Simulation von 2D Spielwelten mit zellularen Automaten

Nico E. Harather

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Dezember 2012

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 28. Dezember 2012

Nico E. Harather

Inhaltsverzeichnis

Vorwort

Bauklötze sind ein großartiges Spielzeug. Ein Kind kann mit ein wenig Vorstellungskraft und einer Kiste voller Bausteine ganze Welten erschaffen. In dieser Fantasiewelt werden die Klötze zu Häusern, Fahrzeugen, Tieren und Menschen. Soziale Beziehungen, Berufe, Freizeitaktivitäten, Politik, Wissenschaft und Wirtschaft; alles das kann Teil einer Welt aus Bauklötzen sein auch wenn es von Kindern nicht explizit als solches deklariert wird. Familien, Städte, Nationen und Zwist zwischen ihnen sind oft ein treibendes Element des Spiels. Aus den einheitlichen, kubischen Klötzen wird eine lebendige und glaubhafte Welt in der eigene Regeln und Gesetze gelten. Das ist die Magie der Vorstellungskraft, die Magie des Spiels.

Erwachsene Menschen spielen – wenn überhaupt – lieber mit komplexeren Spielzeugen. Manche mit Karten oder Brettspielen, andere verbringen ihre Freizeit mit Spielen am Computer oder auf Spielekonsolen. *Dungeons and Dragons* ist ein populäres PenPaper Rollenspiel das von einer Gruppe an Spielern mithilfe von Papier, Stiften und Würfeln gespielt wird. Die Spieler spielen gegen eine zusätzliche Person, den Spielleiter oder Dungeon Master der die Einhaltung der Spielregeln sicherstellt. Diese sind teils sehr aufwendig und in einem langen Regelwerk festgelegt, das die Spieler immer bei sich haben sollten. Wenn es im Spiel beispielsweise zu eine Kampf kommt, müssen die Spieler diverse Werte auswürfeln um die Schadenspunkte für ihren Charakter nach einer Formel auszurechnen. Der Dungeon Master übernimmt diese Aufgabe für alle Nichtspielercharaktere. Das kostet Zeit und es kommt schon mal vor, dass ein Kampf den guten Teil eines Abends in Anspruch nimmt.

Spiele in elektronischen Medien sind großartig weil sie aufwendige Spielmechaniken erlauben ohne dass sich der Spieler um die Einhaltung der Regeln oder irgendwelche Berechnungen kümmern muss. Die Spielmechanik von DungeonsDragons war schon öfters Vorbild für Computerspiele und konnte so umgesetzt werden dass der Spieler sich auf die Entscheidungen des Charakters konzentrieren kann und nichts per Hand ausrechnen oder nachschlagen muss. So hat er mehr Zeit um in die Illusion der Spielwelt abzutauchen. Allerdings geben Computerspiele Optik und Interaktionen vor die der Spieler akzeptieren muss, die Vorstellungskraft wird eingeschränkt.

Völlige Freiheit wird es in Computerspielen wohl nie geben und ist wahr-

scheinlich auch nicht erstrebenswert, schließlich braucht ein Spiel Regeln und einen Rahmen um überhaupt als solches zu gelten. Was aber wenn man die interessante Komplexität von Computerspielen mit dem altbewährten Baukastenprinzip verbindet? Warum nicht den Vorteil von schnellen Computern mit dem Vorteil der Bauklötze kombinieren? Man könnte dem Spieler Klötze geben die ihre eigenen Regeln enthalten und dieser formt sie mithilfe der Vorstellungskraft zu seiner eigenen Welt die sich dann auch ohne zutun weiterentwickelt und wächst. Ein bisschen so als würde der Baukasten zum Leben erwachen.

Kurzfassung

Dynamische Spielwelten sind in den letzten Jahren immer populärer geworden. Zerstörbare Umgebungen und Fluid Dynamics sind schon in vielen aktuellen Spielen enthalten. Auch Sandbox-Games machen umfassenden Gebrauch von dieser Technologie, doch es werden immer noch nur Teile der Welt simuliert. Mithilfe von zellularen Automaten könnte die Umgebung vollständig simuliert werden und Welt zur Mechanik werden. In dieser Arbeit wird ein Framework vorgestellt dass diese Aufgaben übernimmt. Es dient als Basis für eine natürliche Landschaft deren Teile eine Wechselwirkung zueinander und mit den Agenten in der Welt haben. Außerdem werden konkrete Implementierungen für Feuer, Fluide und Vegetation vorgeschlagen.

Abstract

Dynamic game environments are becoming more and more popular. Many games already feature destructible environments and fluid dynamics. Sandbox games make heavy use of those technologies as well, but they only simulate parts of their worlds. The entire environment could be simulated with the aid of cellular automata and the game world could become the game mechanics. This thesis introduces a framework that does exactly that. It allows a game environment based on the interaction between the cells and between the cells and the game agents. Implementations for fire, fluids and vegetation are presented and discussed.

Kapitel 1

Einleitung

Die Welten der Computerspiele haben in den letzten Jahren einen Übergang von statischen Levels hin zu lebendiger wirkenden, dynamischen Landschaften erfahren. *Red Faction* war 2001 eines der ersten Spiele das dem Spieler die teilweise Zerstörung der Umgebung erlaubt hat. 2008 kam *FarCry 2* auf den Markt und beeindruckte mit einer semi-zerstörbaren Welt. Darauf erschien im Oktober 2011 *Battlefield 3* in dem sowohl Gebäude als auch Szenerie sehr realitätnahe zerstört werden kann. Zudem haben die Veränderung des Levels auch Einfluss auf die Spielmechanik, Teile eines Gebäudes können zum Beispiel für einen taktischen Vorteil gesprengt werden.

In den meisten Spielen hat die Veränderung der Welt aber keine oder nur wenig Auswirkung auf das Gameplay. Die Spieleindustrie ignoriert die vielen Möglichkeit die sich aus einer Simulation von besipielsweise physikalischen Vorgängen ergeben können. Eine der erfreulichen Ausnahmen ist *Portal 2* das *Fluid Dynamics* verwendet und diese in die Spielmechanik integriert. Der Spieler kann mehrere Arten von Flüssigkeit erzeugen die unterschiedliche Effekt auf die Welt und den Spielercharakter selbst haben. Gelbe Flüssigkeit auf dem Boden lässt den Charakter schneller laufen, Blaue Flüssigkeit ermöglicht es ihm höher zu springen. Dank dieser Neuerungen im Spielkonzept wurde *Portal 2* von Spielern und Medien als eines der besten Puzzle-Games bisher gefeiert und mit einem *Metascore*[1] von 95 gewürdigt.

In allen diesen Spielen werden aber immer nur wenige Teile der Spiele simuliert. Es stellt sich die Frage ob es möglich wäre die ganze Welt zu simulieren und wenn ja, wie man an eine solche Aufgabe herangehen würde. Da Computer eine begrenzte Rechenleistung haben, muss die Welt und ihre Funktionsweisen zunächst abstrahiert werden. Die kontinuierliche Natur der Dinge kann – wie fast alles im digitalen Zeitalter – in diskreten Schritten abgebildet werden. Die Wechselwirkung zwischen allen Teilen muss aber irgendwie berechnet werden. Ein Ansatz dafür ist der Zellularautomat, ein Konzept zur Modellierung von räumlich diskreten, dynamischen Systemen.

[1] Quelle: http://www.metacritic.com/game/pc/portal-2

Der Einsatz in Spielen könnte eine Reihe von neuen Spielmechaniken zulassen die heutzutage noch nicht möglich sind. Tom Forsyth hat 2002 in [2, S. 201-202] schon drauf aufmerksam gemacht das dieser Ansatz vielversprechend ist:

> Using cellular automata (CA) to simulate these ideas [Anm.: Sich ausbreitendes Feuer, Explosionen mit realistischen Explosionsradien, Hitze und deren Einfluss auf Luftströmungen, Wasser das realistisch fließt, ...] can lead to far more dynamic and realistic behaviour and allow new types of gameplay and new tactics within games.

1.1 Zielsetzung

Diese Arbeit soll deshalb Ansätze finden wie eine solche Aufgabe mithilfe von Cellular Automata bewältigt werden kann. Es soll ein Schritt in die Richtung vollständig simulierter, dynamischer Spielwelten getan werden. Da das Thema sehr umfangreich ist, beschränkt sich der Inhalt dieser Arbeit auf natürliche Landschaften im ökologischen Sinn. Testumgebung für die folgenden Methoden sind daher Wiesen, Wälder und Gewässer. Es sollen Techniken erarbeitet werden mit denen eine Spielwelt simuliert werden kann, die durch ihre diskrete Natur einem Baukasten gleicht.

Die Idee zellulare Automaten für diverse Aufgaben in Computerspielen einzusetzen ist keine neue. Doch das Potenzial ist, wie Minecraft und andere Vertreter des *Sandbox Game* Genres eindrucksvoll bewiesen haben noch lange nicht ausgeschöpft. Diese Art von Spielen geben dem Spieler eine Vielzahl an Interaktionsmöglichkeiten aber lassen das Spielziel meist offen. Der Vergleich mit einer Sandkiste rührt daher dass der Spieler ein paar grundlegende Regeln beachten muss, aber seine eigene Welt bauen kann und sich Ziele und Aufgaben selbst setzt. Spielwelten die auf einem Cellular Automata System basieren sind weitaus interaktiver und dynamischer als herkömmliche, statische Welten. Durch ihre diskrete Zerstückelung kann jeder Teil einer Welt als ein Objekt gesehn werden das verschiedene Interaktionsmöglichkeiten für den Spieler bietet. Zusätzlich interagieren diese Objekt miteinander und schaffen so eine Welt die sich ständig verändert und eine gewisse Eigendynamik entwickelt.

Während die Kommunikation zwischen Spielercharakter und Welt teils als Spielelement eingesetzt wird, sind die Synergie zwischen den einzelnen Objekten die die Welt ausmachen und die daraus resultierenden Möglichkeiten für Spiele bisher großteils ignoriert worden. *Sweetser* stellte 2002 im Zuge ihrer Doktorarbeit [6] einige wichtige Überlegungen zum Thema an. Sie schrieb ein System das Algorithmen zur physischen Modellierung von zellularen Automaten enthält. Diese Algorithmen stammen aus der Arbeit von Forsyth [2] und decken Hitze und Feuer, Druck und Fluide ab.

Die Grundlage dieser Arbeit ist ein, in eine Game Engine eingebettetes Framework das einem Entwickler die Möglichkeit gibt eine zellular aufgebaute Spielwelt zu erschaffen indem er das Verhalten derer Teile bestimmt. Auf dieser Basis soll präsentiert werden wie solche Verhaltensweisen aussehen können und wie sie sich in eine Spielwelt einfügen. Themen sind wie bei Sweetser Hitze und Feuer und Fluide aber auch das Wachstum von Vegetation und wie es vom Spieleentwickler beinflusst werden kann.

Ziel der Arbeit ist einen Beitrag für die Entwicklung dynamischer Spielwelten zu leisten. Ein Beitrag ist das im Rahmen des Diplomprojekts enstandene Framework zur Simulation von 2D Spielumgebung und die Aufbereitung dessen im Kapitel *Diplomprojekt*. Der zweite Beitrag ist die Erarbeitung von drei Ansätzen zur konkreten Modellierung von natürlichen Vorgängen mit diesem System.

1.2 Motivation

Seit erfolgreichen Independent Games wie Minecraft und Terraria wissen wir dass Sandbox-Games großen Anklang bei vielen Spielern finden. Der Reiz dieser Art von Spielen besteht zu einem Großteil aus den Freiheiten die dem Spieler gegeben werden. Riesige, zufällig generierte Welten können entdeckt, erforscht und umgestaltet werden. Eines der wichtigsten Gameplay Elemente ist die Interaktion mit dieser Welt, alles was der Spieler sieht kann er verändern, zerstören und wiederaufbauen; Berge können abgetragen, Bäume gefällt und Flüsse umgeleitet werden.

Alle diese Dinge müssen von der Spiellogik simuliert und gespeichert werden um das Wachstum von Bäumen oder die neue Flussrichtung eines umgeleiteten Gewässers darzustellen. Um den Rechenaufwand gering zu halten werden meist zellulare Automaten und vereinfachte Modelle, beispielsweise für Flüssigkeiten oder Pflanzenpopulation verwendet. Die Welt wird dafür in diskrete Abschnitte oder auch Zellen unterteilt, wobei jede Zelle ihren momentanen Zustand speichern und den neuen Zustand im nächsten Zeitschritt berechnen kann.

Spielewelten sind heutzutage zum Großteil immer noch wenig interaktiv. In vielen Spielen findet man statische Welten in denen interaktive Elemente platziert werden um dem Spieler das Gefühl zu vermitteln, in dieser Umgebung etwas ausrichten zu können. Mit den immer besser werdenden Physik-Engines können hunderte oder tausende Objekte physikalisch glaubhaft berechnet werden. Völlige Freiheit bei der Interaktion mit der Spielwelt findet sich aber in kaum einem Spiel. Oft ist das vom Entwickler auch nicht gewünscht oder die benötigte Rechenzeit wird an anderer Stelle benötigt, manchmal aber leben Spiele von oben genannter Freiheit. Sandbox-Games sind ein Beispiel dafür, sie sind die – mehr oder weniger – erwachsene Version der Bauklötze. Diese Arbeit soll ein Vorstoß in die Richtung der Simulati-

ons –und Sandbox-Games sein, denn die Nachfrage für dieses Genre besteht. Minecraft und seine 42,267,300 registierten Benutzer[2] sind der Beweis dafür.

1.3 Gliederung

Der erste Teil der Arbeit geht zunächst auf das grundlegende Prinzip der zellularen Automaten ein von dem alle weiteren Überlegungen ausgehen. Danach wird die Auswahl der natürlichen Vorgänge besprochen für deren Berechnung Methoden erarbeitet werden. Die Abgrenzung zu anderen Bemühungen die auf diesem Gebiet bereits unternommen wurden schließt das Kapitel 2.

Kapitel 3 beschreibt zunächst die räumlich diskrete Aufteilung einer Spielwelt sowie die dynamischen Systeme die zur Simulation benötigt werden. Darauf folgen einige Beispiel aus bestehenden Spielen.

Das Kapitel 4 ist der Hauptteil dieser Arbeit in dem die erarbeiteten Techniken präsentiert werden. Die Basis sind drei Möglichkeiten zur Generierung einer Testwelt. Dann werden Hitze und Feuer besprochen, gefolgt von möglichen Berechnungen von Fluiden. Schließlich rückt die Vegetation in den Fokus und wird ausführlicher behandelt.

Die Zielsetzung des Diplomprojekts öffnet das Kapitel 5. Ein Framework für die Simulation einer 2D Welt wird vorgestellt und nach einem Abschnitt für Zielsetzung und verwendete Technologien wird die Architektur des Systems ausführlich besprochen. Außerdem sind hier die Teile aufgeführt die kritisch für die Performance sind. Danach finden sich einige Ideen die noch nicht umgesetzt worden sind.

In Kapitel 6 finden sich Überlegungen zur Zukunft der simulierten Spielwelten und mögliche Verbesserungen für die beschriebenen Techniken.

[2]Quelle: https://minecraft.net/ (Stand: 28.09.2012)

Kapitel 2

Thema

2.1 Cellular Automata

Jörg R. Weimar definiert in [7] zelluläre Automaten als Framework für eine große Gruppe an diskreten Modellen mit homogener Interaktion. Ergänzend stellt er eine Liste mit, wie er schreibt, *fundamentalen Eigenschaften* [7, S. 3]:

- They consist of a regular discrete lattice of cells.
- The evolution takes place in discrete time steps.
- Each cell is characterized by a state taken from a finite set of states.
- Each cell evolves according to the same rule which depends only on the state of the cell and a finite number of neighbouring cells.
- The neighborhood relation is local and uniform.

Ein Cellular Automaton ist also ein Gitter aus Zellen wobei jede Zelle immer nur einen Zustand aus einem endlichen Set von Zuständen annimmt. Zu jedem diskreten Zeitschritt ändern sich die Zustände der Zellen entsprechend einer oder mehreren, üblicherweise globalen Regeln. Der Zustand zum Zeitpunkt t ist abhängig von dem Zustand der Zelle und den Zuständen der Nachbarzellen zur Zeit *t-1*. Die Nachbarschaft kann unterschiedlich definiert sein, gebräuchlich sind die Moore-Nachbarschaft (8er-Nachbarschaft) so wie die Von-Neumann-Nachbarschaft (4er-Nachbarschaft) wie in Abb. 2.1 zu sehen.

Das wohl bekannteste Beispiel für Cellular Automata ist *Conway's Game Of Life*, ein zweidimensionales CA mit zwei Zuständen und sehr simplen Regeln die zu erstaunlich komplexen Mustern führen können. John Conway wollte ein System von John Von Neumann vereinfachen und das Resultat fachte 1970 das akademische Interesse für das Gebiet der zellularen Automaten an. Vorallem weil das System Turing-Vollständig ist fand es großen

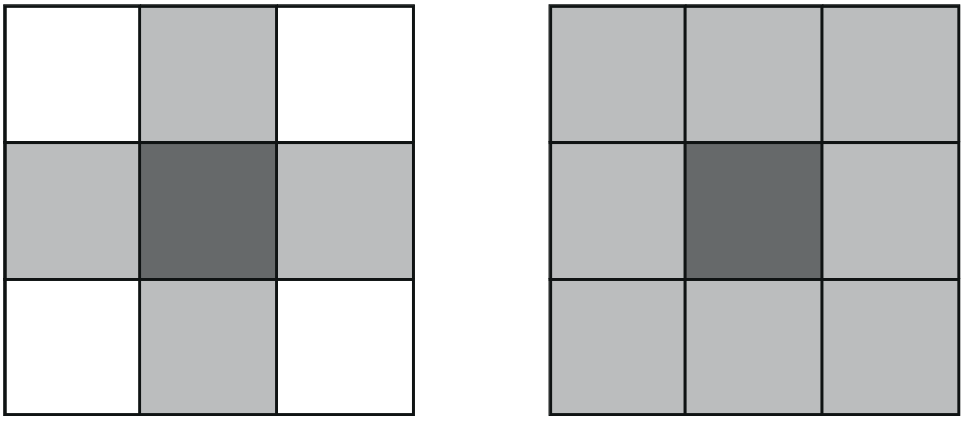

Abbildung 2.1: Von-Neumann –und Moore-Nachbarschaft

Anklang. Seitdem beschäftigen sich vorallem Mathematiker mit dem Thema, doch auch die Spielebranch hat die Vorzüge der Cellular Automata für sich entdeckt. Im Simulationsspiel *Sim City* von 1989 spielt man einen Stadtplaner und kann eine Stadt frei nach den eigenen Vorstellungen gestalten. Für viele Elemente des Spiels kommen CA Methoden zum Einsatz um beispielsweise die Ausbreitung von Feuer zu berechnen und den Verkehr zu simulieren. Cellular Automata in Spielen sind vorallem dann sinnvoll wenn die Zustände einer großen Anzahl von Objekt in Abhängigkeit zueinander stehen. Allerdings nehmen sich Entwickler oft die Freiheit ihre System so zu bauen, dass sie nicht der Definition von zellularen Automaten entsprechen. Es ist zum Beispiel möglich dass in einem Spiel eine Zelle gelöscht oder verschoben wird, das Zellen hinzukommen oder eine Zelle einen Zustandswechsel einer anderen auslöst. Eine gängige Praxis ist es die Charakteristiken eines CA beizubehalten und dort abzuweichen wo es für die Spielmechanik Sinn macht.

2.2 Fragestellung

Das, im Rahmen des Diplomprojekts entstandene Framework stellt einem Spielentwickler die essentiellen Funktionen zur Verfügung um eine 2D CA Welt zu erschaffen. Es verwaltet die Zellen und deren Zustände, stellt Methoden zur Interaktion zwischen Agenten und der Welt bereit und visualisiert das gesamte System. Es ist spezifisch für eine natürliche Landschaft ausgelegt und bietet entsprechende Funktionsweisen wie Ebenen für Boden und Luft, *velocity fields* und das sichere Verändern von Zellen. Das Framework ansich enthält aber noch keine Objekte der realen Welt, keine Bäume oder Gras und keine Tiere die darin leben, es ist lediglich eine Basis dafür. Wie die Welt in einem Spiel aussehen wird ist dem Entwickler überlassen der die-

ses Basis verwendet. Es macht Sinn die Funktionsweise im Hintergrund und die konkrete Implementierung einer Spielwelt zu trennen denn kein Spiel ist wie das andere, jedes hat seine eigenen Mechaniken und Anforderungen die man im Vorhinein nicht kennen kann.

Nun stellt sich für einen Entwickler die Frage, wie das System benutzt werden kann um die Vorgänge in der Natur zu simulieren. Wie können die Regeln lauten die die Welt lebendig machen? Um diese Frage zu beantworten muss zuerst festgelegt werden was denn überhaupt implementiert werden soll. Die Natur ist aus zahllosen Elementen zusammengesetzt und nicht alles kann und muss nachgebaut werden. Eine präzise Abbildung der Realität ist nicht möglich und in Spielen meist auch nicht nötig. Dennoch gibt es Teile der realen Welt die für ein Spiel relevant sein könnten, die folgende Liste zählt einige davon auf:

- Meteorologisch
  Wetter, Schnee, Regen, Luftdruck, Temperatur
- Geologisch
  Erdbeben, Erosion, Lawinen
- Biologisch
  Wanderung und Verbreitung von Populationen
  Vermehrung und Aussterben
  Räuber-Beute-Verhältnis
- Physikalisch
  Wärme und Feuer, Druck und Explosion, Fluide

Um den Rahmen dieser Arbeit nicht zu sprengen können nur ein paar dieser Punkte besprochen werden. Die Auswahl fiel auf biologische und physikalische Erscheinungen die sichtbar zum Bild einer Landschaft beitragen und dabei auch einen spielmechanischen Wert haben. In dieser Arbeit soll deshalb folgende Frage für die Ausbreitung von Feuer, für die Dynamik von Fluiden wie Wasser und für das Wachstum von Vegetation beantwortet werden:

> Welche Methoden können verwendet werden um mit dem CA Framework eine natürliche Umgebung zu simulieren?

2.3 Abgrenzung zu anderen Systemen

In manchen Spielen kommen CA Methoden zum Einsatz um Teile der Spielwelt zu simulieren. Diese werden zusätzlich zu herkömmlichen Spielwelten verwendet. Das hier verwendete Framework setzt vollständig auf Cellular Automata und macht andere Ansätze zur Ausnahme. Das heisst konkret das prinzipiell zu jedem Zeitschritt jeder Teil der Welt überprüft und neu

berechnet wird. Das hat den Vorteil dass der Spieler theoretisch mit jedem dieser Teile interagieren und diesen verändern kann, wenn das vom Entwickler gewünscht wird. Außerdem haben die Teile Einfluss aufeinander und somit eine Eigendynamik die sich wiederum auf den Spieler auswirkt.

2.3.1 EmerGEnT

Die Idee ist nicht ganz neu und schon 2002 von Penelope Sweetser aufgegriffen worden. Sie entwickelte ein System namens *EmerGEnT* das die Gleichungen verwendet die Tom Forsyth in [2] für die vereinfachte physikalische Vorgänge in CA Systemen aufgestellt hat. Sweetser behandelt ebenso wie die vorliegende Arbeit die Dynamik von Fluiden und von Feuer doch die Integration in ihr System ist eine andere als für das im Diplomprojekt entstandene Framework nötig ist. Vegetation und deren natürlich wirkende Verteilung und Vermehrung ist nicht Thema ihrer Arbeit.

2.3.2 Minecraft

Das Sandbox-Game *Minecraft* von Markus Persson ist eines der bekanntesten und meist gespielten Spiele die für die Simulation zumindest teilweise auf Cellular Automata setzen. Im Spiel existieren Objekte mit denen man einfache Schaltungen bauen kann, diese basieren auf einem CA Ansatz. Es wurde damit bereits ein Game Of Life gebaut und laut Angaben vom Entwickler selbst, ist das System Turing-vollständig[1] und nur durch die endliche Größe der Welt eingeschränkt. Diese kann aber enorme Ausmaße annehmen und speichert trotzdem alle Zustände für unbeschränkte Zeit. Die Visualisierung des Spiels ist dreidimensional und die Blocklänge der kubischen Zellen entspricht etwa einem Meter in der realen Welt. Trotzdem gibt es keine nennenswerten Probleme mit der Performance, weil sich Notch eines Tricks bedient: Alle regulären Zellen in der Welt von Minecraft sind passiv, das heisst das sie nicht zu jedem Zeitschritt aktualisiert werden. Die Welt ist trotzdem dynamisch weil es einige aktive Zellen gibt die andere Zellen aktiv schalten können wenn sie in Berührung kommen. So kann eine Kettenreaktion ausgelöst werden die das Bild einer lebendigen Welt vermitteln. Der Nachteil daran ist das viele physikalische und biologische Vorgänge entweder nicht oder nur eingeschränkt simuliert werden können. Diese Arbeit konzentriert sich auf die vollständige Simulation der gesamten Welt.

[1] Quelle: http://www.youtube.com/watch?v=rqUDam_KJno&feature=player_embedded#t=276s

Kapitel 3

Stand der Technik

3.1 Räumliche Aufteilung der Spielwelt

Die diskrete Aufteilung des Spielraums ist für Entwickler schon immer gängige Praxis gewesen. Es erleichtert das Festlegen und Einhalten der Spielregeln. Ein Beispiel aus der frühen Geschichte der Videospiele ist *Snake*, ein Klassiker der seit den späten 1970er Jahren existiert.

Der Spieler steuert eine Schlange in vier Richtungen durch eine Spielwelt lotsen die aus Feldern aufgebaut. Sie rückt in jedem Zeitschritt um ein Feld in die gegebene Richtung voran und zieht den Schwanz auf dem gleichen Weg hinter sich her. Es gibt Futter das je ein Feld belegt und die Schlange um ein Feld wachsen lässt wenn es berührt wird. Wenn die Schlange den eigenen Schwanz oder den Rand der Spielwelt berührt ist das Spiel vorbei. Ziel ist es sie so lange wie möglich werden zu lassen. Je länger die Schlange wird desto schwerer ist es aber auch sie in leere Felder zu steuern. Diese einfachen Regeln, die ein überraschend fesselndes Spiel ergeben, waren nur durch die Aufteilung der Welt in diskrete Schritte möglich. Damals waren Entwickler durch die Rechenleistung und die Möglichkeiten der Darstellung eingeschränkt, heute könnte man die Position der Schlange ohne Probleme mit Gleitkommazahlen kontinuierlich abbilden und ein Physiksystem zur Positionsberechnung verwenden. Der Spieler könnte mithilfe des Analog-Sticks eines Gamecontrollers in jede beliebige Richtung steuern. Das alles würde dann mit Rendering-Techniken aufwendig in Szene gesetzt werden. Allerdings müssten die Kollisionen der Schlange mit anderen Objekten entsprechend genau berechnet werden. Zudem könnte es sich herausstellen dass der Spieler nicht genau einschätzen kann wann die Schlange mit der Wand oder sich selbst kollidiert.

Ob die moderne Variante von *Snake* ein erfolgreiches Spiel sein würde ist aber nicht weiter von Bedeutung. Wichtig ist die Feststellung dass eine räumlich diskrete Spielwelt die Arbeit eines Entwicklers erleichtern kann. Regeln können meist mit weniger Aufwand implementiert und der Zustand

der Spielwelt einfacher überprüft werden. Ein Beispiel ist aber zuwenig um einen solchen Schluss zu ziehen. Deshalb werden im folgenden einige Klassiker aufgelistet bei denen erstens eine hypothetische Modernisierung einen ähnlichen Mehraufwand wie bei *Snake* bedeuten würde und die sich zweitens die Vorteile einer diskreten Aufteilung des Spielraums zu Nutzen machen. Manche dieser Spiele verwenden eine Kombination aus, in Felder eingeteilte Welten und kontinuierlichen Positionen der Agenten. So lässt sich die Welt besser kontrollieren und zugleich der Spielercharakter in alle Richtungen bewegen:

- Frogger,
- Q*bert,
- Harvest Moon,
- Diablo Reihe,
- Sims Reihe,
- SimCity Reihe,
- Heroes of Might and Magic Reihe,
- Anno Reihe.

Die aufgelisteten Spiele profitieren auf unterschiedliche Weisen von der räumlichen Aufteilung der Welt. Bei manchen davon, wie *Heroes of Might and Magic* sind Felder ein Teil des Spielprinzips der sich nicht so einfach ersetzen lässt. Andere, wie Diablo stellen die Spielwelt kontinuerlich dar, verwenden aber im Hintergrund quadratische Felder um Objekte und Gebäude zu platzieren.

3.2 Räumlich diskrete dynamische Systeme

Nur in wenigen Spielen beinflussen sich die Felder der Welt gegenseitig aber es gibt ein paar bei denen das Sinn macht. SimCity ist eine Wirtschaftssimulation deren Ziel der Aufbau und die Instandhaltung einer Stadt ist. Das Spiel verwendet intern ein Raster auf denen Gebäudezonen, Strassen und Spezialgebäude platziert werden können. Das Interessante daran ist dass das Geschehen auf jedem Feld abhängig von den umliegenden Feldern ist und es kann somit als ein diskretes dynamisches System gesehen werden. In der Mathematik ist das ein System dessen Zustand als Punkt in einem geometrischen Raum abgebildet werden kann. Eine Funktion mit einem Zeitwert als Parameter beschreibt dabei den Zustand des Punkts zur gegebenen Zeit. Computerspiele haben in der Regel ein Timer-Objekt das die Zeit zwischen den Rechenschritten festhält, daher ist ein solches System nicht nur räumlich sondern auch zeitdiskret. Die Zeit zwischen den Aktualisierungsschritten eines Spiels sind nahezu ident, deshalb können für die Berechnungen eines zeitdiskreten dynamischen Systems gleich große Zeitabstände verwendet werden.

Abbildung 3.1: Simulierte Stadt in *SimCity 2000*

In *SimCity* ist der Raum in Felder eingeteilt und die Zeit in diskrete Schritte, jedes Feld oder auch jeder Punkt ist abhängig von den umgebenden Feldern und wird zu jedem Zeitpunkt neu berechnet. Es gibt eine endliche Anzahl an Zuständen und feste Regeln für die Zustandsänderungen. Weil alle Anforderungen dafür erfüllt sind bietet sich für die Simulation von Teilen der SimCity Welt das Konzept der zellularen Automaten an. Tatsächlich verwendet das Spiel Techniken die als Cellular Automata qualifizieren und gehört deshalb zu den wenigen die ein solches System verwenden um die Spielwelt zu simulieren.

3.3 Cellular Automata in Games

3.3.1 Sim City

Seit dem ersten Teil der *Sim City* Reihe wird für die Simulation des Verkehrs und der Stadtentwicklung auf zellulare Automaten gesetzt. Überflutungen und Feuer sind damit ebenso möglich wie Verunreinigungen und die Berechnung der Kriminalität in Stadtteilen. So gut wie alles das in Abb. 3.1[1] aus *SimCity 2000* zu sehen ist, ist auf die eine oder andere Art simuliert, meist mit CA Techniken.

[1] Quelle: http://pseudo6man.blogspot.co.at/2010/12/life-of-pc-gamer-simcity-2000_06.html

3.3.2 Minecraft

Minecraft ist ein weiteres Exempel für räumlich diskrete Spielwelten und dynamische Systemen in Spielen. Die 3D Welt ist optisch und spielmechanisch in diskrete Schritte aufgeteilt die der Entwickler *Blocks* nennt. Diese können von einem Charakter abgebaut, in ein Inventar abgelegt und verarbeitet oder zu späterem Zeitpunkt an anderer Stelle wieder aufgbebaut werden.

Redstone

Darunter finden sich auch sogenannte *Redstone* Blocks die zum Bau von elektronischen Schaltungen verwendet werden können. Wie in der Elektronik stehen mehrere Bauteile zur Verfügung deren Kombination alles von einfachen Schaltkreisen bis zu Turingmaschinen zulässt. *Redstone* Blocks nehmen in *Minecraft* eine Sonderstellung ein da sie zu speziellen Bauteilen verarbeitet werden können, den *Redstone Torches* die als Energiequellen dienen. Mithilfe von *Redstone Dust* der auf dem Boden verteilt wird und als Kabel dient kann ein Verbraucher mit Energie versorgt werden. Es existieren Lampen, elektrische Türen und Falltüren, Musikblöcke sowie *Dispenser* die Gegenstände ausgeben können. Außerdem wurde in einem Update der sogenannte *Piston* eingeführt der es ermöglicht elektrische in mechanische Energie zu wandeln. Eine beachtliche Neuerung da dadurch erstmal Blocks kontrolliert bewegt werden können.

Redstone Schaltungen funktionieren wie zellulare Automaten indem die einzelnen Teile ihre unmittelbare Umgebung als Grundlage für ihren Zustand im nächsten Zeitschritt verwenden. Die Signalstärke von *Redstone* Kabeln hat 15 Abstufungen und überträgt nach 15 zusammenhängenden Blocks keine Energie mehr. Die Regel für die Aktualisierung des Energiezustands ist

$$Energie = Max_{Energie}(Nachbarn) - 1. \tag{3.1}$$

Eine sehr einfache Lösung die sich als sehr praktikabel herausgestellt hat. Seit der Alpha die 2009 spielbar ist, hat sich eine große Community um Minecraft gebildet die bis heute noch sehr aktiv ist. Das immer noch rege Interesse ist nicht zuletzt auf die unzähligen Möglichkeiten zurückzuführen die *Redstone* bietet. Denn die meisten Spieler die ihre ersten Gebäude gebaut und Höhlen erforscht haben, wenden sich in der Regel den komplexeren Teilen des Spiels zu. So können Abläufe wie das Bewirtschaften von Felder mit *Redstone* Schaltungen automatisiert werden und dem Spieler so Zeit geben um andere Dinge auszuprobieren. Letzlich ist es dem Prinzip der zellularen Automaten zu verdanken dass dieses Spiel so gut funktioniert. Abb. 3.2[2] zeigt einen einfachen Schaltkreis der in Minecraft als Uhr oder Schleife für größere Schaltungen verwendet wird.

[2] Quelle: http://www.instructables.com/id/Simple-Minecraft-Redstone-Wiring/

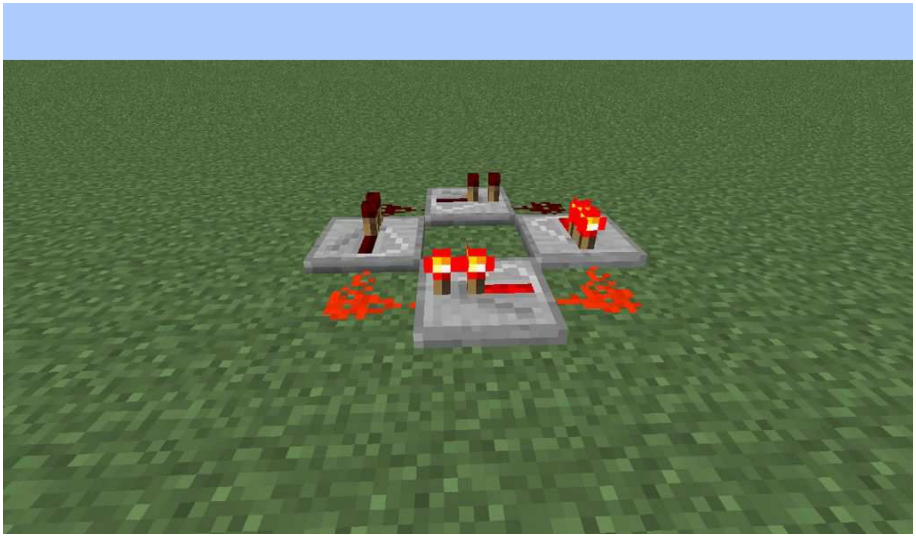

Abbildung 3.2: Clock/Loop Circuit in Minecraft

Wasser

Da die Welt von *Minecraft* zu Beginn hauptsächlich aus natürlicher Landschaft, also Wiesen, Wäldern, Flüssen und Seen besteht, mussten das Wachstum von Gras und Bäumen sowie das Verhalten von Wasser implementiert werden. Alle Techniken dafür sind simpel und abstrahiert aber passen in die Welt. Für Wasser gibt es Quellen die eine unendliche Menge Flüssigkeit erzeugen, diese können mit einem Kübel aufgenommen und dann an einem anderen Ort platziert werden. Wasser das keine Quelle ist, fließt in alle Richtungen solange der Zielblock im Luft-Zustand ist. Fließendes Wasser hat 7 Abstufungen oder auch Level, die die Menge der Flüssigkeit angeben. Drei Regeln bestimmen das Verhalten von Wasser:

- Luft oder Wasser wird zu fließendem Wasser wenn ein direkter Nachbar Wasser ist. Dabei gilt $Level = Max_{Level}(Nachbarn) - 1$.
- Luft oder Wasser wird zu einer Quelle wenn zumindest zwei Nachbarn auch Quellen sind.
- Luft wird zu Wasser wenn der Block darüber Wasser ist.

In Abb. 3.3[3] zu erkennen dass sich das Wasser nach 7 Schritten nicht weiter im Raum verteilt, dies ist der Endzustand den die Flüssigkeit einnimmt nachdem die Quelle gesetzt wurde. Obwohl das Verhalten physikalisch nicht korrekt ist, funktioniert es für die Spieler in diesem Kontext recht gut. Im Framework dass für diese Arbeit erstellt wurde, kommt ein anderer Ansatz zum Einsatz der Flüssigkeiten realistischer berechnen kann. Der Vorteil des

[3] Quelle: http://madflame991.blogspot.co.at/2011/10/cellular-automata-in-minecraft.html

Abbildung 3.3: Wasser in Minecraft

Minecraft Ansatzes ist die relativ einfache Berechnung und die daraus resultierende Randbedingung, ansonsten müsste sich der Entwickler Gedanken darüber machen wie er die Welt vor einer Überflutung durch nur eine Quelle schützt. Der Vorteil der Technik in dieser Arbeit ist, dass sich Fluide realitätsnaher verhalten und somit überzeugender dargestellt und für komplexere spielmechanische Zwecke eingesetzt werden können.

Gras

Der Boden an der Oberfläche besteht in *Minecraft* hauptsäschlich aus Erde auf der Gras wachsen kann. Auf einem Block wächst Gras wenn genügend Licht vorhanden und ein Nachbarblock auch Gras ist. Eine sehr simple Methode die in diesem Kontext gut funktioniert.

Kapitel 4

Techniken

Die in diesem Kaptiel beschriebenen Techniken sind Möglichkeiten für die Simulation einer 2D Spielwelt mit dem, im Diplomprojekt erarbeiteten Framework. Sie dienen der Orientierung beim Einstieg in das Arbeiten mit dem System und zeigen einige erprobte Vorgehensweisen auf. Die meisten Techniken beruhen darauf dass die Blöcke ihre Zustände selbst aktualisieren und verändern, das ist die Essenz der zellularen Automaten. Es ist immer darauf zu achten dass sich die Blocks nicht direkt gegenseitig verändern. Wenn ein Block den Zustand eines Blocks ändert der im selben Zeitschritt nach ihm aktualisiert wird, beinflusst dies das Verhalten aller Nachbarblöcke. Es folgt eine Kettenreaktion die das gesamte Grid verfälschen und die Simulation instabil machen kann. Außerdem sollten die Variablen des Blocks doppelt gespeichert werden. Einmal für den aktuellen und ein zweites Mal für den vorhergehenden Zeitschritt. Ein Block macht die Berechnung des neuen Werts oft vom vorhergehenden abhängig. Würd der neue den alten Wert überschreiben und ein Nachbarblock im selben Aktualisierungsschritt besagten Wert für eine Berechnung verwenden, wäre sie um einen Zeitschritt voraus. Wenn ein Variablenpaar verwendet wird, ist das Risiko einer asynchronen Aktualisierung minimiert. Obwohl das Framework einige Funktionen hat um solche und ähnliche Fehler zu vermeiden, sollte beim Implementieren der Regeln darauf geachtet werden.

4.1 Level-Generierung

Die folgenden Methoden sind teilweise abhängig von den Feuchtigkeitsverhältnissen und Verteilung von Gewässern in der Welt. Deshalb soll schon zu Beginn der Simulation etwas Struktur in die Testwelt gebracht werden. Bei jedem Neustart wird eine neue Landschaft zufällig generiert. Das ermöglicht es die Ansätze auf ihre Brauchbarkeit unter verschiedenen Bedingungen zu untersuchen.

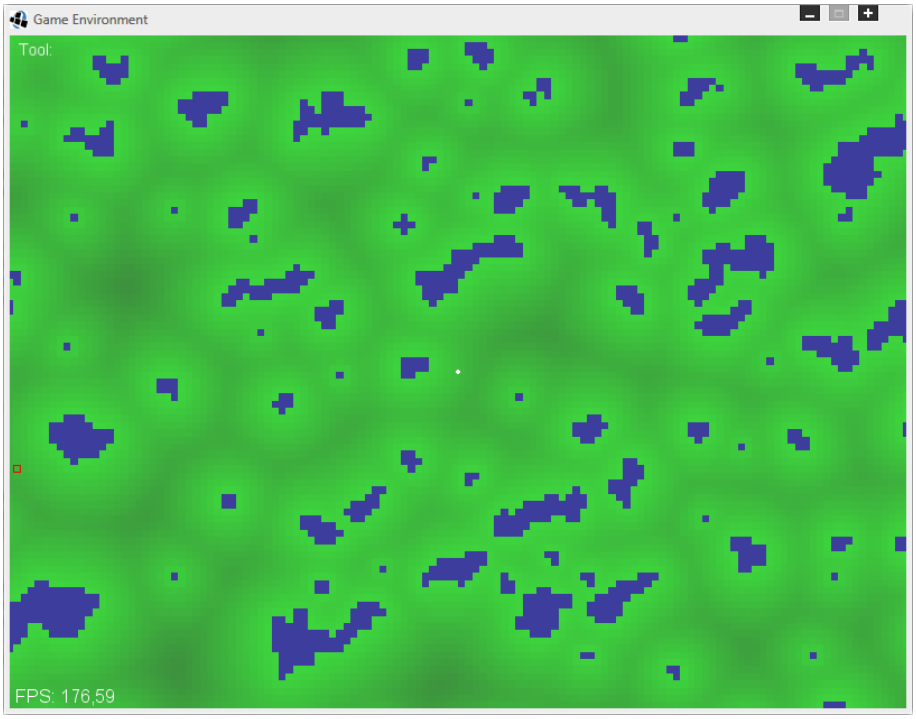

Abbildung 4.1: Levelgenerierung mit anfänglich 60% Land und $k = 10$

4.1.1 Region Growing

Der erste Ansatz für die Levelgenerierung war ein Algorithmus der dem *Region Growing* Algorithmus aus der Bildverarbeitung gleicht. Zu Beginn wird über alle Zellen des Grids iteriert und diese zufällig mit entweder Land– oder Wasserzuständen versehen. Als zweiter Schritt werden die Nachbarn für jeden Block auf Wasser oder Land überprüft. Wenn ein Landblock vier oder weniger Landblocks um sich herum hat, wird er zu Wasser. Wasser wird zu Land wenn um den Block herum fünf oder mehr Nachbarn Land sind. Diese Prozedur hat k Durchläufe wobei sich der Wert $k = (5, ..., 20)$ bewährt hat. Das Ergebnis ist stark abhängig von der ursprünglichen Verteilung von Land –und Wasserblöcken. Abb. 4.1 zeigt das Resultat für $k = 10$ und einer 60% Chance zugunsten der Landblocks. Diese Methode stellt nicht den Anspruch darauf natürlich oder realitätsnah zu wirken, es stellt lediglich ein Testbett für die restlichen Methoden dar.

Das Resultat war ausreichend für einige Experimente mit Waldzonen und Seeds, wie weiter unten noch ausführlicher besprochen wird. Allerdings hat diese Methode den Nachteil dass sie keine Höhendaten produzieren kann. Die Repräsentation der Welt ist zwar eine orthogonale Projektion von oben, Positionswerte auf einer dritten Raumachse machen für die Simulation aber durchaus Sinn. Die weiteren Überlegungen zur Vegetation umfassen bei-

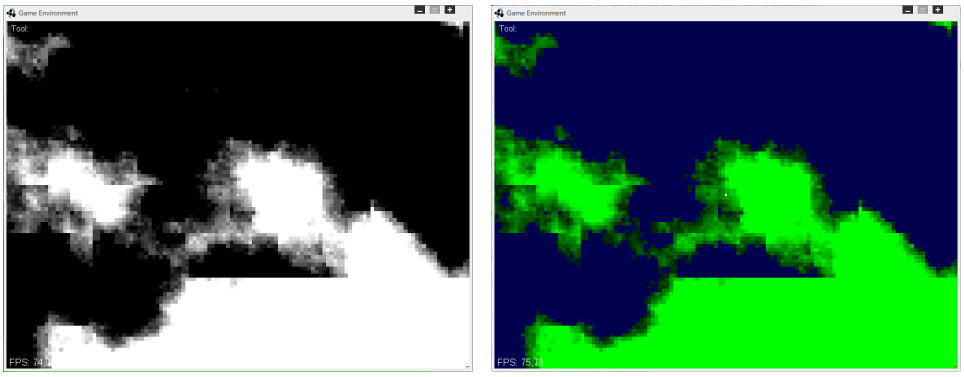

Abbildung 4.2: Levelgenerierung mit dem Random Midpoint Displacement Algorithmus

spielsweise das verringerte Baumwachstum überhalb einer Waldgrenze für die Höhendaten notwendig sind.

4.1.2 Random Midpoint Displacement

Um Höhen und Tiefen in die, zuvor flache Welt zu bringen wird eine *Heightmap* erzeugt. Dazu wurde der rekursive Random Midpoint Displacement Algorithmus verwendet der 1982 in [3] von *Fournier et al.* vorgestellt wurde und seitdem eine wichtige Rolle in der Terraingenerierung spielt. Vereinfacht gesagt wird dabei ein Quadrat in vier gleich große Quadrate geteilt, der Mittelpunkt zufällig auf der z-Achse verschoben und die Prozedur für alle daraus resultierenden Quadrate wiederholt bis die vorgegeben Zahl an Iterationen erreicht ist.

Abb. 4.2 zeigt links die erzeugte Heightmap und rechts die daraus entstandene Landschaft. Für beide Bilder gilt dass die dunkleren Regionen tiefer liegen als die hellen. Die Entwicklung einiger Techniken in dieser Arbeit setzte Wasserblocks vorraus, also wurden die Blöcke unter einem festgesetzten Niveau mit Wasser angefüllt. Die z-Werte werden innerhalb des Programms als Ganzzahlen behandelt und reichen von -255 bis 255, der Wasserpegel im Bild ist bei 50 Einheiten angesiedelt. Vorallem in den Küstenregionen ist eine deutliche Verbesserung zu spüren, allerdings ist auch ein, für den Algorithmus typisches Artefakt zu erkennen. Die gerade Linie im unteren Teil der Abb. 4.2 zeigt deutlich dass der Algorithmus eine Schwachstelle hat die ihn für die Terraingeneration disqualifiziert.

4.1.3 Diamond Square Algorithmus

Eine verbesserte Version des *Random Midpoint Displacement* ist der *Diamond Square* Algorithmus. Der Unterschied ist ein zusätzlicher Schritt der

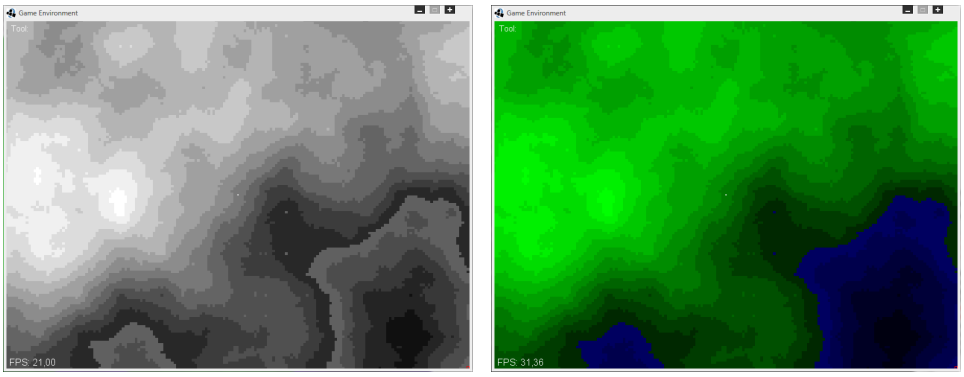

Abbildung 4.3: Levelgenerierung mit dem Diamond Square Algorithmus

die ursprünglichen Quadrate um 45° dreht und die gleiche Prozedur auf die, nun rautenförmigen Teile der *Heightmap* anwendet. Das reduziert die rechteckigen Artefakte deutlich. In Abb. 4.3 sind links wieder die Höhenwerte in Graustufen und daneben die Landschaft zu sehen. Der Wasserpegel ist bei 30 Einheiten angesetzt. Um die Abgrenzung zum Wasser optisch deutlicher zu machen wurde ein kleiner, konstanter Wert zur Helligkeit der entsprechenden Blocks addiert.

Mit diesem Ergebnis lässt sich wesentlich besser arbeiten und testen weil die Höhenunterschiede nun vorhanden sind und die oben genannte Waldgrenze zulässt. Um die Zusammenhänge möglichst einfach erkennen zu können, wird trotzdem zunächst die erste Methode zur Levelgeneration verwendet. Zu beachten ist nämlich dass die Variation der Grüntöne hier die Feuchtigkeit der Grasblocks darstellt. Später wird der Höhenunterschied wichtiger und die Farbtöne zu diesem Zweck genutzt.

4.2 Hitze und Feuer

Um Wärme und davon abhängige Phänomene zu simulieren hat jeder Block einen *temperature* Wert. Dieser kann vom Entwickler für diverse Zwecke verwendet werden, im Prototyp wird er für brennbares Material gebraucht. Blocks haben eine boolesche Variable *onFire* die standardmäßig *false* ist und nur dann *true* wird wenn ein Block brennt. Eine Zelle entzündet sich dann wenn die Temperatur gleich oder höher dem vom Zustand definierten Brennpunkt ist. Diese Bedingung ist so allgemein dass sie direkt in der *Block* Klasse implementiert wurde und somit automatisch zu jedem Aktualisierungsschritt für alle Blocks im Grid geprüft wird. Jeder entzündliche *BlockState* hält eine Konstante *IGNITION_POINT* die den dem Material entsprechenden Schwellwert in Grad Celsius angibt. Sollte das Material nicht entzündbar sein so ist dieser Wert Null und *onFire* wird dadurch nie

true sein.

Ob und wie ein Block Wärme aufnimmt oder abgibt ist wiederum in den *BlockState* Objekten definiert. Im Prototyp können Gras und Bäume Wärme aus ihrer unmittelbaren Umgebung (Moore Nachbarschaft) aufnehmen wenn ein oder mehrere Nachbarn wärmer als sie selbst sind. Mehr braucht es nicht um einen Brand zu simulieren der sich ausbreiten kann, denn wenn ein Block brennt und sehr heiß ist werden alle umliegenden Zellen ebenfalls wärmer und beginnen ab dem Schwellwert zu brennen. Irgendwann ist in der Realität das brennbare Material verbraucht und das Feuer erlischt, deshalb ist in der Implementierung jeder Block mit einer Variable *fuel* versehen. Dieser Wert reduziert sich kontinuierlich im Laufe des Brandes und *onFire* wird wieder auf falsch gesetzt wenn der Wert bei Null angelangt ist. Somit gibt es im Prototyp Feuer das sich ausbreitet, brennbares Material verbraucht und erlischt wenn nichts mehr da ist das verbrannt werden könnte. Die Menge an *fuel* ist dabei auch verantwortlich dafür wie schnell ein Block abbrennt, Bäume brennen länger als Gras. Jeder *BlockState* muss dann selbst bestimmen welche Konsequenzen ein Brand hat, praktikabel ist ein Zustandswechsel von beispielsweise Gras zu Erde. Abb. 4.4 zeigt einen sich ausbreitenden Brand, wobei der Brandherd im Zentrum bereits erloschen ist. Die Gras innerhalb der Flammenwand ist verbrannt aber die Bäume brennen noch weil sie einen höheren *fuel* Wert haben.

4.3 Fluide

Techniken zur realitätsnahen Darstellung von Fluiden werden in Spielen meist nur dann eingesetzt wenn diese eine Bedeutung für die Spielmechanik haben. Oft werden aber weniger rechenaufwendige Methoden verwendet. In vielen Spielwelten kann der Spielercharakter durch Flüsse und andere Gewässer gehen oder schwimmen. Dazu können vom Designer des Levels Zonen definiert werden in denen sich die Fortbewegungsart des Charakters ändert. Wenn sich der Charakter in eine Flusszone bewegt, wird eine Animationen für Schwimmen statt für Laufen abgespielt. Ein Richtungsvektor sorgt dafür dass die Position des Charakters stetig um einen kleinen Wert in die Fließrichtung verschoben wird und schon hat es den Anschein als würde er von der Strömung des Flusses mitgenommen. In den meisten Spielen ist das ausreichend um Wasser und andere Flüssigkeiten glaubhaft zu simulieren, der Spieler muss der Strömung entgegensteuern und es vermittelt das Gefühl, dass sich der Charakter tatsächlich im Wasser befindet.

Diese Methode funktioniert solange das Level statisch aufgebaut ist. Wenn sich die Umgebung verändern können soll – und das ist in dieser Arbeit der Fall – dann benötigt man einen anderen Ansatz. Im Prototyp sollen Fluide etwa so verhalten wie in unserer Welt. Gräbt der Spieler ein Loch neben einem Fluss so fließt Wasser auch dorthin. Vordefinierte Zonen

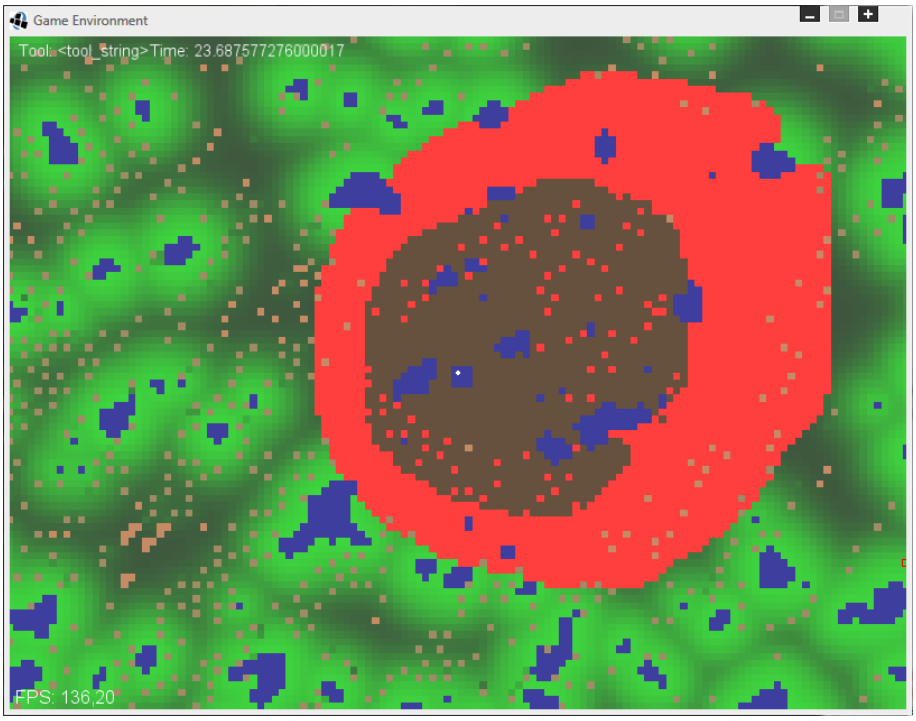

Abbildung 4.4: Feuer im CA System

funktionieren nicht mehr wenn sich der Flussverlauf ändert, dazu muss die Fließrichtung stetig neu berechnnet und die Wassermassen entsprechend verschoben werden. Jos Stam hat in [5] eine Methode entwickelt die mit ein paar Änderungen gut ins System integriert werden konnten. Kleine Anpassungen im *GridService* als auch in der *Block* Klasse erlauben es den Großteil der Berechnungen direkt im betroffenen *BlockState* zu machen. Das Verhalten von Wasser kann somit wie angedacht im Wasser-Block definiert werden.

Stam geht von einer zweidimensionalen diskreten Repräsentation der Fluide aus die in diesem Fall gegeben ist. Der Zustand eines Fluids ist als ein Feld aus Richtungsvektoren, ein *velocity vector field* modelliert. Jeder Block wurde daher mit einer Variable *velocity* versehen. Abbildung 4.5 zeigt eine Testumgebung die nur aus Fluiden besteht mit den entsprechenden Richtungsvektoren der Blocks. Um die Menge der jeweiligen Fluide in einer Zelle festzuhalten wurde außerdem eine Variable für die Dichtheit, namentlich *density* eingeführt. Zu jedem Zeitschritt wird nun berechnet wie hoch der *density* Wert jeder Zelle ist. Dafür sieht Stam drei Schritte vor.

4.3.1 Quellen hinzufügen

Zunächst benötigt jede Flüssigkeit und jedes Gas eine Quelle aus der der Dichtewert kommt. Das können in einem Spiel eine Zigarette, ein brennender Busch, eine Wasserquelle und viele andere Dinge sein. Im Prototyp gibt es theoretisch zwei Wege den *density* Wert zu beeinflussen: Der Spieler kann mithilfe eines Werkzeugs die enstprechende Variable im Block direkt verändern oder es kann einen *BlockState* geben der als Quelle dient. Dieses Objekt kann entweder immer in jedem Aktualisierungsschritt einen festen Dichtewert zum Block hinzufügen oder das nur solange tun bis ein Limit erreicht wurde. Das Limit kann ein Zeitwert, eine gewisse Menge oder eine andere, vom Entwickler festgelegte Bedingung sein. Würde man versuchen Fluide ohne Quelle zu benutzen so wurde sich der Dichtewert von Flüssigkeiten und Gase durch Diffusion rasch gegen Null entwickeln.

4.3.2 Diffusion

Diffusion ist die gleichmäßige Verteilung von Teilchen im Raum, in diesem Fall die Verteilung von Dichtewerten. Weil nicht jedes Fluid gleich schnell diffundiert bekommt jeder Block von seinem *BlockState* einen Diffusionsrate *diffusionRate* zugewiesen. Wenn dieser Wert größer als Null ist wird sich das Fluid über alle umliegenden Zellen ausbreiten. Die Implementierung für Wasser berücksichtigt nur vier Nachbarn wobei jede Zelle Dichte an die Nachbarn verliert aber auch Dichte von den Nachbarn in die Zelle fließt. Der Unterschied ist laut Stam

$$\begin{aligned} x_t(i,j) = x_{t-1}(i,j) - a \cdot (x_{t-1}(i-1,j) + x_{t-1}(i+1,j) \\ + x_{t-1}(i,j-1) + x_{t-1}(i,j+1) - 4 \cdot x_{t-1}(i,j)) \end{aligned} \tag{4.1}$$

Der Wert a errechnet sich auf der Diffusionsrate d, der Zeit seit dem letzten Zeitschritt dt und der Größe der Welt N als

$$a = dt \cdot d \cdot N \cdot N \tag{4.2}$$

4.3.3 Bewegung

Um die Verschiebung der Dichtewerte zu berechnen, könnte man annehmen dass das Zentrum einer Zelle ein Teilchen ist und dieses entlang dem *velocity field* für genau einen Zeitschritt verfolgen. Dann müsste man diese Position aber wieder in einen diskreten Zellenwert zurückrechnen. Stam's Methode versucht daher das Teilchen zu finden dass nach einem Zeitschritt genau im Zentrum der zu berechnenden Zelle ist. Den Dichtewert den dieses Teilchen mitführt erhält man durch eine lineare Interpolation der Dichtewerte der vier nähesten Nachbarn des Startpunkts. Um diese Methode anzuwenden

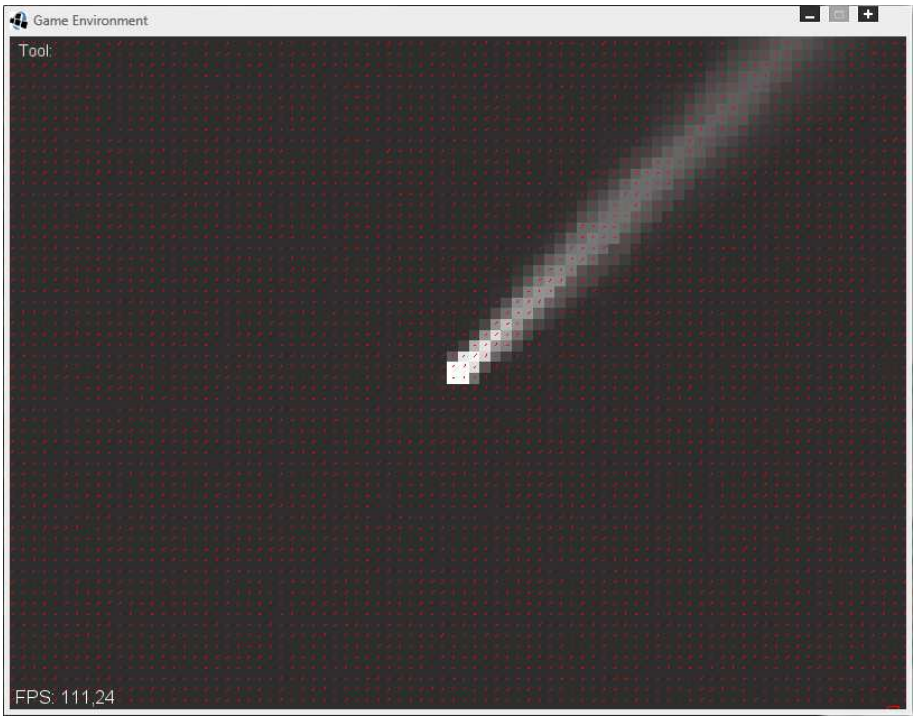

Abbildung 4.5: Fluid Emitter unter Einfluss eines *velocity field*

benötigt man ein Grid bestehend aus den Dichtewerten des Zeitschritts t und eines mit den Werten von $t-1$, im Prototyp implementiert als zwei Blockvariablen *density* und *prevDensity*. Jeder fluide Block verfolgt die Position des Teilchens in seinem Zentrum zum Zeitpunkt t durch das *velocity field* zurück um den Startpunkt desselben zum Zeitpunkt $t-1$ zu finden. Aus den vier nächsten Nachbarn errechnet sich der Dichtewert der verschoben wird.

Abb. 4.5 zeigt einen *Fluid Emitter*, ein Block der als Quelle für Fluide dient. Dieser hat lediglich die Aufgabe seinen *density* Wert in jedem Zeitschritt wieder aufzufüllen. Durch Diffusion verteilt sich das Fluid im Raum. Über das *velocity field* in Rot werden die Dichtewerte bewegt um Wind oder eine Strömung zu simulieren.

4.4 Vegetation

In den meisten Computerspielen findet man eine Variation von Pflanzen die manchmal Teil der Spielmechanik, meist aber nur aus optischen Gründen eingebaut sind. Wenn eine Spielwelt eine natürliche Landschaft enthält ist es wichtig dass die Vegetation – sowie alle anderen Teil der Spielwelt – glaubhaft dargestellt wird um die Immersion des Spielers nicht zu brechen. Nur so

kann sich der Spieler auf die Umgebung einlassen und mit ihr interagieren als wäre sie real. Moderne Game Engines wie Unity und die Cry Engine geben dem Entwickler schon sehr gute Mittel um Gras, Bäume und andere Pflanzen fotorealischtisch zu rendern. Damit das Gesamtbild stimmt müssen diese aber noch richtig positioniert werden und hier liegt auch das Problem. Die natürlich erscheinende Verteilung der verschiedenen Arten von Pflanzen sowie die räumliche Verteilung kann in einem statischen Level recht einfach erreicht werden indem der Designer alles per Hand auslegt. In einem statischen Level kann das mit etwas Verständnis für die Vorgänge in der Natur und einem Auge fürs Detail zu sehr guten Resultaten führen. Doch auch das Gegenteil ist möglich wenn dieses Verständnis weniger ausgeprägt ist.

Eugene Ch'ng macht in seiner Arbeit [1] darauf aufmerksam dass die Repräsentation von Vegetation in vielen Spielen zwar sehr realistisch ist, die Platzierung der einzelnen Pflanzen aber nicht:

> While the graphical representations of plants may look unbelievably real, a crucial factor related to their placements often lacks a natural association with those seen in the physical world.

Seiner Ansicht nach liegt das an den unzureichenden botanischen Kentnissen der Leveldesigner. Um diesen eine Hilfestellung in WYSIWYG Editoren für Landschaften zu geben, will er Algorithmen aus dem Gebiet der Ökologischen Modellierung adaptieren [1]:

> If algorithms can be adopted from this particular research area, creating vegetation covers for outdoor scenes would be made simpler.

Ch'ng verwendet Erkentnisse aus der *Individual Based Ecology* (IBE), einem Teilgebiet der *Ökologischen Modellierung*. Er schlägt unter anderem vor, jeder Pflanze einen Genotyp zuzuweisen, der bestimmt unter welchen Bedingungen diese wachsen kann. Ein paar seiner Methoden, diese eingeschlossen, finden auch in dieser Arbeit Anwendung. Für die folgenden Ansätze sollte man in Erinnerung behalten, dass es nicht nur um die spatiale Verteilung von Pflanzen sondern auch um die potentiellen Interaktionsmöglichkeiten für den Spieler geht.

4.4.1 Einfacher CA-Ansatz

Im System des Prototyp ist es theoretisch möglich Pflanzen manuell zu verteilen und kann auch ein Teil der Spielmechanik sein. Wie in einem statischen Level kann der Designer Bäume an eine bestimmte Position setzen und so die Spielwelt entsprechend den eigenen Vorstellungen gestalten. Sobald die Simulation aber gestartet wird, beginnt sich die Umgebung zu verändern und nach einigen Zeitschritten könnte sie schon ganz anders aussehen als

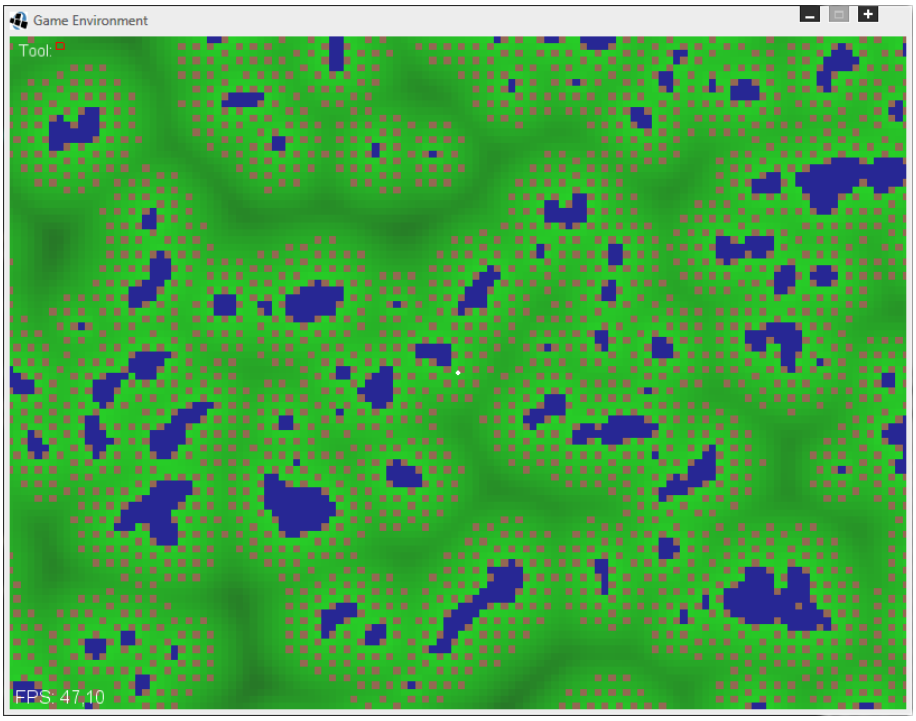

Abbildung 4.6: Wald mit einfachem CA Ansatz

ursprünglich gedacht. Eine bestimmte Anordnung von Vegetation in dieser Welt hat also nicht lange Bestand. Der Sinn und Vorteil des Cellular Automata Systems ist eine dynamische Welt die sich entwickelt und lebendig wirkt. Bäume können ohne das zutun des Designer oder Spielers wachsen. Konkret wird das durch eine Regel im Zustandsobjekt *BlockGras* ermöglicht. Ein Baum wir dann gepflanzt wenn genug Feuchtigkeit im Block vorhanden ist und in der Moore-Nachbarschaft nicht bereits ein Baum wächst. In Abb. 4.6 ist das Ergebnis dieser recht einfachen Anweisung zu sehen. Wasser ist in Blau, Gras in Grüntönen und Bäume in Braun eingezeichnet.

Es ist deutlich erkennbar dass die Baumpopulation um die Wasserstellen herum dicht ist. Auf den eher trockenen, dunkelgrünen Gebieten wächst nichts, das ist auf die Feuchtigkeitsbedingung zurückzuführen. Die Verteilung ist sehr gleichmäßig, kein Baum hat einen anderen als direkten Nachbarn und alle haben exakt acht Zellen um sich herum auf denen kein Baum wächst. Die Blocks können Entscheidungen nur anhand der Nachbarblocks sowie eigener Werte treffen und haben kein Gesamtbild von der Welt. So ist es schwierig eine überzeugende Verteilung darzustellen. Ein Entwickler kann die Bedingungen für das Wachstum noch etwas abändern und beispielsweise andere Pflanzen in der Nachbarschaft zulassen, das Grundproblem bleibt aber bestehen. Eine Zelle in einem Cellular Automaton kennt nur die Zu-

stände seiner direkten Nachbarn. Man kann diese Nachbarschaft erweitern so dass beispielsweise 24 Zellen um die aktuelle Zelle geprüft werden, das ist allerding wenig praktikabel weil es den Rechenaufwand enorm erhöht. Zudem wäre das nur eine Teillösung die weitere Fragen aufwirft. Ein weiteres Problem mit dem vorgestellten Ansatz ist, dass ein Entwickler so gut wie keinen Einfluss auf die Gebiete hat in denen Vegetation entsteht.

4.4.2 Waldzonen

Gerade die Verteilung von Bäumen und Wäldern ist für manche Spielgenres aber wichtig. Holz ist in den meisten Strategiespielen eine, vom Spieler benötigte Ressource und die taktische Positionierung von Wäldern im Level daher eine, für die Spielmechanik kritische Notwendigkeit. Abgesehen davon wachsen Bäume in der Realität nicht nur entlang von Gewässern. Ein Lösungsansatz ist die *Waldzone*.

Zur *Block* Klasse wird ein Boolean *forestZone* hinzugefügt. Der Wert gibt an ob sich ein Block in einer Waldzone befindet oder nicht. Wenn vom System ein Baum gepflanzt wird, überprüft dieser ob er sich auf einem solchen Gebiet befindet. Wenn nicht, dann nimmt der entsprechende Block wieder einen anderen Zustand an. Mit der Einführung der Zonen können Blocks angeben werden auf denen Bäume wachsen dürfen. Der Baumzustand überprüft zusätzlich noch ob sich in der Umgebung andere Bäume befinden – so wie im reinen Cellular Automata Ansatz – und ob die Zelle von Nachbarn umgeben ist die auch in einer Waldzone sind. Letzteres wird verwendet um die Population am Rand eines Waldes etwas einzuschränken. Wenn weniger als fünf Zellen Waldzonen sind, ist die Wahrscheinlichkeit für einen Baum geringer.

Damit der Designer nicht alle Zonen selber auslegen muss, erstellt ein Algorithmus einige Waldzonen. Dieser funktioniert grundlegend wie der Algorithmus für die Levelgenerierung. Das Baumwachstum orientiert sich nun nicht mehr so sehr an den Gewässern und sieht natürlicher aus, wie Abb. 4.7 zeigt. Wenn man die Bilder vergleicht, kann man feststellen dass in manchen Bereichen der lilafärbigen Waldzonen nichts wächst. Bäume können nun nur noch dort existieren wo eine solche Zone festgelegt und ausreichend Feuchtigkeit vorhanden ist. Natürlich spielen in der Realität viele andere Faktoren eine Rolle und es ist dem Entwickler freigestellt noch weitere Bedingungen in die Zustandsobjekte zu implementieren. Denkbar wäre eine Wüsten-Waldgrenze ab der nur noch bestimmte Pflanzen leben können. Wenn eine dritte Dimension verwendet wird ist auch eine alpine Waldgrenze mit wenig Aufwand möglich.

Die Bäume wachsen nun in Gruppen die einem Wald ähneln, die Verteilung der Bäume in diesen Gruppen ist aber immer noch sehr regelmäßig und wirkt nicht überzeugend. Ein zweites Konzept soll dieses Problem beheben, Bäume wachsen nicht willkürlich aus Gras-Blocks sondern verbreiten sich

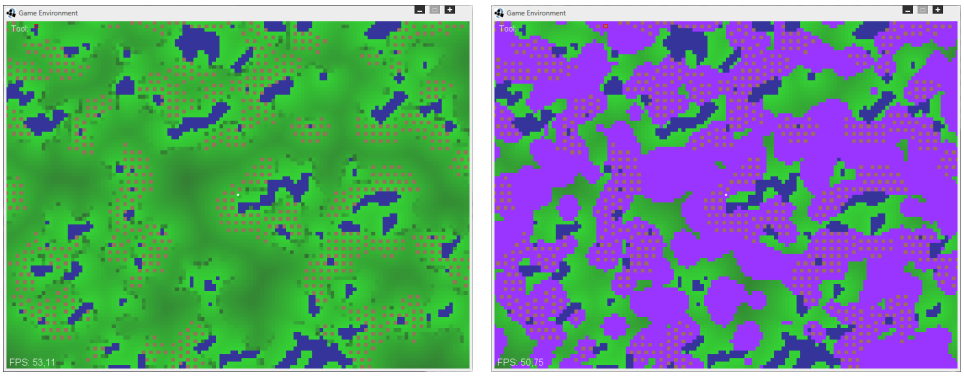

Abbildung 4.7: Baumwachstum in Waldzonen

mithilfe von Samen.

4.4.3 Seeds

Es gibt drei mögliche Wege für die Verbreitung von Pflanzensamen:

- Verbreitung durch Wind
- Verbreitung durch Wasser
- Verbreitung durch Tiere

Für die erste Implementierung des Konzepts wurde die Verbreitung durch Wind gewählt. Die Idee ist einen Samen einer bestimmten Pflanze auf einen Block zu setzen, wenn der Block die Voraussetzungen für die Pflanze erfüllt kann sie in der Zelle wachsen. Die Bedingungen legt zum einen der Samen selbst fest und zum anderen der *BlockState* der Pflanze. Das *Seed* Objekt hat unter anderem eine Liste an Blocks auf denen es wachsen kann, den *BlockState* zu dem es erwachsen kann und eine Lebensspanne. Da es durch den Wind verstreut wird, hält es außerdem einen Wert für die Flugdauer. Wenn eine Pflanze einen *Seed* des selben Typs erzeugt, dann errechnet das Samenobjekt seine Position so lange neu bis die Flugdauer zu Ende ist. Bei der Positionierung richtet es sich nach den Richtungsvektoren der Blocks in der Luftebene, dem Äquivalent des Systems für Wind. Sobald es nicht mehr fliegt, wird aus dem Block der Bodenebene der die Position mit dem Samen teilt, die Pflanze die im *Seed* festgelegt ist. Allerdings nur dann wenn der Zustand des Blocks in der Liste an möglichen Gebieten aufgeführt ist auf denen die Pflanze wachsen kann.

Abb. 4.8 zeigt dass die Verbreitung der Baumsamen durch Luft ein wesentlich realistischeres Ergebnis erzielt. Hier wurden Samen zu Beginn der Simulation zufällig über die ganze Welt verteilt, manche Samen landen auf der richtigen Stelle, wachsen zu einem Baum heran und produzieren selbst Samen die über den Wind verbreitet werden. Nicht alle potenziellen Waldzo-

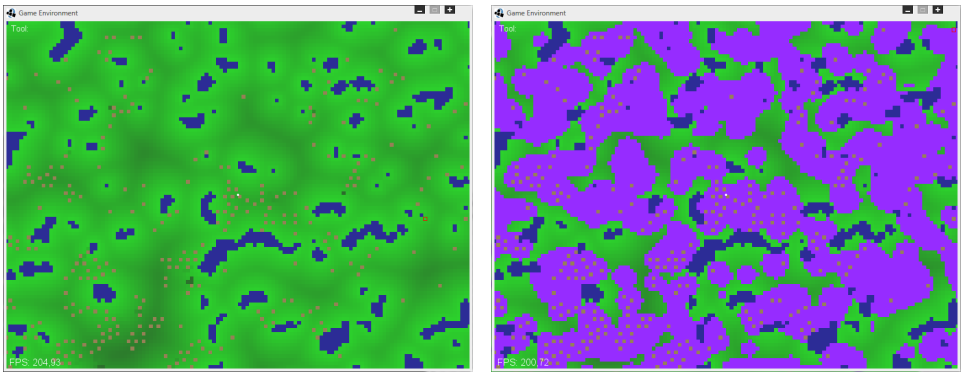

Abbildung 4.8: Baumwachstum mit *Seed* Konzept

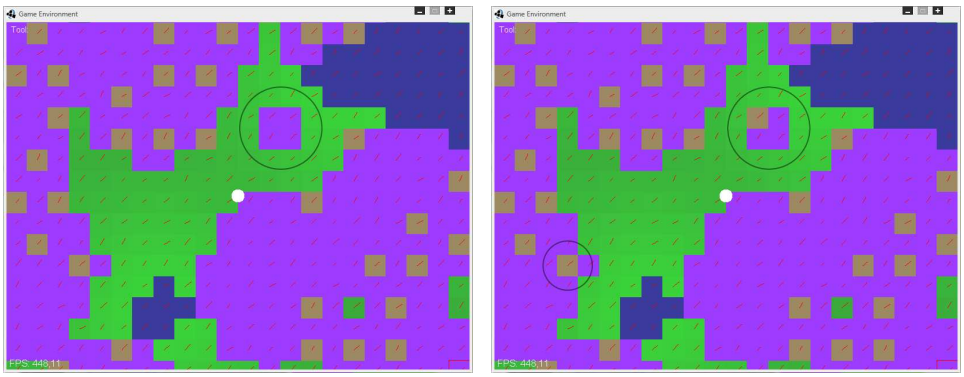

Abbildung 4.9: Verbreitung der Samen durch Wind

nen sind vollständig bewaldet und in manchen, kleinen Zonen wächst nichts weil sie entweder zu trocken oder die Windverhältnisse ungüstig sind.

Es kommt vor dass zu Beginn der Simulation keine Samen auf die Blocks kleinerer Waldgebiete gesetzt werden. In diesem Fall wachsen keine Bäume in diesem Gebiet bis ein *Seed* innerhalb der Zone landet. Abb. 4.9 zeigt eine kleines unbewaldetes Gebiet das bei der Levelgenerierung keinen Samen bekommen hat. Nach einigen Zeitschritten landet ein Samen auf einem der Blocks und wird zu einem Baum. Der Ursprung des *Seed* lässt sich anhand der roten Richtungsvektoren der Windebene auf einen Baum links unten zurückführen.

Mit den Waldzonen, dem *Seed* Konzept und der Verteilung durch Wind ergibt sich eine Landschaft die schon weit natürlicher wirkt als eine die mit einem reinen Cellular Automata Ansatz erstellt werden kann. Die Vermehrung der Bäume geschieht automatisch, der Designer muss nicht jeden einzelnen Baum selbst pflanzen. Aber sie richtet sich auch nach Regeln die vom Designer gemacht werden und erlaubt diesem die Waldzonen selbst festzu-

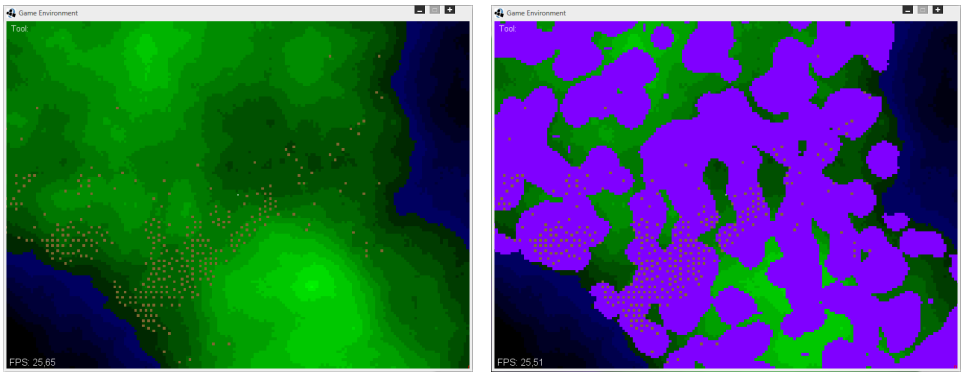

Abbildung 4.10: Die Baumgrenze und die zugehörigen Waldzonen

legen. So hat er ausreichend Einfluss auf die Vegetation um sie an wichtigen Stellen der Welt taktisch oder stilistisch zu positionieren, muss sich aber nicht selbst um den Rest der Vegetation kümmern.

4.4.4 Baumgrenze

Wie zuvor schon angedeutet können mithilfe einer Heightmap nicht nur eine realistischere Aufteilung der Spielwelt sondern auch eine überzeugendere Vegetation erreicht werden. Eine Möglichkeit ist die Einführung einer Höhengrenze ab der keine Bäume mehr wachsen. Konkret wird dabei eine weitere Konstante *MAX_HEIGHT* in das State-Objekt des Baums implementiert. Außerdem wachsen manche Pflanzen erst ab einer gewissen Höhe, also kommt noch eine weitere Konstante *MIN_HEIGHT* dazu.

Bei der Fortpflanzung werden wieder Seeds verwendet die auf beliebige Blocks innerhalb des Grids fallen können. Ein Seed versucht den Zustand des Blocks auf dem er liegt in den Zustand der Pflanze zu ändern von der er abstammt. Im Falle des Baums im Test, liegt *MIN_HEIGHT* bei 50 und *MAX_HEIGHT* bei 150. Auf Abb. 4.10 ist zu sehen dass sich der Großteil der Vegetation in niedrigeren Regionen ansiedelt. In den tiefsten Landgebieten, zu erkennen am dunkelsten Grünton, wächst kein Baum da diese unter 50 Höheneinheiten liegen. Im rechten Bild sind die Waldzonen zusätzlich eingezeichnet um zu zeigen dass durch die Höhenbeschränkung viel weniger Platz innerhalb der Zonen mit Bäumen gefüllt ist. Das Gebiet in dem tatsächlich etwas wächst ist die Schnittmenge aus Blocks die zwischen 50 und 150 Einheiten auf der z–Achse liegen und den Waldzonen. Nicht zu vergessen die ausreichende Feuchtigkeit des Bodens die ebenfalls eine Bedingung für Wachstum ist.

4.4.5 Umgebungsfaktoren

In [1] führt Ch'ng ein Umgebungsobjekt mit drei Attributen ein. Diese sind Sonnenlicht, Temperatur und Wassergehalt der Erde. Die letzten zwei sind im Laufe der vorliegenden Arbeit bereits berücksichtigt worden, Sonnenlicht wurde noch zusätzlich in das *Environment* Objekt übernommen. Es wird als konstanter Wert angenommen da für das geringe Ausmaß der Testumgebung eine räumliche Variation keine Notwendigkeit ist. Obwohl das System zeitabhängig ist, gibt es keine definierten Jahres– oder Tageszeiten und somit spielen Schatten und der Stand der Sonne noch keine Rolle.

4.4.6 Genotyp der Pflanzen

Wie Ch'ng vorschlägt, wurde ein Genotyp für jede Pflanzenart implementiert. Dieser besteht aus den Optimalbedingungen für Sonnenlicht, Temperatur, Toleranz für Bevölkerungsdichte und Qualität der Erde. Dafür gibt es je einen unteren, einen idealen und einen oberen Wert. Außerdem beinhält der Genotyp die Energie, die Samenanzahl, das maximale Alter und das Fortpflanzungsalter der Pflanze. Implementiert sind diese Werte als Konstanten im Zustandsobjekt der jeweiligen Pflanze und sind für jedes Individuum einer Art gleich.

4.4.7 Bodenqualität

Um die Qualität der Erde spatial zu variieren wird wie bei der Heightmap für die Levelgenerierung der Diamond Square Algorithmus eingesetzt. Die Werte der erzeugten Heightmap liegen zwischen -255 und 255, zu diesem Zweck müssen sie allerdings auf eine Menge von Gleitkommazahlen zwischen 0 und 1 abgebildet und als Qualitätsvariablen in den Blocks gespeichert werden. Je größer der Wert desto fruchtbarer der Boden. Im Genotyp jeder Pflanze ist vermerkt unter welchen Bodenbedingungen diese leben kann, im Testfall kann ein Baum überleben wenn die Qualität der Erde bei mindestens 0.5 liegt. Abb. 4.11 zeigt das Wachstum der Bäume unter verschiedenen Bodenbedingungen. Die dunklen Bereiche haben weniger fruchtbaren Boden auf denen kein Baum wachsen kann. Nur die helleren, qualitativ hochwertigere Böden können die Pflanzen mit genug Nährstoffen versorgen. Zum Vergleich eine Aufnahme ohne Qualität der Erde sowie die entsprechenden Waldzonen in Abb. 4.12. Es ist zu erkennen dass die oben besagte Schnittmenge um ein Objekt erweitert wurde.

4.4.8 Artendiversität

Um die obigen Methoden zu testen wurde bisher immer nur eine Art Baum verwendet, doch um eine realitätsnahe Umgebung zu simulieren braucht es etwas mehr. In der Natur findet sich schon auf einem kleinem Raum eine

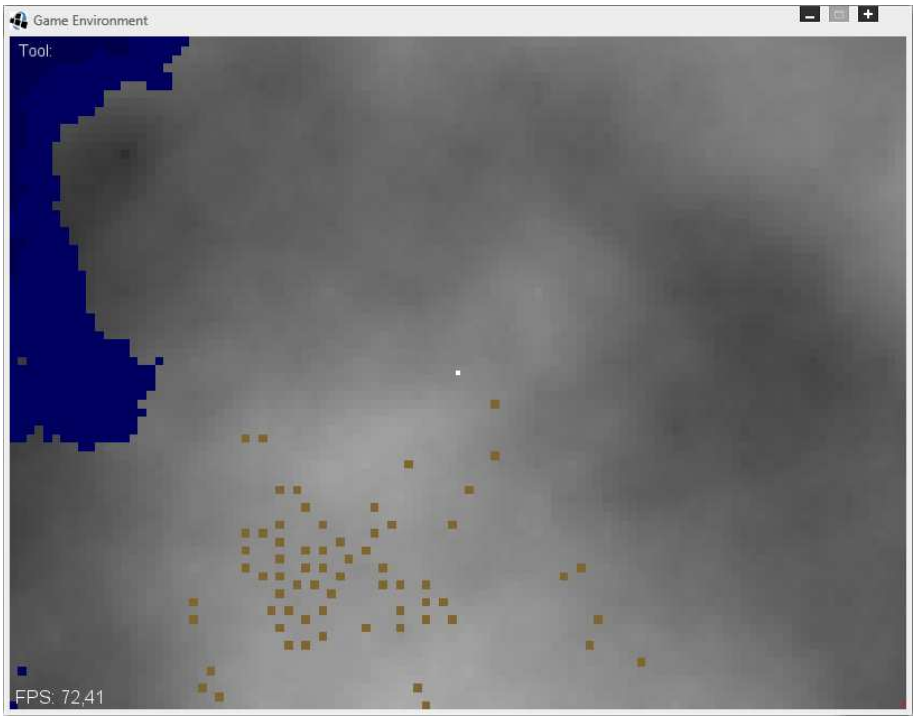

Abbildung 4.11: Vegetation auf qualitativ unterschiedlichen Böden

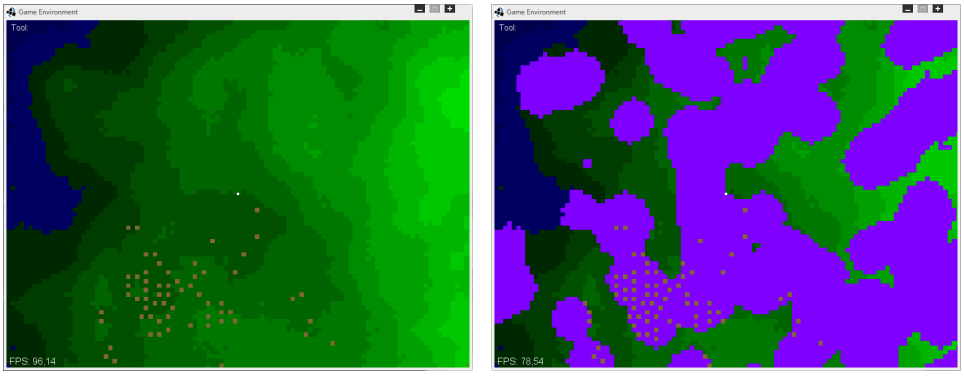

Abbildung 4.12: Höhen und Waldzonen

große Variation an Pflanzen, um konkurrierenden Arten zu simulieren reicht aber schon ein weiterer Baum mit einem anderen Genotyp. Durch die Architektur des Frameworks und den Einsatz der oben bereits besprochenen Regeln für Vegetation, reicht es aus eine zweite Baumklasse zu erstellen und die Konstanten des Genotyps zu ändern. Die beiden Arten konkurrieren automatisch um den verfügbaren Platz und somit auch um Ressourcen wie Bodenqualität und Feuchtigkeit. Für den Test wurden zwei Bäume gewählt

MAX_AGE	300
REPRODUCTION_AGE	5
SEED_COUNT	5
MIN_HEIGHT	50
MAX_HEIGHT	150
MIN_SOIL_QUALITY	0.5
RANGE	5

Tabelle 4.1: Genotyp Ahorn

MAX_AGE	200
REPRODUCTION_AGE	7
SEED_COUNT	3
MIN_HEIGHT	20
MAX_HEIGHT	200
MIN_SOIL_QUALITY	0.3
RANGE	7

Tabelle 4.2: Genotyp Birke

deren Samen sich großteils mit dem Wind verbreiten, Ahorn und Birke. Der Genotyp für Ahorn ist in Tabelle 4.1 und der für Birke in Tabelle 4.2 zu finden. Ein Ahornbaum hat schneller das Reproduktionsalter erreicht und ist langlebiger, dafür haben die Samen einer Birke eine weitere Reichweite und können in kleineren und in größeren Höhen Wurzeln schlagen als Ahornsamen. Das Ergebnis einer Simulation ist auf Abb. 4.13 zu sehen, Ahorn ist dabei in braun gehalten und Birke in einem helleren Braunton. Es ist eine leichte Dominanz der Birke zu erkennen, höchstwahrscheinlich weil sie auch in Gebieten leben kann in denen Ahorn chancenlos ist. Sie hat nicht nur einen Vorteil wegen der höheren Baumgrenze sonder auch wegen der höheren Toleranz für Bodenqualität.

Weitere Pflanzenarten können auf gleichem Weg in die Simulation eingefügt werden, der Entwickler mus lediglich weitere Genotypen definieren.

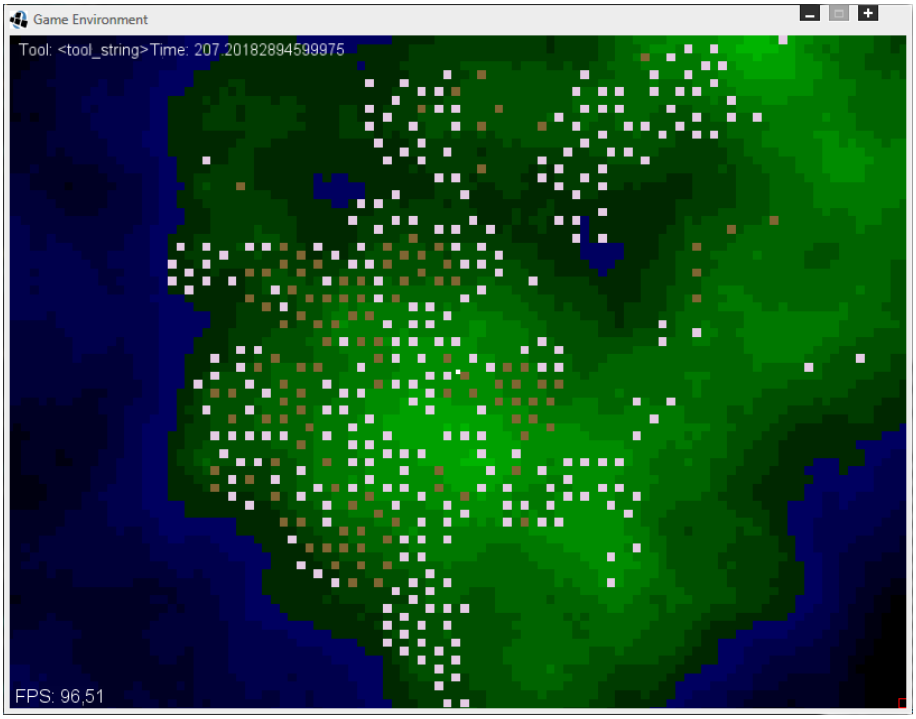

Abbildung 4.13: Konkurrenz von Ahorn und Birke

Kapitel 5

Diplomprojekt

5.1 Zielsetzung

Das Ziel des Diplomprojekts war ein Framework zu bauen das einem Entwickler die grundlegenden Funktionen zu Verfügung stellt um eine Welt, basierend auf einem CA Ansatz zu gestalten. Konkrete Implementierungen für reale Objekte gibt es nicht. Die Frage ist, welche Dinge schon zu Beginn bereitgestellt werden sollen und was vom Entwickler selbstgemacht werden muss.

5.1.1 Planung

Wichtig bei der Planung des Frameworks war, dass die Handhabung der zellularen Automaten für den Entwickler möglichst komfortabel ist. Es wird eine zentrale Stelle benötigt die alle Zellen speichert und verwaltet. Außerdem muss der Entwickler bequem und sicher auf die Zellen zugreifen können und Werte auslesen sowie verändern. Deshalb sind alle Zellen Instanzen der Klasse *Block* und über besagte zentrale Stelle aufrufbar. Die Verhaltensweisen und Regeln der Blocks sollen einfach austausch –und erweiterbar sein, darum kommt das Konzept der *State Machine* zum Einsatz. Das System ist auf 2D Welten ausgelegt, es soll aber trotzdem möglich sein zwischen verschiedenen Ebenen zu unterscheiden. Es soll im Spiel Agenten geben und diese sollen mit der Welt interagieren können. Da das direkte Verändern von Werten der zellulären Automaten zu unvorhersehbarem und instabilem Verhalten führen kann soll es eine sichere, indirekte Möglichkeit dafür geben. Die wichtigsten Anforderund sind daher:

- Thread für die Verwaltung aller Zellen,
  Speichern und Bereitstellen der Blocks,
  Aktualisieren der Welt,
- Grundelement Block für uniformen Zugriff,
- Strukturierung der Verhaltensweisen als State Machine,

Einfaches Austauschen und Hinzufügen,
Einfaches Erweitern der Funktionalität,

- System zur Unterscheidung zwischen Ebenen,
- Möglichkeit für sicheres, indirektes Verändern der Welt.

5.2 Technik

Das Projekt wurde vollständig in Java umgesetzt. Als Basis dient dabei Cogaen, eine Engine für Java Games die sich der LWJGL für die grafische Ausgabe und für Input bedient.

5.2.1 Cogaen

Cogaen steht für Component-based Game Engine und ist eine Engine für alle Spielgenres. Das Framework verwendet die Version 3.1. In dieser Version existieren zwei Module: Der *Cogaen Core* der alle essentiellen Teil der Engine enthält und *Cogaen LWJGL* der die *Light Weight Java Game Library* für visuellen und audiellen Output sowie für User Input verwendet. Das *Core* Objekt enthält die grundlegende Spiellogik und wird allen Teilen des Spiels mitgegeben.

Ein Konzept das für das Realisieren des CA wichtig war ist das *Cogaen Service*. Es erzeugt einen neun Thread der einmal gestartet, beliebig pausiert und wieder aufgenommen werden kann. Außerdem können für ein *Cogaen Service* Abhängigkeiten definiert werden ohne die es nicht arbeiten kann. Auf alle Services kann über den *Core* zugegriffen werden.

5.3 Architektur

Das ganze System basiert auf der Annahme dass ein zellularer Ansatz für die Simulation einer Spielwelt sinnvoll ist. Sinnvoll in Hinsicht auf performante Berechnung und praktikable Verwendung in Spielen. Die kleinste Einheit in der Welt ist ein Block, in der Repräsentation als farbiges Rechteck dargestellt. Es ist zweckdienlich sich diesen Grundbaustein als ein Quadrat mit dem Flächeninhalt von einem Quadratmeter vorzustellen. Ein solcher Baustein kann aus Erde, Gras, Wasser und anderen Materialien bestehen. Die Idee ist, diese Bauklötze aneinander zu reihen und so eine Landschaft aus Flüssen und Wiesen zu bauen. Je mehr Materialien zur Verfügung stehen desto komplexer kann die Welt werden die damit im Baukastenprinzip entsteht.

Wie in unserer Welt stehen die Teile der Spielwelt in ständiger Wechselwirkung miteinander, jeder Block beinflusst seine unmittelbaren Nachbarn. Das ist die Grundvorraussetzung für ein funktionierendes Ökosystem und einer der wichtigsten Bestandteile der Simulation. Ein Block aus Erde saugt

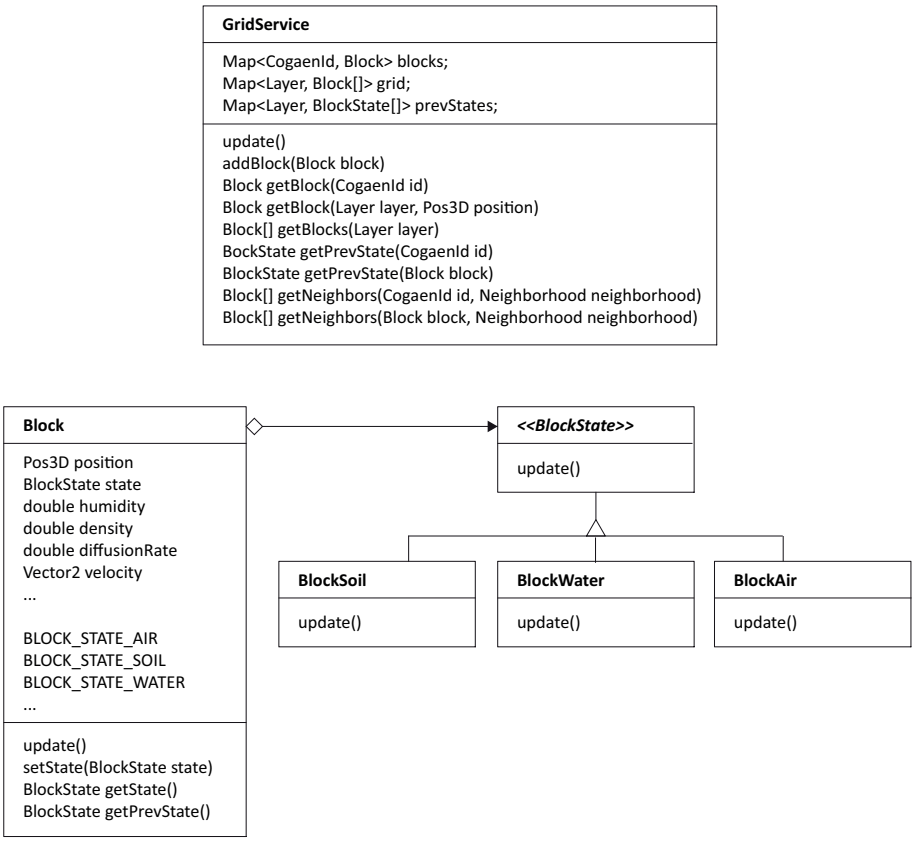

Abbildung 5.1: Architektur der State Machine

Wasser von einem benachbarten Wasser-Block auf und kann dadurch zu einem Gras-Block werden. Ebenso könnte das im Gras-Block gespeicherte Wasser verdampfen wenn ein benachbarter Baum-Block brennt und somit viel Hitze abgibt. Die Anzahl der Blocks in der Welt ist, zumindest im 2D-Raum konstant, deshalb empfiehlt es sich zu Beginn der Simulation alle Blocks zu erzeugen und je nach Material ein State Objekt zuzuweisen. Ändert sich der Zustand des Bausteins, so wird einfach dieses Objekt ausgetauscht.

Das Aktualisieren der Blocks folgt dem State-Machine Prinzip, jeder Block hat eine *update* Methode welche die *update* Methode des States aufruft. Im State wird auf verschiedene Bedingungen überprüft und wenn notwendig ein Übergang zu einem anderen State vorgenommen. So kann sich jeder Block seinen Zustand für den nächsten Schritt ausrechnen. Außerdem ist so das Verhalten der einzelnen Zustände in eine Klasse gekapselt die problemlos ausgetauscht werden kann. Abb. 5.1 ist ein Klassendiagramm das nur die für die Architektur der State Machine wichtigen Klassen, Methoden und Variablen enthält.

Das Ziel des Projekts ist die durchgehende Simulation einer Spielwelt, deshalb wird die *update* Methode jedes Blocks zu jedem Zeitschritt einmal aufgerufen. Diese Aufgabe übernimmt der *GridService* der alle Blocks in einer Liste hält und iterativ aktualisiert. Der Service ist als Verwalter und Motor der Welt zu sehen, er aktualisiert die Blocks in festgelegten Zeitschritten und in der korrekten Reihenfolge, stellt den States auf Abfrage die Blocks ihrer Nachbarschaft zur Verfügung und übernimmt noch ein paar andere Aufgaben. Ohne ihn würde die Welt still stehen. Wenn dieser Service zum Zeitschritt *t* einen Block aktualisiert, ruft der Block die *update* Methode des von ihm gehaltenen States auf und gibt sich selbst als Parameter mit. Der State braucht in den meisten Fällen die Zustände der umliegenden Blocks um seinen neuen Zustand berechnen zu können. Ein Zugriff auf die *getNeihgbors(Neighborhood neighborhood)* des GridServices gibt eine Liste der Nachbarblocks zurück. Zum Berechnen des neuen Zustands darf der State aber nicht den Zustand der Nachbarblocks zum Zeitschritt *t* sondern den zum vorhergehenden Zeitschritt *t-1* verwenden. Denn sonst verwendet er zum Teil schon aktualisierte und zum anderen Teil nicht aktualisierte States. Die Blocks halten nur ihren aktuellen Zustand, deshalb muss der GridService die States für *t-1* speichern und mit *getPrevState(CogaenId id)* sowie mit *getPrevState(Block block)* abrufbar machen.

5.3.1 Grid

Das Grid ist ein Konzept das die Gesamtheit der Blocks auf einer zweidimensionalen Ebene beschreibt und könnte auch als das Level bezeichnet werden. Es besteht aus quadratischen Blocks die lückenlos zu einem Gitter zusammengesetzt sind. Die Welt ist ein Rechteck, zumindest in diesem System. Das Grid ist endlich aber die Spielwelt unendlich denn der obere Nachbar eines Blocks in der obersten Reihe ist der Block an entsprechender Stelle in der untersten Reihe. Geht man oben hinaus so kommt man unten wieder rein. Die Welt hat durch diesen Zusatz die Form eines Torus.

5.3.2 Grid Service

Der Grid Service ist ein *CogaenService* der die Verwaltung und Aktualisierung aller Blocks zur Aufgabe hat. Zwei Hash Maps halten alle Elemente des Grids und stellen den Zugriff auf die Blocks sicher. *Map<CogaenId, Block> blocks* hält alle Blocks mit der jeweiligen *CogaenId* als Schlüssel, wobei die Werte in keiner speziellen Reihenfolge geordnet sind. *Map<Layer, Block[]> grid* enthält mehrere eindimensionale Arrays die jeweils ein Gitter aus Blocks in geordneter Reihenfolge halten. Das erste Element ist dabei der Block links oben in der Welt und das letzte Element entspricht dem Block rechts unten. Jedes Array entspricht einer Ebene der 2D Welt, so kann besipielsweise eine Ebene für Luft und eine für den Boden verwendet werden.

dem Alle Instanzen der *Block* Klasse registrieren sich beim Grid Service und werden entsprechend der Ebene und Position in die Maps eingetragen. Das soll nur beim Start der Simulation passieren da in der Regel Blocks nicht gelöscht oder verschoben sondern nur deren Zustände angepasst werden.

Aufgaben des Grid Service:

- Verwalten und Bereitstellen der Blocks,
- Bereitstellen der Nachbarschaft für jeden Aktualisierungsschritt,
- Aktualisieren des gesamten Grids,
- Bereitstellen der Zustände des Grids im letzten Zeitschritt.

5.3.3 Block

Jedes Element der Spielumgebung ist eine Instanz der *Block* Klasse. Diese hält eine Vielzahl von Werten die den physischen Zustand des Blocks beschreiben, darunter Dichte des Materials, Temperatur, Position und andere. Ergänzend besitzt der Block eine Variable des Typs *BlockState* in der das Verhalten des Blocks festgelegt ist und außerdem verwendet wird um den Zustand des Blocks eindeutig zu identifizieren. Bei einem Zustandswechsel, beispielsweise Erde zu Wasser, kann es sein dass sich Werte wie Dichte oder Temperatur ändern, deshalb wird nach dem Wechsel immer die *setup()* Methode des neuen BlockStates aufgerufen. Diese passt dann die Werte des zugehörigen Blocks entsprechend an. Der Block speichert auch Werte aus dem vorangegangenen Zeitschritt da sie für einige Berechnungen gebraucht werden.

Aufgaben des Blocks:

- Grundelement der Spielumgebung,
- Hält Werte die den physischen Zustand beschreiben,
- Hält alle Implementierungen von *BlockState*.

5.3.4 Block State

Die *BlockState* Klasse ist ein Interface das von jedem möglichen Blockzustand der Simulation implementiert wird. Die *update()* Methode wird vom Block aufgerufen und enthält die Bedingungen für einen Zustandswechsel desselben. Wenn eine Bedingung erfüllt wird, tauscht sich der Block State selbst mit einem anderen aus. States sind *Flyweights* die keine Variablen enthalten, deshalb können die Werte für den physischen Zustand des Blocks nicht in einer Implementierung der *BlockState* Klasse gespeichert werden. In [4, S. 196] ist ein *Flyweight* so definiert:

> A flyweight is a shared object that can be used in multiple contexts simultaneously.

Der Vorteil davon ist dass für jeden möglichen Zustand nur eine Instanz des State Objekts existieren muss das statisch vom Block gehalten wird und den Speicheraufwand so drastisch reduziert.

Aufgaben des Block State:

- Berechnet Zustandswerte für den aktuellen Zeitschritt,
- Geht in einen anderen Zustand über wenn notwendig.

5.3.5 Layer

Layer sind Werte in einer Enumeration die es dem System ermöglichen zwischen verschiedenen Ebenen zu unterscheiden. Für jeden Wert existiert ein Eintrag in *Map<Layer, Block[]> grid*, jede dieser Ebenen repräsentiert also ein Grid. Zudem wird die Reihenfolge der Einträge verwendet um eine horizontale Ordnung für die Darstellung aller Werte in der Map zu finden. Das kann, wie im Prototyp, hilfreich sein um Wolken zu zeichnen oder um spezielle Objekte auf einer eigenen Ebene zusammenzufassen. Das Ebenenkonzept kann als zweidimensionales Äquivalent für eine dritte Raumachse verstanden werden, kann aber auch zur Strukturierung der Spieleelemente verwendet werden.

Die Implementierung ist im vorliegenden Prototyp allerdings noch nicht vollständig ausgereift. Ein Problem ist unter anderem noch die Selektion der Blocks. Da das System auf eine Draufsicht ausgelegt ist kommen bei der Auswahl eines Blocks immer alle Blocks mit selber Position auf allen Ebenen in Frage. Ein Ansatz wäre den Block zurückzugeben der auf der Ebene des Spielers liegt. Eine andere Lösung könnte sein dem Entwickler eine Liste mit allen möglichen Blocks zu geben, dieser müsste dann selbst entscheiden welche die richtig Auswahl ist.

Aufgaben der Layer:

- Dient zur Unterscheidung zwischen verschiedenen Grid-Ebenen,
- Kann bei Bedarf ausgeblendet werden.

5.3.6 Tool

Ein *Tool* ist ein Objekt das ein Agent in der Spielwelt besitzen kann. Angelehnt an die reale Welt ist ein solches Werkzeug für eine spezifische Aufgabe zuständig. Das *Tool* kapselt eine Interaktion zwischen Agent und Welt als Objekt welches dem Ersteren zugewiesen werden kann. Der Agent hat damit eine direkte Möglichkeit die Welt zu verändern und zwar nur in der Form in der es das Werkzeug erlaubt. Er kann zum Beispiel eine Schaufel verwenden um ein Loch zu graben. Um die Kompetenzen des Agenten zu erweitern reicht es aus ein weiteres *Tool* zu seinem Set an Werkezeugen hinzuzufügen. Das kann auch zur Laufzeit passieren und ermöglicht es beispielsweise dem Spielercharakter ein neues Werkzeug in der Spielwelt zu finden und benutzen.

Das System verwendet die komponentenbasierten Entitäten der *Cogaen*. Das heisst dass Entitäten nicht in einer klassisch objektorientierten Hierachie von immer konkreteren Implementierungen spezifiziert sondern aus mehreren wiederverwendbaren Modulen zusammengesteckt werden. *Cogaen* unterscheidet zwischen Attributen und Komponenten, wobei Attribute Variablen darstellen auf die die Komponenten zugreifen können. Der Vorteil ist dass Entitäten rasch aus Komponenten zusammengesetzt werden können, die für deren Funktionsweise relevant sind. Ein statisches Objekt mit einem Positionsattribut beispielsweise, kann durch das Hinzufügen einer *MotionComponent* Komponente beweglich gemacht werden. Ebenso benötigt ein Agent die Komponente *ToolComponent* um ein Werkzeug, bildhaft gesprochen in die Hand zu nehmen. Dieses Modul speichert das Werkzeug das der Agent zu einem Zeitpunkt in der Hand hält und stellt sicher das nicht mehrere gleichzeitig verwendet werden. Trotzdem kann er verschiedenen *Tool* Objekte besitzen und mitführen. Diese sind in einem Attribut der Entität als *ArrayList<Tool>* abgelegt. Der Spielercharakter kann also ein Schaufel in der Hand halten während andere Werkzeuge im Inventar auf ihren Einsatz warten. Ein *Tool* wird benutzt wenn ein *ToolUseEvent* geworfen wird auf welches das *ToolComponent* Objekt hört. Dort wird überprüft ob die Entität im Besitz der entpsrechenden Instanz des Werkzeugs ist und die *use()* Methode desselben aufgerufen. In dieser Methode sitzt die eigentliche Funktion des Werkzeugs aber es werden noch keine unmittelbaren Eingriffe in die Spielwelt vorgenommen. Das *Shovel* Objekt legt darin fest dass ein Loch an der Stelle des Spielers gegraben wird aber manipuliert das Grid noch nicht selbst. Damit die Veränderung in der Welt nochmals eigens abgekapselt und zumindest theoretisch reversibel ist, kommt ein *Action Pattern* zum Einsatz.

Aufgaben des Tool:

- Einfaches Verwalten der Kompetenzen von Agenten,
- Indirektes Einflussnehmen auf die Welt.

5.3.7 Action

Eine *Action* oder auch *Command* ist ein Entwurfsmuster dessen Zweck [4, S. 233] so beschreibt:

> Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Weil eine Aktion so wenig sein kann wie eine Variable eines Blocks zu verändern aber auch komplexere Anweisungen enthalten kann, empfiehlt es sich diese in ein einheitliches Objekt zu kapseln. Was eine Aktion ist und wo sie aufhört entscheidet der Entwickler aber es empfiehlt sich sie elementar zu halten. Für komplexere Aufgaben können entsprechende Werkzeuge zur Verfügung gestellt werden, die mehrere Aktionen hintereinander ausführen.

Will der Spieler nun ein Loch graben, so muss das *Shovel* Werkzeug angewählt sein wenn das *ToolUseEvent* geworfen wird. Das *ToolComponent* fängt das Event und weist die Schaufel an die Aktion *DigAction* aufzurufen welcher die gewünschte Position des Lochs mitgegeben wird. Die Aktion zwingt darauf den Block an entsprechender Stelle den Zustand zu ändern. Da der Prototyp zweidimensional ist wird ein Block der auf dem gegraben wurde zu einem Erdblock. In einer dreidimensionalen Welt kann der Block mit dem höchsten y Wert einfach entfernt werden. Natürlich kann mit einer Schaufel nicht in jedem Material gegraben werden. Bedingungen dieser Art müssen im *Tool* festgelegt werden und erst dann dürfen die zugehörigen Aktionen ausgeführt werden.

Theoretisch können auch alle Anweisungen an die Welt gespeichert werden und mit etwas Aufwand rückgängig gemacht werden. Da sich die Welt aber ständig selbst verändert macht das nicht viel Sinn. Die Spieleraktionen zu speichern kann sich aber durchaus als hilfreich herausstellen.

Aufgaben der Action:

- Kapselung für Agent-Welt-Interaktion,
- Speichern aller Aktionen,
- Modulare Zusammenstellung komplexer Aufgaben.

5.4 Performance

5.4.1 Neighborhood

Im Prototyp werden zwei Nachbarschaftsgrößen verwendet. Zum einen die Moore Nachbarschaft mit acht umgebenden Blöcken und zum anderen die Van Neumann Nachbarschaft mit vier Nachbarn. In der Regel sind Updates mit mehr Nachbarn teurer im Bezug auf benötigte Rechenzeit. Der Einfluss auf die Performance ist aber nicht direkt abhängig von der Anzahl an Nachbarblöcken. In der *update* Methode der States wird meist ein Schleifendurchlauf pro Nachbar benötigt, die Zeit eines Durchlaufs ist aber abhängig von den Anweisungen in der Schleife die von State zu State stark variieren können.

5.4.2 Updates und Updateraten

Die Performance des Prototypen ist stark abhängig von der Anzahl der Blocks in der Welt da in jedem Update-Schritt die *update* Methode jedes Blocks aufgerufen wird. Der GridService ist zuständig für den Zeitintervall in dem der Zustand aller Blocks neu berechnet wird, er kann beispielsweise alle 10 ms ein Update ausführen. Wie oft die Welt aktualisiert werden sollte ist dem Entwickler überlassen, die meisten Spielegenres haben hier unterschiedliche Ansprüche: Schnelle Shooter brauchen hohe Updaterates weil Präzision und Spielerreaktion eine enorm wichtige Rolle spielen. Man stelle

sich ein Szenario vor in dem sich ein Spieler hinter einer Barriere versteckt um nicht von einem Gegner getroffen zu werden. Der gegnerische Spieler zerstört die Barriere und der bedrohte Spieler weicht als Reaktion darauf weiter zurück. Wenn man die Barriere als einen Block betrachtet der vom Zustand *Holz* zu *Luft* wechselt dann ist eine Update-Intervall von 10ms oder weniger durchaus gerechtfertigt damit der Spieler rechzeitig reagieren kann. Das Wachstum eines Weizenfelds in einer Bauernhof-Simulation wie *Harvest Moon* kann auch mit einem längeren Intervall von mehreren Sekunden oder Minuten zweckdienlich berechnet werden.

Kapitel 6

Zusammenfassung

Diese Arbeit hat einige Möglichkeiten aufgezeigt um die Simulation einer Spielwelt zu berechnen, dabei ist darauf geachtet worden die Bereiche abzudecken die für eine möglichst große Zahl an potentiellen Spielen relevant sind. Sowohl für die Architektur als auch für die erarbeiteten Techniken sind viele Abänderungen und Verbesserungen denkbar.

6.1 Weitere Verbesserungen

6.1.1 Vegetationszonen

Es wurden bisher Bäume und Wälder stellvertretend für die gesamte Vegetation verwendet. Waldzonen lassen sich aber recht einfach in Vegetationszonen umwandeln, dazu bräuchte ein Block statt dem Boolean *forestZone* ein Objekt in dem eine Vegetationszone angegeben ist. Diese würde wiederum angeben welche Pflanzen in dem Gebiet leben können und alle anderen Samen einfach ignorieren. So könnte der Entwickler relativ einfach festlegen welche Art von Vegetation in welchen Zonen enstehen soll.

6.1.2 Artenverteilung

Es könnte sein dass verschiedene Arten von Bäumen in einem Wald gewünscht werden. Ein Mischwald soll beispielsweise zu 80% aus Laub– und zu 20% aus Nadelbäumen bestehen. Um das zu realisieren könnte der *GridService* eine Liste an Vegetationszonen führen die zusätzlich die aktuelle, prozentuelle Verteilung der Arten enthält. Ein *Seed* Objekt würde dann bevor es einen Nadelbaum setzt, überprüfen ob sich der Anteil noch unter 20% befindet. Wenn nicht, wird mit einem Laubbaum entgegengesteuert.

6.1.3 Verbreitung von Seeds

Wasser und Tiere sind auch Möglichkeiten um Samen in der Welt zu verbreiten. Wasser ist einfach zu implementieren, anstatt die Richtungsvektoren der Luftblocks werd die der Wasserblocks verwendet. Der Samen schwimmt mit der Strömung bis er auf Land trifft, dann wird auf alle Bedingungen überprüft und wenn diese erfüllt sind kann eine Pflanze wachsen.

Für die Verbreitung durch Tiere muss etwas mehr implementiert werden. In der Natur hat diese Art von Samen meist eine Möglichkeit sich in Fell oder auf Stoff festzuharken und sich ein Stück mittragen zu lassen. Das würde sich simulieren lassen indem ein Samen von eine vorbeigehendem Tier in eine Liste von Samen gespeichert wird und nach einer Zeit, an der Position des Tieres von selbst wieder abfällt.

6.1.4 Verwendungszwecke für Seeds

Seeds können theoretisch auch für andere Zwecke als die Verbreitung von Vegetation eingesetzt werden. Nachdem jeder Samen selbst bestimmt welche Auswirkungen er auf die Umgebung hat, ist es damit möglich einen beliebigen Befehl an einer bestimmten Stelle der Welt und zu einer beliebigen Zeit auszuführen. Das kommt dem *Scripting* sehr nahe das viele Spiele verwenden um die Handlung voranzutreiben oder Rätsel umzusetzen. Eine typische Situation die machbar wäre, ist dass der Spieler einen Schalter findet und betätigt und sich an anderer Stelle eine Tür öffnet. *Seed* Objekt sind also vielfältig verwendbar und können, wenn noch etwas Entwicklungszeit hineingesteckt wird dem Designer helfen eine noch lebendigere Welt zu schaffen.

6.1.5 Veränderbare Genotypen

Wenn der Genotyp von Zustandsobjekt einer Pflanze abgekapselt würde, könnte man diesen im Lauf der Simulation verändern. So könnte sich zum Beispiel eine Art an die äußeren Umstände anpassen und der Spieler könnte den evolutionären Vorgang beobachten. Der Konkurrenzkampf der Arten um die räumliche Dominanz würde so eine weitaus höhere Komplexität erreichen und die Vegetation im Spiel an Glaubhaftigkeit gewinnen.

6.1.6 3D Grid

Es ist theoretisch möglich das gesamte System um eine dritte Raumachse zu erweitern. Dazu müsste der Grid Service entsprechend angepasst und die Regeln in den Zustandsobjekten auf ihre Brauchbarkeit überprüft werden. Das Positionsobjekt der Blöcke enthält bereits einen Wert für die z–Achse, allerdings existiert für jede 2D Position nur ein Block. Das heisst dass für jede dieser Positionen Blocks für jeden z–Wert zwischen -255 und 255 weitere

Blocks erzeugt, verwaltet und simuliert werden müssen. Das würde unmittelbar zu einem intolerablen Performance–Einbruch führen.

Eine Lösung wäre, nur die Blocks an der Oberfläche zu jedem Zeitschritt zu aktualisieren und die darunter liegenden entweder zu ignorieren oder die Abstände zwischen den Aktualisierungen zu vergrößern.

6.1.7 Area Tiling

Um die Performance der Simulation zu verbessern könnte die Welt in *Tiles* aufgeteilt und diese seperat berechnet werden. Die *Tiles* die für den Spieler gerade sichtbar sind, würden vollständig simuliert werden, alle restlichen Bereiche könnten mit geringerer Geschwindigkeit berechnet werden. Das würde aber dazu führen dass in diesen Bereichen die Zeit weniger schnell läuft als in den sichtbaren Tiles. Deshalb wäre es von Vorteil eine Methode zu haben die das Verhalten eines Blocks über mehrere Zeitschritte, simplifiziert in einem Rechenvorgang simulieren kann.

Ein weiteres Problem ist die Synchronisation der Tiles, denn die äußeren Blocks eines Bereichs sind abhängig von den anliegenden Blocks des anderen Bereichs. Das ist vorallem für Regeln wichtig deren Effekt sich rasch ausbreitet, so wie Feuer oder Wasser. Es muss also eine Möglichkeite für Tiles geben, die Zustände ihrer Randbereiche zu kommunizieren. Dabei ist es sehr wichtig das die Zustände immer die des aktuellen Zeitschrittes sind.

6.1.8 Flexible Update Rates

Die Zeit zwischen den Aktualisierungsschritten aller Blocks ist zentral vom GridService festgelegt. Es wäre möglich unterschiedliche Intervalle abhängig von den Zuständen der Blocks zu bestimmen. Wasser könnte beispielsweise öfters berechnet werden als Erde. Das Einführen unterschiedlicher Aktualisierungsraten bedarf nur einer simpler Abfrage im GridService. Um noch einen Schritt weiter zu gehen können flexible Intervalle implementiert werden die von den Blocks selbst bestimmt werden. Nicht fließendes Wasser in einem kleinen Teich könnte die eigenen Updates auf eine Sekunde reduzieren indem der entsprechende Block nur eines von zehn Updates in der Sekunde zulässt. Allerdings hat diese Methode zur Folge das alle Blocks mit solchen flexiblen Update–Raten ein Timer Objekt benötigen. An diesem Punkt muss wieder der Entwickler entscheiden ob die Rechenleistung dafür verwendet werden soll.

6.2 Schlusssatz

Eine vollständig simulierte CA Welt hat den Vorteil dass alle Dinge die in statischen Leveln geskriptet werden müssen, vollständig ins System integriert werden können. Ein Ereignis ist also keine vom Entwickler definierte

Ausnahme sondern ein Teil der Spielwelt wie alle anderen auch. Der Nachteil daran ist die eingeschränkte Kontrolle über das genaue Verhalten dieser Teile. Obwohl die Kontrolle an vielen Stellen eines Spieles wichtig ist, muss der Entwickler akzeptieren dass die Landschaft nicht entworfen wird sondern anhand der gegebenen Regeln entsteht. Er entwirft diese Regeln und lässt der Entwicklung nach dem Start der Simulation freien Lauf. Sweetser schreibt, wie auch im Zuge dieser Arbeit festgestellt wurde [6, S. 77]:

> It is important to note that emergent systems based on cellular automata possess drawbacks and issues. Often, emergent systems require extra time in development to design and tune the rules and properties.

Durch die Verwendung von Cellular Automata wird die Spielwelt zur Spielmechanik, Spieler haben theoretisch soviele Interaktionsmöglichkeiten wie die Welt groß ist. Ein Nachteil ist der große Rechenaufwand für das CA System, ein nicht unerheblicher Teil davon geht alleine in die Aktualisierung der Welt. Deshalb müssen in der Zukunft noch Methoden entwickelt werden die Berechnungen günstiger machen oder Gebiete in denen sich der Spieler gerade nicht befindet, zusammengefasst berechnen. Eine denkbare Alternative ist eine Kombination aus herkömmliche Methoden und CA Ansätzen die dort zum Einsatz kommen wo es spielmechanisch Sinn macht.

Die erarbeiteten Methoden für die Generierung der Welt, Hitze und Feuer, Fluide und die Simulation der Vegetation sind ein Anfang um die Bauklötze zum Leben zu erwecken. Die Techniken sind austauschbar und können vom Entwickler je nach Bedarf beliebig abgeändert werden. Die Idee der vollständig simulierten Umgebungen in Computerspielen ist eine die noch in ihren Kinderschuhen steckt. Doch die rapide Entwicklung der Computer Hardware und das wachsende Interesse an zellularen Automaten seitens der Spieleentwickler, lässt auf eine gute Zukunft hoffen.

Quellenverzeichnis

Literatur

[1] Eugene Ch'ng. „Ground Cover and Vegetation in Level Editors: Learning from Ecological Modelling“. In: *Proceedings of CGAMES'2009 USA*. Louisville, Kentucky, 2009.

[2] Tom Forsyth. „Cellular Automata for Physical Modelling“. In: *Game Programming Gems 3*. Charles River Media, 2002, S. 200–213.

[3] Alain Fournier, Don Fussell und Loren Carpenter. „Computer rendering of stochastic models“. In: *Communications of the ACM* 25.6 (1982), S. 371–384.

[4] Erich Gamma u. a. *Design Patterns*. 37. Aufl. Addison-Wesley Professional, 2009.

[5] Jos Stam. „Real-Time Fluid Dynamics for Games“. In: *Proceedings of the Game Developer Conference*. Bd. 18. San Jose, California, 2003, S. 17.

[6] Penelope Sweetser. „An Emergent Approach to Game Design-Development and Play“. Diss. School of Information Technology und Electrical Engineering, The University of Queensland, 2006.

[7] Jörg R. Weimar. *Simulation with Cellular Automata*. 2. Aufl. Logos-Verlag, 1997.