Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sven Dettmers

Prozedurale Generierung von Siedlungen in einer mutable cube world

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sven Dettmers

Prozedurale Generierung von Siedlungen in einer mutable cube world

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jenke
Zweitgutachter: Prof. Dr. Thiel-Clemen

Eingereicht am: 12. Juni 2014

Sven Dettmers

Thema der Arbeit
Prozedurale Generierung von Siedlungen in einer mutable cube world

Stichworte
Prozedurale Contentgenerierung, mutable cube world, Siedlungsgenerierung, Stadtgenerierung

Kurzzusammenfassung
Mutable cube worlds sind Welten die aus einer Blockmatrix bestehen. Sie ermöglichen es, mit einfachen Mitteln, komplexe und unendliche Welten zu generieren und darüber hinaus erlauben sie eine einmalige Interaktion zwischen Spieler und Welt. Derzeit bestehen die Siedlungen, in diesen Welten, meist aus einfachen Templates und beinhalten nur sehr wenige Gebäude. In dieser Arbeit wird ein Konzept vorgestellt, mit dem es möglich ist sehr große Siedlungen zu generieren. Hierzu wird prozedurale Generierung verwendet. In diesem Zusammenhang werden diverse prozedurale Generierungstechniken vorgestellt. Zuletzt werden Herausforderungen bei der Umsetzung und Limitationen der Lösung vorgestellt.
Sven Dettmers

Title of the paper
Procedural village generation in a mutable cube world

Keywords
Procedural Content Generation, mutable cube world, village generation, city generation

Abstract
Mutable cube worlds define worlds as a cubematrix. They can be used to effortlessly generate complex and endless worlds and allow for a unique interaction between the player and the world. Current games lack a scalable way to generate villages and citys. This thesis presents a concept, that allows the generation of very large villages and citys. To achieve this procedural generation is used. In this context procedural generation techniques are introduced. Lastly the challenges and limits of the solution are presented.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation des Themas . . . 1
- 1.2 Ziele der Arbeit . . . 2
- 1.3 Abgrenzungen . . . 2
- 1.4 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Überblick . . . 3
- 2.2 Einführung des Begriffs mutable cube world . . . 3
- 2.3 Definition mutable cube world . . . 3
- 2.4 Prozedurale Content Generierung . . . 4
- 2.5 Ersetzungssysteme . . . 4
  - 2.5.1 L-Systeme . . . 5
- 2.6 Generierung durch noise . . . 7
  - 2.6.1 Noise Verfahren . . . 7
  - 2.6.2 Parametrisierung und Komposition von noise . . . 10

3 Stand der Technik **12**
- 3.1 Überblick . . . 12
- 3.2 Generierung von Siedlungsstrukturen . . . 12
  - 3.2.1 Straßengenerierung . . . 12
  - 3.2.2 Multiagenten . . . 13
- 3.3 Generierung von Gebäuden und Details . . . 15
  - 3.3.1 CGA Shape . . . 15

4 Analyse **17**
- 4.1 Überblick . . . 17
- 4.2 Mutable cube worlds in der Spielentwicklung . . . 17
- 4.3 Eigenschaften von mcw . . . 18
- 4.4 Eigeschaften PCG . . . 19
- 4.5 Eignung von PCG Methoden . . . 20
- 4.6 Grafikframework . . . 21
- 4.7 Bewertung von generierten Städten . . . 21

5 Anforderungen **23**
- 5.1 Überblick . . . 23

5.2 Anforderungen . . . 23
5.2.1 Anforderungen an die Welt . . . 23
5.2.2 Anforderungen an die Siedlungen . . . 24
5.3 Ausschlüsse . . . 24

6 Konzeption **25**
6.1 Überblick . . . 25
6.2 Aufbau des Prototypen . . . 25
6.3 Aufbau der Darstellungs- und Steuerungskomponente . . . 26
6.4 Aufbau der Generatorkomponente . . . 26
6.4.1 Zusammenspiel mehrerer Generatoren . . . 26
6.4.2 Caching von Blöcken . . . 27
6.5 Terraingenerator . . . 28
6.6 Populationsgenerator . . . 28
6.7 Straßengenerator . . . 29
6.7.1 Großgrundstücke . . . 30
6.8 Grundstücksgenerator . . . 31
6.9 Gebäudegenerator . . . 31
6.9.1 Gebäudegrammatik . . . 33
6.9.2 Terminalsymbole . . . 33

7 Umsetzung **39**
7.1 Architektur . . . 39
7.2 Systemstart . . . 39
7.3 Cubeframework . . . 39
7.4 Meshgenerator . . . 39
7.5 Siedlungsgenerator . . . 40
7.6 Grammatiksystem . . . 40

8 Diskussion **41**
8.1 Zusammenfassung . . . 41
8.2 Architektur . . . 41
8.3 Straßennoise . . . 43
8.4 Performance Analyse des mcw Adapters . . . 44
8.5 Bewertung nach Kelly und McCabe (2011) . . . 46
8.6 Unity als mcw Framework . . . 47
8.7 Fazit . . . 47

9 Ausblick **48**
9.1 Begrenzte Welten . . . 48
9.2 Straßen und Siedlungsstruktur . . . 48
9.3 Einbezug des Terrains . . . 48
9.4 Stadtgrammatik und Stadtmodellierung . . . 49

9.5 Innenräume und kleine Strukturen 49
9.6 Benutzerkontrolle 50

1 Einleitung

1.1 Motivation des Themas

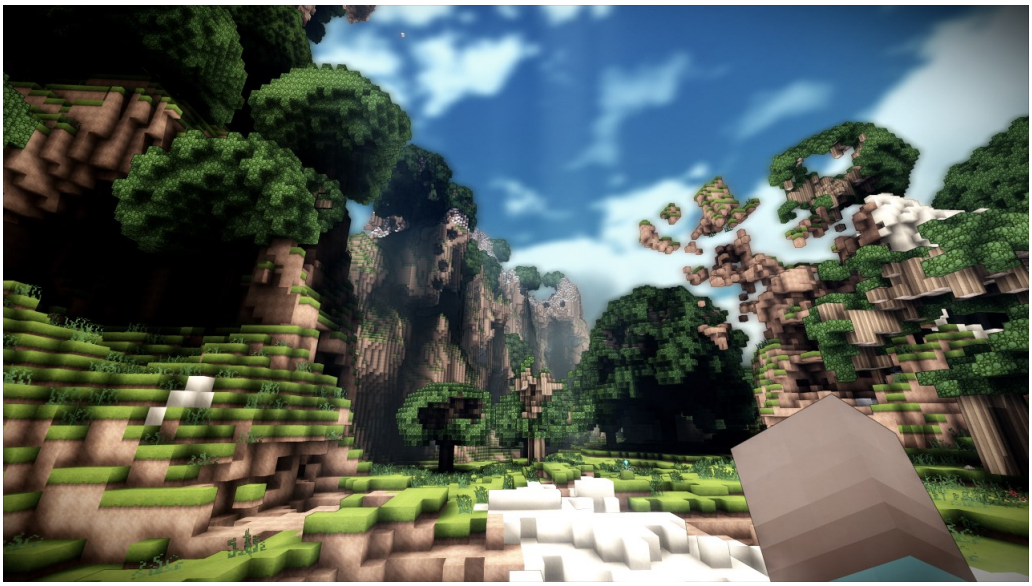

Abbildung 1.1: Terasology: Ein Beispiel für eine mutable cube world. Quelle: blog.movingblocks.net (2014)

Städte, Dörfer oder Siedlungen sind oftmals eine wichtige Komponente in der Umwelt von Computerspielen. Sie sind der Fingerabdruck einer Zivilisation und bieten den Gamedesigner viele Möglichkeiten, eine lebhafte und entdeckungswürdige Umgebung zu schaffen. Das Ausmaß einer solchen Siedlung kann von wenigen unkomplizierten Häusern bis zu riesigen detailliert erstellten Städten reichen. Das Problem, dass durch diese Skalierung entsteht, ist ein hoher Arbeitsaufwand von Designern. So ist es nicht unüblich das in einem Gamestudio wesentlich mehr Designer als Programmierer arbeiten. Um dieses Problem anzugehen und somit die Entwicklungskosten zu senken kann prozedurale Contentgenerierung (PCG) eingesetzt werden. Diese Technik ermöglicht es auch sehr kleinen Teams, beliebig große Siedlungen anzubieten.

In den letzten Jahren ist eine Form von Spielweltdarstellung beliebt geworden, die auch eine Reduzierung von Designaufwand ermöglicht. In dieser Arbeit werde ich diese Welten unter dem Begriff mutable cube world(mcw) (siehe Abbildung 1.1) definieren. Da das mcw Konzept, sowie das prozedurale Generieren von Siedlungen Entwicklungskosten reduziert, soll in dieser Arbeit das Zusammenspiel dieser beiden Konzepte untersucht werden.

1.2 Ziele der Arbeit

Ziel der Arbeit ist es, die Vor- und Nachteile der prozedurale Generierung von Siedlungen, in einer mcw, im Vergleich zu konventionellen Welten, zu untersuchen. Dabei werden speziell unendliche Welten untersucht und Algorithmen, die in der Lage sind, diese konsistent und zur Laufzeit zu generieren.

1.3 Abgrenzungen

In dieser Arbeit wird kein lauffähiges Spiel entwickelt. Es werden keine Innenräume von Gebäuden generiert. Das Interagieren mit der Welt, in der Form von manueller Entfernen und Platzieren von Blöcken wird nicht realisiert.

1.4 Aufbau der Arbeit

Das Kapitel Grundlagen stellt die grundlegenden Begriffe dieser Arbeit, sowie einige gängige Methoden der PCG vor. Im Kapitel Stand der Technik stelle ich Arbeiten zum Thema der Siedlungsgenerierung vor. Im Kapitel Analyse werden die verschiedenen Methoden zur Generierung auf ihre Vor- und Nachteile untersucht, sowie die Eigenheiten der mcw und der PCG. Folgend werden im Kapitel Anforderungen die Anforderungen für einen Prototypen spezifiziert. Anschließend wird im Kapitel Konzept die Auswahl der Methoden und deren Verwendung erörtert werden. Im Kapitel Implementierung werden die Details der Implementierung, sowie der Architektur aufgezeigt. Im Kapitel Diskussion werden die Ergebnisse bewertet. Abschließend werden im Kapitel Ausblick mögliche Erweiterungen, die im Zusammenhang mit dieser Arbeit möglich sind, vorgestellt werden.

2 Grundlagen

2.1 Überblick

In diesem Kapitel werden die Begriffe mutable cube world (mcw) und prozedurale Contentgenerierung (PCG) genauer beschrieben. Folgend werden grundsätzliche Methoden, die allgemein in der PCG verwendet werden, im Detail erklärt.

2.2 Einführung des Begriffs mutable cube world

Das Einteilen einer Spielwelt in eine Matrix aus einfachen geometrischen Formen ermöglicht es, mit geringen Aufwand und wenigen Regeln, ein Spiel zu definieren, welches dennoch eine hohe Komplexität aufweist. Einige Beispiele hierfür sind Game of Life[1], Schach oder Go[2]. Aus Sicht eines Spielentwicklers ist diese Eigenschaft besonders interessant, da sie mehrere Vorteile mit sich bringt. Durch die geringe Anzahl von Regeln lässt sich solch ein Spiel einfach umsetzten und bietet neuen Spielern einen schnellen Einstieg.

Durch den großen Erfolg von Minecraft[3] wurde eine spezielle Form einer solchen Matrixwelt sehr beliebt. Für diese Arbeit wird der Begriff mutable cube world eingeführt, um eine klare Definition von deren Eigenschaften zu haben.

2.3 Definition mutable cube world

Eine mutable cube world(mcw) besteht aus einer einheitlichen Hyperwürfelmatrix, die mit Blöcken befüllt werden kann. Ein Block ist eine Geometrie, deren Größe kleiner oder gleich dem eines Hyperwürfels der Welt ist. Jeder Block aus der Blockmenge kann grundsätzlich an einer beliebigen Position platziert bzw. entfernt werden. Objekte die nicht zur mcw gehören, wie zum Beispiel Spieler oder Monster, unterliegen nicht den Regeln der mcw und können

[1] http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens
[2] http://de.wikipedia.org/wiki/Go_(Spiel)
[3] https://minecraft.net

sich frei bewegen. Es ist für sie möglich mit der mcw zu interagieren. Ein einfaches Beispiel hierfür ist die Kollision mit der mcw Geometrie.
Die Hyperwürfelmatrizen für konventionelle Anwendungen haben entweder zwei oder drei Dimensionen. Die Konzeption dieser Arbeit wird sich auf die dreidimensionalen mcw beziehen.

2.4 Prozedurale Content Generierung

Prozedurale Contentgenerierung oder Prozedurale Synthese oder auch einfach Prozedurale Generierung genannt ist das Generieren von Spieleinhalten durch Algorithmen (Wikipedia Foundation, 2014b). Sie ermöglichen es, durch Einbringen von Zufallswerten, im Prinzip unendliche Spieleinhalte zu generieren. Sie können eingesetzt werden um ein Spiel auch auf lange Sicht spannend zu gestalten, oder um Entwicklungskosten und -zeit zu sparen. Eine weitere Anwendung von PCG findet sich im geringen Speicheraufwand, der für den zu generierenden Content benötigt wird. Für den einfachsten Fall muss lediglich die Seed des random number generators und der Algorithmus zum Generieren gespeichert werden.

2.5 Ersetzungssysteme

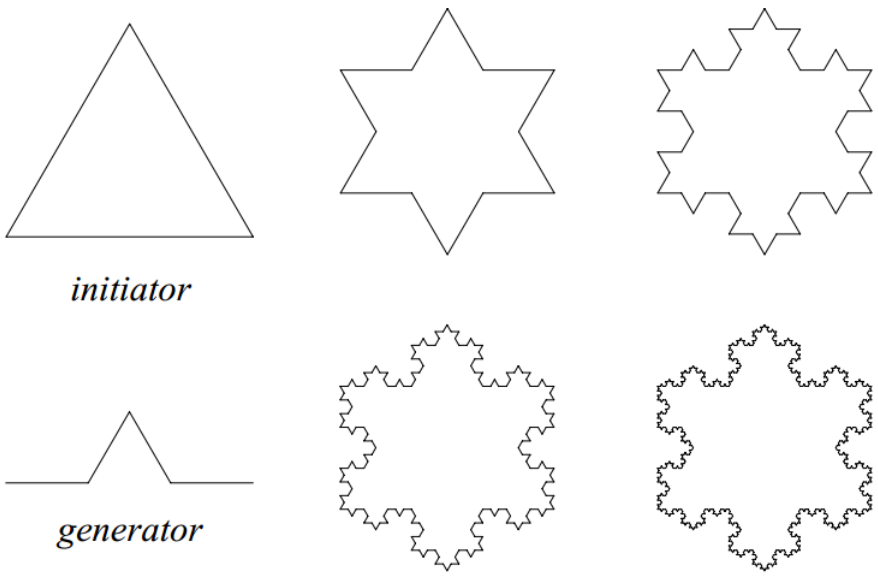

Abbildung 2.1: Kochsche Schneeflocke. (Entnommen aus Prusinkiewicz und Lindenmayer (1990))

Regelbasierte Ersetzungssysteme bilden die Grundlage für viele Algorithmen der prozeduralen Generierung. Ausgehend von einem Startzustand werden weitere Zustände schrittweise abgeleitet (siehe Abbildung 2.1). Die Überführung erfolgt dabei gemäß eines Regelwerks. Eine Regel besteht aus einer oder mehrerer Bedingungen und der Ersetzung. Ein Ersetzungssystem prüft beim Ableiten eines Zustandes auf die Bedingungen der Regeln und ersetzt Teile des Zustandes durch seine Ersetzung. In welcher Reihenfolge und welcher Teil eines Zustandes ersetzt wird, hängt vom jeweiligen System ab.

2.5.1 L-Systeme

Der gesamten Abschnitt der L-Systeme bezieht sich auf Prusinkiewicz und Lindenmayer (1990) als Quelle.

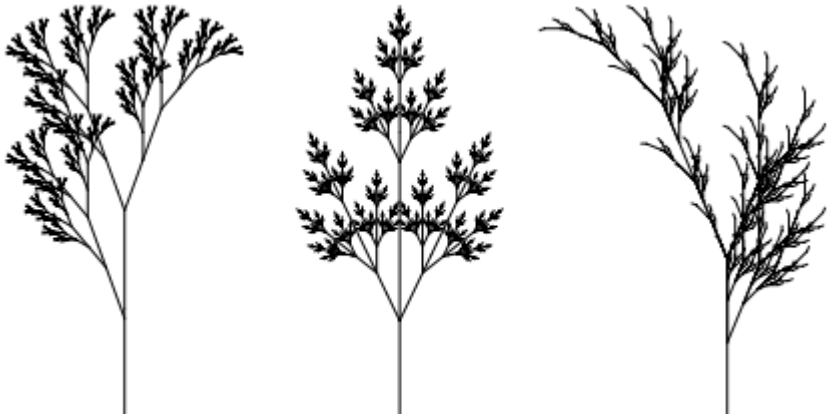

Abbildung 2.2: Pflanzenskelette generiert über L-Systeme. (Entnommen aus Prusinkiewicz und Lindenmayer (1990))

Ein Lindenmeyer-System oder kurz L-System ist eine Grammatik, in der Zeichenketten gemäß Ersetzungsregel manipuliert werden. Im Vergleich zu Chomsky Grammatiken werden hier alle Symbole gleichzeitig ersetzt. Dadurch sind L-Systeme sehr gut geeignet, um Wachstum zu simulieren (siehe Abbildung 2.2). L-Systeme wurden von Lindenmeyer entwickelt um biologische Vorgänge zu simulieren. L-Systeme treten in verschiedenen Klassen und mit verschiedenen Erweiterungen auf.

Einfache Grammatik

Die einfachste Form von L-Systemen besteht aus den Übergangsregeln, dem Alphabet und dem Eingabewort. Bei einer Ableitung werden alle Symbole gleichzeitig abgeleitet. Gibt es für ein Symbol A eine Regel A -> XYZ wird A durch XYZ ersetzt.

Turtleinterpretation

Um ein visuelles Ergebnis zu erzielen, kann eine Turtleinterpretation verwendet werden. Dabei werden die Symbole der Reihenfolge nach eingelesen und als Befehl an einen Schreibkopf (Turtle) weitergegeben. Der Zustand der Turtle kann viele verschiedene Informationen enthalten, jedoch mindestens die Position und die Ausrichtung der Turtle. Ein mögliches Alphabet für solch eine Turtle könnte wie folgt aussehen.
F : 1 Einheit nach vorne bewegen und zeichnen
f : 1 Einheit ohne zeichnen nach vorne bewegen
+ : Linksdrehung um einen vordefinierten Winkel
- : Rechtsdrehung um einen vordefinierten Winkel

Stack

Die einfache L-Grammatik erlaubt nur einen linearen Verlauf einer Turtle. So gibt es kein Symbol, das den Sprung einer Turtle auf einen vorherigen Punkt erlaubt. Um einen älteren Punkt zu erreichen, muss der gesamte Pfad zurück gelaufen werden. Vor allen Dingen bei verzweigten Strukturen wird die einfache Grammatik somit sehr kompliziert und viele Schritte werden unnötig mehrfach ausgeführt.
Um nun leichter mit verzweigten Strukturen umzugehen, wird die Grammatik um einen Stack, Push Symbol '[' und Pull Symbol ']' erweitert. Auf den Stack kann nun der Zustand der Turtle gespeichert und geladen werden. Die Wahl der Klammern als Symbol ist hier geschickt, da jede Klammerverschachtlung als ein Pfad der Verzweigung gelesen werden kann.

Zufallsauswahl

Um zufällig verschiedene Ergebnisse zu erzielen, werden die Regeln mit einem Wahrscheinlichkeitswert P erweitert. Die Summe von allen P, die ein Symbol A verarbeiten, muss hierbei 100% betragen. Soll das Symbol A abgleitet werden, wird ein Zufallswert bestimmt, über den eine der Regeln für A ausgewählt wird.

Kontextsensitive

Um eine mächtigere Klasse von Grammatiken zu verwenden kann der Kontext miteinbezogen werden. Für die Auswahl der Regel zur Ableitung werden jetzt, über das zu ersetzende Symbol A hinaus, vorherige V und nachfolgende Zeichen N berücksichtigt. Ist in einem Wort die Zeichenfolge V A N -> B vorhanden, kann die Regel angewendet und A durch B ersetzt werden.

Parametrisierung

Eine weitere Erweiterung für L-Systeme sind Parameter. Ein Symbol kann hierbei einen Parameter erhalten. Regeln können sich mithilfe von Bedingungen auf diese Parameter beziehen. Bei der Ableitung können Parameter manipuliert werden.

2.6 Generierung durch noise

Der gesamten Abschnitt der Noisefunktionen bezieht sich auf Perlin (2001) als Quelle. Noise Funktionen werden verwendet um zufällig Muster zu erzeugen. Sie bilden von einem Eingabevektor eines beliebigen Grades auf einen Wert ab. Dieser Wert kann beliebig interpretiert werden. So werden häufig noise Funktionen eingesetzt um Höhenwerte für Terrains zu ermittelt. Die Funktionswerte verhalten sich dabei Deterministisch zu einer Seed. Um ein Muster zu speichern genügt es demnach die Seed und die Funktion zu speichern. Um ein gewünschtes Muster zu erhalten ist die Parametrisierung, Komposition, sowie die Auswahl des noise Verfahrens entscheidend.

2.6.1 Noise Verfahren

White noise

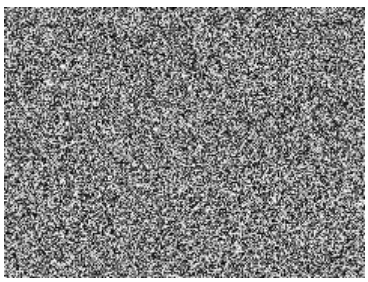

Abbildung 2.3: Typisches white noise Muster. (Quelle: Wikipedia Foundation (2014c))

White noise ist die einfachste Art der Noisefunktionen. Sie bilden den Eingabevektor auf eine Zufallszahl ab. Hierbei habe die Zufallszahlen keinen Bezug zu einander. Die einzige Berechnung, ist die der Zufallszahl. White noise erzeugt das typische Rausch-Muster (siehe Abbildung 2.3). Die Anwendungsgebiete für solch ein Muster sind sehr gering, da Muster in der Natur einer gewissen Ordnung folgen. Betrachtet man das Höhenprofil eines Berges, unterscheiden sich die Höhenwerte nur sehr gering, wenn man zwei Punkte in unmittelbarer Nähe betrachtet. In dem white noise Muster wären dieser Höhenunterschiede jedoch möglich.

Value noise

Value noise setzt die Zufallswerte in Bezug zueinander, um Ordnung in den zufälligen Mustern darzustellen. Dies geschieht durch das Verwenden eines n-dimensionalen Gitters, dabei entspricht n dem Grad der Funktion. An den Eckpunkten des Gitters werden Zufallswerte generiert, entsprechend wie bei white noise. Um einen Wert, der nicht auf den Eckpunkten des Gitters liegt, zu ermitteln, wird zwischen den n anliegenden Eckpunkten interpoliert. Hierbei zeigt sich auch, dass dieses Verfahren nicht in höhere Dimensionen skaliert, da die Laufzeit der Interpolation exponentiell mit $2^n - 1$ wächst. Die Interpolation muss nicht zwangsweise linear verlaufen. So eignet es sich für übliche Anwendungsfälle, eine Glättungsfunktion zu verwenden, um einen seichteren Übergang zu den Eckpunkten des Gitters zu erhalten.

Gradient noise

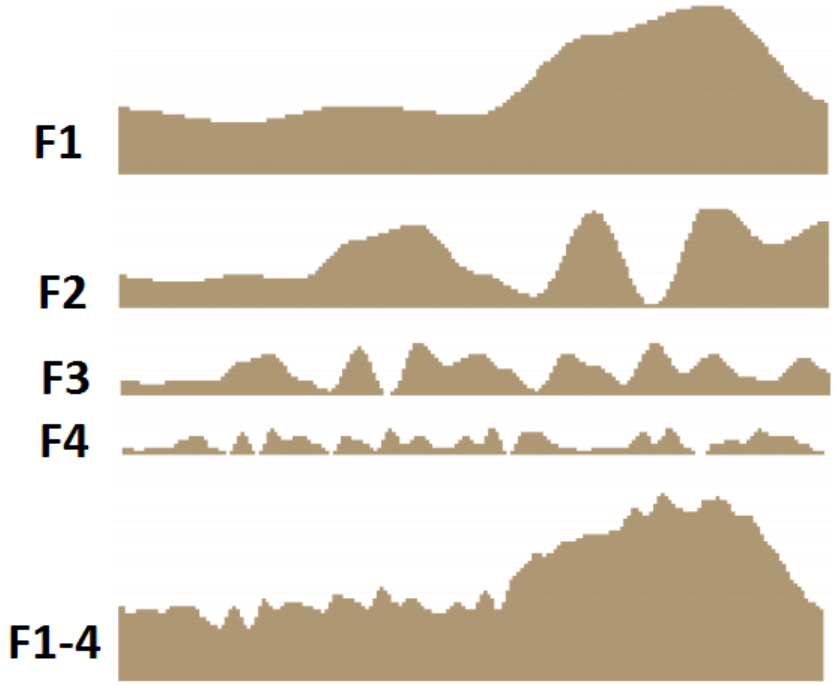

Abbildung 2.4: Ein generierter Berg im Querschnitt. Die Wellenlänge und Amplitude wird im verlauf von der Funktion F1 zur F4 reduziert

Gradient Noise Verfahren verwenden Steigungsvektoren statt Zufallswerten, um die einzelnen Funktionswerte zu berechnen.
Perlin Noise gehört zu den Gradient Noise Verfahren und ist eine der am häufigst verwendeten Noisefunktionen. Es verwendet wie Value noise ein n-dimensionales Gitter. Die Zuweisung und Interpolation der Eckpunkte des Gitters ist auch gleich, jedoch werden die Eckpunktwerte anders berechnet. Hierfür werden zwei Vektoren benötigt. Der erste ist ein zufälliger n-dimensionaler Vektor. Der zweite ist der Vektor vom jeweiligen Eckpunkt des Gitters zu dem Punkt des gesuchten Wertes. Das Skalarprodukt der Vektoren ergibt nun den Zufallswert des Eckpunktes. Um Effekte der Selbstähnlichkeit darzustellen, werden mehrere unterschiedlich parametrisierte Funktionen verwendet, um den endgültigen Funktionswert zu errechnen (siehe Abbildung 2.4).

Neben Perlin Noise gibt es noch Simplex Noise als Gradient Noise Verfahren. Dieses Verfahren soll den hohen Rechenaufwand der Interpolation bei höheren Dimensionen lösen.

Hierzu wird das n-dimensionale Gitter in Simplexe aufgeteilt. Dies hat den Vorteil, dass die Anzahl der beteiligten Ecken nur n+1 ist im Gegensatz zu n^2 bei einem Würfelgitter. Die Zuweisung von Zufallswerten erfolgt wie bei Perlin Noise. Der finale Wert für einen gesuchten Punkt X wird jedoch nicht über die Interpolation der beteiligten Ecken berechnet. Stattdessen wird der Einfluss der Eckpunkte, mit steigender Distanz zum Punkt, reduziert und die resultierenden Werte aufaddiert.

2.6.2 Parametrisierung und Komposition von noise

Noisefunktionen werden hauptsächlich durch die Parameter Wellenlänge bzw. Frequenz und Amplitude gesteuert. Dabei beschreibt die Wellenlänge die Auflösung des zugrundeliegenden Gitters und die Amplitude beschreibt das Intervall, in dem die Zufallswerte vorkommen können. Der Grad einer NoiseFunktion ist beliebig und wird abhängig von der Verwendung gewählt. Hierzu einige Beispiele.

- Eindimensional: Um gerade Linien zu verzerren, um einen Sketchlook zu erhalten.
- Zweidimensional: Zum Erstellen von Höhenkarten, die als Terrain interpretiert werden können. Zum Generieren von Texturen.
- Dreidimensional: Animierte 2D Bilder. Hierbei wird die dritte Dimension als Zeit interpretiert.
- Vierdimensional: Animierte 3D Objekte, zum Beispiel Wolken.

Noisefunktionen werden durch mathematische Operationen manipuliert und verknüpft, um somit gewünschte Muster, wie in der Abbildung 2.5 gezeigt, zu erhalten.

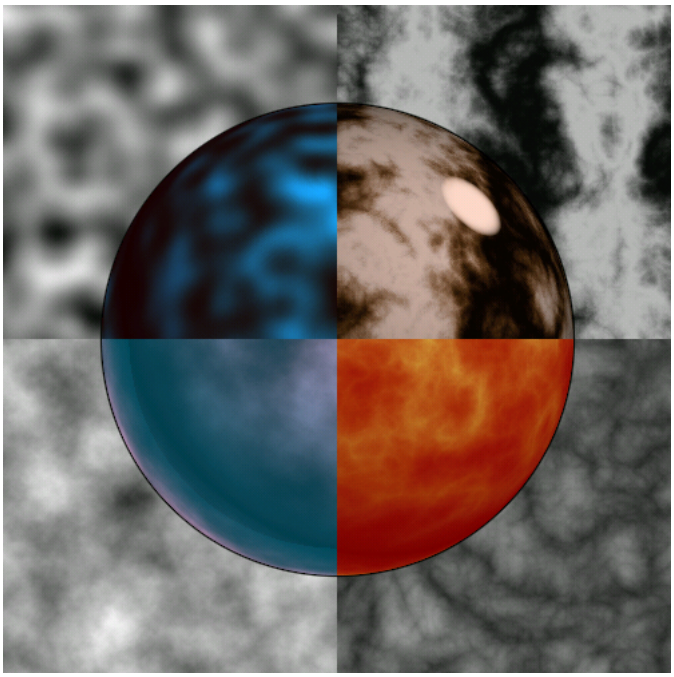

Abbildung 2.5: Verschiedene Muster durch Manipulation von Perlin Noise. (Quelle: Perlin (1999))

3 Stand der Technik

3.1 Überblick

Dieses Kapitel zeigt den Stand der Technik bezüglich zweier Kernthemen der Siedlungsgenerierung auf. Die Generierung von Siedlungsstrukturen definiert eine grobe Sicht auf Siedlungen. Die größten Einflüsse sind hierbei das Straßennetz, sowie die Verteilung und Position von Gebäudearten. Die Gebäudearten haben hierbei oftmals eine grobe Unterteilung in Wohn- ,Kommerziell, und Industriegebäude, oder werden in Gebiete solcher Art zusammengefasst. Die Generierung von Gebäuden und Details stellt die feine Sicht auf eine Siedlung dar. Anders als bei der groben Unterteilung in Wohn- ,Kommerziell, und Industriegebiete geht es hierbei um die tatsächliche visuelle Generierung von Gebäudegeometrien und Fassadentexturen, sowie Grundstücke, Gärten, Plätze und Vegetation.

3.2 Generierung von Siedlungsstrukturen

3.2.1 Straßengenerierung

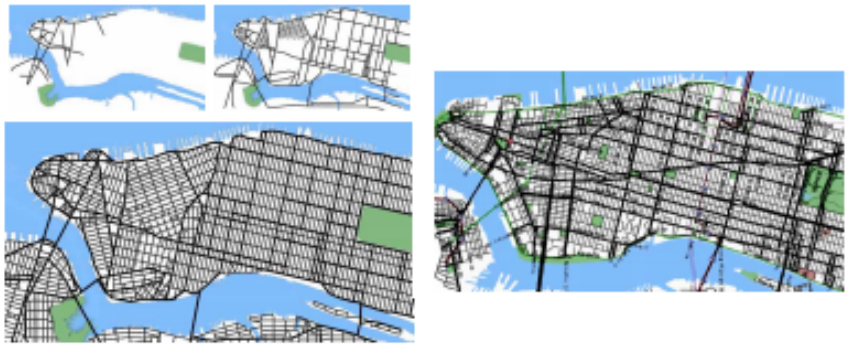

Abbildung 3.1: Straßengenerierung Manhattan. Links oben: Generierung nach wenigen Schritten. Links unten: Nach vollendeter Generierung. Rechts: Straßen vom echten Manhattan(Entnommen aus I H Parish, Yoav und Müller (2001))

In dem Paper von I H Parish, Yoav und Müller (2001) wird beschrieben, wie L-Systeme zur Straßengenerierung verwendet werden können (siehe Abbildung 3.1). Da bei einem Regelwerk für Straßen viele Parameter und Bedingungen eine Rolle spielen, ist ein gewöhnliches L-System nur schwer erweiterbar. Durch das Hinzufügen von Bedingungen müssen viele Regeln überarbeitet werden. Um dies zu umgehen, wird aus dem L-System nur ein generisches Template generiert. Die Parameter werden nun von einer GlobalGoal-Funktion, abhängig von den dominierenden globalen Zielen, gesetzt. Zuletzt werden die Parameter durch eine Funktion LokalConstrains überprüft. Dabei werden die Parameter so angepasst, dass sie zu den lokalen Bedingungen passen. Falls die lokalen Bedingungen nicht erfühlt werden können, wird der entsprechende Teil entfernt. Die lokalen Bedingungen prüfen, ob die Straße einen validen Pfad hat und prüft auf andere Straßen in der Umgebung, um Kreuzungen zu erstellen.

3.2.2 Multiagenten

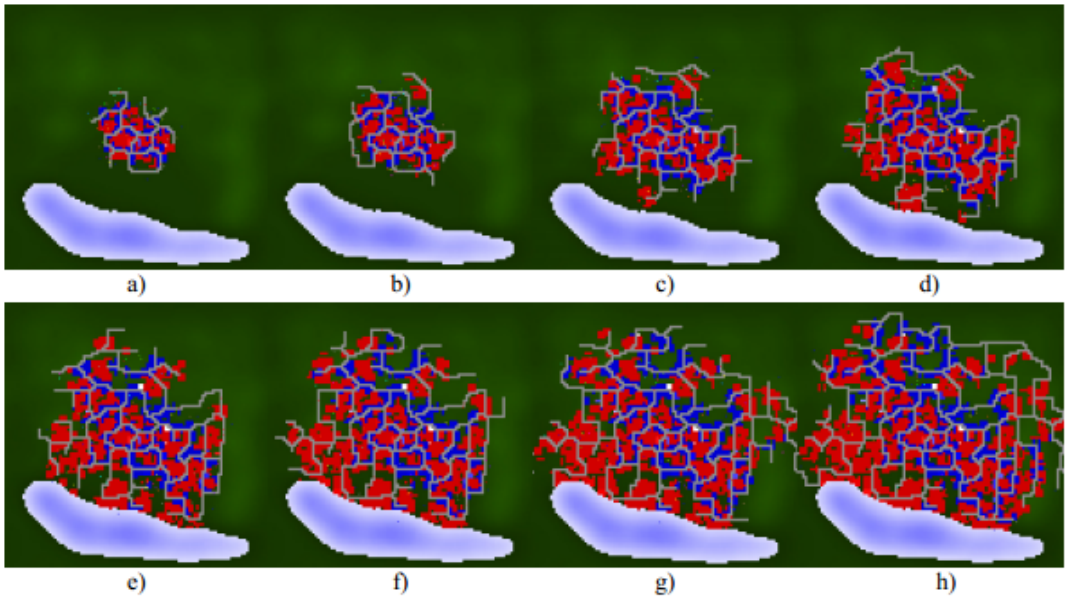

Abbildung 3.2: Zeitliche Entwicklung einer agentenbasierenden Generierung. Die roten Felder repräsentieren Wohngebiete, die blauen kommerzielle Gebäude.(Entnommen aus Lechner u. a. (2003))

Multiagentensysteme finden sich eher im Bereich der Simulation als in der PCG. Die Ausarbeitung von Lechner u. a. (2003) zeigt jedoch, das auch mit Multiagenten die Generierung von Städten möglich ist (siehe Abbildung 3.2).

Die Welt, in der die Stadt generiert wird, ist eine zweidimensionale Matrix. Zunächst werden Terraininformationen als Input benötigt. Hierzu können Gewässer, sowie Höheninformationen für jedes Feld definiert werden. Diese Informationen können durch Prozedurale Generierung, aus existierenden Daten, oder auch manuell entworfen werden.

Der Generierungsprozess beginnt mit der Entwicklung der Straßen. Hierzu gibt es zwei Arten von Agenten. Der Extender Agent wandert über das Terrain, bis er ein Feld findet, das von keiner Straße unterstützt wird. Dies kann der Agent feststellen, da alle Straßen den Feldern, in einem bestimmten Radius, Distanzinformationen zuordnen. Hat ein Extender ein nicht unterstütztes Feld gefunden, läuft er in Richtung des Straßennetzes. Er versucht dabei starke Steigungen des Terrains zu meiden. Sobald er auf das Straßennetzwerk trifft, werden Bedingungen wie Straßendichte, die Nähe zu anderen Kreuzungen usw. geprüft. Ist die Prüfung erfolgreich, wird die Straße erstellt und der Agent wiederholt den Prozess.
Der Connector Agent wandert über das bestehende Straßennetz. Dabei versucht er, in einem gewissen Radius, ein Straßenfeld als Ziel zu definieren. Folgend versucht er dieses Ziel über das Straßennetz mittels Breitensuche zu erreichen. Überschreitet der Versuch einen gewissen Radius, versucht der Agent eine Straße zwischen Start und Endpunkt zu erstellen. Dabei gelten die selben Bedingungen zur Straßenerstellung wie bei dem Extender Agent.

Um Gebäude zu generieren, werden weitere Agenten eingesetzt. Bisher sind zwei Agenten implementiert. Residential Agenten prüfen auf Population und setzten Wohngebäude an Orte, die kleine Straßendichte, sowie eine gewisse Entfernung von den Straßen haben. Die Commercial Agenten platzieren kommerzielle Gebäude an Orten, die hohe Straßendichte und nahe an Straßen sind.

3.3 Generierung von Gebäuden und Details

3.3.1 CGA Shape

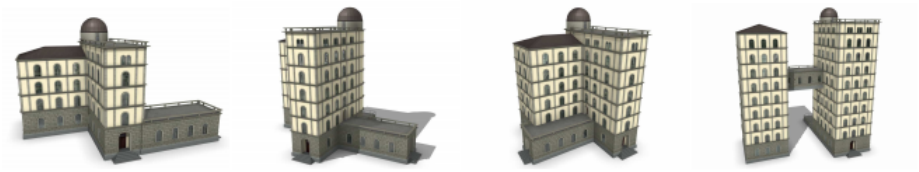

Abbildung 3.3: Gebäude generiert durch die CGA Shape Grammatik (Entnommen aus Müller u. a. (2006))

In dem Paper von Müller u. a. (2006) wird die CGA Shape Grammar zum Generieren von Gebäuden vorgestellt (siehe Abbildung 3.3). Die vorgestellte Grammatik CGA (Computer generated architectur) shape ist einer Weiterentwicklung von Wonka u. a. (2003). Im Gegensatz zur simultanen Ableitung bei L-Systemen, wird diese Grammatik sequenziell abgeleitet, um mehr Kontrolle über die Struktur zu erhalten.
Die Grammatik verwendet, ähnlich wie erweiterte L-Systeme, Stackoperationen, Parameter und Zufallsauswahl. Ein Symbol repräsentiert eine Form und hält als Parameter: Positionsvektor P, drei orthogonale Vektoren X, Y und Z, die ein Koordinatensystem beschreiben, sowie einen Vektor S, der die Skalierung beschreibt. Diese Parameter dienen als Bounding Box für die Form und werden Scope genannt. Regeln sind nach folgender Struktur aufgebaut:

ID : Vorgänger : Bedingung -> Nachfolger : Wahrscheinlichkeit.

Da die Ableitung sequenziell erfolgt, muss eine Entscheidung getroffen werden, in welcher Reihenfolge Symbole abgeleitet werden. Hierfür werden die Regeln so mit Prioritäten versehen, dass sie sich proportional zum Detailgrad der Form verhalten. Demnach wird immer die Regel mit der höchsten Priorität ausgewählt. Der Vorteil dieser Wahl ist, dass der Detailgrad des Gebäudes gleichmäßig mit der Anzahl der Ableitungen wächst. Die CGA Shape Grammatik erhält bereits abgeleitete Formen als inaktive Objekte, um die Hierarchie einer Form nachvollziehen zu können.
Um mit der Grammatik Formen abstrakt entwerfen zu können, werden spezielle Produktionsregeln angewandt. Hierbei handelt es sich um Regeln, wie zum Beispiel Translation, Skalierung, oder Rotation. Eine besondere Rolle spielt die Split Regel. Sie teilt eine Form entlang der angegeben Achse. Die Parameter der Split Regel zeigen, wie groß der jeweilige geschnittene

Teil der Form sein wird. Diese Angaben können auch relativ sein. Eine spezielle Form des Splits ist der Component Split. Er teilt eine Form in Formen niederer Dimensionen. So kann man zum Beispiel, das Volumen eines Gebäudes in seine zweidimensionalen Fassaden aufteilen.

Um mit der Grammatik Gebäude zu entwerfen, wird zunächst das Grundstück als Axiom gegeben. Mit Skalierungs-, Rotations-, und Translationsregeln wird folgend ein Grundvolumen erstellt. Danach kann die Split Regeln angewendet werden, um individuelle Geometrien zu erhalten. Um unterschiedliche Gebäude zu erzeugen, können die Wahrscheinlichkeiten der Regeln genutzt werden, oder die Parameter der Regeln mit Zufallszahlen manipulieren werden. Zum Erstellen von Dächern können über die Straight Skeleton Methode Geometrien generiert werden, welche auf das Volumen des Gebäudes aufgesetzt werden können.
Es ist möglich, dass die Texturen der Fassade teilweise verdeckt werden und somit zum Beispiel eine Tür nur zur Hälfte sichtbar ist. Um dies zu umgehen, gibt es einen Occlution Test, der überprüft ob eine Geometrie teilweise, ganz, oder gar nicht, verdeckt wird. Es kann auch überprüft werden, ob eine klare Sichtline zu einer bestimmten Geometrie, zum Beispiel Straße, gegeben ist.

4 Analyse

4.1 Überblick

In diesem Kapitel werden zunächst die Eigenschaften von mcw und Siedlungen untersucht, um mögliche Anforderungen und Restriktionen der Konzeptionierung aufzuzeigen. Folgend sollen die möglichen Techniken der PCG und deren Verwendungsmöglichkeiten, im Zusammenhang mit mcw, untersucht werden. Danach werden mögliche Grafikframeworks zur visuellen Darstellung erörtert. Zuletzt werden Kriterien zur Bewertung der Resultate aufgezeigt.

4.2 Mutable cube worlds in der Spielentwicklung

Das Prinzip der mcw lässt sich in der Spielentwicklung nicht nur in neueren Spielen finden. So verwendete schon das Spiel 'Dungeon Keeper', erschienen im Jahr 1997 eine zweidimensionale mcw (Wikipedia Foundation, 2014a). Die große Welle an Spielen mit einer mcw startete mit dem großen Erfolg des Studios Mojang, mit dem Spiel Minecraft. Es wurde alleine für den PC 15 Millionen mal verkauft (Makuch, 2014). In den folgenden Abschnitten sollen einige der entstandenen mcw Spiele und die Art der Verwendung der mcw in diesen Spielen vorgestellt werden, um die Möglichkeiten von mcw darzustellen. Sofern nicht anders angegeben, beziehen sich diese Informationen aus eigener Spielerfahrung. Ansonsten wurden Gamereview Seiten, oder Angaben der Spielentwickler verwendet, um einen Überblick über das jeweilige Spiel zu erhalten. Diese Quellen können unpräzise und unzureichende Informationen enthalten. Diese Quellen werden verwendet, da ich trotz intensiver Recherche keine wissenschaftlichen Texte mit Bezug auf die Thematik gefunden habe.

Ace of Spades

Ace of Spades ist ein First-Person-Shooter. Hier werden die einfachen Manipulationsregeln einer mcw genutzt, um es dem Spieler zu ermöglichen, Schützengraben, Bunker oder ähnliches zu errichten bzw. zu zerstören (Johnson, 2013).

Cube World

In Cube World wird die mcw prozedural generiert. Die Welt wird hinter dem Sichtbereich des Spielers weiter generiert, sobald er sich in die Richtung des Horizontes bewegt. Das Spiel gehört zu dem Rollenspiel Genre. Die Verwendung der mcw dient hier zum einen als Stilmittel, so bestehen auch die Spieleravatare aus Blöcken. Zum anderen ermöglicht die mcw eine einfachere Implementierung der Weltgenerierung, sowie reduzierten Designaufwand. Eine wichtige Eingenschaft, da Cube World, gemäß eigenen Angaben (siehe Funck und Funck (2014)) nur von zwei Person entwickelt wird .

Eldrich

In dem Spiel Eldrich wird die mcw verwendet, um aus vielen Modulen und Segmenten, Level zu generieren. Zudem können Blöcke im Level zerstört werden, um Abkürzungen oder Umwege zu erschaffen und verleiht somit dem Leveldesign mehr Abwechselung (Kaiser, 2013).

Minecraft

Die Generierung der Welt erfolgt in Minecraft nach dem selben Prinzip wie bei Cube World. Dabei gibt das Spiel kein konkretes Ziel vor. Eine solche Art von freiem Spiel wird auch Sandboxspiel genannt. Im Gegensatz zu Cube World steht die Manipulation der Welt im Mittelpunkt des Spieles. Die mcw eignet sich hierbei perfekt. Sie bietet einfache Manipulationsregeln für die Welt, sowie die Möglichkeit jeden Teil der Welt zu ändern.

4.3 Eigenschaften von mcw

Wie sich zeigt, wird im Zusammenhang mit mcw oftmals Prozedurale Generierung verwendet. Dies zeigt auch die weiteren Vorteile einer mcw auf. Es können leichter Algorithmen entworfen werden, welche die Spieleinhalte erzeugen und somit viele Designaufgaben vereinfacht oder reduziert werden. Des weiteren können, durch das Verwenden von PCG, variierende Inhalte erzeugt werden, was dazu führt das jeder Spieldurchlauf eine neue Erfahrung für den Spieler darstellt, oder eine unendliche Welt zum Entdecken bereitstellt.
Ein weiterer Vorteil von mcw liegt in dem reduzierten Designaufwand in Bezug auf die Levelelemente. Da sich die Geometrie auf einen Würfel reduziert, müssen keine 3D Assets erstellt werden und auch die Texturierung gestaltet sich einfacher, da sie nur auf die Seiten des Würfels angepasst werden müssen. Da sich Dinge in einer mcw sich aus Blöcken zusammenstellen, reichen oft generische Texturen wie z.B. Stein, Holz, Gras, Erde, und so weiter.

Mutable cube worlds bieten die Möglichkeit, unendliche Welten zu generieren. Dabei wird die Welt um den Spieler herum generiert. Bewegt er sich, wird die Welt in seiner Bewegungsrichtung weiter generiert. Da nicht unendlich Speicher zur Verfügung steht, müssen irgendwann Teile der Welt verworfen werden. Optimaler Weise werden hier Bereiche gewählt, die weit vom Spieler entfernt sind. Dies erzeugt jedoch ein Problem. Kehrt der Spieler zurück an einen Ort, der zuvor gelöscht wurde, muss dieser neu generiert werden. Dabei erwartet der Spieler, dass er den Ort so vorfindet, wie er ihn verlassen hat. Daraus folgt, dass die Welt an jeder Stelle konsistent sein muss.
Ein weiteres Problem der endlosen Welt ist, dass sich der Spieler theoretisch unendlich weit von einem definierten Ursprung entfernen kann. Besonders Ersetzungssysteme, wie zum Beispiel L-Systeme, wachsen meist aus einem Ursprung heraus. Ortsabhängige Informationen dieser Ersetzungssysteme können nicht ohne weiteres gelöscht werden und die Informationen vom Ursprung bis zum Spieler zu speichern ist, ab einer bestimmten Entfernung nicht möglich.

4.4 Eigeschaften PCG

Die Natur verwendet oftmals einfache Regeln, um komplexe Geometrien zu erzeugen. Welche Geometrie letztendlich erzeugt wird, ist oftmals nicht ohne weiteres möglich vorherzusagen, wenn nur die ursprünglichen Regeln betrachtet werden. Sind gewünschte Muster bekannt, können Algorithmen entwickelt werden, die, ebenfalls aus einfachen Regeln, komplexe Geometrien erzeugen. Beispiel sind Gebirge oder das charakteristische Muster von Hölzern. Welche Geometrie ein solches Gebirge hat, kann über wenige Parameter gesteuert werden. Diese Form der Generierung ist sehr mächtig und kann in vielen Formen in der Natur beobachtet werden. Menschliches Design fordert Optimierung für Zweck und Ästhetik und wird mehr über Design, anstatt Konstruktionsregeln, entschieden. Dies führt oftmals zu einfacheren und klarer definierten Geometrien, jedoch gibt es kein einfaches Verfahren, um diese Geometrien zu erzeugen.

Oftmals ist das unvorhersehbare Muster der Generierung erwünscht, was ein weiterer Vorteil der PCG ist. In solchen Fällen interessiert nicht die genaue Gestalt, sonder nur eine gewisse Authentizität. Wälder sind ein gutes Beispiel für einen Anwendungsfall. Die genaue Anzahl der Äste und deren Ausrichtung ist unwichtig, solange sie nicht bei jedem Baum gleich ist. PCG ist eine sehr gute Lösung für dieses Problem, da es nahezu unendlich einzigartige Bäume, ohne Aufwand, generieren kann.
Steht man jedoch nur vor der Aufgaben z.B. Stühle zu generieren, oder jegliches anderes

menschliches Design, muss zumindest eine abstrakte Modellierung stattfinden. Der unterschied zu den Bäumen ist dabei, dass Stühle nicht von einer Zelle aus wachsen, sondern einem bestimmten Design folgen. In diesem Punkt unterscheidet sich das Generieren von Natur und dem Design des Menschen.

4.5 Eignung von PCG Methoden

Verschiedene PCG Methoden haben verschiedene Vor- und Nachteile. Im Folgenden werde ich diese Methoden, im Bezug auf die Generierung von Siedlungen in mcw, untersuchen.

Grammatiken

Wie das Kapitel Stand der Technik zeigt, können Grammatiken zum Generieren von Straßennetzen, sowie Gebäuden sehr gut eingesetzt werden. Problematisch wird es wenn, endlose und konsistente Welten generiert werden sollen, wie im Abschnitt 'Eigenschaften mcw' schon erwähnt wurde.

CGA Shape

Die CGA Shape Grammatik zeigt, wie sich durch abstrakte Definition von Gebäudeteilen komplexe und vielfältige Gebäude generieren lassen. Dabei ist zu erwähnen, dass dies auch einen gewissen Aufwand an Modellierung mit sich bringt und somit, für eine spezifische Anwendung, ein Initialaufwand für den Entwickler entsteht.

Noise Analyse

Noisefunktionen haben den Vorteil gegenüber Ersetzungssystemen, dass sie, ohne zusätzlichen Aufwand, an jeder Stelle ausgewertet werden können. Diese Eigenschaft ist bestens geeignet für die Anwendung auf endlosen Welten, da die Berechnung eines Funktionswertes unabhängig von der Distanz zu einem Ursprung ist.
Auch die Berechnung der Konsistent ist automatisch gewährleistet, da die Funktionwerte immer eindeutig sind und keine Seiteneffekte berücksichtigt werden müssen.

Agenten

Eine Modellierung für das endlose Generieren von Siedlungen mit Agenten ist denkbar. Problematisch ist jedoch die Berechnung der Konsistenz. Falls sich in der Vergangenheit bereits geladene und aktuelle Abschnitte mischen, fehlen den Agenten die Teile der Umwelt, die zur

Berechnung des vergangen Abschnittes verwendet wurde. Somit ist es nicht ohne weiteres möglich, dass die Agenten diesen Abschnitt exakt gleich neu aufbauen.

4.6 Grafikframework

Zur Darstellung der Siedlungen wird ein Grafikframework benötigt. Hierbei gibt es die Möglichkeit von Grund auf an OpenGL[1] oder DirectX[2] zu verwenden. Es gibt jedoch eine Vielzahl von Grafikengines, die einen Einstieg auf einer höheren Ebene erlauben. Zu den frei erhältlichen und bekannten Engines zählen, unter anderem, Unity[3] und Quake[4]. Kostengünstige alternativen wäre, zum Beispiel, die Cry Engine[5] oder die Unreal Engine[6]. Diese Engines sind jedoch eher für größere Projekte ausgelegt.

4.7 Bewertung von generierten Städten

In einem Paper von Kelly und McCabe (2011) werden folgende Bewertungskriterien für prozedurale Stadtgenerierung vorgestellt.

- Realismus - Wie nahe an der Realität ist die generierte Stadt.
- Skalierung - Hat die generierte Stadt die Größe einer Stadt.
- Variation - Welchen Grad von Variation ermöglicht der Algorithmus für die erzeugten Elemente (e.g. Häuser, Straßennetzwerk).
- Input - Wie gering kann der Input für einfachen Output sein. Wie viel Input ist für einen speziellen Output nötig.
- Effizienz - Benötigte Rechenzeit für die Generierung.
- Kontrolle - Inwiefern kann der Benutzer den Output beeinflussen.
- Echtzeit - Ist es möglich, die generierte Stadt in Echtzeit zu rendern.

[1] http://www.opengl.org
[2] http://de.wikipedia.org/wiki/DirectX
[3] http://unity3d.com
[4] http://ioquake3.org/get-it/
[5] http://cryengine.com
[6] https://www.unrealengine.com

Im Kontext dieser Arbeit macht es Sinn, Realismus zu Plausibilität abzuschwächen. Da eine mcw eine abstrakte Welt darstellt und von vornherein keinen Anspruch hat, real zu wirken. Stattdessen spielt Plausibilität eine Rolle. Häuser und Straßennetze sollten zumindest dem Anspruch genügen, plausibel zu sein. Ein Gegenbeispiel hierfür wären Haustüren im 5. Stockwerk oder Straßen, die in einem See enden.

5 Anforderungen

5.1 Überblick

Mutable cube worlds eignen sich für grundsätzlich unterschiedliche Einsatzszenarien, wie sich an der Vielzahl unterschiedlicher mcw Spiele zeigt. Es kann somit PCG auf verschiedene Weise eingesetzt werden. Man könnte ein Tool entwickeln, das es Spielentwickler ermöglicht, schnell auf definiertem Raum eine Stadt zu generieren und diese nach Bedarf anzupassen. Zum Beispiel könnten Straßenzüge angepasst werden oder Gebäude neu mit anderen Parametern generiert werden, um der Vision des Designers zu entsprechen.
In dieser Arbeit werde ich jedoch auf einen anderen Bereich eingehen. In einigen der vorgestellten Spielen, wie zum Beispiel Minecraft oder Cubeworld, ist die Welt unendlich und wird auch nicht hinterher von Designern modifiziert. Eine solche grenzenlose Welt hat erheblich unterschiedliche Anforderungen und Eigenschaften, die berücksichtigt werden müssen. In den folgenden Unterkapiteln werden die, aus den Eigenschaften hervorgehenden Anforderungen und die Anforderungen an die letztendlich generierten Siedlungen festgehalten. Zudem werden Ausschlüsse definiert, die definitiv nicht umgesetzt werden.

5.2 Anforderungen

5.2.1 Anforderungen an die Welt

Da die Welt unendlich ist, kann nur jeweils ein Teil der Welt visuell dargestellt und auch nur ein Teil der Welt in einem Cache gehalten werden. Der Benutzer erwartet, dass wenn er einen Ort verlässt, diesen auch genauso vorfindet, wie er ihn verlassen hat. Daher muss eine Konsistenzbedingung für die Welt gelten. Daraus ergeben sich folgende Anforderungen:

ANF 1.0 Ein Block B(X,Y,Z) verhält sich zum Seed-Wert der Welt deterministisch.
ANF 1.1 Die Welt hat keine definierte Größe.
ANF 1.2 Die Vielfalt der Welt wird nicht durch Wiederholungen erzeugt.

ANF 1.3 Die Welt ist in Echtzeit generierbar.
ANF 1.4 Das Rendering der Welt geschieht in Echtzeit.

5.2.2 Anforderungen an die Siedlungen

Zwar können aufgrund der Blockstrukturen keine realistischen Gebäude erstellt werden, jedoch erwartet der Benutzer eine gewisse Plausibilität in Bezug auf diese Gebäude.

ANF 2.0 Ein Gebäude hat mindestens eine Tür, die nicht in einen Innenhof zeigt.
ANF 2.1 Ein Gebäude hat, mit Ausnahme von Fenstern, ein geschlossenes Dach.
ANF 2.2 Die Mauern eines Gebäudes sind, mit Ausnahme von Türen und Fenstern, geschlossen.

ANF 3.0 Das Aussehen der Gebäude hat eine hohe Variation.
ANF 3.1 Der Benutzer hat die Möglichkeit, sich in der X- und Z-Dimension über die Welt zu bewegen.
ANF 3.2 Der Benutzer hat die Möglichkeit, vor der Erstellung der Welt, die Parameter der Welt zu verändern.

5.3 Ausschlüsse

AUS 1.0 Die Innenräume von Gebäuden werden nicht generiert.
AUS 1.1 Das manuelle Entfernen und Einfügen von Blöcken wird nicht implementiert.
AUS 1.2 Die Geometrie der mcw Elemente beschränkt sich auf Würfel.

6 Konzeption

6.1 Überblick

Der Prototyp, den ich im Umfang dieser Arbeit entwickle, soll Unterschiede in den Herausforderungen und Eigenheiten von der Siedlungsgenerierung in mcw, im Vergleich zu konventionellen Welten aufzeigen. Hierzu sollen sowohl der feine sowie der grobe Sicht (siehe Kapitel 3.1) untersucht werden und eine grafische Ausgabe in Echtzeit erfolgen.
Für die grobe Sicht werde ich die Anwendung von Noisefunktionen auf Straßennetze, sowie Verteilung von Population untersuchen. In der feinen Sicht wird der Ansatz der CGA Shape Grammatik verfolgt. Hierbei werde ich eine stark vereinfachte Form implementieren und auch dazu eine einfache Gebäudegrammatik modellieren, die zum Testen der Eigenschaften ausreicht.

6.2 Aufbau des Prototypen

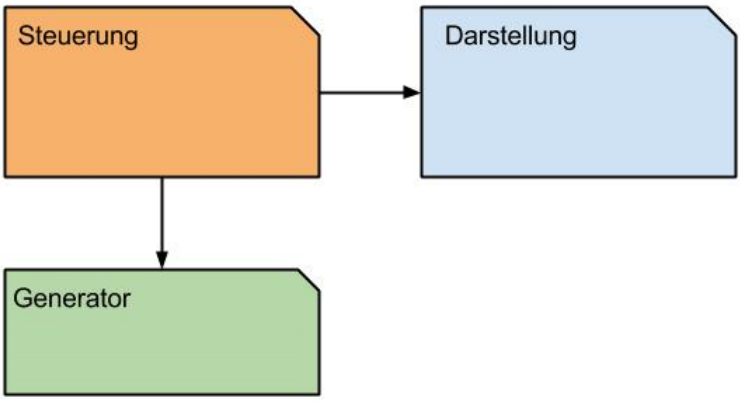

Abbildung 6.1: Aufteilung des Systems

Der Prototyp lässt sich in drei Komponenten aufteilen (siehe Abbildung 6.1). Die Steuerungskomponenten bestimmt, welche Blöcke geladen werden. Die Darstellungskomponente stellt

gegebene Blöcke grafisch dar und die Generatorkomponente bestimmt welcher Block sich an welcher Position befindet.
In meinem Prototypen ist die Generatorkomponente von der Steuerungs- und Darstellungskomponente getrennt. Die Generatorkomponente bietet ein Interface, mit der Methode getBlock(x,y,z) an, welche für eine gegebene Weltkoordinate (x,y,z) einen Blocktyp zurückgibt. Somit verhält sich die Generatorkomponente wie eine Datenstruktur und hat damit auch keine Informationen über die Steuerung.

6.3 Aufbau der Darstellungs- und Steuerungskomponente

Für die Darstellung habe ich mich für die Engine Unity entschieden. Der Hauptgrund hierfür ist, dass ich einen Teil der Umsetzung aus einem vorhergehenden Projekt verwenden kann.
In der Darstellung soll ein Ausschnitt der Welt gezeigt werden. Über die Pfeiltasten lässt sich der Ausschnitt verschieben und somit werden Teile der Welt gelöscht und neue geladen. Die Steuerungskomponente gibt dabei an, welche Blöcke genau geladen und gelöscht werden sollen.

6.4 Aufbau der Generatorkomponente

6.4.1 Zusammenspiel mehrerer Generatoren

Die Generatorkomponente kann für jeden Block der Welt bestimmen, welcher Blocktyp an dieser Stelle verwendet wird. Um diese Funktion zu gewährleisten, müssen verschiedene Generatoren zusammenarbeiten.
Hierbei bewältigt jeder Generator eine Teilaufgabe im Generierungsprozess. Je nach Generator bestehen keine bis mehrere Abhängigkeiten zu anderen Generatoren, da teilweise die Ergebnisse dieser zusammenhängen. Zum Beispiel ist die Höhe eines Hausblocks von der Höhe des Terrains abhängig.

Der Aufruf, für einen Block B(x,y,z) der Generatorkomponente, verläuft wie in der Abbildungen 6.2 gezeigt.

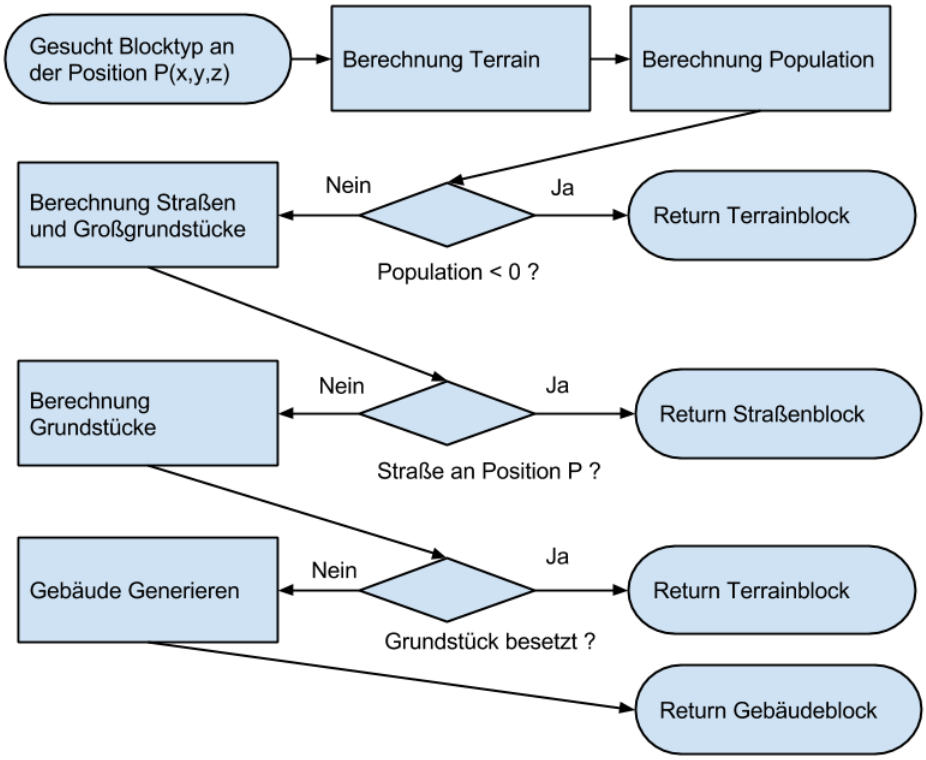

Abbildung 6.2: Ablauf der Blockbestimmung

6.4.2 Caching von Blöcken

Wie schon erwähnt, steuert die Steuerungskomponente, welche Blöcke geladen werden. Der Vorteil hierbei ist, dass der Siedlungsgenerator keine Informationen über die Anwendung benötigt. Diese Modellierung hat auch einen Nachteil, der berücksichtigt werden muss. Dieses wird anhand eines Beispiel genauer erläutert.
Wenn ein Block (X=1,Y=1,Z=1) eines Gebäudes geladen wird, muss, nach Vorbild der CGA Shape Grammar, ein Startsymbol bis hin zu den geometrischen Informationen des fertigen Hauses abgeleitet werden. Der Block (X=2,Y=1,Z=1) liegt auch in diesem Gebäude und wurde demnach auch schon berechnet. Sofern keine Zwischenspeicherung erfolgt, müsste dennoch für den Block das gesamte Gebäude neu generiert werden. Dies würde zu erheblichen, redundanten Berechnungen führen.
Eine Lösung hierfür ist, alle Blöcke, die bei der Berechnung eines Blockes entstehen, zwischenzuspeichern. Die Herausforderung hierbei ist, zu bestimmen, welche Informationen wann aus den Zwischenspeicher gelöscht werden können.

Eine Lösungsmöglichkeit ist es, das Interface der Abfrage auf ein Volumen von Blöcken zu erweitern. Hierdurch können redundante Berechnungen verringert werden, indem für jede Abfrage ein Zwischenspeicher angelegt und bei Rückgabe gelöscht wird. Da die Steuerung unabhängig von dem Generator ist, hat dieser keine Informationen, welche Abfragegröße besonders effektiv ist. So ist es möglich, durch zu kleine oder zu große Wahl der Abfragegröße, ein Gebäude mehrfach zu generieren.
Eine weitere Lösungsmöglichkeit, die im Prototypen auch so umgesetzt wird, ist das Ausnutzen von zeitlicher und räumlicher Lokalität. Es werden grundsätzlich alle generierten Blöcke zwischengespeichert. Diese Informationen im Zwischenspeicher können, nach Ablauf einer gewissen Zeit, gelöscht werden. Hierbei wird sich auf die Heuristik verlassen, das die Anwendung, die einen Block lädt, in zeitlicher Nähe auch naheliegende Blöcke laden wird. Daher habe ich mich für diese Art des Cachings entschieden, da für den Prototypen dieses Verhalten vorliegt. Zu beachten ist, dass kurze Löschintervalle zu redundanten Berechnungen führen können und zu lange Löschintervalle zu höherem Speicherbedarf führen.

6.5 Terraingenerator

Da der Fokus dieser Arbeit auf der Generierung von Siedlungen liegt, wird der Terraingenerator sehr einfach gestaltet. Es soll jedoch möglich sein, an dieser Stelle Erweiterungen vorzunehmen.
Das Terrain kann mit einer Kombination von Noisefunktionen entworfen werden. Hierzu können 2D Noisefunktionen als Höhenfunktion, oder 3D Noisefunktionen als direkte Repräsentation des Terrain verwendet werden. In diesem Prototypen soll zunächst nur eine grüne Wiese als Grundlage für die Siedlungen dienen. Somit ist der Output des Terraingenerators für alle Blöcke mit y=0 ein Grassblocktyp. Grundsätzlich wird für diese einfache Version kein Cachingverfahren benötigt. Bei der Verwendung von Noisefunktionen könnten hier die Zufallswerte/Zufallsgradienten in einem Cache gespeichert werden, da eine räumliche Lokalität besteht.

6.6 Populationsgenerator

Um der Welt Zivilisationen hinzuzufügen, werden verschiedene Siedlungen generiert, die sich in ihrer Gesamteinwohnerzahl, sowie Einwohnerdichte und Position unterscheiden. Hierbei sollen diese drei Eigenschaften durch eine Methode gelöst werden.
Um Unterschiede in der Einwohnerdichte zu erreichen, eignen sich Noisefunktionen. Das

Problem hierbei ist, dass nun die ganze Welt von einer einzigen großen Siedlung bedeckt wird. Um verschiedene Siedlungen zu erhalten und somit Position und Gesamteinwohnerzahl festzulegen, wird nun die Noisefunktion, unter einem gewissen Schwellenwert, auf Null gesetzt. Dadurch entstehen nun Siedlungsgrenzen. Dies hat auch zur Folge, dass die Einwohnerdichte in Richtung des Zentrums der Siedlungen steigt.
Die Einwohnerwerte können nun, als Informationen für den Straßengenerator, verwendet werden. Hinzu kommt, dass der Gebäudegenerator diese Informationen nutzen kann, um Gebäudegrößen und Typ zu bestimmen.

6.7 Straßengenerator

In dieser Arbeit möchte ich untersuchen, wie sich eine Noisefunktion, zum Generieren von Straßen, eignet. Das entstehende Straßennetz bestimmt die Struktur der Siedlung und darüber hinaus auch die Position der einzelnen Grundstücke.
Um eine grundlegende Struktur und eine Basis zur Berechnung der Grundstücke zu haben, wird zunächst ein quadratisches Straßenmuster gelegt. Die Quadrate haben hierbei eine feste Größe und werden vom Weltursprung aus definiert. Somit ist die Konsistenz der Straßen und Grundstücke gewährleistet. Falls ein Block, der sich auf diesem Straßengitter befindet, eine Einwohnerzahl größer Null hat, wird ein Straßenblock an dieser Stelle platziert.
Über eine Noisefunktion werden nun radial verlaufende Straßen generiert. Da die Verteilung der Einwohner eine Ringform hat, wird diese als Ausgangspunkt verwendet. Die Idee ist es, die Funktion nur bei einem bestimmten Einwohnerwert zu zeichnen. Das Problem hierbei ist, dass die Breite der Straße abhängig vom Grad der Steigung ist. Um Straßen gleichbleibender Größe zu generieren, werden die Einflusswerte abhängig von ihrer Größe auf Null oder Eins gesetzt. Somit ist gewährleistet, dass der Grad der Steigung, bei höheren Werten der Funktion, sich sehr ähnlich ist (siehe Abbildung 6.3).

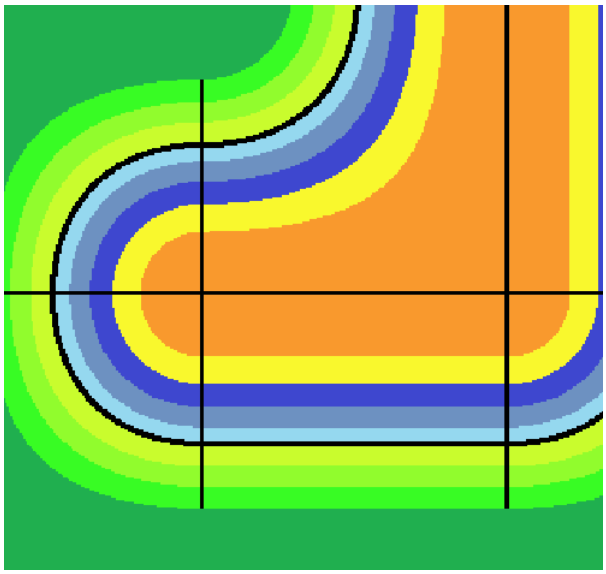

Abbildung 6.3: Top-Down Sicht auf die Siedlungsstruktur. Die schwarzen Linien symbolisieren die Straßen. Der Farbverlauf zeigt die unterschiedlichen Einwohnerwerte, die zum Zentrum hin steigen

Diese Art der Modellierung führt jedoch dazu, dass die Funktion zum Bestimmen der Einwohnerzahl an die Straßenfunktion angepasst werden muss. Falls dies nicht getan wird, verlaufen die Straßen nicht im Zusammenhang mit den Siedlungen. Dies hat zur Folge, dass sich die unterschiedlichen Einwohnerverteilungen der Siedlungen nach dem selben Muster aufbauen.

6.7.1 Großgrundstücke

Großgrundstücke sind Sammlungen von quadratischen Grundstücken, die von einem Straßenquadrat umschlossen sind. Um festzustellen, welche Grundstücke zum Generieren von Gebäuden genutzt werden können, muss zunächst festgestellt werden, durch welche Grundstücke eine Straße verläuft.

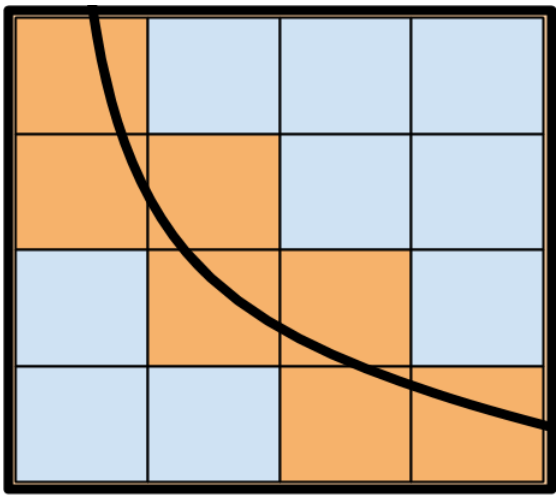

Abbildung 6.4: Eine radiale Straße verläuft durch ein Großgrundstück. Die orangen Felder sind als besetzt markiert Grundstücke

Die quadratischen Straßenverläufen können hiervon ausgeschlossen werden, da Großgrundstücke nur innerhalb dieser definiert sind. Die radialen Straßen, die von der Straßenfunktion generiert werden, können jedoch durch ein Großgrundstück verlaufen. Um nun festzustellen welche Grundstücke betroffen sind, werden die Einwohnerwert an eine den Eckpunkten berechnet und mit den Werten der Straßenfunktion verglichen. Wird ein Schwellwert an einer dieser Eckpunkten überschritten wird dieser als 'Besetzt' interpretiert (siehe Abbildung 6.4).

6.8 Grundstücksgenerator

Grundstücke bilden die Grundlagen für die Gebäude und definieren ihre maximale Größe auf der X und Z Achse, sowie die Position in der Welt. Die Großgrundstücke werden gleichmäßig in quadratische Grundstücke aufgeteilt. Es ist denkbar, dass man nun Grundstücke innerhalb eines Großgrundstückes verbindet, sodass größere Rechtecke bzw. Quadrate entstehen. In diesem Prototypen wird jedoch vereinfachend jedes Grundstück für sich betrachten.

6.9 Gebäudegenerator

Das Generieren von Gebäuden lehnt sich an die Arbeit von Müller u. a. (2006) an. Zunächst wird für das Grundstück das Ausmaß über zwei Vektoren bestimmt. Der erste Vektor gibt die Position in Weltkoordinaten an. Der zweite gibt die Größe an. Das Gebäude kann nur innerhalb der Größe, die von diesem Vektor angegeben ist, generiert werden.

Um Gebäude in einem sinnvollen Kontext zur Stadt darzustellen, werden Metadaten für jedes Gebäude generiert oder ermittelt. Zum Beispiel verfügt jedes Gebäude über einen Einwohnerwert. Es können noch weitere Parameter, wie zum Beispiel Wohlstand, Straßenanbindung, Nähe zu Schulen, Nähe zu Kirchen, etc. berücksichtigt werden. Für den Prototypen werde ich mich auf die Einwohnerzahl beschränken.
Um ein einheitliches Äußeres zu generieren, werden bestimmte Blocktypen als Materialien für das Gebäude bestimmt. Dies kann zufällig oder auf Basis der Metadaten geschehen. Ein Gebäude speichert in diesem Kontext mehrere Blocktypen für bestimmte Konstrukte. Ein Beispiel hierfür wäre: Mauerwerk1 -> Holz , Mauerwerk2 -> Stein, Dach -> Ziegel, Fenster -> Gerahmt.

Sind Metadaten und Material für das Gebäude bestimmt, wird die Geometrie des Gebäudes generiert. Hierbei wird eine Grammatik, die ähnlich wie die CGA Shape Grammar von Müller u. a. (2006) funktioniert, eingesetzt.
Zu jedem Symbol werden zwei Vektoren, genauso wie bei dem Grundstück, die Ausmaße und Position definiert. Die Modellierung der Grammatik erfolgt über Ableitungen der Symbole und Manipulierung der Vektoren. Hierzu werden Operatoren definiert.
Bei näherer Betrachtung fällt auf, dass einige Operationen, die für die CGA Shape Grammar definiert sind, für den Zweck der mcw angepasst werden müssen. Durch die Split Operation lässt sich im Normalfall eine Form in zwei gleichgroße Hälften teilen. Dies ist wichtig, um die Symmetrie des Gebäudes zu gewährleisten. Der Unterschied zur mcw ist hierbei, dass die kleinste Einheit ein Block ist. Somit kann eine Kette von fünf Blöcken nicht in zwei gleich große Hälften geteilt werden. Somit müssen beim Teilen von Formen gerade und ungerade Zahlen berücksichtigt werden.
Die Componentsplit Operation, die eine Form in Formen niederer Dimension teilt, wird im Falle einer mcw nicht benötigt. Der Blocktyp, bestimmt welche Textur verwendet wird. Somit wird ein Gebäude nur von dreidimensionalen Objekten, in diesem Fall die Blöcke, zusammengestellt. Es ist dennoch sinnvoll, eine ähnliche Operation zu definieren, die eine Form in einen Innenraum und eine Hülle trennt. Ein Anwendungsbeispiel hierfür ist die Trennung von Innenräumen und Wänden. Das Problem, das sich hierbei ergibt, sind die Schnittstellen der Ecken, da sich zum Beispiel die Nord- und Ostwand eine Blockreihe teilen. Um eine einheitliche Trennung zu schaffen, wird beim Splitten einer Form angegeben, welche Seite die größere Blockmenge erhält. Diese Eigenheiten zeigen auch, dass die Modellierung einer Gebäudegrammatik anders gehandhabt werden muss, als bei der CGA Shape Grammatik.

6.9.1 Gebäudegrammatik

Um eine möglichst ergiebige Gebäudegenerierung zu erlangen, müssen die Gebäude auf einer abstrakten Ebene modelliert werden, die wiederum auf flexible, konkrete Terminalsymbole zurückgreift. Hierzu werde ich zunächst einfache Terminalsymbole für die Grammatik definieren: Türen, Fenster, Wände, Böden, sowie Dächer. Für diese Einheiten werden exemplarisch Terminalsymbole definiert. Es wäre jedoch möglich, jederzeit diese Listen zu erweitern.

6.9.2 Terminalsymbole

Bei der Definition müssen gerade und ungerade Symmetrien berücksichtigt werden, sowie die Skalierung von relativen Blöcken. Im Folgenden werden alle Terminaltsymbole in den Abbildungen skizziert. Abbildung 6.5 zeigt die Legende für diese Skizzen, dabei beziehen sich die jeweiligen Typen auf die Blocktypen, die in den Metadaten des konkreten Gebäudes gespeichert sind.

Abbildung 6.5: Legende der folgenden Skizzen

Türen

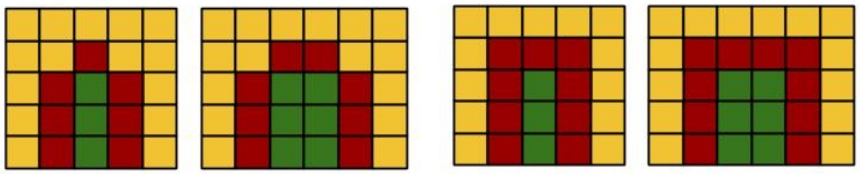

Abbildung 6.6: Von links nach rechts: Tür1 ungerade, Tür1 gerade, Tür2 ungerade, Tür2 gerade.

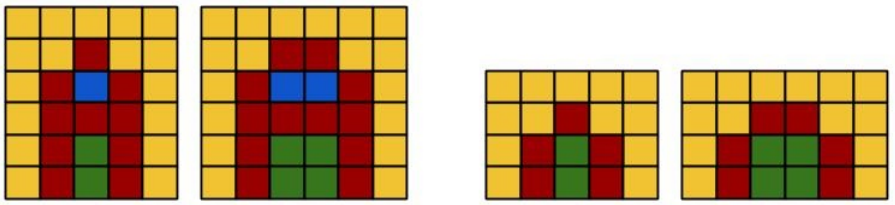

Abbildung 6.7: Von links nach rechts: Tür3 ungerade, Tür3 gerade, Tür4 ungerade, Tür4 gerade.

Abbildungen 6.6 und 6.7 zeigen die vier Türtypen. Der Rahmen aus Wandblöcken skaliert hierbei mit der Größe. Die Blöcke, welche Tür ausmachen, sind in der Mitte unten zentriert. Je nach Breite wird eine der gerade/ungerade Optionen gewählt.

Fenster

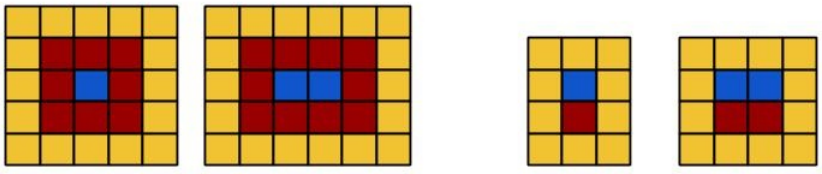

Abbildung 6.8: Von links nach rechts: Fenster1 ungerade, Fenster1 gerade, Fenster2 ungerade, Fenster2 gerade.

Abbildung 6.8 zeigt die zwei Fenstertypen. Der Rahmen aus Wandblöcken skaliert ebenfalls mit der Größe. Die Blöcke, welche das Fenster ausmachen, werden in der Mitte zentriert.

Boden und Wand

Abbildung 6.9: Links: Wand. Rechts: Boden.

Abbildung 6.9 zeigt die Anordnung einfacher Bodenblöcke und Wandblöcke. Diese Terminalsymbole füllen die gesamte Größe mit dem entsprechendem Blocktyp.

Dächer

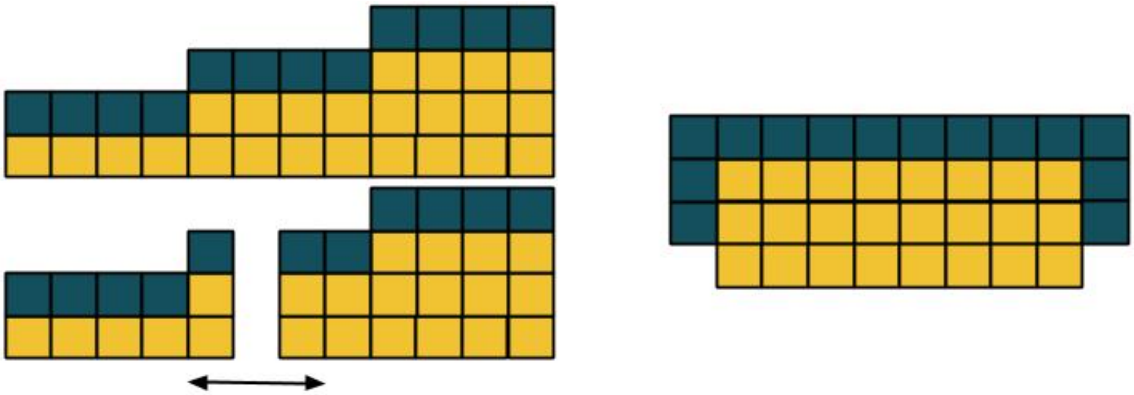

Abbildung 6.10: Oben links: Pultdach seitlich. Rechts: Front. Unten links: Skalierung wird im Mittelteil eingefügt.

Abbildung 6.10 zeigt den Aufbau eines Pultdaches. Das Pultdach hat eine Steigung von eins zu vier und einen Überhang an den Seiten. Das Pultdach wird in der Mitte gestaucht, falls die Länge kein Vielfaches von vier ist.

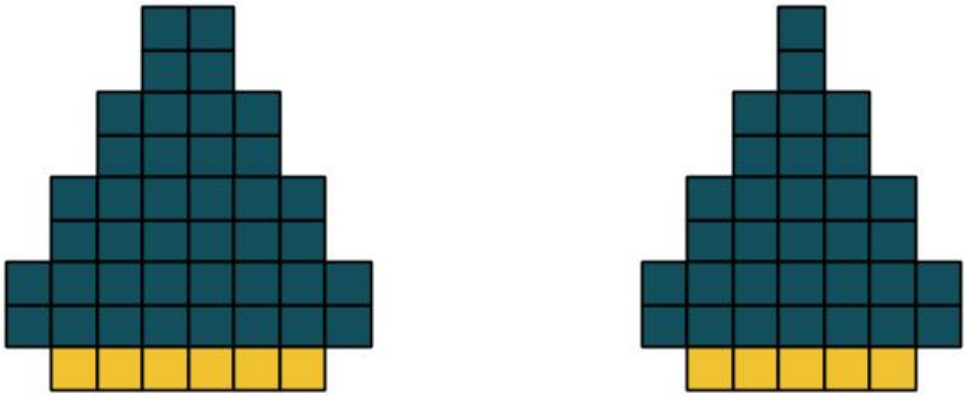

Abbildung 6.11: Links: Turm gerade. Rechts: Turm ungerade.

Abbildung 6.11 zeigt den Aufbau eines Turmdachs. Das Turmdach hat eine Steigung von zwei zu eins. Alle Seiten laufen in einer Spitze zusammen. Auch hier wird zwischen geraden und ungerade Breiten unterschieden. Das Turmdach ist für quadratische Dächer ausgelegt.

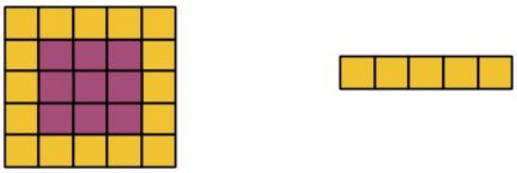

Abbildung 6.12: Flachdach. Rechts: Ansicht Oben. Links: Ansicht seitlich.

Abbildung 6.12 zeigt den Aufbau eines Flachdaches. Das Flachdach verwendet den Dachblocktyp zwei für die Innenseiten und wird mit Wandblöcken umrahmt. Die Innenseiten skaliert dabei mit der Größe.

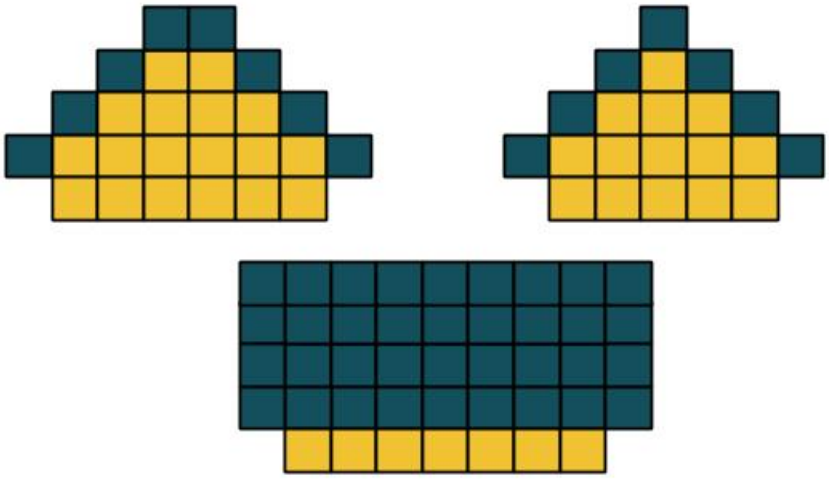

Abbildung 6.13: Satteldach. Links oben: Frontansicht gerade. Rechts oben: Frontansicht ungerade. Unten: Seitenansicht

Abbildung 6.13 zeigt den Aufbau eines Satteldachs. Das Satteldach hat einen Überhang und eine Steigung von eins zu eins. Bei dem Satteldach wird zwischen geraden und ungeraden Breiten unterschieden.

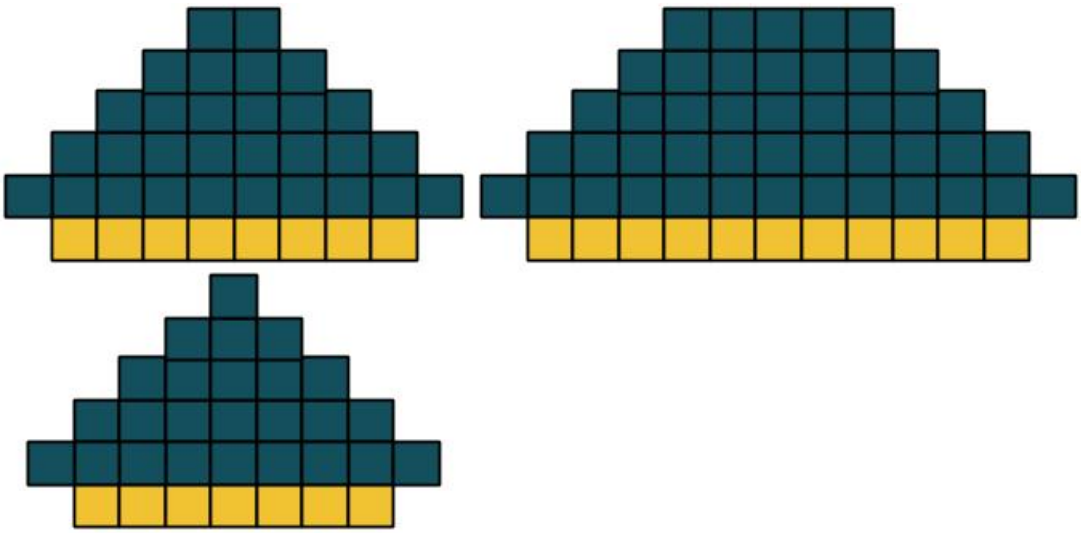

Abbildung 6.14: Walmdach. Links oben: Frontansicht gerade. Rechts oben: Frontansicht ungerade. Unten: Seitenansicht

Abbildung 6.14 zeigt den Aufbau eines Walmdachs. Das Walmdach hat einen Überhang in alle Richtungen. Die Steigung zur Dachmitte ist eins zu eins aus allen Richtungen.

7 Umsetzung

7.1 Architektur

Die Architektur teilt sich in das Cubeframework und den Siedlungsgenerator. Das Cubeframework ist für die Steuerung und Darstellung zuständig und kümmert sich um das Laden und Entfernen von Blöcken, sowie eine effiziente Darstellung.
Der Siedlungsgenerator ist über das IWorld Interface komplett von der Unityengine entkoppelt. Das IWorld Interface erlaubt es für einen beliebigen Block(X,Y,Z) einen Blocktyp zu erhalten. In dem Siedlungsgenerator ist eine einfache Umsetzung eines Grammatiksystems enthalten. Diese findet jedoch noch keine Verwendung, da keine Gebäudegrammatik existiert.

7.2 Systemstart

Durch die Unity Methode Start() werden über das Starterscript alle Abhängigkeiten gesetzt und die Initialisierung gestartet. Hierbei werden die, in der Klasse Constants definierte Größe, des Weltausschnittes vom Chunkloader geladen.

7.3 Cubeframework

Nach der Initiierung kann der Benutzer sich, mit den Pfeiltasten, über die Welt bewegen. Hierzu werden die entsprechenden Blockgruppen gelöscht, beziehungsweise geladen. Um eine effiziente Darstellung zu ermöglichen, muss ein optimiertes Dreiecksnetz generiert werden. Dies werde ich im nächsten Unterkapitel genauer darlegen.

7.4 Meshgenerator

Um mehr Blöcke mit höherer FPS Zahl zu rendern, habe ich die Dreiecksnetze von Blöcken zu einem Objekt zusammengefasst, um somit die Anzahl der Batchingvorgänge zu verringern, da diese einen Overhead pro Vertex erzeugen. Hierbei werden jedoch nicht alle Blöcke der Welt in einem Objekt gebündelt, da Unity eine Obergrenze von 65000 Vertices pro Objekt vorgibt.

Somit werden Blöcke in Blockgruppen zusammengefasst.
Da das Dreiecksnetz nun manuell generiert wird, können auch direkt Optimierungen angewendet werden. Die Seiten der Blöcke, die einen soliden Block als Nachbarn haben, müssen nicht generiert werden. Um diese Optimierung umzusetzen, speichert jede Blockgruppe einen Blocktyp zu jeder seiner Positionen (X,Y,Z) -> Blocktyp. Die Blocktypen sind als Integer kodiert. Eine Null steht für keinen Block. Eine negative Zahl für einen transparenten Block. Eine positive Zahl steht für einen opaken Block. Eine Blockseite ist somit Teil des Dreiecksnetzes der Blockgruppe, wenn der Blocktyp, des derzeit betrachteten Blocks, kleiner oder größer als null ist und der Nachbarblock einen Blocktyp kleiner eins hat.
Verändert sich eine Blockgruppe, wird der Blocktyp an der entsprechenden Stelle ersetzt und das Dreiecksnetz angepasst. Beim Laden werden jedoch aus Effizientsgründen zunächst alle Blöcke für eine Blockgruppe geladen und danach erst das Dreiecksnetz generiert, da das Neugenerieren des Dreiecksnetzes für jeden einzelnen Block zu teuer wäre.

7.5 Siedlungsgenerator

Der Siedlungsgenerator koordiniert den Generierungsprozess über die Klasse WorldGenerator. Von dort aus wird der TerrainGenerator, CityGenerator und BuildingGenerator angesteuert.

7.6 Grammatiksystem

Das Grammatiksystem stellt ein sehr einfaches Ersetzungssystem dar. Eine Klasse 'Symbol' definiert die Grundeingenschaften von Symbolen. Hierzu zählt die Definition als Terminal- oder Nichtterminalsymbol, sowie die Positions- und Größenvektoren.
Die Methode Replace wird von den konkreten Symbolen implementiert und gibt Symbole zurück, die durch die Ersetzung erzeugt werden.
Das Regelsystem erhält ein Symbol als Startsymbol und leitet die Symbolkette solange ab, bis ausschließlich Terminalsymbole in ihr enthalten sind. Hierbei ist die Reihenfolge der Ableitung irrelevant, da es sich um ein kontextfreies Grammatiksystem handelt und immer alle Ableitungsschritte durchgeführt werden.

8 Diskussion

8.1 Zusammenfassung

Mutable cube worlds bieten für Gamedesigner ähnliche Vorteile wie die PCG. Diese beiden Methoden erlauben es, mit wenig Aufwand, Spiele mit hoher Komplexität und Vielfalt zu erstellen.
In der Untersuchung vom Kapitel Stand der Technik zeigt sich, dass Grammatiken, wie zum Beispiel L-Systeme, sich für die Generierung von Gebäuden, sowie Straßennetzen sehr gut eignen. Die Verwendung von Noisefunktionen findet sich eher im Bereich der Terraingenerierung. Dabei spielen Noisefunktion eine große Rolle beim Generieren von mcw.
In der Konzeption konzentriere ich mich auf mcw mit speziellen Eigenschaften. Diese Welten sind unendlich. Dies stellte sich, bei gleichzeitiger Konsistenzerhaltung , als erhebliche Herausforderung. Insbesondere gilt dies in Verbindung mit komplexeren Strukturen wie Gebäuden oder Siedlungen.
Ich habe gezeigt, wie abgewandelte Noisefunktionen zur Straßengenerierung genutzt werden können. Diese Umsetzung wird den Anforderungen zur konsistenten Generierung gerecht, erweist sich jedoch als unpraktisch und ineffektiv. Dies werde ich näher im Unterkapitel '8.3 Straßennoise' erläutern.
Um eine Darstellungsmöglichkeit in Echtzeit zu gewährleisten, habe ich eine Optimierung der Dreiecksnetze in Unity implementiert. Diese erweist sich als ein Vielfaches effizienter als eine naive Lösung. Dies wird genauer im Kapitel '8.4 Performance Analyse des mcw Adapters' erläutert.
Da die Anforderungen der unendlichen konsistenten Welt nur schwer in Verbindung mit den komplexen Strukturen der Siedlungen gebracht werden konnten, umfasst die Umsetzung nur das Generieren der Straßen und der grundlegenden Struktur für Städte.

8.2 Architektur

Durch die Trennung der Steuerung von der Generatorkomponente kann der Generator auch für anderen Anwendungsfälle eingesetzt werden, ohne das er angepasst werden muss. Zudem

ist der Generator von der Darstellungkomponente unabhängig. Somit ist es möglich über einen Adapter andere Darstellungsframeworks zu verwenden.
Jede zusammenhängende Struktur, wie zum Beispiel Gebäude, Straßennetze oder Siedlung, muss, genauso wie das Terrain, konsistent sein. Ein generiertes Haus muss von links nach rechts geladen, genauso aufgebaut sein, wie umgekehrt. Um dies zu gewährleisten, muss es möglich sein, von jedem Block des Gebäudes auf das Gebäude als Gesamtobjekt zu schließen.
Die Trennung der Steuerungskomponente von dem Generator erschwert dessen Implementierung. Der Generator hat keine Kontrolle über die Steuerung. Mit dieser Kontrolle wäre es möglich, besser auf spezielle Fälle zu reagieren und Performanceoptimierungen vorzunehmen. Zum Beispiel fällt das Zwischenspeichern von Blöcken weg, da Strukturen komplett generiert und an die Darstellungskomponente weitergegeben werden könnten. Bei der Trennung von Steuerung und Generator müssen zudem alle Fälle betrachtet werden, die durch die Steuerung entstehen könnten. Neben dem erhöhten Implementierungsaufwand kann dies auch stellenweise negativen Einfluss auf die Performance haben.
Zu erwähnen ist, dass die Lösung, unabhängig von der Trennung der Komponenten erreicht werden kann. Die Trennung von der Steuerung und dem Generator ermöglicht eine bessere Kapselung, zu Kosten von leichten Performanceverlusten.

8.3 Straßennoise

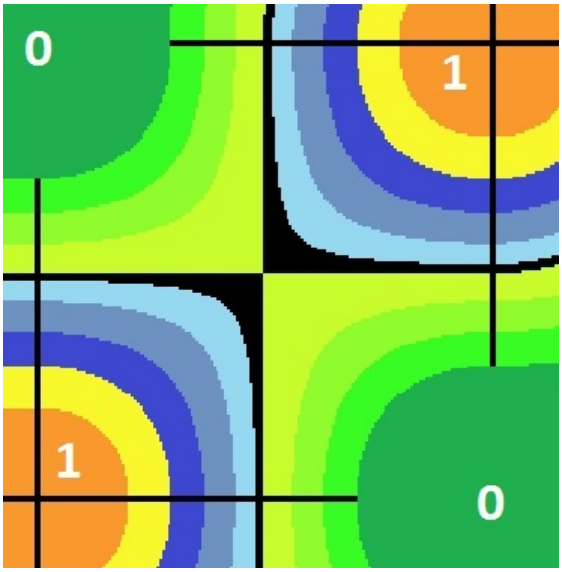

Abbildung 8.1: Topdown Sicht eines Straßennoiseartefaktes. Der Farbverlauf skizziert die Einwohnerzahl. Die schwarzen Blöcke sind die Straßen

Die von mir vorgestellte Straßennoisefunktion kann ohne weiteres in unendlichen Welten eingesetzt werden. Allerdings ist sie nicht besonders flexibel und kann auch nicht auf andere Strukturen, wie zum Beispiel Einwohnerverteilung angepasst werden. Somit findet die Anpassung umgekehrt statt und alle Elemente, die mit den Straßen zu tun haben, müssen sich auf diese einstellen.
Die Straßennoisefunktion hat mit dem ursprünglichen Konzept der Noisefunktionen wenig zu tun. Sie bildet lediglich ein Gitter aus binären Zahlen ab, wobei größere, zusammenhängende Gebiete mit jeweils Einsen oder Nullen entstehen.
Die Menge der möglichen Kurven ist endlich und relativ klein. Man könnte diese Kurven als Template definieren. Durch sinnvolles Aneinanderreihen der Templates könnte der selbe Straßenverlauf generiert werden. Dies würde den Rechenaufwand, im Vergleich zur Straßennoiselösung, reduzieren, da keine Interpolationsberechnungen gemacht werden müssen. Insofern stellt die Straßennoisefunktion nicht die einfachste Lösung für dieses Problem dar.
Hinzu kommt, dass die Straßennoisefunktion sich in ihrer Anwendung auf radiale Straßenmuster reduziert. Somit bietet sie keine Variation bezüglich der Straßenmuster.
Es ist weiterhin möglich, dass die Straßennoisefunktion Artefakte erzeugt (siehe Abbildung 8.1). Hierbei wird über die Interpolation die Straße verzerrt, da sich zwei Gebiete mit einer hohen

Einwohnerzahl quer gegenüber stehen. Um diesen Effekt zu erzeugen, dürfen die gespiegelten Eckpunkte keine Einwohner haben.

Ich komme zu den Schluss, dass mein Konzept zur Straßengenerierung, über Noisefunktionen, unerwartet viele Nachteile hat. Sie bietet kaum Möglichkeiten zur Variation und schränkt alle anderen Bereiche, die mit ihr zusammenhängen, ein. Die Straßengenerierung alleinig auf die Berechnung von Funktionswerten zurückzuführen, erlaubt zu wenig Kontrolle, um eine vernünftige Siedlungsstruktur zu generieren. Es wäre jedoch denkbar, eine grammatikbasierte Lösung mit geeignet modellierten Noisefunktionen zu verknüpfen.

8.4 Performance Analyse des mcw Adapters

Für die folgende Analyse wurde folgende Hard- und Software verwendet: Windows 7 64 bit Servicepack 1, Intel i5-2500, 8 GB RAM, AMD Radeon HD 6900 series und Unity 4.1.2f1.
Um eine möglichst große Sicht auf Siedlungen zu ermöglichen, müssen viele Blöcke flüssig dargestellt werden. Ich habe eine Optimierung durch manuelle Meshgenerierung implementiert. Diese wurde im Vergleich mit Unity Standardblockobjekten getestet.
In diesem Test werden Blockreihen mit einer Länge von 200 Blöcken generiert. Nach der Generierung von 200 x 2 x 200 (80000) Blöcken erreicht der Renderer nur noch 6,2 Frames pro Sekunde (FPS), wie in Abbildung 8.2 zu sehen ist.

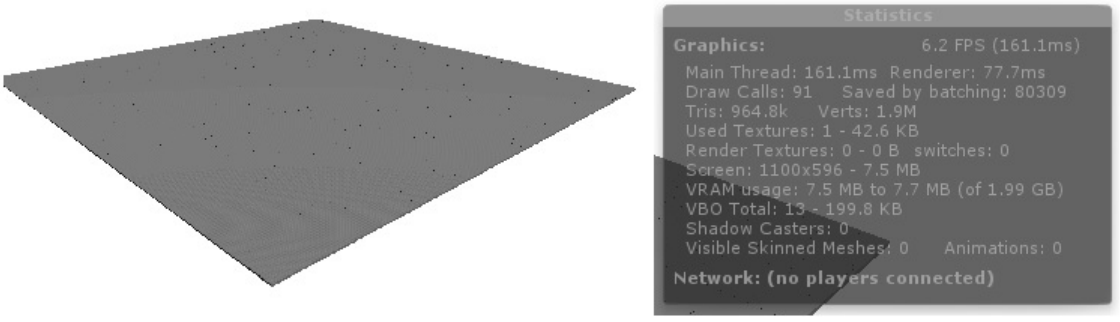

Abbildung 8.2: Visuelle Darstellung der Blöcke, ohne Optimierung

Um die Ausmaße der Blockmenge einschätzen zu können, nehme ich für einen Block einen Maßstab von einen Meter Kantenlänge an. Damit ist es möglich die Grundfläche von 100 Grundstücken der Größe $800m^2$ darzustellen. Hierbei ist zu bedenken, dass dies eine zweidimensionale Anschauung wäre. Höhenunterschiede, sowie Gebäude etc. wären im Kontingent

der 80000 Blöcke somit nicht enthalten. Hinzu kommt, dass 6,2 FPS zu wenig für eine Echtzeitdarstellung sind. Das einfache Erstellen von Blöcken ist somit, für die Anforderungen an den Prototypen, unzureichend.

Der Ursache der niedrigen FPS Zahlen vermute ich im dynamischen Batching von Unity. Um draw calls[1] zu minimieren, fasst Unity automatisch gleichartige Objekte zusammen. Diese Funktion erzeugt einen Overhead pro Vertex.
Um den worst-case der manuellen Meshgenerierung zu Testen, wird ein Blockfeld generiert, welches möglichst viele sichtbare Blockseiten hat. Hierzu wird ein Karomuster auf 25 unabhängigen Plattformen generiert. Für diesen Test sind demnach 200 x 25 x 200 Blöcke sichtbar (Abbildung 8.3). Im Vergleich hierzu wird ein Testlauf mit einer besseren Blockverteilung für das manuelle Meshgenerierungsverfahren verwendet. Dabei werden 200 x 50 x 200 Blöcke ohne Lücken dargestellt (Abbildung 8.4).

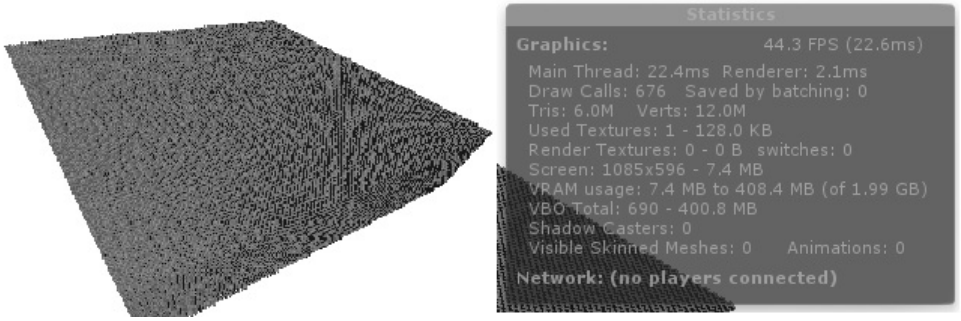

Abbildung 8.3: Visuelle Darstellung der Blöcke mit Optimierung. Worst-case.

[1] http://docs.unity3d.com/Manual/DrawCallBatching.html

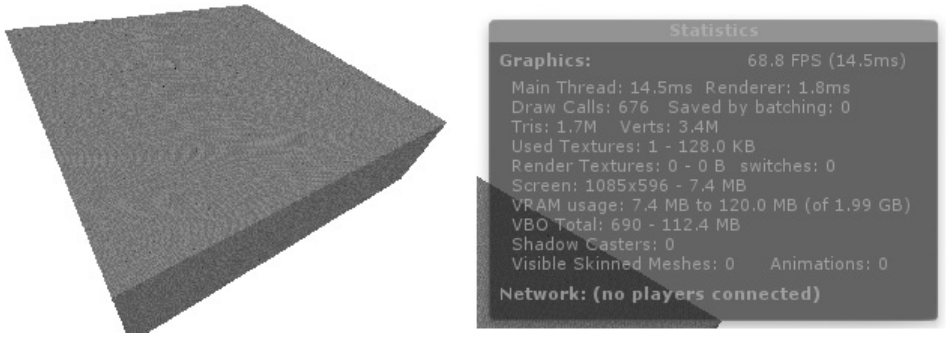

Abbildung 8.4: Visuelle Darstellung der Blöcke, mit Optimierung. Für die Optimierung besserer Fall.

Der worst-case Test zeigt 44 FPS auf 1.000.000 Blöcken. Im Vergleich zur Unity Block Lösung ist dies mehr als das 12-fache an Blöcken. Der Test mit der besseren Blockverteilung erreicht 68 FPS auf 2.000.000 Blöcken.
Durch die Optimierung des Dreiecksnetzes sind die Ausmaße der Höhenunterschiede weniger relevant, da nur noch Blöcke an der Oberfläche betrachtet werden.

8.5 Bewertung nach Kelly und McCabe (2011)

In diesem Abschnitt wird die Lösung des Prototypen nach den Bewertungskriterien, die in Kapitel 4.7 'Bewertung von generierten Städten' vorgestellt wurden, untersucht. Da für das Konzept keine vollständige Umsetzung vorliegt, lassen sich nur über die folgenden Punkte Aussagen treffen.

Skalierung

Generell lassen sich beliebig große Städte generieren, jedoch zu jedem Zeitpunkt nur ein Ausschnitt.

Variation

Durch die Streetnoisefunktion lassen sich verschieden geformte Städte generieren. Das Schema hinter der Generierung ist jedoch leicht zu erkennen und bietet innerhalb dieses Schemas keine Variation. Dies gilt auch für die Grundstücksverteilung, da diese vom Großgrundstück auf feste Positionen zurückzuführen sind.

Echtzeit

Die Performanceanalyse zeigt, dass eine Darstellung in Echtzeit möglich ist.

8.6 Unity als mcw Framework

Unity als mcw Framework erweist sich als problematisch. Die Begrenzung der maximalen Vertices, von 65000 zu einem Objekt, verhindert eine bessere Kontrolle über Blockgruppierungen und somit über genauere Optimierung der FPS. Hinzu kommt, dass sowieso zunächst eine Optimierung gemacht werden muss. Dies kann jedoch auch für andere Frameworks gleichermaßen gelten.
Durch das ständige Generieren von Blöcken steigt der Verbrauch von Grafikkartenspeicher stetig an. Normalerweise würde man sich deswegen um das Freigeben von Speicher kümmern. Der Grafikkartenspeicher wird in Unity ausschließlich beim Laden einer neuen Szene geleert. Die einzig mögliche Lösung, das Neuladen der Szene zum alleinigen Zweck der Speicherfreigabe, ist keine nutzerfreundliche Lösung. Da die unendliche Welt nahtlos verläuft, wäre das neu Laden der Szene nicht optimal.

8.7 Fazit

Mein ursprünglich gesetztes Ziel der vollständigen, unendlichen, konsistenten Echtzeitgenerierung von Siedlungen in einer mcw wurde leider nicht in seiner Gesamtheit erreicht. Aufgrund des begrenzten Umfangs und der Probleme, diese Anforderungen, war es nicht möglich die Umsetzung zu vollenden.
Die angesprochenen Probleme werfen jedoch auch interessante Fragen auf. Für das Generieren von komplexen Strukturen fehlt eine überzeugende Lösung, die in der Lage ist sehr ästhetische und detailierte Siedlungen zu erzeugen.
Die Verwendung von mutable cube worlds in Verbindung mit PCG hat nicht die erwarteten Vereinfachungen mit sich gebracht, da die grundsätzlichen Schwierigkeiten die gleichen bleiben.
Ich bin der Meinung, dass es dennoch möglich ist eine zufriedenstellende Lösung für diese Fragestellung zu finden. Die entstehenden Herausforderungen sind spannend und lehrreich. Einige mögliche Ansätze für das weitere Vorgehen wird im letzten Kapitel erörtert.

9 Ausblick

9.1 Begrenzte Welten

In dieser Arbeit stehen unendliche Welten im Fokus. Im Kapitel Analyse wurden teilweise Spiele vorgestellt, die auch mit begrenzten mcw Leveln arbeiten.
In diesem Zusammenhang könnten andere Algorithmen entworfen werden, da dort andere Anforderungen gelten. Zum Beispiel müssen die Level nicht on-demand generiert werden, sondern vorab berechnet werden. Dies erlaubt auch aufwendige Algorithmen zu verwenden. Durch die Begrenzung der Welt, können andere Optimierungen der Renderverfahren interessant sein. Hinzu kommt, dass die Konsistenz der Welt einfacher zu realisieren ist, da ein Level meist vollständig oder in vordefinierten Stücken geladen werden kann.

9.2 Straßen und Siedlungsstruktur

Da sich die Straßennoisefunktion als schlechte Grundlage für das Generieren von Siedlungen erwies, wäre der nächste Schritt, eine neue Grundlagen für die Siedlungsgenerierung zu schaffen.
Die Berechnung von Einwohnerzahlen über eine Noisefunktion, könnte mit einer grammatikbasierten Straßengenerierung verbunden werden. Hierbei stellt sich die Herausforderung, eine performante Lösung zu finden, die eine schnelle Auswertung der Grammatik, sowie einen Mechanismus zum konsistenten Generieren in der unendlichen Welt ermöglicht. Notwendigerweise müsste das Verteilen von Grundstücken für solch eine Lösung überarbeitet werden.

9.3 Einbezug des Terrains

Für die Konzeption habe ich vereinfacht ein flaches Terrain angenommen. Flüsse, sowie Seen und hohe Berge müssen bei der Generierung von Siedlungen berücksichtige werden, wenn diese eine Rolle spielen sollen.
Hierbei sollten Gebäude, und Straßen nicht Unterwasser verlaufen und die Höhenunterschiede

sollten bei der Einwohnerverteilung, Gebäudeverteilung, sowie dem Straßenverlauf eine Rolle spielen.

Das Terrain kann auch mit dem Siedlungsgenerator zusammenspielen. Es könnten Siedlungen bevorzugt an Flüssen platziert werden. Spezielle Strukturen, die an das Terrain gebunden sind, wie zum Beispiel Brücken, Bergwerke, oder Häfen sind somit auch möglich.

9.4 Stadtgrammatik und Stadtmodellierung

In der Konzeption habe ich beispielhaft Terminalsymbole definiert. Ein weiterer Schritt wäre nun, eine konkrete Grammatik zu entwickeln, die eine Vielzahl von Gebäuden und verschiedenen Gebäudetypen generieren kann.
Auch die Verteilung der Gebäude in der Stadt sollte über den Generator bestimmt werden. Zum Beispiel die Verteilung von Schulen oder Kirchen innerhalb einer Stadt. Des weiteren kann man die Stadt um Parameter, wie zum Beispiel Kriminalitätsrate, Wohlstand oder Industrie-/Gewerbegebiet, für die Bestimmung von Gebäuden und deren Metadaten erweitern.

9.5 Innenräume und kleine Strukturen

Die mcw bietet, im Gegensatz zu kontinuierlichen Welten, einen großen Vorteil bei der Generierung von Innenräumen. Bei kontinuierlichen Welten besteht ein Haus aus einer Dreiecksnetzfassade. Um Innenräume zu generieren, bedarf es einer Erweiterung des Dreiecksnetzes. Dies ist keine triviale Aufgabe.
Die Gebäude in mcws haben den Vorteil, dass sie aufgrund ihres Aufbaus schon Innenräume besitzen. Es wäre nun möglich eine Zimmeraufteilung zu generieren und sogar diese Zimmer mit entsprechenden Möbeln auszustatten.

Auch die Generierung von Bäumen innerhalb einer Stadt, sowie Mülleimer, Bushaltestellen und Straßenlampen wären möglich. Hierbei ist ein großer Vorteil, dass die Positionierung und Ausrichtung dieser kleinen Elemente dank des Blockrasters einfacher ist.

9.6 Benutzerkontrolle

Die in dieser Arbeit vorgestellte Konzeption berücksichtige kaum eine Benutzerinteraktion. Die Benutzerkontrolle könnte dahingehen erweitert werden, dass der Benutzer vorab Parameter, wie Wohlstandsverteilung, Religiosität, etc., für Städte bestimmen kann. Auch für Gebäude an sich wäre eine direkte Ansteuerung möglich. Hierbei könnten Parameter wie Zeitalter, oder Architekturstil eine Rolle spielen.

Eine vollkommen andere Art der Benutzerkontrolle wäre das Manipulieren der Welt in einem Editor. Hierbei könnte zunächst der Generator eine Siedlung generieren und der Benutzer hinterher Anpassungen vornehmen. Oder der Benutzer zeichnet nur eine grobe Grundriss, den der Generator als Input verwendet, um die endgültige Stadt zu generieren.

Literaturverzeichnis

[blog.movingblocks.net 2014] BLOG.MOVINGBLOCKS.NET: *Terasology*. 2014. – URL http://blog.movingblocks.net/wp-content/uploads/Terasology-130630003731-3840x2160.jpg. – Zugriffsdatum: 11.06.14

[Funck und Funck 2014] FUNCK, Sarah v. ; FUNCK, Wolfram v.: *About Pcroma*. 2014. – URL https://picroma.com/about. – Zugriffsdatum: 30.04.14

[I H Parish, Yoav und Müller 2001] I H PARISH, YOAV ; MÜLLER, Pascal: Procedural Modeling of Cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), S. 301–308

[Johnson 2013] JOHNSON, Leif: *PLAYING WITHOUT A FULL DECK*. 2013. – URL http://www.ign.com/articles/2013/01/05/ace-of-spades-review. – Zugriffsdatum: 11.06.14

[Kaiser 2013] KAISER, Rowan: *Eldrich Review*. 2013. – URL http://www.ign.com/articles/2013/11/02/eldritch-review. – Zugriffsdatum: 11.06.14

[Kelly und McCabe 2011] KELLY, George ; MCCABE, Hugh: A Survey of Precedural Techniques for City Generation. In: *Computational Intelligence and AI in Games, IEEE Transactions on* 3 (2011), Nr. 3

[Lechner u. a. 2003] LECHNER, Thomas ; WATSON, Ben ; WILENSKY, Uri ; FELSEN, Martin: Procedural City Modeling. In: *In 1st Midwestern Graphics Conference* (2003)

[Makuch 2014] MAKUCH, Eddie: *Minecraft PC sells 15 million copies, Notch celebrates with a can of Red Bull*. 2014. – URL http://www.gamespot.com/articles/minecraft-pc-sells-15-million-copies-notch-celebrates-with-a-can-of-1100-6419294/. – Zugriffsdatum: 11.06.14

[Müller u. a. 2006] MÜLLER, Pascal ; WONKA, Peter ; HAEGLER, Simon ; ULMER, Andreas ; GOOL, Luc van: Procedural Modeling of Buildings. In: *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2006* 3 (2006), Nr. 25

[Perlin 1999] Perlin, Ken: *Noise*. 1999. – URL http://www.noisemachine.com/talk1/imgs/elements.jpg. – Zugriffsdatum: 11.06.14

[Perlin 2001] Perlin, Ken: Real-Time Shading Languages. In: *SIGGRAPH Course Notes* 2001 (2001), Nr. 36, S. 2.1 – 2.24

[Prusinkiewicz und Lindenmayer 1990] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: *The algorithmic beauty of plants*. New York : Springer-Verlag, 1990 (The Virtual laboratory). – ISBN 0387946764

[Wikipedia Foundation 2014a] Wikipedia Foundation: *Dungeon Keeper*. 2014. – URL http://de.wikipedia.org/wiki/Dungeon_Keeper. – Zugriffsdatum: 11.06.14

[Wikipedia Foundation 2014b] Wikipedia Foundation: *Prozedurale_Synthese*. 2014. – URL http://de.wikipedia.org/wiki/Prozedurale_Synthese. – Zugriffsdatum: 11.06.14

[Wikipedia Foundation 2014c] Wikipedia Foundation: *White noise*. 2014. – URL http://en.wikipedia.org/wiki/White_noise#mediaviewer/File:White-noise-mv255-240x180.png. – Zugriffsdatum: 11.06.14

[Wonka u. a. 2003] Wonka, Peter ; Wimmer, Michael ; Sillion, Francois ; Ribarsky, William: Instant Architecture. In: *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003* 3 (2003), Nr. 22, S. 669–677

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Juni 2014 Sven Dettmers