es werden Netzwerkmuster der Straßen aus Vorlagen auf die Höhenkarte übertragen. Im nächsten Schritt wird der Straßenverlauf den Gegebenheiten angepasst. Im Gegensatz zum Vorgehen nach Sun et al. [10] werden bei diesem Ansatz jedoch nur die ersten und formgebenden Straßen erstellt. Sobald diese generiert wurden, können sie vom Nutzer verändert werden und damit die Generierung der sekundären Straßen indirekt beeinflussen. Diese sekundären Straßen werden innerhalb des primären Straßennetzes mittels eines L-System basierten Algorithmus, ähnlich wie von Parish und Müller [7] beschrieben, generiert. Damit die Anwendung an Speicherplatz sparen kann, werden die Gebäude ebenfalls prozedural aus simplen geometrischen Grundformen generiert.

In erster Linie war die Arbeit von Kelly und McCabe [9] eine Untersuchung der bestehenden Techniken zur prozeduralen Generierung von Städten. Sie untersuchten die Techniken auf deren Funktionalität und sie hinterfragten die daraus resultierenden Ergebnisse kritisch. Punkte zur Verbesserung im Bereich der Nutzerfreundlichkeit werden von ihnen, wie oben beschrieben wurde, insbesondere vorgeschlagen. Kelly und McCabe versuchen also nicht das Rad neu zu erfinden, sondern die Techniken optimal zu vereinen. Zum Beispiel wird die variable Darstellung von Gebäuden nach Wonka et al. [11] bei der Generierung ebenfalls verwendet. Aus dieser Fusion der Techniken entsteht eine natürlich wirkende Stadt.

3.4 Auf Tensoren basierender Ansatz

Dieser Ansatz von Chen et al. (2008) [12] benötigt als Grundlage für die Generierung Land-Wasserkarten, Höhenkarten und Karten der Bevölkerungsdichte. Mit diesen Datenkarten als Grundlage wird ein erstes Tensorfeld generiert. Dabei werden die Land-Wasserkarten genutzt um die Grenzen zu definieren und die Tensoren werden dementsprechend angepasst, dass ihre Haupt-Eigenvektoren parallel zu den Grenzen verlaufen. Die Höhenkarte wird bei der Generierung genutzt, um das Tensorfeld so zu beeinflussen, dass der entstehende Straßenverlauf dem geringsten Höhenunterschied folgt. Nachdem das Tensorfeld erstellt wurde, kann es direkt zur Generierung des primären Straßennetzes genutzt werden oder vom Nutzer mittels „Pinselstrich“ nach den eigenen Wünschen bearbeitet werden. Anders als bei den vorherigen Ansätzen ist es damit möglich, dass der Nutzer direkter das resultierende Straßennetz beeinflusst. Es ist dem Nutzer nicht nur möglich den Straßenverlauf zu deformieren, sondern auch zu bestimmen, wo welche Art von Straßenmuster entstehen soll. Dieses Vorgehen wird iterativ angewandt, um auch sekundäre Straßen innerhalb des primären Straßennetzes zu erstellen oder auch um nachträglich ganze Abschnitte zu bearbeiten, zum Beispiel um einen Park zu platzieren.

Im Gegensatz zu den vorhergehenden Ansätzen hat diese Methode den Vorteil, dass der Nutzer direkt auf die Generierung Einfluss nehmen kann. Damit man qualitativ hochwertige Resultate erzielt, muss der Nutzer diese Möglichkeit auch tatsächlich nutzen.

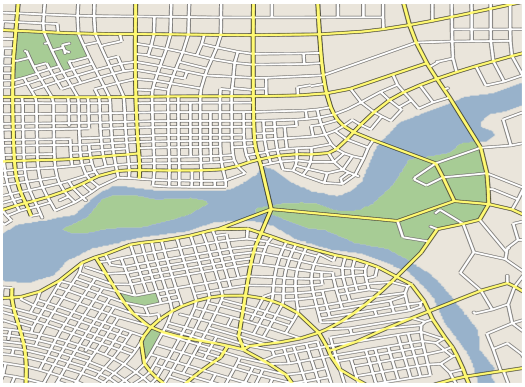


Abbildung 3.3: Generiertes Straßennetz mit Hilfe des Tensorfeld und manueller Bearbeitung auf Grundlage des Benue Fluss in Nigeria (Chen et al. (2008), Fig. 1)

3.5 Auf Entwicklung nach Zeit basierender Ansatz

Der hier beschriebene Ansatz von Weber et al. (2009) [13] verwendet Höhenkarten und Land-Wasserkarten als Grundlage für die Generierung. Zu Beginn muss der Nutzer noch ein Start-Setup festlegen, was eine einzelne Straße, aber auch eine ganze Stadt sein kann. Außerdem müssen gewisse Parameter vordefiniert werden. Dazu gehört zum Beispiel die prozentuale Verteilung der Landnutzung, das prozentuale Wachstum von Straßen oder die Wertsteigerung für das Land. Es ist auch möglich zu bestimmen, welche Muster sich im Straßennetz bilden können, wie zum Beispiel Radial- oder Rastermuster. Die prozedurale Generierung dieses Ansatzes ist nicht statisch, sondern geschieht im Laufe der Zeit. Das macht es dem Nutzer möglich, die Werte der Parameter auch während der Generierung zu ändern und damit den Prozess kontinuierlich zu beeinflussen.

Wie schon bei dem Ansatz von Parish und Müller [7], werden auch bei diesem Ansatz die Teilstücke als bestmögliches neues Segment aus einer Anzahl von Proben ausgewählt und danach der lokalen Situation angepasst. Der Unterschied dieser beiden Methoden liegt jedoch darin, dass die Generierung nicht jedem Teilstück die gleiche Bedeutung zumisst. Je näher ein neues Segment einem Wachstumszentrum ist, desto höher ist dessen zeitliche Bedeutung für die Stadt und hat eine höhere Chance generiert zu werden. Im nächsten Schritt wird Verkehrsaufkommen auf den bestehenden und geplanten Straßen

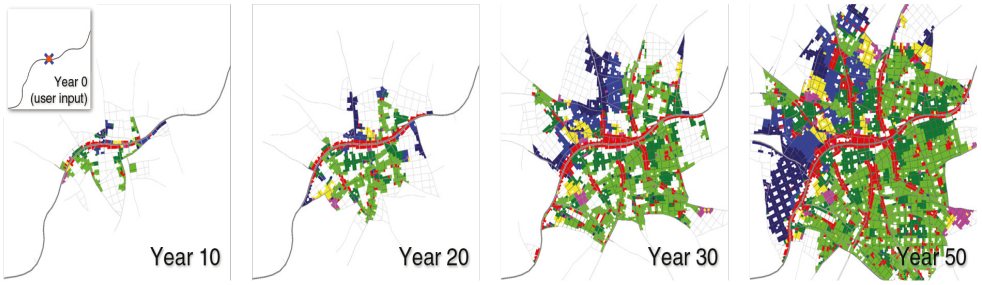


Abbildung 3.4: Beispiel einer Stadtgenerierung in mehreren Entwicklungsphasen - basierend auf ein Zentrum und eine Startstraße; geplante Straßen sind ausgegraut (grün: Wohnraum, blau: Industrie, rot: Kommerziell) (Weber et al. (2009), Fig. 4)

simuliert. Dafür werden zwei Punkte zufällig auf dem Netzwerk gewählt und der kürzeste Weg dazwischen gesucht (Pathfinding). Dabei wird aber nicht nur die Distanz, sondern auch die Fließgeschwindigkeit des Verkehrs bewertet. Das bedeutet, dass die Straßenführung keine oder nur geringe Richtungswechsel vornehmen soll. Der daraus resultierende Verkehrswert der neuen Straßenteilstücke wird überprüft. Liegt die Nutzungswahrscheinlichkeit unter einem definierten Schwellenwert, wird das Segment als nicht benötigt angesehen und damit nicht errichtet. Wird der Schwellenwert jedoch erreicht, entsteht eine neue Straße und die bestehenden Segmente werden dem Verkehrswert angepasst. Im folgenden Schritt wird entlang des Straßennetzes die Nutzung des Landes bestimmt. Genau wie beim Straßennetz kann das dazu führen, dass sich die Art der Nutzung und die Form ändern.

In diesem Ansatz kann also, wie beschrieben wurde, der zeitliche Faktor bezüglich des Wachstums der Stadt miteinbezogen werden. Des weiteren werden die Gebäude während der Generierung aktualisiert und können ihre Art der Nutzung verändern. Außerdem kann der Nutzer auch noch während der Erstellung der Stadt ihr weiteres Wachstum beeinflussen und nicht erst das Endresultat. Die zu steuernden Parameter sind aber nicht so intuitiv zu verstehen wie die Bearbeitungswerkzeuge im Ansatz von Chen et al. [13]. Damit man also qualitativ hochwertige Resultate erzielen kann, wird eine längere Zeitspanne für die Einarbeitung in den Ansatz benötigt.

Kapitel 4

Städtewachstum: Merkmale in der Realität

Städte sind hochfunktionale und visuell komplexe Systeme. Sie stellen den historischen, kulturellen, ökonomischen und sozialen Wandel über den Verlauf der Zeit dar [7]. Aus mehr oder weniger isolierten Einheiten (Urhütten und Sippenunterständen) sind verzweigte und miteinander verknüpfte Systeme entstanden. Diese Verknüpfung ist schon so weit gewachsen, dass sich weltumspannende Verkehrsnetze gebildet haben. Versucht man also die Entwicklung von Städten zu betrachten, muss man die Welt als System und nicht isoliert ansehen. Die folgenden fünf Punkte basieren auf der Grundlage von Füsser [14].

4.1 Altertum

Mit dem Aufkommen von Ackerbau und dem Handel von begehrten Gütern (lokal und regional) begann die allmähliche Sesshaftwerdung des Menschen. Ab dem neunten Jahrtausend vor Christus entwickelten sich erste Handelsbeziehungen zwischen den entstehenden Siedlungen. Aus dem Erfolg von Handel soll sich zu dieser Zeit bereits die erste Großsiedlung Jericho gebildet haben und eine etwaige Einwohnerzahl von 3000 erreicht worden sein. Die Ernährung dieser Einwohner war kaum durch die Oase alleine zu gewährleisten. Jericho war also vermutlich auf ein gut ausgebautes Straßen- und Handelsnetz angewiesen.

Mit dem Aufkommen der Seefahrt dehnte sich die Weltbevölkerung weiter aus und begann mit dem Besiedeln größerer Inseln. Der stetige Wachstum der Bevölkerung, kultureller Wandel und technischer Fortschritt legten

den Grundstein für einen florierenden Handel. Es konnten dadurch größere Städte entstehen und versorgt werden. Die Stadt Uruk am Euphrat zum Beispiel versorgte 40000 bis 50000 Einwohner und benötigte dafür weniger als die Hälfte der Bevölkerung. Weiterer organisatorischer und technischer Fortschritt ermöglicht die Entstehung erster großer Reiche (Ägypten, Babylonien). Babylon soll um 600 vor Christus 300000 bis 400000 Einwohner bei einer geschätzten Weltbevölkerung von 120 Millionen Menschen gehabt haben. Damit die Bedürfnisse der Menschen befriedigt werden konnten, mussten Wasserversorgungs- und Abwasserleitungssysteme errichtet und Städte zu Verwaltungszentren werden. Mit dem persischen und griechischen Reich begannen Städte, sich um Tempel und Marktplätze als Zentrum zu entwickeln und die Straßen passten sich den Landschaftsformen und der Steigung an. Neu gegründete Städte in Kolonien hingegen wurden nach einem Gittermuster errichtet, wie zum Beispiel die Städte Ephesus oder Milet.

Mit dem römischen Reich wandelte sich auch wieder das Bild der Städte. Die Römer errichteten ihre Städte axial orientiert und ordneten Stadtinhalte (Behörden, Tempel, Tore, etc.) und Straßen symmetrisch an. Zudem war ihr Straßennetz erstmals größtenteils befestigt und Innerorts mit Bordsteinen und erhöhten Gehwegen versehen. Dies kann noch heute zum Beispiel in Rom betrachtet werden.

4.2 Mittelalter

Mit dem Untergang des römischen Reiches werden viele ihrer Städte zerstört und/oder zerfallen. Der Handel und Verkehr brechen mehrheitlich zusammen. Erst mit dem Aufkommen der Feudalherrschaft kommt wieder ein vermehrter Handel über Grenzen und größere Distanzen auf. An wichtigen Verkehrsknotenpunkten wurden Siedlungen durch Kaufleute gegründet und diese entwickelten sich zu Städten. Die verkehrsgünstige Lage an zumeist natürlichen Häfen, Furten und Straßenkreuzungen generierten großen Reichtum, welcher durch Burgen und Kirchen geschützt wurde. Im Gegensatz zum römischen Straßennetz sind die neuen Straßen aber meistens in schlechtem Zustand und bei schlechter Witterung nur schwer zu befahren, weshalb die Wasserwege zu den wichtigsten Verkehrsverbindungen wurden. Ein weiteres Problem für Reisen zu Lande war der Zerfall der römischen Brücken. Römische Brücken waren meistens aus Stein gebaut und hatten breite Fahrspuren. Dieses Wissen der Baukunst ist jedoch im Laufe der Zeit verloren gegangen. Es war also nicht mehr möglich die Brücken in der selben Qualität zu reparieren. Neue Brücken konnten nur noch aus Holz errichtet werden. Dies war ein technologi-

scher Rückschritt, der den Handel negativ beeinflusste. Oft war nicht einmal mehr eine Reparatur der Brücken möglich, weshalb die Flussüberquerung meistens bei Furten geschah.

Auch sank die Einwohnerzahl in den Städten des Mittelalters im Vergleich zum römischen Reich. Während Rom zur Zeit der Zeitenwende (Beginn der christlichen Zeitrechnung) ungefähr 1 Million Einwohner oder Trier als Kaiserresidenz (um 280 nach Christus) etwa 60000 Einwohner hatte, konnten Städte im Mittelalter höchsten 30000 bis 200000 Einwohner (Weltstädte wie Byzanz, Paris, Venedig) erreichen. Zudem hat sich das städtische Straßennetz erneut gewandelt. Hauptstraßen gehen radial von einem Mittelpunkt aus in alle Himmelsrichtungen und Nebenstraßen sind rasterförmig und oft verwinkelt angeordnet.

4.3 Neuzeit

Mit dem Beginn der Renaissance im 15. Jahrhundert orientiert man sich wieder in der Städteplanung an der Antike. Schifffahrtskanäle werden neu angelegt und Straßenverbindungen werden verbessert. Der Straßen- und Brückenbau nimmt um 1800 im Rahmen der Industrialisierung einen großen Aufschwung und es wird ein neuer Straßenaufbau entwickelt. Mit der Erfindung der Dampfmaschine zu Beginn des 18. Jahrhundert wird der Grundstein für die Entwicklung des Schienennetzes im 19. Jahrhundert gelegt. Gemeinsam mit dem Liberalismus [15] und der Industriellen Revolution wandelt sich das Bild der Städte. Gründungen neuer Unternehmen und Fabriken in den Städten führten zu Landflucht und rasantem Bevölkerungswachstum in den Städten. Die neuen Einwohner siedelten sich vor allem entlang der Ausfallstraßen und Eisenbahnlinien an. Beim Bau dieser neuen Gebäude wurde kaum auf die Bedürfnisse der Menschen geachtet (z. B. wenig natürliches Licht in den Wohnungen oder fehlende sanitäre Anlagen). Ebenso wenig wurde Rücksicht auf die Umwelt genommen (z. B. werden Häuser im Überschwemmungsgebiet der Flüsse gebaut). Aus Mangel einer ordnenden Leitlinie im Städtebau, entstehen beengte und unhygienische Wohnquartiere, in denen sich Krankheiten und Epidemien leicht ausbreiten können. Ab der Mitte des 19. Jahrhunderts versucht man diese Probleme mit der Konstruktion erster Stadtparks und Werksiedlungen in den Griff zu bekommen. Es werden außerdem Kläranlagen, Krankenhäuser und Schulen errichtet. Nach dem ersten Weltkrieg beginnt man außerdem mit dem Bau gemeinnütziger und sozialer Wohnungen entlang von Straßenbahnen.

4.4 Gegenwart

Nach dem zweiten Weltkrieg begann der Anstieg der privaten Motorisierung und die Verkehrsplanung der Städte versuchten von vornherein, verstopfte Straßen zu vermeiden, indem breite Straßen mit zügiger Linienführung errichtet wurden. Oft wurde dabei auf die städtebaulichen und landschaftlichen Strukturen keine Rücksicht genommen. Im Nachhinein betrachtet muss gesagt werden, dass nur Städte, die im ursprünglich historischen Rahmen gelassen wurden, heute noch als schön und lebenswert empfunden werden. Die erhöhte private Motorisierung erlaubt es, dass die aus der Industrialisierung strahlenförmig gewachsenen Städte nun flächendeckend weiter wachsen. Die bestehenden freien Flächen werden ungeordnet bebaut und daraus resultiert eine Zersiedlung der Landschaft. Um den neuen Städten mehr Verkehrssicherheit zu bieten und für die Idee der Verkehrsberuhigung wurden Parkmöglichkeiten, Fußgängerzonen und Umgehungsstraßen angelegt. Wird die Verkehrsberuhigung gut in das bestehende Straßennetz eingebunden, kann die Lebensqualität von Geschäfts- und Wohngebieten erhöht werden.

4.5 Mögliche Merkmale von Städten in der Zukunft

Wie in den vier vorhergehenden Abschnitten aufgezeigt wurde, werden Siedlungsräume größer und deren Verkehrsnetze dichter und schneller, je weiter der technische und organisatorische Entwicklungsstand ist. Auch wurde der Grad der Arbeitsteilung immer weiter erhöht und wird vermutlich so weit gehen, dass Zwischenprodukte nur noch von Zulieferern bezogen werden. Der Güterverkehr wird also immer mehr dem Trend der kurzfristigen Dispositionen und weiten Transportentfernungen folgen. Wachsende Immobilienpreise werden immer mehr dazu führen, dass vor allem Verwaltung und Handel im Zentrum und Wohnräume vor allem am Stadtrand gefunden werden. Diese neuen Wohngebiete bilden sich wie neue Subzentren und führen zu polyzentrischen Städten mit überlagerten Mustern. Aber auch andere Szenarien sind denkbar. Zum Beispiel die Rückbesiedlung und Wohnraumerschaffung in Stadtzentren, verdichtetes Bauen zum Schutz der Natur oder die Planung von Grün- und Erholungszentren sind denkbar.

Kapitel 5

Welche Methoden werden für das Produkt verwendet?

Wie schon in Kapitel 2 angedeutet wurde, ist die Produktion eines neuen Videospiels ein komplexes Unterfangen. Neben der Spiele-Idee, welche neu erschaffen oder erweitert wird, muss auch eine Welt dafür gestaltet werden. Die Erstellung dieser Spiele-Welten ist zurzeit nur unter dem Einsatz vieler Ressourcen möglich. Wünschenswert ist es also, dass man ein Produkt hätte, das die Arbeitszeit und die benötigte Anzahl der Arbeitskraft reduzieren könnte. Die Qualität der Spiel-Welt muss aber konstant gehalten werden.

In dieser Arbeit wurde die Entscheidung getroffen, dass die Generierung so weit wie möglich ohne manuell geschaffene Grundlagen auskommen muss. Aus diesem Grund wird Perlin-Noise zur Generierung dieser Grundlagen verwendet. Damit jedoch keine Reduktion des natürlichen Erscheinungsbildes der Stadt erfolgt, zum Beispiel in Folge einer Bindung an ein Raster (siehe Ansatz von Olsson und Frank [16]), wird diese Technik in Kombination mit einem parametrischen L-System, wie in der Arbeit von Parish und Müller [7] vorgeschlagen, angewandt werden. Ausschlaggebend für die Entscheidung diesen Ansatz zu verfolgen war, dass diese Technik bereits von vielen Arbeiten ebenfalls aufgegriffen wurde (siehe Kapitel 3) und damit erfolgreich natürliche Stadtbilder geschaffen werden konnten.

In Kapitel 3 wurden bereits einige Ansätze aufgezeigt, die dem Nutzer ein höheres Maß an Kontrolle über die Generierung gewähren. Aus zeitlichen Gründen wurde jedoch in dieser Arbeit auf eine direkte Einflussname, wie in dem Ansatz von Chen et al. [12] formuliert, verzichtet. Das gleiche gilt für das zeitliche Wachstum der Stadt, wie in der Arbeit von Weber et al. [13] beschrieben.

Wie im Kapitel 4 aufgezeigt wurde, sind die Kernmerkmale von wachsenden, sich entwickelnden Städten in der Realität ein florierender Handel sowie technischer Fortschritt. Diese Merkmale führen dazu, dass das Verkehrsnetz ausgebaut und um neue Reisemöglichkeiten erweitert werden muss. Dieses gilt es nun in die Spiele-Welt zu übertragen. Wie schon in Kapitel 3 aufgezeigt wurde, wird dies auch in den bereits bestehenden Ansätzen versucht. Dementsprechend wird auch in dieser Arbeit zuerst ein Straßennetz entwickelt und daraufhin mit Gebäuden versehen. Weitere Kernmerkmale wie zum Beispiel die Wasserversorgung (z. B. Aquädukte) oder Marktplätze und alternative Reisevarianten wie zum Beispiel via Zug, werden aus zeitlichen Gründen in dieser Arbeit nicht berücksichtigt werden.

Kapitel 6

Theoretische Grundlagen

In diesem Kapitel wird die Theorie für die nachher in Kapitel 7 angewendeten Methoden erläutert. Weil das Ziel der Arbeit nicht die Aufarbeitung theoretischer Grundlagen für prozedurale Generierung ist, wird die Theorie nur kurz zusammengefasst und Probleme aufgezeigt.

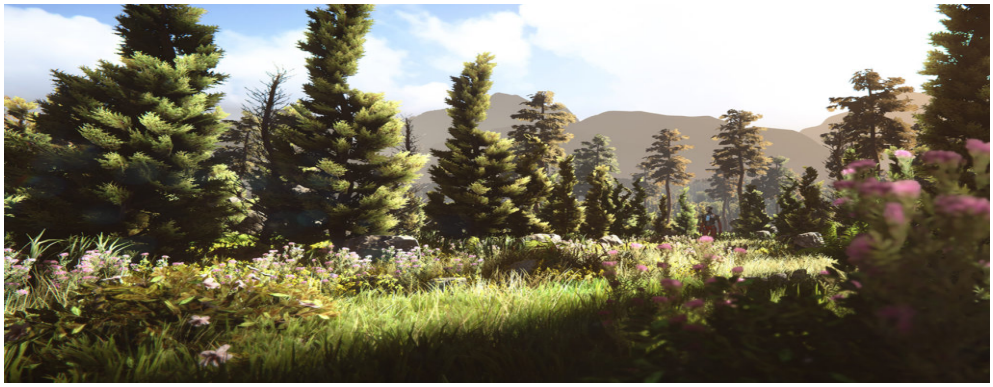


Abbildung 6.1: Mit SpeedTree erstellte Landschaft in Unity

Prozedurale Generierung ermöglicht es, aus einer endlichen Zahl von statischen Datenblöcken ein Vielfaches an Ergebnissen zu generieren. Diese Replikation umfasst sowohl geometrische Objekte, als auch Texturen und Effekte. Ein Beispiel dafür, ist die Erstellung eines Waldes mit prozedural generierten einzigartigen Bäumen auf der Grundlage von SpeedTree (für mehr Informationen siehe Website: `https://store.speedtree.com/#`). Dieses Beispiel verwendet ein L-System für die Generierung der Bäume, aber es könnten auch andere Methoden verwendet werden. Eine alternative Methode wären zum Beispiel Fraktale. Diese können für die Visualisierung von Schneeflocken und

anderen geometrischen Objekten verwendet werden. Eine weitere Möglichkeit der prozeduralen Generierung ist der Noise-Ansatz. Dieser wird häufig für die prozedurale Erstellung von Texturen und natürlicher Geländeformen [17] angewandt. Die wichtigsten Punkte prozeduraler Techniken sind jedoch gemäß Ebert et al. [18], dass die Funktionalität für den Nutzer abstrahiert wird (Nutzung ohne großes Fachwissen möglich), dass die Parameter der Generierung direkt beeinflussbar sind und dass sich das Produkt flexibel anwenden lässt, selbst über die Grenzen der Realität hinaus.

Großes Potential besteht in der prozeduralen Generierung von Inhalten für 3D Rendering Systeme vor allem darin, dass im Rahmen von Performanz-Optimierung das gleiche Objekt in unterschiedlicher Auflösung generiert werden kann. Diese Produkte mit unterschiedlicher Auflösung können dann abhängig vom Level of Detail (LOD) in Echtzeit gerendert werden [18]. Diese Art von Optimierung wird auch von der Demoszene genutzt. Diese ist schon seit den 1980er-Jahren aktiv und erstellt komplexe Szenen in Daten die gerade 2KB umfassen (für mehr Informationen siehe Website: `http://awards.scene.org/archive.php`). Mit dem Fortschritt der Hardware wurde es auch möglich, komplexe prozedurale Generierung wie zum Beispiel Volumen-Rendering in Echtzeit auf die GPU auszulagern [19], [20, Hart, Kapitel 3D Textures and Pixel Shaders].

6.1 Lindenmayer-System

Anders als Fraktale sind Lindenmayer-Systeme (L-Systeme) bei der Generierung von Inhalten flexibler in der Anwendung. Deshalb haben L-Systeme Fraktale auch in vielerlei Hinsicht in prozeduralen Generierung abgelöst. Ursprünglich kommen L-Systeme aus der Biologie und wurden von Lindenmayer [21] für die Beschreibung des Wachstums von Bakterien und simplen Organismen entwickelt.

Das Vorgehen von L-Systeme besteht darin, dass mittels iterativer Ausführung, sogenannter Produktionsregeln, sukzessive Teile eines zu Beginn simplen Objekts ersetzt werden. Obwohl L-Systeme entwickelt wurden, um Pflanzen und andere natürliche Strukturen zu definieren und zu visualisieren, gab es nicht nur im Bereich der Botanik, sondern auch im Bereich der prozeduralen Generierung signifikante Fortschritte. Produkte wie zum Beispiel von SpeedTree (für mehr Informationen siehe Website: `https://store.speedtree.com/#`) nutzen diese Technik, um ganze Landschaften mit detaillierter und diverser Flora generieren zu können.

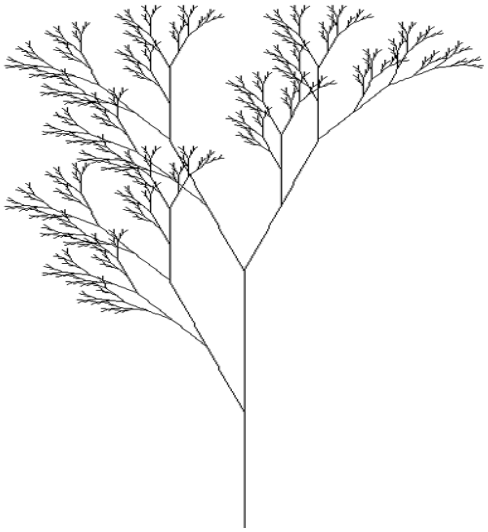


Abbildung 6.2: Lindenmayer-System

6.1.1 Einfache Lindenmayer-Systeme

Wie von Prusinkiewicz in [22, Kapitel 1] beschrieben, ist das zentrale Konzept von L-Systemen das Umschreiben eines Axioms. Kurz gesagt, das System beginnt mit einem simplen Startobjekt und generiert durch iterative Ausführung von Umschreibe-Regeln oder Methoden ein großes komplexes Produkt. In Abbildung 6.2 kann man die Visualisierung eines L-Systems mit $\omega = X$, $P = \{F \to FF; X \to F[+X]F[-X] + X\}$ und Winkel $= 20°$ sehen. Die Komponenten eines L-Systems sind:

- V (der Buchstabe) ist eine Menge an Symbolen die ersetzt werden können (Variablen)
- S ist eine Menge an Symbolen die nicht ersetzt werden können (Konstanten)
- ω (Start, Axiom oder Initiator) ist eine Reihe von Symbolen und Konstanten und beschreibt das Startobjekt
- P ist eine Menge an Regeln oder Methoden, welche die Art des Umschreibens definieren

L-Systeme können deterministisch (jedes Element von V besitzt genau eine Regel P) oder nicht-deterministisch sein. Genauso sind diese Systeme entweder kontextsensitiv oder kontextfrei.

6.1.2 Parametrische Lindenmayer-Systeme

Die von Hanan entwickelte Erweiterung von L-Systemen [23], ermöglicht die Eingliederung willkürlicher Funktionen. Das heißt, die Symbole aus der Menge V können numerische Parameter haben und mit regulären arithmetischen Operatoren und Variablen kombiniert werden. Daraus werden arithmetische und logische Ausdrücke gebildet, welche auch konditional sein beziehungsweise externe Funktionen aufrufen können. Diese Assoziierung zwischen nummerischen Parametern mit den Symbolen V erlaubt es eine einfache Quantifizierung der geometrischen Attribute des Modells vorzunehmen und daraus Regel für den Prozess der Generierung zu erstellen.

6.2 Noise: Gitterbasierter Ansatz

Noise wurde Mitte der 1980er Jahre als eine Alternative für die Texturierung von Objekten entwickelt. Einer der Gründe für die Entwicklung dieser Technik war, dass wegen der geringen Speicherkapazität der Computer den Objekten oft nur einfarbige Texturen gegeben werden konnte. Mit Hilfe von Noise konnte nun die reine Fläche mit Varianz (Farbe, Glanz, Verschiebung, etc.) versehen werden. Ein Lösungsansatz für die Generierung von Noise war, einen pseudo Zufallszahlengenerator zu verwenden. Das aus diesem Ansatz resultierende Ergebnis lieferte aber nicht das gewünschte Resultat. Man erzielte damit ein weißes Rauschen statt ein natürliches Erscheinungsbild zu haben. Um den Zufallswerten einen Zusammenhang geben zu können, müssen die Werte zuerst mittels einer Funktion geglättet werden. Der von Perlin [17] entwickelte Perlin-Noise verwendete dafür die Smoothstep-Funktion, aber auch andere Interpolationen wie zum Beispiel die lineare oder quadratische Interpolation können verwendet werden.

Die Visualisierung von Noise geschieht meistens in Form von Alpha-Maps, weshalb die Rückgabewerte von Noise meistens im Bereich $\big[0, 1\big]$ liegen. Ohne eine Neuzuordnung der Noise-Werte, liegen die Werte aber normalerweise im Bereich $\big[-1, 1\big]$, was zum Beispiel für Animationen wünschenswert sein kann.

Bei der Nutzung von Noise muss beachtet werden, dass er periodisch ist. Entweder ist die Wiederholung von Mustern ein gewünschter Effekt oder es muss ein großes Intervall für die Noise-Werte gewählt werden, damit die repetitiven Stellen erst beim Herauszoomen auftreten und wegen der geringeren Auflösung nicht auffallen. Wie von Perlin selbst vorgeschlagen und weil größere Intervalle kaum einen Mehrwert haben, werden heutzutage Intervalle von 2^8 oder 2^9 Werten verwendet. Historisch gesehen ist der Grund für ihren

binären Charakter, dass man sie mit dem Bit-Operator (&) bearbeiten konnte. Dieser war performanter als der Modulo-Operator (%). Heutzutage ist die Performanz der Operatoren vergleichsweise gut. Dennoch wird weiterhin der Bit-Operator bevorzugt, weil dieser Operator keine Probleme mit negativen Zahlen hat ($-10\%256 = -10, -10\&256 = 146$).

Noise ist n-dimensional generierbar und benötigt dafür lediglich entsprechenden Input. Würde man jedoch wie oben beschrieben die Werte in entsprechenden Arrays speichern, ist ab spätestens 3D der Vorteil der reduzierten Speichermenge hinfällig. Deshalb wird nur ein 1D-Array als Permutationstabelle erstellt, welches bei der Wertzuweisung als Hashtabelle verwendet wird. Ist die Permutationstabelle erstellt, kann der Noise mittels veränderter Werte von Frequenz und Amplitude zu unterschiedliche Resultate liefern. Diese Resultate können dann auch aufsummiert und dadurch lokale und globale Merkmale gebildet werden. Brownsches Rauschen ($1/f^2$-Rauschen, siehe Abbildung 6.5) kann zum Beispiel generiert werden, indem bei jeder weiteren Oktave die Frequenz verdoppelt und die Amplitude halbiert wird. Noise kann aber auch für die Generierung von Mustern verwendet werden. Diese sind zum Beispiel nützlich bei der Darstellung von natürlichen Objekten wie Feuer, Wolken oder Holz. In Abbildung 6.3 kann man die Darstellung einer Holztextur mittels Noise sehen. Für die Generierung dieser Textur wird der Noise-Wert mit dem auf die nächste ganze Zahl abgerundeten Noise-Wert subtrahiert ($noiseValue - (int)noiseValue$).

Eine simple Marmortextur kann beispielsweise erstellt werden, indem Noise auf eine Sinus-Funktion angewandt wird. Für eine realistischere Textur müssten jedoch noch weitere Farben eingebracht werden.

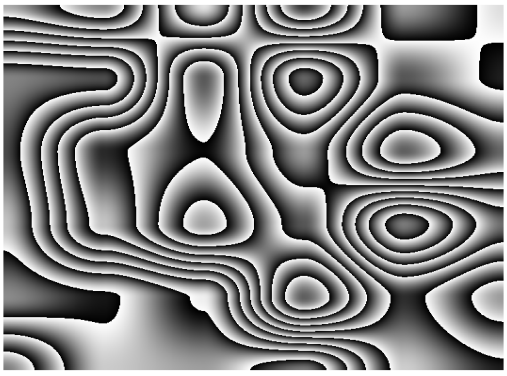


Abbildung 6.3: Holztextur erstellt mit Noise

6.2.1 Gradient-Noise: Perlin-Noise

Perlin-Noise [17] wurde von Perlin 1982 für den Film *Tron* entwickelt und brachte ihm den *Academy Award for Technical Achievement* 1997 ein. Diese Technik wurde entwickelt, um natürlicher erscheinende Texturen generieren zu können. Im Gegensatz zu Value-Noise werden bei Gradient-Noise aber nicht einfache Werte, sondern Gradienten (normalisierte Vektoren) interpoliert. Die Nutzung von Gradienten anstelle von einfachen Werten führt dazu, dass große Wertänderungen zwischen zwei Punkten nicht mehr von Relevanz sind.

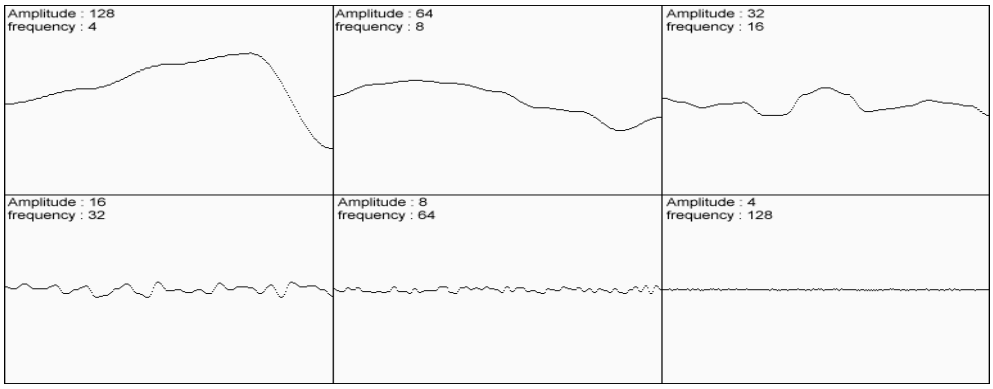


Abbildung 6.4: Sechs Noise Beispiele mit unterschiedlichen Frequenzen und Amplituden

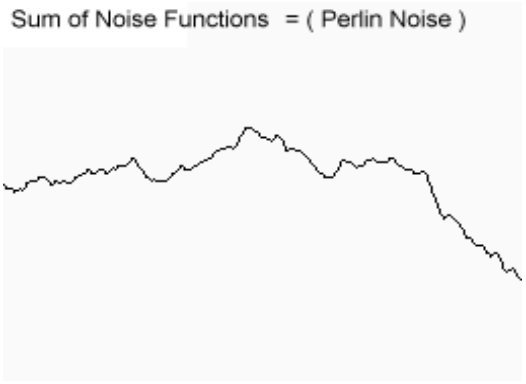


Abbildung 6.5: Brownsche Rauschen mit sechs Oktaven (Überlagerung der sechs Werte in Abbildung 6.4)

Betrachtet man den Noise-Graphen eines 1D-Noise, können Gradienten als

Tangenten der jeweiligen Schritte angesehen werden. Die Ausrichtung aufeinanderfolgender Gradienten ist jedoch irrelevant, denn der aus den Gradienten resultierende Graph wird eine natürlichere und homogenere Form haben, als jener Graph von Value-Noise. Der Grund für die natürlichere Form liegt darin, dass zwischen zwei aufeinanderfolgenden Werten nicht nur die Werte steigen oder fallen können. Zeigen beide Gradienten nach oben, entsteht ein „Hügel“ und zeigen beide nach unten, entsteht ein „Tal“. Selbst wenn beide Gradienten radikal verschieden (entgegengesetzte Ausrichtung) sind, besteht kein Problem, weil das daraus resultierende Teilstück eine „S-Form“ bildet (siehe Abbildung 6.6). Beim Value-Noise hingegen wäre daraus ein starkes Gefälle entstanden.

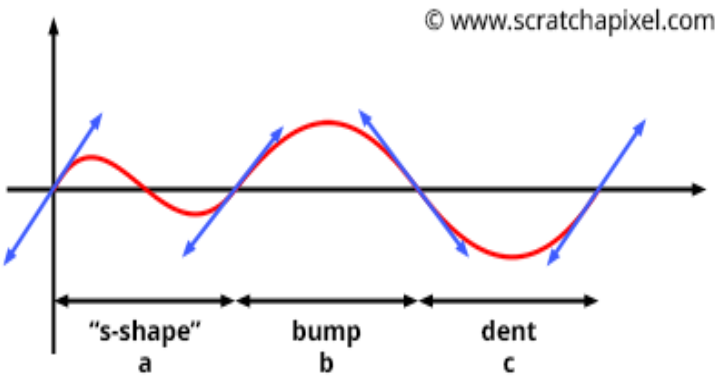


Abbildung 6.6: Mögliche Formen des Graphen durch die Nutzung von Gradienten

Ein Problem, das der Perlin-Noise Ansatz hat, ist die uniforme Verteilung im Raum. Werden die Gradienten mittels pseudo zufälliger Werte erstellt, sind gewisse Richtungen bevorzugt vorhanden. Man könnte nun leicht glauben, dass das zurückgreifen auf Polarkoordinaten das Problem für die Gradienten lösen würde. Doch selbst wenn die Werte im Koordinatensystem $\left[0, 2\pi\right]$ für ϕ und $\left[0, \pi\right]$ für θ uniform verteilt sind, muss festgestellt werden, dass sich die Werte dennoch im 3D-Raum an den Polen logischerweise häufen. Eine Möglichkeit zur Lösung dieses Problems besteht darin, mit Hilfe der Wahrscheinlichkeitsdichtefunktion und Verteilungsfunktion von ϕ und θ auf der Einheitssphäre, eine Verklumpung zu verhindern. Mit dem Wissen, dass $4\pi sr$ der Raumwinkel der Einheitssphäre ist und das Integral der Wahrscheinlichkeitsdichtefunktion $\int_0^{4\pi} p(\omega)d\omega = 1$ daraus folgt $p(\omega) = 1/4\pi$.

Wird nun die Funktion in Polarkoordinaten ausgedrückt, erhält man:

$$p(\phi,\theta)d\phi d\theta = p(\omega)d\omega = p(\omega)\sin(\theta)d\phi d\theta$$

daraus folgt:

$$p(\phi,\theta) = p(\omega)\sin(\theta) = \frac{\sin(\theta)}{4\pi}.$$

Die Wahrscheinlichkeitsdichtefunktion von θ ist also:

$$p(\theta) = \int_{\phi}^{2\pi} p(\phi,\theta)d\phi = \int_{\phi}^{2\pi} \frac{\sin(\theta)}{4\pi}d\phi = 2\pi * \frac{\sin(\theta)}{4\pi} = \frac{\sin(\theta)}{2}$$

und die Verteilungsfunktion:

$$\begin{aligned} Pr(X \leq \theta) = P(\theta) &= \int_0^{\theta} p(\theta)d\theta \\ &= \int_0^{\theta} \frac{\sin(\theta)}{2}d\theta \\ &= \left[\frac{-\cos(\theta)}{2}\right]_0^{\theta} \\ &= \left[\frac{-\cos(\theta)}{2} - \frac{-\cos(0)}{2}\right] \\ &= \frac{1}{2} - \frac{\cos(\theta)}{2}. \end{aligned}$$

Entsprechend erhält man die Wahrscheinlichkeitsdichtefunktion von ϕ durch:

$$p(\phi) = \frac{p(\phi,\theta)}{p(\theta)} = \frac{\frac{\sin(\theta)}{4\pi}}{\frac{\sin(\theta)}{2}} = \frac{1}{2\pi}$$

und die Verteilungsfunktion erhält man aus:

$$\begin{aligned} Pr(X \leq \phi) = P(\phi) &= \int_0^{\phi} \frac{1}{2\pi}d\phi \\ &= \frac{1}{2\pi}[\phi - 0] \\ &= \frac{\phi}{2\pi}. \end{aligned}$$

Sei nun y eine zufällige Zahl (uniform verteilt) erhält man durch die Invertierung der Verteilungsfunktionen von θ:

$$\begin{aligned} y &= \frac{1}{2} - \frac{\cos\theta}{2} \\ \Rightarrow \quad \frac{\cos(\theta)}{2} &= \frac{1}{2} - y \\ \Rightarrow \quad \cos(\theta) &= 1 - 2y \\ \Rightarrow \quad \theta &= \cos^{-1}(1-2y) = \cos^{-1}(2y-1) \end{aligned}$$

($1-2y$ und $2y-1$ liefern im Bereich $y \in \left[0,1\right]$ die selben Resultate) und ϕ:

$$\begin{aligned} y &= \frac{\phi}{2\pi} \\ \Rightarrow \phi &= 2\pi y. \end{aligned}$$

Diese Methode ist jedoch relativ schlecht in der Performanz, weshalb von Perlin selbst [24] als Lösung zwölf fixe Vektoren als Gradienten zu verwenden vorgeschlagen wurde. Diese ermöglichen eine uniforme Verteilung und reduzieren die durch die Permutationstabelle generierte Zufälligkeit nur in geringem Maße. Dabei werden die Vektoren (1,1,0), (-1,1,0), (0,-1,1) und (0,-1,-1) doppelt als mögliche Gradienten verwendet um 16 Werte zu haben und wieder binär Rechnen zu können. Das Hinzufügen dieser Vektoren führt aber zu keiner Verzerrung, weil: „*The final result has the same non-directional appearance as the original distribution but less clumping.*“ [24].

Neben dieser Verbesserung wurde im gleichen Artikel eine Gleichung fünften Grades ($6t^5 - 15t^4 + 10t^3$, *smootherstep function*) für die Interpolation vorgeschlagen. Der Vorteil gegenüber der bisher verwendeten Gleichung dritten Grades ($3t^2 - 2t^3$, *smoothstep function*) ist, dass diese neue Gleichung in der Ableitung zweiter Ordnung an den Stellen 0 und 1 den Wert 0 hat. An den Extremwerten handelt es sich also um Wendepunkte und nicht um Sattelpunkte. Dieses Problem hatte zuvor noch zu Diskontinuität geführt und Artefakte an den Zellgrenzen (Randbereich des zu bestimmenden Wertes) generiert.

Damit man nun zum Beispiel ein 2D-Noise-Wert berechnen kann, bestimmt man die Randpunkte unten links (x0, y0) und oben rechts (x1, y1) (siehe Abbildung 6.7). Mit Hilfe dieser Randpunkte können die anderen beiden Randpunkte ebenfalls bestimmt werden und sie werden jeweils mit einem Gradienten versehen. Damit man wieder von Vektoren zu einfachen Zahlenwerten kommt, wird das Skalarprodukt aus dem Gradienten und der Verbindungsgeraden zwischen Ansatzpunkt des Gradienten und dem Ort des zu

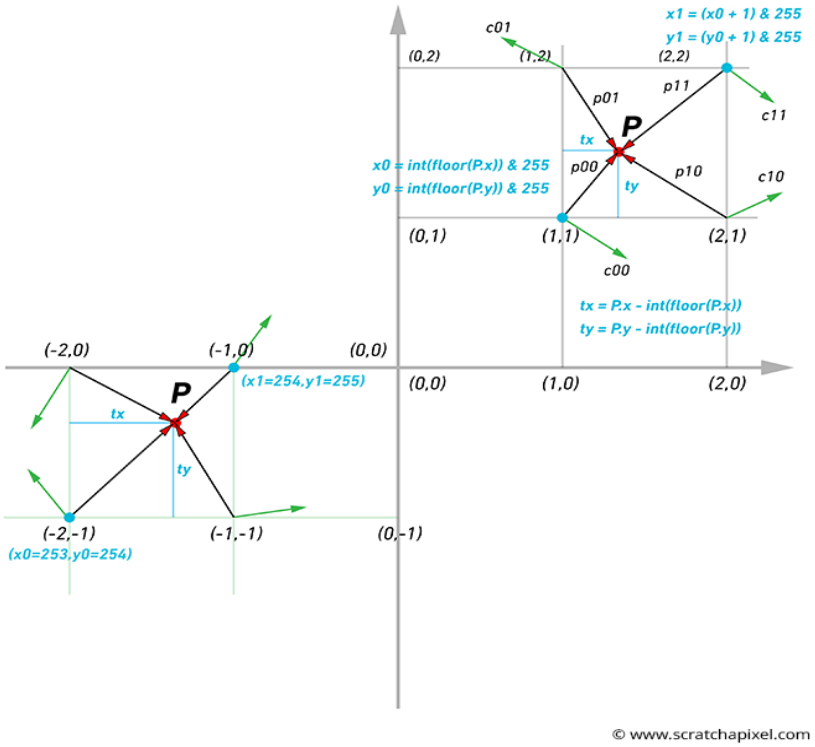


Abbildung 6.7: Darstellung der Berechnung von Perlin-Noise

berechnenden Punktes gebildet. Die daraus resultierenden Werte sind, wie oben beschrieben, standardisiert im Bereich $[-1, 1]$ und sollten zum Bereich $[0, 1]$ neu zugeordnet werden, falls man die Werte für Texturen verwenden möchte. Es ist außerdem möglich, mehrere Layer von Noise (Oktaven) zu einer Textur zusammenzusetzen. Damit können fraktal-ähnliche Details generiert werden (siehe Abbildung 6.4 und 6.5). Texturen welche mit dieser Methode generiert werden, können zum Beispiel für die Darstellung von Gebirgen verwendet werden. Dabei wird für die Erstellung der verschiedenen Layer nicht die Permutationstabelle verändert, sondern die verwendete Amplitude und die verwendete Frequenz werden modifiziert. Die Überlagerung dieser Layer resultiert darin, dass in der Textur nicht nur globale sondern auch lokale Details entstehen. Andere mögliche Anwendungsbereiche sind zum Beispiel die Generierung natürlicher Texturen, wie Marmor oder Holz.

Noise kann aber auch (siehe Abschnitt 6.2) 3D (volumetrisch) generiert werden. Wird eine solche Textur anstelle von 2D-Textur für ein Objekt verwendet, erweckt das 3D-Objekt ein natürlicheres Erscheinungsbild(siehe Abbildung 6.8). Es erscheint, als wäre das Objekt aus einem Block des Materials

hergestellt worden. Das ermöglicht es zum Beispiel die Maserung durch Marmor hindurch zu replizieren. Dieses höhere Maß an Realismus ist jedoch aufwendiger zu rendern und volumetrische Texturen haben aufwendige Speicher- und Lagerungs-Voraussetzungen [20, Kapitel 3D Textures and Pixel Shaders].

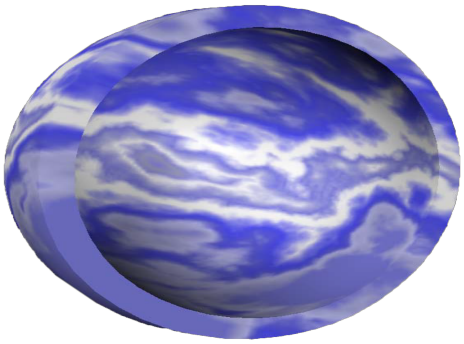


Abbildung 6.8: Volumetrische Marmor-Textur generiert durch Noise mit acht Oktaven

6.2.2 Simplex-Noise

Diese Methode der Noise Generierung wurde ebenfalls von Perlin [25, Kapitel 2] entwickelt und ist wie Perlin-Noise eine n-dimensionale Rauschfunktion. Der Vorteil dieser Art der Generierung liegt darin, dass es vor allem in höheren Dimensionen einen geringeren Rechenaufwand hat (Komplexität von $O(n^2)$ statt $O(2^n)$ bei n-Dimensionen). Neben der verbesserten Performanz des Algorithmus, sollte auch das Problem der Artefakte an den Zellgrenzen (Anisotropie) gelöst werden.

Die Verbesserung der Performanz gelingt durch eine geringere Rechenkomplexität und einer reduzierten Anzahl an Multiplikationen. Dies ist möglich, weil nicht mehr die Rasterpunkte interpoliert werden. Stattdessen wird der Raum in Simplizes (n-dimensionale Dreiecke) eingeteilt und darin interpoliert.

Dennoch bestehen auch bei dieser Methode Grenzen bei der Generierung von höheren Dimensionen. Bei höheren Dimensionen sind n-Kugeln mit n-Simplex-Ecken nicht dicht genug bemessen. Dadurch verringert sich der Träger der Funktion auf Null in großen Teilen des Raumes.

Kapitel 7

Methoden

In diesem Kapitel werden die in der prozeduralen Generierung für das Projekt angewandten Methoden für die jeweiligen Prozesse aufgezählt und erläutert.

7.1 Straßen

Für die Erschaffung des Straßennetzes wird mit Hilfe von Perlin-Noise die Grundlagen geschaffen. Die Bevölkerungsverteilung sowie die Höhenkarte wird mittels dieser Technik generiert. Ein weiterer Lösungsansatz für diese Generierungsfragen wäre der Simplex-Noise Ansatz. Es wurde jedoch entschieden, Perlin-Noise weiterhin zu verwenden, weil Perlin-Noise bereits im Programm implementiert war. Außerdem wurde Perlin-Noise bereits weiterentwickelt. Bedingt durch diese Weiterentwicklung ist der Unterschied des Rechenaufwandes in 2D zwischen den Ansätzen von Perlin-Noise und Simplex-Noise inexistent (siehe Kapitel 6.2.2).

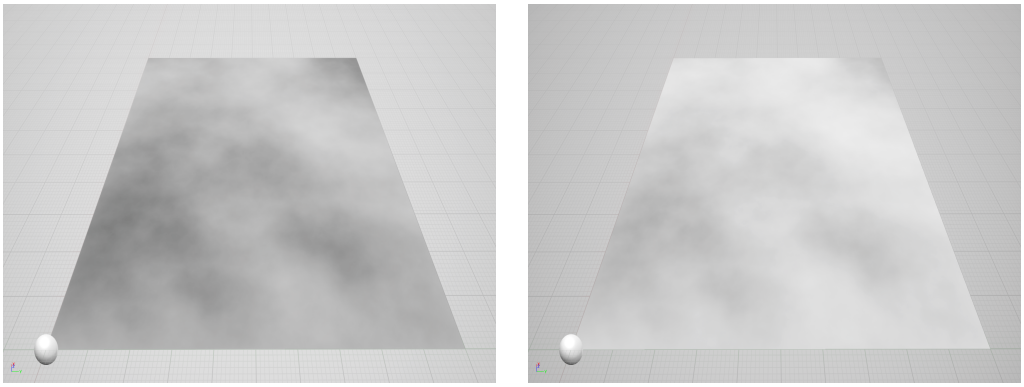
Als Startpunkte/Wachstumszentren für die Generierung werden die Bereiche mit der höchsten Bevölkerungsdichte verwendet. Diese werden ermittelt, indem der Noise im definierten Bereich nach Werten == 1 durchsucht wird. Damit aber garantiert mindestens ein Startpunkt gefunden wird, müssen die Werte im gewünschten Bereich normiert werden.

```
createNoise() {
  for (int x = 0; x < areaWidth; x++) {
    for (int y = 0; y < areaHeight; y++) {
      add noise value to array
      if (new value > highest value) {
        highest value = new value;
      }
    }
```

```
  }
  for (int i = 0; i < noise array size; i++) {
    noise[i] /= highest value;
  }
}
```

Listing 7.1: Pseudo Code für die Normierung von Noise

Diese Normierung gelingt, indem bei der Generierung des Noise der höchste Wert zwischengespeichert wird. Bevor die Generierung zu Ende ist, wird dann jeder Wert nochmals mit dem höchsten Wert dividiert (siehe Abbildung 7.1).



(a) Tatsächlicher Perlin-Noise (b) Perlin-Noise normiert

Abbildung 7.1: Textur mit Perlin-Noise generiert

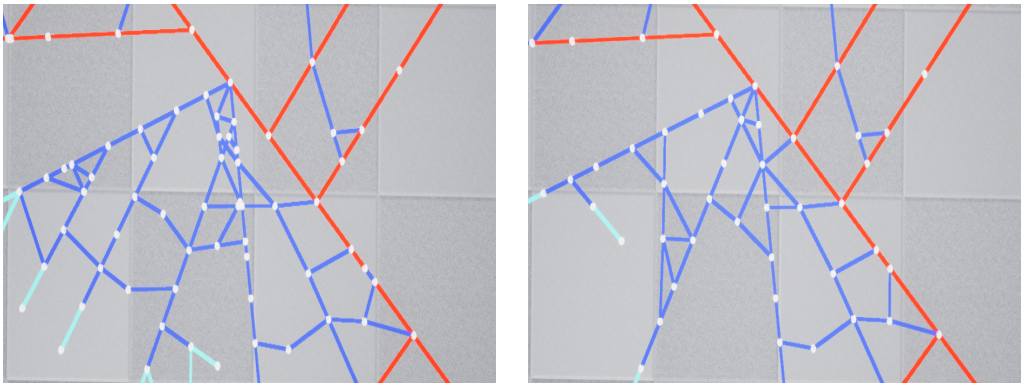
Weil dieser Noise mit Perlin-Noise generiert wird, haben Punkte die nahe beieinander liegen, auch ähnliche Werte. Dadurch besteht aber auch die Möglichkeit, dass sich nach der Normierung mehrere Werte == 1 zusammen liegen. Damit sich die Wachstumszentren dennoch nicht zu Clustern formen, wird ein vom Nutzer definierter Mindestabstand zwischen den möglichen Startpunkten gewahrt. Eine alternative Lösung für den Mangel an Startzentren ist die zufällige Generierung solcher Punkte. Nach der Bestimmung der Position der Wachstumszentren, wird in Abhängigkeit ihrer Position deren Ursprungszustand ermittelt.

```
createInitialSetup() {
  get startpoints from noise;
  calculate number of highways;
  calculate number of sub streets;
  if (number of startpoints > 1) {
    try to connect the startpoints with highways;
  }
  use remaining highways and sub streets and distribute them
     pseudo random around the startpoint;
```

}

Listing 7.2: Pseudo Code für die Generierung des Ursprungszustandes

Das bedeutet, dass die Anzahl der Haupt- und Nebenstraßen festgelegt wird, beziehungsweise deren Ausrichtung. Gebäude werden bei dieser Initialisierung noch keine errichtet. Der Grund dafür liegt darin, dass man dadurch etwas Platz im Zentrum der Stadt hat und dort zum Beispiel später ein markantes Gebäude oder ein Denkmal positionieren kann.



(a) Priorität auf neue Kreuzung gelegt (b) Priorität auf „Schnappen“ gelegt

Abbildung 7.2: Unterschiedliche Priorität bei der Erstellung des Straßennetzes

Für den Wachstum des Straßennetzes wird, wie von Parish und Müller [7] vorgeschlagen, zuerst in einem Bereich nach potentiellen Endpunkten gesucht und das bestmögliche Resultat zwischengespeichert. Die Bewertung der Endpunkte wird anhand globaler Ziele (Wachstumsrichtung, Bevölkerungsdichte, etc.) durchgeführt. Abhängig davon, wie der Nutzer die Parameter steuert, können dadurch von den globalen Zielen rasterförmige Muster, wie zum Beispiel in New York, oder ein verzweigtes Straßennetz, wie zum Beispiel in Berlin, generiert werden. Im nächsten Schritt werden diese neuen Segmente mit lokalen Bedingungen (bestehendes Straßennetz, Höhenkarte, etc.) verglichen. Für bestehende Konflikte werden Lösungen gesucht, indem die Ausrichtung und die Länge der Teilstücke angepasst werden. Neben diesen Veränderungen wird auch überprüft, ob der neue Abschnitt bereits bestehende Straßen kreuzt oder in der Nähe eines bestehenden Knotenpunktes endet. Auch hier wird die selbe Methodik wie in der Arbeit von Parish und Müller [7] angewandt. Das Programm versucht die Erschaffung neuer Knotenpunkte zu vermeiden (siehe Abbildung 7.2a). Dies geschieht, indem der Schwerpunkt auf das „Schnappen“ von Straßenenden zu bestehenden Knotenpunkten gelegt wird (siehe

Abbildung 7.2b). Eine möglichst geringe Anzahl an Kreuzungen entspricht der Realität und ein ruhiges, natürliches Stadtbild entsteht. Ist es jedoch nicht möglich eine angepasste Lösung zu finden, wird das Segment gelöscht und aus dem Straßennetz entfernt.

```
adjustSegment(bool reset = false) {
  if(current length < minimal length) {
    if(reset) resize segment to original size;
    rotate current direction to new direction;
    if(deviation to original direction > maximum deviation angle
       ) {
      adjustSegment(true);
    }
    else {
      update segment end to new location;
      adjustSegment(true);
    }
  else {
    if(value at segment end < minimal growth value) {
      shorten segment;
      adjustSegment();
    }
    else {
      adjustment went well;
    }
  }
}
```

Listing 7.3: Pseudo Code für die Anpassung des neuen Straßenteilstücks

Bei der Generierung von neuen Segmenten wird auch überprüft, ob der lokale Wert der Noise-Map den Grenzwert für die Entstehung einer Kreuzung überschreitet. Ist dies der Fall, wird mit Hilfe eines pseudo Zufallszahlengenerator entschieden, ob und wie viele beziehungsweise in welche Richtung neue Straßen sich abspalten. Dabei wird auch geprüft, um welche Art der Straße es sich handelt und deren Kategorie in Abhängigkeit zur Entfernung ihres Ursprungs. Dadurch wird verhindert, dass an Punkten weit vom Wachstumszentrum entfernt, zufällig ein unnötiges Cluster von Hauptstraßen entsteht.

Das Wachstum des Straßennetzes schreitet solange voran, bis ein neues Teilstück mit einem bestehenden Segment verbunden oder ein Grenzwert unterschritten wird. Zudem wird der Prozess beendet, wenn ein Segment die Grenzen überschreitet. Die Grenzwerte und Grenzen können vom Nutzer definiert werden.

7.2 Gebäude

Die Generierung von Gebäuden findet nach der Positionierung der neuen Straßensegmente statt. Falls sich der Straßenverlauf in ein bestehendes Gebäude hinein entwickelt hat, wird dieses aktualisiert und die Form des Hauses der Straße angepasst (siehe Abbildung 7.3). Durch das Update der Form des Gebäudes entstehen zwei konvexe Polygone. Im nächsten Schritt wird geprüft, ob entlang des neuen Straßenteilstückes links und rechts davon ein neues Gebäude errichtet werden kann. Kollidiert eines der geplanten Gebäude mit einem bereits existierenden Haus, wird es nicht errichtet. Als Grundlage wird ein Rechteck entlang des Segments verwendet und wie schon beim Update der bestehenden Gebäude anhand des Straßennetzes angepasst.

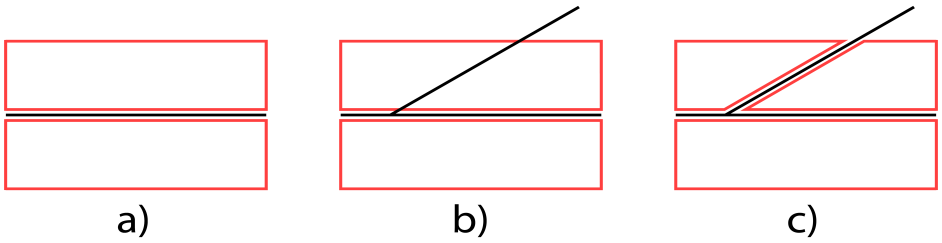


Abbildung 7.3: Vorgehen bei Gebäude-Update: a) Ausgangslage, b) neu konstruierte Straße kreuzt ein Gebäude, c) neue Gebäude nach Update

Sowohl bei der Aktualisierung der Form der Gebäude, als auch bei der ersten Konstruktion, wird die Grundfläche der Gebäude berechnet und mit einem vom Nutzer definierten Grenzwert verglichen. Fällt die Größe der Fläche unter den Grenzwert, wird das Gebäude als zu klein erachtet und nicht errichtet. Mit der Definition der Grenzwerte kann der Nutzer das Stadtbild stark beeinflussen und zum Beispiel die Bebauungsdichte verändern.

Die Bestimmung des Gebäudetyps wird mit Hilfe der Bevölkerungsdichte vorgenommen. Indem der lokale Wert der Bevölkerungsdichte mit vordefinierten Grenzwerten verglichen wird, kann entschieden werden, in welchem Bezirk sich das Gebäude befindet. Die Bezirke sind im Programm bereits vordefiniert (Park, Industrie, Kommerz und Wohnraum). Vorerst werden aber die Bezirke nur dafür verwendet, um die maximale Höhe der Gebäude zu definieren. Auf das Design, der in den Bezirken entstehenden Häuser, hat es bisher noch keinen Einfluss.

Nachdem die Generierung des Grundplans beendet ist, wird für jedes Gebäude ein Modell prozedural generiert. Das geschieht, indem der zuvor generierte Grundriss für die Berechnung der Vertices verwendet wird. Damit sich die Häuser aber nicht nur im Grundriss unterscheiden, wird bei der Erstellung des Modells zufällig ein Dachdesign (siehe Abbildung 7.4) zugewiesen. Mit dieser Technik wird die visuelle Diversität des Stadtbildes deutlich gesteigert. Die möglichen Formen sind zurzeit abhängig vom Grundriss und haben eine gleiche Wahrscheinlichkeit für die Generierung, unabhängig vom Bezirk oder der Höhe des Gebäudes.

Damit die endgültige Darstellung der Gebäude besser beleuchtet werden kann, wird bei der Berechnung der Vertices auch die entsprechende Vertex-Normale berechnet.

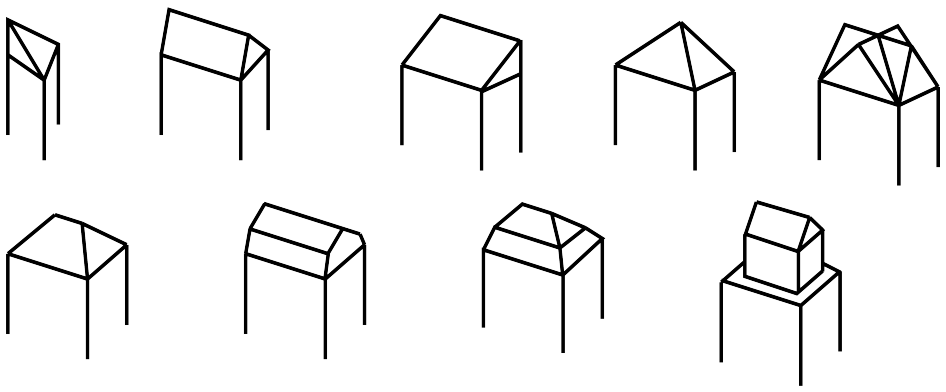


Abbildung 7.4: potentielle Dachformen

Kapitel 8

Quellcode

In diesem Teil der Arbeit werden die vier wichtigsten Funktionen für die Generierung von Städten dargestellt. Der gesamte Code kann auf dem beigelegten USB-Stick oder auf meiner Website (`fabiankopp.com/misc/bachelor_thesis.html`) heruntergeladen werden. Der Code ist in C++ für eine Nutzung in der Unreal Engine 4 verfasst.

```
void AStreetGenerator::Generate() {
  if (config->segmentLimit < qT_Street.GetAllObjects().Num())
      return;

  TArray<Segment> newActivSegments;
  for (int i = 0; i < activSegments.Num(); i++) {
    TArray<Segment> newBranches = GlobalGoals(activSegments[i]);
    for (int j = 0; j < newBranches.Num(); j++) {
      bool valid = LocalConstraints(newBranches[j]);
      if (valid) {
        TArray<Segment> localSegments;
        newBranches[j].furtherConnections.Sort([](const
            FVector2D& A, const FVector2D& B) {
          return A.Size() < B.Size();
        });
        for (int b = 0; b < newBranches[j].furtherConnections.
            Num(); b++) {
          if (b == 0) {
            Segment s = Segment(newBranches[j].GetStart(),
                newBranches[j].GetStart() + newBranches[j].
                furtherConnections[b], newBranches[j].isHighway,
                newBranches[j].category);
            if (s.length > 0.0f) localSegments.AddUnique(s);
          }
        else {
```

```
            Segment s = Segment(newBranches[j].GetStart() +
                newBranches[j].furtherConnections[b - 1],
                newBranches[j].GetStart() + newBranches[j].
                furtherConnections[b], newBranches[j].isHighway,
                newBranches[j].category);
            if (s.length > 0.0f) localSegments.AddUnique(s);
          }
          }
          if (localSegments.Num() > 0) {
          for (int l = 0; l < localSegments.Num(); l++) {
            if (config->segmentLimit < qT_Street.GetAllObjects().
                Num()) break;
            //check if new segment intersects with existing
                buildings and update them
            FVector2D rotDirection = localSegments[l].direction.
                GetRotated(-90.0f) * config->minBuildingLength;
            FVector2D mid = localSegments[l].GetEnd() -
                localSegments[l].GetStart();
            float height = GetNoiseValue(mid.X, mid.Y);
            UpdateBuildings(localSegments[l], rotDirection);
            qT_Street.AddSegment(localSegments[l]);
            ConstructBuildings(localSegments[l], rotDirection,
                height);
          }
          }
          else {
          if (config->segmentLimit < qT_Street.GetAllObjects().Num
              ()) break;

          if (!newBranches[j].isEndConnected) newActivSegments.Add
              (newBranches[j]);
          FVector2D rotDirection = newBranches[j].direction.
              GetRotated(-90.0f) * config->minBuildingLength;
          FVector2D mid = newBranches[j].GetEnd() - newBranches[j
              ].GetStart();
          float height = GetNoiseValue(mid.X, mid.Y);
          UpdateBuildings(newBranches[j], rotDirection);
          qT_Street.AddSegment(newBranches[j]);
          ConstructBuildings(newBranches[j], rotDirection, height)
              ;
          }
        }
      }
    }
    activSegments.Empty();
    if (!(config->segmentLimit < qT_Street.GetAllObjects().Num()))
        activSegments = newActivSegments;
}
```

```
void AStreetGenerator::CreateInitialSetup() {
  TArray<FVector2D> startPoints;
  if (config->startAtCenter) startPoints.Add(FVector2D(areaWidth
      / 2.0f, areaHeight / 2.0f));
  else startPoints = GetStartpoints();
  if (startPoints.Num() == 0) startPoints.Add(FVector2D(683,
     354));
  for (int i = 0; i < startPoints.Num(); i++) {
    TArray<Segment> newSegments;
    TArray<FVector2D> relativPoints;
    for (int p = 0; p < startPoints.Num(); p++) {
      relativPoints.Add(startPoints[p] - startPoints[i]);
    }
    relativPoints.Sort([](const FVector2D& A, const FVector2D& B
        ) {
      return A.Size() < B.Size();
    });
    int highways = (FMath::Cos((startPoints[i].X / areaWidth +
       startPoints[i].Y / areaHeight) / 2 * 360) + 1) * 3;
    int subs = (FMath::Sin((startPoints[i].X / areaWidth +
       startPoints[i].Y / areaHeight) / 2 * 360) + 1) * 5;
    FVector2D straightH = FVector2D(0, config->highway_minLength
        );
    FVector2D straightS = FVector2D(0, config->sub_minLength);
    TArray<bool> connectedPoints;
    connectedPoints.Init(false, startPoints.Num());
    connectedPoints[0] = true;
    for (int h = 0; h < highways; h++) {
      int cV = 0;
      FVector2D start = startPoints[i];
      for (int cp = 0; cp < connectedPoints.Num(); cp++) {
        if (connectedPoints[cp] == false) {
        connectedPoints[cp] = true;
        FVector2D dir = relativPoints[cp];
        dir.Normalize();
        FVector2D end = start + dir * config->highway_minLength;
        Segment newSegment = Segment(start, end, true);
        newSegments.Add(newSegment);
        qT_Street.AddSegment(newSegment);
        break;
        }
        else cV++;
      }
      if (cV == connectedPoints.Num()) {
        float angle = FMath::Acos(startPoints[i].X / areaWidth)
            * 180 / PI + float(h) / float(highways) * 360.0f; //
            angle in degree
        FVector2D end = start + straightH.GetRotated(angle);
        Segment newSegment = Segment(start, end, true);
```

```
        newSegments.Add(newSegment);
        qT_Street.AddSegment(newSegment);
      }
    }
    for (int s = 0; s < subs; s++) {
      float angle = FMath::Acos(startPoints[i].X / areaWidth) *
          180 / PI + float(s) / float(subs) * 360.0f; //angle in
          degree
      FVector2D start = startPoints[i];
      FVector2D end = start + straightS.GetRotated(angle);
      Segment newSegment = Segment(start, end, false);
      float valid = true;
      for (int l = 0; l < newSegments.Num(); l++) {
        float midAngle = FMath::Acos(FVector2D::DotProduct(
            newSegment.direction, newSegments[l].direction) / (
            newSegment.direction.Size() * newSegments[l].
            direction.Size())) * 180 / PI;
        if (midAngle > 90) midAngle = 180 - midAngle;
        if (midAngle < config->minIntersectionAngle) {
        valid = false;
        break;
        }
      }
      if (valid) {
        newSegments.Add(newSegment);
        qT_Street.AddSegment(newSegment);
      }
    }
    activSegments.Append(newSegments);
  }
}

TArray<Segment> AStreetGenerator::GlobalGoals(Segment segment) {
  TArray<Segment> newBranches;
  float newLength;
  if (segment.isHighway) newLength = FRandomStream(segment.
      GetEnd().X + segment.GetEnd().Y).FRandRange(config->
      highway_minLength, config->highway_maxLength);
  else newLength = FRandomStream(segment.GetEnd().X + segment.
      GetEnd().Y).FRandRange(config->sub_minLength, config->
      sub_maxLength);
  Segment newSegment = Segment(segment.GetEnd(), segment.GetEnd
      () + segment.direction * newLength, segment.isHighway,
      segment.category);

  float straightPop = GetNoiseValue(newSegment.GetEnd().X,
      newSegment.GetEnd().Y);
  if (segment.isHighway) {
    float maxPop = straightPop;
```

```
Segment bestSegment = newSegment;
for (int i = 0; i < config->highway_sampleAmount; i++) {
  //deviation of straigt segment
  float deviatedAngle = i * (2 * config->
      highway_maxDeviationAngle / (config->
      highway_sampleAmount - 1)) - config->
      highway_maxDeviationAngle;
  Segment curSegment = Segment(segment.GetEnd(), segment.
      GetEnd() + segment.direction.GetRotated(deviatedAngle)
      * newLength, segment.isHighway, segment.category);
  float curPop = GetNoiseValue(curSegment.GetEnd().X,
      curSegment.GetEnd().Y);
  if (curPop * FMath::Cos(deviatedAngle / config->
      highway_deviationWeight) > maxPop) {
    maxPop = curPop;
    bestSegment = curSegment;
  }
}
newBranches.Add(bestSegment);

float rotAngle = FRandomStream(segment.GetEnd().X + segment.
    GetEnd().Y).FRandRange(-1.0f, 1.0f) * (90.0f - config->
    minIntersectionAngle);

if (maxPop > config->highway_splitThreshold) {
  if (FRandomStream(bestSegment.GetEnd().X).FRand() < config
      ->highway_splitProbability) {
    FVector2D newDirection = segment.direction.GetRotated
        (-90.0f + rotAngle);
    float newLengthSplit = FRandomStream(segment.GetEnd().X
        + segment.GetEnd().Y).FRandRange(config->
        highway_minLength, config->highway_maxLength);
    if (segment.category >= maxCategory) newBranches.Add(
        Segment(segment.GetEnd(), segment.GetEnd() +
        newDirection * newLengthSplit, false));
    else newBranches.Add(Segment(segment.GetEnd(), segment.
        GetEnd() + newDirection * newLengthSplit, segment.
        isHighway, segment.category + 1));
  }
  else if (FRandomStream(bestSegment.GetEnd().Y).FRand() <
      config->highway_splitProbability) {
    FVector2D newDirection = segment.direction.GetRotated
        (90.0f + rotAngle);
    float newLengthSplit = FRandomStream(segment.GetEnd().X
        + segment.GetEnd().Y).FRandRange(config->
        highway_minLength, config->highway_maxLength);
    if (segment.category >= maxCategory) newBranches.Add(
        Segment(segment.GetEnd(), segment.GetEnd() +
        newDirection * newLengthSplit, false));
```

```
        else newBranches.Add(Segment(segment.GetEnd(), segment.
            GetEnd() + newDirection * newLengthSplit, segment.
            isHighway, segment.category + 1));
      }
      else if (FRandomStream(bestSegment.GetEnd().X +
          bestSegment.GetEnd().Y).FRand() < config->
          highway_splitProbability) {
        //split in both ways
        FVector2D newDirection = segment.direction.GetRotated
            (90.0f + rotAngle);
        float newLengthSplit = FRandomStream(segment.GetEnd().X
            + segment.GetEnd().Y).FRandRange(config->
            highway_minLength, config->highway_maxLength);
        if (segment.category >= maxCategory) {
        newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
            () + newDirection * newLengthSplit, false));
        newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
            () - newDirection * newLengthSplit, false));
        }
        else {
        newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
            () + newDirection * newLengthSplit, !segment.
            isHighway));
        newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
            () - newDirection * newLengthSplit, !segment.
            isHighway));
        }
      }
    }
  }
}
else {
  float maxPop = straightPop;
  Segment bestSegment = newSegment;
  float startHeight = GetHeight(segment.GetEnd().X, segment.
      GetEnd().Y);
  for (int i = 0; i < config->sub_sampleAmount; i++) {
    float deviatedAngle = i * (2 * config->
        sub_maxDeviationAngle / (config->sub_sampleAmount - 1))
         - config->sub_maxDeviationAngle;
    Segment curSegment = Segment(segment.GetEnd(), segment.
        GetEnd() + segment.direction.GetRotated(deviatedAngle)
        * newLength, segment.isHighway);
    float curPop = GetNoiseValue(curSegment.GetEnd().X,
        curSegment.GetEnd().Y);
    float heightDiv = fabsf(GetHeight(curSegment.GetEnd().X,
        curSegment.GetEnd().Y) - startHeight);
    if (curPop * FMath::Cos(deviatedAngle / config->
        sub_deviationWeight) / (heightDiv * config->
        sub_heightWeight + 1) > maxPop) {
```

```
    maxPop = curPop;
    bestSegment = curSegment;
  }
}
newBranches.Add(bestSegment);

float rotAngle = FRandomStream(segment.GetEnd().X + segment.
    GetEnd().Y).FRandRange(-1.0f, 1.0f) * (90.0f - config->
    minIntersectionAngle);

if (maxPop > config->sub_splitThreshold) {
  if (FRandomStream(bestSegment.GetEnd().X).FRand() < config
      ->sub_splitProbability) {
    FVector2D newDirection = segment.direction.GetRotated
        (-90.0f + rotAngle);
    float newLengthSplit = FRandomStream(segment.GetEnd().X
        + segment.GetEnd().Y).FRandRange(config->
        sub_minLength, config->sub_maxLength);
    newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
        () + newDirection * newLengthSplit, segment.isHighway
        ));
  }
  else if (FRandomStream(bestSegment.GetEnd().Y).FRand() <
      config->sub_splitProbability) {
    FVector2D newDirection = segment.direction.GetRotated
        (90.0f + rotAngle);
    float newLengthSplit = FRandomStream(segment.GetEnd().X
        + segment.GetEnd().Y).FRandRange(config->
        sub_minLength, config->sub_maxLength);
    newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
        () + newDirection * newLengthSplit, segment.isHighway
        ));
  }
  else if (FRandomStream(bestSegment.GetEnd().X +
      bestSegment.GetEnd().Y).FRand() < config->
      sub_splitProbability / 2.0f) {
    //split in both ways
    FVector2D newDirection = segment.direction.GetRotated
        (90.0f + rotAngle);
    float newLengthSplit = FRandomStream(segment.GetEnd().X
        + segment.GetEnd().Y).FRandRange(config->
        sub_minLength, config->sub_maxLength);
    newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
        () + newDirection * newLengthSplit, segment.isHighway
        ));
    newBranches.Add(Segment(segment.GetEnd(), segment.GetEnd
        () - newDirection * newLengthSplit, segment.isHighway
        ));
  }
```

```
    }
  }

  return newBranches;
}

bool AStreetGenerator::LocalConstraints(Segment &segment) {
  if (config->ignoreConstraints) return true;

  if (segment.GetStart().X < 0.0f || segment.GetStart().Y < 0.0f
      || segment.GetStart().X > areaWidth || segment.GetStart().
     Y > areaHeight) return false;
  //check if segment need adjustments or cant be adjusted to fit
      at all
  if (!AdjustSegment(segment, true)) return false;

  FVector2D tempEnd = segment.GetEnd() + segment.direction *
     config->snapDistance;
  Segment temp = Segment(segment.GetEnd(), tempEnd, segment.
     isHighway);
  TArray<Segment> newSegments;
  TArray<Segment> allStreets = qT_Street.GetAllObjects();
  TArray<Segment> other = qT_Street.GetObjectsInNode(Segment(
     segment.GetStart(), tempEnd, segment.isHighway));
  for (int i = 0; i < other.Num(); i++) {
    allStreets.Remove(other[i]);
    if (!segment.isEndConnected) {
      // snap to crossing within radius check
      if ((segment.GetEnd() - other[i].GetEnd()).Size() <=
          config->snapDistance && segment.GetEnd() != other[i].
          GetEnd()) {
        segment.GetEnd() = other[i].GetEnd();
        segment.furtherConnections.Add(other[i].GetEnd() -
            segment.GetStart());
        segment.isEndConnected = true;
      }
      else if ((segment.GetEnd() - other[i].GetStart()).Size()
         <= config->snapDistance && segment.GetEnd() != other[i
          ].GetStart()) {
        segment.GetEnd() = other[i].GetStart();
        segment.furtherConnections.Add(other[i].GetStart() -
            segment.GetStart());
        segment.isEndConnected = true;
      }

      //  extend road and create new intersection within snap
          distance
      else if (temp.CheckForIntersection(other[i])) {
```

```
    if (temp.GetStart() == other[i].GetStart() || temp.
        GetStart() == other[i].GetEnd()) continue;
    else {
    float midAngle = FMath::Acos(FVector2D::DotProduct(
        segment.direction, other[i].direction)) * 180 / PI;
    if (midAngle > 90) midAngle = 180 - midAngle;
    if (midAngle < config->minIntersectionAngle) {
      if (segment.furtherConnections.Num() > 0) continue;
      else return false;
    }
    }
    FVector2D intersection = temp.GetIntersectionLocation(
        other[i]);
    // might be redundant because theese points should have
        been checked with crossing snapping
    if (!(other[i].GetStart() == intersection || other[i].
        GetEnd() == intersection)) newSegments.Add(
        SplitSegment(other[i], intersection));
    segment.GetEnd() = intersection;
    segment.furtherConnections.AddUnique(intersection -
        segment.GetStart());
    segment.isEndConnected = true;
  }
}

// intersection check
if (segment.CheckForIntersection(other[i])) {
  // dont check between origin and its new segments
  if (segment.GetStart() == other[i].GetEnd() || segment.
      GetEnd() == other[i].GetStart() || segment.GetEnd() ==
      other[i].GetEnd()) continue;
  // check angle if both segments have same origin
  if (segment.GetStart() == other[i].GetStart()) {
    float midAngle = FMath::Acos(FVector2D::DotProduct(
        segment.direction, other[i].direction)) * 180 / PI;
    if (midAngle > 90) midAngle = 180 - midAngle;
    if (midAngle < config->minIntersectionAngle) {
    if (segment.furtherConnections.Num() > 0) continue;
    else return false;
    }
    else continue;
  }

  FVector2D intersection = segment.GetIntersectionLocation(
      other[i]);
  if (!(other[i].GetStart() == intersection || other[i].
      GetEnd() == intersection)) newSegments.Add(SplitSegment
      (other[i], intersection));
  if (segment.isEndConnected) {
```

```
        segment.furtherConnections.Insert(intersection - segment
            .GetStart(), segment.furtherConnections.Num() - 1);
      }
      else segment.furtherConnections.AddUnique(intersection -
          segment.GetStart());
      //to prevent snapping to the same element
      continue;
    }

  }
  for (int s = 0; s < other.Num(); s++) {
    allStreets.AddUnique(other[s]);
  }
  allStreets.Append(newSegments);
  qT_Street.Clear();
  for (int a = 0; a < allStreets.Num(); a++) {
    qT_Street.AddSegment(allStreets[a]);
  }
  return true;
}
```

Kapitel 9

Fazit

Das primäre Ziel dieser Bachelorarbeit, ein Programm zur prozeduralen Generierung einer Stadt zu erstellen, konnte erfolgreich erfüllt werden (siehe USB-Stick oder Website `fabiankopp.com/misc/bachelor_thesis.html`). Das sekundäre Ziel, dass das Resultat ein natürliches Erscheinungsbild betreffend des Straßennetzes und der Gebäude abbilden muss, konnte ebenfalls viel versprechend umgesetzt werden. In den folgenden Bildern sind die Gebäude und Straßen farblich gekennzeichnet. Hauptstraßen sind Rot und Nebenstraßen sind Blau eingefärbt. Bei den Gebäuden sind Wohnungen Blau, kommerzielle Gebäude Gelb, industrielle Gebäude Rot und Parks sind Grün eingefärbt.

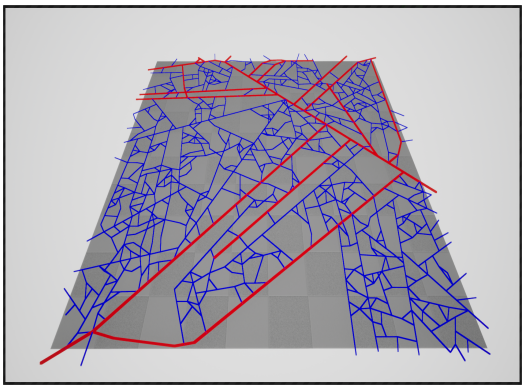
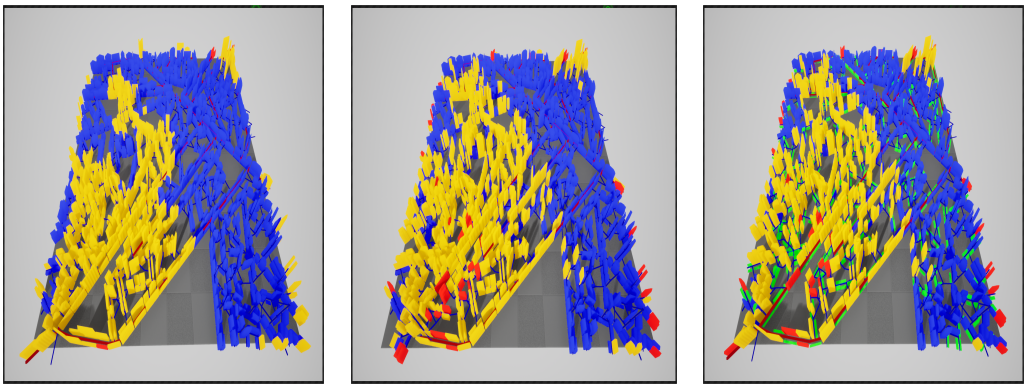


Abbildung 9.1: Organisches Straßennetz

Die Funktionalität des Programms wurde bereits in Kapitel 7 beschrieben. Mit Hilfe des Programms lassen sich verschiedene Straßenmuster erstellen. In Abbildung 9.1 ist zum Beispiel ein organisches Straßennetz zu sehen, das mit diesem Programm generiert wurde und als Grundlage für die Städte in

der Abbildung 9.2 verwendet wird. In allen drei Städten wird der Bezirk der Gebäude anhand des Perlin-Noise Wertes in der Mitte des Hauses bestimmt. Die Grenzen der Bezirke unterscheiden sich aber. In Abbildung 9.2a ist die Grenze zwischen den Bezirken klar definiert. In den anderen beiden Abbildungen wird stattdessen die Verteilung unscharf vorgenommen. Das führt zu einem etwas natürlicheren Erscheinungsbild der Stadt. In Abbildung 9.2c besteht zusätzlich noch das Potential, dass sich an jedem Ort ein Park anstatt eines Gebäudes entstehen kann.

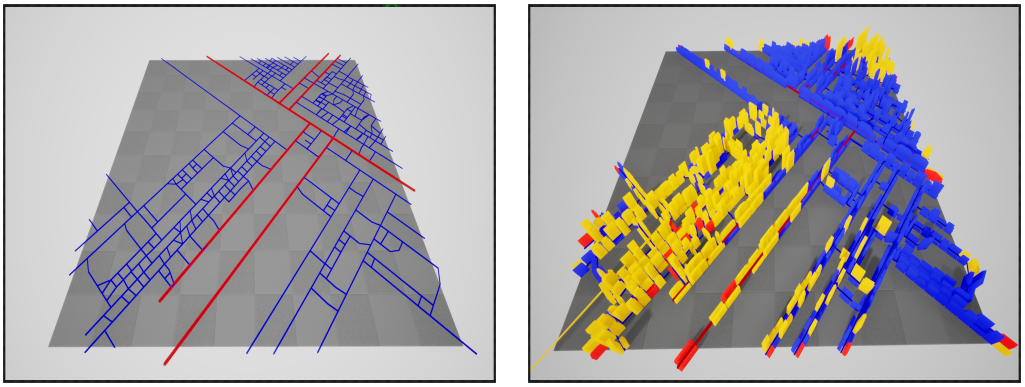


(a) Harte Grenzen (b) Unscharfe Grenzen (c) Stadt mit Parks

Abbildung 9.2: Unterschiedliche Bestimmung der Gebäude-Bezirke

Ein weiteres Ziel der Bachelorarbeit war es, die Benutzerfreundlichkeit insofern zu steigern, dass die Komplexität der prozeduralen Generierung auf simple Parameter reduziert werden kann. Auch in diesem Bereich konnten Erfolge erzielt werden. Durch die Veränderung der Parameter zum Beispiel, kann eine Stadt mit einem Straßennetz im Rastermuster entstehen (siehe Abbildung 9.3a). Wie man im Vergleich zwischen den Städten in den Abbildungen 9.2b und 9.3b sehen kann, verändert sich der Charakter des Erscheinungsbildes der Stadt massiv, wenn das Straßenmuster verändert wird. Dank der direkten Beeinflussbarkeit der Parameter, lässt sich das Produkt also flexibel anwenden, selbst über die Grenzen der Realität hinaus. In der Arbeit konnte sogar ein Stretch Goal erreicht werden. Die Gebäude und Straßen werden nicht nur als simple Grafik dargestellt. Stattdessen kann das Programm diese als 3D-Objekte generieren und darstellen.

In der ursprünglichen Fragestellung ist ebenfalls formuliert gewesen, dass das ausgearbeitete Programm bei der Entwicklung von Videospielen helfen soll, die Personalressourcen zu reduzieren. Ob dies gelungen ist, kann noch nicht abschließend bewertet werden, weil Zahlen und Daten bezüglich der Erschaffung einer Stadt auf herkömmliche Art und Weise nicht in Erfahrung



(a) Raster-Straßennetz (b) Endresultat

Abbildung 9.3: Stadt generiert mit anderen Parametern

gebracht werden konnten. Auf jeden Fall hat die Arbeit aufgezeigt, dass es auch mit kleinsten Teams möglich ist, ein Programm für die prozedurale Generierung von Städten in kürzester Zeit zu entwickeln. Konkret konnte dieses Programm selbstständig in ca. zwei Wochen erarbeitet werden. Aus diesem geringen Aufwand heraus, wird die These aufgestellt, dass das Programm selbst zu einem Mehrwert für die Entwicklung von Videospielen führen wird. Außerdem wird vermutet, dass bei Videospielen die prozedurale Generierung in der Zukunft vermehrt zum Einsatz kommen wird. Deshalb kann diese Arbeit als guter Ausgangspunkt für weitere Entwicklung gesehen werden.

Kapitel 10

Ausblick

Wie mit dieser Arbeit aufgezeigt werden konnte, ist es möglich, in relativ kurzer Zeit ein Programm zu entwickeln, das eine Stadt generieren kann. Die benötigte Zeit für die Entwicklung dieses Programms, ist im Vergleich zu der Zeit, die für die manuelle Erstellung von Städten benötigt wird, relativ gering. Es ist also durchaus für Entwickler von Videospielen von Interesse, dieses Prinzip zu verfolgen, denn es ermöglicht die Arbeitszeit und Personalkosten zu reduzieren. Besonders für Entwickler mit kleinen Teams ist dies interessant, weil ein solches Programm auch mit geringen Personalressourcen entwickelt werden kann.

In den folgenden drei Punkten werden mögliche Richtungen für weitere Entwicklung in der Zukunft aufgezeigt.

10.1 Visualisierung

Dieses Projekt ermöglicht es, ein Straßennetz und angegliederte Gebäude zu generieren. Die Straßen und Häuser sind aber noch ohne Textur und unterscheiden sich lediglich in ihrer Form. Damit man sich nicht nur durch eine kahle Stadt bewegen kann, müssen die Gebäude und Straßensegmente texturiert und/oder mehr visuelle Ankerpunkte haben. Ansätze für eine Umsetzung kann in der Arbeit von Parish und Müller [7, Kapitel 4.2 und 4.3] oder in der Arbeit von Wonka et al. [11] gefunden werden.

Ein weiterer Punkt um ein höheres Maß an Realismus zu erzielen, wird in der Arbeit von Olsson und Frank [16] beschrieben. Sie schlagen unter anderem vor, dass die zufällige Zuweisung der Bezirke hilft, das Erscheinungsbild der Stadt natürlicher zu gestalten.

Eine weitere Möglichkeit die Vielfalt der Gebäude in der Stadt zu erhöhen, ist eine zufällige Unterteilung der generierten Gebäude vorzunehmen. Außerdem könnten die Polygone zu einem neuen Polygon vereint werden. Bisher wurde die Konstruktion des neuen Gebäudes bei einem Konflikt abgebrochen, nun würde der Grundriss erweitert werden.

10.2 Erweiterung der Parameter

Es wird empfohlen, die Menge der zu betrachtenden Parameter und die möglichen Inputs zu erweitern. Dazu gehört zum Beispiel die Erweiterung der genutzten Grundlagen um Land-Wasser-Karten und um Vegetationskarten. Gewässer sind nicht als reine Grenzen zu betrachten, sondern es sollte für Hauptstraßen die Möglichkeit bestehen, Brücken über die Gewässer errichten zu können. Das Konzept von Brücken sollte jedoch nicht nur an Gewässern, sondern auch bei möglichen Straßenkreuzungen Anwendung finden. Ein mögliches Anwendungsbeispiel wäre, wenn eine sekundäre Straße höherer Kategorie oder eine tertiäre Straße mit einer primären Straße kreuzen würde.

Es wird auch empfohlen, dass das Projekt um mehrere Parameter für die Bevölkerung erweitert wird. Damit kann zum Beispiel ein Limit für die Generierung der Gebäude gesetzt oder die potentielle Höhe der Gebäude variabel bestimmt werden.

10.3 Erweiterung der Methoden

Damit die Möglichkeit besteht, Städte verschiedener Epochen zu generieren oder eine größere Anzahl an Straßenmustern zu erreichen, sollten die bestehenden Methoden zum Beispiel um die Generierung von Zuglinien und Autobahnen erweitert werden. Da das Projekt bereits eine Berücksichtigung der Höhe implementiert hat, kann auch darüber nachgedacht werden, das Projekt um eine Methode zur Generierung von Tunneln zu erweitern.

Eine weitere mögliche Erweiterung wäre es, tertiäre Straßen mit einzubeziehen und/oder, wie von Lechner und Wilensky [8] vorgeschlagen, den Straßen einen Einflussbereich zu geben. Dadurch wäre es auch möglich Gebäude ohne direkte Anbindung an das Straßennetz zu generieren und die Städte dichter zu besiedeln.

Aus zeitlichen Gründen wurde für diese Arbeit nur das Raster- und das orga-

nische Muster implementiert. Wie man in den bereits bestehenden Beispielen sehen kann, ist das fehlende Radialmuster auch oft anzutreffen. Daher sollte das Projekt unbedingt um diese Methode noch erweitert werden.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Berlin, den 14. Februar 2019, ...
Fabian Oyvin Kopp

Abbildungsverzeichnis

2.1 Prozedural generierte Inhalte 4
a Prozedural generiertes Level in *Rogue*; Quelle: `https://de.wikipedia.org/wiki/Rogue_(Computerspiel)`; abgerufen am 06.02.2019 4
b Prozedural generiertes Sonnensystem in *Elite*; Quelle: `https://www.c64-wiki.de/wiki/Elite`; abgerufen am 06.02.2019 4
2.2 Prozedural generierte Inhalte 4
a Fraktale Berge in *Rescue on Fractalus*; Quelle: `https://www.c64-wiki.de/wiki/Rescue_on_Fractalus!`; abgerufen am 06.02.2019 4
b Screenshot aus *.kkrieger*; Quelle: `https://www.researchgate.net/figure/Screenshot-from-the-fully-procedural-game-kkrieger-th13a-The-graphics-and-details_fig2_282334316`; abgerufen am 06.02.2019 4

3.1 Prozedurale Straßennetzgenerierung auf Grundlagen des Stadtteils Manhatten; Quelle: Parish und Müller, *Procedural modeling of cities*, 2001, Figure 9 7
3.2 Beispiele unterschiedlicher Stadtstrukturen basierend auf unterschiedlichen Straßennetzwerken; Quelle: Lechner und Wilensky, *Procedural city modeling*, 2003, Figure 5 8
3.3 Generiertes Straßennetz mit Hilfe des Tensorfeld und manueller Bearbeitung auf Grundlage des Benue Fluss in Nigeria; Quelle: Chen et al., *Interactive procedural street modeling*, 2008, Figure 1 11
3.4 Beispiel einer Stadtgenerierung in mehreren Entwicklungsphasen; Quelle: Weber et al., *Interactive geometric simulation of 4d cities*, 2009, Figure 4 12

6.1 Mit SpeedTree erstellte Landschaft in Unity; Quelle: https://store.speedtree.com/store/speedtree-modeler-for-unity/; abgerufen am 06.02.2019 19
6.2 Lindenmayer-System; Quelle: http://paulbourke.net/fractals/lsys/; abgerufen am 06.02.2019 21
6.3 Holztextur erstellt mit Noise; Quelle: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples; abgerufen am 06.02.2019 23
6.4 Sechs Noise Beispiele mit unterschiedlichen Frequenzen und Amplituden; Quelle: http://flafla2.github.io/2014/08/09/perlinnoise.html; abgerufen am 06.02.2019 24
6.5 Brownsches Rauschen mit sechs Oktaven; Quelle: http://flafla2.github.io/2014/08/09/perlinnoise.html; abgerufen am 06.02.2019 24
6.6 Mögliche Formen des Graphen durch die Nutzung von Gradienten; Quelle: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise; abgerufen am 06.02.2019 25
6.7 Darstellung der Berechnung von Perlin-Noise; Quelle: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise; abgerufen am 06.02.2019 28
6.8 Volumetrische Marmor-Textur; Quelle: Hart, *3D Textures and Pixel Shaders*, 2002, Figure 3 29

7.1 Textur mit Perlin-Noise generiert 31
a Tatsächlicher Perlin-Noise; Eigene Darstellung 31
b Perlin-Noise normiert; Eigene Darstellung 31
7.2 Unterschiedliche Priorität bei der Erstellung des Straßennetzes 32
a Priorität auf neue Kreuzung gelegt; Eigene Darstellung 32
b Priorität auf „Schnappen“ gelegt; Eigene Darstellung . 32
7.3 Vorgehen bei Gebäude-Update; Eigene Darstellung 34
7.4 potentielle Dachformen; Eigene Darstellung 35

9.1 Organisches Straßennetz; Eigene Darstellung 46
9.2 Unterschiedliche Bestimmung der Gebäude-Bezirke 47
a Harte Grenzen; Eigene Darstellung 47
b Verschwommene Grenzen; Eigene Darstellung 47
c Verschwommene Grenzen und potentielle Parks in jedem Bezirk; Eigene Darstellung 47

9.3 Stadt generiert mit anderen Parametern 48
a Raster-Straßennetz; Eigene Darstellung 48
b Endresultat; Eigene Darstellung 48

Literaturverzeichnis

[1] Tom Hatfield. Rise of the roguelikes: A genre evolves. Website, 01 2013. Online erhältlich unter `http://pc.gamespy.com/pc/ftl-faster-than-light/1227287p1.html`; abgerufen am 13.01.2019.

[2] Francis Spufford. *Backroom Boys: The Secret Return of the British Boffin*. Faber & Faber, 2003.

[3] Paul Hellquist. Inside the box: The borderlands 2 loot system. Website, 09 2013. Online erhältlich unter `http://www.gearboxsoftware.com/2013/09/inside-the-box-the-borderlands-2-loot-system/`; abgerufen am 13.01.2019.

[4] Charlie Hall. Dwarf fortress will crush your cpu because creating history is hard. Website, 07 2014. Online erhältlich unter `https://www.polygon.com/2014/7/23/5926447/dwarf-fortress-will-crush-your-cpu-because-creating-history-is-hard`; abgerufen am 13.01.2019.

[5] Jonah Weiner. The brilliance of dwarf fortress. Website, 07 2011. Online erhältlich unter `https://www.nytimes.com/2011/07/24/magazine/the-brilliance-of-dwarf-fortress.html`; abgerufen am 13.01.2019.

[6] Heather Alexandra. A look at how no man's sky's procedural generation works. Website, 10 2016. Online erhältlich unter `https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446`; abgerufen am 13.01.2019.

[7] Yoav I. H. Parish und Pascal Müller. Procedural modeling of cities. volume 2001, pages 301–308, 08 2001.

[8] Thomas Lechner und Uri Wilensky. Procedural city modeling. 01 2003.

[9] George Kelly und Hugh McCabe. A survey of procedural techniques for city generation. *Institute of Technology Blanchardstown Journal*, 14, 01 2006.

[10] Jing Sun, Xiaobo Yu, George Baciu, und Mark Green. Template-based generation of road networks for virtual city modeling. pages 33–40, 01 2002.

[11] Peter Wonka, Michael Wimmer, François Sillion, und William Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 669–677, New York, NY, USA, 2003. ACM.

[12] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, und Eugene Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27, 08 2008.

[13] Basil Weber, Pascal Müller, Peter Wonka, und Markus H. Gross. Interactive geometric simulation of 4d cities. *Comput. Graph. Forum*, 28:481–492, 04 2009.

[14] Klaus Füsser. *Stadt, Straße und Verkehr: Ein Einstieg in die Verkehrsplanung.* Vieweg+Teubner Verlag, 1997.

[15] Adam Smith. *The Wealth of Nations - An Inquiry Into the Nature and Causes of the Wealth of Nations.* University Of Chicago Press, 1977.

[16] Niclas Olsson und Elias Frank. Procedural city generation using perlin noise, 2017. Online erhältlich unter `http://www.diva-portal.org/smash/get/diva2:1119094/FULLTEXT02.pdf`; abgerufen am 09.11.2018.

[17] Ken Perlin. An image synthesizer. volume 19, pages 287–296, 07 1985.

[18] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, und Steve Worley. *Texturing And Modeling. A Procedural Approach.* The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 3 edition, 2002.

[19] John Spitzer, Simon Green, und NVIDIA Corporation. Noise and procedural techniques. 2003.

[20] Wolfgang F. Engel, editor. *Direct3d Shaderx: Vertex and Pixel Shader Tips and Tricks (Wordware Game Developer's Library).* Wordware Publishing, Inc., 2002.

[21] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[22] Przemyslaw Prusinkiewicz und Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual laboratory. Springer-Verlag, first edition second printing edition, 1990.

[23] James Scott Hanan. *Parametric L-systems and Their Application to the Modelling and Visualization of Plants*. PhD thesis, 1992. AAINN83871.

[24] Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, 2002. ACM.

[25] Ken Perlin. Noise hardware. In M. Olano, editor, *ACM SIGGRAPH 2001 Course Notes*, SIGGRAPH '01. ACM Press, 2001.