Prozedurale Generierung von endlosen Landschaften mit softwarebasierten Agenten

David Eibensteiner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2015

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 26. Juni 2015

David Eibensteiner

Inhaltsverzeichnis

Kurzfassung

In der herkömmlichen Spieleentwicklung ist das Erstellen von interessanten Landschaften ein sehr kosten- und zeitintensiver Prozess. In den letzten Jahren sind deshalb immer häufiger Spiele auf den Markt gekommen, die prozedurale Landschaftsgenerierung verwenden um ihre Spielwelt zu erzeugen. Ein Problem dieser Welten ist jedoch die Tatsache, dass sie zum Teil unkontrollierbare Ergebnisse liefern. Die vorliegende Arbeit stellt daher einen Ansatz vor, der diese Probleme lösen soll.

Zu Beginn wird ein Überblick über Methoden gegeben, die im Bereich der prozeduralen Landschaftsgenerierung bereits existieren. Diese werden anhand ihrer Fähigkeit endlose Landschaften zu generieren miteinander verglichen.

Basierend auf einer dieser Methoden, den sogenannten softwarebasierten Agenten – [Doran und Parberry J.: *Controlled procedural terrain generation using software agents.*. In: IEEE Transactions on Computational Intelligence and AI in Games 2 (2010), S. 111-119, 1997] – wurde ein Ansatz entwickelt der es ermöglicht, endloses Terrain kontrolliert zu erzeugen. Dieser wird im Rahmen dieser Arbeit vorgestellt und evaluiert.

Abstract

The process of developing complex game worlds can be a very challenging one, especially for smaller studios. For this particular reason a lot of recent games use procedural terrain generation methods (PTG). One problem that these methods have, is that their results are quite unpredictable and can lead to unwanted results when used to generate endless terrain. This thesis tries to solve that problem.

First the reader will get an introduction to the topic of procedural terrain generation and existing methods in this field. These methods will be compared to each other, to provide an overview over methods capable of creating endless game worlds.

Based on one of these methods, the software based agents – [Doran und Parberry J.: *Controlled procedural terrain generation using software agents.*. In: IEEE Transactions on Computational Intelligence and AI in Games 2 (2010), S. 111-119, 1997] – a solution is presented, that makes it possible to generate endless terrain in a controlled fashion.

Kapitel 1

Einleitung

1.1 Motivation

Mit immer besser werdender Hardware steigt auch stetig der Anspruch an neue Spiele und deren Umfang. Spieler erwarten sich immer größere und bessere Spiele. Ein zentraler Bestandteil vieler Spiele ist dabei die Spielwelt. Diese besteht häufig aus großen Landschaften, die möglichst viele unterschiedliche Gegenden zum Erkunden bieten sollen. Es ist zeitaufwändig und anspruchsvoll solche Spielwelten zu designen.

Die prozedurale Landschaftsgenerierung bietet dazu einen guten Lösungsansatz. Sie ermöglicht es, endlose Landschaften mit geringem Aufwand zu erstellen. Ein Nachteil der derzeit gängigen Ansätze ist, dass Benutzer die Ergebnisse nur minimal beeinflussen können. Dies zeigt sich vor allem beim Erzeugen endloser Landschaften.

Die Motivation für diese Arbeit war es zu ermöglichen, endloses Terrain zu erzeugen und gleichzeitig die Kontrolle über den Prozess der Generierung von diesem zu behalten.

1.2 Problemstellung und Zielsetzung

Der folgende Abschnitt dient dazu Probleme, aus den Bereichen der Spieleentwicklung und Landschaftsgenerierung, aufzuzeigen die im Rahmen dieser Arbeit gelöst werden.

1.2.1 Indie-Gameentwicklung

Nachfolgend wird erklärt, welche Probleme sich bei der Umsetzung von Spielen mit geringem Budget und kleinem Entwicklerteam ergeben.

1.2.1.1 Kosten und Ressourcen

Wie auch in anderen Arbeiten über Prozedurale Generierung beschrieben, ist das Erstellen von Spielinhalten teilweise sehr kosten- und ressourcenaufwändig [2]. Kleine Studios haben oft kein großes Budget zur Verfügung um detaillierte Welten per Hand zu erzeugen, die zusätzlich eine Variation an spielbaren Inhalten bieten. Das Modellieren einer Landschaft per Hand ist darüber hinaus eine sehr zeitaufwändige Arbeit, was sich vor allem in Kosten für Designer niederschlägt. Ein weiteres Problem ist, dass sich die Inhalte eines Spiels während der Entwicklung häufig ändern. Generierte Inhalte bieten hier einen entscheidenden Vorteil, da sie robuster gegenüber Änderungen sind. Bei ihnen ist lediglich eine Anpassung des Algorithmus nötig.

1.2.1.2 Langzeitmotivation

Kleinere Studios haben nicht die Möglichkeit, dem Spieler eine Vielfalt unterschiedlichster Inhalte zu bieten. Über kurz oder lang führt das Wiederverwenden derselben Landschaften oder Modelle dazu, dass sich der Spieler langweilt.

1.2.2 Generierung von endlosen Landschaften in Echtzeit

Im nachfolgenden Abschnitt wird erklärt, welche Probleme man lösen muss, um die Generierung von endlosem Terrain in Echtzeit zu ermöglichen.

1.2.2.1 Erweiterung von Landschaftsteilen zur Laufzeit

Da es nicht möglich ist, eine endlose Landschaft auf einmal zu berechnen, ergibt sich die Notwendigkeit, dass Teile der Landschaft nach Bedarf berechnet werden müssen.

Ein Problem, das sich vor allem dabei ergibt, ist der Einfluss von noch nicht existierenden Landschaftsteilen auf bereits generierte Teile. Unter anderem kann ein Berg zum Beispiel genau am Rand eines Segmentes auftreten, sodass es nötig ist, ein bereits erzeugtes Segment nochmals zu verändern.

1.2.2.2 Kontrolle über die Generierung

Da prozedurale Generierung automatisch erfolgt, geht damit auch einher, dass Designer nur schwer kontrollieren können, was generiert wird [5]. Oft ist es nur möglich, Anpassungen anhand eines Seeds[1] oder weniger Parameter vorzunehmen.

[1] Wert auf den der Zufallszahlengenerator basiert, der für die Erzeugung des Terrains zuständig ist

1.3 Gendering

In der vorliegenden Arbeit wird auf eine geschlechtsgerechte Schreibweise verzichtet, um einen schlüssigen Aufbau sowie angenehmen Lesefluss bzw. eine angenehme Lesbarkeit und Verständlichkeit zu verwirklichen. Es wird darauf hingewiesen, dass das generische Maskulinum verwendet wird, obwohl gleichzeitig die weibliche als auch die männliche Person angesprochen ist.

Kapitel 2

Grundlagen und Überblick

Das folgende Kapitel dient dazu, den Leser in das Thema der prozeduralen Generierung einzuführen. Es beschreibt, welche Inhalte mit solchen Methoden erzeugt werden können. Der Fokus liegt dabei auf der prozeduralen Landschaftsgenerierung.

2.1 Geschichte

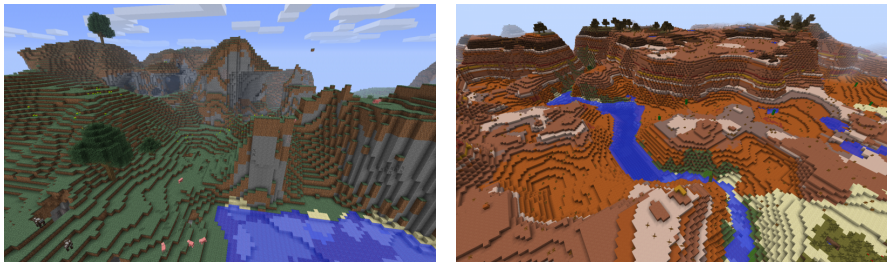
Da die ersten Videospiele noch mit begrenzter Speicherkapazität zu kämpfen hatten, war es nötig, Methoden zu entwickeln die es ermöglichten, große Welten zu erzeugen, die wenig Festplattenspeicher benötigten. Die frühesten Ansätze in den 80er Jahren verwendeten dazu Pseudo-Zufallszahlen-Generatoren [5, S. 3]. Das Weltraumspiel *Elite* war eines der ersten, das diesen Ansatz verwendete um ein großes Universum zu erzeugen.

2004 kam das Spiel *.kkrieger* auf den Markt, das Texturen, Modelle und auch Sounds generierte (Abb. 2.1). Das gesamte Spiel war unter 100KB groß, was deutlich weniger war als bei vergleichbaren Spielen [5, S. 3].

Im Jahr 2008 stellte *Valve Software* das Spiel *Left 4 Dead* vor, das erstmals prozedurale Generierung als eine zentrale Spielmechanik [27] verwendete. Es benützte eine künstliche Intelligenz mit dem Namen *Director*. Diese adaptierte Gegner und Items, basierend auf verschiedenen Variablen wie Leben, Munition oder Anzahl der Spieler.

Der schwedische Entwickler *Mojang* brachte 2011 mit *Minecraft* eines der mittlerweile meistverkauften Videospiele auf den Markt. Zu dessen Erfolg trug der Einsatz von prozeduralen Techniken einen großen Teil bei [27]. *Minecraft* verwendete einen auf Perlin Noise basierenden Algorithmus, um dessen endlose Welten zu generieren. Beispiele für auf diese Art generierte Landschaften sind in Abb. 2.2 zu finden.

Abbildung 2.1: Spiel *.kkrieger* von 2004 [28].



(a) (b)

Abbildung 2.2: In Minecraft generierte Landschaften: (a) Hügeliges Biom, (b) Mesa Biom [23].

2.2 Prozedurale Generierung

Prozeduraler Generierung ist das Erzeugen von Inhalten mittels meist deterministischen Algorithmen[1]. Diese Algorithmen arbeiten häufig mit einem sogenannten Seed, welcher es ermöglicht, reproduzierbare Ergebnisse zu erzeugen. Häufige Anwendung finden sie in Simulationen, Computerspielen und auch Filmen. Einige konkrete Beispiele für Anwendungsgebiete der prozedurale Generierung sind nachfolgend aufgezählt [2, 4, 7]:

- Generierung von Landschaften mit Bergen, Flüssen und Seen,
- Simulierte Städte bestehend aus Häuserblöcken und Straßen,
- Texturen für Modelle oder Landschaften,
- Objekte wie zum Beispiel Bäume, Waffen oder gar Modelle für be-

[1] Ein deterministischer Algorithmus ist ein Algorithmus, bei dem nur definierte und reproduzierbare Zustände auftreten.

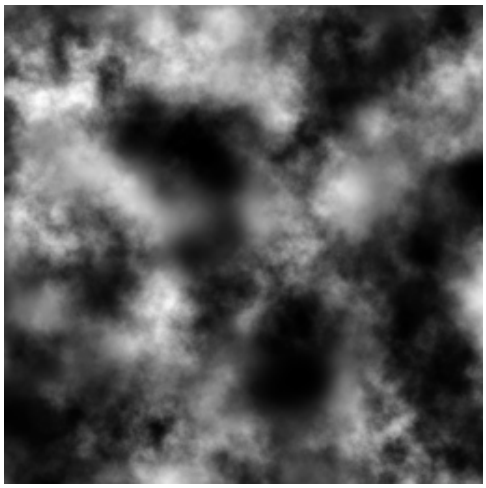


Abbildung 2.3: Mittels dem Programm Terragen erzeugte Heightmap [26].

stimmte Spielinhalte. Ein sehr bekanntes Beispiel dafür ist Speed Tree.[2]

2.2.1 Seed

Der Seed dient bei der prozeduralen Generierung als Grundlage für die Zufallsgenerierung. Ein Seed ist im Normalfall eine einfache Zahl. Bei der erneuten Eingabe eines Seeds soll ein prozeduraler Algorithmus jedesmal dasselbe Ergebnis zurückliefern. Wenn sich zwei Seeds auch nur um eine Zahl unterscheiden, liefern diese komplett verschiedene Ergebnisse.

2.3 Generierung und Darstellung von Landschaften

Die Generierung von Terrain zählt mitunter zu den interessanteren und komplizierteren Themen der Spieleprogrammierung. Allgemein ist darunter das Erzeugen von Landschaften anhand von Algorithmen zu verstehen, diese erstellen häufig eine Heightmap als Resultat.

2.3.1 Heightmap

Eine sogenannte Heightmap (im Deutschen “Höhenfeld”) dient als Grundlage für die Darstellung von Landschaften. Unter ihr kann man sich ein zweidimensionales Grid von Höhenwerten [13, S. 3] vorstellen. Ein Beispiel für eine solche ist in Abb. 2.3 zu sehen.

[2] http://www.speedtree.com/

0.35	0.35	0.35	0.35	0.35
0.35	0.7	0.7	0.7	0.35
0.35	0.7	1.0	0.7	0.35
0.35	0.7	0.7	0.7	0.35
0.35	0.35	0.35	0.35	0.35

Tabelle 2.1: Ausschnitt aus einer Heightmap für einen gleichmäßig abfallenden Berg.

Durch den simplen Aufbau der Heightmaps können Operationen wie das Glätten von Terrain oder ein simpler Erosion-Algorithmus[3] einfach darauf angewendet werden [8, S. 5].

Ein Nachteil der Heightmaps ist, dass es mit ihnen nicht möglich ist, Überhänge und Höhlen abzubilden [13, S. 3]. Wenn dies jedoch das Ziel ist, dann muss ein komplexerer Ansatz, wie der des *Arches* Framework verwendet werden [9]. Dieser benützt eine komplexere Struktur zum Abbilden der Landschaft, die sich aus mehreren Materialschichten zusammensetzt.

2.3.2 Generierung

Die Generierung eines kompletten Terrains besteht aus mehreren Schritten:

1. Formen von Küstenlinien,
2. Erzeugen von Bergen und Hügel,
3. Erzeugen von Flüssen und Seen,
4. Platzierung von Vegetation in der Welt wie z.B. Bäume oder Gräser.

In welcher Reihenfolge diese Schritte ausgeführt werden, hängt vom verwendeten Ansatz ab, sie können auch gleichzeitig passieren [2, 3, 6].

2.3.2.1 Berge und Hügel

Ein zentraler Schritt beim Generieren von Landschaften ist das Platzieren von Hügeln und Bergen. Diese werden durch das Erhöhen von Werten in der Heightmap erzeugt. Da sich ein Berg jedoch nicht nur mit einem einzigen Höhenwert abbilden lässt, ist es nötig, rund um seine Position passend abfallende Höhenwerte einzutragen. Diese Abfallrate ergibt die Möglichkeit verschiedenste Berge zu erzeugen. Ein Beispiel wie die Werteverteilung für einen gleichmäßig abfallenden Berg in einer Heightmap aussehen könnte ist in Tab. 2.1 zu sehen.

[3] Erosion ist der Substanzverlust der Erdoberfläche durch Einwirkung von Umwelteinflüssen wie Wasser.

2.3.2.2 Flüsse und Seen

Die Platzierung von Flüssen und Seen ist ein weiterer Schritt um glaubwürdige Landschaften zu erzeugen. Wenn das Ziel angestrebt wird, realistische Flussverläufe zu generieren, kann das unter Umständen ein sehr komplexer und aufwändiger Prozess sein [2, S. 5]. Ein Beispiel für einen möglichen Ansatz zur Erzeugung von Flussverläufen wird von Son Tee Teoh in [14] beschrieben. Die Idee dahinter ist, dass auf erhöhten Punkten einer Landschaft Wasserquellen platziert werden. Diese fließen unter Einfluss einer simplen Physiksimulation die Berge hinunter, wodurch sich Flüsse bilden.

2.3.2.3 Vegetation

Ein anderer Aspekt beim Erstellen von Landschaften ist die Vegetation. Objekte wie z.B. Bäume können nicht direkt in der Heightmap gespeichert werden und müssen deshalb in einem separaten Schritt direkt auf einer dargestellten Landschaft platziert werden. Die Modelle für die Vegetation können entweder vorgefertigt oder auch generiert sein.

2.3.3 Darstellung

Die Darstellung des Terrains erfolgt basierend auf der erzeugten Heightmap, ein Beispiel für das Mappen von Höhenwerten auf eine Landschaft ist in Abb. 2.4 zu sehen. Die so erzeugte Landschaft kann mit unterschiedlichster Genauigkeit dargestellt werden. Dieser Effekt kann vor allem in Spielen genutzt werden, weit entfernte Landschaftsteile können mit weniger Details gerendert werden, was einen Performancegewinn bringt. Sobald sich ein Spieler einem Gebiet nähert, lässt sich dieses nachträglich mit einem höheren Detailgrad darstellen.

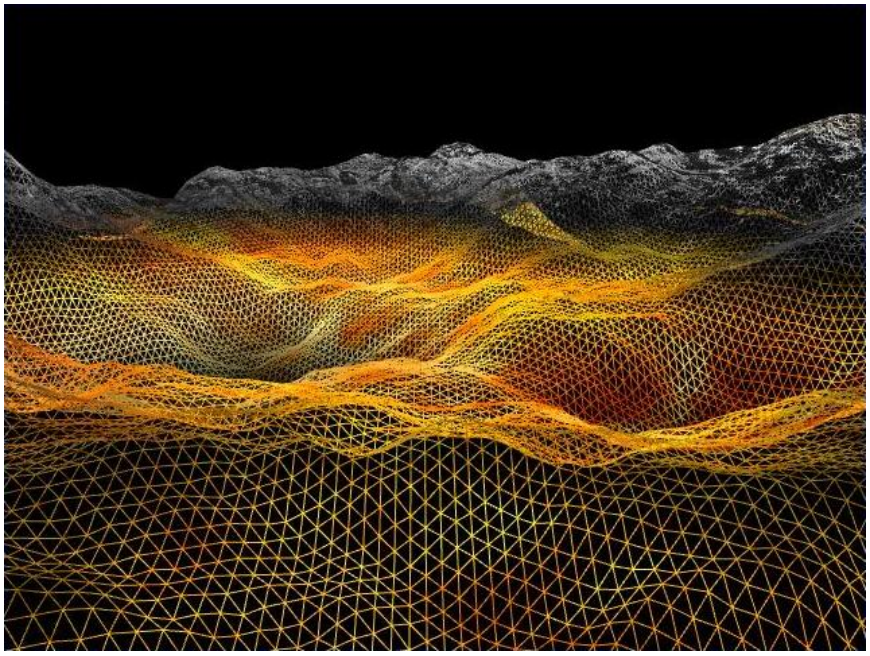


Abbildung 2.4: Wireframe-Landschaft in der jeder Eckpunkt eines Dreiecks auf einem Wert der Heightmap liegt [24].

Kapitel 3

Methoden zur Generierung von Landschaften

Das folgende Kapitel dient dazu, einen Überblick über existierende Verfahren zum Generieren von Landschaften zu geben. Die untersuchten Ansätze werden anhand von mehreren Kriterien aus einem Bewertungskatalog miteinander verglichen. Ziel dieses Kapitels ist es, einen Überblick zu geben, welche Verfahren sich zur Generierung von endlosem Terrain in Echtzeit eigenen. Im Besonderen dient es auch dazu aufzuzeigen, weshalb der Ansatz der softwarebasierten Agenten ausgewählt wurde und wie dieser im Detail funktioniert.

3.1 Bewertungskriterien

Die zu untersuchenden Ansätze in dieser Arbeit sollen, basierend auf ihrer Fähigkeit zur Generierung von endlosen Landschaften, miteinander verglichen werden. Dies passiert mithilfe der nachfolgenden Kriterien:

1. *Kontrollierbarkeit*: darunter wird verstanden in welchem Ausmaß ein User Kontrolle über den Generierungsprozess und über die erzeugten Ergebnisse hat.
2. *Echtzeitfähigkeit*: bedeutet ob ein verwendeter Ansatz es ermöglicht, Landschaften im Hintergrund zu erzeugen ohne den User einzuschränken wenn er diese durchquert.
3. *Endlosigkeit*: sagt aus, ob es mit einem Algorithmus möglich ist, nach und nach weitere Teile der Landschaft zu erzeugen um so endlos lange neues Terrain zu erhalten.
4. *Nötige Benutzereingaben*: darunter ist der Aufwand zu verstehen, der für den User anfällt um eine Landschaft zu erzeugen, sowohl vor der Generierung als auch währenddessen.
5. *Qualität*, des erzeugten Terrains: diese Kategorie ist eine teils subjek-

tive Einschätzung des Autors, wie gut ein Algorithmus sich eignet um glaubhafte Landschaften zu erzeugen.

3.2 Ansätze

Im Rahmen dieser Arbeit wurden verschiedene Ansätze zur Generierung von Landschaften untersucht. Nachfolgend sind diese aufgezählt:

- *Zufallszahlen-basierte Methoden*: Pseudozufällige mathematischen Reihen werden zum Erzeugen der Landschaften verwendet.
- *Sketchbasierte Methoden*: Terrain wird anhand eines von einem User gezeichneten Sketches und eines Terrainmodels generiert.
- *Evolutionäre Algorithmen*: Landschaften werden mithilfe eines Genotypen und einer Fitnessfunktion erzeugt.
- *Softwarebasierte Agenten*: verwenden Agenten um jeweils Teile des Terrains zu erzeugen.

3.3 Zufallszahlen-basierte Methoden

Es gibt verschiedenste Methoden, um Terrain basierend auf zufälligen Reihen zu erzeugen. Die gängigsten dieser Methoden sind [1]:

- Midpoint Displacement,
- Diamond-Square,
- Value Noise,
- Perlin Noise,
- Simplex Noise,
- Cell/Whorley Noise.

Nachfolgend werden die nützlicheren dieser Algorithmen erklärt.

3.3.1 Midpoint Displacement

Der Midpoint-Displacement-Algorithm ist ein sehr einfaches Verfahren, das als Basis für das Diamond-Square Verfahren dient [1]. Die grundlegenden Schritte des Algorithmus sind folgende:

- Zuerst werden zwei Ausgangspunkte p_s, p_e ausgewählt, die eine Linie l bilden.
- Der Mittelpunkt p_m dieser Linie l wird berechnet und ihm wird der Durchschnittswert der Werte der zwei Punkte p_s, p_e zugewiesen plus einem zufälligen Fehler f. Der Wert des Fehlers f sollte maximal so groß wie die Länge der Linie sein, um gute Ergebnisse zu erzeugen.
- Die zuvor durchgeführten Schritte werden nochmals, für die Linien zwischen den Punkten p_s und p_m sowie p_m und p_e ausgeführt. Dieser

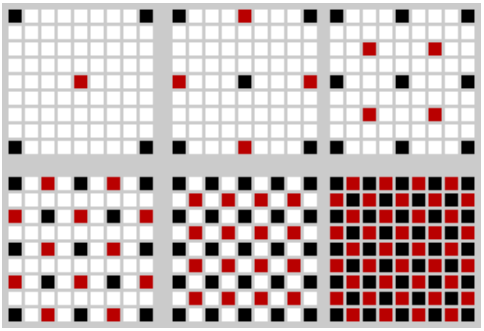


Abbildung 3.1: Schrittweises Vorgehen des Midpoint-Displacement-Verfahrens. Die roten Punkte stellen immer den im aktuellen Schritt neu berechneten Wert dar [18].

Vorgang des Zerteilens wird solange wiederholt, bis ein zufriedenstellendes Ergebnis erreicht wurde.

Will man den Algorithmus nun benützen um Heightmaps für 3D-Landschaften zu erzeugen, ist es nötig, statt einer Linie ein Quadrat zu verwenden. Die schrittweise zu berechnenden Mittelpunkte des Quadrats sind in Abb. 3.1 dargestellt. Hier werden im Gegensatz zur Linienversion abwechselnd folgende Punkte berechnet:

- Mittelpunkt p_m ergibt sich aus dem Durchschnittswert der Höhenwerte der vier umliegenden Eckpunkte,
- Mittelpunkte der seitlichen Punkte werden, basierend auf den Werten der Eckpunkte, zwischen denen sie liegen berechnet.

3.3.2 Diamond-Square

Der Diamond-Square-Algorithmus ist eine verbesserte Variante des Midpoint Displacement Algorithmus. Hier wird zusätzlich verhindert, dass Artefakte entstehen. Der Unterschied beider Algorithmen besteht hauptsächlich in der Berechnung der seitlichen Mittelpunkte. Im Gegensatz zu Midpoint Displacement werden für die Berechnung der seitlichen Mittelpunkte jeweils vier umliegende Punkte herangezogen, wie in Abb. 3.2 dargestellt.

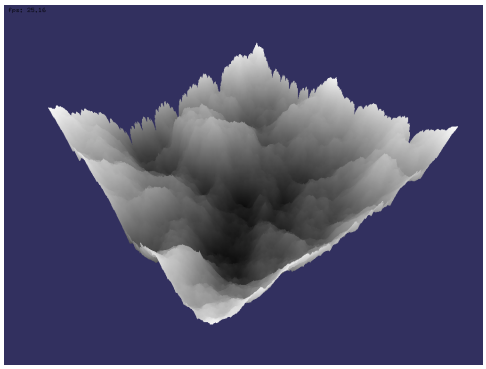
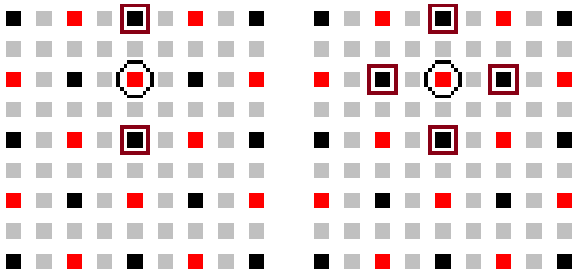


Abbildung 3.3: Mittels Diamond-Square erstellte Landschaft [20].



(a) (b)

Abbildung 3.2: Unterschied bei der Berechnung der Seitenpunkte zwischen Diamond-Square und Midpoint Displacement. (a) Nur zwei Punkte werden verwendet bei Midpoint Displacement, (b) Diamond Square benützt hingegen vier umliegende Punkte [25].

Eine Besonderheit die bei Diamond-Square beachtet werden muss, ist die Möglichkeit, dass der aktuell zu berechnende Punkt am Rand des Quadrats liegen kann, wo nur drei umliegende Punkte zur Verfügung stehen. Um dieses Problem zu beheben gibt es folgende Ansätze:

- Der Mittelpunkt wird ohne einen vierten Eckpunkt berechnet.
- Es wird für den vierten Punkt ein zufälliger Wert verwendet, der ähnlich zum Durchschnittswert der drei anderen Werte ist.
- Es wird ein Standardwert verwendet, wie für die vier initialen Eckpunkte.
- Es wird der nächstgelegene Punkt am gegenüberliegenden Ende des Quadrats verwendet.

Ein Beispiel für eine Landschaft, die mittels Diamond-Square erzeugt wurde, ist in Abb. 3.3 dargestellt.

3.3.3 Value Noise

Value Noise ist ein auf Zufallszahlen basierender Algorithmus, der initial keine fraktalen Ergebnisse erzeugt. Das bedeutet dass die Selbst-Ähnlichkeit nicht gegebenen ist. Unter der Selbstähnlichkeit versteht man die Ähnlichkeit kleiner zu großen Features, sprich es kann beliebig rein gezoomt werden, und man sieht ein immer detaillierteres Ergebnis. Man bezeichnet diese Eigenschaft als Fraktal.

Um Value Noise selbstähnlich zu machen wird eine sogenannte "Fractional Brownian Motion" Funktion benützt. Die Idee hinter dieser Funktion ist, dass man verschiedenste Oktaven von Rauschen zusammenzählt, mit immer höher werdender Frequenz und immer niedriger Amplitude (siehe Abb. 3.4).

Die konkrete Umsetzung von Value Noise ist relativ einfach[1]:

- Zuerst wird ein Grid g mit zufälligen Werten erzeugt.
- Danach werden die konkreten Höhenwerte berechnet, indem man die Koordinaten eines Punktes über das Grid g legt und dessen vier umliegende Gridpunkte g_p herausfindet.
- Die Berechnung des konkreten Höhenwertes erfolgt letztendlich durch die Interpolation der Werte der vier umliegenden Gridpunkte g_p.

Das erzeugte Ergebnis von Value Noise ist sehr stark davon abhängig welche Interpolationsfunktion verwendet wird. Die gängigen Interpolationsvarianten sind die lineare-, die kosinus- oder die kubische Interpolation. Die lineare Variante ist die schnellste, erzeugt jedoch die schlechtesten Ergebnisse. Die Kosinusvariante ist etwas langsamer als die lineare, erzeugt jedoch rundere Resultate. Die Kubische ist die langsamste der drei, die Ergebnisse sind aber am besten [1].

Im Fall von Value Noise ist es für den User auch möglich, zu einem gewissen Grad zu kontrollieren, was für Ergebnisse erzeugt werden. Folgende Parameter können frei konfiguriert werden:

- Durch die Anzahl der verwendeten *Oktaven* kann der User einstellen, wie detailliert das zu erzeugende Ergebnis werden soll (siehe Abb. 3.4).
- Die *Frequenz* dient dazu anzugeben, wie nahe Features beieinander liegen. Wird ein hochaufgelösteres Ergebnis benötigt, dann sollte eine höhere Frequenz verwendet werden.
- Mit der *Amplitude* wird die Höhe der zu erzeugenden Features bestimmt. Die Amplitude fällt für jede Oktave etwas ab um zu verhindern, dass die Details die groben Features zu sehr verändern. Die Rate, in der die Amplitude abfällt, kann durch den sogenannten *Persistence* oder *Härtewert* definiert werden.

[1] Für eine Implementierung des Algorithmus siehe http://www.avanderw.co.za/value-noise-terrain-generation/

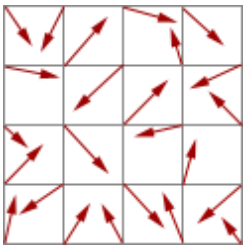
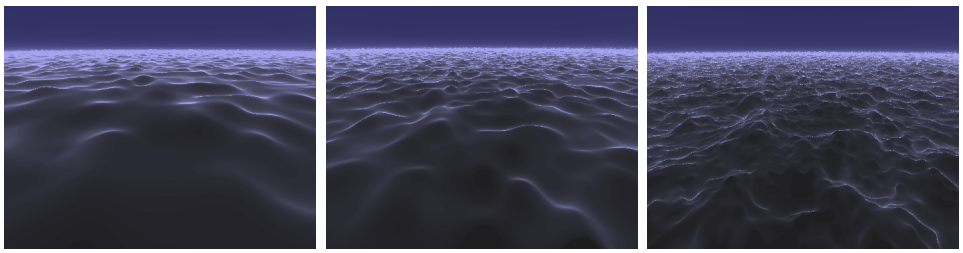


Abbildung 3.5: Visualisierung des Steigungsvektor-Grids, das bei Perlin Noise verwendet wird [19].



(a) (b) (c)

Abbildung 3.4: Veränderung des Ergebnisses von Value Noise bei einer höheren Anzahl von Oktaven: (a) eine, (b) zwei, (c) vier [22].

3.3.4 Perlin Noise

Perlin Noise ist der am häufigsten verwendete Noise-Algorithmus zum Generieren von Terrain, der Grund dafür ist, dass dieser schnell ist und auch gute Ergebnisse liefert [1]. Im Vergleich zu anderen Algorithmen derselben Klasse ist er jedoch nicht ganz so einfach zu Verstehen.

Perlin Noise fängt, wie auch schon Value Noise, mit einem Grid an, dieses besteht jedoch aus Steigungsvektoren, die alle in eine zufällige Richtung zeigen. Jedem Punkt im Grid ist genau ein solcher Vektor zugeordnet, wie in Abb. 3.5 dargestellt.

Wie auch schon die anderen Noise Algorithmen kann Perlin Noise dazu verwendet werden, eine Heightmap zu erzeugen. Dazu wird ein 2D Steigungsvektor-Grid g verwendet und für jeden Punkt in der Heightmap werden folgende Schritte ausgeführt:

- Zu Beginn werden vier Steigungsvektoren $v_1, \ldots, v_4$ aus dem Grid g ausgewählt, die rund um dem aktuellen Punkt p liegen.
- Für jeden dieser Steigungsvektoren $v_1, \ldots, v_4$, wird das Skalarprodukt[2] s_n zwischen dem Steigungsvektor v_s und einem Distanzvektor v_d be-

[2] http://de.wikipedia.org/wiki/Skalarprodukt

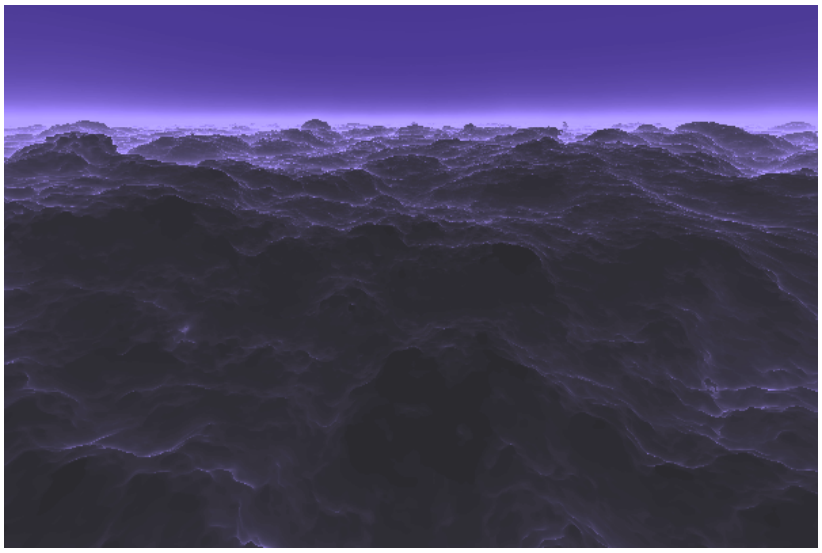


Abbildung 3.6: Mittels Perlin Noise generierte Landschaft [22].

rechnet. Der Distanzvektor v_d ergibt sich aus dem aktuellen Punkt p und dem nächstgelegenen Gridpunkt p_g.

- Zuletzt wird lineare Interpolation[3] benützt um aus den Ergebnissen der Skalarprodukte $s_1, \ldots, s_4$ Höhenwerte zu berechnen. Zusätzlich kann eine Fade-Funktion verwendet werden um die Einwirkung der umliegenden Punkte zu verringern.

Für eine detailierte mathematische Erklärung des Perlin Noise Algorithmus siehe [1, S. 5, Kapitel 2.4]. Ein Beispiel für Terrain, das mit Perlin Noise generiert wurde zeigt Abb. 3.6.

3.3.5 Simplex Noise

Simplex Noise ist eine verbesserte Version des Perlin Noise Algorithmus von Ken Perlin. Simplex Noise bietet den Vorteil, dass es in höheren Dimensionen (4D, 5D, ...) um einiges schneller ist als Perlin Noise [1, S. 6]. Die Laufzeit von Perlin Noise ist exponentiell in Bezug auf die Dimension n, $O(n \cdot 2^n)$. Simplex Noise hingegen ist polynomial $O(n^2)$, wodurch es vor allem in einer höheren Dimension bedeutend schneller ist.

Der Grund, warum Simplex Noise schneller ist, hat damit zu tun, dass Perlin Noise in 2D ein Quadrat und in 3D einen Würfel als Grundlage für die Berechnung benützt. Simplex Noise verwendet hingegen ein Dreieck in 2D und eine Pyramide in 3D, wodurch für die Berechnung eines Wertes weniger Rechenschritte nötig sind. Ein weiterer Vorteil von Simplex Noise ist, dass

[3] http://de.wikipedia.org/wiki/Interpolation

es im Vergleich zu Perlin Noise keine lineare Interpolation benötigt. Eine ausführliche Erklärung dieses Verfahrens und den Gründen, warum Simplex Noise ohne Qualitätseinbußen funktioniert, ist in [1, S. 6] zu finden.

3.3.6 Bewertung

Im folgenden Abschnitt werden die auf Zufallszahlen basierten Methoden anhand der in Abschn. 3.1 definierten Kriterien bewertet:

1. *Kontrollierbarkeit*: Es kann initial Einfluss auf die Verfahren genommen werden durch die Verwendung verschiedener Seeds. Durch die Einschränkungen des zu erzeugenden Wertebereichs kann noch weiterer Einfluss ausgeübt werden, jedoch lässt sich das Ergebnis dadurch nur geringfügig kontrollieren.
2. *Echtzeitfähigkeit*: Zwischen den einzelnen Verfahren bestehen teilweise größere Unterschiede in Bezug auf die Geschwindigkeit der Algorithmen. Alle Varianten sind jedoch schnell genug, dass sie sich für die Erzeugung von Terrain in Echtzeit eignen.
3. *Endlosigkeit*: Ist bei allen Verfahren möglich, da man endlos lange neue Abschnitte berechnen kann.
4. *Nötige Benutzereingaben*: Bis auf den Seed sind keine zwingenden Eingaben nötig, vor allem nicht während der Generierung, was diese Verfahren weitgehend unabhängig von Benutzereingaben macht.
5. *Qualität*: Die betrachteten Methoden schaffen es alle realistisch wirkendes Terrain zu erzeugen, wobei Perlin Noise und Simplex Noise die besten Resultate liefern.

3.3.6.1 Algorithmen-Bewertung

In der nachfolgenden Tabelle 3.1 werden die existierenden Algorithmen miteinander verglichen, die Bewertungen basieren auf der Arbeit “Procedurally Generating Terrain” [1]:

- A Midpoint Displacement,
- B Diamond Square,
- C Value Noise,
- D Perlin Noise,
- E Simplex Noise,
- F Cell/Whorley Noise.

	A	**B**	**C**	**D**	**E**	**F**
Geschwindigkeit (Je größer desto schneller, relativ zueinander)	64	57	3.7	1.5	3.2	1
Benötigter Arbeitsspeicher (Je größer desto mehr Speicher, von 0–10)	2	2	10	8	8	4
Qualität der Ergebnisse (Je höher desto besser von 0–10)	4	8	7	9	10	4

Tabelle 3.1: Gegenüberstellung von Eigenschaften der auf Zufallszahlen basierenden Verfahren.

3.4 Sketchbasierte Methoden

Ein anderer Ansatz um Terrain zu erzeugen ist der von Huafei Yin und Changwen Zheng [17]. Ihr Ansatz basiert auf der Idee, Landschaften basierend auf einem gezeichneten Sketch zu generieren. Ein solcher beinhaltet verschiedenste Feature-Informationen. Diese Features werden durch die Verwendung bestimmter Farben definiert. Ein Beispiel für einen solchen Sketch zeigt Abb. 3.7.

Der konkrete Algorithmus von Huafei und Changwen besteht aus drei Schritten:

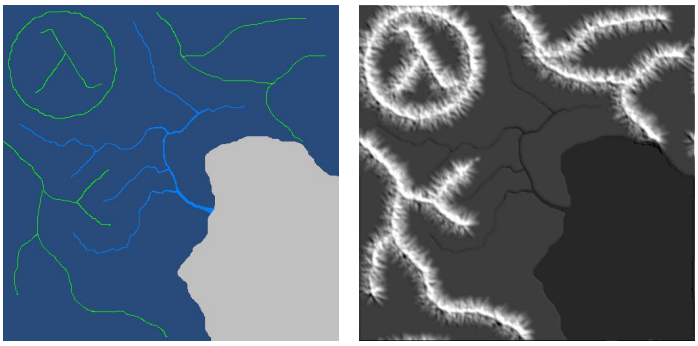
- Initialisierung,
- Erstellung der Terrain-Heightmap,
- Erweiterung der Features.

3.4.1 Initialisierung

Zuerst wird ein Sketch $S(m \cdot n)$ eingescannt, wobei dessen Zuordnung von Farben zu Features bereits berücksichtigt wird. Anschließend wird aus diesem eine Matrix $M(m \cdot n)$ erstellt, die dazu dient, einzelne Features zu kategorisieren. Ein beliebiger Punkt p in M stimmt immer genau mit demselben Punkt p_s in S überein und gibt Auskunft darüber, was für eine Art Feature sich an diesem Punkt befindet. In Abb. 3.7 wäre Blau zum Beispiel ein Fluss und Grün ein Berg.

3.4.2 Erstellung der Terrain-Heightmap

In diesem Schritt wird eine Heightmap H basierend auf der Matrix M erstellt. Für jeden Wert in M wird ein Höhenwert an der selben Stelle in H berechnet. Der Höhenwert wird durch die Art von Feature bestimmt die in M an dessen Position steht. Für Berge und Schluchten wird jeweils ein



(a) (b)

Abbildung 3.7: Sketch basierte Terrain Generierung, a) Sketch mit Feature Informationen, b) erzeugtes Ergebnis des Algorithmus [17, S. 778].

bestimmter Grundwert plus einem zufälligen Wert verwendet. Features wie Seen, Flüsse oder Ebenen verwenden ebenfalls nur einen Grundwert.

3.4.3 Erweiterung der Features

Manche Features müssen noch erweitert werden nach der Erstellung der Heightmap H, um glaubhaft wirkende Landschaften zu erzeugen. Diese sind zum Beispiel Gebirgskämme und Schluchten. Um diese Features zu verbessern werden folgenden Schritte für jeden Punkt p in H ausgeführt:

- *Berechnung der Aspekte* wie Abschrägungen und Steigungen zu benachbarten Punkten von p.
- *Suche nach geeigneten Heightmap-Punkten*, rund um p die zu Bergen oder Schluchten zusammengefasst werden können.
- *Erzeugung der Höhenwerte* für die gefundenen Punkte N und den Startpunkt p.

3.4.3.1 Berechnung der Aspekte

Zur Berechnung der Aspekte wird der Horn Algorithmus benützt. Für die Berechnung eines Aspekts werden die 8 umliegenden Nachbarshöhenwerte herangezogen. Direkte Nachbarn werden mit dem Wert 2 gewichtet und die diagonal liegenden Nachbarn mit 1. Zur Berechnung des Aspektes wird nun folgende Formel benützt:

h_1	h_2	h_3
h_4	C	h_5
h_6	h_7	h_8

Tabelle 3.2: Nachbarspunkte, die bei Berechnung des Aspektes herangezogen werden.

$$n_x = (h_1 + 2 \cdot h_4 + h_6) - (h_3 + 2 \cdot h_5 + h_8), \tag{3.1}$$

$$n_y = (h_6 + 2 \cdot h_7 + h_8) - (h_1 + 2 \cdot h_2 + h_3), \tag{3.2}$$

$$A = \arctan(\frac{n_y}{n_x}). \tag{3.3}$$

Die Werte $h_1, \ldots, h_8$ stellen die Höhenwerte der Nachbarn dar, A ist der berechnete Aspekt für einen ausgewählten Punkt.

3.4.3.2 Suche nach geeigneten Heightmap-Punkten

Der berechnete Aspekt A ist ein Winkel, der Auskunft darüber gibt, in welcher Richtung nach geeigneten Höhenwerten gesucht werden soll, um Kandidaten für das Zusammenfassen zu einem Berg oder einem Tal zu finden. Es wird ebenfalls in der genau gegenüberliegenden Richtung nach Kandidaten gesucht, alle Punkte, die kein Fluss oder See sind, werden in das Punkteset N aufgenommen. Die Suche nach geeigneten Punkten erfolgt bis zu einem bestimmten Punkt p_e. Dieser wird mittels des Aspektes A und einer vom Nutzer einstellbaren Distanz berechnet (siehe [17, S. 776] für Details).

3.4.3.3 Erzeugung der Höhenwerte

Zuletzt werden für alle Punkte im Punkteset N zugehörige Höhenwerte berechnet. Dazu wird ein eindimensionaler Midpoint Displacement (siehe Abschn. 3.3.1) Algorithmus verwendet. Dieser wird benützt um die Berghänge und Schluchten zu erzeugen (siehe [17, S. 776] für Details).

3.4.4 Bewertung

Nachfolgend werden die sketchbasierten Methoden anhand der Kriterien des Bewertungskataloges (siehe Abschn. 3.1) analysiert:

1. *Kontrollierbarkeit*: Der Nutzer kann sehr gut kontrollieren, wo welche Features erzeugt werden sollen, dies gibt ihm ein hohes Maß an Kontrolle über den Generierungsprozess. Lediglich über das konkrete Aussehen von Bergen hat der Benutzer nur indirekte Kontrolle.

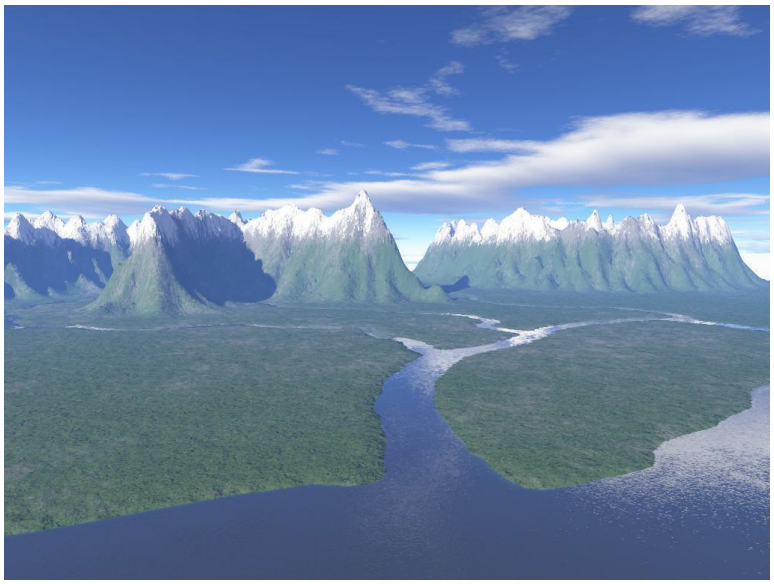


Abbildung 3.8: Resultat der sketchbasierten Methode, das Berge und Flüsse enthält [17, S. 778], basierend auf dem Sketch in Abb. 3.7.

2. *Echtzeitfähigkeit*: Wie in [17, S. 777] beschrieben, braucht das Verfahren 0.437 Sekunden um das in Abb. 3.8 dargestellte Terrain mit einer Größe von 512^2 Pixeln zu erzeugen. Da eine große Fläche in weniger als einer Sekunden erzeugt werden kann, wäre es möglich, dieses Verfahren in einer Echtzeitanwendung zu benützen.
3. *Endlosigkeit*: Da es in diesem Verfahren immer notwendig ist, einen Sketch für ein Stück Terrain zur Verfügung zu stellen, ist die Generierung einer endlosen Welt, die sich nicht wiederholt, nicht möglich.
4. *Nötige Benutzereingaben*: Es wird vom Benutzer erwartet, einen Sketch zur Verfügung zu stellen, anhand dessen eine Landschaft erzeugt wird, dies erfordert ein hohes Maß an Benutzereingaben.
5. *Qualität*: Die betrachtete Methode schafft es gut, Terrain basierend auf der Vorlage eines Sketches zu erzeugen, wie man in Abb. 3.8 sehen kann. Die Qualität der erzeugten Ergebnisse hängt bei dieser Methode hauptsächlich vom Benutzer ab.

3.5 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind ein weiterer Ansatz, der es ermöglicht, kontrolliert Terrain zu erzeugen. Die bekannten Algorithmen erzeugen alle eine Heightmap als Ausgabe. Evolutionäre Algorithmen basieren auf der Idee, das man einen Genotyp und eine Fitnessfunktion benützen kann, um ein

Terrain zu erzeugen. Der Genotyp stellt dabei immer den aktuellen Zustand des Systems dar, und die Fitnessfunktion gibt Auskunft darüber, wie gut eine Genotyp-Repräsentation ist. Evolutionäre Algorithmen folgen im Groben immer folgendem Schema [16]:

1. *Initialisierung*: Generierung von mehreren initialen Lösungen, meistens zufällig.
2. Nachfolgend werden folgende Schritte wiederholt, bis ein Abbruchkriterium erreicht wird.
   2.1. *Evaluation*: Jeder Lösung wird mithilfe der Fitnessfunktion ein Fitnesswert zugewiesen.
   2.2. *Selektion*: Lösungen mit den besten Fitnesswerten werden für die Reproduktion ausgewählt.
   2.3. *Rekombination und Mutation*: Neue Lösungen werden durch Mutation und Kreuzung erzeugt.
   2.4. Für die erzeugten Lösungen wird der Prozess erneut gestartet.

Es gibt ein paar konkrete Umsetzungen von evolutionären Algorithmen um Terrain zu generieren, diese sind nachfolgend aufgeführt[4] [10]:

1. Terrainosaurus,
2. Ashlock,
3. Walsh und Gade,
4. Frade et al.,
5. Togelius,
6. Raffe.

Die beiden Varianten “Terrainosaurus” und “Walsh und Gade” sind in den folgenden Abschnitten genauer erklärt.

3.5.1 Terrainosaurus

Der Terrainosaurus Algorithmus besteht aus drei Phasen wie in Abb. 3.9 dargestellt:

1. Die erste Phase ist ein Klassifikationsprozess vom Terrain. Der Benutzer muss dazu Beispiele für Terrain sowie eine Klassifizierung von diesem zur Verfügung stellen. Anschließend werden diese zu einer Datenbank zusammengefasst und dienen als Palette von Terraintypen.
2. In der zweiten Phase muss der Nutzer sein gewünschtes Terrain definieren, indem er jeweils, wie in Abb. 3.9 dargestellt, Bereiche mit einer ausgewählten Palette skizziert.

[4]Für die Algorithmen, die keinen eigenen Namen haben, wurde der Name des Autors verwendet

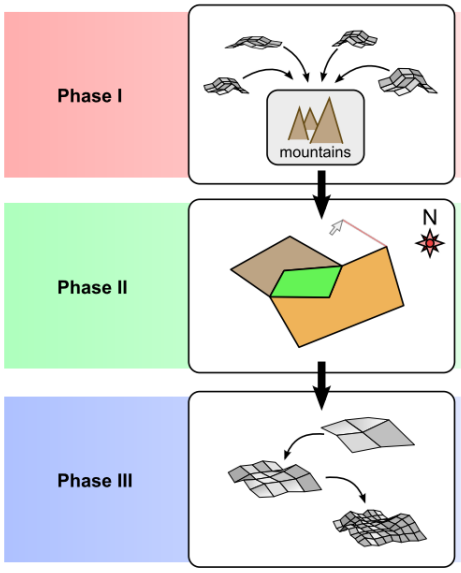


Abbildung 3.9: Drei Phasen des Terrainosaurus Algorithmus [12, S. 4].

3. Die letzte Phase beinhaltet die konkrete Erzeugung der Heightmap, in dieser sind vom Nutzer keine Eingaben mehr nötig. Hier kommt der evolutionäre Algorithmus zum Einsatz, wie nachfolgend erklärt.

3.5.1.1 Erzeugung der Heightmap

Um das Terrain zu erzeugen werden zuerst einige Kontrollpunkte auf der Oberfläche ausgewählt. Die Genotyp-Repräsentation wird als eine Liste von Genen definiert, die nacheinander auf ein Ausgangsterrain angewendet werden, um so ein bestimmtes Ergebnis zu erzeugen. Ein Gen ist dabei jeweils genau einem Kontrollpunkt zugewiesen und beinhaltet eine Liste von Operationen, die auf diesem Punkt ausgeführt werden sollen. Solche Operationen können das Erhöhen, Rotieren oder Abflachen von Terrain sein. Die Mutationen und Kreuzungen sind in diesem Fall ein zufälliges Ändern und Kombinieren dieser Operationen.

Die in Terrainosaurus verwendet Fitnessfunktion ist eine Funktion, die die Ähnlichkeit der generierten Terrainstücke mit den Terrainbeispielen, auf denen sie basieren, vergleicht.

Die genaue Formel der Fitnessfunktion ist

$$f = \alpha \sum_{R} f_R \cdot \frac{A_R}{A_T} + (1 - \alpha) \cdot \sum_{G} \frac{c_G}{N}. \tag{3.4}$$

f ist der berechnete Fitnesswert (zwischen 0–1), α ist ein Gewichtungsfaktor und A_R die Fläche von R. f_R ist die Fitness der Region R, diese ergibt sich

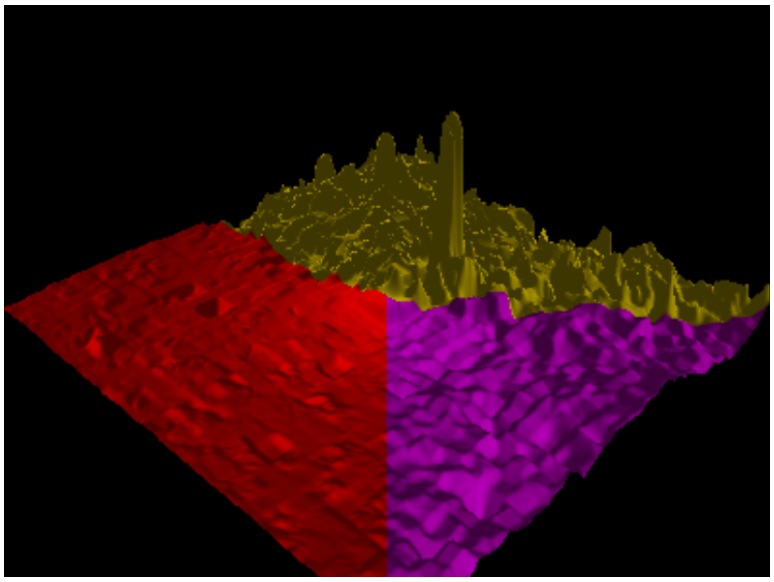


Abbildung 3.10: Beispiel des genetischen Algorithmus Terrainosaurus. Das erzeugte Terrain ist in drei Teile aufgeteilt, die jeweils unterschiedliche Paletten verwenden [12, S. 106].

aus einer statistischen Bewertung. A_T ist die totale Größe der Heightmap, c_G die Kompatibilität zwischen dem Gen G und seinem Umfeld und N die Anzahl der Gene. Für eine detaillierte Erklärung dieser Berechnung sowie der Variablen siehe [12, S. 49].

3.5.2 Walsh und Gade

Der Ansatz von Walsh und Gade [15] basiert auf der Idee evolutionäre Algorithmen auf die Eingangsparameter eines Terrain-Generators anzuwenden. Die verwendeten Parameter sind nachfolgend aufgeführt:

1. *Höhe*: sagt aus, wie hoch die Berge eines Terrains allgemein werden dürfen (siehe Abb. 3.11).
2. *Unebenheit*: bestimmt den Grad der Glattheit eines Terrains (siehe Abb. 3.11).
3. *Wasserhöhe*: definiert das grundlegende Wasserlevel.
4. *Wolkendichte*: gibt die Häufigkeit von Wolken an.
5. *Sonnenrichtung*: gibt an, von welcher Richtung aus die Szene beleuchtet werden soll.

Die verwendete Genotyp-Repräsentation ist eine Gruppe von Parametereinstellungen, die jeweils als 8-Bit Strings gespeichert werden. Die Mutation dieses Genotyps erfolgt durch das zufällige Ändern einzelner Bits. Die Kombination zweier Genotypen passiert durch das Zusammenfassen von jeweils

(a) (b)

Abbildung 3.11: (a) Erzeugtes Terrain mit niedrigen Höhen- und Unebenheitseinstellungen. (b) Terrain mit großen Höhen- und Unebenheitseinstellungen [15].

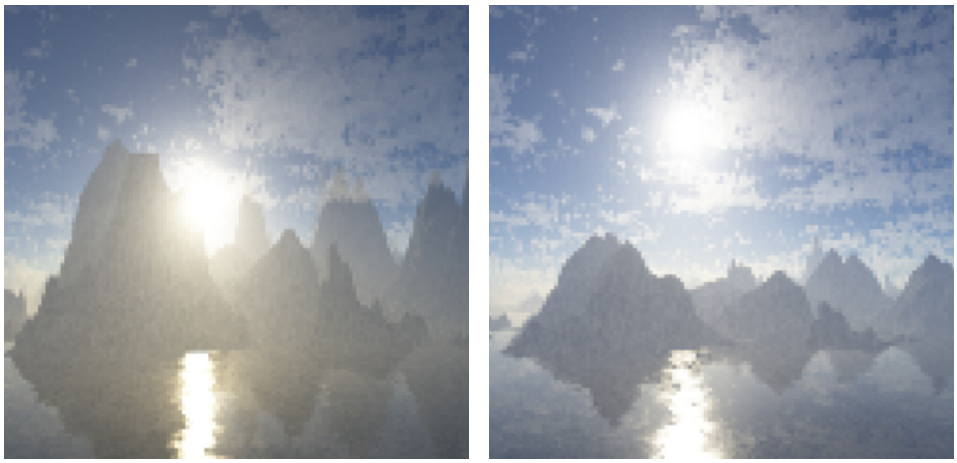
zwei 4-Bit-Teilen zu einem neuen 8-bit String.

Walsh und Gade benützen keine automatische Fitnessfunktion, sondern eine interaktive, das bedeutet, dass die Fitness eines Genotyps ausschließlich von Benutzer des Programms bestimmt wird. Dazu werden diesem Bilder vom erzeugten Terrain gezeigt, wie in Abb. 3.12 dargestellt. Basierend auf den Entscheidungen des Users wird die ausgewählte Genotyp-Repräsentation weiter verwendet. Das Ganze wird so lange durchgeführt, bis der User mit dem Ergebnis zufrieden ist und abbricht.

3.5.3 Bewertung

In diesem Abschnitt werden die evolutionären Algorithmen anhand der im Bewertungskatalog (s. Abschn. 3.1) definierten Kriterien bewertet:

1. *Kontrollierbarkeit*: Abhängig vom verwendeten Ansatz bieten evolutionäre Algorithmen sehr gute Kontrolle über den Generierungsprozess. Beim Ansatz von Walsh und Gade kann der User zum Beispiel sogar selbst bestimmen, welche Ergebnisse er für gut befindet.
2. *Echtzeitfähigkeit*: Da die evolutionären Ansätze jeweils mehrere Durchgänge benötigen um brauchbare Ergebnisse zu erzeugen, sind sie nicht für den Echtzeiteinsatz geeignet.
3. *Endlosigkeit*: Es werden jeweils mehrere Durchgänge gebraucht um ein Terrain zu erstellen, was diesen Ansatz für endloses Terrain ungeeignet macht.
4. *Nötige Benutzereingaben*: Bei Terrainosaurus sind zu Beginn einige Benutzereingaben nötig, bei Walsh und Gade sogar während des Ge-



(a) (b)

Abbildung 3.12: Zwei Beispiele (a), (b) für erzeugtes Terrain durch den Algorithmus von Walsh und Gade [15].

nerierungsprozesses.

5. *Qualität*: Das generierte Terrain ist bei beiden betrachteten Verfahren durchaus brauchbar, wie man in Abb. 3.10 und Abb. 3.12 sehen kann.

3.6 Softwarebasierte Agenten

Der Ansatz der softwarebasierten Agenten baut auf der Idee auf, Terrain durch den Einsatz von Agenten zu erzeugen. Wie in [11, S. 31] beschrieben ist ein solcher Agent definiert als ein Objekt, das seine Umgebung durch Sensoren wahrnimmt und durch Aktoren mit ihr interagiert.

Die Arbeit von Doran und Parberry [3] basiert auf der Idee dieser Agenten. Beginnend mit einem leeren Terrain werden verschiedenste Typen von Agenten verwendet, um dieses zu verändern und so nach und nach eine Landschaft zu erzeugen. In ihrem Ansatz benützen sie drei Klassen von Agenten, die Küstenlinien-, Landschafts- und Erosionsagenten. Der Prozess zum Erzeugen der Landschaft ist ebenfalls in drei Phasen aufgeteilt.

- Die *Küstenlinienphase* ist die erste Phase, und ist dafür zuständig, jene Küstenlinien zu erzeugen, die Ozeane vom Land abgrenzen.
- Die zweite Phase, die sogenannte *Landschaftsphase*, ist für das Formen der Landschaft zuständig. Sie erzeugt Berge, Strände und vor allem die grobe Struktur der Landschaft.
- In der dritten und letzten Phase, der *Erosionsphase* werden Flüsse durch das Erodieren von Landschaftsteilen erzeugt.

In sämtlichen Phasen arbeiten mehrere Agenten gleichzeitig, diese haben jeweils Zugriff auf das gesamte bereits erzeugte Terrain um Entscheidungen treffen zu können. Ein Agent kann zu jedem Zeitpunkt beliebige Punkte des Terrains editieren.

Zu Beginn einer Phase werden den Agenten sogenannte Token zugewiesen, durch diese wird bestimmt, wie viele Aktionen ein Agent durchführen darf. Hat ein Agent alle seine Token eingelöst, wird er vom Generierungsprozess entfernt. Die Anzahl der Tokens, das heißt die Lebensdauer eines Agenten, kann vom Benutzer des Algorithmus eingestellt werden. Er erhält damit zum Teil Kontrolle über den Generierungsprozess. Die softwarebasierten Agenten erzeugen als Ausgabe erneut Heightmaps.

3.6.1 Agententypen

Um ein vollständiges Terrain zu generieren sind verschiedenste Arten von Agenten nötig. Der Ansatz von Doran und Parberry [3] erwähnt fünf davon:

1. *Küstenlinienagenten*: erzeugen grobe Landmassen,
2. *Glättungsagenten*: glätten Unebenheiten,
3. *Strandagenten*: formen Strände,
4. *Bergagenten*: erheben Berge,
5. *Flussagenten*: erodieren Flüsse.

3.6.1.1 Küstenlinienagenten

Küstenlinienagenten arbeiten darauf hin, Landmassen zu erzeugen. Sie beginnen auf einer Landschaft, die komplett unter Wasser steht. Der Prozess des Generierens von Landmassen ist ein hierarchischer, ein Agent ist jeweils für eine große Landfläche zuständig. Nachdem er einen groben Landschaftsbereich erzeugt hat, erstellt er wiederum zwei neue Agenten und teilt diesen jeweils eine Hälfte der von ihm erzeugten Fläche zu. Diese Agenten führen denselben Prozess durch, bis ein Grenzwert für die Größe des zu bearbeitenden Gebiets erreicht ist, danach teilen sich Agenten nicht mehr auf. Auf diese Art und Weise werden global zusammenhängende Küstenlinien erstellt, die lokale Unterschiede aufweisen.

Jeder Agent startet am Rand seines zugewiesenen Bereiches, mit einer vorgebenden Richtung und einer Anzahl von Punkten, die er generieren soll. Um die konkrete Landmasse zu erzeugen werden folgende Schritte ausgeführt:

1. Suchen der bereits bestehenden Küstenlinie; falls keine gefunden wird, fährt der Agent mit dem zweiten Schritt fort.
2. Auswählen von zwei zufälligen Punkten, die sogenannten *Attractor* p_a und *Repulsor* p_r Punkte. Diese werden so gewählt, dass sie ausgehend

(a) (b)

Abbildung 3.13: Zwei generierte Küstenlinien [3, S. 113]. (a) Niedrige Untergrenze für das Aufteilen der Agenten, (b) hohe Untergrenze für das Aufteilen der Agenten.

von der aktuellen Position des Agenten in entgegengesetzter Richtung zueinander stehen.

3. Bewerten von Punkten in der Umgebung des Agenten; der Wert jedes Punktes p wird durch $d_r(p) - d_a(p) + 3d_e(p)$ berechnet. Dabei ist $d_r(p)$ das Quadrat der Distanz von p zum *Repulsor* p_r, $d_a(p)$ das Quadrat der Distanz von p zum *Attractor* p_a und $d_e(p)$ die kleinste Distanz von p zum Rand der Karte. Diese Berechnung verleitet den Agenten dazu, dass er Richtung *Attractor* p_a und weg vom Rand seines Bereiches bzw. weg vom *Repulsor* p_r wandert. Während er sich bewegt werden die Punkte, die am höchsten bewertet wurden, zur Landmasse hinzugefügt.
4. Der dritte Schritt wird solange wiederholt, bis ein Agent keine Token mehr hat.

Jeder Küstelinienagent hat ein paar einstellbare Parameter. Den Seed, den er für das zufällige Generieren der Landschaft benützen soll, und die Anzahl der verfügbaren Token. Zusätzlich kann der User noch einstellen, wie oft sich der Küstenagent aufteilen soll. In Abb. 3.13 sieht man zwei Beispiele für erzeugte Küstenlinien.

3.6.2 Glättungsagenten

Ziel der Glättungsagenten ist es, grobe Unebenheiten der Landschaft auszubessern. Jeder dieser Agenten arbeitet in einem lokalen Bereich, zu dem

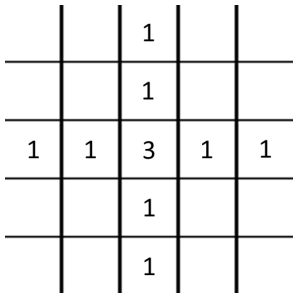


Abbildung 3.14: Auswahl des Extended Van Neumann Algorithmus, der in [3, S. 114] beschrieben wird. Die Zahlen in den Feldern stellen die Gewichtung dar.

er wiederholt zurückkehrt. Zum Glätten des Terrains wird ein *Extended van Neumann* Algorithmus verwendet, dieser sieht folgendermaßen aus:

1. Vom Ausgangspunkt p_c ausgehend werden alle orthogonal zu p_c stehenden Punkte p_s bis zu einer Entfernung 2 ausgewählt (siehe Abb. 3.14).
2. Im nachfolgenden Schritt werden alle orthogonal stehenden Punkte $p_1, \ldots, p_n$ mit 1 gewichtet und der Ausgangspunkt p_c mit 3, wie in Abb. 3.14 dargestellt. Die Gewichtung wird so gewählt, damit bestehende Steigungen nicht zu drastisch geglättet werden.
3. Zuletzt wird der neue Höhenwert des Punkts p_c durch

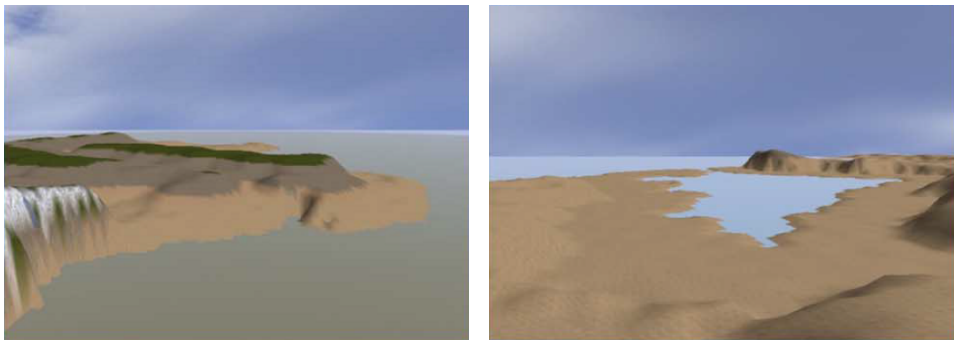
$$h_{cn} = \frac{3 \cdot h_c + \sum(h_1, \ldots, h_n)}{9} \tag{3.5}$$

   ermittelt, das Ergebnis ist der neue Höhenwert h_{cn}. Für dessen Berechnung werden der Höhenwert h_c des Punktes p_c verwendet und die Werte $h_1, \ldots, h_n$ der Punkte $p_1, \ldots, p_n$.

Wie die Küstenlinienagenten verwendet der Glättungsagent auch Token um Aktionen auszuführen. Mit einem Token kann der Glättungsagent genau einen Punkt glätten, danach wird ein benachbarter Punkt für die nächste Glättungsoperation ausgewählt. Mit einem zusätzlich Parameter lässt sich festlegen, wie oft der Agent zum Ausgangspunkt zurückkehren soll.

3.6.3 Strandagenten

Strandagenten sind dafür zuständig, flache Bereiche in der Nähe der Küstenlinien zu schaffen. Sie starten jeweils an einem beliebigen Punkt der Küste, für die Suche von Küstenpunkten benützten sie einen *Breath-First-Search* Algorithmus. Ausgehend von gefundenen Küstenpunkten führen sie zufällige Abflächungsoperationen entlang der Küstenlinien durch, dadurch erzeugen sie Strände. Stoßen sie dabei auf einen Höhenwert, der über einem einstellbaren Grenzwert liegt, springen sie zurück an die Küste.



(a) (b)

Abbildung 3.15: Durch Strandagenten erzeugte Strände mit unterschiedlichen Strandgrenzen [3, S. 115]. (a) Niedrige Strandgrenze, (b) hohe Strandgrenze.

Strandagenten bieten dem User auch einige Einstellungsmöglichkeiten, zum Beispiel das Höhenlimit und den Höhenwert, ab dem das Gelände nicht mehr als Strand klassifiziert werden soll. Zusätzlich kann der User bestimmen, in welchem Höhenbereich sich die generierten Strandabschnitte befinden sollen, und wie weit ins Landesinnere sich Strände ausbreiten dürfen. In Abb. 3.15 sieht man einige so erzeugte Strände.

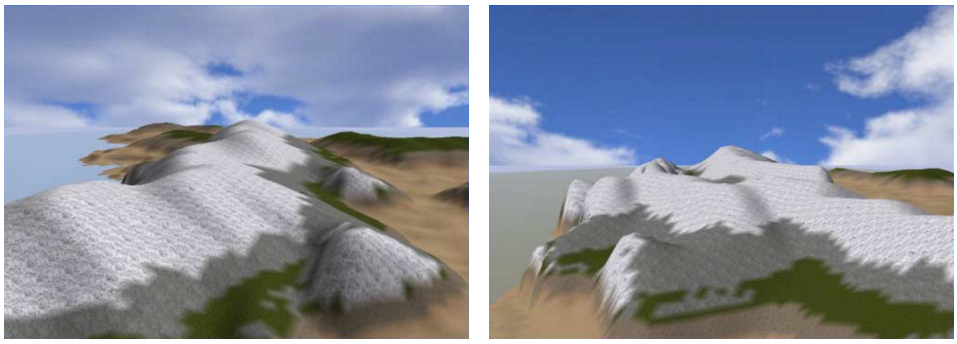
3.6.4 Bergagenten

Ziel eines Bergagenten ist es Bergketten zu erzeugen, dazu starten er auf einem zufälligen Punkt in der Landschaft. Im nächsten Schritt wählt der Agent eine bestimmte Richtung aus und bewegt sich in diese. Trifft er dabei auf ein Hindernis, wie zum Beispiel einen Ozean, versucht er diesem auszuweichen. Während der Bergagent sich bewegt erzeugt er eine Bergkette, die einem invertierten V ähnelt. Die Breite des invertierten V definiert, wie weit ein Berg sich ausbreitet und wie steil er abfällt. Ein Beispiel für diese Einstellungen zeigt Abb. 3.16. Zum Strukturieren der seitlichen Hänge werden Werte aus einem definierten Wertebereich ausgewählt, durch diesen können unterschiedlich abfallende Hänge erzeugt werden.

Eine besondere Form der Bergagenten sind die sogenannten Hügelagenten. Diese funktionieren genauso wie die Bergagenten mit dem Unterschied, dass sie keine Nebenhügel sondern nur kurze Hügelketten mit niedriger Höhe erzeugen. Abb. 3.17 zeigt ein Beispiel dazu.

3.6.5 Flussagenten

Flussagenten erzeugen Flussverläufe zwischen einem zufälligen Punkt der Küstenlinie und einem zufälligen Punkt auf einer Bergkette. Zum Formen



(a) (b)

Abbildung 3.16: Durch Bergagenten erzeugte Berge [3, S. 115]. (a) Berg mit geringer Ausbreitungsweite, (b) Berg mit hoher Ausbreitungsweite.

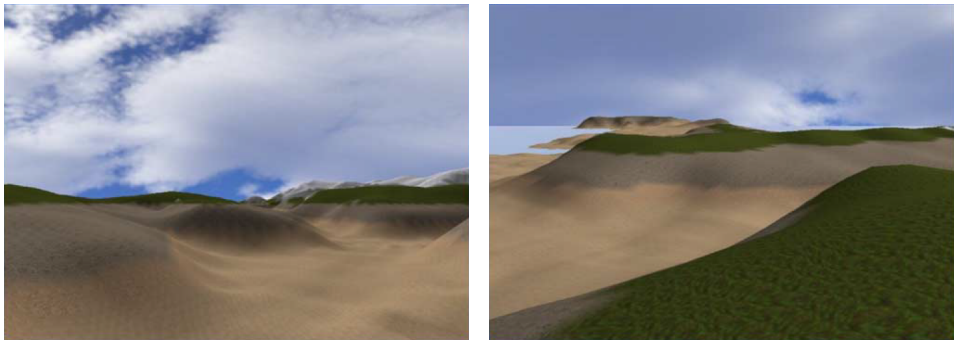
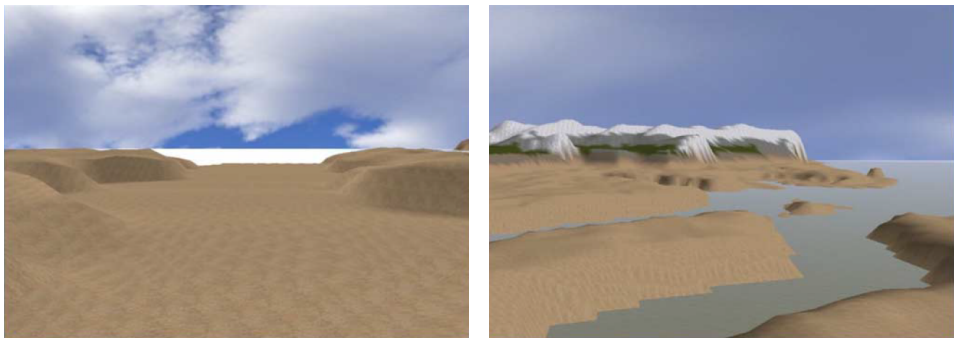


Abbildung 3.17: Hügel, die durch Hügelagenten generiert wurden [3, S. 116].

des Flussverlaufes geht der Agent folgendermaßen vor:

1. Auswählen zweier zufälliger Startpunkte p_c und p_m, wobei p_c ein zufälliger Punkt auf einer Küstenlinie ist und p_m ein zufälliger Punkt auf einer Bergkette.
2. Im nächsten Schritt wandert der Agent von p_c nach p_m. Um eine realistische Darstellung des Flussverlaufes zu erreichen, berücksichtigt er dabei die Steigung der Landschaft und die generelle Richtung von p_c nach p_m. Er speichert sich seinen gewanderten Pfad.
3. Zuletzt wandert der Agent den Pfad vom Berg zur Küste zurück und formt dabei eine Schneise im Gelände. Die so erzeugte Struktur wird gleichzeitig geglättet. Diese beginnt sehr schmal und wird größer, je länger der Agent sich bewegt. Trifft der Agent dabei auf einen anderen Fluss, beendet er diesen Vorgang und führt die Flüsse zusammen.



(a) (b)

Abbildung 3.18: (a) Erzeugter Fluss mit kleinem initialen Höhenabfall. (b) Einsatz von mehreren Flussagenten mit niedriger minimaler Distanz zwischen Berg und Küstenlinie [3, S. 117].

Findet der Flussagent den Berg zu früh, startet er den Algorithmus erneut mit anderen Startwerten. Sollte aufgrund der Struktur des Geländes kein Fluss erzeugt werden können bricht der Agent automatisch nach einigen Versuchen ab. Damit wird verhindert dass zu viele kurze Flüsse in einer Landschaft entstehen.

Flussagenten sind wie auch andere Agenten konfigurierbar, ihre Parameter sind:

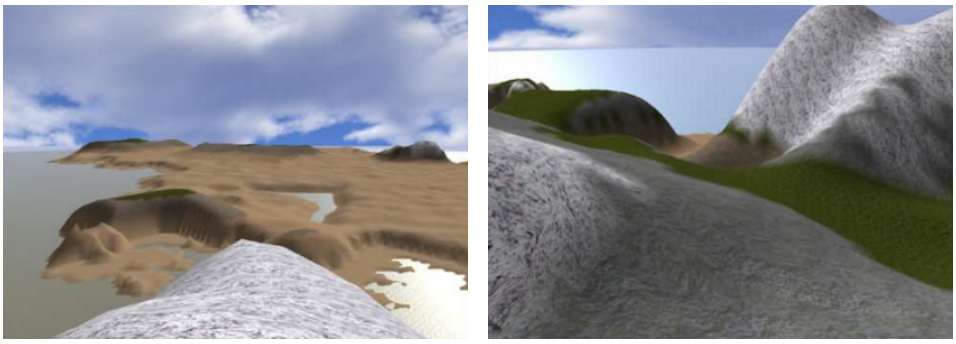
1. *Minimale Höhe des Punktes* p_m (Punkt auf der Bergkette): verhindert, dass Flüsse in zu flachen Landschaften starten.
2. *Maximale Höhe des Punktes* p_m: gibt an, wie hoch maximal der Punkt auf der Bergkette sein darf, damit Flüsse nicht direkt auf einer Bergspitze beginnen.
3. *Minimale Distanz*: gibt die Mindestlänge des Flusses an, diese wird mit der Entfernung von Punkt p_c auf der Küstenlinie zum Punkt p_m auf dem Berg verglichen.
4. *Anfängliche Breite des Flusses.*
5. *Frequenz*: bestimmt die Rate mit der sich die anfängliche Breite erhöht.
6. *Tiefe des Flussbettes.*

Ein Beispiel für Ergebnisse mit unterschiedlichen Parametereinstellungen ist in Abb. 3.18 ersichtlich.

3.6.6 Bewertung

In diesem Abschnitt werden die softwarebasierten Agenten durch die im Bewertungskatalog (s. Abschn. 3.1) definierten Kriterien bewertet:

1. *Kontrollierbarkeit*: Der Ansatz der softwarebasierten Agenten ist zu



(a) (b)

Abbildung 3.19: Zwei Terrains (a) und (b), die mittels softwarebasierten Agenten erzeugt wurden [3, S. 118].

einem hohen Grad kontrollierbar. Die gute Kontrolle über den Generierungsprozess ergibt sich dadurch, dass der Benutzer sowohl Anzahl als auch Art der verwendeten Agenten bestimmen kann. Zusätzlich kann er pro Agent auch noch detaillierte Einstellungen vornehmen.

2. *Echtzeitfähigkeit*: In dem Paper von Doran und Parberry [3, S. 117], wird erwähnt, dass ihr Ansatz nicht auf Echtzeitanwendungen ausgelegt ist. Da jeder Agent jedoch nur über eine fixe Anzahl von Aufgaben verfügt, die er in einem Durchgang ausführen muss, ist es denkbar, dass der Ansatz auch in Echtzeit funktionieren könnte.
3. *Endlosigkeit*: Basierend auf den bisherigen veröffentlichten Arbeiten zu den softwarebasierten Agenten ist es schwer zu sagen, ob der Ansatz sich für endloses Terrain eignen würde. Da einzelne Agenten jedoch immer in einem lokalen Bereich arbeiten, besteht die Möglichkeit, dies auf endlose Welten auszuweiten.
4. *Nötige Benutzereingaben*: Bei den softwarebasierten Agenten sind zu Beginn viele Benutzereingaben nötig, im eigentlichen Prozess der Generierung werden jedoch keine mehr gebraucht.
5. *Qualität*: Wie in Abb. 3.19 zu sehen ist, sind die softwarebasierten Agenten durchaus in der Lage, realitätsnahes Terrain zu generieren.

3.7 Ansatzvergleich

Der nachfolgende Abschnitt liefert, basierend auf den Kriterien des Bewertungskataloges (s. Abschn. 3.3), einen Vergleich der untersuchten Verfahren zueinander.

Die in diesem Kapitel vollzogene Analyse (Tab. 3.3) der Methoden zeigt sehr gut, dass die verschiedenen Ansätze unterschiedliche Stärken und Schwä-

	Pseudo-Zufallszahlen	**Sketchbasierte Methoden**	**Evolutionäre Algorithmen**	**Softwarebasierte Agenten**
Maß der Kontrollierbarkeit	gering	hoch	mittelmäßig	sehr hoch
Echtzeitfähigkeit	ja	nein	vielleicht	vielleicht
Endlosigkeit	ja	nein	nein	vermutlich
Initial nötige Benutzereingaben	wenige	viele	mittelmäßig viele	sehr viele
Nötige Benutzereingaben während der Generierung	keine	keine	methodenabhängig (viele oder keine)	keine
Qualität	sehr gut	sehr gut	mittelmäßig	gut

Tabelle 3.3: Überblick über die Eigenschaften verschiedener Ansätze zur Generierung von Landschaften, basierend auf den definierten Kriterien des Bewertungskataloges (siehe Abschn. 3.1).

chen aufweisen. So sind zum Beispiel die auf Pseudo-Zufallszahlen basierenden Methoden sehr schnell, die sketchbasierten hingegen wesentlich kontrollierbarer.

Hauptziel dieser Bewertung war es herauszufinden, welche Methoden sich am besten eignen, um kontrolliert endloses Terrain in Echtzeit zu erzeugen. Aus den gewonnen Erkenntnissen in Tab. 3.3 scheint der Ansatz der softwarebasierten Agenten der vielversprechendste zu sein, da dieser sowohl ein hohes Maß an Kontrolle als auch gute Ergebnisse liefert. Lediglich die Eignung des Ansatzes zur Generierung von endlosem Terrain in Echtzeit bleibt unklar. Dies führt uns zum Ziel dieser Arbeit, den Ansatz der softwarebasierten Agenten zur Generierung von endlosem Terrain in einer Echtzeit-Anwendung zu verwenden.

Kapitel 4

Lösungsansatz und Umsetzung

Das nachfolgende Kapitel erklärt den im Rahmen dieser Arbeit umgesetzten Ansatz. Es wird darauf eingegangen, welche Probleme sich ergeben, wenn man den Ansatz der softwarebasierten Agenten zur Generierung von endlosem Terrain verwenden möchte. Darüber hinaus wird die Architektur der entwickelten Agenten beschrieben sowie deren Vorgehensweise.

4.1 Anforderungen an den Lösungsansatz

Aufgrund der verfolgten Ziele bei der Umsetzung des Lösungsansatzes ergeben sich einige Anforderungen. Diese sind nachfolgend aufgelistet.

4.1.1 Endloses Terrain

Ein Ziel ist es, endloses Terrain erzeugen zu können. Dadurch ergibt sich die Anforderung, den Ansatz der softwarebasierten Agenten so verändern zu müssen, dass dies ermöglicht wird. Endlos bedeutet in diesem Kontext, dass ein Benutzer sich unbegrenzt lange über die erzeugte Landschaft bewegen kann, ohne zweimal an derselben Stelle anzukommen.

4.1.2 Echtzeitfähigkeit

Die Echtzeitfähigkeit der entwickelten Lösung ist eine weitere Anforderung. Es ist nicht möglich, eine endlose Landschaft in einem Schritt zu generieren, diese muss nach und nach erzeugt werden. Dieser Prozess sollte für den Benutzer unsichtbar sein und im Hintergrund passieren, um dies zu erreichen muss die Landschaft schneller generiert werden als der Benutzer sich im Spiel bewegt.

4.1.3 Kontrollierbarkeit

Die Forderung nach Echtzeitfähigkeit und Endlosigkeit führt dazu, dass eine modifizierte Variante des Ansatzes der softwarebasierten Agenten (siehe Abschn. 3.6) benötigt wird, da dieser nicht auf das Generieren von einzelnen Terrainsegmenten ausgelegt ist. Gleichzeitig muss aber die Kontrollierbarkeit in der modifizierte Version, mindestens so gut wie im originalen Ansatz, erhalten bleiben.

4.2 Lösungsidee

Im Rahmen dieser Arbeit war es nötig einen Lösungsansatz zu entwickeln, der all diese in Abschn. 4.1 definierten Anforderungen erfüllen kann. Dieser muss sowohl endloses Terrain erzeugen können, als auch vom Benutzer kontrollierbar sein. Der ursprüngliche Ansatz der softwarebasierten Agenten bietet bereits einen sehr hohen Grad an Kontrolle. Ein Ziel bei der Entwicklung des Lösungsansatzes war es, diesen nicht grundlegend zu verändern.

Eine positive Eigenschaft des Ansatzes von Doran und Parberry die hier zum Tragen kommt, ist die Tatsache, dass dieser bereits unterschiedlich große Terrains erzeugen kann [3, S.117]. Basierend auf dieser Eigenschaft ist es auch möglich, Terrain in Segmente aufzuteilen. Einzelne Segmenten ergeben so richtig angeordnet das komplette Terrain (siehe Abb. 4.1).

Die Idee der Segmentierung hat zusätzlich die gute Eigenschaft, die Anforderung der Echtzeitfähigkeit besser zu erfüllen. Einzelne Segmente können je nach Bedarf generiert werden. Ein Segment hat dabei immer eine fixe Größe und ist quadratisch. Zu Beginn könnten 3×3 Segmente erzeugt werden. Wenn ein Spieler über den Rand eines solchen Segmentes schreitet, kann das nächste passende berechnet und angezeigt werden. In der Abb. 4.2 wird diese Idee grafisch veranschaulicht. Dabei würden immer genau acht Segmente das aktuelle umschließen und so dem Spieler den Eindruck einer endlosen Welt geben.

Die nötigen Veränderungen am Ansatz der softwarebasierten Agenten verursachen auch einige Probleme. Diese sind nachfolgend erklärt.

4.2.1 Zuständigkeitsbereich der Agenten

Der ursprüngliche Ansatz baut auf der Idee auf, dass Agenten jeweils Zugriff auf das gesamte bestehende Terrain haben und basierend auf diesem Entscheidungen treffen können. Die Segmentierung führt jedoch dazu, dass dieser Zugriff nicht mehr möglich ist. Daraus ergibt sich die Notwendigkeit, dass Agenten jeweils in einem zugewiesenen Abschnitt arbeiten müssen.

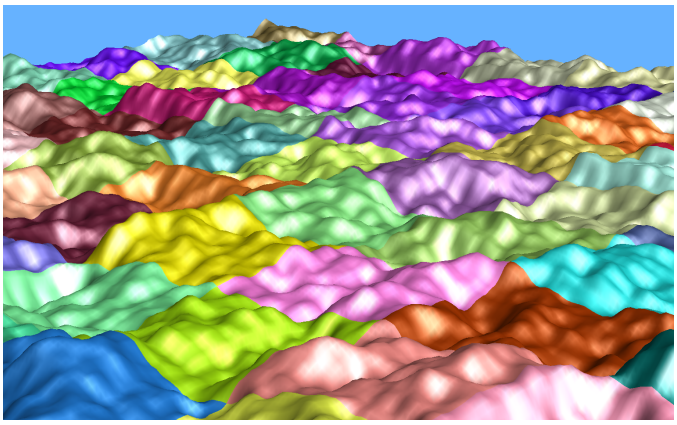
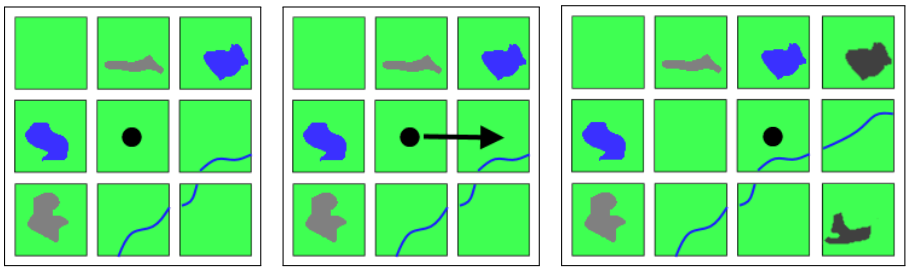


Abbildung 4.1: Segmentiertes Terrain, in dem jede Farbe ein Segment darstellt [21].



(a) (b) (c)

Abbildung 4.2: Prozess der Segmentgenerierung. (a) Ausgangslage, Spieler befindet sich im Segment mit den Koordinaten (1/1). (b) Spieler bewegt sich vom Segment (1/1) ins Segment (2/1). (c) Während der Spieler die Grenze zwischen (1/1) und (2/1) überschreitet, werden rechts 3 neue Segmente generiert und dargestellt.

4.2.2 Glättung des Terrains

Die Glättungsagenten (siehe Abschn. 3.6.2) arbeiten ausschließlich in einem lokalen Bereich, eine Vielzahl dieser Agenten glättet so nach und nach die Landschaft. Dies ist jedoch ein zeitintensiver Prozess. Dieses Problem kann gelöst werden, indem ein einziger Agent für das Glätten eines kompletten Segmentes zuständig ist.

4.2.3 Performance

Der ursprüngliche Ansatz von Doran und Parberry ist nicht auf den Echtzeiteinsatz ausgelegt. Dies zeigt sich vor allem bei den Glättungsagenten, die wie bereits erwähnt optimiert wurden.

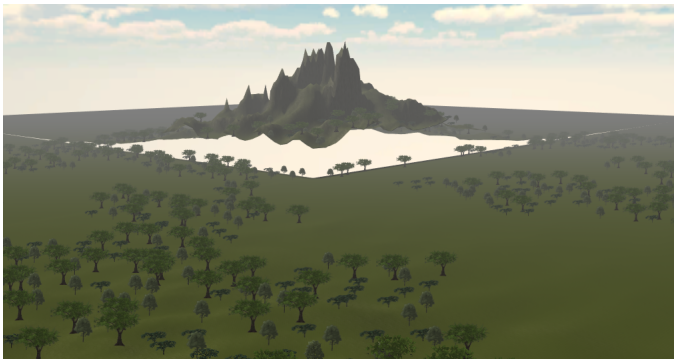


Abbildung 4.3: Darstellung eines Berges ohne Berücksichtigung von Überlappungen am Rand.

Eine weitere Verbesserung in Bezug auf die Performance des Ansatzes konnte durch die Aufteilung der Agenten in Filter- und Landschaftsagenten erzielt werden. Die Filteragenten werden auf das Resultat der Landschaftsagenten angewendet, um dieses zu verbessern. Sie benützen dazu Glättungs-, Erosions- und Verschiebungsoperationen. Durch die Filterung des Terrains genügt es, wenn Landschaftsagenten nur simple Berge und Seen erzeugen. Diese einfachen Strukturen können sehr effizient generiert werden.

4.2.4 Segmentübergänge

Eines der Hauptprobleme das mit der Segmentierung einhergeht, sind Übergänge an den Rändern der Segmente (siehe Abb. 4.3). Es kann der Fall eintreten, dass ein Bergagent einen Berg erzeugt, dessen Ausläufer den Rand seines Segmentes überschreiten. Wenn das passiert, ist es nötig, Teile des Berges in die Heightmaps der benachbarten Abschnitte einzutragen. Sollte ein benachbarter Abschnitt noch nicht existieren, muss dieser erzeugt werden.

4.2.5 Effekt von neuem Terrain auf bereits bestehendes Terrain

Bei der Erzeugung eines neuen Segmentes kann es passieren, dass da die Ausläufer eines Berges sich auch auf bereits dargestelltes Terrain auswirken können. Wenn dieser Fall eintritt, müssen bestehende Heightmaps nochmals gefiltert werden. Da Filteroperationen jeweils die ganze Struktur des Terrains verändern, ist es nötig, für jedes Segment eine Heightmap mit ungefiltertem und gefiltertem Terrain zu speichern. Dadurch ist es möglich ungefiltertes Terrain erneut filtern zu können.

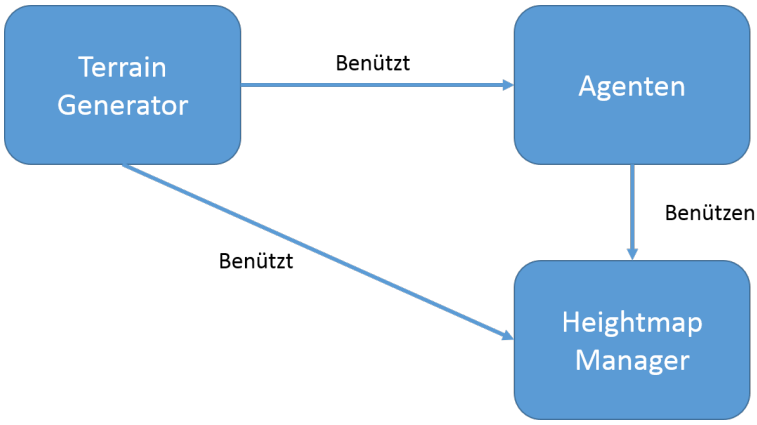


Abbildung 4.4: Übersicht über die zentralen Komponenten dieses Ansatzes.

4.3 Architektur

Dieser Abschnitt dient dazu dem Leser einen Überblick über die entwickelten Komponenten dieses Ansatzes zu geben. Diese sind die folgenden drei:

- *Terrain Generator*: steuert den Generierungsprozess.
- *Heightmap/Segment Manager*: dient als Zugriffsstelle auf bestehende Heightmaps.
- *Agenten*: erzeugen die Features der einzelnen Segmente.

Wie diese drei Komponenten zueinander stehen, ist in Abb. 4.4 zu sehen. Nachfolgend werden die einzelnen Komponenten und deren Beziehungen zueinander genauer erklärt.

4.3.1 Terrain Generator

Die Aufgabe des Terrain Generators ist es, den Generierungsprozess zu steuern. Er ist die zentrale Komponente, die alle anderen miteinander verbindet. Zum Generieren des endlosen Terrains geht er folgendermaßen vor:

1. Basierend auf den Einstellungen des Benutzers werden zuerst n Segmente erzeugt. Wäre $n = 9$ würden, wie in Abb. 4.5 dargestellt, alle Segmente rund um $(x = 0/z = 0)$ angeordnet werden. Im ersten Schritt wird dabei ein ungefiltertes Terrain erzeugt. Dies geschieht über den Einsatz von Berg- und Seeagenten.
2. Im zweiten Schritt wird das bisher ungefilterte Terrain gefiltert und dargestellt (siehe Abb. 4.6). Der Spieler wird anschließend in der Mitte des Segmentes (0/0) platziert.
3. Nach den ersten Schritten beobachtet der Terrain Generator immer die aktuelle Position des Spielers. Sobald sich dieser über die Grenzen

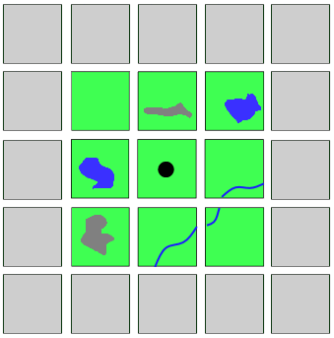


Abbildung 4.5: Anordnung der Segmente, die bei einer initialen Einstellung von $n = 9$ erzeugt werden. Erzeugte Segmente werden grün dargestellt. Der Schwarze Punkt markiert den Spieler, dieser befindet sich in diesem Fall auf $(x = 0/z = 0)$.

eines Segments hinaus bewegt, wird ein weiterer Generierungsprozess gestartet. Dieser besteht aus folgenden Schritten:

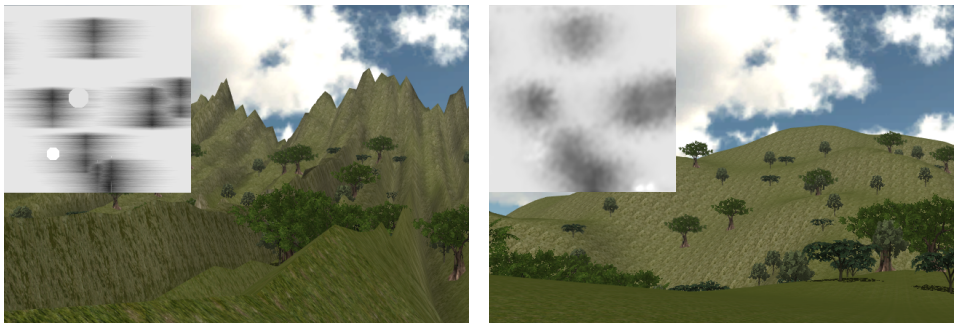
(a) Zuerst wird festgestellt, in welche Richtung der Spieler sich bewegt hat. Wenn dieser zum Beispiel vom Segment $(x = 0/z = 0)$ in das Segment $(1/0)$ gewandert ist, müssen alle noch nicht existierenden Segmente rund um $(1/0)$ erzeugt werden (s. Abb. 4.2).

(b) Da neu generierte Segmente auch Einfluss auf bereits existierende haben können, ist es nötig das der Heightmap Manager sich merkt, welche Segmente verändert wurden. Wurden bestehende geändert, müssen diese nochmals angepasst werden. Dies geschieht durch das erneute Filtern des veränderten Terrains. Führt man diesen Schritt nicht durch, ergeben sich nur teilweise gefilterte Strukturen wie in Abb. 4.7 dargestellt.

4.3.2 Heightmap Manager

Der Heightmap Manager dient dazu, die erzeugten Heightmaps aller Segmente zu verwalten. Er benützt dafür eine *DoubleKeyHashMap*, um den schnellen Zugriff auf einzelne Heightmaps zu gewährleisten. Als Schlüssel der *HashMap* werden dabei die x/z Koordinaten der Segmente verwendet. Zusätzlich hat der Heightmap Manager noch die Aufgabe, gefilterte und ungefilterte Varianten des Terrains zu verwalten und bereitzustellen.

Eine andere Funktion des Heightmap Managers ist es, das Erstellen neuer Agententypen zu vereinfachen. Er bietet dazu Methoden an, die es dem User ermöglichen, auf einer Heightmap zu arbeiten ohne Größenbeschränkung berücksichtigen zu müssen. Dies geschieht indem er automatisch Überläufe am Rand einer Heightmap erkennt und die zugehörigen Werte ins passende



(a) (b)

Abbildung 4.6: Unterschiede zwischen gefiltertem und ungefiltertem Terrain. (a) Ungefiltertes Terrain mit Heightmap, (b) gefiltertes Terrain mit Heightmap.

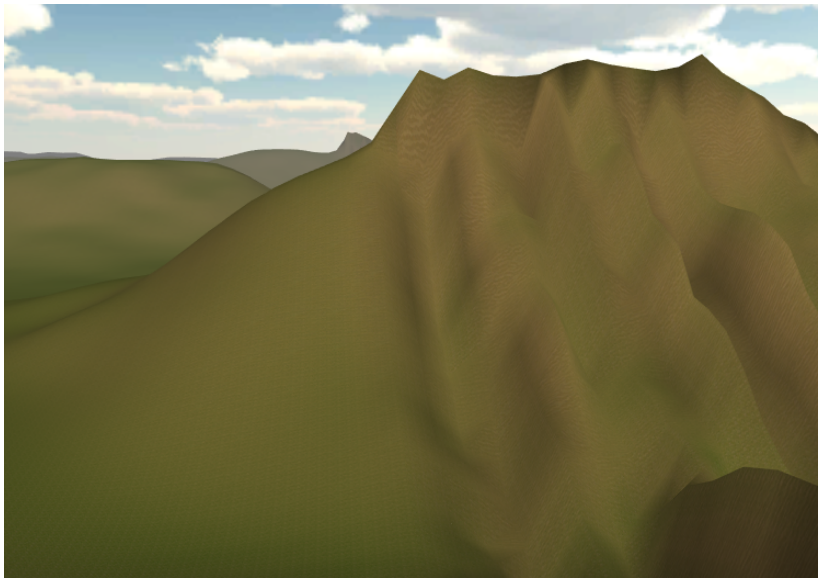


Abbildung 4.7: Ergebnis, wenn ein bestehendes und verändertes Segment nicht nochmals gefiltert wurde. Der linke Abschnitt des Bildes wurde gefiltert, der rechte nicht.

benachbarte Segment überträgt. Dadurch wird das Umsetzen von neuen Agenten sehr vereinfacht.

4.3.3 Agenten

Der folgende Abschnitt beschäftigt sich mit den im Rahmen dieser Arbeit umgesetzten Agenten. In Abb. 4.8 ist eine Übersicht aller Agentenkatego-

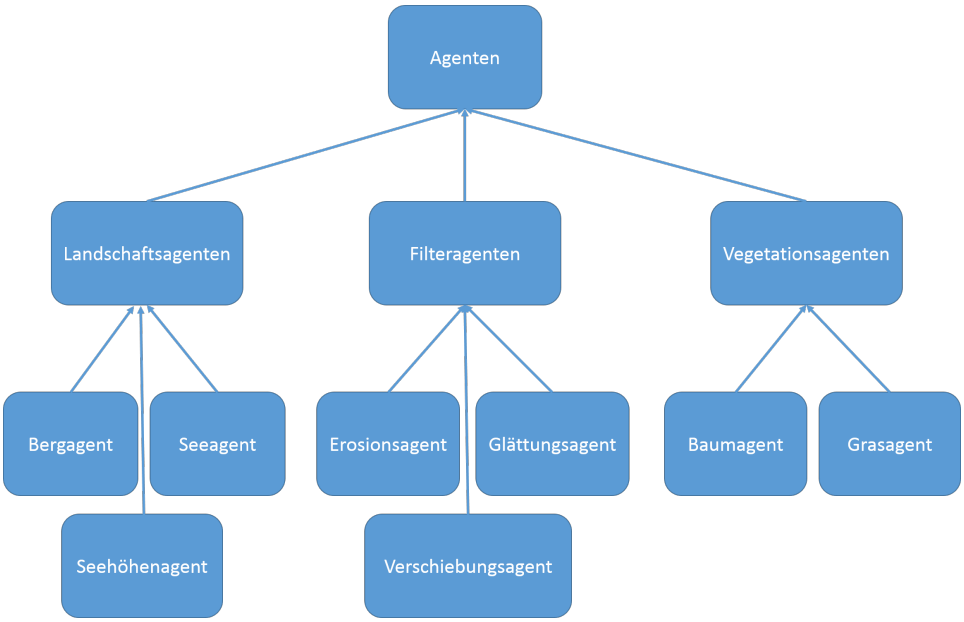


Abbildung 4.8: Übersicht der entwickelten Agenten unterteilt nach Kategorien.

rien ersichtlich sowie deren zugehörige Agenten. Die Agenten sind in drei Kategorien aufgeteilt:

1. *Landschaftsagenten*: diese sind dafür zuständig die grundlegende Struktur der Landschaft zu generieren.
2. *Filteragenten*: sind dafür zuständig von Landschaftsagenten erzeugtes Terrain zu überarbeiten.
3. *Vegetationsagenten*: dienen dazu Objekte wie Bäume und Pflanzen auf der gerenderten Landschaft zu platzieren.

Für die Erzeugung eines Segmentes werden jeweils Agenten aus allen Kategorien verwendet. Es ist dabei dem User überlassen, wie oft ein bestimmter Agent zum Einsatz kommt. Ist dessen Ziel zum Beispiel eine Landschaft mit vielen Bergen zu erzeugen, würde man mehrere Bergagenten verwenden. Will der Benutzer hingegen viele Seen und vereinzelt Berge erzeugen, würde man mehrere Seeagenten verwenden und nur einen oder zwei Bergagenten.

4.3.3.1 Landschaftsagenten

Im Rahmen dieser Arbeit wurden drei Landschaftsagenten umgesetzt. Der Seehöhen-, der Berg- und der Seeagent. Diese sind dafür zuständig, eine leere Heightmap mit interessanten Strukturen zu füllen. Die Reihenfolge, in der diese Agenten ausgeführt werden, wirkt sich dabei auf das erzeugte Ergebnis aus. Folgende Reihenfolge wurde in der Umsetzung verwendet:

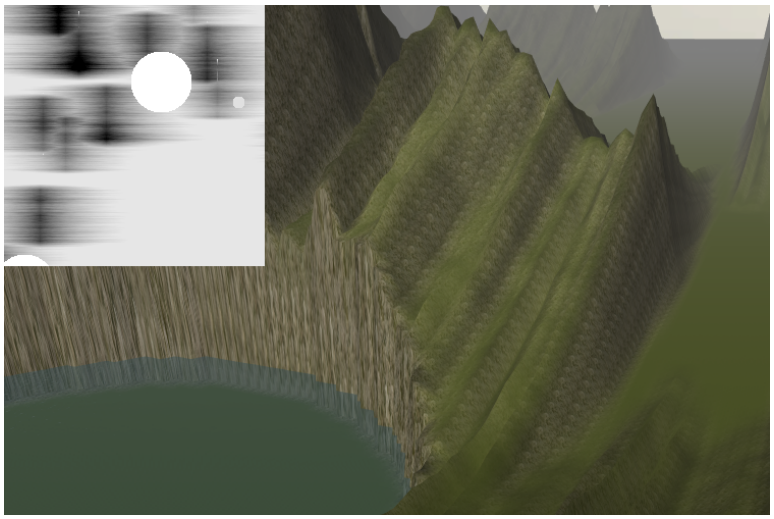


Abbildung 4.9: Ungefilterte Landschaft mit Heightmap, die durch Berg- und Seeagenten erzeugt wurde.

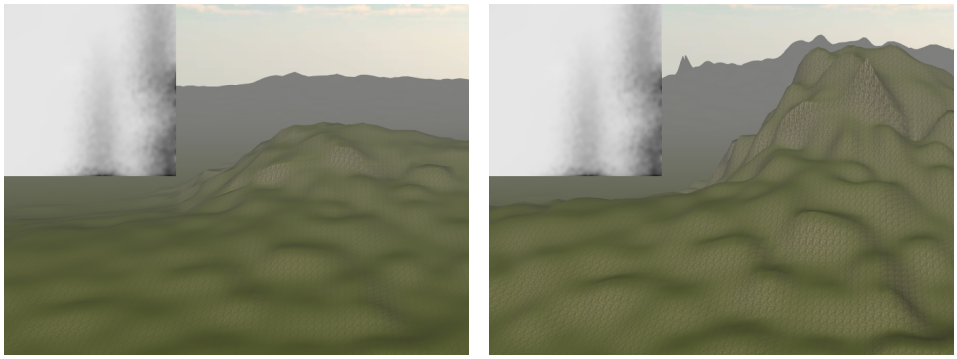
1. Der *Seehöhenagent* wird als erster ausgeführt um die allgemeine Höhe des Terrains festzulegen.
2. Als nächstes wird der *Bergagent* verwendet um Berge und Hügel zu erzeugen.
3. Zuletzt wird der *Seeagent* verwendet.

Grund für diese Reihenfolge ist, dass Ausläufer von neuen Bergen teilweise Seen verdecken können, dies kann zu schlechteren Ergebnissen führen. Dasselbe kann natürlich auch umgekehrt passieren, jedoch ist das Ergebnis dabei meist besser. In Abb. 4.9 ist eine Landschaft zu sehen die so generiert wurde.

Seehöhenagent

Der Seehöhenagent ist ein spezieller Agent, da er im Vergleich zu allen anderen Agenten höchstens einmal vorkommt. Die Aufgabe dieses Agenten ist es, die Grundhöhe h der Landschaft festzulegen, er wird beim Erzeugen jedes neuen Landschaftssegments genau einmal ausgeführt.

Seine Vorgehensweise ist im Vergleich zu den anderen Agenten relativ einfach, er setzt lediglich alle Werte einer Heightmap auf die Grundhöhe h. Dabei ist jedoch zu beachten, dass die Höhe immer in einem Wertebereich von 0–1 angegeben wird und einer relativen Größe entspricht. Es bleibt letztendlich dem User überlassen, mit welchem Höhenwert die Heightmap gerendert wird. So hat zum Beispiel eine Landschaft die mit 512 Höheneinheiten dargestellt wird, viel steilere Berge als eine mit 256 dargestellte (siehe Abb. 4.10).



(a) (b)

Abbildung 4.10: Unterschied zwischen zwei gleichen Landschaften, die mit unterschiedlichen Höhenwerten dargestellt werden. (a) Höhenwert von 256 Einheiten, (b) Höhenwert von 512 Einheiten.

Bergagenten

Der Bergagent ist einer der wichtigsten Agenten, da er dafür zuständig ist, markante Landschaftsmerkmale wie Hügel, Berge und Gebirgsketten zu erzeugen. Ziel bei der Entwicklung des Bergagenten war es, dem User möglichst viel Kontrolle über das zu erzeugende Ergebnis zu geben. Da die Filteragenten für das Verbessern der Optik zuständig sind, reicht es für den Bergagenten, einfache Berge effizient erzeugen zu können. Dazu geht er folgendermaßen vor:

- Zuerst wird ein zufälliger Startpunkte p_c ausgewählt, der innerhalb des Segmentes liegt, in dem der Berg platziert werden soll. Anschließend werden die folgenden drei Phasen zum Erzeugen des Berges ausgeführt:

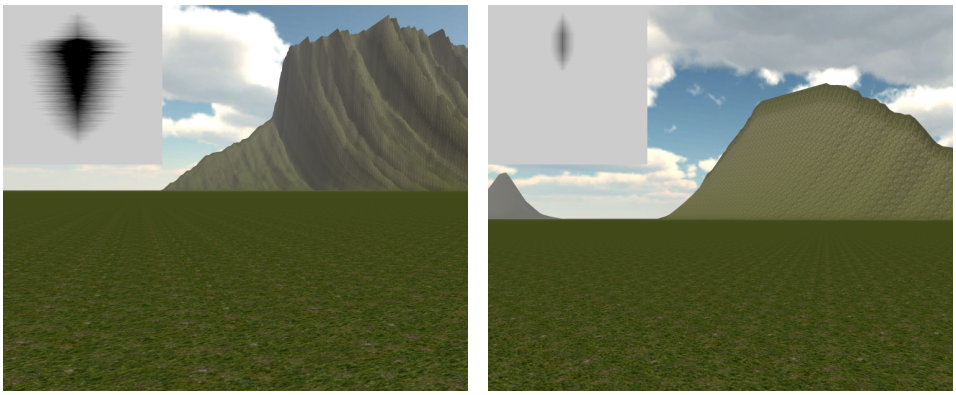
  1. In der ersten ist das Ziel, eine vom User eingestellten Grundhöhe h_g zu erreichen und somit den Anfang des Berges zu erzeugen. Dazu wird beginnend bei der eingestellten Grundhöhe des Seehöhenagents h_s, in einem vom User einstellbaren Wertebereich, ein Höhenwert h schrittweise erhöht. Nach jeder solchen Erhöhung wandert der Agent in eine vom User eingestellte Richtung r. Zusätzlich werden bei jeder Erhöhung die Ausläufer für den aktuellen Höhenwert erzeugt.

     Die Ausläufer starten jeweils mit dem aktuellen Höhenwert h. Sie verlaufen dabei immer rechts und links in einem 90° Winkel zur aktuellen Ausrichtung r der Bergkette. Mittels eines vom User einstellbaren Wertebereichs wird bestimmt, wie schnell diese Ausläufer abfallen sollen. Dieser Wertebereich beschreibt eine prozentuale Änderung pro Höhenwert. Der nächste Höhenwert h

der Ausläufer ergibt sich immer mittels $h = h \cdot sp$ wobei sp ein zufälliger Wert aus einem einstellbaren Wertebereich ist. Dieser Prozess wird solange wiederholt, bis $h \leq h_s$ erreicht ist, wobei h_s dabei die Seehöhe ist.

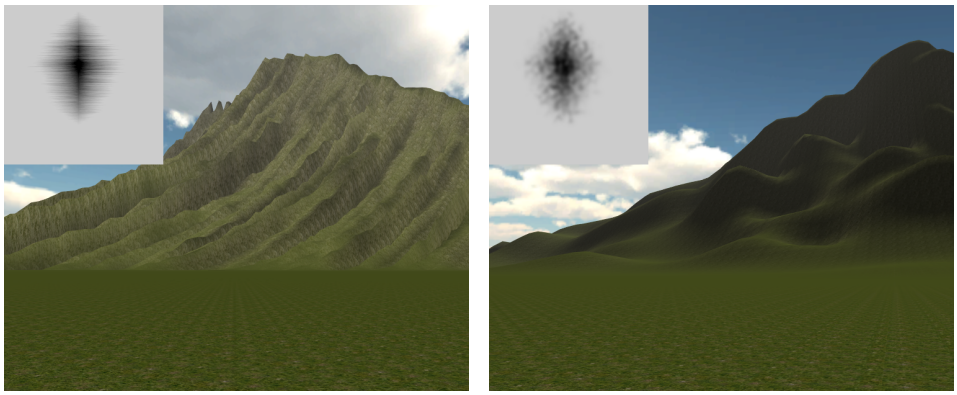
2. In der zweiten Phase wird der Hauptteil der Bergkette erzeugt. Es wird dabei für jeden vom User eingestellten Token t genau ein Höhenwert in der Heightmap generiert. Die Position des nächsten Höhenwertes ist dabei jeweils von der Richtung r abhängig. Für r wird ein Vektor verwendet, der alle r_n Schritte 90° nach rechts oder links rotiert. Für jeden durch einen Token erzeugten Höhenwert werden wie im ersten Schritt auch die Ausläufer erzeugt. Zusätzlich kann der Benutzer noch bestimmen in welchem Bereich die Höhenwerte des Berges variieren sollen.
3. In der letzten Phase wird das Bergende generiert. Statt von der Grundhöhe h_s zur Berghöhe h_g zu gehen, wird mit h_g beginnend solange der Höhenwert h reduziert, bis die Grundhöhe h_s erreicht ist. In dieser Phase werden ebenfalls Ausläufer erzeugt.

Der vorgestellte Algorithmus bietet dem User einige Einstellungsmöglichkeiten um verschiedenste Berge zu erzeugen. In Abb. 4.11 sind so erzeugte Berge mit variierenden und gleichbleibenden Ausläufern zu sehen. Der Agent kann sehr effizient Bergketten erzeugen, jedoch sind letztendlich die Filteragenten für das interessante Aussehen zuständig, siehe Abb. 4.12.



(a) (b)

Abbildung 4.11: Ungefilterte von Bergagenten erzeugte Bergketten. (a) Bergkette mit $t = 30$ Token, einer Grundhöhe $h_g = 60\,\%$ mit einer Variation v von $20\,\%$, wobei h_g, v relative Prozentwerte zur vom User eingestellten Grundhöhe sind. Weiters fällt die Bergkette jeweils mit $5\,\%$ bis $0\,\%$ pro Höhenwertänderung ab. (b) Bergkette mit $t = 20$ Token, einer Grundhöhe $h_g = 60\,\%$ mit einer Variation v von $0\,\%$ und einem seitlichen Abfall von konstanten $5\,\%$.



(a) (b)

Abbildung 4.12: Unterschied zwischen einer gefilterten und ungefilterten Bergkette. (a) Ungefilterte Bergkette mit $t = 40$ Token, einer Grundhöhe $h_g = 60\,\%$ und einer Variation v von $10\,\%$ und einem seitlichen Abfall von $5\,\%$ bis $0\,\%$, pro Höhenwertänderung. (b) Dieselbe Bergkette in Kombination mit den Filteragenten.

Seeagenten

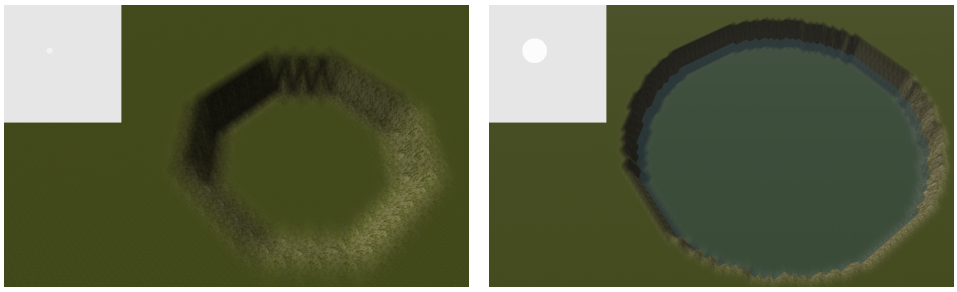
Die Seeagenten werden zum Erzeugen von Seen verwendet. Der User kann dabei einstellen, wie groß und wie tief die zu generierenden Seen werden sollen. Zusätzlich besteht die Möglichkeit einen Wertebereich anzugeben, in dem der Radius r bzw. die Tiefe t variiert werden soll. Um einen See für ein Segment zu erzeugen sind folgende Schritte nötig:

1. Zufällige Auswahl eines Punktes p_c, der den Mittelpunkt des Sees darstellt.
2. Basierend auf dem ausgewählten Radius r werden alle Punkte im Abstand von r zu p_c betrachtet. Mittels folgender Überprüfung

$$\sqrt{p_{cx}^2 + p_{cy}^2} < r \tag{4.1}$$

   wird jeweils berechnet, ob ein Punkt zum See gehört. Für alle Punkte, die diese Bedingung erfüllen, wird der zugehörige Wert in der Heightmap auf den ausgewählten Tiefenwert t geändert.

Der Algorithmus kann wie in Abb. 4.13 dargestellt verschieden große und tiefe Seen erzeugen. Diese sehen jedoch noch unspektakulär aus, erst in gemeinsamer Verwendung mit den Filteragenten werden sie zu interessanteren Seen, siehe Abb. 4.14. Weiters könnte man den Seeagenten auch verwenden um Plateaus zu erzeugen, wenn dessen verwendete Tiefeneineinstellung die allgemeine Landhöhe übersteigt.



(a) (b)

Abbildung 4.13: Zwei unterschiedlich große von Seeagenten erzeugte Seen, mit der allgemeinen Landhöhe $s = 10\,\%$. (a) See mit Radius $r = 10$ Einheiten und einer Tiefe von $t = 8\,\%$. Eine Radius Einheit entspricht jeweils genau einem Heightmapwert. (b) See mit Radius $r = 30$ Einheiten und einer Tiefe von $t = 5\,\%$.

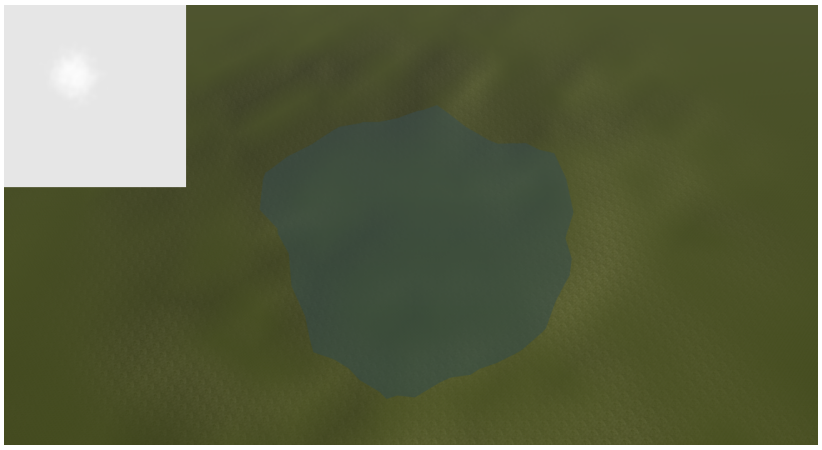


Abbildung 4.14: Von See- und Filteragenten erzeugter See mit Radius $r = 30$, einer Tiefe von $t = 5\,\%$ sowie der allgemeinen Landhöhe $s = 10\,\%$.

4.3.3.2 Filteragenten

Die Aufgabe der Filteragenten ist es, die von Landschaftsagenten erzeugten Heightmaps zu überarbeiten. Im Rahmen dieser Arbeit wurden drei dieser Agenten umgesetzt. Der Verschiebungs-, der Erosions- und der Glättungsagent. Beim Filtern spielt es eine Rolle, in welcher Reihenfolge die Agenten verwendet werden. Der Glättungsagent muss dabei immer als letztes ausgeführt werden, da dessen Ergebnis sonst von den anderen Agenten zunichte gemacht werden würde. Die besten Resultate erzielt man, indem man zuerst den Verschiebungs-, danach den Erosions- und zuletzt den Glättungsagenten einsetzt.

Verschiebungsagenten

Die Aufgabe des Verschiebungsagenten ist es, bestehendes Terrain so zu vertauschen, dass in einem lokalen Bereich unstrukturiertes Terrain entsteht (siehe Abb. 4.15). Die Idee für diesen Algorithmus basiert auf der Arbeit von Olsen [8]. Die Verschiebung wird mithilfe des Perlin Noise Algorithmus (siehe Abschn. 3.3.4) erzielt. Dieser wird verwendet, da durch ihn gleichmäßige Verschiebungen erzeugt werden können, die vom User durch die Frequenz f gesteuert werden können. Zusätzlich wird ein weiterer Parameter d verwendet, der für die maximale Weite steht, um die ein Punkt verschoben werden darf (siehe Abb. 4.16). Ein Verschiebungsagent arbeitet immer auf genau einem Segment und geht folgendermaßen vor:

1. Der Agent iteriert über jeden Punkt p der Heightmap h, die von den Landschaftsagenten erzeugt wurde. Zusätzlich erstellt er eine leere Heightmap h_n.
2. Danach wird für jeden Punkt p eine zugehörige Position p_n berechnet. p_n wird folgendermaßen ermittelt:

$$p_{nx} = p_x + f_{pn}(x, y, 0) \cdot d, \tag{4.2}$$
$$p_{ny} = p_y + f_{pn}(x, y, 1) \cdot d. \tag{4.3}$$

   p_{nx} und p_{ny} sind dabei jeweils die Koordinaten von p_n und d ist die Weite, um die der Punkt p maximal verschoben werden soll. $f_{pn}(x, y, z)$ ist eine Perlin Noise Funktion, die einen Wert zwischen 0–1 liefert. x und y sind dabei Koordinaten, die zur Berechnung dieses Wertes benötigt werden. Diese ergeben sich folgendermaßen:

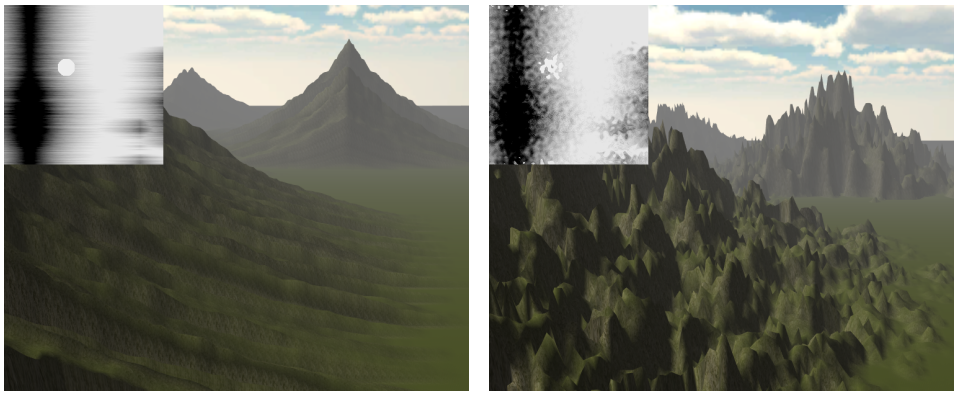
$$x = \frac{f \cdot (p_x + t_x \cdot s)}{s}, \tag{4.4}$$
$$y = \frac{f \cdot (p_y + t_y \cdot s)}{s}. \tag{4.5}$$

   Wobei hier f die Frequenz des Perlin Noise Algorithmus ist, t_x, t_y die Koordinaten der Position des aktuellen Segmentes sind und s die Größe des Segmentes. p_x und p_y sind die Koordinaten des Punktes p.
3. Zuletzt werden jeweils die Höhenwerte von p und p_n vertauscht.

Erosionsagenten

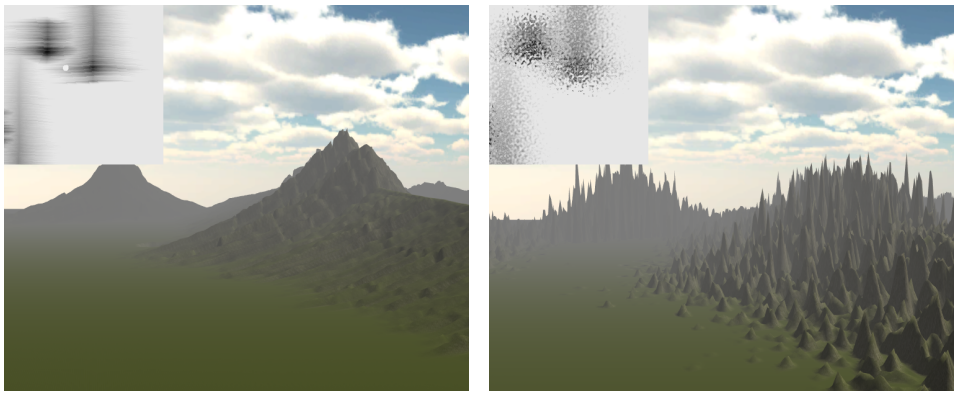
Ziel des Erosionsagenten ist es, die Effekte von Erosion zu simulieren und dabei möglichst viele begehbare Flächen für Spieler zu erzeugen. Die Umsetzung dieses Algorithmus basiert ebenfalls auf der Arbeit von Olsen [8]. Um Terrain zu erodieren, sind folgende Schritte nötig:

- Es wird über jeden Punkt p der Heightmap iteriert und dabei folgendes für jeden Punkt ausgeführt:



(a) (b)

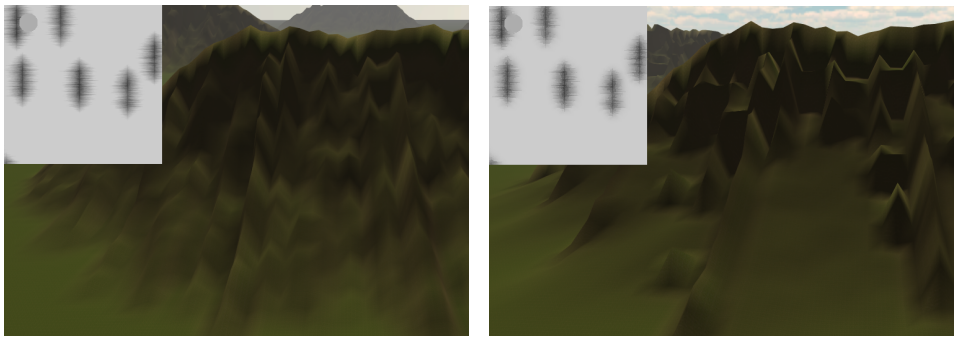
Abbildung 4.15: Unterschied zwischen verschobenem und unverschobenem Terrain. (a) Unverschobenes Terrain, (b) verschobenes Terrain mit Perlin Noise Frequenz $f = 32$ und einer maximalen Verschiebung von $d = 32$.



(a) (b)

Abbildung 4.16: Unterschiedliche Einstellungen der Verschiebungsagenten, die auf das selbe Terrain angewendet wurden. (a) Verschobenes Terrain mit $f = 2$ und $d = 16$, (b) verschobenes Terrain mit $f = 64$ und $d = 64$.

1. Es werden alle an p anliegenden Punkte betrachtet. Für jeden benachbarten Punkt wird die Höhenwertdistanz d_h zum Ausgangspunkt p berechnet. d_h berechnet sich durch $d_h = h_o - h_i$, wobei h_o der Höhenwert des Punktes p ist und h_i der Höhenwert des aktuell betrachteten Nachbarpunktes ist.
2. Basierend auf den berechneten Distanzen wird der Punkt p_c bestimmt, dessen Höhenwertdistanz d_h am größten war zum Höhenwert des Punkts p.



(a) (b)

Abbildung 4.17: Effekt des Erosionsagenten. (a) Nicht erodiertes Terrain, (b) erodiertes Terrain mit $t = 5$ Durchläufen und einem Glättungsfaktor von $g = 16$.

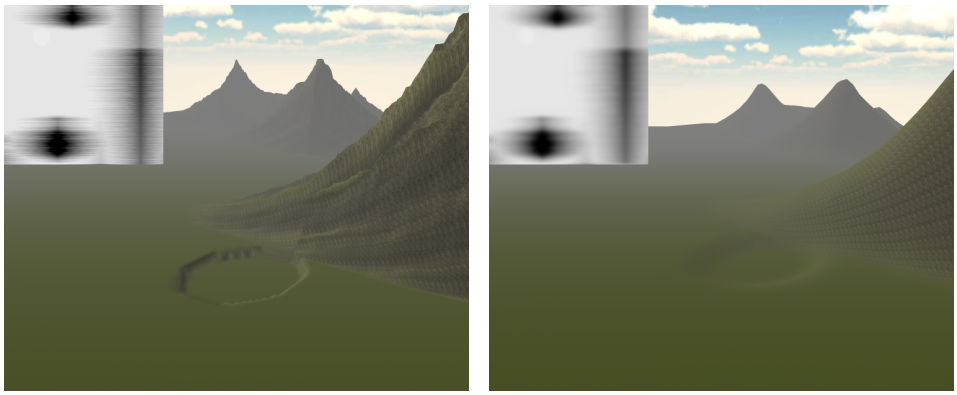
3. Für die Distanz d_h zum ausgewählten Punkt p_c werden anschließend folgende Überprüfungen durchgeführt: $d_h > 0$ und $d_h \leq \frac{g}{s}$. g ist dabei ein Glättungswert, der angibt, wie begehbar die Flächen werden sollen und s ist die Größe des aktuellen Segmentes. Sollte d_h diese Voraussetzung erfüllt haben, wird der Wert d_a mittels $d_a = 0.5 \cdot d_h$ berechnet. d_a wird vom Höhenwert des aktuellen Punktes p abgezogen und ebenfalls zum ausgewählten Punkt p_c hinzugezählt.

Der Algorithmus zum Erodieren einer Landschaft liefert die besten Ergebnisse, wenn er mehrmals ausgeführt wird. In der konkreten Umsetzung des Agenten werden die Token t verwendet um zu bestimmen, wie oft dieser Algorithmus durchgeführt werden soll. Der Effekt des Erosionsagenten auf die Heightmap bzw. die Landschaft ist in Abb. 4.17 dargestellt.

Glättungsagenten

Der Glättungsagent kommt als letzter der Filteragenten zum Einsatz. Seine Aufgabe ist es, das Terrain abzurunden und begehbarer zu machen. Die Implementierung des Glättungsagenten basiert ebenfalls auf einer in [8] beschriebenen Idee. Zum Glätten des Terrains wird dabei folgendermaßen vorgegangen:

- Es wird über jeden Punkt p der Heightmap iteriert und folgende Berechnung durchgeführt:
  - Zum Glätten des Punktes p wird ein gewichteter 3×3 Boxfilter verwendet. Das bedeutet, dass der Durchschnittswert von p und dessen umliegenden Punkten p_u berechnet wird. Dazu werden die Höhenwerte h_u all dieser Punkte p_u und der Höhenwert h



(a) (b)

Abbildung 4.18: Effekt des Glättungsagenten. (a) Nicht geglättetes Terrain, (b) geglättetes Terrain mit $t = 6$ Durchläufen und einer Gewichtung von $c = 1$.

von p zusammengezählt und im Wert s gespeichert. s ergibt sich konkret aus $s = h_u + h \cdot c$, der Faktor c gibt dabei den Wert an mit dem h gewichtet werden soll. Der neue Höhenwert h_n für p ergibt sich schließlich aus der Berechnung $h_n = \frac{s}{n}$. Dabei ist $n = 8 + c$, was der Anzahl der umliegenden Punkte samt Gewichtungsfaktor c entspricht.

Die Anzahl der Token n bestimmt beim Glättungsagenten wie oft der Glättungsprozess durchgeführt werden soll. Mehrere Durchläufe erzeugen jeweils glatteres Terrain. Zusätzlich kann der Benutzer noch die Gewichtung c für den Boxfilter bestimmten, je höher diese ist, desto weniger werden harte Kanten geglättet. Die Effekte der Glättungsoperationen sind in Abb. 4.18 dargestellt.

Der sogenannte Randagent ist eine spezielle Variante des Glättungsagenten und wird direkt nach diesem verwendet, um die Ränder zwischen den Segmenten zu glätten. Er geht grundsätzlich genau wie diese vor, allerdings iteriert er nur über den Rand der Segmente und berücksichtigt auch die Werte in den Nachbarsegmenten.

4.3.3.3 Vegetationsagenten

Ziel der Vegetationsagenten ist es Bäume und Pflanzen auf der Landschaft zu platzieren. Im Vergleich zu den bisher vorgestellten Agenten arbeiten die Vegetationsagenten nicht mehr auf den Heightmaps sonder auf dem gerenderten Terrain. In Rahmen dieser Arbeit wurden zwei dieser Agenten umgesetzt, der Baumagent und der Grasagent.

Baumagenten

Der Baumagent ist dafür zuständig, Modelle von Bäumen auf dem Terrain an realistischen Stellen zu platzieren. Dazu werden pro Terrainsegment folgende Schritte ausgeführt:

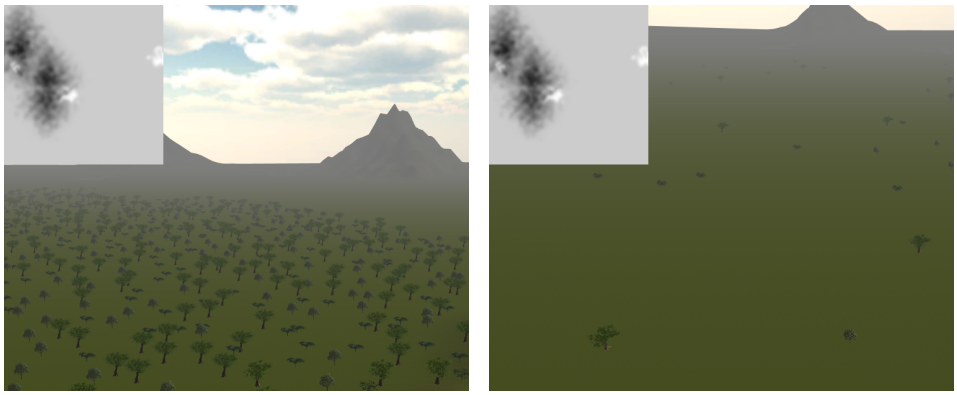
- Zu Beginn wird mittels eines vom User einstellbaren Abstandsfaktors a über die Heightmap iteriert. Dieser Faktor bestimmt dabei, wie nahe Bäume beieinander stehen sollen. Zusätzlich zum aktuellen Punkt p wird ein zufälliger Verschiebungsvektor v erzeugt, dessen Koordinaten sind dabei immer kleiner als der Abstandsfaktor a. Dieser Vektor v dient dazu zu verhindern, dass die Bäume zu gleichmäßig platziert werden. Der Punkt p_b, auf dem ein Baum nun platziert werden soll, wird nun mittels Vektoraddition[1] des Punktes p und des Verschiebungsvektor v berechnet $p_b = p + v$. Für jeden so berechneten Punkt p_b werden anschließend die Steigung s, die Höhe h der Heightmap an dem Punkt p_b und der Perlin Noise Algorithmus verwendet, um zu überprüfen, ob ein Baum platziert werden soll.
  - Die Steigung des Punktes kann über den passenden Heightmapwert zu p_b und dessen Nachbarn berechnet werden (Unity stellt für diese Berechnung eine Funktion zur Verfügung). In dieser Umsetzung wurden Bäume nur platziert, wenn die Steigung nicht größer als 45° war.
  - Ob die Höhe h passend ist, kann mittels eines vom User einstellbaren Wertebereiches bestimmt werden. Dieser dient dazu zu verhindern, dass Bäume unterhalb der Seehöhe oder auf Gipfeln von Bergen platziert werden.
  - 2D Perlin Noise wird verwendet um den Eindruck zu vermitteln, dass Bäume neben anderen Bäumen wachsen und somit eine Gesamtheit bilden. Es können separate Flächen sowohl mit, als auch ohne Bäumen generiert werden. Die von Perlin Noise verwendete Frequenz f ist dabei vom User einstellbar. Als Überprüfung, ob ein Baum platziert werden soll, wird folgender Test durchgeführt: $p > 0$, wobei p hier der Wert von Perlin Noise an der Stelle des Punktes p_b ist.

Die Abb. 4.19 zeigt von Baumagenten erzeugten Verteilungen.

Grasagenten

Die Aufgabe des Grasagenten ist es, Gräser und andere kleine Details auf dem bereits erzeugten Terrain zu platzieren. Im Gegensatz zum Baumagenten platziert er jedoch keine Modelle sonder nur simple Texturen, die durch

[1] https://de.wikipedia.org/wiki/Vektor#Addition_und_Subtraktion



(a) (b)

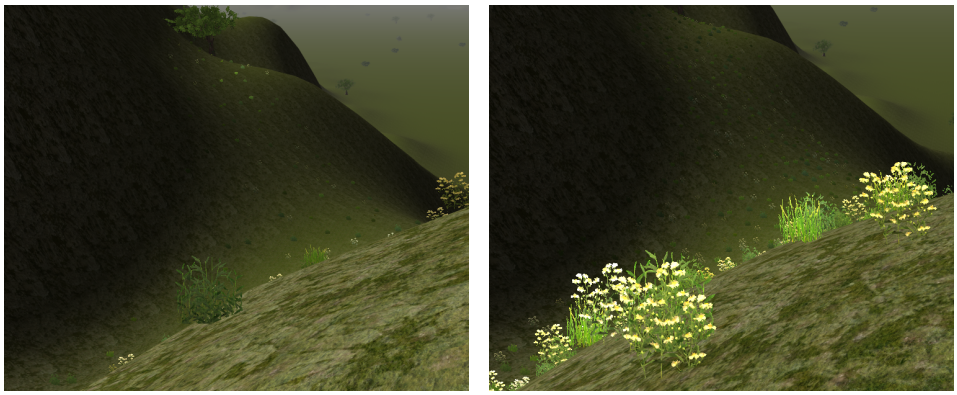
Abbildung 4.19: Unterschiedliche von Baumagenten erzeugte Baumplatzierungen. (a) Baumagenten mit einem Abstandsfaktor $a = 5$ und einer Perlin Noise Frequenz von $f = 500$, (b) Baumagenten mit einem Abstandsfaktor $a = 20$ und einer Perlin Noise Frequenz von $f = 50$.

einen Billboardshader immer in Richtung Kamera schauen. Der Algorithmus, der für das Auswählen der Punkte zuständig ist, an denen Details platziert werden sollen, ist dabei derselbe wie der für die Baumagenten (siehe Abschn. 4.3.3.3). Das konkrete Darstellen der Details auf dem Terrain ist dabei von der verwendeten Engine abhängig. In Unity wird dazu für jeden Detailtyp eine Detailmap angelegt, die jeweils angibt, an welchen Punkten der Landschaft sich Details befinden sollen. Die Detailmap ist genauso aufgebaut wie eine Heightmap. Die Auflösung dieser Detailmap bestimmt dabei, wie nahe erzeugtes Gras beieinander liegt. Je dichter Gras wachsen soll, desto ähnlicher muss die Größe der verwendeten Detailmap zu der Größe sein, mit der das Terrain gerendert wird. In Abb. 4.20 ist jeweils dieselbe Landschaft mit unterschiedlich aufgelösten Detailmaps zu sehen.

4.4 Umsetzung mit Unity

Der in dieser Arbeit vorgestellte Ansatz wurde mit der Game Engine Unity umgesetzt. Unitys Aufgabe ist es, erzeugte Heightmaps als Terrain darzustellen und verwendete Modelle für die Vegetation darauf zu platzieren. Das Erzeugen der Heightmaps ist dabei jedoch unabhängig von Unity. Nachfolgend sind einige Gründe aufgezählt, warum Unity verwendet wurde:

- Unity besitzt einen Terrain Renderer, der Heightmaps rendern kann. Unity übernimmt dabei die Kollisionsbehandlung des Spielers mit der Landschaft.
- Unity macht es einfach, Bäume und Gräser auf dem Terrain zu plat-



(a) (b)

Abbildung 4.20: Unterschied der Gräserhäufigkeit in Bezug auf die Detailmap-Auflösung. Für beide wurde jeweils Perlin Noise mit der Frequenz $f = 500$ sowie eine Heightmap mit 256×256 Werten verwendet. (a) Gerenderte Landschaft mit 2048^2 Einheiten und einer Detailmap mit 256×256 Werten. (b) Gerenderte Landschaft mit 2048^2 Einheiten und einer Detailmap mit 512×512 Werten.

zieren.

- Der Editor von Unity ermöglicht es, für selbst erstellte Objekte eine UI generieren zu lassen. Diese dient dazu, die Parameter bzw. Einstellungen der Objekte zu verändern. In Abb. 4.21 ist ein Ausschnitt der GUI zum Editieren der Agenteneinstellungen zu sehen.
- Allgemeine Vorteile einer Game Engine (Beleuchtung, Schatten, Texturierung).

Das Rendern von endlosen Landschaften mit Unity hat jedoch auch ein paar Tücken. Es ist nicht möglich, neue Terrainsegmente im Hintergrund darzustellen. Dies führt dazu, dass nicht mehrere Segmente gleichzeitig angezeigt werden können. Hierbei handelt es sich jedoch nur um ein kleines Problem, da neue Terrainsegmente auch nach und nach dargestellt werden können.

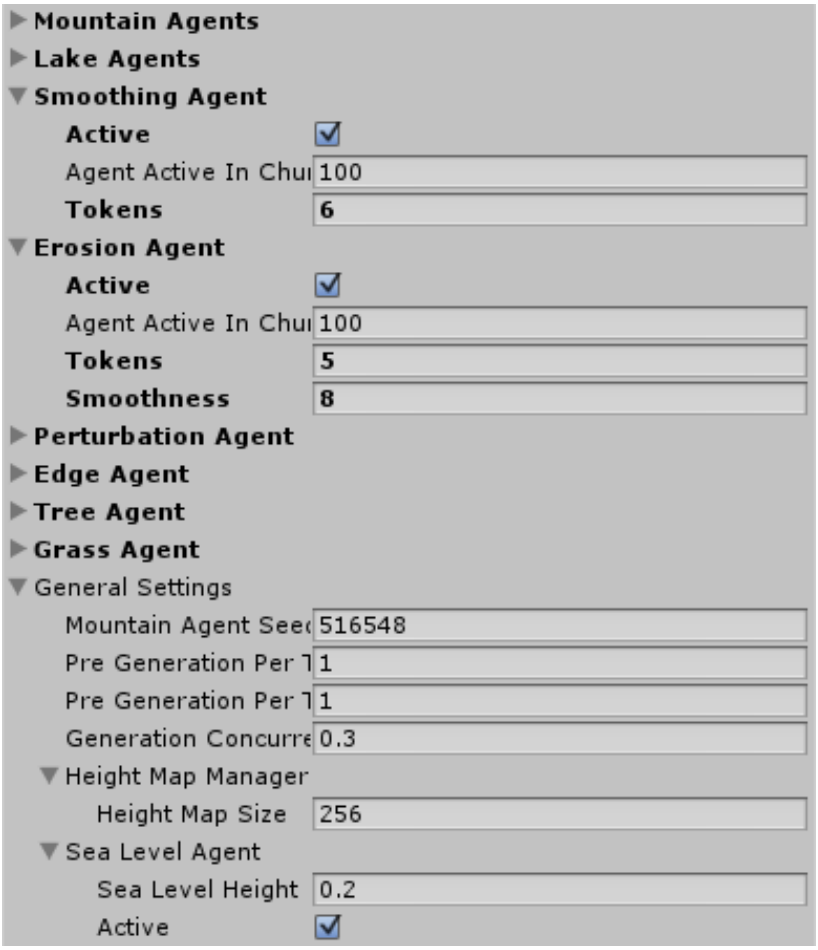


Abbildung 4.21: Ausschnitt der von Unity erzeugten GUI um die Einstellungen der Agenten zu editieren.

Kapitel 5

Evaluierung

In diesem Kapitel wird der vorgestellte Lösungsansatz, basierend auf den Kriterien des Bewertungskataloges (siehe Abschn. 3.1), bewertet.

5.1 Kontrollierbarkeit und nötige Benutzereingaben

Bei der Umsetzung dieses Ansatzes war eines der Ziele, dem Benutzer möglichst viel Kontrolle über den Generierungsprozess zu geben. Jeder der entwickelten Agenten ist deshalb zu einem gewissen Grad vom Benutzer kontrollierbar[1]. Den größten Einfluss auf das erzeugte Ergebnis haben dabei die Landschaftsagenten. Der Benutzer kann zum Beispiel für Bergagenten sehr gut bestimmen, wie die zu generierenden Berge aussehen sollen. Das geschieht über die in Abb. 5.1 dargestellten Benutzeroberfläche. Mit diesen Optionen kann bereits eine große Variation an Bergen erzeugt werden.

In Abb. 5.2 und Abb. 5.3 sind Landschaften zu sehen, die durch verschieden eingestellte Berg- und Seeagenten erzeugt wurden. Diese unterschiedlichen Ergebnisse zeigen, dass das Kriterium der Kontrollierbarkeit zu einem zufriedenstellenden Maß erfüllt ist.

Zusätzlich spielt auch die Anzahl der nötigen Benutzereingaben eine Rolle. In dem vorgestellten Ansatz müssen zu Beginn für alle Agententypen einige Parameter konfiguriert werden. Während der Generierung der Landschaft sind jedoch keine Benutzereingaben mehr nötig. Ein Produkt das diesen Ansatz verwenden würde, könnte voreingestellte Agenten ausliefern. Ein Spieler müsste lediglich einen Seed auswählen um unterschiedliche Spielwelten zu erzeugen.

[1] In welchem Ausmaß das pro Agenten zutrifft, wurde in Abschn. 4.3.3 beschrieben

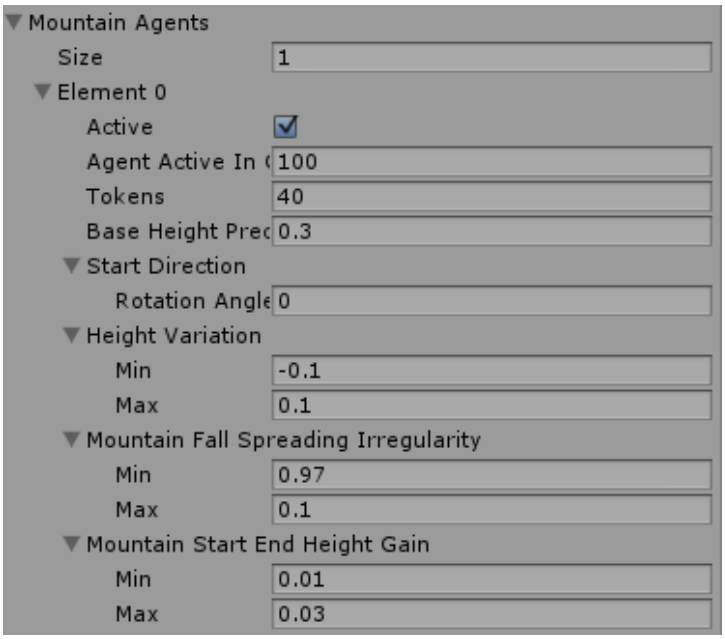
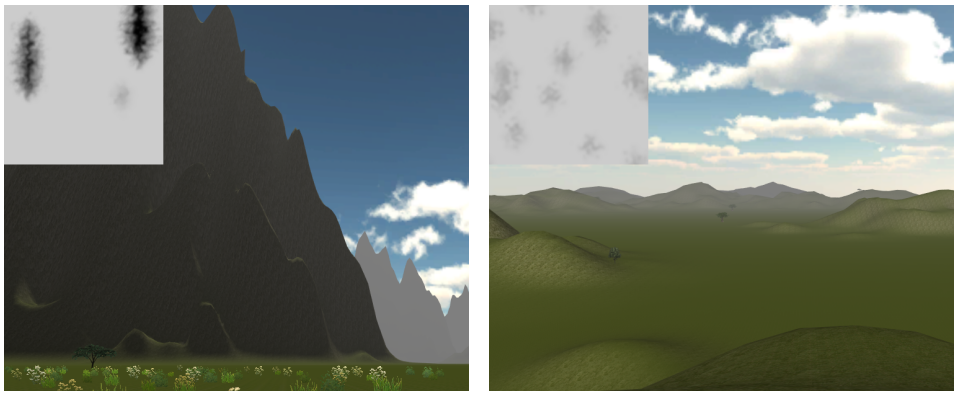


Abbildung 5.1: Übersicht über Einstellungen, die der User für einen Bergagenten vornehmen kann. *Size* bestimmt dabei die Anzahl der verwendeten Agenten, *Tokens* wie lange die Bergkette wird und *Base Height Percent* die allgemeine Grundhöhe des Berges. Die Faktoren *Height Variation* und *Mountain Fall Spreading Irregularity* dienen dazu, Wertebereiche zu definieren, die bestimmt wie stark die Grundhöhe und die seitlichen Ausläufer variiert werden sollen.

5.2 Echtzeitfähigkeit/Performance

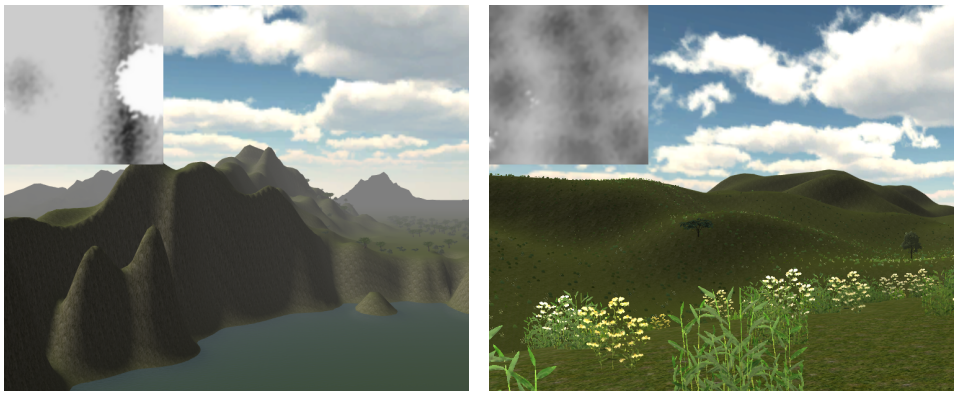
Aufgrund der Tatsache, dass man endloses Terrain nicht auf einmal generieren kann, ergibt sich die Notwendigkeit, dass neue Segmente nach und nach in Echtzeit erzeugt werden müssen. Zu Beginn können ein paar generiert werden um zu verhindern, dass der Spieler das Ende der Landschaft sieht. Die Umsetzung des vorgestellten Lösungsansatzes startet dabei jeweils mit neun initial erzeugten Segmenten. Sollte sich der Spieler über den Rand eines solchen bewegen, werden jeweils drei weitere generiert. Die Platzierung der Segmente hängt dabei von der Richtung ab, in die sich der Spieler gerade bewegt. Diese Vorgehensweise ermöglicht es, scheinbar endloses Terrain zu erzeugen.

Ist das Ziel, das Ganze in Echtzeit zu betreiben, so ist es nötig, dass die Zeit zum Erzeugen der drei neuen Segmente kürzer ist als die Zeit, die ein Spieler braucht um sich über ein ganzes Segment zu bewegen. In Abb. 5.4 ist eine Übersicht zu sehen, wie lange ein Spieler in Relation zu seiner Geschwindigkeit braucht, um verschieden groß gerendertes Terrain



(a) (b)

Abbildung 5.2: Generierte Landschaften mit jeweils unterschiedlichen Einstellungen der Agenten. (a) Einsatz von einem Bergagenten pro Segment mit einer Grundhöhe von $h = 85\,\%$, einem seitlichen Abfall von $0 - 10\,\%$ und einer Höhenvariation von $-1\,\%$ bis $1\,\%$. (b) Einsatz von 10 Bergagenten pro Segment mit einer Grundhöhe von $h = 35\,\%$, einem seitlichen Abfall von $0 - 10\,\%$ und einer Höhenvariation von $-0.5\,\%$ bis $0.5\,\%$.



(a) (b)

Abbildung 5.3: Generierte Landschaften mit jeweils unterschiedlichen Einstellungen der Agenten. (a) Einsatz von 2 Bergagenten pro Segment mit einer Grundhöhe von $h = 80\,\%$, einem seitlichen Abfall von $0 - 5\,\%$ und einer Höhenvariation von $-2\,\%$ bis $2\,\%$. Sowie einem Seeagenten mit einem Radius von $r = 60$ Heightmap Einheiten und einer Tiefe von $0\,\%$ bis $10\,\%$ der allgemeinen Höhe. (b) Einsatz von 15 Bergagenten pro Segment mit einer Grundhöhe von $h = 36\,\%$, einem seitlichen Abfall von $0 - 2\,\%$ und einer Höhenvariation von $-0.2\,\%$ bis $0.2\,\%$.

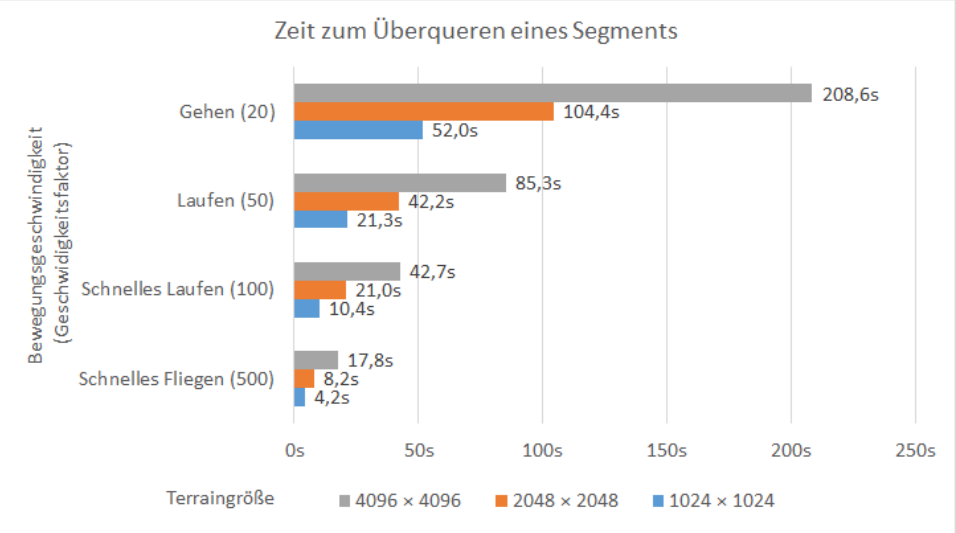


Abbildung 5.4: Übersicht über die Zeit die ein Spieler braucht um unterschiedlich große Segmente zu überqueren. Die Geschwindigkeitskategorien wurden subjektiv vom Autor eingeschätzt, der Wert in der Klammer bezeichnet dabei den in Unity eingestellten Wert für die Geschwindigkeit. Die gemessene Zeit sagt aus, wie lange ein Spieler von der Position $(0/0)$ bis zur Position $(0/s)$ braucht, wobei s immer die verwendete Terraingröße ist.

zu überqueren. Die Umsetzung dieser Arbeit verwendet dazu eine Größe von 2048^2 Einheiten[2]. Im schlimmsten Fall würde das bedeuten, dass drei Segmente in $t = 8.2s$ erzeugt und dargestellt werden müssen. Da sich in den meisten Anwendungen der Spieler jedoch nicht so schnell bewegt, wäre die Zeit $t = 21.0s$ für *Schnelles Laufen* auch ausreichend.

Um die *Echtzeitfähigkeit* nun zu gewährleisten, ist es nötig, dass drei Segmente in unter $t = 21.0s$ erzeugt werden können. In Abb 5.5 ist ein Diagramm zu sehen, das zeigt wie lange der umgesetzte Ansatz braucht, um unterschiedlich große Heightmaps zu erzeugen und darzustellen. Eine kleine Heightmap der Größe 256×256, die mit 2048^2 Einheiten gerendert wird, braucht dabei $673ms$. Für den Fall, dass sich ein Spieler nun sehr schnell bewegt, müssten drei Segmente in $t = 8.2s$ erzeugt werden. Dieser Ansatz benötigt jedoch nur $0.673s \cdot 3 = 2.019$ Sekunden. Dadurch ist die *Echtzeitfähigkeit* sogar in diesem Fall noch gegeben.

Da die Konfiguration der einzelnen Agenten jeweils einen großen Einfluss auf die Performance hat, wurden nachfolgend weitere Performanceanalysen durchgeführt. Zur Durchführung der Analysen dieses Kapitels wurde folgende Hardware benützt:

1. *Grafikkarte*: Nvidia Geforce GTX 770, GDDR5-Speicher 2GB

[2]Die Größe einer Einheit, wird in diesem Fall von Unity bestimmt.

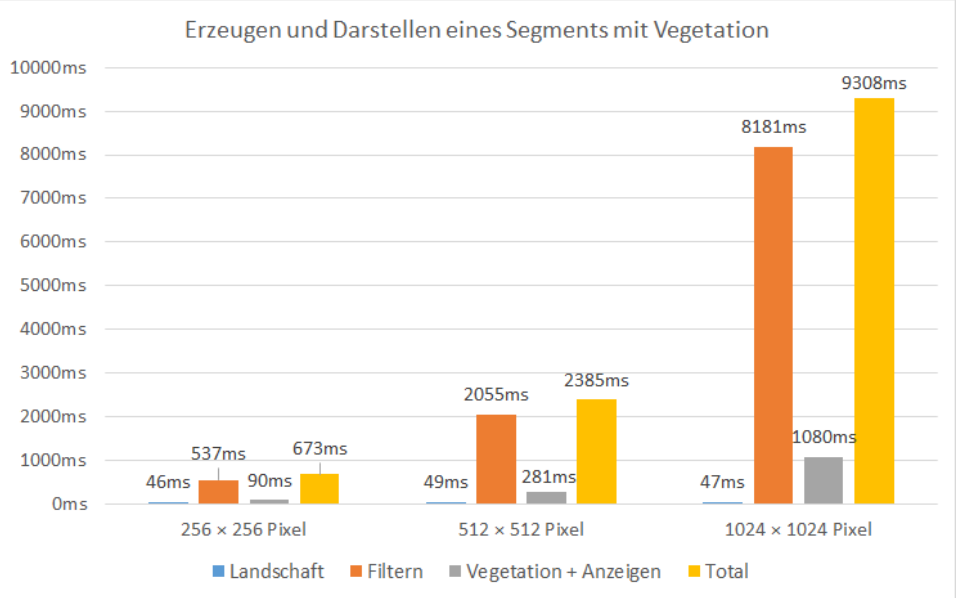


Abbildung 5.5: Vergleich der Zeiten die es braucht, um unterschiedlich große Heightmaps für ein Segment zu erzeugen und mit 2048^2 Einheiten darzustellen. Für die Landschaft wurden 5 Bergagenten mit jeweils 40 Tokens und 2 Seeagenten verwendet. Zum Filtern wurde ein Durchlauf für den Verschiebungsagenten, fünf für den Erosionsagenten und sechs für den Glättungsagenten verwendet. Die Baumagenten verwenden einen Baumabstand von 2 und $f = 100$ für Perlin Noise. Für die Detailmap des Grasagenten wurde eine Auflösung von 256×256px verwendet.

2. *Prozessor*: Intel Core i7-4770K CPU@3.50GHz (8 CPUS)
3. *Arbeitsspeicher*: 16384MB DDR3 RAM
4. *Festplatte*: 512 GB SSD

5.2.1 Landschaftsagenten

Wie in Abb 5.5 ersichtlich, brauchen die Landschaftsagenten nur einen Bruchteil der Zeit der anderen Agenten. Das liegt daran, dass sie auch bei höher aufgelösten Heightmaps nicht mehr Operationen durchführen müssen, in diesen werden Strukturen jedoch kleiner dargestellt. Die Geschwindigkeit der Landschaftsagenten hängt hauptsächlich von der Anzahl der verwendeten Agenten sowie der Tokenanzahl ab. In Abb. 5.6 ist eine Übersicht zu sehen, wie lange Landschaftsagenten mit unterschiedlichen Einstellungen jeweils brauchen.

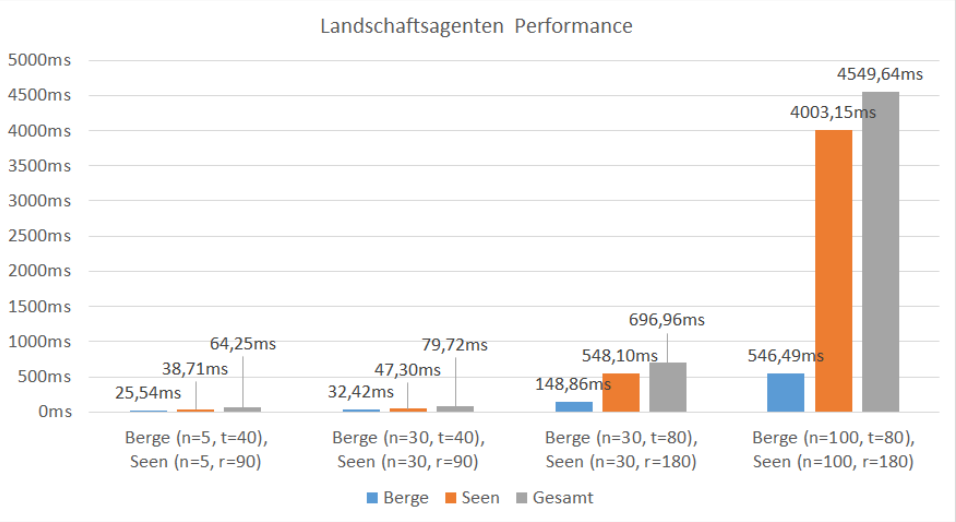


Abbildung 5.6: Performance von unterschiedlich konfigurierten Landschaftsagenten. Die untere Achse beschreibt dabei jeweils wie viele Berg- und Seeagenten verwendet wurden. n ist jeweils die Anzahl der verwendeten Agenten, t die Anzahl der Token und r der Radius der Seeagenten.

5.2.2 Filteragenten

Die Klasse der Filteragenten braucht insgesamt am meisten Zeit. Dies liegt vor allem daran, dass sie jeweils mehrfach eine ganze Heightmap durchlaufen und verändern müssen. Das ist auch der Grund dafür, warum sie bei einer höher aufgelösten Heightmap um einiges länger brauchen. Ein Vorteil dieser Agenten ist jedoch, dass sie komplett unabhängig von den Ergebnissen der Landschaftsagenten sind und in der Regel für eine bestimmte Anzahl an Token immer genau gleich lange brauchen. Dies kann auch in Abb. 5.7 gesehen werden. Hier wurde die Performance der einzelnen Filteragenten jeweils gegenübergestellt. Ein großer Vorteil des Verschiebungsagenten im Vergleich zu den anderen ist, dass er jeweils nur einen Durchlauf benötigt.

5.2.3 Vegetationsagenten

Die Vegetationsagenten sind hauptsächlich dafür da, Modelle auf der erzeugten Landschaft zu platzieren, dieser Prozess hängt stark von der verwendeten Engine ab. In Abb. 5.8 ist sowohl die Performance der Gras- als auch der Baumagenten zu sehen. Wie man sieht ist die Performance der Grasagenten davon abhängig, wie groß die verwendete Detailmap ist. Die Leistung der Baumagenten hängt von der Größe der Heightmap und dem Abstand s, indem die Bäume platziert werden, ab. Wenn für s ein kleiner Wert gewählt wird, dauert es bei doppelter Heightmap Größe bereits viermal so lange, bis

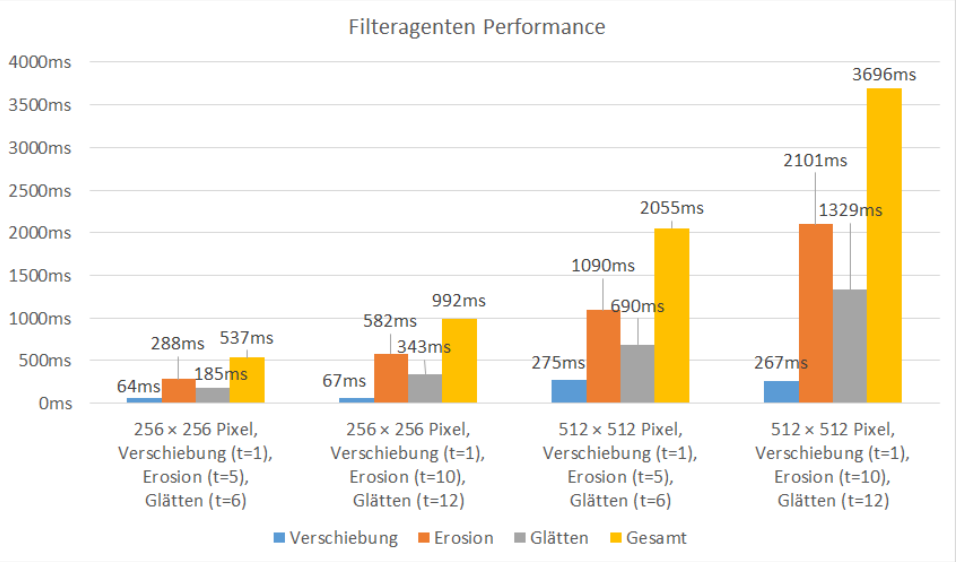
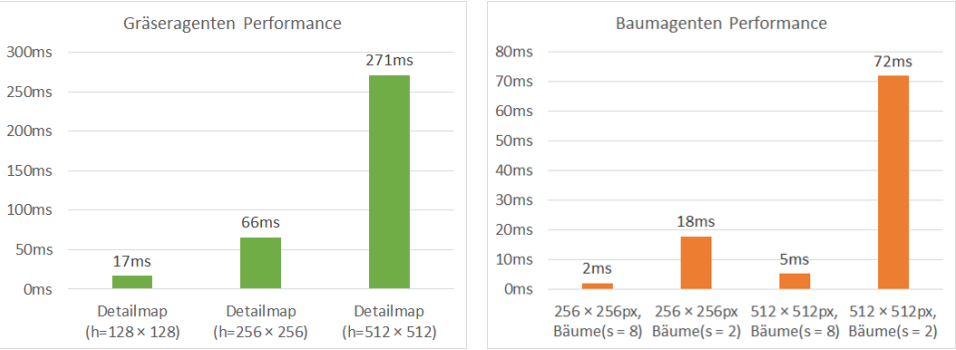


Abbildung 5.7: Gegenüberstellung der Performance der einzelnen Filteragenten. t ist dabei die Anzahl der verwendeten Token für die einzelnen Agenten, die Pixelangabe ist jeweils die Größe der erzeugten Heightmap.



(a) (b)

Abbildung 5.8: Performance der Vegetationsagenten. (a) Performance der Grasagenten in Abhängigkeit von der Größe der erzeugten Detailmaps. (b) Performance der Baumagenten in Abhängigkeit von der Größe der erzeugten Heightmap und des Abstandes s zwischen den Bäumen.

die Bäume platziert wurden. Allgemein gesehen sind beide Agenten jedoch sehr performant, da sie jeweils weit unter einer Sekunde bleiben.

5.3 Endlosigkeit

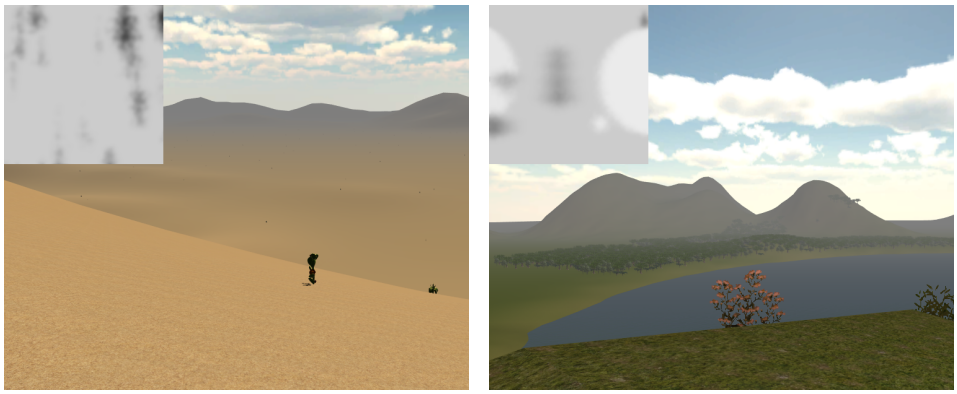
Sobald sich der User am Rand eines Segments befindet, werden jeweils umliegende Segmente generiert. Der Prozess des Erzeugens von Landschaftsteilen kann beliebig oft passieren, was letztendlich die Möglichkeit schafft, endloses Terrain zu erzeugen. Die Bezeichnung "endlos" ist aber in diesem Zusammenhang nicht ganz korrekt, da dem Computer irgendwann der verfügbare Speicher ausgeht und bereits generiertes Terrain nicht mehr abgespeichert werden kann. Ein Nachteil der aktuellen Umsetzung zeigt sich vor allem beim endlosen Terrain, es ist nicht möglich, unterschiedliche Landschaftstypen dabei zu erzeugen. Agenten können jedoch über einen auswählbaren Prozentwert pro Segment deaktiviert werden, dass würde bei 10 % bedeuten, dass ein Agent nur in jedem 10 Segment verwendet wird.

5.4 Qualität

Dieser Abschnitt gibt einen Überblick über die Qualität der erzielten Ergebnisse. Eine objektive Bewertung dieser ist schwierig, da es keine eindeutigen Kriterien für die visuelle Qualität gibt, daher entspricht die Bewertung der Qualität der Meinung des Autors. Diese wird vor allem mittels Screenshots untermauert, damit sich der Leser selbst ein Bild über die Brauchbarkeit der erzeugten Ergebnisse machen kann.

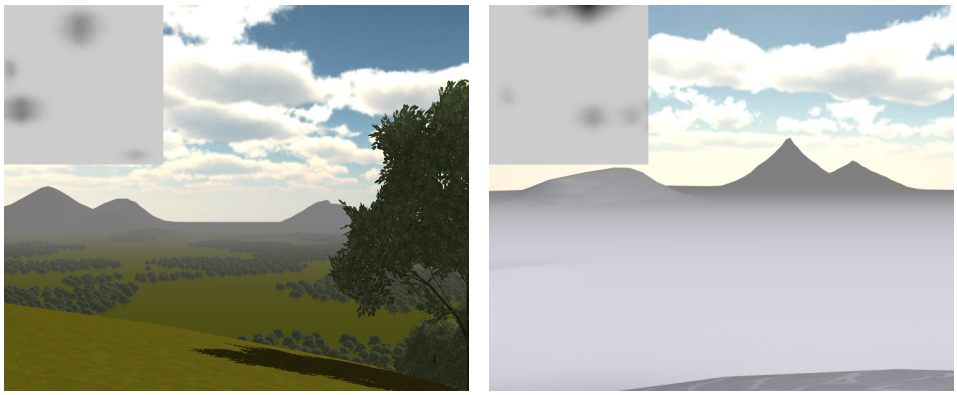
In Abb. 5.9 und Abb. 5.10 sind generierte Landschaften zu sehen, die durch die entwickelten Agenten erzeugt wurden. Diese zeigen unterschiedliche Arten von Landschaften. Wie man in den Abbildungen sieht, können durchaus realitätsnahe Landschaften erzeugt werden. Der Ansatz ist auch flexibel genug um verschiedenste Typen von Terrain zu erzeugen.

Ein Manko der aktuellen Umsetzung ist, dass die Agenten jeweils nur eine Art von Terrain auf einmal generieren können. Es wäre in der aktuellen Umsetzung nicht möglich, die vier gezeigten Landschaftsarten in eine Welt zu packen. Wenn dies das Ziel sein sollte, könnte man den Ansatz noch um Biome erweitern. Dazu müsste man verschiedenste Einstellungen der Agenten speichern und diese abwechselnd verwenden, um einzelne Segmente zu erzeugen. So könnte man endlose Landschaften erzeugen, die sowohl eine Wüsten- als auch eine Graslandschaft beinhalten. Die Übergänge zwischen den Segmentgrenzen der unterschiedlichen Landschaftstypen wären dabei jedoch ein Problem.



(a) (b)

Abbildung 5.9: Generierte Landschaften mit jeweils unterschiedlichen Texturen. (a) Wüstenlandschaft erzeugt durch 20 Bergagenten mit einer niedrigen variierenden Grundhöhe von $h = 30\,\%$, (b) Graslandschaft mit einem großen See, erzeugt durch einen Seeagenten mit einem Radius von $r = 90$.



(a) (b)

Abbildung 5.10: Erzeugte Landschaften, die mit unterschiedlichen Texturen dargestellt wurden. (a) Graslandschaft mit sehr vielen Bäumen, Baumagenten mit einer Dichte $d = 2$ und einer Perlin Noise Frequenz von $f = 100$, (b) Schneelandschaft mit hohen Bergen die mit einer 30-prozentigen Wahrscheinlichkeit auftreten und eine Grundhöhe von $h = 80\,\%$ aufweisen.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wird ein Ansatz zur prozeduralen Generierung von endlosen Landschaften vorgestellt. Als Ausgangsbasis für die Umsetzung dieser Idee dienen die softwarebasierte Agenten (s. Abschn. 3.6) von Doran und Parberry. Aufbauend auf diesen wurde ein Lösungsansatz entwickelt, der es ermöglicht, effizient und kontrolliert endloses Terrain zu erzeugen. Das zentrale Konzept dahinter ist die Idee der Segmentierung. Basierend auf dieser wurden verschiedenste Agenten entwickelt, die es ermöglichen, Segmente einer Landschaft nacheinander zu erzeugen.

Um dem Leser einen Überblick über die Qualität dieses Ansatzes zu geben, wurde zusätzlich eine Evaluierung durchgeführt. Diese beinhaltet sowohl eine Analyse der Performance als auch der Qualität der erzeugten Landschaften.

6.2 Mögliche Verbesserungen und Erweiterungen

Als Fortführung dieser Arbeit könnten noch zusätzliche Typen von Agenten entwickelt werden. Dies gilt vor allem für die Landschaftsagenten. Solche Agenten könnten eine Landschaft zum Beispiel noch um Features wie Flüsse oder Meere erweitern. Für bestimmte Arten von Terrain könnten auch spezielle Agenten entwickelt werden, zum Beispiel ein Dünen Agent für das Generieren von Wüstenlandschaften.

Eine andere Erweiterung könnte beinhalten, dass eine Vielfalt von Landschaftstypen gemeinsam in einer Welt generiert werden. Aktuell ist es zum Beispiel nur möglich eine Wüsten- oder Graslandschaft separat zu erzeugen, dadurch wirken die Landschaften über kurz oder lang eintönig. Würde

man zusätzlich ein Biom-System[1] verwenden, könnte dieses Problem gelöst werden. Es müsste allerdings eine Lösung gefunden werden, um ansehnliche Übergänge zwischen verschiedenen Biomen generieren zu können.

6.3 Fazit

Der vorgestellte Ansatz bietet eine Alternative zu herkömmlichen Methoden, um endlose Landschaften zu erzeugen. Im Vergleich zu diesen ermöglicht er ein höheres Maß an Kontrolle über den Generierungsprozess und ist zusätzlich zur Nutzung in Echtzeitanwendungen geeignet. Ein Nachteil im Vergleich zu bestehenden Methoden ist, dass er etwas langsamer ist als diese.

[1] Unter einem Biom ist der Typ einer Landschaft zu verstehen (http://de.wikipedia.org/wiki/Biom).

Anhang A

Inhalt der DVD

A.1 PDF-Dateien

Pfad:

/

/Eibensteiner_David_2015.pdf Masterarbeit (Gesamtdokument)

A.2 Literatur

Pfad:

/Literatur

*.pdf . . . . . Kopien der verwendeten Literatur. PDFs sind immer nach dem Namen der Arbeiten benannt.

A.3 Literatur-Online

Pfad:

/Literatur-Online

Noise/*.html . . . . . Kopien der Online-Quellen zu den Zufallszahlen basierten Methoden.

Terrain/*.html . . . . . Kopien der Online-Quellen zur Terrain Generierung.

Sonstige/*.html . . . . Kopien von nicht kategorisierten Online-Quellen.

A.4 Projekt

Pfad:

/Projekt

readme.txt	Textdatei mit Anleitungen zur Installation und Verwendung des Projekts.
SystemDoc.pdf	Dokument das die Systemarchitektur des Projektes erklärt.
UserDoc.pdf	Dokument das die Einstellungsmöglichkeiten der Agenten erklärt.
Source Code/	Enthält den Source Code des Unity-Projektes.
Executables/	Enthält mehrere ausführbare Beispiele von vordefiniertem Terrain.
Screenshots/	Enthält mehrere Bilder von erzeugtem Terrain.

A.5 Abbildungen

Pfad:

/Abbildungen

*.pdf	Vektorgrafiken
*.png	Screenshots und Grafiken

Quellenverzeichnis

Literatur

[1] Travis Archer. *Procedurally Generating Terrain.* Techn. Ber. Morningside College, Iowa, 2011 (siehe S. 11, 14–17).

[2] Daniel Michelon De Carli u. a. „A survey of procedural content generation techniques suitable to game development". In: *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*. 2011, S. 26–35 (siehe S. 2, 5, 7, 8).

[3] Jonathon Doran und Ian Parberry. „Controlled procedural terrain generation using software agents". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2 (2010), S. 111–119 (siehe S. 7, 26–33, 36).

[4] David S Ebert u. a. *Texturing and modeling: a procedural approach.* 3. Aufl. Morgan Kaufmann Verlag, 2002, S. 1–721 (siehe S. 5).

[5] Mark Hendrikx u. a. „Procedural content generation for games". In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 9.1 (2013), S. 1–22. URL: http://dl.acm.org/citation.cfm?doid=2422956.2422957 (siehe S. 2, 4).

[6] Houssam Hnaidi u. a. „Feature based terrain generation using diffusion equation". In: *Computer Graphics Forum* 29.7 (2010), S. 2179–2186. URL: http://doi.wiley.com/10.1111/j.1467-8659.2010.01806.x (siehe S. 7).

[7] George Kelly und Hugh McCabe. „A survey of procedural techniques for city generation". In: *ITB Journal* (2006), S. 87–130 (siehe S. 5).

[8] Jacob Olsen. *Realtime procedural terrain generation.* Techn. Ber. University of Southern Denmark, 2004, S. 20 (siehe S. 7, 48, 50).

[9] Adrien Peytavie u. a. „Arches: A framework for modeling complex terrains". In: *Computer Graphics Forum* 28.2 (2009), S. 457–467 (siehe S. 7).

[10] William L. Raffe, Fabio Zambetta und Xiaodong Li. „A Survey of Procedural Terrain Generation Techniques using Evolutionary Algorithms“. In: *IEEE Congress on Evolutionary Computation* (2012), S. 1–8 (siehe S. 22).

[11] Stuart Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3. Aufl. Prentice Hall, 2009 (siehe S. 26).

[12] Ryan L. Saunders. „Realistic Terrain Synthesis using Genetic Algorithms“. Diss. Texas A&M University, 2006 (siehe S. 23, 24).

[13] Ruben M. Smelik u. a. „A survey of procedural methods for terrain modelling“. In: *3AMIGAS - 3D Advanced Media In Gaming And Simulation* (2009), S. 25–34 (siehe S. 6, 7).

[14] Soon Tee Teoh. „River and coastal action in automatic terrain generation“. In: *CGVR - Computer Graphics and Virtual Reality*. 2008, S. 3–9 (siehe S. 8).

[15] Paul Walsh und Prasad Gade. „Terrain generation using an Interactive Genetic Algorithm“. In: *IEEE Congress on Evolutionary Computation* (2010), S. 1–7. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5585913 (siehe S. 24–26).

[16] Yu Xinjie Yu Xinjie. „Introduction to evolutionary algorithms“. In: *40th International Conference on Computers and Industrial Engineering (CIE)*. 2010, S. 1 (siehe S. 22).

[17] Huafei Yin und Changwen Zheng. „A Parametrically Controlled Terrain Generation Method“. In: *7th International Conference on Computer Science & Education*. 2012, S. 775–778 (siehe S. 18–21).

Online-Quellen

[18] Travis Archer. *Midpoint Displacement Algorithmus*. URL: https://code.google.com/p/fractalterraingeneration/wiki/Midpoint%5C_Displacement (siehe S. 12).

[19] Travis Archer. *Perlin Noise Algorithm*. URL: https://code.google.com/p/fractalterraingeneration/wiki/Perlin%5C_Noise (besucht am 19. 04. 2015) (siehe S. 15).

[20] Daniel Beard. *Diamond Square Terrain*. URL: https://danielbeard.wordpress.com/2010/08/07/terrain-generation-and-smoothing/ (besucht am 21. 04. 2015) (siehe S. 13).

[21] Simon Davis. *Terrain Segmentation*. 2012. URL: http://www.shrimpcave.com/2012/05/opengl-terrain-colouring-fog-and-more.html (siehe S. 37).

[22] Digitalerr0r. *Perlin Noise*. URL: https://digitalerr0r.wordpress.com/2011/05/25/procedural-landscapes-using-perlin-noise/ (besucht am 19. 04. 2015) (siehe S. 15, 16).

[23] Minecraft-Wiki. *Minecraft Biomes Wiki*. URL: http://minecraft.gamepedia.com/Biome (siehe S. 5).

[24] Darko Stosic. *Terrain Heightmap Mapping*. 2010. URL: http://ddstosic.50webs.com/dwarf.htm (besucht am 19. 03. 2015) (siehe S. 9).

[25] Archer Travis. *Diamond Square Algorithmus*. URL: https://code.google.com/p/fractalterraingeneration/wiki/Diamond%5C_Square (siehe S. 13).

[26] User A3r0. *Image of a heightmap*. URL: http://en.wikipedia.org/wiki/Heightmap (siehe S. 6).

[27] PG-Wiki. *Procedural Generation Wiki*. URL: https://en.wikipedia.org/wiki/Procedural%5C_generation (siehe S. 4).

[28] Wikipedia. *.kkrieger*. URL: http://en.wikipedia.org/wiki/.kkrieger (siehe S. 5).