Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Katharina Mulack

Parametrisierbare Generierung einer Dynamischen Spielewelt

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Katharina Mulack

Parametrisierbare Generierung einer Dynamischen Spielewelt

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Kai von Luck

Eingereicht am: 17. November 2016

Thema der Arbeit

Parametrisierbare Generierung einer Dynamischen Spielewelt

Stichworte

Dynamische Spielewelt, Prozedurale Generierung, Labyrinth, Landschaft, Unity Engine, EmotionBike, Baukastensystem, DSL

Zusammenfassung

In heutigen Computerspielen kommt den Interaktionen, wie beispielsweise Emotionen, des Spielers zunehmend mehr Bedeutung zu, konsequenterweise sollten zukünftig nicht mehr nur Spiele den Spieler beeinflussen können, sondern auch umgekehrt. Diese Arbeit versucht einen derartigen Ansatz exemplarisch zu realisieren, wobei der Fokus auf prozeduraler Synthese zur dynamischen Generierung von Spielewelten liegt. Hierzu werden in einer Game Engine zwei verschiedene Szenen prozedural generiert und mit einer DSL klassifiziert. Hierbei handelt es sich jeweils um eine Labyrinth- und Landschaftsszene. Im Rahmen der Implementierung werden verschiedene Algorithmen zur Geländeerzeugung, Wegfindung und Labyrinthbildung analysiert und angewendet. Zusätzlich wird ein Baukastensystem vorgestellt, welches diese Szenen anhand von Spielerinteraktionen kombinieren kann.

Title of the paper

Parametrizable generation of a dynamic game-world

Keywords

dynamic game-world, procedural generation, maze, landscape, Unity engine, EmotionBike, combination system, DSL

Abstract

Player interactions like emotions become more and more important in computer games these days. Consistently it should be possible that players can affect games as well as being affected by them. This thesis tries to implement such an approach with focus on procedural synthesis for dynamic generation of game-worlds. Two scenes will be procedurally generated for this purpose and classified by a DSL. There will be a landscape and a maze scene. As part of the implementation different algorithms will be analyzed and used. Additionally a combination system for joining scenes based on player interactions will be introduced.

Inhaltsverzeichnis

Listings

1. Einleitung

Computerspiele haben in den letzten Jahrzehnten einen drastischen Wandel durchlaufen. Nicht nur Grafik und Spielgeschehen sind über die Jahre zunehmend komplexer geworden, auch der Stellenwert von Spielerinteraktionen, wie zum Beispiel Emotionen, ist stetig gestiegen. In jüngster Zeit eröffnen sich dieser Entwicklung gänzliche neue Möglichkeiten ausgehend von technischen Innovationen, wie beispielsweise das Erschaffen virtueller Realitäten. Im Rahmen umfassender Immersion ist es möglich beim Spieler vielfältige Emotionen hervorzurufen. Komplementär hierzu macht auch das Feld der Emotionsanalyse große Fortschritte, so zum Beispiel in der Automobilindustrie in Form von Fahrerassistenzsystemen. Obwohl beide Bereiche für sich genommen durchaus etabliert sind, gibt es keine prominenten Ansätze diese zu kombinieren. Während also Spiele inzwischen die Möglichkeit haben, den Spieler zu beeinflussen, ist eine Rückkopplung der Interaktionen des Spielers auf das Spielgeschehen derzeit nicht vorgesehen.
Ein Ansatz dies zu ändern sind sogenannte Companion Systeme. Die Universität Ulm, die Otto-von-Guericke Universität Magdeburg und das Leibniz-Institut für Neurobiologie setzen sich seit 2009 mit den Eigenschaften und der Realisierung solcher Systeme auseinander [UMN09].

> Das Forschungsvorhaben folgt der Vision, dass technische Systeme der Zukunft Companion-Systeme sind – kognitive technische Systeme, die ihre Funktionalität vollkommen individuell auf den jeweiligen Nutzer abstimmen: Sie orientieren sich an seinen Fähigkeiten, Vorlieben, Anforderungen und aktuellen Bedürfnissen und stellen sich auf seine Situation und emotionale Befindlichkeit ein [UMN09].

Ein Schritt dahin, ein System, in diesem Fall das Spielgeschehen, zu beeinflussen ist, die Spielewelt hochdynamisch an die Interaktionen des Spielers anzupassen. Diese Arbeit versucht einen derartigen Ansatz exemplarisch zu realisieren, wobei der Fokus auf prozeduraler Synthese zur dynamischen Generierung von Landschaften mit geringen Latenzzeiten liegt. Eine Herausforderung im Rahmen dieser Entwicklung wird die Verbindung einzelner Landschaftssegmente (im Folgenden: Szenen) darstellen. Dieses Problem wird durch ein einfaches und flexibles Baukastensystem adressiert.

1.1. Zielsetzung

Ziel dieser Arbeit ist es, wie zuvor dargelegt, die Möglichkeiten zur prozeduralen Generierung einer hochdynamischen Spielewelt anhand von Spieleremotionen aufzuzeigen. Hierzu sollen in Hinblick auf das EmotionBike der HAW Landschaften erzeugt werden, deren konkrete Gestalt parametrisierbar von Nutzeremotionen abhängen kann. Entsprechend werden Szenen in einer Game Engine erstellt, welche per Fahrrad passierbar sind. Neben einer Geländeszenerie, die in vielen Parametern einstellbar ist, wird außerdem ein Labyrinth implementiert, sodass mit wenigen Szenen abwechslungsreiche Eindrücke und herausfordernde Level erzeugt werden können. Da eine realistische Levelgestaltung Aspekte wie Geländeerzeugung, Wegfindung und Vegetation umfasst, ist es notwendig, hierfür geeignete Algorithmen zu ermitteln. Des Weiteren wird ein leicht verständliches Baukastensystem entwickelt um Szenen zu kombinieren, wobei diesem Verfahren eine grobe Klassifikation nach Emotionen (in Form einer DSL) zugrunde gelegt wird.

1.2. Aufbau der Arbeit

Die nachfolgende Arbeit ist inhaltlich in sechs weitere Kapitel unterteilt. Zunächst werden die Einsatzumgebung für die Arbeit und unterschiedliche Techniken zur Realisierung im Kapitel 2 “Grundlagen” beschrieben. Das nachfolgende Kapitel 3 “Analyse” beschreibt, welche Anforderungen an die Arbeit gestellt werden und klärt, ob es bereits Arbeiten gibt, die sich mit diesem Thema beschäftigen. Die Kapitel 4 “Konzept” und 5 “Implementierung” beschäftigen sich mit der geplanten und tatsächlichen Realisierung des Projektes. Abschließend erfolgt in Kapitel 6 die Evaluation der Arbeit hinsichtlich der Fragestellung, ob das Ergebnis den Anforderungen entspricht, wo die Stärken und Schwächen der Implementierung liegen und wie der technische Ausblick des Projektes aussieht. Zuletzt wird das Erreichte zusammengefasst und Perspektiven für zukünftige Arbeiten aufgezeigt.

2. Grundlagen

Das Kapitel Grundlagen befasst sich mit der Einsatzumgebung des Projektes und bietet einen Überblick über Methoden und Techniken um Bestandteile von Szenen, wie beispielsweise Wege, zu generieren. Zusätzlich werden grundlegende Begriffe geklärt.

2.1. Einsatzumgebung

Die entwickelte Software soll im EmotionBike Projekt des Living Place der HAW Hamburg eingesetzt werden.

2.1.1. Living Place Hamburg

Beim Living Place handelt es sich um eine Smart Home Umgebung an der Hochschule für Angewandte Wissenschaften Hamburg, welche seit Januar 2009 stetig weiterentwickelt wird. Es befindet sich in einem Loft mit voll ausgestatteten Ess-, Wohn-, Koch- und Schlafbereichen sowie einem Badezimmer. In dieser Umgebung ist es möglich, Experimente unter nahezu realen Bedingungen durchzuführen. Dabei stehen vor allem die Analyse der Beziehung zwischen Bewohner und Smart Home sowie die Balance zwischen Automatisierung und Nutzerinteraktion im Vordergrund. Beispiele für angewendete Technologien sind Multi-Touch und Sprach- und Bewegungserkennung [HAW].
Um die Analyse zu gewährleisten ist die gesamte Wohnung mit Kameras, Mikrofonen, diversen Sensoren und anderen modernen Überwachungstechnologien ausgestattet, welche in einem separaten Kontrollraum ausgewertet werden. Um eine möglichst vielschichtige und realistische Umgebung zu schaffen arbeiten an diesem Labor Studierende und Mitarbeiter verschiedener Bereiche, wie zum Beispiel Architektur, Lichtdesign oder Interaction Design [HAW].

2.1.2. EmotionBike

Das EmotionBike Projekt der HAW ist ein Forschungsprojekt aufbauend auf einem verbesserten Fahrradergometer. Der Nutzer des Ergometers fährt auf diesem durch eine virtuelle Welt und interagiert mit dieser über die Tretbewegung beziehungsweise -geschwindigkeit und die

Bewegungen des Lenkers innerhalb eines Winkelbereichs von 180 Grad. Der Tretwiderstand kann softwaregestützt bis zu 1000 Watt angepasst werden. Dabei wirkt sich die dargestellte Szene aktiv auf das Verhalten des Ergometers aus. Zu der Installation gehören auch ein vor dem Ergometer installierter 42 Zoll Bildschirm, eine Lampe um das Gesicht des Fahrenden zu beleuchten und für die Analyse der Mimik eine Kinect[Mic] welche auf den Fahrenden gerichtet ist [Bie16][Mül+].

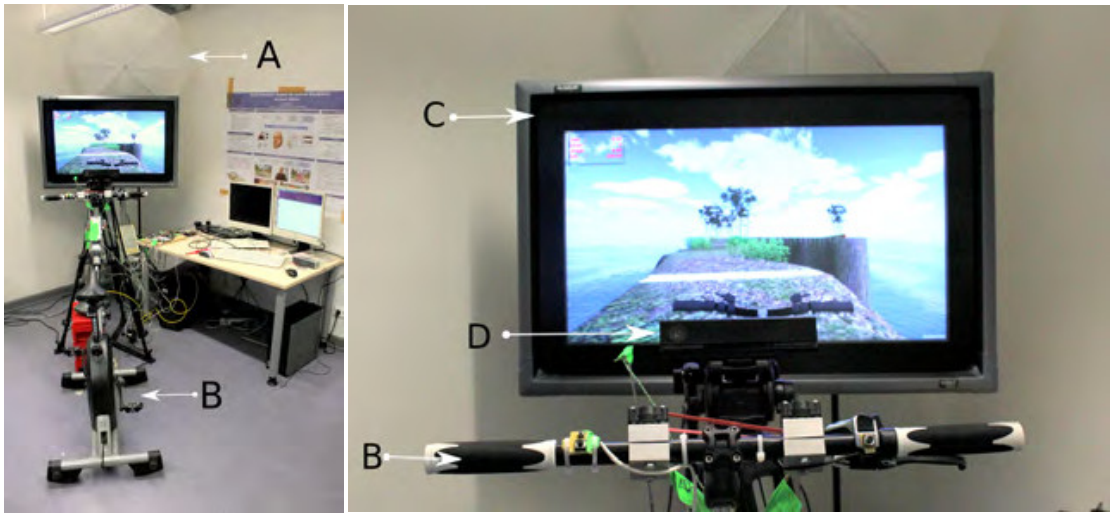


Abbildung 2.1.: Aufbau des EmotionBike Projektes. A: Lampe um das Gesicht des Fahrers zu beleuchten, B: Ergometer, C: Bildschirm, D: Kinect [Mül+]

In Abbildung 2.1 ist der Aufbau des EmotionBike mit seinen Komponenten zu sehen. Es ist anzumerken, dass kein realistisches Fahrrad-Fahrerlebnis mit dem Ergometer erzeugt werden soll.
Ziel des Projektes ist es, die Emotionen des Fahrenden anhand von Mimik und physiologischem Zustand (wie Atmung und Puls) zu erfassen und entsprechend auf den Nutzer und das nachfolgende Level beziehungsweise die nachfolgende Szene einzuwirken. Nachfolgend ein Beispiel zur Erläuterung: Der Nutzer fährt durch eine sehr bergige Landschaft und muss dementsprechend stärker in die Pedale treten. Dies macht sich beim Fahrenden bemerkbar, er ist sichtlich erschöpft. Das System erkennt dies und soll als nächstes ein entspannteres Level auswählen [Bie16][Mül+].
Derzeit ist die Auswahl der Level statisch. Initial wurden fünf verschiedene Level generiert, welche durch bestimmte Ereignisse gezielte Emotionen auslösen sollen.

2.2. Prozedurale Generierung

Bei prozeduraler Generierung, auch prozedurale Synthese genannt, handelt es sich um das automatische Erzeugen von Inhalten mittels parametrisierbarer Algorithmen. Diese Generierung lässt sich wahlweise mit sogenannten Seeds kombinieren, welche es ermöglichen die Ergebnisse zu reproduzieren. Wird etwas mit dem selben Seed mehrfach generiert wird stets das selbe Ergebnis erzeugt. Anwendung findet die prozedurale Generierung beispielsweise bei der Erzeugung von Landschaften, Städten oder auch Texturen. Es wird unterschieden zwischen Echtzeit- und Vorberechnung von Daten. Ein Vorteil von prozeduraler Generierung ist das Einsparen von Speicherplatz. Dies kommt daher, dass nicht die generierte Welt gespeichert werden muss, sondern lediglich die Parameter um diese zu erzeugen [Hön12][Eib15].

2.3. Szenengenerierung

Im Folgenden wird beschrieben, wie Komponenten von Szenen generiert werden können. Dabei handelt es sich um das Gelände, die Wege, die Vegetation und Labyrinthe. Nachfolgend werden folgende Begriffe geprägt: Eine Szene ist abstrakt ein bestimmter Typ Level. Aus einer Szene können durch geeignete Parameter diverse Level generiert werden.

2.3.1. Gelände

Um Landschaften zu generieren, sollte das Gelände gewisse Höhenunterschiede aufweisen. Dabei gibt es, je nach Spieleengine, die Option das Gelände von Hand oder prozedural zu generieren. Bei einer prozeduralen Generierung des Geländes muss die Heightmap ebenfalls prozedural generiert werden. In einer Heightmap wird zu jedem Punkt des Terrains die Höhe gespeichert. Um dies zu tun werden fraktale Algorithmen wie Noise Funktionen, also Rauschsignale, genutzt. Nachfolgend werden vier davon beschrieben und analysiert. Zunächst wird aber erklärt, warum Fraktale generiert werden sollen: Fraktale sind komplexe geometrische Objekte welche in der Natur sehr oft vorkommen. So sind beispielsweise Berge, Wolken, Bäume, Pflanzen und sogar Tiere fraktal [Ben12]. Daher ist es für ein realistisches Gelände von Bedeutung Fraktale aufzuweisen.

Zufallszahlen

Das einfachste Vorgehen wäre, die Heightmap ohne jegliche Noise Funktionen zu generieren. Dazu werden Zufallszahlen für alle Punkte des Geländes generiert und als Werte in der

Heightmap gesetzt. Im Gegensatz zu allen nachfolgend genannten Methoden lässt sich die Generierung mittels Zufallszahlen nicht parametrisieren.

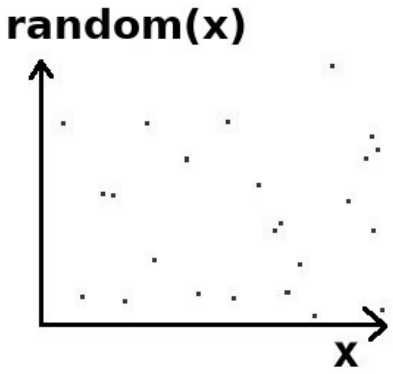


Abbildung 2.2.: Graph mit Zufallswerten [Pig14][Knö14]

In Abbildung 2.2 ist zu sehen, wie ein Graph aussieht, der mit Zufallswerten generiert wurde.

Perlin Noise

Die erste Noise Funktion ist das Perlin Noise. Der Algorithmus eignet sich gut zur Generierung von Wolken, Landschaften und unter Umständen auch Wasser und Feuer [Wik12]. Das originale Perlin Noise wurde 1983 von Ken Perlin entwickelt und Juli 1985 in dem Siggraph Paper "An Image Synthesizer" veröffentlicht [Per85]. Schließlich wurde es 2001 verbessert. Der Algorithmus erzeugt das Rauschen durch die Überlagerung einer Summe von kohärentem Rauschen mit stetig steigenden Frequenzen und fallenden Amplituden. Bei niedrigen Frequenzen wird zwischen weit voneinander entfernten Werten interpoliert. Bei hohen Frequenzen wird für jeden dargestellten Wert, zum Beispiel einen Pixel, ein anderer Wert aus Zufallszahlen für eine Position berechnet. Diese Angabe kann beliebig dimensioniert sein. Beispielsweise reichen für das Generieren einer Heightmap zwei Dimensionen, für die Generierung einer volumetrischen Textur werden drei benötigt. Nachfolgend wird beschrieben, wie auf den Algorithmus Einfluss genommen werden kann [Bev07][Wik12][Bia14].

- **Oktaven:** Die Anzahl der Oktaven bestimmt den Detailgrad des Rauschens. Je mehr Oktaven verwendet werden desto höher ist das Detail aber desto höher ist auch der Rechenaufwand [Bev07][Wik12].

- **Frequenz:** Hiermit kann die Frequenz der ersten Oktave gesetzt werden. Im Normalfall hat jede folgende Oktave die doppelte Frequenz der vorangegangenen Oktave [Bev07][Wik12].
- **Persistenz:** Durch die Persistenz wird angegeben, wie stark der Einfluss hoher Frequenzen auf das Ergebnis ist, und damit die Rauheit des Perlin Noise bestimmt. Je höher dieser Wert ist, desto größer die Rauheit. Dies hat Einfluss darauf, wie schnell die Amplituden für aufeinanderfolgende Oktaven verringert werden. Die Amplitude der ersten Oktave ist eins und die Amplituden von allen nachfolgenden Oktaven sind das Produkt der Amplitude der vorherigen Oktave und dem Wert der Persistenz [Bev07][Wik12].
- **Wertebereich:** Der Wertebereich bestimmt in welchem Bereich das Ergebnis liegt [Wik12].

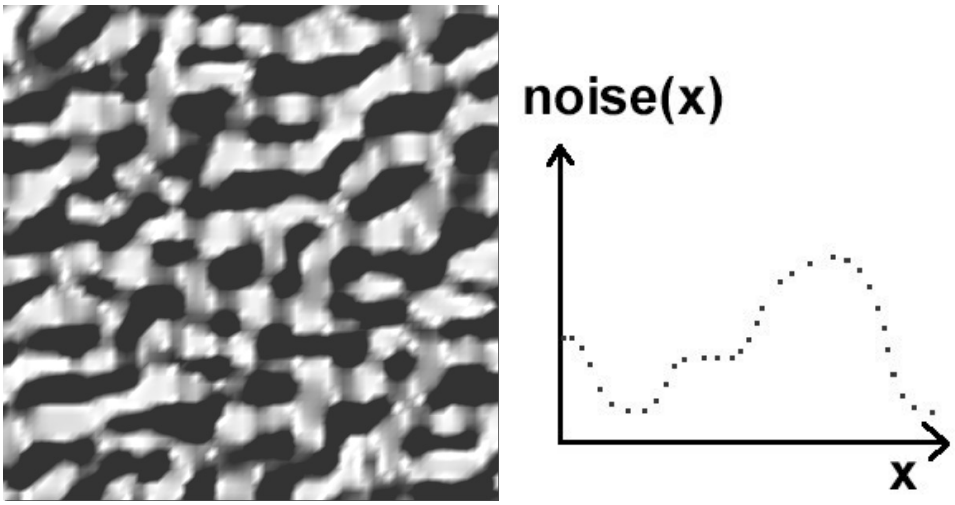


Abbildung 2.3.: Mit Perlin Noise generierter Graph und generierte Heightmap [Pig14][Knö14]

Die Abbildungen 2.3 zeigen beispielhaft einen Graphen mit den generierten Punkten und eine Heightmap die mit Perlin Noise generiert wurde. Die beiden Bilder dienen nur der Anschauung und sind unabhängig von einander. Das verlaufsartige Rauschen entsteht dadurch, dass von einem beliebigen Punkt ein zufälliger Verlauf zum nächsten Eckpunkt erzeugt und mit Spline-Interpolation behandelt wird [Knö14].

Simplex Noise

Wie oben bereits erwähnt, hat Ken Perlin 2001 das Perlin Noise überarbeitet. Das Ergebnis davon ist die folgende Noise Funktion, das Simplex Noise. Dies hat gegenüber Perlin Noise einige Vorteile: Zum einen erfolgt das Generieren von Heightmaps schneller (etwa 27%). Zum anderen wird das Gelände nicht in Quadraten berechnet sondern in Simplexen, wodurch das generierte Gelände natürlicher wirkt. Simplexe sind die n-dimensionale Verallgemeinerung des Tetraeders zwischen denen interpoliert wird [Ull13]. Als letztes ist anzuführen, dass bei Simplex Noise weniger Artefakte auftreten als bei Perlin Noise [mik12].
Abbildung 2.4 zeigt den Unterschied zwischen den beiden Funktionen deutlich [Mah].

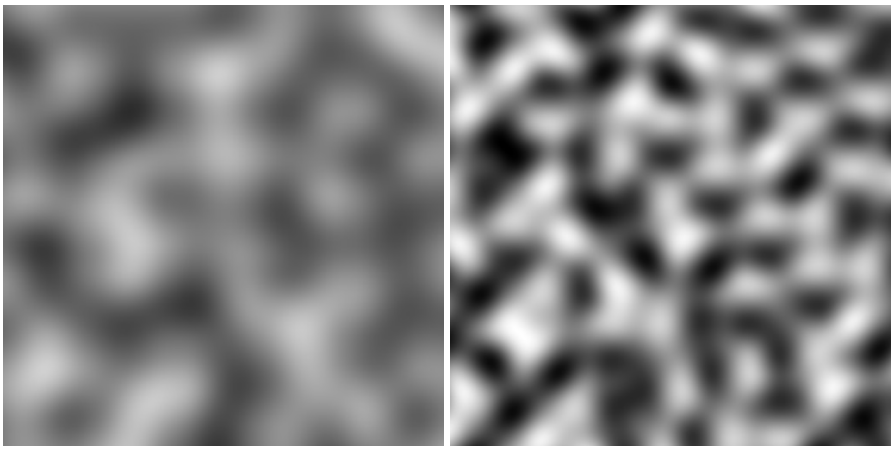


Abbildung 2.4.: Mit Perlin Noise (links) und Simplex Noise (rechts) generierte Heightmaps zum Vergleich [Bèg]

Fractional Brownian Motion

Bei Fractional Brownian Motion (kurz fBm) handelt es sich um einen fraktalen Algorithmus. Durch den fraktalen Grundcharakter des Algorithmus kann die generierte Landschaft realistischer wirken. Bei fBm handelt es sich um eine Klasse von zentrierten Gauß-Prozessen. Der Algorithmus addiert mehrere Schichten unterschiedlicher Noise Funktionen mit verschiedenen Frequenzen. Dabei lässt sich definieren, wie groß der Abstand zwischen den einzelnen Frequenzen sein soll (kurz G), die Größe des fraktalen Inkrements (kurz H) und die Anzahl der Oktaven (kurz O), die, wie beim Perlin Noise, ausmachen, wie oft die Addition ausgeführt wird. Das fraktale Inkrement stellt in der Regel einen Wert zwischen 0.0 und 1.0 dar und beeinflusst, wie glatt beziehungsweise kantig das Gelände ist. Ist das fraktale Inkrement bei 0.0, ist das Ergebnis

ähnlich zu weißen Rauschen und bei 1.0 wäre das generierte Gelände sehr glatt. Nachfolgendes Beispiel illustriert diese Wirkung [Die][Mus][Lin15][Knö14].

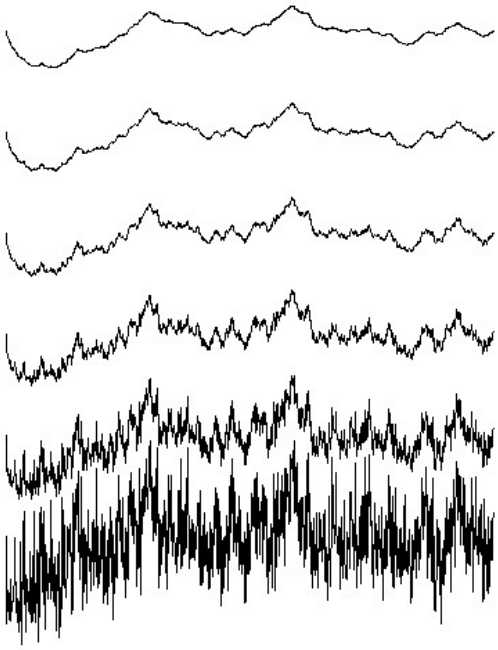


Abbildung 2.5.: Einfluss des fraktalen Inkrements auf das Fractional Brownian Motion [Mus]

Die einzelnen Kurven in Abbildung 2.5 zeigen von oben nach unten fBm-Noise bei einem H von etwa 1.0 bis 0.0 in 0.2 Schritten. Für den Abstand zwischen den Frequenzen wird ein Wert von 2.0 als sinnvoll erachtet.

Hybrid-Multifractal-Algorithmus

Eine Weiterentwicklung des Fractional Brownian Motion ist das sogenannte "Heterogeneous Fractional Brownian Motion", auch genannt Hybrid-Multifractal-Algorithmus. fBm hat das bekannte Problem, dass es statistisch homogen und isotrop ist. Dies bedeutet:

> Homogeneous means "the same everywhere" and isotropic means "the same in all directions" [Mus].

Dies ist in Betracht auf die Natur ein eher unrealistisches Erscheinungsbild. Um dies auszugleichen und ein heterogeneres Ergebnis zu erzielen wurde der Hybrid-Multifractal-Algorithmus entwickelt. Die Idee dahinter ist, dass Ebenen im Vergleich zu Bergen glatter sein sollen. Dazu

muss der Wertebereich der Basisfunktion angepasst werden, sodass das Ergebnis der Noise-Funktion näher an dem richtigen Wertebereich liegt. Dieser Bereich ist von Noise-Funktion zu Noise-Funktion unterschiedlich. Bei Perlin Noise zum Beispiel liegt dieser Wert zwischen 0 und 2 statt standardmäßig -1 und 1 [Knö14][Mus].

2.3.2. Wege

Um die Wege in einem Level zu realisieren, gibt es verschiedene Ansätze. Nachfolgend werden vier davon untersucht: der A^*-Algorithmus, Bézierkurven, Béziersplines und Catmull-Rom Splines.

A^*-Algorithmus

Der A^*-Algorithmus wird dazu verwendet, den kürzesten Pfad zwischen zwei Knoten zu bestimmen. Bei dem Algorithmus handelt es sich um ein informiertes Suchverfahren, da er Knoten auf Basis mehrerer Informationen auswählt. Dabei ist der Algorithmus in der Lage, einen geeigneten Pfad um Hindernisse herum zu finden. Der Algorithmus funktioniert wie folgt: Zunächst wird die gesamte Fläche in Zellen unterteilt. Eine Zelle besteht aus den Koordinaten an denen sie sich befindet, den Vorgänger dieser Zelle, der Information ob sie durch ein Hindernis blockiert ist und wie aufwendig das Betreten der Zelle ist (die Bewegungskosten, B). Die Startzelle wird zu einer offenen Liste von Zellen hinzugefügt und alle möglichen betretbaren Nachbarzellen dieser ermittelt und ebenfalls der offenen Liste hinzugefügt. Anschließend wird die Startzelle aus der offenen Liste entfernt und einer geschlossenen hinzugefügt. Für jede Zelle der offenen Liste wird ein Faktor F wie folgt berechnet: $F = B_{Start} + B_{Ziel}$. Bei B_{Start} handelt es sich um die Kosten um vom Startpunkt zur aktuellen Zelle zu gelangen und bei B_{Ziel} um die geschätzten Kosten um von der aktuellen Zelle aus den Zielpunkt zu erreichen. Als nächster Punkt des Weges wird die Zelle ausgewählt, welche den geringsten Faktor aufweist. Diese wird dann aus der offenen Liste entfernt und zu der geschlossenen hinzugefügt. Für diese Zelle werden wieder die Nachbarzellen ermittelt und geprüft. Befindet sich eine der Nachbarzellen bereits in der offenen Liste, wird geprüft ob die Bewegungskosten des Weges über die Zelle geringer sind. Falls ja, wird der Vorgänger der Zelle zu der günstigeren geändert. Dies wird für alle Zellen wiederholt bis die aktuelle Zelle dem Ziel entspricht oder die offene Liste leer ist, also kein Pfad gefunden wurde [SF][Les06][Ant14][JL05].

Bézierkurven

Bézierkurven wurden 1959 entwickelt um Kurven mit wenigen Punkten zu erzeugen. Eine Bézierkurve des n-ten Grades wird durch n+1 Punkte beschrieben. Es werden drei verschiedene Arten von Bézierkurven betrachtet: gibt es nur zwei Kontrollpunkte, so ist die Bézierkurve eine Grade zwischen diesen beiden Punkten. Diese Grade nennt sich lineare Bézierkurve. Sofern es drei Kontrollpunkte gibt, entsteht eine quadratische Bézierkurve. Diese beginnt beim ersten Punkt und endet beim Letzten. Die Kurve schlägt in Richtung des zweiten Punktes aus, schneidet ihn aber nicht. Um eine kubische Bézierkurve handelt es sich, wenn es vier Punkte gibt [Mat15][Qua][tut].

Béziersplines

Eine weitere Möglichkeit um Wege darzustellen ist über Splines. Der Begriff Spline stammt aus dem Schiffbau und bezeichnet dort Holzplatten am Schiffsrumpf, welche so gebogen werden, dass sie festgelegte Konturpunkte schneiden [Schb]. Eine mögliche Art von vielen verschiedenen Splines-Arten sind Béziersplines, auch B-Splines genannt. Dabei handelt es sich um folgendes: Will man eine komplexe Kurve darstellen kann eine einzelne kubische Bézierkurve unter Umständen dafür nicht mehr ausreichen. Zudem kann bei Bézierkurven sehr hohen Grades ein ungewolltes Rauschen auftauchen. Um dies zu vermeiden wird eine Bézierkurve in einzelne Segmente unterteilt, welche jeweils eine Bézierkurve eines niedrigeren Grades darstellen. Eine Kurve die stückweise durch Segmente dargestellt wird nennt man dann Bézierspline [Vuy04].

Catmull-Rom Splines

Eine weitere Art von Splines sind Catmull-Rom Splines. Diese stellen mathematische Kurven dar, welche, ebenso wie Bézierkurven, durch eine Liste von Punkten definiert werden. Die zwei Kurventypen unterscheiden sich jedoch dadurch, dass Catmull-Rom Splines alle Punkte der Kurve schneiden. Um einen einzelnen Punkt auf der Kurve zu berechnen werden die beiden nächsten Punkte und die Distanz des gewünschten Punktes zu ihnen betrachtet [Dun05a][Arm06][Twi03].

2.3.3. Labyrinthe

Es gibt sehr viele Möglichkeiten um Labyrinthe zu generieren. Nachfolgend werden sechs davon beschrieben und miteinander verglichen. Grundsätzlich gibt es beim Generieren von Labyrinthen zwei Ansätze. Der eine ist das “Graben” eines Pfades durch ein Feld aus Wänden,

der andere das Ziehen von Wänden auf einer anfangs leeren Karte. Nachfolgend wird es eine Beschreibung von grabenden und Wände ziehenden Algorithmen geben. Die folgenden Algorithmen werden hierbei für die Erstellung von Labyrinthen in Erwägung gezogen: Rekursion, Recursive Division, Recursive Backtracking, Binary Tree, Kruskal's Algorithmus und Prim's Algorithmus.

Rekursive Generierung

Der erste zu untersuchende Algorithmus ist das rekursive Erstellen eines Labyrinthes. Das Vorgehen ist dabei das folgende: Es wird mit der Zelle an der Position 0,0 begonnen. Von dieser Zelle aus wird geprüft, ob die direkt angrenzenden Zellen noch nicht besucht wurden. Ist dem so, werden sie einer Liste möglicher Richtungen hinzugefügt und die Anzahl möglicher Züge um eins erhöht. Falls nicht, wird geprüft, ob die aktuelle Zelle noch nicht besucht wurde und ob der vorherige Zug sich nicht in die exakt entgegensetzte Richtung bewegte. Heißt, es wird geprüft, dass sich nicht im Kreis bewegt wird. Ist dies nicht der Fall, wird an dieser Stelle in die entsprechende Richtung eine Wand platziert. Dieses Vorgehen wird für alle vier Richtungen (oben, unten, links und rechts) ausgeführt. Wenn es keine möglichen Schritte mehr gibt und die aktuelle Zelle noch nicht besucht wurde, handelt es sich dabei um das Ende des Algorithmus. Andernfalls wird die aktuelle Zelle als besucht markiert und es wird zufällig eine mögliche Richtung ausgewählt und das Vorgehen für die Zelle, die in dieser Richtung liegt, ausgeführt [sty15a].

Recursive Division

Bei dem Algorithmus "Recursive Division" werden ebenfalls Wände platziert. Dementsprechend beginnt der Algorithmus mit einem leeren Feld. Im ersten Schritt wird das Feld vertikal oder horizontal durch eine Wand halbiert. Innerhalb dieser ist lediglich ein Durchgang. Dieser Schritt wird auf beiden Seiten der Wand so lange wiederholt, bis die gewünschte Auflösung erreicht wurde [Buc11d].
Nachfolgend finden sich Beispiele des Algorithmus.

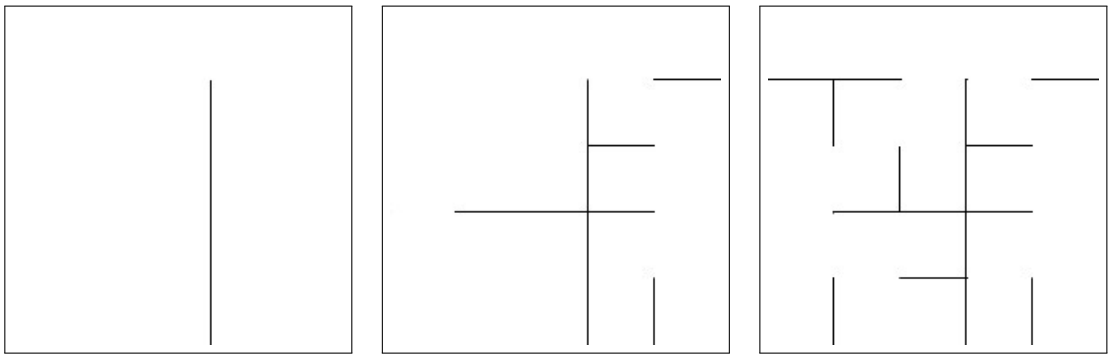


Abbildung 2.6.: Ausgewählte Schritte aus dem Algorithmus Recursive Division [Buc11d]

Recursive Backtracking

Der Recursive Backtracking Algorithmus gehört zu den grabenden Algorithmen und funktioniert wie folgt: zuerst wird ein beliebiger Startpunkt gewählt. Von diesem aus wird zufällig eine angrenzende Wand ausgewählt und ein Weg hindurch geformt, sofern die hinter der Wand liegende Zelle noch nicht besucht wurde. Dieser Schritt wird von der jeweils aktuellen Zelle aus wiederholt, bis alle angrenzenden Zellen besucht wurden. Ist dies der Fall, wird die letzte Zelle, die noch unbehandelte Wände hat, ausgewählt und diese als aktuelle Zelle behandelt. Der Algorithmus ist beendet, wenn die letzte mögliche Zelle wieder der Startpunkt ist [Buc10]. Nachfolgend eine Übersicht über die Funktionsweise des Algorithmus.

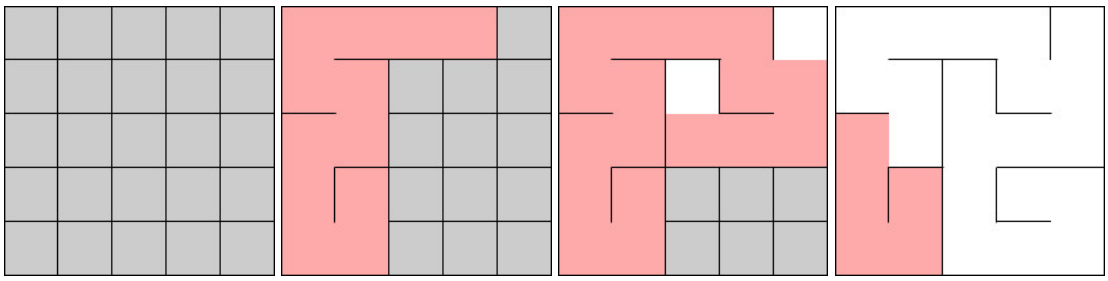


Abbildung 2.7.: Ausgewählte Schritte aus dem Recursive Backtracking Algorithmus [Buc10]

Binary Tree

Der Binary Tree Algorithmus ist von der Funktionsweise her ein sehr simpler grabender Algorithmus. Er funktioniert wie folgt: Für jede Zelle auf dem Feld wird zufällig ein Weg entweder in Richtung Norden oder in Richtung Westen gegraben. Diese Richtungen können

auch gegen Norden/Osten, Süden/Westen oder Süden/Osten getauscht werden [Buc11a]. Das folgende Beispiel zeigt den Algorithmus für die Ausrichtung Nord/Ost.

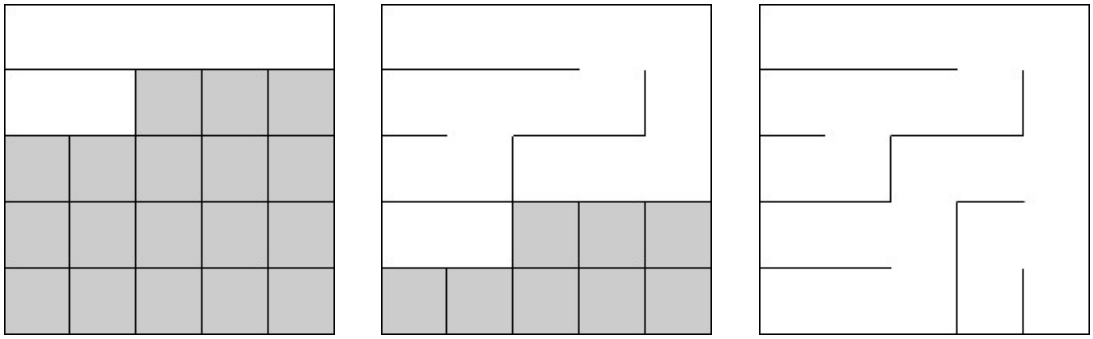


Abbildung 2.8.: Ausgewählte Schritte aus dem Binary Tree Algorithmus mit Ausrichtung Nord/Ost [Buc11a]

Kruskal's Algorithmus

Beim Algorithmus von Kruskal wird das Labyrinth als gewichteter Graph betrachtet und der minimale Spannbaum erzeugt. Der Algorithmus funktioniert im Standardfall wie nachfolgend beschrieben. Alle Kanten des Graphen werden "gesammelt" und anschließend wird die Kante mit dem geringsten Kantengewicht ausgewählt. Wenn diese Kante zwei disjunkte Bäume verbindet, werden die beiden Bäume zusammengeführt. Andernfalls wird die Kante verworfen. Dies wird wiederholt bis es keine Kanten mehr gibt. Der Algorithmus kann auch zufällig ausgeführt werden. In diesem Fall wird nicht die Kante mit dem geringsten Gewicht ausgewählt sondern eine zufällige [Buc11b].
Die unten stehenden Beispiele zeigen die Funktionsweise des Algorithmus, wenn man ihn zufällig ausführt.

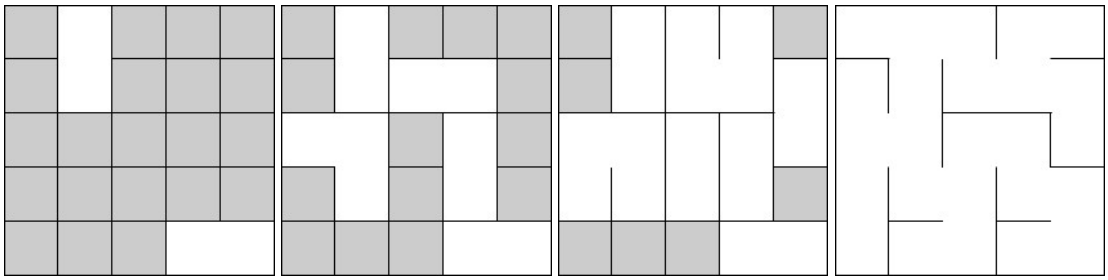


Abbildung 2.9.: Ausgewählte Schritte aus Kruskal's Algorithmus in der zufälligen Variante [Buc11b]

Prim's Algorithmus

Ebenso wie der Algorithmus von Kruskal bildet auch Prim's Algorithmus einen minimalen Spannbaum. Im Gegensatz zu der zufälligen Version von Kruskal's Algorithmus, bei welcher das Labyrinth von vielen verschiedenen Punkten aus über die Karte "wächst", wählt Prim's Algorithmus nur einen Punkt aus und breitet sich von diesem ausgehend aus. Zunächst wird ein zufälliger Eckpunkt aus dem Graphen ausgewählt und einer Liste, hier nachfolgend L, hinzugefügt (1). Anschließend wird aus dem Graphen die Kante mit dem kleinsten Kantengewicht ausgewählt, welche einen Eckpunkt aus L mit einem anderen Eckpunkt, der nicht in L enthalten ist, verbindet (2). Diese Kante wird danach zum minimalen Spannbaum hinzugefügt und der noch nicht in L enthaltene Eckpunkt der Kante L hinzugefügt (3). Die Schritte 2 und 3 werden wiederholt, bis L alle Eckpunkte des Graphen enthält. Auch dieser Algorithmus lässt sich zufällig durchführen, indem man in Schritt 2 statt der Kante mit dem geringsten Kantengewicht eine zufällige auswählt, welche zum Rand des Labyrinthes weist [Buc11c].
Die Beispiele zeigen die Standardvariante des Algorithmus.

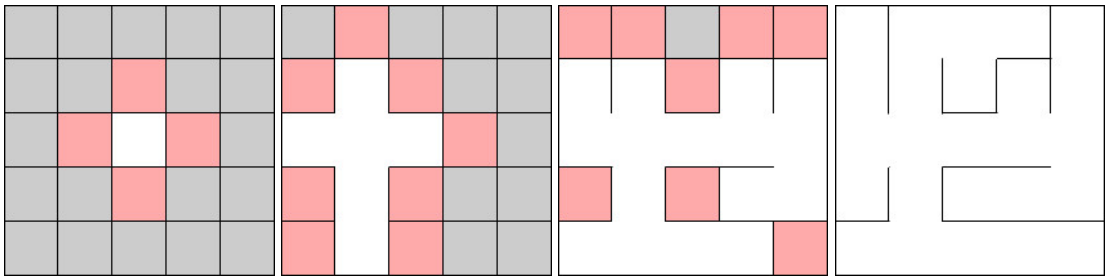


Abbildung 2.10.: Ausgewählte Schritte aus Prims's Algorithmus [Buc11c]

3. Analyse

In diesem Kapitel wird näher auf das eigentliche Ziel der Arbeit eingegangen. Es werden die Anforderungen an einzelne Komponenten gelistet und verwandte Arbeiten analysiert. Als Letztes wird aufgezeigt, wie sich das Projekt von diesen abgrenzt.

3.1. Anforderungen

Nachdem das Einsatzgebiet der zu entwickelnden Software erfasst wurde, sind Anforderungen zu erstellen. Das Programm ist dabei in mehrere Komponenten zu unterteilen: Die zu generierenden Szenen, das Baukastensystem und die domänenspezifische Sprache. Die Anforderungen sind für alle diese Komponenten zu erfassen.

3.1.1. Szenen

Es sind mindestens zwei Szenen zu generieren, welche über Parameter angepasst werden können. Jede parametrisierte Szene (= Level) bedient dabei eine bestimmte Emotion. Die generierte Szene muss dabei mindestens eine Größe von 100x100 Punkte unterstützen. Zusätzlich besitzt jede Szene einen definierten Start- und Endpunkt. Das Level ist erst erfolgreich beendet, wenn der Spieler den Endpunkt erreicht. Es ist dafür zu sorgen, dass der Spieler das Level nicht auf anderem Wege verlassen kann. Dies bedeutet, dass der Rand der Szene nicht überquerbar sein darf. Die Level sind geeignet zu kombinieren. Um die Szenen zu parametrisieren und klassifizieren ist des Weiteren eine domänenspezifische Sprache (kurz: DSL (**D**omain **S**pecific **L**anguage)) zu entwickeln.

Landschafts-Szene

Eine der zu generierenden Szenen stellt eine Landschaft dar. Diese Landschaft lässt sich parametrisieren und kann somit verschiedene Emotionen bedienen. Nachfolgend eine Übersicht über die Einstellmöglichkeiten.

- Bergigkeit: Es ist anzugeben, wie bergig das generierte Level sein soll. Dabei sind sowohl sehr flache als auch sehr bergige Level denkbar. Dazu ist eine geeignete Funktion zu wählen.
- Vegetation: Entsprechend eines realistischen Landschaftsbildes ist für eine zur Landschaft passende Vegetation zu sorgen. Denkbar ist auch eine Landschaft ohne jegliche Vegetation, wie beispielsweise eine Wüste.
- Wege: Die Landschaft verfügt über einen Weg vom Start- zum Endpunkt. Die Komplexität des Weges ist beeinflussbar zu gestalten. Dabei sollen sowohl sehr geradlinige als auch sehr kurvenreiche Wege möglich sein.

Bei der Bergigkeit des Geländes ist zu beachten, dass die generierte Landschaft eine maximale Steigung nicht übersteigen sollte. Als maximale Steigung ist die höchste Steigung einer Straße auf der Welt von 35% zu nehmen. Dies bedeutet konkret, dass auf 100m Strecke ein Höhenunterschied von etwa 35m vorliegt [Tra15]. Für die Wege gilt, dass sie passierbar sein müssen. Dies bedeutet konkret, dass der Weg zwischen dem Startpunkt und dem Endpunkt nicht unterbrochen wird und für den Nutzer durchgängig befahrbar ist. Um dies zu gewährleisten muss der Kurvenradius des Weges beachtet werden. Dazu liegen Richtwerte aus verschiedenen Ländern für Kfz-Straßen vor, allerdings im Geschwindigkeitsbereich von 70km/h bis 120km/h. Ein durchschnittlicher Radfahrer fährt nur etwa 10km/h bis 20km/h, ein Radprofi zwischen 40km/h und 50km/h [Scha]. Die Abbildung 7 “Minimaler Kurvenradius bei unterschiedlichen Geschwindigkeiten: ein internationaler Vergleich” in “Die neuen Entwurfsstandards für Außerortsstraßen im internationalen Vergleich” von Werner Brilon und Ray Krammes[BK] gibt für 70km/h einen minimalen Kurvenradius von 200m an. Es wird berechnet, dass bei 1km/h der Kurvenradius 2.9m betragen muss. Daraus folgt für eine Geschwindigkeit von 20km/h ein minimaler Kurvenradius von 58m. Die meisten Radfahrer werden allerdings vor einer Kurve langsamer. Um weiterhin ein kontrolliertes Fahren zu ermöglichen beträgt die minimale Geschwindigkeit etwa 8km/h [Wik16]. Daher sollte der Kurvenradius zwischen 23.2m und 58m liegen. Zudem soll es in dem generierten Weg keine Schleifen geben und die Richtung stets vorwärts sein. Die Fläche des Weges sollte möglichst ebenmäßig sein, um ein gleichmäßiges Fahren zu ermöglichen. Es ist für einen geeigneten Himmel zu sorgen.

Labyrinth-Szene

Die zweite zu generierende Szene stellt ein Labyrinth dar. Wie die Landschafts-Szene verfügt auch die Labyrinth-Szene über einen vordefinierten Start- und Endpunkt. Um Labyrinthe von

ansprechender Komplexität zu generieren ist ein geeigneter Algorithmus auszuwählen. Die Schwierigkeit ist dabei anpassbar zu gestalten. Die Höhe der Wände ist so zu wählen, dass es für den Spieler unmöglich ist, über sie hinweg zu sehen. Ebenfalls ist darauf zu achten, dass die Breite der Wege im Labyrinth groß genug ist, um das Labyrinth mit dem Fahrrad passieren zu können. Zusätzlich zur Komplexität soll das optische Erscheinungsbild des Labyrinthes anpassbar sein. Dies bezieht sich sowohl auf die Farbe als auch die Textur des Labyrinthes.

3.1.2. Emotionen

Wie in Kapitel 2.1.2 beschrieben, geht es im EmotionBike Projekt um die Erkennung und den Ausgleich von Emotionen. Die oben genannten Szenen sollen durch ihre Parametrisierbarkeit verschiedene Emotionen bedienen können. Es müssen daher Emotionen ausgewählt werden, welche mit den oben beschriebenen Szenen vermittelt werden können.

3.1.3. Baukastensystem

Es ist ein Baukastensystem zu entwickeln, welches geeignete Level kombiniert. Einzelne Level sind passend zu verbinden und müssen dementsprechend kategorisiert werden können. Eine geeignete Kombination kann beispielsweise das Verstärken einer Emotion sein, oder aber deren Ausgleich. In Abbildung 3.1 wird die Funktionsweise des Systems skizziert.

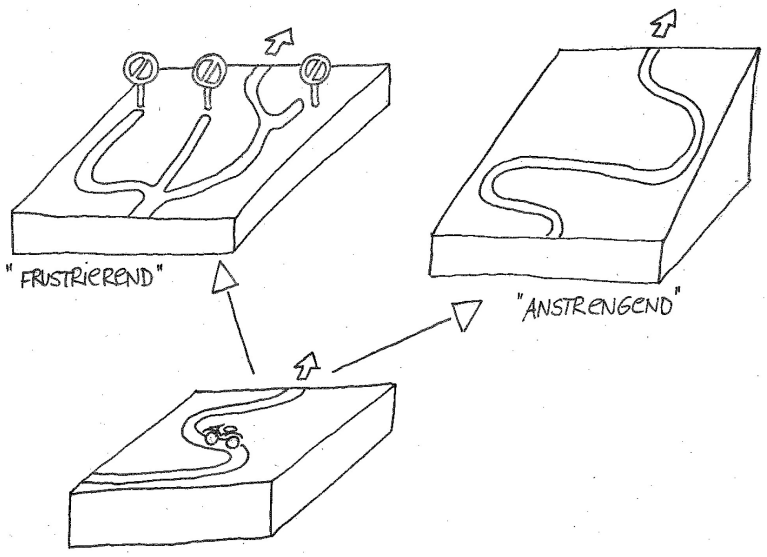


Abbildung 3.1.: Übersicht über die Funktionsweise des Baukastensystem [Jen]

Da das System, wie oben aufgeführt, dazu dient, Emotionen auszugleichen, wird nachfolgend in Tabelle 3.1 dargestellt, welche Emotionen jeweils auf eine spezifische Ausgangssituation folgen können.

entspannend	->	anstrengend / konzentriert / beunruhigend / spannend
anstrengend	->	entspannend / spaßig
konzentriert	->	entspannend / spaßig
beunruhigend	->	entspannend / spaßig / anstrengend
spannend	->	entspannend / konzentriert / spaßig
spaßig	->	anstrengend / konzentriert / beunruhigend / spannend

Tabelle 3.1.: Übersicht über die Emotionen und deren mögliche Folgeemotionen

Domänenspezifische Sprache

Es soll eine domänenspezifische Sprache (DSL) entworfen werden, um die Levelbausteine zu beschreiben. DSLs werden in interne und externe domänenspezifische Sprachen unterteilt. Die interne DSL kennzeichnet die direkte Nutzung von Ausdrucksmitteln. Die Überprüfung erfolgt durch einen Compiler und erfordert keinen speziellen Parser. Hierzu zählen beispielsweise Ruby und XML. Der Vorteil von internen DSLs liegt darin, dass diese weniger Implementierungsaufwand aufweisen. Bei einer externen DSL wird eine eigene Sprache mit eigener Syntax und Semantik entworfen und damit einhergehend auch eigene Mechanismen um diese Sprache zu übersetzen und auszuführen. Dabei handelt es sich beispielsweise um SQL oder reguläre Ausdrücke. Die Verwendung einer externen DSL hat den Vorteil, dass diese wesentlich flexibler sein kann. [ITW][Til12][Fow10].
Die DSL soll geeignet sein, um eine Szene in Bezug auf ihre Eigenschaften, wie beispielsweise Größe des Levels, und ihre emotionale Ausrichtung zu beschreiben.

3.2. Verwandte Arbeiten

Das Verwenden von prozeduraler Generierung zur Erzeugung von Landschaften ist nicht neu. Dementsprechend existieren bereits Arbeiten, welche sich mit diesem Vorgehen auseinandersetzen. Nachfolgend werden vier davon beschrieben und bewertet. Dies sind “Moderne Techniken zur Generierung prozeduraler Landschaften” von Stephan Knödler, “Prozedurale Modellierung: Terrains” von Katrin Scharnowski, das Unity-Asset “World Builder” und das Paper “A Survey of Procedural Methods for Terrain Modelling” von Ruben M. Smelik, Klaas Jan de Kraker, Saskia A. Groenewegen, Tim Tutenel und Rafael Bidarra.

3.2.1. Stephan Knödler: *Moderne Techniken zur Generierung prozeduraler Landschaften*

Die Bachelorarbeit “Moderne Techniken zur Generierung prozeduraler Landschaften” von Stephan Knödler von der Fakultät für Elektrotechnik und Informatik der Technischen Hochschule Ingolstadt[Knö14] beschäftigt sich mit Algorithmen und Techniken, um eine unendliche Landschaft generieren zu können. Zunächst wird darin definiert, was eine realistische Landschaft ausmacht. Anschließend werden verschiedene Methoden, wie beispielsweise Perlin Noise und Fractional Brownian Motion, beschrieben und bewertet. Dann werden Konzepte zur Generierung genannt und die Implementierung mit OGRE und Unity beschrieben und analysiert.
Die Analyse und Implementierung realistischer Landschaften Knödlers erweist sich in einigen Punkten, wie beispielsweise dem Vorhandensein von Flüssen und Erosion, als zu umfangreich, um sie in dieser Arbeit zu berücksichtigen, ist allerdings als Ausblick und Weiterentwicklungsmöglichkeit zu betrachten. Die dargestellten Methoden zur Generierung sind das Perlin Noise, Simplex Noise, Fractional Brownian Motion, der Hybrid-Multifractal-Algorithmus und das Domain Warping. Bis auf das Domain Warping sind die Ergebnisse dieser Arbeit zu den anderen Methoden ausgesprochen hilfreich und können für diese Arbeit zurate gezogen werden. Das Domain Warping beschreibt die Kombination der Fractional Brownian Motion mit der Skalierung der Koordinaten und der Verwendung eines Koeffizienten. Das in Abbildung 3.2 gezeigte Ergebnis entspricht optisch nicht dem gewünschten Ergebnis. Zudem ist die Komplexität dieser Methode zu hoch [Knö14].

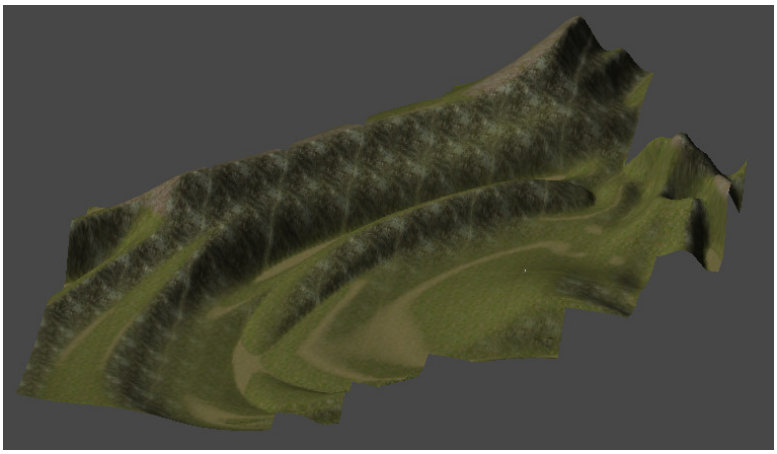


Abbildung 3.2.: Beispiel für ein mittels Domain Warping erstelltes Terrain [Knö14]

3.2.2. Katrin Scharnowski: *Prozedurale Modellierung: Terrains*

In der Arbeit "Prozedurale Modellierung: Terrains" von Katrin Scharnowski des Instituts für Visualisierung und Interaktive Systeme der Universität Stuttgart[Sch10] geht es um die Generierung von Terrains auf Basis prozeduraler Modellierung. Dabei wird die Modellierung von Terrain mittels Heightmaps und Voxeln beschrieben. Als Heightmap-basierte Verfahren werden folgende Methoden genannt und verglichen: der Fault-Algorithmus, Voronoi-Diagramme, Noise-Funktionen und thermale Erosion. Als Voxel-basiertes Modell wird der Marching-Cubes-Algorithmus genannt. Der Unterschied zwischen Heightmap- und Voxel-basierten Modellen ist, dass es bei ersteren nur einen Höhenwert pro Koordinatenpunkt gibt und sich dementsprechend weder Höhlen noch Brücken generieren lassen. Bei Voxel-basierten Modellen wird das Terrain hingegen in einem dreidimensionalen Array gespeichert. Dieses Array wird Dichtefunktion genannt. Der Wechsel des Vorzeichens innerhalb der Dichtefunktion bestimmt die Terrainoberfläche. Nachfolgend ein Beispiel zu dieser Methode [Sch10].

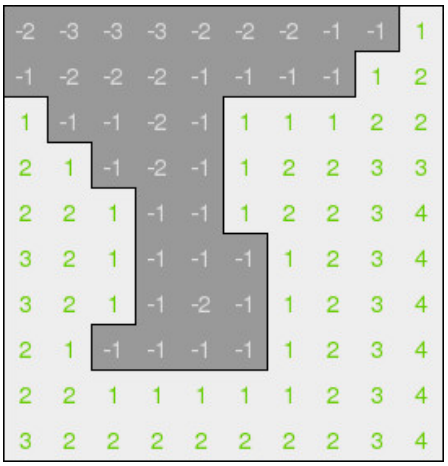


Abbildung 3.3.: Beispiel für ein Voxel-basiertes Terrain

Die positiven Punkte befinden sich dabei unterhalb der Terrain-Oberfläche, die negativen oberhalb.
Beim Marching-Cubes-Algorithmus wird zunächst das gesamte Gelände in Voxel, würfelförmige Unterabschnitte, unterteilt und anschließend werden in jedem Voxel die Vorzeichen der Dichtefunktion aller acht Ecken betrachtet und wie folgt bewertet [Sch10].

- Vorzeichen aller Ecken sind positiv: Der Voxel liegt komplett innerhalb des Terrain.
- Vorzeichen aller Ecken sind negativ: Der Voxel liegt komplett außerhalb des Terrain.

- Es gibt positive und negative Werte: Der Voxel schneidet die Terrain-Oberfläche. Mit Hilfe der Werte der Dichtefunktion der Ecken können die Schnittpunkte der Polygone mit den Würfelkanten interpoliert werden. Diese liegen auf dem Nullpunkt zwischen Ecken mit jeweils unterschiedlichen Vorzeichen.

Für diese Verfahren muss allerdings zunächst eine Dichtefunktion geeignet generiert werden [Sch10]. Auf Grund des höheren Aufwandes durch die Tatsache, dass jene Funktion erst erstellt werden muss und das gesamte Gelände zunächst in geeignet kleine Voxel unterteilt werden muss, ist die Voxel-basierte Generierung ungeeignet. Zumal derzeit weder Höhlen noch Brücken geplant sind.
Die Arbeit hat auch einige Heightmap-basierte Verfahren beleuchtet. Das erste davon ist der Fault-Algorithmus. Bei diesem wird das Terrain durch eine Gerade geteilt und Punkte der Heightmap einem der Abschnitte zugeteilt. Die Punkte der einen Seite werden um einen bestimmten Offset abgesenkt und die der anderen Seite angehoben. Dieses Verfahren muss geeignet oft wiederholt werden um ein Terrain zu generieren. Das erzeugte Terrain ist eher eckig und damit ungeeignet. Eine weitere Methode, die Voronoi-Diagramme, definiert eine Aufteilung des Raumes in Regionen in Abhängigkeit einer Teilmenge von Punkten. Diese Region eines Punktes enthält die Punkte, die zu diesem Punkt einen geringeren Abstand haben als zu allen anderen Punkten. Das generierte Resultat zeigt allerdings sehr viele unnatürliche, gerade Kanten und ist daher ebenfalls ungeeignet. In Scharnowskis Arbeit werden die Noisefunktionen lediglich zur Korrektur und Verbesserung anderer Methoden verwendet [Sch10]. Die Ergebnisse dieser Arbeit sind dementsprechend eher in Bezug auf den Ausschluss bestimmter Methoden aufschlussreich [Sch10].

3.2.3. AJOpinnayteyo: *World Builder*

Bei “World Builder” handelt es sich um ein Unity Asset von AJOpinnayteyo. Der Nutzen von “World Builder” ist es, ein möglichst realistisches Gelände zu generieren. Dazu sind folgende Schritte nötig: Zunächst wird das Asset aus dem Asset Store kostenlos heruntergeladen. Anschließend muss ein leeres Terrain erstellt und das Konfigurationsfenster von “World Builder”, sichtbar in Abbildung 3.4, geöffnet werden [AJO13].

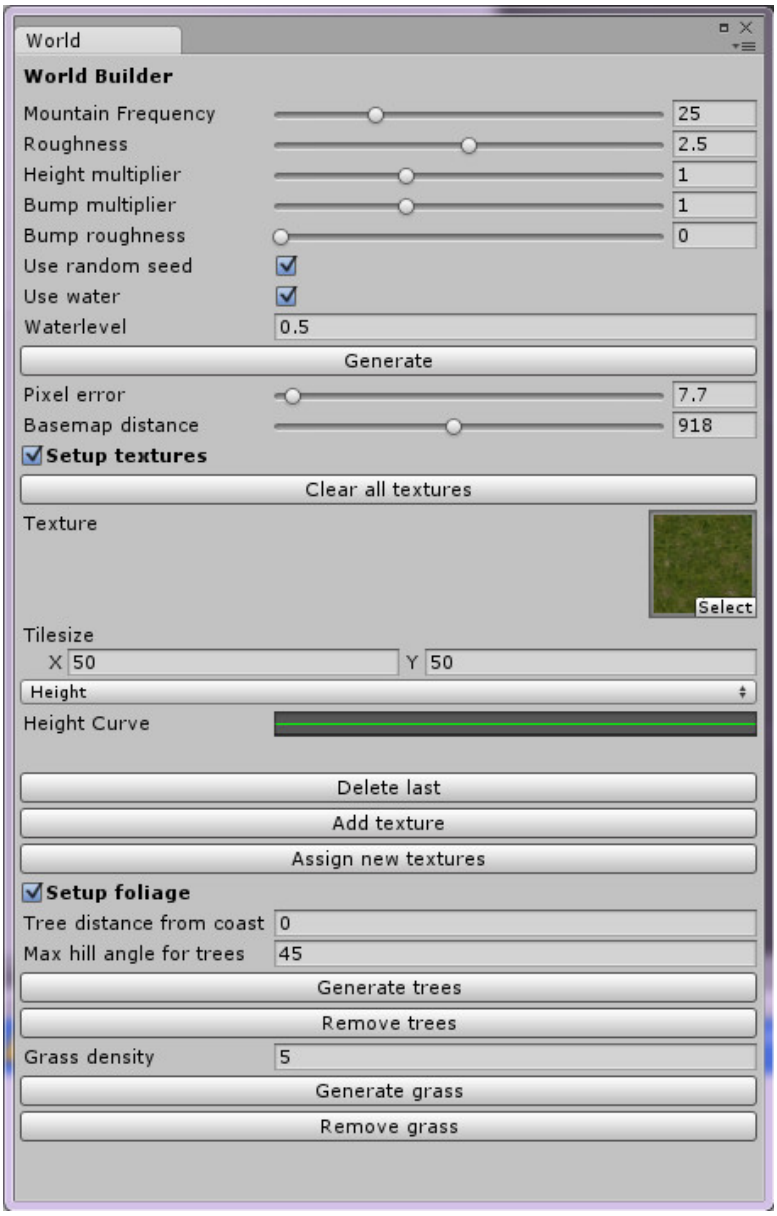


Abbildung 3.4.: Übersicht über das Konfigurationsfenster von World Builder

In diesem hat man folgende Einstellmöglichkeiten:

- **Mountain Frequency:** Mit Mountain Frequency kann eingestellt werden, wie bergig das Gelände sein soll.

- **Roughness:** Roughness bestimmt, wie spitz die erzeugten Berge sind. Wird ein geringer Wert gewählt, kann das Ergebnis ein einzelner eher rundlicher Berg sein. Bei einem hohen Wert entstehen, je nach Mountain Frequency, einige spitze Berge.
- **Height Multiplier:** Die gesamte Höhe des Gelände kann über Height Multiplier bestimmt werden. Je höher der Wert, desto höher das Ergebnis.
- **Bump Multiplier:** Der Bump Multiplier bestimmt wie viele kleine Hügel (Bumps) vorhanden sind.
- **Bump Roughness:** Um zu beeinflussen wie Spitz die Bumps sind, wird die Bump Roughness eingestellt.
- **Use Random Seed:** Dabei handelt es sich um die Option, Gelände mit einem bestimmten Seed zu generieren um das Gelände mit Kenntnis dieses Seed immer wieder exakt so generieren zu können. Alternativ wird ein zufälliger Seed verwendet.
- **Use Water:** Bei dieser Checkbox wird bestimmt ob Wasser im generierten Gelände vorkommen soll und wie tief dieses sein soll (**Waterlevel**).

Zusätzlich ist es möglich, dem Gelände direkt Texturen zuzuweisen und Vegetation zu generieren. Bei letzterem kann definiert werden, wie weit Bäume von der Küste entfernt sein sollen (**Tree Distance From Coast**, bis zu welcher Steigung Bäume an Bergen generiert werden können (**Max Hill Angle For Trees**) und wie dicht Gras platziert werden soll (**Grass Density**) [AJO13].
Die letzte Version von "World Builder" wurde am 20. September 2013 für die Unity Version 4.2.1 veröffentlicht [AJO13] und kaum bis überhaupt nicht dokumentiert. Es ist auch nach vielfachen Versuchen nicht gelungen, ein ansprechendes Gelände mit "World Builder" zu generieren. Nachfolgend wird in Abbildung 3.5 ein Ergebnis des Herstellers im Vergleich zu einem selbst erstellten Ergebnis, in Abbildung 3.6, gezeigt.

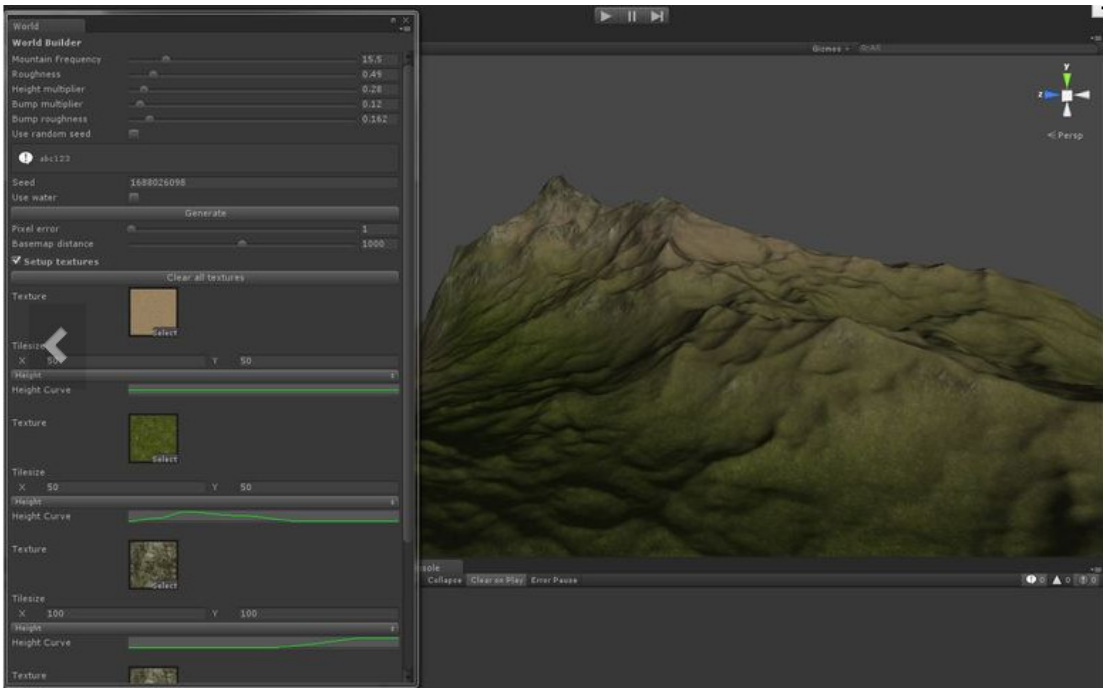


Abbildung 3.5.: Beispiel Terrain von World Builder aus dem Asset Store [AJO13]

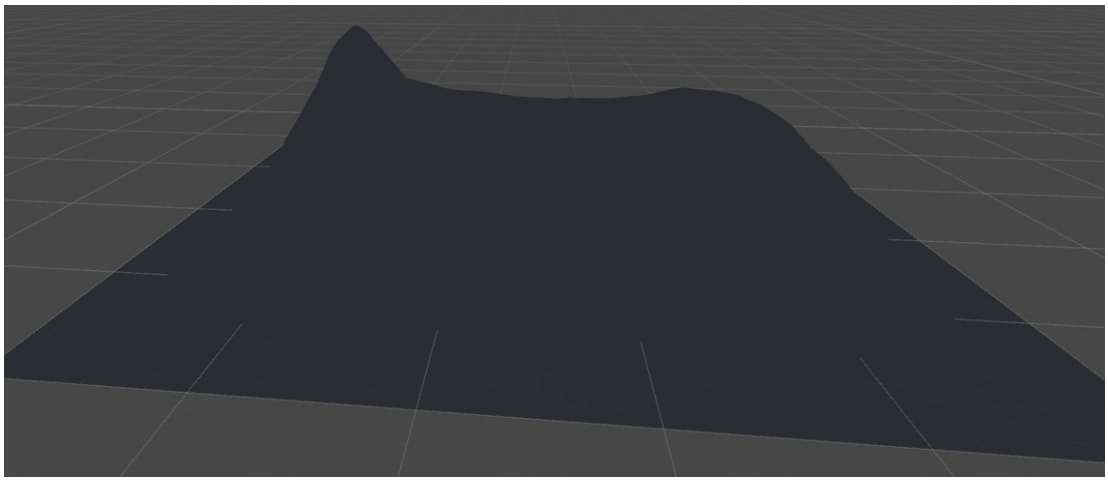


Abbildung 3.6.: Ein mit World Builder generiertes Terrain

Anhand der Abbildung 3.6 ist gut erkennbar, dass “World Builder” mit der aktuellen Unity Version vollkommen ungeeignet für die Erstellung von Gelände ist.

3.2.4. R. M. Smelik et al.: *A Survey of Procedural Methods for Terrain Modelling*

Das Paper "A Survey of Procedural Methods for Terrain Modelling" von Ruben M. Smelik, Klaas Jan de Kraker und Saskia A. Groenewegen von TNO Defence, Security and Safety und Tim Tutenel und Rafael Bidarra von der Delft University of Technology [Sme+09] beschäftigt sich mit Methoden der prozeduralen Generierung. Diskutiert werden für diese Methoden wichtige Faktoren, unter anderem der Realismus des Ergebnisses und die Performance. Es werden fünf verschiedene Bereiche, in denen prozedurale Generierung zum Einsatz kommen kann, beleuchtet: Heightmaps, Wasser, Vegetation, Straßen und urbane Umgebungen. Für die Generierung von Heightmaps werden verschiedene Algorithmen genannt. Zunächst wird erklärt, wie Heightmaps früher generiert wurden: eine grobe Heightmap wird iterativ zufällig unterteilt um Details hinzuzufügen. Heutzutage finden eher fraktale Algorithmen Anwendung. Die generierten Heightmaps können transformiert werden um beispielsweise geglättet zu werden oder um Erosion darzustellen. Die grundlegenden Noise-Funktionen generieren recht zufällige Ergebnisse und der Einfluss des Nutzers beschränkt sich auf die Eingabe von oftmals kontraintuitiven Parametern. Es wird aufgezeigt, dass es Methoden gibt, welche über Constraints konfiguriert werden und auf Basis derer ein geeignetes Terrain finden. Ebenfalls möglich sind Algorithmen welche mittels Grauwert-Bildern oder Verlaufslinien Terrain generieren. Eine weitere beleuchtete Methode ist das Verwenden von Tools um von Hand das Terrain in 3D zu malen [Sme+09].
Zu Gewässern sind grundsätzlich zwei Arten von Algorithmen zu nennen: Solche, die während der Generierung der Heightmap ausgeführt werden, und solche, die danach erst angewendet werden. Eine Möglichkeit für das Generieren von Flüssen ist, der Start mit einem einzelnen geraden Fluss welcher rekursiv unterteilt wird. Das Gelände wird anschließend "drum herum" generiert. Eine andere Methode ist die Kombination von einem geschwungenen Fluss mit einer Heightmap, welche durch Unterteilungen entstanden ist. Das Gelände wird dabei in Dreiecke unterteilt und in jeder Iteration weiter unterteilt. Nach acht oder mehr Schritten entsteht ein natürlich aussehendes Terrain. Nachteilig dabei ist, dass das Flussbett stetig sehr tief liegt und sich entsprechend unnatürlich durch das Gelände gräbt. Zu dem Zeitpunkt der Entstehung des Papers ist festzuhalten, dass die Generierung von Gewässern wenig beachtet wurde [Sme+09].
Bei der Generierung von Vegetation wird unterschieden in das prozedurale Erstellen von Pflanzen und die Platzierung dieser. Prozedurale Pflanzen "wachsen" vom Stamm aus und werden durch das Hinzufügen zunehmend kleinerer Äste und Blätter vervollständigt. Dies kann beispielsweise über L-Systeme oder Graphen und Subgraphen erfolgen. Um Vegetation zu platzieren wird ein Modell genannt, bei welchem auf Basis der Heightmap, einer Wasser-

karte, Informationen über zu platzierende Pflanzen, wie beispielsweise die Wachstumsrate, und anderer Kriterien platziert wird. Zu all diesen Methoden wird das Fazit gezogen, dass die Platzierung glaubwürdige Ergebnisse liefert [Sme+09].
Die Schwerpunkte des Paper scheinen auf der Generierung von Straßen und urbanen Umgebungen zu liegen. Da es jedoch derzeit keine Relevanz dieser Themen für diese Arbeit gibt, wird auf diese nachfolgend nur kurz eingegangen. Um Straßen zu generieren, können beispielsweise auf Mustern basierende Algorithmen, L-Systeme, Agentensimulationen oder auch Tensorfelder verwendet werden. Die einfachste Methode ist, das Gelände in ein Gitter zu unterteilen. Anschließend wird ein Rauschen zu dem Gitter hinzugefügt um es weniger wiederholend wirken zu lassen. Eine andere Möglichkeit ist die Verwendung von Templates beziehungsweise die Verwendung von Mustern von echten Straßen. Bei allen genannten Methoden werden die generierten Straßen vom unterliegenden Terrain beeinflusst. Bei zu rauem Gelände wird das generierte Ergebnis dementsprechend nicht geeignet sein und das Gelände muss angepasst werden. Für urbane Landschaften wird im Normalfall mit einer leeren Karte mit Straßen begonnen und polygon-förmige Bereiche zwischen den Straßen identifiziert. Diese Regionen werden weiter unterteilt. Auf eine ähnliche Weise können Flurpläne von Gebäuden generiert werden. Andere Möglichkeiten sind auch hier wieder L-Systeme, *shape grammar* und *wall grammar*. Diese Methoden sind schnell und liefern optisch ansprechende Ergebnisse. Allerdings sind die generierten Städte oftmals unrealistisch. Es werden Methoden gesucht, welche auf Basis bereits existierender Städte urbane Landschaften generieren [Sme+09].

Das Paper reißt viele mögliche Methoden an, ohne näher auf sie einzugehen. Dementsprechend sind die Ergebnisse eher für die Sichtung von Möglichkeiten geeignet und erfordern weitere Nachforschungen.

3.3. Abgrenzung

Die vorliegende Arbeit soll einige grundlegende Emotionen mit zwei parametrisierbaren Szenen abdecken. Zusätzlich wird ein leicht erweiterbares Baukastensystem und eine leicht verständliche DSL entwickelt. Es wird Wert darauf gelegt, dass die Einarbeitungszeit in das System gering ist, also für Personen mit Kenntnissen in Informatik, insbesondere XML und C#, gut verständlich. Eine umfassende Übersicht über Emotionen, beziehungsweise Szenen die Emotionen repräsentieren, soll aufgrund des zeitlichen Umfangs nicht gegeben werden. Ebenso wird auf eine engineunabhängige Entwicklung sowie das Darstellen von mehreren Biomen innerhalb einer Szene beziehungsweise eines Levels verzichtet.

4. Konzept

Das Kapitel Konzept beleuchtet, wie genau die Bestandteile der Arbeit implementiert, wie die Bausteine verbunden und ob und wie das Resultat in bereits vorhandene Systeme integriert werden soll.

4.1. Baukastensystem

Um das Baukastensystem zu realisieren und Bausteine zu kombinieren, gibt es mehrere Möglichkeiten. Nachfolgend werden drei davon beleuchtet: die dynamische Anpassung eines Levels, das Verbinden von Levelbausteinen durch Verbindungsobjekte und die Verwendung von Chunks.

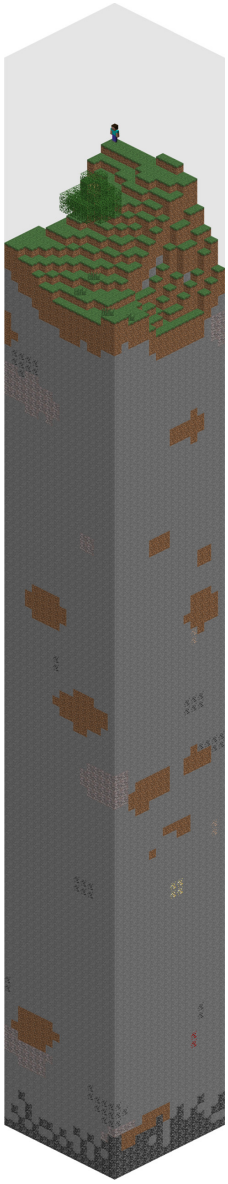
4.1.1. Dynamische Anpassung

Die erste Methode um Bausteine zu kombinieren ist die dynamische Anpassung. Diese funktioniert wie folgt: Es wird initial ein Levelbaustein beliebiger Größe generiert. Anschließend wird dynamisch Gelände nach-generiert und entsprechend der neuen Emotionen angepasst. Folgendes Szenario sei als Beispiel gegeben: Der Nutzer fährt durch eine entspannte Welt und zeigt erste Anzeichen von Langeweile in seinem Gesicht. Das System erhöht die Steigung des Geländes um so die Schwierigkeit zu steigern.

4.1.2. Verbinden von einzelnen Levelbausteinen

Eine einfache Möglichkeit, Levelbausteine zu verbinden, ist folgende: Es wird ein Baustein ausgewählt, in welchem der Spieler startet. Nachdem er diesen durchquert hat, gibt es eine Verbindung zu einem anderen Baustein. Um dies zu gewährleisten, benötigt jede Szene beziehungsweise jedes Level einen festen Start- und Endpunkt und ein Verbindungsobjekt. Bei diesem kann es sich beispielsweise um eine Höhle oder auch einen Tunnel handeln. Denkbar wäre auch ein Übergang ins “Nichts”. Damit ist gemeint, ein Spieler verlässt einen Levelbaustein und fährt durch ein helles Licht und verlässt dieses wieder, um den neuen Baustein zu betreten.

4.1.3. Verwendung von Chunks



Ähnlich wie bei Minecraft [Moj] ist es denkbar, eine "unendliche" Spielwelt in Chunks zu unterteilen. Ein Chunk ist in Minecraft ein Bereich mit einer Breite und Länge von 16 und einer Höhe von 256 Blöcken [Gam16a]. Die Größe eines Chunks ist so gewählt worden, da sich Chunks mit dieser Größe schnell und mit wenig Performanceeinbrüchen generieren beziehungsweise laden lassen [Not10].
In Abbildung 4.1 ist ein Beispielchunk zu sehen. Chunks halten jeweils Informationen über die Blöcke innerhalb des Chunks, das Biom, Blockobjektdaten und Objektdaten. Standardmäßig werden um den Chunk herum auf dem sich ein Spieler befindet in alle Richtungen 8 Chunks geladen. Dabei handelt es sich um 289 um den Spieler herum geladene Chunks. Dies soll dafür sorgen, dass seltener Chunks neu generiert beziehungsweise nachgeladen werden müssen. Wird zur Verbindung der Level ebenfalls eine Verwendung von Chunks gewünscht, muss dementsprechend auch eine geeignete Chunkgröße gewählt werden, sodass Chunks zur Laufzeit nachgeladen werden können. Und zudem muss entschieden werden, wie viele Chunks aktiv gehalten werden sollen. Aufgrund des fehlenden Bedarfs, Höhlen in den Untergrund zu bauen und dementsprechend der fehlenden Notwendigkeit einen Chunk in die Tiefe gehen zu lassen, beschränkt sich die Größe auf eine Breite und Länge.

Abbildung 4.1.: Beispiel für einen Chunk in Minecraft [Gam16a]

4.1.4. Vergleich

Nachfolgend werden kurz die Schwierigkeiten der einzelnen Methoden beleuchtet und eine geeignete Methode ausgewählt.

Dynamische Anpassung

Die dynamische Anpassung beinhaltet mehrere Schwierigkeiten. Zum einen gibt es derzeit das Problem, dass eine Steigungsänderung sich erst nach etwa drei Minuten auf den Widerstand der Pedale des EmotionBikes auswirkt. Dieses Problem wäre dadurch zu lösen, dass die Anpassung des Geländes erst nach einer gewissen Zeit eintritt und der Befehl an das EmotionBike zu diesem Zeitpunkt bereits abgearbeitet wurde. Dieses Vorhersehen des Geländes setzt allerdings voraus, dass weite Teile bereits generiert und bekannt sind. Dies widerspricht allerdings der Idee der prozeduralen Generierung. Des Weiteren ist die Generierung von Gelände sehr rechenintensiv und somit kann bei einer Live-Generierung kein verzögerungsfreies Fahren garantiert werden. Als Letztes sei zu erwähnen, dass es denkbar ist, dass sich Level komplett ändern. Beispielsweise kann auf eine Welt, bestehend aus einem Waldgebiet, eine Höhle folgen. Solch ein dynamischer und realistischer Übergang ist bei einem live angepassten System nur schwer realistisch realisierbar.

Verbinden von einzelnen Levelbausteinen

Die nächste Variante, das Verbinden von Levelbausteinen, hat den Vorteil, dass die Levelübergangsobjekte beliebig lang sein können. Somit ließen sich Ladezeiten der neuen Bausteine bequem verstecken. Nachteilig an diesem System ist die Berechenbarkeit: der Spieler wird schnell erkennen, dass diese Objekte nur dem Übergang dienen und Ladezeiten verstecken sollen.

Verwendung von Chunks

Problematisch an der Verwendung von Chunks sind die Übergänge zwischen diesen. Ohne geeignete Übergangsfunktion ist es möglich, dass sehr harte und unpassende Übergänge entstehen. Beispielsweise wenn auf einen sehr flachen Chunk ein sehr bergiger folgt, müssen geeignete Übergänge geschaffen werden. In Minecraft ist dies aufgrund der Tiefe eines Chunks weniger problematisch, an solch einem Übergang befindet sich im schlimmsten Fall lediglich eine hohe Wand. Im vorliegenden Programm ist dies jedoch kritisch, da lediglich die Terrainoberfläche festgelegt wird und fehlerhafte Übergänge entsprechend zu “Löchern im Boden” führen können, in welche der Nutzer “fallen” kann.

Aufgrund der Nachteile der dynamischen Anpassung und der Verwendung von Chunks und der einfacheren Realisierbarkeit der zweiten Methode, dem Verbinden einzelner Bausteine, wird diese für die Arbeit gewählt.

4.2. Domänenspezifische Sprache

Aufgrund des geringeren Implementierungsaufwandes und der fehlenden Notwendigkeit großer Flexibilität wird eine interne DSL verwendet. Es wird XML (E**x**tensible **M**arkup **L**anguage) als DSL gewählt. Dabei handelt es sich um ein text-basiertes Format welches software- und hardwareunabhängig ist. Informationen werden dabei in Tags mit dem Schema $< tagAnfang > < /tagEnde >$ verpackt. Diese Tags und die Struktur müssen vom Anwender selbst definiert und ausgewertet werden [sel16][W3S].

```
<vorlesung>
        <beginn>08:15</beginn>
        <ende>11:30</ende>
        <raum>10.60</raum>
        <name>Programmieren I</name>
</vorlesung>
```

Listing 4.1: XML Beispiel

In Listing 4.1 ist ein Beispiel für XML zu sehen. Die verwendeten Tags sind so gewählt, dass sie eine Vorlesung beschreiben können. Der Vorteil von XML ist seine große Verbreitung. Zudem lässt es sich sehr bequem in viele Programmiersprachen einbinden. Ein weiterer Vorteil ist die gute Menschenlesbarkeit [sel16][W3S].

4.3. Emotionen

Es wurden aus einer Vielzahl möglicher Emotionen einige ausgewählt, welche mit den generierten Leveln abgedeckt werden. Zu jeder Emotion werden mögliche Darstellungsformen dieser Emotion dargestellt.

- Entspannend: Die erste Emotion soll Entspannung sein. Dabei ist eine Landschaft zu erstellen, welche maximal eine leichte Steigung aufweist, verhältnismäßig viel Vegetation und einen relativ entspannten Weg hat. Die Lichtverhältnisse und dementsprechend auch der Himmel sollten einen höchstens leicht bewölkten Tag oder einen sommerlichen Abend porträtieren. Die Textur des Terrains sollte dabei relativ grün sein und das Level eventuell Elemente aus Wasser beinhalten [GK16].

- Anstrengend: Anstrengung ist eine Emotion, die auf verschiedene Arten interpretiert und dargestellt werden kann. Zum einen ist physische Anstrengung darzustellen. Dies wird über ein mittel bis stark bergiges Gelände erreicht. Der Weg sollte zudem eine mittlere

bis hohe Komplexität aufweisen. Die Textur und die Vegetation können unterschiedlich ausfallen. So ist zum einen denkbar, das Level im Stil eines Waldes beziehungsweise einer Wiese mit Waldanteil, ähnlich wie der entspannten Landschaft, zu gestalten. Zum anderen jedoch auch beispielsweise die Gestaltung im Stil einer Wüstenlandschaft mit eher wenig Vegetation. Die andere Art der Anstrengung ist die Psychische. Diese lässt sich über das Lösen eines relativ komplexen Labyrinthes realisieren. Die Stimmung von selbigem sollte eher düster aber nicht beängstigend sein.

- Konzentriert: Konzentration kann über das Lösen von Aufgaben erreicht werden. Dies kann zum einen das Lösen eines Labyrinthes sein, zum anderen aber auch beispielsweise das Balancieren durch einen Parcours. Dieser Parcours ließe sich zum Beispiel durch eine Kombination aus schmalen Planken, die einen Pfad ergeben, realisieren. Die Lösung dieser Aufgabe sollte optional sein und das Scheitern nicht das erfolgreiche Beenden des Levels verhindern.

- Beunruhigend: Das beunruhigende Element soll in diesem Falle in einem Labyrinth erzeugt werden. Zum einen ist dies realisierbar durch ein sehr düsteres Level mit wenig Beleuchtung. Zum anderen auch durch ein komplexes Labyrinth mit sehr vielen Sackgassen und auch möglicherweise plötzlich eintretenden Aktionen [GK16].

- Spannend: Die Spannung kann über ein sehr bergiges Level mit einem sehr komplexen Weg erzeugt werden oder ebenfalls durch ein Labyrinth. Allerdings ist Spannung ein sehr subjektives Empfinden und somit schwierig zu erreichen.

- Spaßig: Spaß kann durch vielfältige Dinge erzeugt werden. Zum einen kann das Lösen eines simpleren Labyrinthes mit einem helleren Thema Spaß auslösen, zum anderen aber auch das Fahren durch eine Landschaft mit eintretenden Aktionen oder zu lösenden Aufgaben.

Generell ist es denkbar, alle diese Emotionen durch geeignete Soundkulissen zu unterstreichen. Für das Hervorrufen von Emotionen spielen Ton und Lichtverhältnisse oftmals eine entscheidende Rolle.

4.4. Generierung der Spielewelt

Nachfolgend werden die möglichen Methoden aus Kapitel 2.3 verglichen und bewertet sowie die Verfahren ausgewählt, welche für die Realisierung der jeweiligen Anforderung am geeignetsten sind.

4.4.1. Gelände

Es muss aus den im Kapitel 2.3.1 beschriebenen Algorithmen ein geeigneter für die Generierung des Geländes ausgewählt werden.

Zufallszahlen

Zunächst wird betrachtet ob die Generierung mittels Zufallszahlen ausreichend ist, da dies die einfachste Variante darstellt.

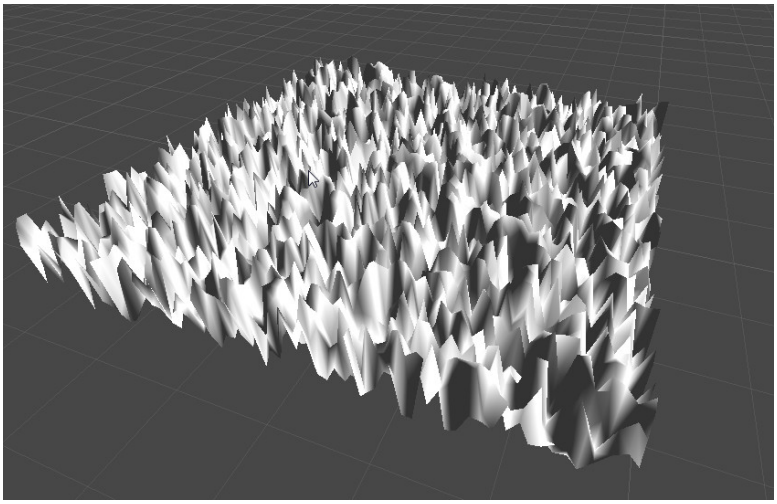


Abbildung 4.2.: Terrain mit mit Zufallswerten generierter Heightmap [Pig14][Knö14]

Wie in Abbildung 4.2 zu sehen ist, wirkt das Ergebnis sehr willkürlich und hat wenig mit einer sinnvollen Landschaft zu tun. Dies macht deutlich, dass Geländegenerierung mehr erfordert als Zufallszahlen [Knö14].

Perlin Noise

Die erste und einfachere Noise Funktion ist das Perlin Noise. Das generierte Gelände in Abbildung 4.3 ist vergleichsweise wellig und die Berge sind zusammenhängend und wirken daher nicht wie echte Berge. Trotz dieses leicht unnatürlichen Aussehens ist der Algorithmus gut parametrisierbar und durch das Vorhandensein von Bibliotheken [Bev07][Mén14] eine gute Option um Gelände zu generieren.

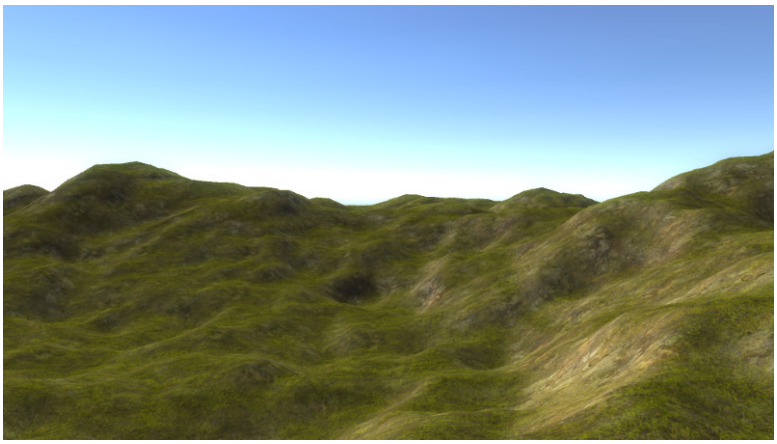


Abbildung 4.3.: Mit Perlin Noise generiertes Gelände [Kul15]

Simplex Noise

Simplex Noise ist eine Weiterentwicklung des Perlin Noise, welches natürlichere Ergebnisse liefern soll. Jedoch scheint das Simplex Noise in Reinform in der Praxis weniger weit verbreitet zu sein. Es wird anscheinend eher als Grundlage für Funktionen wie das Fractional Brownian Motion verwendet. Aufgrund dieser Tatsache wird davon abgesehen, Simplex Noise zum Generieren des Geländes zu nutzen.

Fractional Brownian Motion

Anschließend wurde Fractional Brownian Motion beleuchtet.

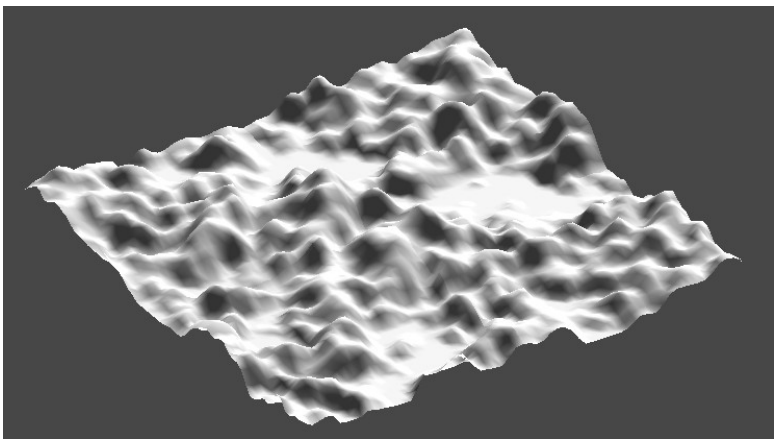


Abbildung 4.4.: Mit Fractional Brownian Motion generiertes Gelände mit H = 0.25, G = 0.7 und O = 7 [Knö14]

In Abbildung 4.4 ist ein mittels Fractional Brownian Motion generiertes Terrain mit einem Abstands-Wert von 0.7, einem fraktalen Inkrement von 0.25 und 7 Oktaven zu sehen. Die zugrunde liegende Basisfunktion ist das Simplex Noise. Zu sehen ist eine natürlichere Verteilung der Erhebungen als bei den zuvor genannten Noise Funktionen. Jedoch erscheint dieses natürlichere Aussehen in Betracht auf die wachsende Komplexität und die schwierigere Parametrisierung des Algorithmus nicht ausreichend um das Fractional Brownian Motion in Betracht zu ziehen [Knö14].

Hybrid-Multifractal-Algorithmus

Als Letztes wurde der Hybrid-Multifractal-Algorithmus untersucht.

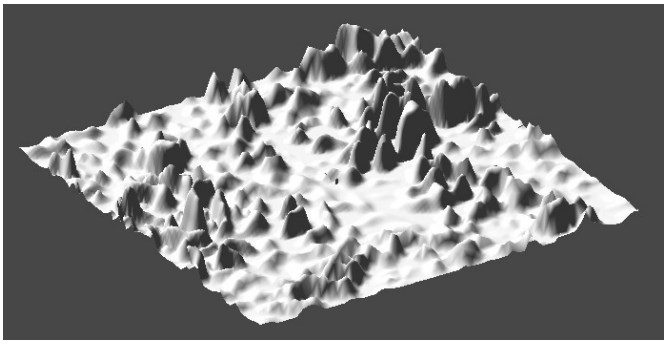


Abbildung 4.5.: Mit dem Hybrid-Multifractal-Algorithmus generiertes Gelände mit den folgenden Parametern: H = 0.25, G = 0.3 und O = 7 [Knö14]

Abbildung 4.6.: Mit dem Hybrid-Multifractal-Algorithmus generiertes Gelände [Mus]

Die Abbildung 4.5 zeigt den Algorithmus mit einem Simplex Noise als Basisfunktion und die Abbildung 4.6 den Algorithmus mit Worly's Voronoi Noise Funktion als Basis. Die Abbildung 4.5 zeigt sehr gut, dass sich die generierten Berge gruppieren. Dieses Verhalten gibt der Landschaft ein natürlicheres Aussehen. Auffällig ist jedoch auch, dass die Berge immer noch relativ runde Spitzen haben. Ändert man hingegen die Basisfunktion, wie in Abbildung 4.6, sieht das generierte Gelände sehr natürlich aus: es weist spitze Berge, Bergketten und ein flacheres und ebenmäßigeres Tal auf.
Der Hybrid-Multifractal-Algorithmus liefert die natürlichsten Ergebnisse. Allerdings ist auch hier wieder das Problem, dass zusätzlich auch eine Noise Funktion benötigt wird. Zudem ist die Parametrisierbarkeit wie beim Fractional Brownian Motion schwierig und daher zeitaufwendig. Da dies den zeitlichen Rahmen einer Bachelorarbeit übersteigen würde, ist von der Verwendung des Hybrid-Multifractal-Algorithmus abzusehen.

Auswahl

Alle oben genannten Algorithmen haben Vor- und Nachteile, jedoch kann die Verwendung von Zufallszahlen aufgrund des willkürlichen Ergebnisses ausgeschlossen werden. Nachfolgend werden kurz die Argumente für und gegen die einzelnen verbleibenden Algorithmen aufgezeigt.

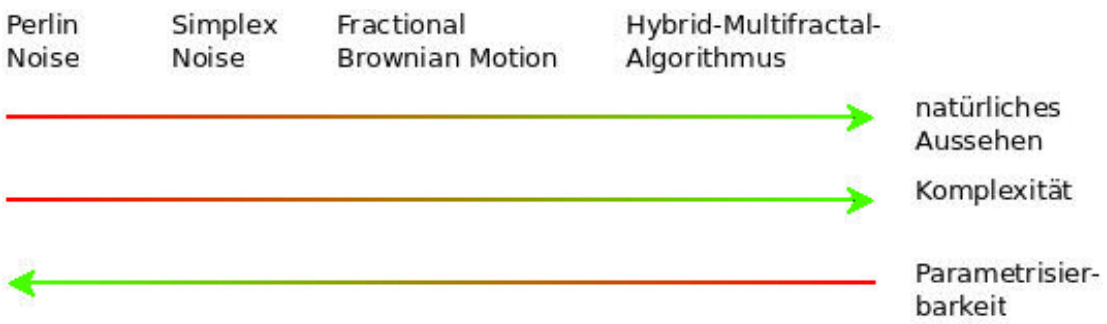


Abbildung 4.7.: Vergleich der Algorithmen zur Generierung von Terrain

In Abbildung 4.7 wurden die Algorithmen Perlin Noise, Simplex Noise, Fractional Brownian Motion und der Hybrid-Multifractal-Algorithmus in Bezug auf das natürliche Aussehen, die Komplexität des Algorithmus und die Parametrisierbarkeit verglichen. Es ist gut zu erkennen, dass das natürliche Aussehen aber auch die Komplexität steigen während die Parametrisierbarkeit sinkt. Obwohl der Hybrid-Multifractal-Algorithmus eine sehr gute Option für das Generieren von Terrain bietet, sofern ein sehr natürliches Ergebnis gewünscht wird, wird dieser aufgrund der hohen Komplexität und der schwierigen Parametrisierbarkeit nicht gewählt.

Simplex Noise und Fractional Brownian Motion bieten beide ein mittelmäßig natürliches Ergebnis bei mittelmäßiger Komplexität und Parametrisierbarkeit. Auch diese beiden wurden nicht gewählt. Der hier gewählte Algorithmus ist dementsprechend das Perlin Noise. Das Aussehen ist zwar vergleichsweise nicht sehr natürlich, allerdings ist es einfach zu parametrisieren und weist die geringste Komplexität auf. Zudem bietet es diverse Bibliotheken für die Verwendung.

4.4.2. Landschaftsgestaltung

Um ein möglichst realistisches Gelände zu generieren, ist es nötig, der Landschaft Vegetation, Gestein und Textur zu geben. Bei Vegetation handelt es sich in diesem Fall um Bäume, Sträucher und Gras. Gestein bezeichnet die Platzierung von kleineren und größeren Felsen. Es ist darauf zu achten, dass die Platzierung gewissen Regeln folgt:

- Es sollten keine Objekte auf einem Weg platziert werden.
- Ist das Gelände zu hoch beziehungsweise zu steil sollten dort keine Pflanzen mehr platziert werden. Felsen können auch bei hoher Steigung platziert werden.
- Bei Höhenunterschieden ist darauf zu achten, dass Objekte nicht zu hoch, also oberhalb des Bodens, platziert werden.
- Objekte sollten nicht ineinander platziert werden.
- Es ist eine geeignete Anzahl für die jeweiligen Typen zu bestimmen, um ausreichende Diversität zu bieten.

Dafür müssen geeignete Objekte erstellt werden. Entsprechend der Natur sollten es verschiedene Bäume, Sträucher und Felsen sein. Die Platzierung dieser Objekte erfolgt zufällig.

4.4.3. Wege

Auch für die Generierung von Wegen wurden vier Möglichkeiten genannt. Anders als bei der Generierung von Landschaften ist die Entscheidung, wie nachfolgend deutlich wird, hier einfacher zu treffen.

A*-Algorithmus

Der erste dieser Algorithmen, der A*-Algorithmus, hat einige Probleme, welche auch teilweise in Abbildung 4.8 deutlich werden. Die mit X markierten Felder stellen Hindernisse wie zum Beispiel Bäume dar. Ein blaues Kästchen entspricht fünf mal fünf Punkten.

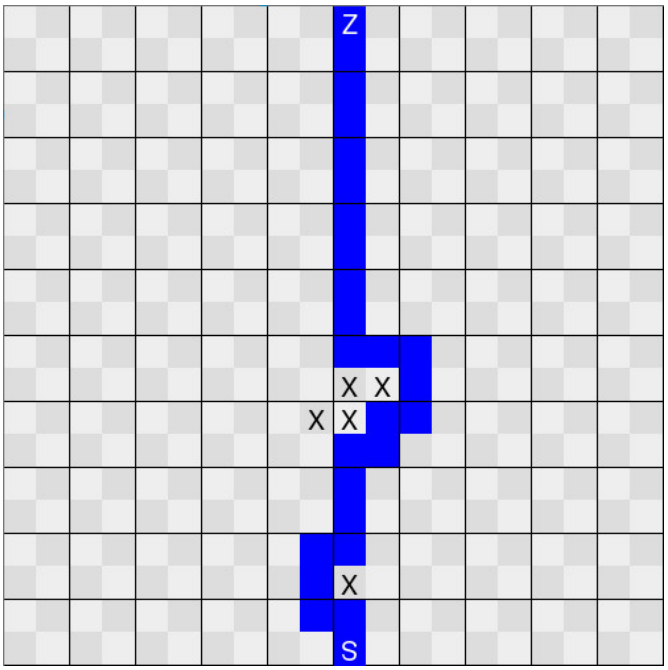


Abbildung 4.8.: Mit A*-Algorithmus generierter Weg

Der Algorithmus sucht den kürzesten Weg von einem Start- zu einem Endpunkt. Im, für das Level, schlechtesten Fall ist dies eine Gerade. Dies kann auch dann noch vorkommen, wenn bereits Vegetation platziert wurde, falls hierbei zufällig diese Grade unbesetzt blieb. Andererseits kann üppig platzierter Bewuchs eine derart umfangreiche Blockade darstellen, dass es keinen freien Weg mehr vom Start zum Ziel gibt. Ein weiteres Problem ist die Performance. Man nehme ein 100 Punkte breites und 100 Punkte langes Feld als Bodenfläche. Diese Fläche muss für den Algorithmus in Quadrate unterteilt werden. Bei zum Beispiel einer Kantenlänge von fünf Punkten beträgt die Anzahl der Quadrate:

$$\frac{100^2}{5^2} = 400$$

Damit hätte der Pfad allerdings eine minimale Breite von 5 Punkten. Das Ergebnis wäre eher eckig, der generierte Weg optisch nicht ansprechend und als Pfad in einer Landschaft ungeeignet. Um also einen weniger eckigen Pfad zu generieren müssten die Quadrate wesentlich kleiner sein. Dadurch steigt aber auch ihre Anzahl und somit sinkt die Performance.

Bézierkurven

Die Bézierkurven sind im Vergleich mit dem A*-Algorithmus besser für die Generierung von Wegen geeignet. Die Abbildung 4.9 zeigt die drei gängigen Typen von Bézierkurven. Für diesen Anwendungsfall sind nur quadratische oder kubische Bézierkurven geeignet.

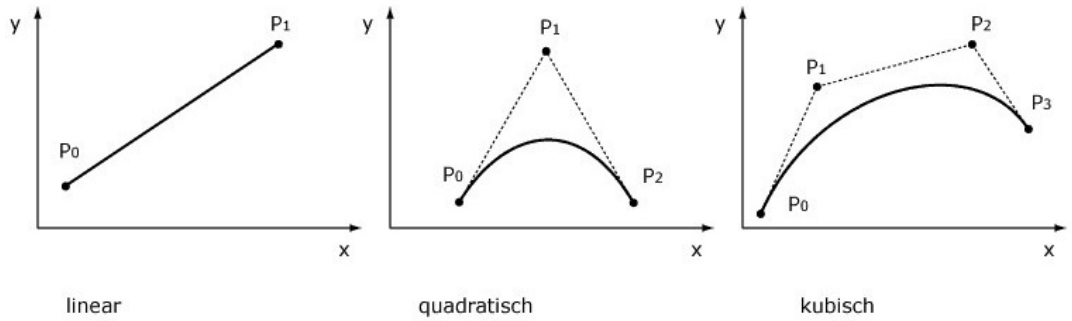


Abbildung 4.9.: Vergleich Bézierkurven [Hag06]

Der entscheidende Nachteil ist, dass die Kurve durch die Kontrollpunkte lediglich beeinflusst wird, sie aber nicht schneidet. Dies führt dazu, dass sie sich schwieriger platzieren lässt und daher unerwünschte Ergebnisse auftreten können. Daher sind auch Bézierkurven nicht ideal für die Generierung von Wegen geeignet.

Béziersplines

Béziersplines, die Unterteilung von komplexen Bézierkurven, eignen sich, wie in Abbildung 4.10 sichtbar, optisch gut für die Generierung von Wegen.

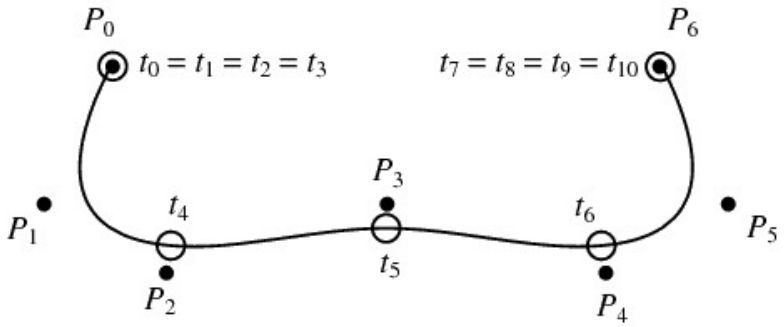


Abbildung 4.10.: Beispiel Bezierspline [Tea]

Sie haben aber den Nachteil, mit hohem Rechenaufwand und komplexen Formeln verbunden zu sein. Zudem bleibt das Problem, dass die Kurve die Kontrollpunkte nicht direkt schneidet, bestehen und macht das Ergebnis somit nicht berechenbarer.

Catmull-Rom Splines

Diese Art von Splines, die Catmull-Rom Splines, haben im Vergleich zu den Bézierkurven und Béziersplines den Vorteil, dass die Kurve die Kontrollpunkte schneidet und somit direkt Einfluss auf das Aussehen der Kurve genommen werden kann. Dies ist sehr gut in Abbildung 4.11 zu sehen.

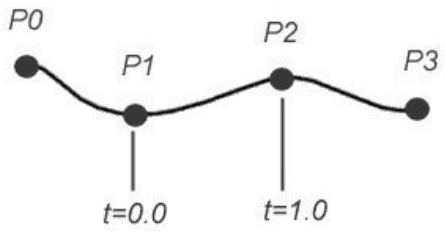


Abbildung 4.11.: Beispiel Catmull-Rom Spline [Dun05b]

Bei P_0 bis P_3 handelt es sich im Vektorgrößen. Um das Spline Segment zwischen P_1 und P_2 berechnen, wird folgende Formel mit t im Intervall von $[0, 1]$ angewendet und je Schleifendurchlauf um *step* erhöht: $q(t) = 0.5 * ((2 * P_1) + (-P_0 + P_2) * t + (2 * P_0 - 5 * P_1 + 4 * P_2 - P_3) * t^2 + (-P_0 + 3 * P_1 - 3 * P_2 + P_3) * t^3)$
Das Segment besteht aus $\frac{1}{step}$ Punkten. Um ein möglichst gleichmäßiges Ergebnis zu erzielen, sollte *step* sehr klein sein. Das Ergebnis ist optisch ansprechend und die Formel für die Generierung eines Segmentes relativ simpel, muss aber für ein geeignetes Ergebnis relativ häufig ausgeführt werden. Trotz des dadurch entstehenden Rechenaufwandes sind die Catmull-Rom Splines am besten für die Generierung geeignet.

4.4.4. Labyrinthe

In Kapitel 2.3.3 wurden einige Möglichkeiten genannt um Labyrinthe zu generieren und ihre Funktionsweise erläutert. In diesem Kapitel werden Beispiellabyrinthe der einzelnen Algorithmen gezeigt und analysiert. Es werden die Anzahl der toten Enden (weniger ist besser), die Anzahl der Zellen auf dem Lösungsweg (mehr ist besser), die Abbiegungen des Lösungsweges (mehr ist besser) und die möglichen Abzweigungen vom Lösungsweg (mehr ist besser), die keine offensichtlichen Sackgassen darstellen, betrachtet. Offensichtliche Sackgassen

sind solche, die keinerlei Abbiegungen haben und nicht tiefer als zwei Zellen sind.
Die roten Felder in den Beispielen stellen Sackgassen dar, der blaue Pfad ist der Weg vom Startpunkt (Mitte unten) zum Ziel (Mitte oben). Zu den Grafiken sei nachfolgende Legende gegeben:

Abbildung 4.12.: Legende für die Labyrinth-Analyse

Rekursive Generierung

Der erste zu untersuchende Algorithmus ist das komplett rekursive Generieren eines Labyrinthes.

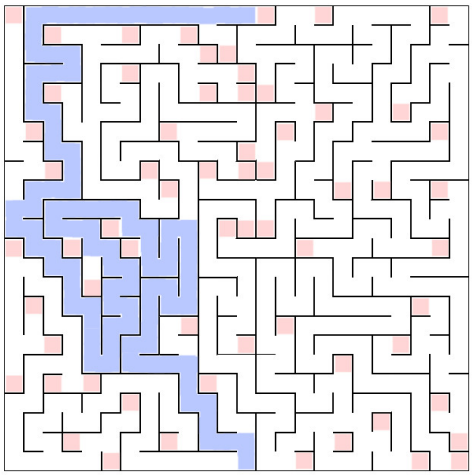


Abbildung 4.13.: Beispiel für ein komplett rekursiv erzeugtes Labyrinth

Anzahl Zellen auf dem Weg zum Ziel	16.3% (94)
Anzahl toter Enden	11% (64)
Abbiegungen des Lösungsweges	9.5% (55)
Mögliche Abzweigungen vom Lösungsweg	0.9% (5)

Tabelle 4.1.: Analyse eines rekursiv erzeugten Labyrinthes

Der Algorithmus ist in Bezug auf Laufzeit und Speicherbedarf durch die rekursiven Aufrufe für größere Labyrinthe nicht gut geeignet, erzeugt aber optisch ein ansprechendes Ergebnis. Betrachtet man die Analyse, wird deutlich, dass die Anzahl der toten Enden relativ gering und

die Anzahl der Zellen auf dem Lösungsweg gut ist. Es gibt jedoch nur wenige Abbiegungen und mögliche Abzweigungen, weshalb die Komplexität vergleichsweise eher gering ausfällt.

Recursive Division

Als nächstes wird der Recursive Division Algorithmus beleuchtet.

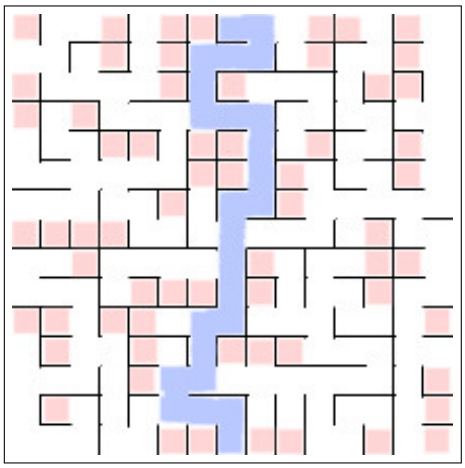


Abbildung 4.14.: Komplexeres Beispiel zum Recursive Division Algorithmus [Buc11d]

Anzahl Zellen auf dem Weg zum Ziel	11% (25)
Anzahl toter Enden	28.4% (64)
Abbiegungen des Lösungsweges	5.8% (13)
Mögliche Abzweigungen vom Lösungsweg	3.6% (8)

Tabelle 4.2.: Analyse eines mit Recursive Divison erzeugten Labyrinthes

Ein Nachteil des Recursive Division Algorithmus ist, dass es, wie auch in Abbildung 4.14 und Tabelle 4.2 gut zu erkennen, relativ viele Sackgassen und verhältnismäßig lange Wände gibt. Dies lässt das Labyrinth wenig ansprechend wirken. Zudem kann bei nicht quadratischen Feldern die erste Trennung des Feldes unglücklich erfolgen und erfordert so ein Verändern des Algorithmus dahingehend, dass beispielsweise ein Feld das breiter ist als hoch öfter durch vertikale Wände geschnitten wird. Unabhängig davon ist dieser Algorithmus relativ schnell da er in der Lage ist mehrere Zellen gleichzeitig fertig zu stellen. Das erzeugte Labyrinth ist für den Spieler schwierig zu lösen, da es viele mögliche Abzweigungen vom Lösungsweg gibt [Buc11d][Pul15].

Recursive Backtracking

Ein weiterer zu untersuchender Algorithmus ist das Recursive Backtracking.

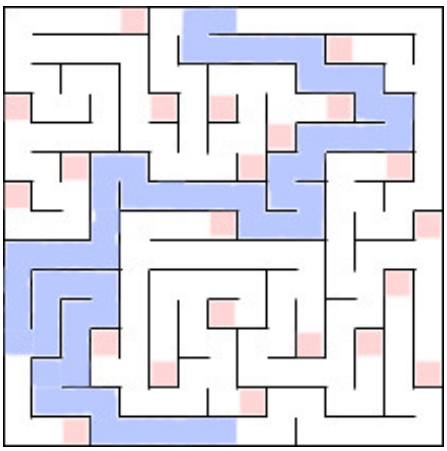


Abbildung 4.15.: Komplexeres Beispiel zum Recursive Backtracking Algorithmus [Buc10]

Anzahl Zellen auf dem Weg zum Ziel	25.3% (57)
Anzahl toter Enden	9.8% (22)
Abbiegungen des Lösungsweges	14.2% (32)
Mögliche Abzweigungen vom Lösungsweg	1.3% (3)

Tabelle 4.3.: Analyse eines mit Recursive Backtracking erzeugten Labyrinthes

Das in Abbildung 4.15 generierte Labyrinth ist auf den ersten Blick sehr ansprechend. Dies ist vor allem durch den sehr langen Lösungsweg und die wenigen toten Enden begründet. Beachtet man jedoch die Analyse des Labyrinthes in Tabelle 4.3 fällt auf, dass es lediglich drei mögliche Abzweigungen von dem Lösungsweg gibt. Vom Startpunkt aus hat der Spieler nur eine Entscheidung zwischen links und rechts, wobei der rechte Weg sehr schnell in eine Sackgasse führt. Dies ist bei den anderen zwei Abzweigungen nicht anders. Dadurch ist die Komplexität des Labyrinthes trotz des langen Lösungsweges eher gering und das Level schnell gelöst. Die Laufzeit des Algorithmus ist dafür sehr gering [Pul15].

Binary Tree

Die folgenden Beispiele zeigen den Binary Tree Algorithmus für die vier möglichen Ausrichtungskombinationen Nord/West, Nord/Ost, Süd/West und Süd/Ost. Analysiert werden hingegen nur die gemittelten Werte aller vier Beispiele. Der Binary Tree Algorithmus zeigt

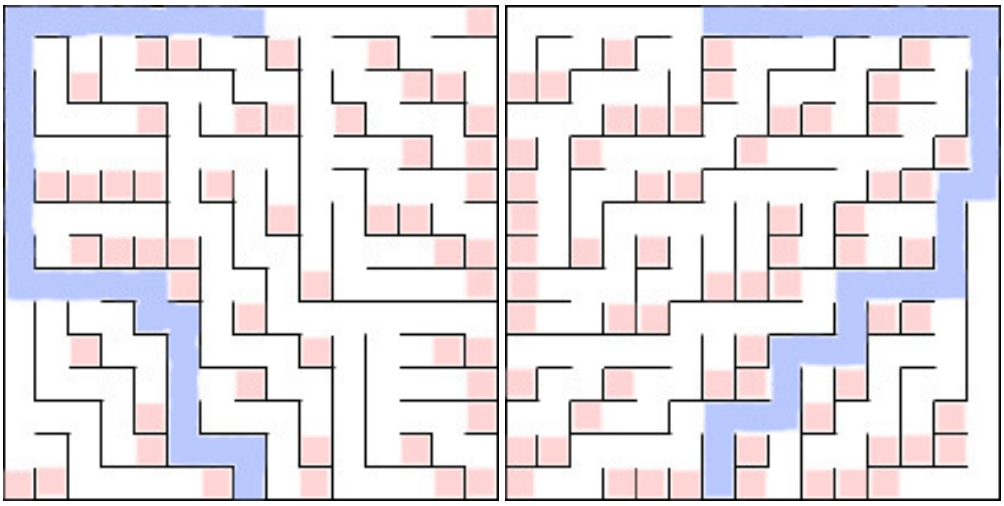


Abbildung 4.16.: Komplexere Beispiele des Binary Tree Algorithmus für die Ausrichtungen Nord-West (links) und Nord-Ost (rechts) [Buc11a]

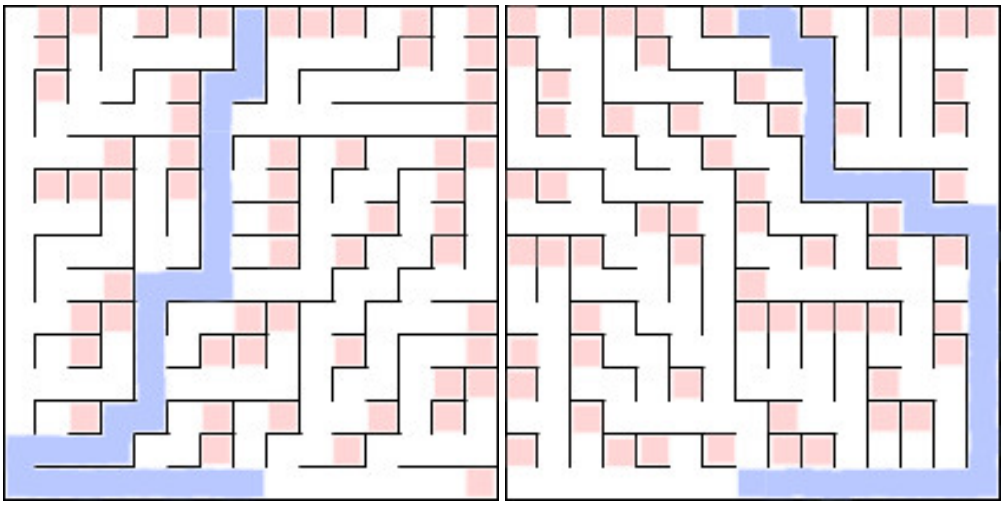


Abbildung 4.17.: Komplexere Beispiele des Binary Tree Algorithmus für die Ausrichtungen Süd-West (links) und Süd-Ost (rechts) [Buc11a]

Werte gemittelt	
Anzahl Zellen auf dem Weg zum Ziel	13.1% (29.5)
Anzahl toter Enden	25.6% (57.5)
Abbiegungen des Lösungsweges	3.7% (8.3)
Mögliche Abzweigungen vom Lösungsweg	4% (9)

Tabelle 4.4.: Analyse von mittels Binary Tree erzeugten Labyrinthen

im Vergleich zu den anderen Algorithmen eine deutliche Anisotropie. Dies spiegelt sich darin wider, dass alle toten Enden immer in die beiden jeweiligen Ausrichtungen zeigen. Zudem ist auffällig, dass bei allen vier erzeugten Labyrinthen in den Abbildungen 4.16 und 4.17 jeweils eine Zeile und Spalte von einem Eckpunkt aus komplett passierbar sind. In allen vier Fällen wurde diese freie Fläche für den Lösungsweg genutzt. Dadurch sinkt die Komplexität erheblich. Diese Struktur führt dazu, dass die Navigation von beispielsweise unten rechts nach oben links einfacher ist als eine Kombination aus anderen möglichen Ecken. Ein Vorteil dieses Algorithmus ist jedoch, dass er einer der schnellsten und einfachsten ist. Zudem ist der Speicherbedarf sehr gering und daher die mögliche Labyrinthgröße nicht begrenzt [Buc11a][Pul15].

Kruskal's Algorithmus

Der vorletzte untersuchte Algorithmus ist Kruskal's Algorithmus in der zufälligen Variante.

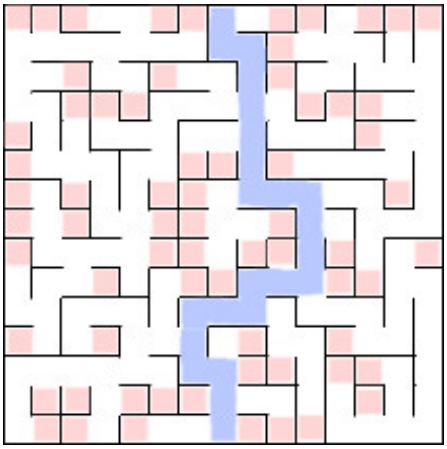


Abbildung 4.18.: Komplexeres Beispiel zum zufälligen Algorithmus von Kruskal [Buc11b]

Anzahl Zellen auf dem Weg zum Ziel	10.2% (23)
Anzahl toter Enden	30.2% (68)
Abbiegungen des Lösungsweges	4% (9)
Mögliche Abzweigungen vom Lösungsweg	5.3% (12)

Tabelle 4.5.: Analyse eines mit Kruskal's Algorithmus erzeugten Labyrinthes

Obwohl der Lösungsweg in dem generierten Labyrinth in Abbildung 4.18 auf ersten Blick eher einfach aussieht, besitzt er einige Abbiegungen und sehr viele mögliche Abzweigungen, so dass das Ziel schwierig zu erreichen sein kann. Der Algorithmus erzeugt allerdings vergleichsweise viele tote Enden [Buc11b][Pul15].

Prim's Algorithmus

Zuletzt ist der Algorithmus von Prim zu untersuchen.

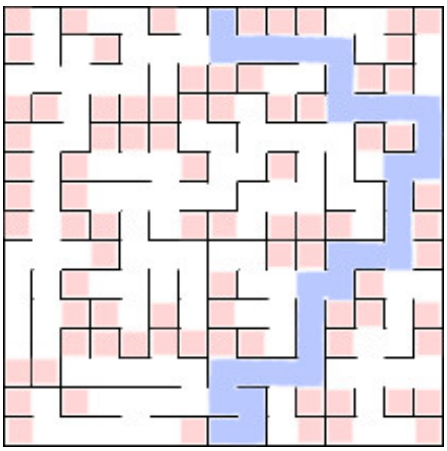


Abbildung 4.19.: Komplexeres Beispiel zu Prim's Algorithmus [Buc11c]

Anzahl Zellen auf dem Weg zum Ziel	13.8% (31)
Anzahl toter Enden	36.9% (83)
Abbiegungen des Lösungsweges	7.1% (16)
Mögliche Abzweigungen vom Lösungsweg	4.4% (10)

Tabelle 4.6.: Analyse eines mit Prim's Algorithmus erzeugten Labyrinthes

Prim's Algorithmus gehört zu den eher langsamen Algorithmen und generiert Labyrinthe, die denen mit Kruskal's Algorithmus erzeugten sehr ähneln. In diesem Fall ist das mit Prim erzeugte Labyrinth etwas besser als das mit Kruskal erzeugte, da es mehr Abbiegungen und

mögliche Abzweigungen bietet und der Lösungsweg zudem länger ist. Jedoch ist die Anzahl der toten Enden nochmals deutlich höher (etwa 22% mehr). Trotz dieser Tatsache erzeugt der Algorithmus ein komplexes Labyrinth [Buc11c][Pul15].

Vergleich

Die unten stehende Tabelle 4.7 vergleicht die oben für die einzelnen Algorithmen analysierten Labyrinthe.

	Rekursive Generierung	Recursive Division	Recursive Backtracking	Binary Tree (gemittelt)	Kruskal's Algorithmus	Prim's Algorithmus
Anzahl Zellen auf dem Weg zum Ziel	16.3% (94)	11% (25)	25.3% (57)	13.1% (29.5)	10.2% (23)	13.8% (31)
Anzahl toter Enden	11% (64)	28.4% (64)	9.8% (22)	25.6% (57.5)	30.2% (68)	36.9% (83)
Abbiegungen des Lösungsweges	9.5% (55)	5.8% (13)	14.2% (32)	3.7% (8.3)	4% (9)	7.1% (16)
Mögliche Abzweigungen vom Lösungsweg	0.9% (5)	3.6% (8)	1.3% (3)	4% (9)	5.3% (12)	4.4% (10)

Tabelle 4.7.: Bewertung der Algorithmen für die Generierung von Labyrinthen

In Bezug auf den längsten Lösungsweg, die wenigsten toten Enden und die Anzahl der Abbiegungen des Lösungsweges schneidet das Recursive Backtracking mit 25.3%, 9.8% und 14.2% am Besten ab. Nachfolger ist in allen diesen drei Punkten der rekursive Algorithmus mit 16.3%, 11% und 9.5%. Bei der Anzahl der möglichen Abzweigungen vom Lösungsweg liegen jedoch Kruskal's Algorithmus in der zufälligen Variante mit 5.3% und Prim's Algorithmus mit 4.4% vorne. Recursive Backtracking und Binary Tree sind in allen Punkten eher im Mittelfeld angesiedelt. Sucht man also einen komplexen Algorithmus wird der rekursive Algorithmus in Betracht gezogen. Denn trotz des schlechten Abschneidens in Bezug auf die möglichen Abzweigungen vom Lösungsweg werden die Labyrinthe für den Spieler schwierig zu lösen sein.

4.5. Integration in vorhandene Systeme

Die zu entwickelnde Software ist in das bereits bestehende System des EmotionBike zu integrieren. Dies kann problemlos erfolgen. Szenen müssen einen Start- und Endpunkt implementieren,

sich im entsprechenden Projekt mit allen nötigen Assets und Scripten befinden und können dann einfach gestartet werden. Der Vorteil dieser Integration liegt darin, dass bereits als Prefab verwendbare Objekte und Scripte für die Steuerung und den Spieler vorhanden sind. Zudem bietet das System bereits Scripte für plötzlich eintretende Aktionen. Dementsprechend ist die Integration als positiv und unkompliziert zu betrachten.

5. Implementierung

In diesem Kapitel wird sich mit der Implementierung auseinandergesetzt. Zunächst wird genannt, mit welcher Entwicklungsumgebung das Projekt realisiert wird, welche Architektur es aufweist und welche Bibliotheken genutzt werden. Zuletzt wird die konkrete Implementierung einzelner Komponenten beschrieben.

5.1. Unity

Da im EmotionBike Projekt für die bestehenden Level Unity[Tec16a] verwendet wird, wird auch für diese Arbeit Unity verwendet.
Bei Unity handelt es sich um eine Entwicklungsumgebung für 2D und 3D Spiele. Mit Unity ist es möglich, Spiele für verschiedene Plattformen zu entwickeln und sogar virtuelle Realität zu verwenden. Von Haus aus stellt Unity Systeme für beispielsweise Grafik, Physik, Audio, Steuerung und Scripting zur Verfügung. Des Weiteren besteht die Möglichkeit eine Vielzahl von Assets herunterzuladen. Assets sind Zusatzinhalte wie zum Beispiel Animationen, Texturen, 3D-Modelle oder auch Scripte. Ein weiterer großer Vorteil von Unity ist die Tatsache, dass Spielparameter auch während der Laufzeit angepasst werden können [Chi15][Twe][Win]. Das Scripting basiert auf dem Mono-Framework und stellt damit eine, dem .NET-Framework ähnliche, Bibliothek zur Verfügung. Als Sprachen stehen C#, JavaScript, beziehungsweise UnityScript, und Boo zur Verfügung. Unity ist in der Standardversion, welche praktisch alle Funktionen enthält, kostenfrei. Wird Unity in professionellem Umfang genutzt und setzt das anwendende Unternehmen mehr als 100.000$ jährlich um, fallen Lizenzgebühren an [Chi15].
In dieser Arbeit wird Unity in Version 5.3.4 mit C# unter Windows 8.1 genutzt.

5.2. Architektur

In Abbildung 5.1 werden die Komponenten des Systems dargestellt und nachfolgend kurz beschrieben.

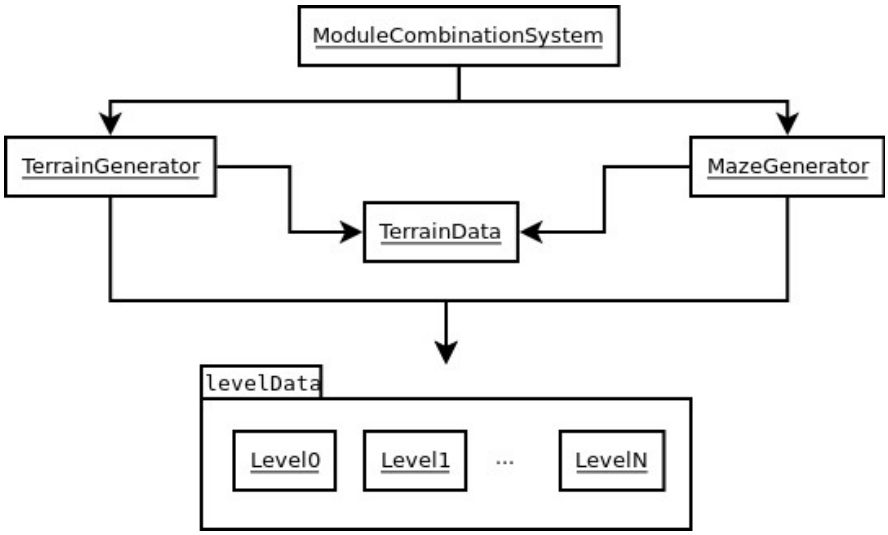


Abbildung 5.1.: Übersicht über die Architektur des Systems

ModuleCombinationSystem

Das *ModuleCombinationSystem* ist derjenige Teil des Systems, welcher für die Kombination der Levelbausteine genutzt wird. Er implementiert somit das Baukastensystem. Innerhalb dessen wird entschieden, welche Emotion als nächstes dargestellt werden soll. Dementsprechend ist das *ModuleCombinationSystem* direkt mit den Generatoren verbunden. Wurde eine geeignete Emotion ausgewählt, wird der entsprechende Generator angesteuert um ein Level zu generieren.

TerrainGenerator

Der *TerrainGenerator* wird dazu genutzt, landschaftliche Level zu generieren und bereitzustellen. Dieser bekommt mitgeteilt, welche Emotion er darstellen soll und greift auf Basis dessen auf die entsprechende *LevelData* und auf die *TerrainData* zu.

MazeGenerator

Der *MazeGenerator* generiert Level mit Labyrinthen und stellt diese anschließend zur Verfügung. Auch dieser greift auf *TerrainData* und *LevelData* zu.

TerrainData

Bei *TerrainData* handelt es sich um eine XML Datei, welche allgemeine Informationen zu den Bausteinen enthält. Diese Informationen sind für alle Bausteine strukturgleich. Dabei handelt es sich beispielsweise um die Größe des Levels und die Position des Levelein- und -ausganges.

LevelData

Im Ordner *LevelData* befinden sich die XML-Dateien um einzelne Szenen zu parametrisieren. Ein Level bedient eine bestimmte Emotion. In einer Level-Datei, wie zum Beispiel *Level0*, befinden sich Informationen wie die Pfadkomplexität oder die Bergigkeit des Geländes.

Der Aufbau des Systems hat folgenden Vorteil: Will man einen neuen Generator hinzufügen, muss hierzu nur das *ModuleCombinationSystem* angepasst werden. Soll lediglich ein neues Level hinzugefügt werden, welches durch die bereits vorhandenen Generatoren abgedeckt wird, so erstellt man die entsprechende Level-Datei im *levelData*-Ordner und macht anschließend dieses neue Level im *ModuleCombinationSystem* bekannt. Eine Anpassung in den Generatoren ist nicht erforderlich.

Ein Klassendiagramm und die ausführliche Auflistung aller Klassen sind im Anhang unter A.2 und A.3 zu finden.

5.3. Bibliotheken

Für die Generierung der Heightmap des Geländes wurde die open-source Bibliothek *libNoise* benutzt. *libNoise* ist von Jason Bevins in und für C++ entwickelt worden. Mit der portablen Bibliothek ist es möglich kohärentes Rauschen, wie beispielsweise Perlin Noise, zu erzeugen. Die einzelnen Noise Funktionen sind jeweils in Module gekapselt, wobei die Option besteht Module zu kombinieren [Bev07]. Ein großer Vorteil von *libNoise* ist die gute und leicht verständliche Dokumentation.
Da die Bibliothek jedoch nur für C++ geeignet ist, kann sie in ihrer ursprünglichen Form nicht genutzt werden. Ricardo J. Méndez hat die oben genannte Bibliothek beziehungsweise LibNoise.Xna für Unity portiert und das Ergebnis unter LGPL Lizenz unter dem Namen LibNoise.Unity zur Verfügung gestellt [Mén14]. LibNoise.Unity kommt leider ohne jegliche eigene Dokumentation und erfordert daher eine gewisse Einarbeitung. Die Grundfunktionen der Noise Funktionen können jedoch aus der Dokumentation von *libNoise* entnommen werden. Die Verwendung dieser Bibliothek wird in Kapitel 5.7 beschrieben.

5.4. Assets

Unity bietet den großen Vorteil, dass zusätzliche Inhalte aus dem Asset Store [Tec16b] heruntergeladen werden können. Konkret bedeutet das, dass nicht alle Elemente dieses Projektes

selbst entwickelt werden müssen. Nachfolgend wird eine Übersicht der wichtigsten Assets gegeben. Eine Auflistung aller genutzten Assets erfolgt im Anhang unter A.4.

Vegetation

Eines der wichtigsten Elemente von Landschaften ist Vegetation. Diese muss in Bezug auf Diversität und Herkunft zum entsprechenden Level passen. Eine Landschaft wirkt kaum natürlich entstanden, wenn es nur eine Sorte Bäume gibt. Dementsprechend wurden hier die meisten Assets zu Hilfe gezogen. In Summe wurden zwölf Assets genutzt. Davon acht für normale Landschaften, zwei für Wüstenlandschaften und zwei für beides. Die meisten Objekte, in Summe neun, stellen dabei Bäume dar. Assets die Büsche oder beides enthalten sind nur eins beziehungsweise zwei.

Gestein

Steine sollen dazu genutzt werden, um den Levelein- und -ausgang zu verstecken und um das Level zu begrenzen. Zusätzlich stellt Gestein ein dekorierendes Element in den Leveln dar. Dazu wurden neun verschiedene Assets mit sehr unterschiedlichen Steinen sowohl in Bezug auf Größe als auch Form und Farbe herausgesucht.

Texturen

Da Texturen von großer Bedeutung sind, werden hier sehr viele Assets verwendet. Diese sind nötig, um den generierten Leveln ein natürliches Aussehen zu geben. Besonders hilfreich sind dabei die Texture-Packs von Yughues beziehungsweise Nobiax. Diese sind für alle möglichen Anwendungsfälle und Themen verfügbar. Für waldartige Landschaften wurden fünf Assets mit diversen Texturen für den Boden ausgewählt, wie beispielsweise diverse Grasarten oder Blätter auf dem Boden. Für die Wüste hingegen wurde lediglich ein Asset mit verschiedenen Sand-Arten ausgewählt. Mehr Vielfalt ist wiederum bei den Labyrinthen vorhanden, dort wurden fünf Assets mit verschiedenen Gesteinsarten und Bodenbelägen herangezogen.

Strukturen

Bei Strukturen handelt es sich beispielsweise um Gebäude oder Ähnliches. Diese werden eher weniger verwendet könnten aber im Rahmen eines Wüsten-Themas in Form von Ruinen oder Brunnen vorkommen.

Skybox

Um eine geeignete Atmosphäre zu erzeugen, wird eine sogenannte Skybox verwendet. Bei einer Skybox handelt es sich um eine quasi unendlich große texturierte Kugel beziehungsweise

einen Würfel (eine sogenannte Cubemap) welche/n man von innen sieht und welche/r als Himmel für das Level fungiert [Chi15]. Für dieses Projekt werden zwei verschiedene Skyboxen verwendet: ein Himmel für Level am Tage und ein Abendhimmel.

Labyrinthe

Für die Generierung von Labyrinthen wird lediglich ein Asset als Vorlage genutzt und angepasst. Bei dem entsprechenden Asset handelt es sich um den Maze Generator von styanton [sty15b], mit welchem folgende Arten von Labyrinthen generiert werden können: Rekursive Generierung, Recursive Disivion, Newest Tree beziehungsweise Recursive Tree, Random Tree und Oldest Tree. Das Asset ist ursprünglich dazu gedacht, Spiele zu generieren, bei welchem ein Spieler, repräsentiert durch einen kleinen Ball, Münzen einsammelt, welche über das ganze Labyrinth verteilt sind. Die Labyrinthe werden aus Vorlagen, sogenannten Prefabs, für Bodenplatten, Wänden und Pfeilern zwischen den Wänden generiert.

5.5. Domänenspezifische Sprache

Die entwickelte domänenspezifische Sprache wurde so entwickelt, dass alle Parameter einen Integer-Wert darstellen. Dies hat den Vorteil, dass es keine Probleme durch Groß- und Kleinschreibung beziehungsweise Rechtschreibfehler gibt. Zudem lässt sich das Programm so in unterschiedlichen Sprachen verwenden, da lediglich die Kommentare innerhalb der XML-Dateien angepasst werden müssen und nicht die Auswertung der DSL im Baukastensystem beziehungsweise im entsprechenden Generator. Eine andere gut geeignete Option um sicherzustellen, dass die Werte in der XML-Datei korrekt sind und innerhalb des erwarteten Rahmens liegen, ist die Verwendung einer Dokumententypdefinition (**D**ocument **T**ype **D**efinition, kurz DTD) [w3s]. Dies kommt derzeit allerdings nicht zur Anwendung.
Die DSL ist in mehrere Dateien unterteilt: *TerrainData.xml* einerseits und spezifische Dateien für jedes Level andererseits. In *TerrainData.xml* werden allgemeine Parameter, welche für alle Szenen gelten, konfiguriert. Dabei handelt es sich konkret um Folgendes.

- ***terrainSize:*** Bestimmt die Größe des Terrains. Dieser Wert definiert die Höhe und die Breite der Geländekarte. Standardmäßig ist dieser Wert 100. Es ist zu beachten, dass größere Level auch mehr Rechenaufwand und -zeit erfordern.
- ***cellSize:*** Größe einer Zelle auf dem Gelände, beispielsweise für das Generieren des Weges nötig. Die Breite des Weges entspricht diesem Wert. Der Standardwert beträgt 5, möglich sind allerdings Werte zwischen 2 und 10. Kleinere Werte erhöhen die Rechenzeit, größere Werte sorgen für eckigere Wege.

- ***vegetation:*** Damit wird bestimmt, wie viel Vegetation das Terrain aufweisen soll. Gültige Werte befinden sich zwischen 0 und 100. Standardmäßig liegt dieser Wert bei 5.
- ***rocks:*** Werte zwischen 0 und 100 bestimmen die Anzahl der Steine auf dem Terrain. Je höher der Wert, desto mehr Steine sind vorhanden. Auch dieser Wert liegt im Normalfall bei 5.
- ***transitionPosition:*** Hiermit wird bestimmt, an welcher Position sich der Leveleinbeziehungsweise -ausgang befinden. Ein- und Ausgang befinden sich auf der selben Höhe. Dieser Wert liegt für gewöhnlich bei der Hälfte der Größe des Terrains, also 50.

Die Dateien für jedes Level besitzen Werte, die das jeweilige Level betreffen. Dabei handelt es sich um folgendes.

- ***terrainSlope:*** Dieser Wert bestimmt, wie anstrengend beziehungsweise wie bergig das Gelände sein soll. Zulässige Werte liegen zwischen 0 und 100, Standard bei 0.
- ***pathComplexity:*** Mit der *pathComplexity* wird bestimmt, wie komplex, also wie gewunden, der Weg ist. Auch hier sind Werte zwischen 0 und 100 zulässig.
- ***labCellSize:*** Dieser Wert kommt nur bei der Verwendung von Labyrinthen zum Einsatz und bestimmt, welche Kantenlänge eine quadratische Labyrinth-Zelle aufweist.

Diese DSL wird in den Unity Dateien ausgelesen und mittels eines XML-Interpreters ausgewertet. Die Werte werden dann zur Generierung des Geländes genutzt.
Nachfolgend wird in Listing 5.1 die generelle Struktur der XML Dateien beispielhaft aufgezeigt.

```
<terrainData>
        <!-- Groesse des Terrain
        values:
                numerischer Wert zwischen 100 und 500
        default:
                100
        -->
        <terrainSize>
        100
        </terrainSize>
        ...
</terrainData>
<moduleTransition>
        <!-- Bestimmt an welcher Stelle ein Uebergang des Weges
```

```
        zwischen den Bausteinen stattfindet (in x Richtung)
        values:
                Wert zwischen 0.0 und terrainSize
        default:
                terrainSize / 2
        -->
        <transitionPosition>
        50
        </transitionPosition>
</moduleTransition>
```

Listing 5.1: Struktur der XML-Dateien

5.6. Baukastensystem

Wie bereits in Kapitel 4.1 beschrieben gibt es drei Möglichkeiten das Baukastensystem zu realisieren. Aufgrund der bereits genannten Vorteile, beziehungsweise der ausbleibenden Nachteile, im Vergleich zu den anderen Methoden wird hier die Verwendung von Objekten zum Verbinden der Bausteine gewählt. Bei diesen Objekten handelt es sich um Tunnel, die sich sowohl am Levelein- als auch -ausgang befinden. Am Anfang des Spiels wird der Spieler in einem Leveleingangstunnel gespawnt. Aus diesem heraus fährt man in das erste generierte Level.

Abbildung 5.2.: Leveleingangstunnel mit Blick auf das Level

Ziel eines jeden Levels ist, den Levelausgangstunnel zu erreichen. Ist dies erreicht, gilt das Level als beendet und kann vom Spieler aus dem Tunnel heraus nicht mehr betreten werden.

Der Tunnel wird dann genutzt, um die Zeit bis zum nächsten Level zu überbrücken. Das heißt, dass der Spieler in diesem Tunnel fährt, bis das nächste Level fertig generiert wurde. Danach ist der Tunnelausgang zugleich der Eingang für das neue Level.

ID	Emotion
0	relaxed
1	exhausting
2	focused
3	scared
4	exciting
5	playful

Tabelle 5.1.: Übersicht über die Emotionen im Baukastensystem

Die Tabelle 5.1 zeigt die Liste der derzeit möglichen Emotionen. Die Tabelle besteht aus der eindeutigen ID und der entsprechenden Emotion.

ID	Emotion	Type (/ Szene)	Atmosphere
0	0	0	0
1	1	0	0
2	1	1	1
3	2	0	0
4	2	1	1
5	3	1	1
6	4	0	0
7	4	1	1
8	5	0	0

Legende

	Type	Atmosphere
0	Landschaft	Wald
1	Labyrinth	Höhle

Tabelle 5.2.: Übersicht über die Level im Baukastensystem

Obenstehende Tabelle 5.2 zeigt die Liste der derzeit generierbaren Level. Sie besteht aus Folgendem: Die erste Spalte zeigt dabei die eindeutige ID jedes Levels. In der Spalte *Emotion* geht es um die darzustellende Emotion. Eine Emotion kann von mehreren Leveln bedient werden. Durch den *Type* wird konkretisiert, mit welcher Szene eine Emotion dargestellt werden soll. Folgende Typen sind derzeit vorhanden: 0 stellt eine landschaftliche Szene, wie beispielsweise einen Wald, dar, 1 hingegen ein Labyrinth. Auch hier ist es möglich, dass mehrere Level mit der

selben *Emotion* den selben *Type* haben. Die letzte Unterscheidung erfolgt über die *Atmosphere*. Diese bestimmt welche Stimmung das Level haben soll. Hier gibt es in Summe derzeit nur zwei Möglichkeiten: eine Waldlandschaft (0) und ein eher düsteres höhlenartiges Labyrinth (1). Dieses Feld wurde eingeführt um beispielsweise Wüsten oder freundlichere Labyrinthe darstellen zu können.
Das Baukastensystem funktioniert wie folgt: Es hält eine Liste aller möglichen Level, gezeigt in Tabelle 5.2. Am Anfang wird ein Level aus der oben genannten Liste zufällig ausgewählt. Tabelle 5.3 zeigt welche anschließenden Level möglich sind. Dabei wird unterschieden, ob die derzeitige Emotion verstärkt oder ausgeglichen werden soll. Kann die Emotion des Spielers nicht gefunden werden beziehungsweise derzeit nicht dargestellt und entsprechend nicht ausgeglichen oder verstärkt werden, wird wieder Level 0 dargestellt um den Spieler "zurückzusetzen". Zeigt der Spieler Anzeichen von Langeweile wird die Emotion ausgeglichen.

Level		**Ausgleichend**	**Verstärkend**
0		1, 2, 3, 4, 5, 6, 7	*0*, 8
1, 2		0, 8	*1*, *2*, 5
3, 4		0, 5, 6, 7, 8	*3*, *4*
5	$->$	0, 8	1, 2, (3, 4), *5*, 6, 7
6, 7		0, 3, 4, 5	*6*, *7*, 8
8		0, 1, 2, 5	6, 7, *11*

Tabelle 5.3.: Übersicht der Levelkombinationen

Wird beispielsweise als erstes Level ein entspanntes ausgewählt, kann, um die Emotion zu verstärken, wieder ein entspanntes Level generiert werden oder auch ein spaßiges. Soll eine ausgleichende Emotion erzeugt werden, sind zum Beispiel Anstrengung oder Angst denkbar. Diese nachfolgenden Level werden aus den aufgelisteten Folgeleveln zufällig ausgewählt.

Um die Funktionsweise des Bauskastensystems zu verdeutlichen, wird in Abbildung 5.3 beispielhaft ein Sequenzdiagramm für einen konkreten Fall gezeigt: Das Diagramm zeigt das Verhalten des Systems nachdem bereits ein Level geladen wurde und ein neues nötig wird. In diesem Fall ist Level 0 derzeit aktiv.

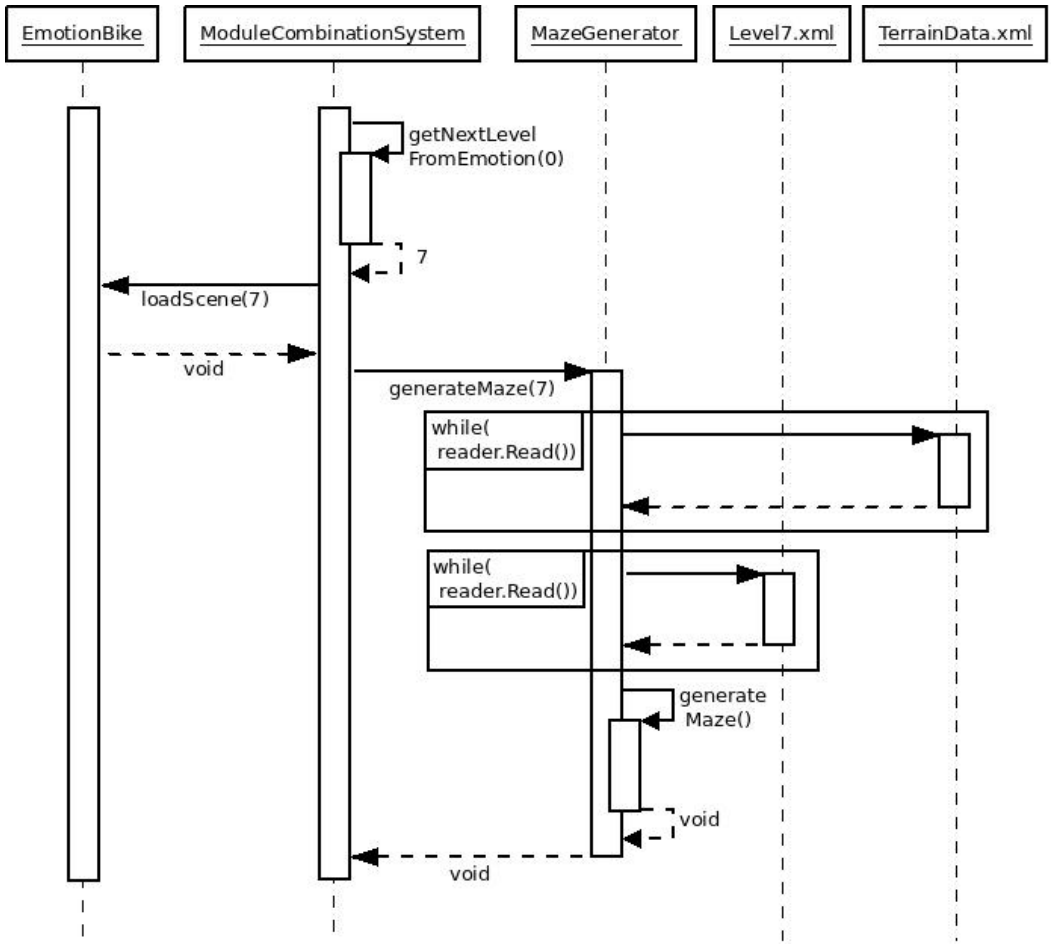


Abbildung 5.3.: Sequenzdiagramm für die Generierung eines Labyrinth

Konkret passiert in Abbildung 5.3 folgendes: Es wurde Level 0 generiert und nun soll ein ausgleichendes Level generiert werden. Das ModuleCombinationSystem bestimmt zufällig aus den in Tabelle 5.3 gezeigten Kombinationsmöglichkeiten ein geeignetes Folgelevel. In diesem Fall ist dies Level 7 für die Emotion 4 “exciting”. Anschließend wird das entsprechende, noch leere Level geladen und der Generator, in diesem Fall der MazeGenerator, mit der Generierung eines Levels für die gewählte Emotion beauftragt. Der Generator liest die nötigen Parameter aus den XML-Dateien aus und generiert das Level.

5.7. Generierung des Geländes

Nachfolgend wird beschrieben, wie einzelne Teile des Geländes generiert werden.

Gelände

Zunächst wird das Gelände generiert und geglättet. Dazu wird die in Kapitel 5.3 genannte Bibliothek LibNoise.Unity verwendet. Es wird Perlin Noise mit einem zufälligen Seed zwischen eins und tausend und acht Oktaven verwendet. Dies sorgt für den nötigen Detailgrad und die Diversität der generierten Level.

```
1 Perlin perlinNoiseGenerator = new Perlin();
2 perlinNoiseGenerator.Seed = UnityEngine.Random.Range(1, 1000);
3 perlinNoiseGenerator.OctaveCount = 8;
4 heightMap[(int)xRes, (int)zRes] = (float)
5         (perlinNoiseGenerator.GetValue(x, 0, z)) / factor + 0.5f;
```

Listing 5.2: Perlin Noise

In Listing 5.2 ist ein Ausschnitt der Verwendung der Bibliothek zu sehen. Es wird zunächst das Rauschen in Zeilen eins bis drei entsprechend initialisiert. In Zeile vier wird diese anschließend verwendet. Dazwischen muss die Heightmap geeignet initialisiert werden. Zeile vier muss für jede XZ-Koordinate des Terrain durchgeführt werden und dem entsprechenden Wert in der Heightmap zugewiesen werden. Eine Besonderheit stellt das Dividieren des Wertes *factor* dar. Dabei handelt es sich um einen Wert der dazu dient, die Werte auf die gewünschte Höhe beziehungsweise Bergigkeit des Terrains anzupassen. Dazu wird zunächst die Höhe invertiert. Und anschließend wie folgt verrechnet:

```
1 float scale =d 0.15f;
2 int invertedHeight = heightMax - height;
3 if(invertedHeight == 0) {
4         height = 3f;
5 } else {
6         height = (invertedHeight * scale) + 2.5f;
7 }
```

Listing 5.3: Berechnung des Höhenfaktors

Dies ergibt etwa folgende Werte:

Höhe	***factor***
0	17.5
25	13.75
50	10
75	6.25
100	3

Diese Berechnung hat folgenden Zweck: Das Ergebnis, welches von *perlinNoiseGenerator.GetValue(...)* ausgegeben wird, ist sehr hoch und das Gelände in jedem Fall sehr bergig. Dieser Faktor wurde so gewählt, dass das Gelände sich entsprechend des gewünschten Wertes verhält. Das heißt, ein sehr hoher Bergigkeits-Wert sorgt für einen kleinen Faktor durch den das Ergebnis dividiert wird und führt somit zu einem bergigen Ergebnis. Der Wert von 0.15 hat sich optisch als ansprechend und geeignet erwiesen um sowohl bergige als auch flache Gelände zu generieren.

Tabelle 5.4.: Beispiele für Höhenskalierungsfaktoren

Wege

Nachdem das Terrain generiert wurde, wird ein geeigneter Weg für das Gelände generiert. Die Generierung erfolgt mit Catmull-Rom Splines. Dafür wird die geeignete Anzahl an Punkten wie folgt ermittelt.

```
1 int factor = terrainResolution / 100;
2 int pointCount =
3         Mathf.RoundToInt(((challenge * 0.1f) + 5) * factor) + 2;
```

Listing 5.4: Berechnung der Anzahl der Punkte auf dem Weg

Dies macht für verschiedene *challenge*-Werte bei einer Terraingröße von 100 beziehungsweise 200 folgende Anzahl an Punkten auf dem Weg:

challenge	*pointCount*	
	100x100	**200x200**
0	7	12
25	10	17
50	12	22
75	15	27
100	17	32

Diese Anzahl an Punkten mag auf den ersten Blick sehr hoch erscheinen. Es ist jedoch eine große Anzahl an Punkten nötig, damit der Weg nicht zu geradlinig beziehungsweise eckig erscheint. Zudem sind vier dieser Punkte jeweils fest gesetzt um einen geeigneten Ein- und Ausgang für jedes Level zu bieten, alle anderen werden nach gewissen Regeln quasi zufällig gewählt.

Tabelle 5.5.: Beispiele für die Berechnung von Wegpunkten

Nachfolgend wird gezeigt, wie die Start- und Endpunkte gesetzt werden:

```
1 points[0] = new Vector3(levelEntryPoint, 0, 0);
2 points[1] = new Vector3(levelEntryPoint, 0, 5);
3 points[pointCount - 2] =
4         new Vector3(levelExitPoint, 0, terrainResolution - 5);
5 points[pointCount - 1] =
6         new Vector3(levelExitPoint, 0, terrainResolution);
```

Listing 5.5: Definition der vier Start- und Endpunkte des Weges

Die Verwendung von einem zweiten Punkt jeweils knapp vor beziehungsweise hinter dem End- beziehungsweise Startpunkt hat den Sinn, dass der Weg am Eingang und Ausgang gerade und somit für den Levelübergang gut geeignet sein soll. Alle anschließenden Punkte werden nach folgenden Regeln ausgewählt: Es wird bestimmt, wie weit der X-Wert des neuen Punktes in X-Richtung nach unten und oben vom alten Punkt abweichen darf (+-). Unterschreitet beziehungsweise überschreitet der berechnete Wert einen bestimmten Wert, wird dieser korrigiert. Anschließend wird ein Zufallswert zwischen der ermittelten oberen und unteren Grenze gewählt. Ähnlich wird für den Z-Wert des neuen Punktes verfahren. Dieser wird so berechnet, dass alle Punkte etwa gleich verteilt auf der Z-Achse des Terrains liegen. Auch dieser Wert wird wenn nötig korrigiert und anschließend ein Zufallswert aus der oberen und unteren Schranke gebildet. Dies wird so lange ausgeführt, bis alle nötigen Punkte ermittelt wurden.
Nachdem alle Punkte festgelegt wurden, wird mit der eigentlichen Generierung des Weges begonnen. Die Berechnung erfordert das Vorhandensein von vier Punkten und wird wie folgt angewendet: Es wird die in Kapitel 4.4.3 genannte Formel so angewendet, dass alle Punkte verbunden werden. Für alle Punkte vom zweiten bis zum Vorvorletzen kann die Berechnung einfach mit $p0 = i-1, p1 = i, p2 = i+1$ und $p3 = i+2$ durchgeführt werden. Jedes Wegstück

besteht aus einer Liste von Vektoren. Die Anzahl der Vektoren kann dabei durch $step$ beeinflusst werden. In diesem Fall wurde für ein möglichst wenig eckiges Ergebnis $step = 0.005$ gesetzt und somit besteht ein Wegstück aus 200 Vektoren.
Wurde die Berechnung wie oben beschrieben durchgeführt, wird die Liste aller Vektoren aller Wegstücke zurückgegeben. Um ein natürliches Ergebnis zu erzeugen wird für jeden Punkt des Weges der Mittelwert der Höhe der umliegenden Punkte gebildet und gesetzt. Zusätzlich wird der Weg im Vergleich zum umliegenden Gelände leicht abgesenkt.

Texturen

Wenn sowohl Gelände als auch Weg generiert wurden, müssen geeignete Texturen aufgebracht werden. Dazu werden diese zunächst in Unity für das Terrain zur Verfügung gestellt. Danach wird eine Splatmap, eine Map mit der Gewichtung der Texturen für jeden Punkt, erstellt und die Heightmap des Terrains abgerufen. Es wird unter anderem die Steilheit jedes Punktes des Terrains abgerufen und ein Array initialisiert, in welches für jede Textur eingetragen wird, wie sehr sie in diesem Punkt vorhanden ist. Für die Anpassung der Texturen wurde [Ait13] zur Hilfe genommen. Folgende Regeln sind möglich.

```
1  // Textur[0] wirkt konstant
2  splatWeights[0] = 0.5f;
3
4  // Textur[1] ist staerker in niedrigeren Hoehen
5  splatWeights[1] =
6          Mathf.Clamp01(
7          (terrainData.heightmapHeight - terrainHeight));
8
9  // Textur[2] wirkt mehr auf flacherem Terrain
10 splatWeights[2] =
11         1.0f - Mathf.Clamp01(steepness * steepness /
12          (terrainData.heightmapHeight / 5.0f));
13
14 // Textur[3] wird staerker mit zunehmender Hoehe
15 splatWeights[3] = terrainHeight * Mathf.Clamp01(normal.z);
```

Listing 5.6: Regeln für das Auftragen von Texturen

Konkret wurden hier nur die unteren drei Varianten genutzt. Zudem bekommen alle Punkte, die auf dem Weg liegen, eine eigene Textur um sich vom Rest des Terrains abzuheben.

Vegetation

Das Platzieren der Vegetation setzt sich aus zwei Teilen zusammen. Zum einen aus dem Generieren von Gras und zum anderen dem Erstellen von Bäumen. Eingestellt werden beide Werte zunächst über *vegetation*, anschließend jedoch unterschiedlich verrechnet. Für die Menge an zu platzierendem Gras wird *vegetation* verdoppelt und mit der Größe des Terrains verrechnet. Die Anzahl der Bäume richtet sich nach der Größe des Terrains, *vegetation* und einem Faktor. Wird beispielsweise eine *vegetation* von 50 angestrebt, werden bei einer Terraingröße von 100 $100 * 100 * ((50/100) * 2) = 10.000 * (0,5 * 2) = 10.000 * 1$ Einheiten Gras und $100 * 100 * (50/5000) = 10.000 * 0,01 = 100$ Bäume platziert.
Die Platzierung erfolgt für beide Vegetationssorten ähnlich: Es gibt verschiedene Grasarten beziehungsweise Bäume. Zuerst wird ermittelt, wie viele Objekte jedes Typs erstellt werden können, der Rest wird durch einen zufälligen Typ abgedeckt. Bevor das Graselement oder der Baum jedoch platziert wird, wird geprüft, ob sich die zufällig bestimmte Position auf dem Weg oder auf bereits platziertem Gras beziehungsweise Bäumen befindet. Falls dem so ist wird ein neuer Punkt zufällig bestimmt.

Gestein

Die Platzierung von Gestein im Gelände soll im Prinzip genauso wie die Platzierung von Vegetation erfolgen: Es wird ein Wert *rocks* zwischen 0 und 100 angegeben. Dieser Wert wird gemäß der Beziehung $100 * 100 * \textit{rocks} * \frac{1}{20.000}$ in die Anzahl platzierbarer Steine umgerechnet. Dabei wurde der Skalierungsfaktor $\frac{1}{20.000}$ so gewählt, dass auf die übliche Terraingröße (100x100) maximal 50 Steine verteilt werden. Zudem sollen als Begrenzung des Geländes an den Rändern rundherum große Felsen platziert werden. Dies soll verhindern, dass der Spieler aus dem Level heraus fällt. Zusätzlich sollen die Levelein- und -ausgänge mit Steinen verkleidet werden um von außen den Eindruck einer Höhle zu erwecken. Die Verwendung von Gestein ist derzeit noch nicht integriert, allerdings vorbereitet und somit vergleichsweise einfach zu realisieren.

Labyrinthe

Wie in Kapitel 5.4 bereits erwähnt, wird für die Generierung des Labyrinthes das “Maze Generator”-Asset von styanton abgewandelt. Das Asset sieht vor, dass ein Labyrinth aus folgenden drei Bauteilen besteht: einer Bodenplatte, einer Wand und einem Pfeiler. Diese Bauteile werden mit einem wählbaren Algorithmus zusammengesetzt. Das Asset wurde dahingehend umgebaut, dass die Labyrinthe ohne Pfeiler zusammengesetzt werden und dass an der Position des Ein- und Ausgang keine Wände platziert werden. Die Generierung kann wie folgt konfiguriert werden.

- ***algorithm:*** Der verwendete Algorithmus kann über dieses Feld gewählt werden. Zur Wahl stehen *PureRecursive*, *RecursiveTree*, *RandomTree*, *OldestTree* und *RecursiveDivision*. Für einen Vergleich der einzelnen Algorithmen siehe Kapitel 4.4.4. In dieser Arbeit werden Labyrinthe lediglich mit dem *PureRecursive*-Algorithmus generiert.
- ***cellSize:*** Die *cellSize* bestimmt, wie hoch und wie breit eine Zelle des Labyrinthes ist. Da die Gesamtgröße des Geländes durch diesen Wert geteilt wird, ist zu beachten, dass dieser Wert gerade sein sollte.
- ***floorPrefab:*** Bei dem *floorPrefab* handelt es sich um die Vorlage für den Boden des Labyrinthes. Sollen beispielsweise die Texturen geändert werden, muss dies nur einmalig im Prefab geschehen.
- ***wallPrefab:*** Das zweite verwendete Prefab ist das *wallPrefab*, welches als Vorlage für die einzelnen Wände genutzt wird.

Die oben genannten Parameter werden, bis auf die beiden Prefabs, in der entsprechenden Level-Datei festgelegt. Die Prefabs werden direkt in Unity gesetzt. Durch die *terrainResolution* und *cellSize* wird die Anzahl der Spalten und Reihen ermittelt um das Labyrinth möglichst groß und daher auch komplex zu machen.

6. Evaluation

In diesem Kapitel werden die Resultate der Arbeit unter vier verschiedenen Gesichtspunkten beleuchtet und bewertet, wobei jedem Aspekt ein eigenes Unterkapitel gewidmet ist. Zunächst wird der Frage nachgegangen, wie gut das Ergebnis die in Kapitel 3.1 aufgestellten Anforderungen erfüllt. Anschließend wird die Performance des Programms analysiert und es erfolgt eine allgemeine Einschätzung der Stärken und Schwächen des Programms. Zuletzt wird ein technischer Ausblick aufgezeigt.

6.1. Erfüllung der Anforderungen

Es wurden in Kapitel 3.1 für die einzelnen Komponenten des Programms Anforderungen aufgestellt. Die überwiegende Mehrheit dieser Anforderungen konnte im Rahmen der Arbeit erfüllt werden, während sich zugleich die verbleibenden Einschränkungen allesamt als nicht wesentlich für die generelle Funktion erweisen. Dennoch sollen diese hier im Vordergrund stehen um aufbauenden Arbeiten sowohl eine kritische Grundlage als auch eine Perspektive zu bieten.

Die maximale Steigung des Weges beträgt 35%
Da das Höhenprofil des Weges sich dem Verlauf des Geländes anpasst, kann nicht ausgeschlossen werden, dass die Steigung 35% übersteigt. Ungeachtet dessen hat sich das Resultat in allen getesteten Fällen als angemessen und optisch ansprechend herausgestellt.

Der maximale Kurvenradius beträgt 58m, der minimale 23.2m
Diese Anforderungen gelten für die Realität, lassen sich aber nur eingeschränkt auf das EmotionBike übertragen. Dies hat den Grund, dass die Geschwindigkeit des Spielers sich beliebig skalieren lässt und damit weit abseits der realistischen Durchschnittsgeschwindigkeit von 20km/h liegen kann. Dementsprechend kann der minimale und maximale Kurvenradius von den oben genannten Werten abweichen. Angesichts dieses potentiell sehr großen Wertebereichs, konnte der Kurvenradius bislang nicht berücksichtigt werden. Jedoch wurden bei der Generierung des Weges die Zufallswerte der Wegpunkte dahingehend eingeschränkt, dass ein

attraktives und sinnvolles Ergebnis erzeugt wird. Zukünftig ließe sich diese Platzierungslogik verhältnismäßig einfach erweitern, indem der Krümmungsradius der Catmull-Rom Splines ausgewertet und als Kriterium herangezogen wird.

Der Weg ist möglichst ebenmäßig

Initial wurde diese Anforderung nicht erfüllt, da der Weg je nach Struktur des Terrains große Unebenheiten aufwies. Daraufhin wurde als Korrekturverfahren eine Mittelwertbildung auf den Weg angewandt, welche das Ergebnis optisch verbessert hat. Allerdings wären hierzu weitere Nacharbeiten mit geeigneteren Verfahren wünschenswert.

Die Schwierigkeit des Labyrinthes ist einstellbar

Die Schwierigkeit des Labyrinthes ist nur indirekt parametrisierbar. Dazu wird der Algorithmus zum Generieren ausgetauscht oder die Größe des Labyrinthes verändert.

Die Farbe und Textur des Labyrinthes sind einstellbar

Die Farbe und die Textur des Labyrinthes sind ebenfalls nur indirekt einstellbar. Dazu müssen die entsprechenden Prefabs für die Wände und den Boden des Labyrinthes angepasst werden. Dies kann allerdings in Unity nicht zur Laufzeit erfolgen. Denkbar wäre das Verwenden von mehreren Prefabs welche je nach Bedarf geladen werden.

Es ist dafür zu sorgen, dass der Spieler das Level nicht auf anderem Wege verlassen kann

Diese Anforderung konnte bislang nicht zufriedenstellend erfüllt werden, sodass es noch möglich ist, das Gelände außerordentlich zu verlassen. Grund hierfür ist, dass derzeit keine Methode zur Verfügung steht, um attraktive Levelbegrenzungen aus Felsobjekten performant zu generieren.

6.2. Performance

Nachfolgend wird aufgezeigt wie es um die Performance von Landschafts- und Labyrinth-Szenen steht. Die Tests wurden unter folgendem System durchgeführt.

- **Prozessor:** Intel i7 960, 4 Kerne, 3,2 GHz
- **Grafikkarte:** NVIDIA GeForce GTX 970
- **Arbeitsspeicher:** 12 GB DDR3

6.2.1. Generierung von Landschaften

Um die Performance zu messen, wurden mehrere Level mit verschiedenen Parametern generiert. Die Vegetationserzeugung wurde gesondert untersucht. Es wurden 10 Level für jede dieser Kombinationen generiert und anhand dessen jeweils der Mittelwert der Berechnungsdauer ermittelt. Die Messungen wurden für eine Terraingröße von 100x100 Punkten durchgeführt. Zum Vergleich wurden auch einige Messungen für eine Größe von 200x200 Punkten durchgeführt.
In den Tabellen 6.1 und 6.2 sind diese Ergebnisse aufgeführt.

terrainSlope	***pathComplexity***	***vegetation***	**Dauer (Min:Sek:MilliSek)**	
			100x100	200x200
10	10	5	0:23:5	5:40
50	10	5	0:27:9	6:18
90	10	5	0:24:7	5:59
100	10	5	0:22:5	6:04
10	50	5	0:35:3	8:15
50	50	5	0:35:0	8:54
90	50	5	0:33:7	7:55
100	50	5	0:37:8	8:20
50	100	5	0:44:4	9:46

Tabelle 6.1.: Übersicht über die Generierungsdauer verschiedener Landschafts-Szenen

terrainSlope	***pathComplexity***	***vegetation***	**Dauer (Min:Sek:MilliSek)**
50	50	10	1:06:9
50	50	20	2:06:6
50	50	30	3:10:3
50	50	40	4:17:4
50	50	50	5:10:0

Tabelle 6.2.: Übersicht über die Generierungsdauer verschiedener Landschafts-Szenen

Die ersten Werte der Tabelle 6.1 wurden mit einem Vegetationswert von 5 und verschiedenen Werten zwischen 10 und 90 für *terrainSlope* beziehungsweise *pathComplexity* generiert. Dort wird deutlich, dass die Performance in diesen Fällen sehr gut ist: Das Terrain lässt sich in nur 22 bis 44 Sekunden generieren. Problematisch wird es, sobald mehr Vegetation gewünscht wird, sichtbar in Tabelle 6.2. Nachfolgende Abbildung 6.1 zeigt, wie sich die *vegetation* bei *terrainSlope* und *pathComplexity* von 50 auf die Dauer auswirkt.

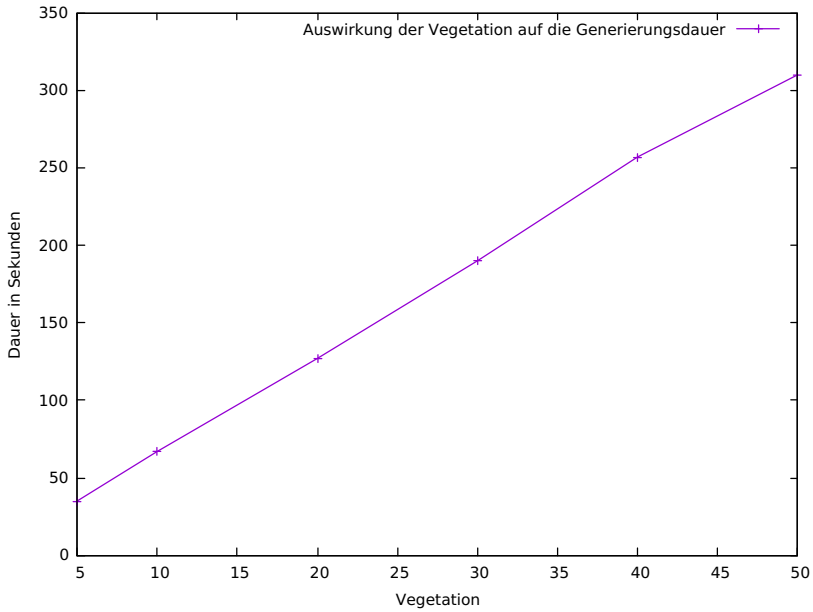


Abbildung 6.1.: Zusammenhang zwischen dem Vegetationsgrad und der Generierungsdauer

Gut erkennbar ist, dass die Generierungsdauer etwa linear mit der gewünschten *vegetation* steigt. Dies liegt daran, dass die Vegetation mit zufälligen Werten platziert wird und für jeden Wert geprüft werden muss, ob sich dieser auf dem Weg befindet oder anderweitig blockiert ist. Zudem stellt jeder Baum ein eigenes Objekt dar. Dadurch hat das Generieren von Vegetation zeitlich einen großen Einfluss auf das Generieren von Leveln.
Da die Generierung von Gestein derzeit mit der selben Methode erfolgen müsste, würde sich die Performance in diesem Fall noch verschlechtern. Der Zeitbedarf zur Terraingenerierung ist ab einer *vegetation* von 20 nicht mehr tragbar, da der Spieler in diesem Fall bereits mehr als zwei Minuten warten, beziehungsweise durch den Übergangstunnel fahren, müsste, bis die Generierung abgeschlossen ist.

6.2.2. Generierung von Labyrinthen

Um die Performance des Labyrinthes zu messen, werden mehrere Level generiert. Diese unterscheiden sich jedoch in der Terraingröße. Für jede Größe wurden hier 20 Level generiert, da die Möglichkeiten wesentlich eingeschränkter sind als bei der Generierung von Landschafts-Szenen. Tabelle 6.3 zeigt, wie sich die Performance dieser Labyrinth-Szenen verhält. Der verwendete Algorithmus ist dabei jeweils rein rekursiv.

terrainSize	**Dauer (Min:Sek:MilliSek)**
100x100	0:03:1
150x150	0:03:0
200x200	0:03:5
250x250	0:03:2
350x350	0:04:0
500x500	0:05:3
900x900	0:10:1

Tabelle 6.3.: Übersicht über die Generierungsdauer von Labyrinthen

Die Ergebnisse in Tabelle 6.3 zeigen, dass die Generierung von Labyrinthen bis zu einer Terraingröße von 900x900 Punkten sehr schnell erfolgt. Bei einer Terraingröße von 1000x1000 Punkten trat bei der Generierung eine StackOverflowException auf. Allerdings sind Labyrinthe bei einer Größe von 900x900 Punkten sehr schwierig für den Spieler und somit ist die Generierung von Labyrinthen mehr als zufriedenstellend.

6.3. Stärken und Schwächen

Nachfolgend werden weitere Stärken und Schwächen sowohl der Umgebung als auch des Programms dargestellt.
Die Wahl von Unity zur Realisierung war naheliegend, da dies bereits im EmotionBike Projekt verwendet wurde. Die Einarbeitung in diese Umgebung erfolgte mittels des Buches “Das Unity-Buch” von Jashan Chittesh [Chi15]. Um sich darüber hinaus in die Realisierung von Terrains in Unity einzuarbeiten, wurden zudem diverse Foren genutzt. Gleizeitig bietet Unity selbst auch einige Manuals [Man16]. Trotz dieser Möglichkeiten gestaltete sich die Einarbeitung allerdings eher als schwierig. Viele Möglichkeiten zum Generieren sind nur unzureichend oder gar nicht dokumentiert und viele der existierenden Beispiele liegen für ältere Unity Versionen vor, weshalb sie oftmals nicht lauffähig sind. Hinzu kommt, dass häufig die Bedienung und der Umgang mit Komponenten nicht intuitiv ist. Dennoch bietet Unity durchaus Vorteile. So sind viele grundlegende Dinge durch die oben angesprochene Dokumentation gut abgedeckt und teilweise sogar in Form von vorgefertigten Objekten in Unity verfügbar. Beispielsweise lässt sich ein Terrain, das heißt eine ebene quadratische Platte, mit wenigen Klicks erstellen. Des Weiteren bietet Unity gute Tools [Tec], um das erstellte Terrain von Hand anzupassen. Auch ist das Exportieren von erstellter Software einfach möglich und das Erstellen von neuen Szenen ebenfalls. Nachteilig hierfür ist wiederum die Bindung an Windows oder macOS als Betriebssysteme, da die Linux-Variante nicht in dem benötigten Umfang funktionstüchtig ist.

Ein entscheidender Vorteil von Unity sind allerdings die sehr zahlreich verfügbaren, zum Großteil kostenlosen, Assets. So müssen Objekte wie Bäume und Steine oder aber Texturen nicht selbst erstellt werden und sogar zum Generieren von Terrain sind zum Teil Assets vorhanden. Jedoch sind Assets oft von Privatpersonen entwickelt worden, und bringen somit unter Umständen auch Probleme mit sich. Manche Assets wurden für eine ältere Version von Unity bereit gestellt und seither nicht gewartet und sind daher nicht mehr funktionstüchtig. Andere Assets sind wiederum nicht oder nur sehr wenig dokumentiert. Die Benutzung ist in diesen Fällen schwierig bis unmöglich. Ein ebenfalls öfter eingetretener Fall ist, dass Assets bei ihrer Verwendung Fehler erzeugen, welche nicht behoben werden können. Trotz dieser Nachteile erweisen sich Assets in Summe als große Stärke von Unity.

Nachdem die Umgebung oben bereits bewertet wurde, folgt nun die Bewertung des entworfenen Produktes. Hierbei bleiben Fragen zum Realitäts- und Empfindungsgrad bewusst unberücksichtigt, da sie ausschließlich subjektiv bewertet werden können. Während die Beurteilung des Programmes in emotionaler Hinsicht weiterreichende (psychologische) Studien erforderlich machen würde, lassen sich die funktionalen Aspekte jedoch an dieser Stelle bewerten.
Ein wesentlicher Nachteil, der in Kapitel 6.2 bereits ausführlich diskutiert wurde, ist die Performance. Gerade bei der Generierung von Landschafts-Szenen sind problematische Defizite vorhanden, wohingegen die Generierung von Labyrinthen akzeptabel ist. Diese Problematik muss für eine geeignete Verwendung gelöst werden. Die Architektur des Programms ist positiv zu bewerten. Dies liegt zum einen daran, dass neue Level einfach durch eine entsprechende Level-XML-Datei hinzugefügt werden können, als auch daran, dass neue Generatoren für Szenen ebenso einfach eingebracht werden können. Generell wurde Wert darauf gelegt, dass einzelne Module lose gekoppelt sind und sich jeweils aus für das Modul relevanten Teilen zusammensetzen (starke Kohäsion). Dies führt sowohl zu guter Übersichtlichkeit und somit einer geringen Einarbeitungszeit als auch zu wenig doppeltem Code. In generell allen Dateien des Projektes wurde auf eine geeignete Dokumentation geachtet, was wiederum die Einarbeitungszeit reduzieren und das Verständnis erhöhen soll. Zudem wurden mögliche Sonderfälle durch geeignetes Exception-Handling gesichert.

Ein optischer Eindruck der Resultate lässt sich exemplarisch dem Anhang unter Punkt A.1 entnehmen.

6.4. Technischer Ausblick

Das in dieser Arbeit entwickelte Baukastensystem und die Möglichkeiten der Generierung sind bei Weitem nicht ausgeschöpft. Nachfolgend gibt es eine Übersicht über denkbare Verbesserungen beziehungsweise Erweiterungen.

Auswahl eines Platzierungsalgorithmus für Objekte

Ein erster Verbesserungsansatz ist bei dem Platzierungsalgorithmus für Objekte wie Bäume oder Felsen zu finden. Die derzeitige Form der Generierung erfolgt durch zufällige Positionierung nach der Wegeerzeugung. Denkbar wäre auch, die Objekte zuerst zu generieren, anschließend einen Weg zu berechnen und Objekte, die sich auf diesem befinden, zu entfernen. Hierdurch ließe sich voraussichtlich die Performance des Programmes erhöhen.

Umsetzung der fehlenden Anforderungen

Wie in Kapitel 6.1 beschrieben konnten nicht alle in Kapitel 3.1 genannten Anforderungen erfüllt werden. Besonders wichtig ist dabei eine Begrenzung des Levels durch eine geeignete Methode. Neben der Verwendung von Felsen ist auch eine Felswand durch beispielsweise eine Erhöhung des Terrains mit sehr hoher Steigung an den Rändern denkbar.

Auswahl eines natürlicheren Algorithmus zur Terrain-Generierung

In Kapitel 4.4.1 werden geeignete Methoden zur Generierung von Terrain genannt. Obwohl bereits optisch ansprechende Ergebnisse erzeugt werden können, ist gut erkennbar, dass es Algorithmen gibt, die bei wesentlich höherer Komplexität ein natürlicheres Ergebnis bieten. Die Verwendung dieser sollte in Betracht gezogen werden.

Entwicklung neuer Szenen und Erweiterung bereits bestehender Szenen

Derzeit wurden zwei verschiedene Szenen zur Darstellung von Emotionen umgesetzt. Und obwohl diese Szenen parametrisierbar sind, ist dies nur ein kleiner Teil an Möglichkeiten. Beispielsweise ist die Generierung von Wüsten weitestgehend vorbereitet aber noch nicht realisiert. Denkbar wären weitere Arten von Terrain, beispielsweise Schneelandschaften, oder auch die Verwendung von Wasser. Des Weiteren wäre die Verwendung von Planken, auf welchen der Nutzer balancieren kann, eine gute Option, um bereits bestehende Landschafts-Szenen zu erweitern und somit weitere Emotionen abzudecken.

Erweiterung der darstellbaren Emotionen

Hier wurden nur einige Basisemotionen dargestellt, die Auswahl an möglichen Emotionen

ist allerdings wesentlich größer. Dementsprechend bieten sich hier vielfältige Weiterentwicklungsmöglichkeiten.

Verwendung von Events

Um mehr Emotionen darstellen zu können oder Emotionen zu verstärken ist die Verwendung von Events denkbar. Dabei ist zum Beispiel das Einstürzen von Wänden in Labyrinthen oder das Herunterfallen von Felsen in Landschafts-Szenen ein möglicher zu untersuchender Ansatz.

Erweiterung des Auftragens von Texturen

Ebenfalls eine Möglichkeit die Natürlichkeit des Ergebnisses weiter zu erhöhen, ist die Erweiterung von Texturen. Die aus dem Asset-Store geladenen Texturen sind teilweise sehr unnatürlich und schwierig kombinierbar. Deshalb sollte in Erwägung gezogen werden, neue eigene Texturen zu entwickeln und zu verwenden. Im Zuge dessen sollte auch der Algorithmus zum Auftragen der Texturen verbessert werden. Derzeit ist grade das Auftragen in höheren Bereichen des Geländes stark von der Steigung abhängig und erzeugt dadurch oftmals wenig natürliche Ergebnisse.

Verwendung einer Dokumenttypdefinition (DTD)

Um das erzeugte Baukastensystem noch zu verbessern, das heißt noch robuster zu gestalten und sicherzustellen, dass die Werte in den XML-Dateien der DSL korrekt sind, sollte eine interne oder auch externe DTD entwickelt werden.

7. Schluss

7.1. Zusammenfassung

Ziel dieser Arbeit war es, Level einer Spielewelt dynamisch anhand von Spieleremotionen zu erzeugen. Der Fokus lag hierbei auf der prozeduralen Generierung dieser parametrisierbaren Szenen, welche in Unity implementiert wurden. Zur passenden Kombination dieser Szenen wurde ein einfaches und flexibles Baukastensystem entwickelt.
Zunächst wurde sich mit der Auswahl von Emotionen und der Szenengenerierung auseinander gesetzt. Um einen kleinen Überblick über die Möglichkeiten dieser Methode zu geben, wurden sechs Basisemotionen gewählt. Als parametrisierbare Szenen wurden eine Landschafts-Szene mit einem natürlichen Gelände und entsprechender Vegetation sowie eine Labyrinth-Szene entwickelt. Hierzu wurden verschiedene Methoden beziehungsweise Algorithmen zu der Erzeugung von Gelände, Wegen, und Labyrinthen gesichtet und analysiert. Für die Generierung von Terrain wurde das Perlin Noise gewählt, da es eine unkomplizierte Möglichkeit darstellt, ansprechendes Terrain zu generieren. Die Generierung der Wege erfolgte mittels Catmull-Rom Splines womit schnell geeignete Wege generiert werden könnten. Für die Diversität von Texturen und Objekten wie Bäumen oder Gras wurden diverse Assets aus dem Unity Assetstore zu Hilfe gezogen. Das Generieren von Labyrinthen wurde anhand von diversen Beispiellabyrinthen untersucht, wobei Algorithmen in Hinblick auf die Komplexität der erzeugten Labyrinthe bewertet wurden. Gewählt wurde der rein rekursive Algorithmus aufgrund überzeugender Ergebnisse in der Analyse. Zur Vervollständigung der Landschafts-Szene wurde eine zufällige Platzierung von Vegetation ermöglicht, deren Dichte parametrisierbar ist, und außerdem die Texturierung des Geländes in Abhängigkeit seiner Höhe und Steilheit implementiert.
Zuletzt wurde das Baukastensystem und die dazugehörige DSL realisiert, mit Hilfe derer jede Szene über XML-Dateien parametrisiert werden kann. Unterteilt wird diese DSL in eine Datei, welche für alle Szenen gilt, und einzelne Dateien spezifisch für die jeweiligen Level. Für das Baukastensystem wurde des Weiteren untersucht, wie sich Levelbausteine geeignet kombinieren lassen. Gewählt wurde aufgrund der einfacheren Realisierbarkeit das Verwenden von Tunneln.

Die Evaluation zeigte, dass die Generierung von Labyrinth-Szenen durchweg gute Performance aufweist. Landschafts-Szenen lassen sich ebenfalls in akzeptabler Zeit generieren, weisen allerdings sobald umfangreiche Vegetation gewünscht wird, Defizite in Bezug auf Zeitbedarf zur Generierung auf.

Abschließend bleibt zu sagen, dass die Themen der prozeduralen Generierung und der Verarbeitung von Emotionen beziehungsweise Spielerinteraktionen sehr komplex sind und stetig mehr an Bedeutung gewinnen. Diese Arbeit bietet lediglich einen initialen Überblick über die Möglichkeiten in diesen Gebieten und dementsprechend viele Optionen für Weiterentwicklungen.

7.2. Ausblick

Aufbauend auf den Ergebnissen dieser Arbeit ergeben sich vielfältige Optionen zu weiterer Entwicklung und Anstöße zu innovativen Konzepten. Viele konkrete Möglichkeiten zur Weiterentwicklung der gegenwärtigen Implementierung wurden bereits in Kapitel 6.4 aufgezeigt, allerdings erschöpfen diese nicht im Mindesten das Potential dynamisch generierter Szenarien. Basierend auf Spielerinteraktionen und -emotionen ließe sich dynamische Generierung auch für Spielatmosphäre und -inhalte in Betracht ziehen, womit beispielsweise im Rahmen klassischer Rollenspiele das Spielerlebnis personalisiert und dadurch intensiviert werden könnte. Gerade die Klasse der openworld Spiele stellt weitere Entwicklungen jedoch auch vor große Herausforderungen, so müssen einerseits Landschaften reproduzierbar und andererseits ihre Übergänge nahtlos sein. Nachfolgende Arbeiten sollten hierzu unter anderem das Generieren und Verwenden von Seeds und die geeignete Verbindung von Biomen auf Basis von Chunks untersuchen.
Will man die Empfindungen des Spielers noch stärker in den Mittelpunkt stellen, wäre eine Übertragung ins Horror-Genre denkbar. Hier können hochdynamische Reaktionen der Spielumgebung auf Spieleremotionen unter psychologischen Gesichtspunkten eingesetzt werden, um die Intensität des Angstempfindens zu vervielfachen. Ergänzend ließe sich durch den Einsatz von virtueller Realität die Immersion maximieren.
Jenseits konventioneller Computerspiele eröffnen sich gerade in Verbindung mit virtueller Realität auch gänzlich neuartige Nutzungsfelder. Im wachsenden Fitness- und Healthcare-Sektor wären Sportgeräte möglich, die, vergleichbar mit dem EmotionBike, dem Nutzer attraktive Trainingsprogramme in adaptiven virtuellen Landschaften ermöglichen. Auch medizinische Schmerzbehandlung, wie sie derzeit mithilfe virtueller Realität erprobt wird [Inc], könnte von

besserem emotionalen Feedback und prozeduraler Synthese profitieren.
Ein weiteres Anwendungsgebiet prozedural generierter Landschaften wären Simulatoren deren Nutzwert wesentlich von der zufälligen Neuerung des Geländes erhöht würde. Naheliegende Beispiele fänden sich im polizeilichen oder militärischen Einsatztraining oder aber in der Fahrschulausbildung. In beiden Fällen könnte emotionales Feedback genutzt werden, um den psychologischen Druck der Situation anzupassen. Während polizeiliche Einsätze hier eventuelle Härtefallreaktionen provozieren können, ließe sich umgekehrt für nervöse Fahranfänger dynamisch Erleichterung schaffen.

In Anbetracht der stetig wachsenden Bedeutung von virtueller Realität und Companion Systemen muss dieser Ausblick zwangsläufig unvollständig bleiben, jedoch zeigt er gerade hierdurch, dass der dynamischen Generierung von Inhalten in Zukunft ein immer größerer Stellenwert zukommen wird.

A. Anhang

A.1. Ergebnis-Bilder

A.1.1. Landschaften

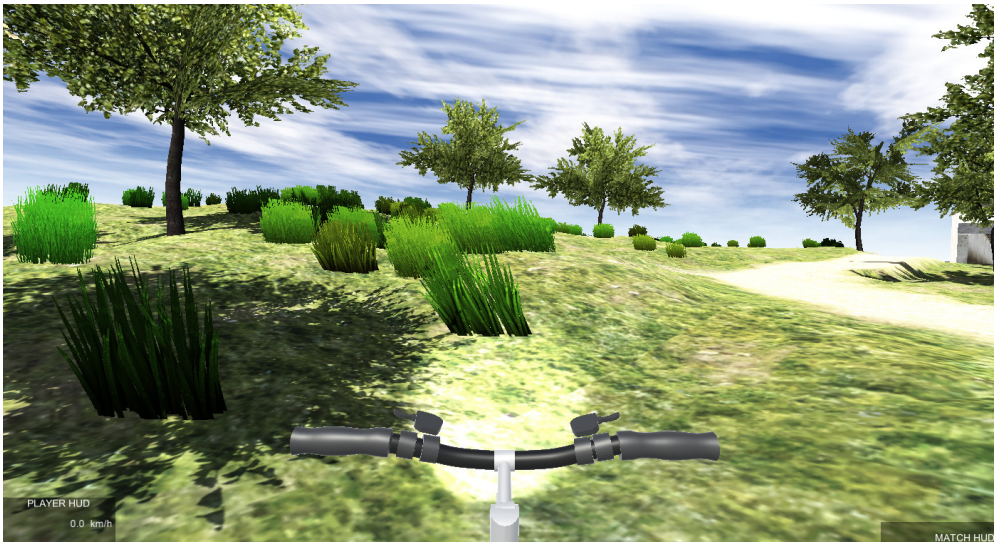


Abbildung A.1.: Generiertes Terrain *terrainSlope = 10*, *pathComplexity = 10*, *vegetation = 5*

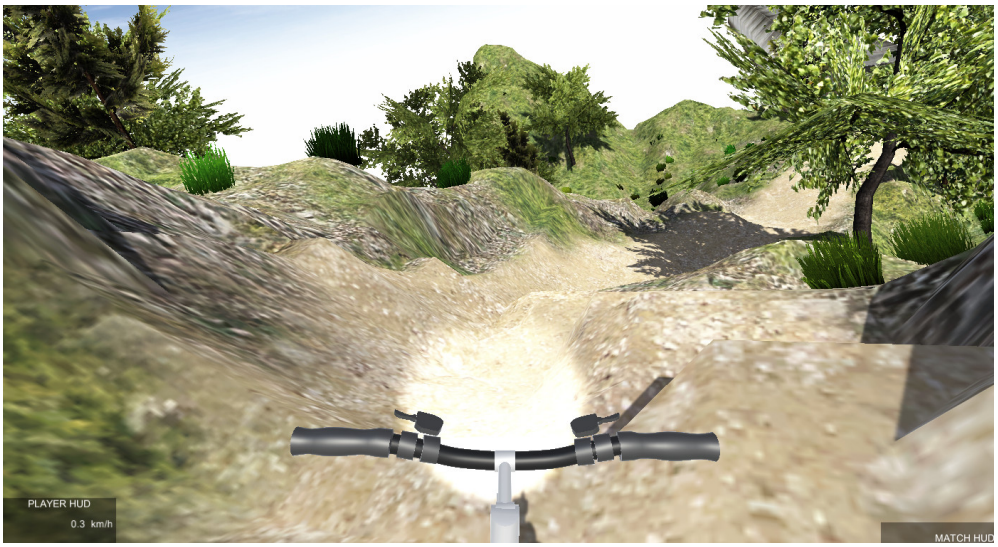


Abbildung A.2.: Generiertes Terrain *terrainSlope = 100*, *pathComplexity = 10*, *vegetation = 5*

Die Abbildungen A.1 und A.2 zeigen deutlich den Unterschied zwischen wenigen und vielen Bergen. In Zahlen ausgedrückt sind die Unterschiede wie folgt: Abbildung A.1 besitzt *terrainSlope* und *pathComplexity* von 50 sowie *vegetation* von 5. Abbildung A.2 unterscheidet sich durch einen *terrainSlope* von 100.

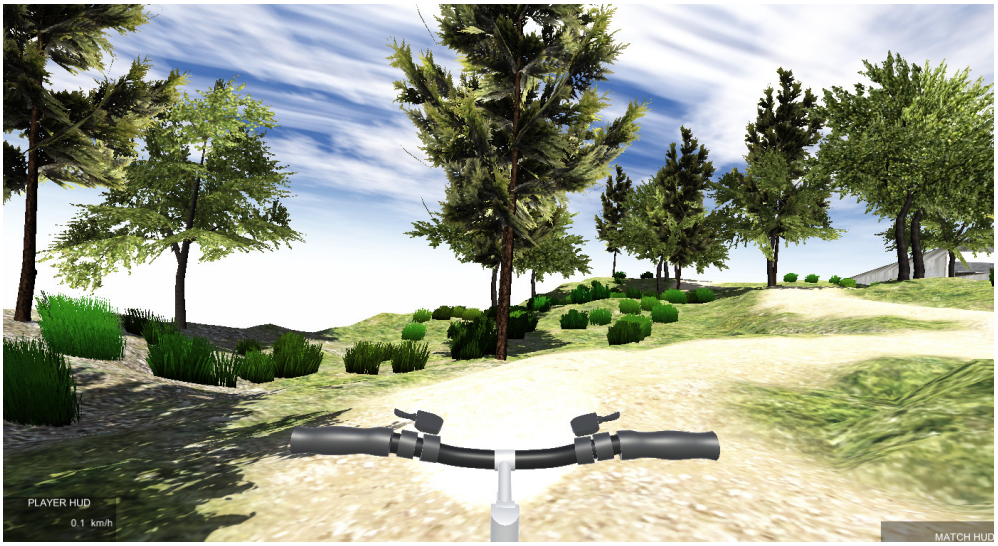


Abbildung A.3.: Generiertes Terrain *terrainSlope = 50*, *pathComplexity = 50*, *vegetation = 10*

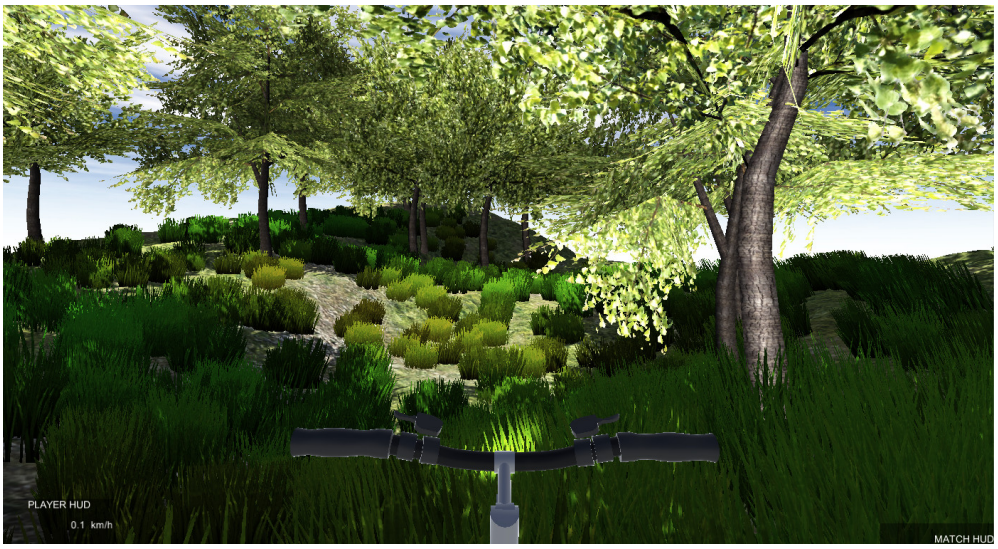


Abbildung A.4.: Generiertes Terrain *terrainSlope = 50, pathComplexity = 50, vegetation = 50*

Die Abbildungen A.3 und A.4 beleuchten den Unterschied zwischen wenig und viel Vegetation. In beiden Leveln wurde für einen geeigneten Vergleich *terrainSlope* und *pathComplexity* jeweils auf 50 gestellt. Der Wert für *vegetation* ist in Abbildung A.3 bei 10 und in Abbildung A.4 bei 50. Nachfolgend noch drei Bilder in Draufsicht. Abbildung A.5 zeigt ein sehr bergiges Level, Abbildung A.6 ein Level mit einem sehr komplexen Weg und Abbildung A.7 viel Vegetation.

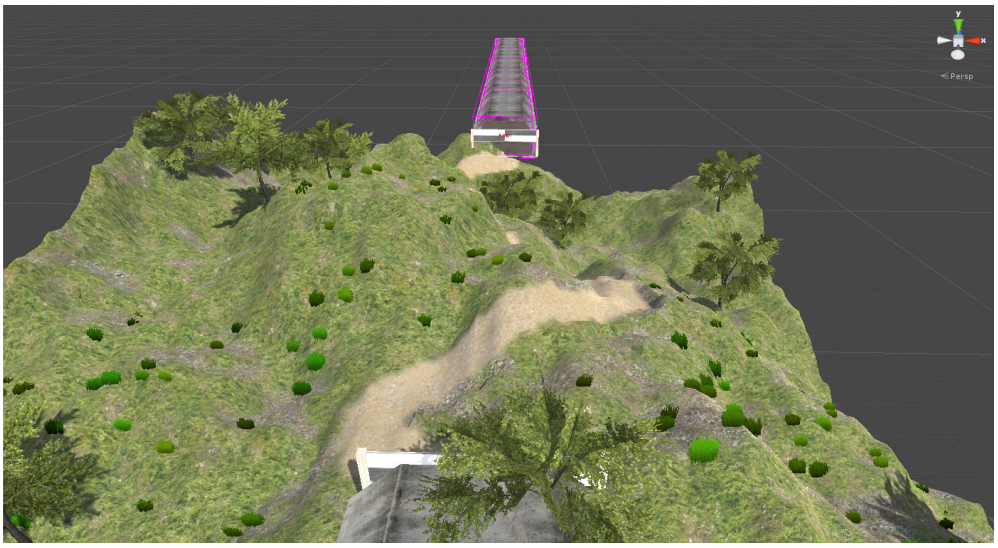


Abbildung A.5.: Generiertes Terrain *terrainSlope = 100, pathComplexity = 50, vegetation = 5*

Abbildung A.6.: Generiertes Terrain *terrainSlope = 50, pathComplexity = 100, vegetation = 5*

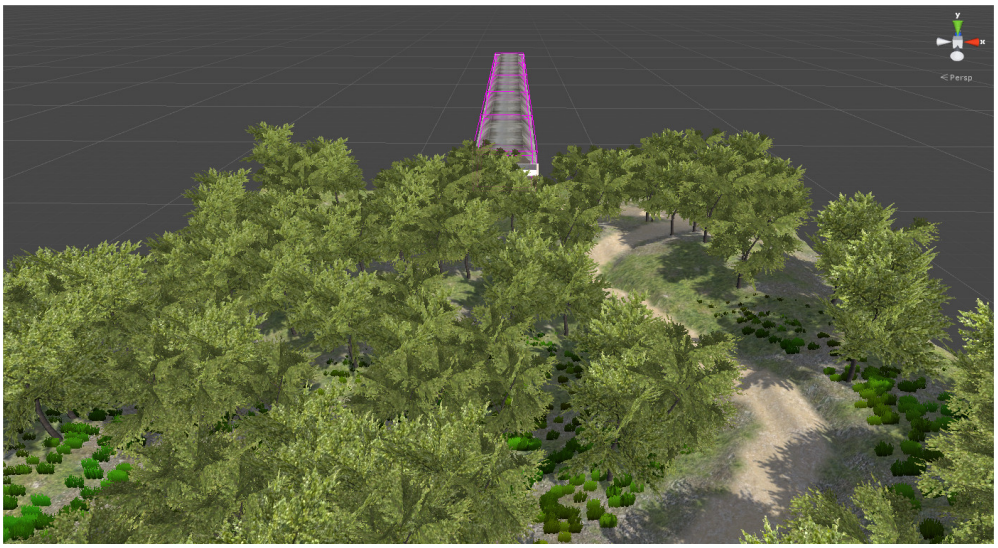


Abbildung A.7.: Generiertes Terrain *terrainSlope = 50, pathComplexity = 50, vegetation = 50*

A.1.2. Labyrinthe

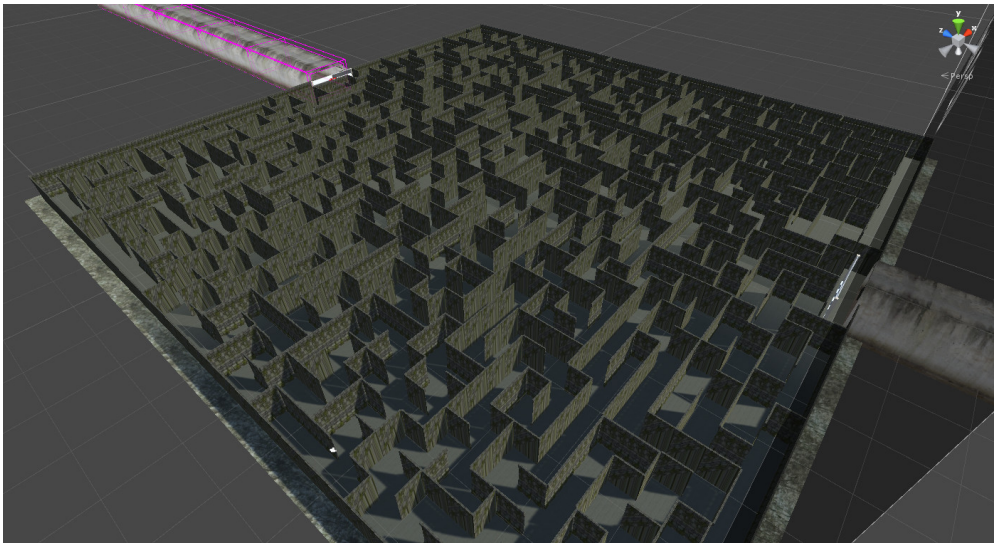


Abbildung A.8.: Generiertes Labyrinth *terrainSize = 150*

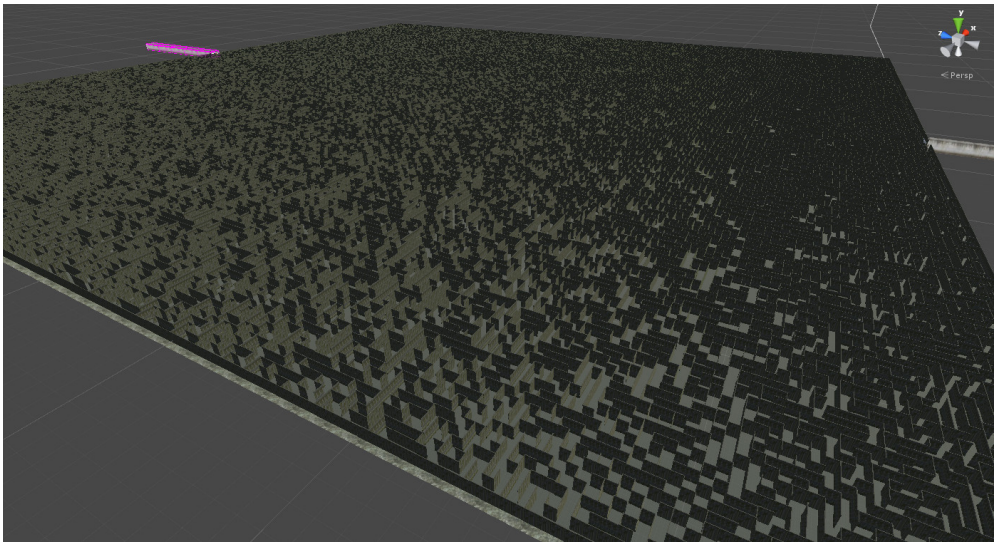


Abbildung A.9.: Generiertes Labyrinth *terrainSize = 900*

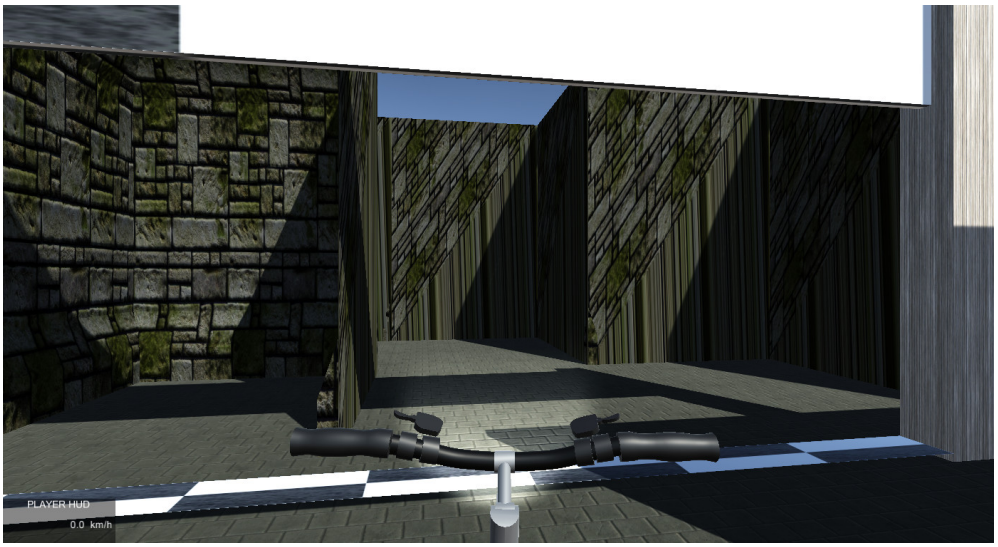


Abbildung A.10.: Generiertes Labyrinth aus Spielerperspektive

A.2. Quellcode-Listing

- eigene Klassen
  - ModuleCombinationSystem.cs
  - LandscapeGenerator.cs
  - MazeGenerator.cs
  - PathGenerator.cs
  - CatMullRom.cs
  - TextureGenerator.cs
  - VegetationGenerator.cs
- XML-Dateien
  - TerrainData.xml
  - Level0.xml
  - Level1.xml
  - Level2.xml
  - Level3.xml
  - Level4.xml

  - Level5.xml
  - Level6.xml
  - Level7.xml
  - Level8.xml

- Angepasst / Extern
  - BasicMazeGenerator.cs
  - DivisionMazeGenerator.cs
  - OldestTreeMazeGenerator.cs
  - RandomTreeMazeGenerator.cs
  - RecursiveMazeGenerator.cs
  - RecursiveTreeMazeGenerator.cs
  - TreeMazeGenerator.cs
  - MazeCell.cs
  - LibNoise-Klassen

A.3. Klassendiagramm

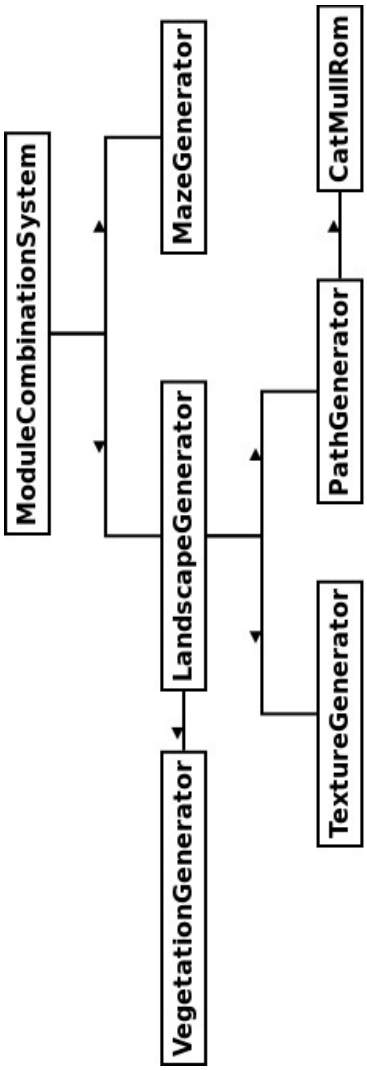


Abbildung A.11.: Klassendiagramm

A.4. Assets

- Labyrinth
  - Maze Generator von styanton[sty15b]

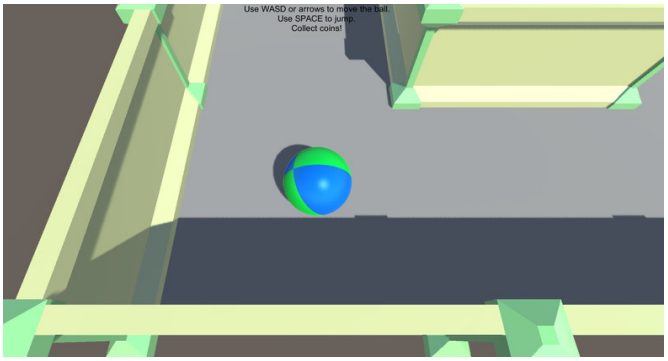


Abbildung A.12.: Maze Generator

- Skybox
  - Free HDR Sky von ProAssets[Pro16]

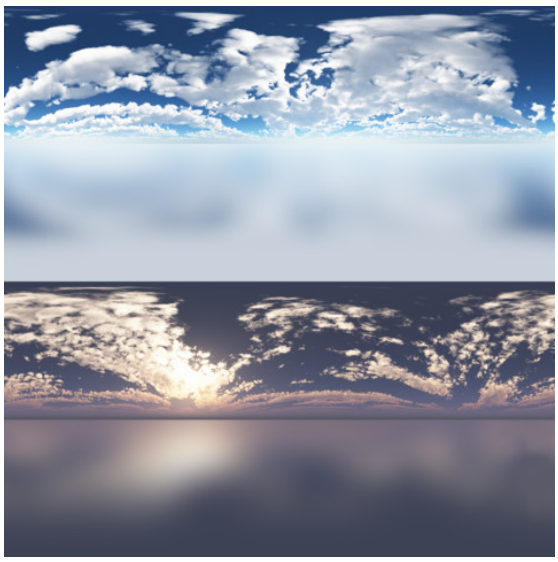


Abbildung A.13.: Free HDR Sky

- Texturen
  - Yughues Free Architectural Materials von Nobiax and Yughues[NY15a]
  - Yughues Free Stone Materials von Nobiax and Yughues[NY15i]
  - Yughues Free Cobble Materials von Nobiax and Yughues[NY15c]
  - Yughues Free Pavement Materials von Nobiax and Yughues[NY15f]
  - Yughues Free Nature Materials von Nobiax and Yughues[NY15e]
  - Yughues Free Sand Materials von Nobiax and Yughues[NY15h]
  - Yughues Free Ground Materials von Nobiax and Yughues[NY15d]
  - Forest Grounds Texture Pack von Brain Blinks[Bli12]
  - QS Materials Nature - Pack Grass vol.2 von Quadrante Studio[Stu16c]
  - QS Materials Nature - Pack Grass vol.1 von Quadrante Studio[Stu16b]
  - Natural Tiling Textures von Terramorph Workshop[Wor16]

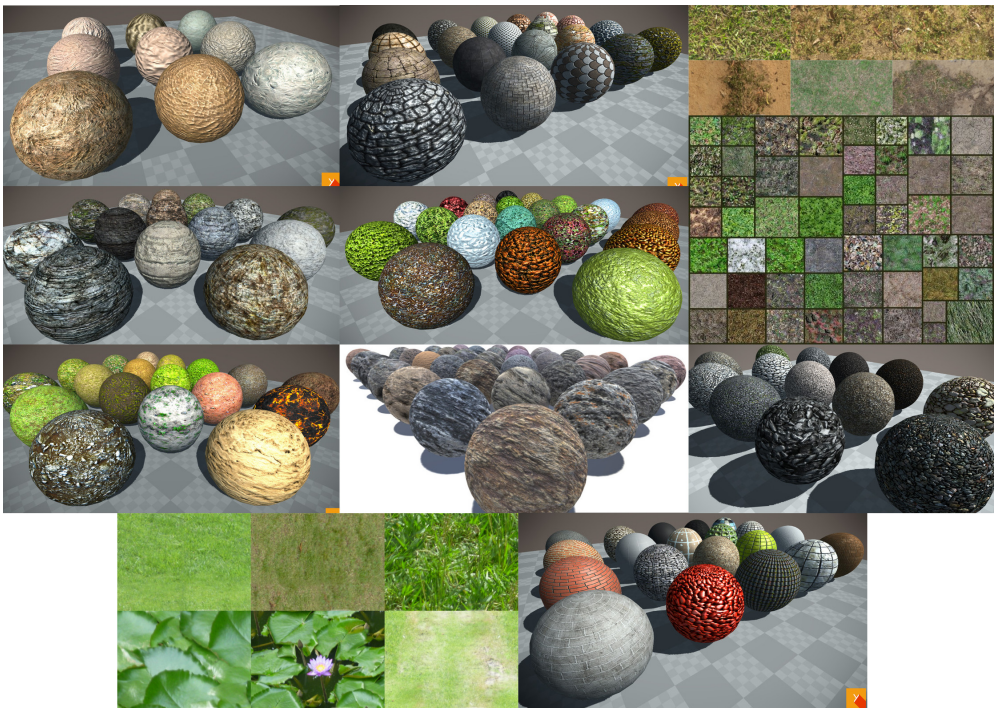


Abbildung A.14.: Texturen

- Gestein
  - Small Stones von DzenGames[Dze16]
  - Realistic Rock von Rakshi Games[Gam16b]
  - Scanned Rocks von RRFreelance[RRF16]
  - Free pack with stones von Igrodelsky[Igr15]
  - HQ Big Rock von NOT_Lonely[NOT15]
  - Mountains Canyons Cliffs von Infinita Studio[Stu16a]
  - Rock & Boulders von Manufactura K4[K415]
  - Rock & Boulders 2 von Manufactura K4[K416]
  - Free Rocks von TripleBrick[Tri14]

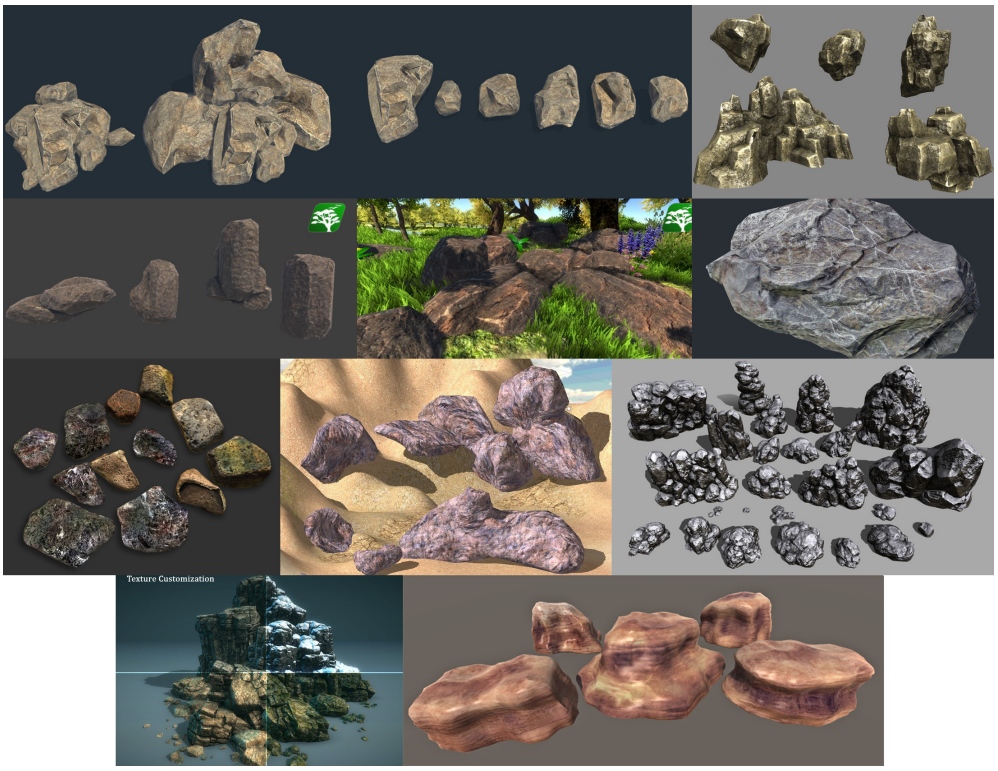


Abbildung A.15.: Felsobjekte

- Strukturen
  - Ancient Ruins in the desert - Part1 von NEKCOM Entertainment[Ent14a]
  - Ancient Ruins in the desert - Part2 von NEKCOM Entertainment[Ent14b]
  - Desert Fortress Pack von Torsten Heldmann[Hel13]

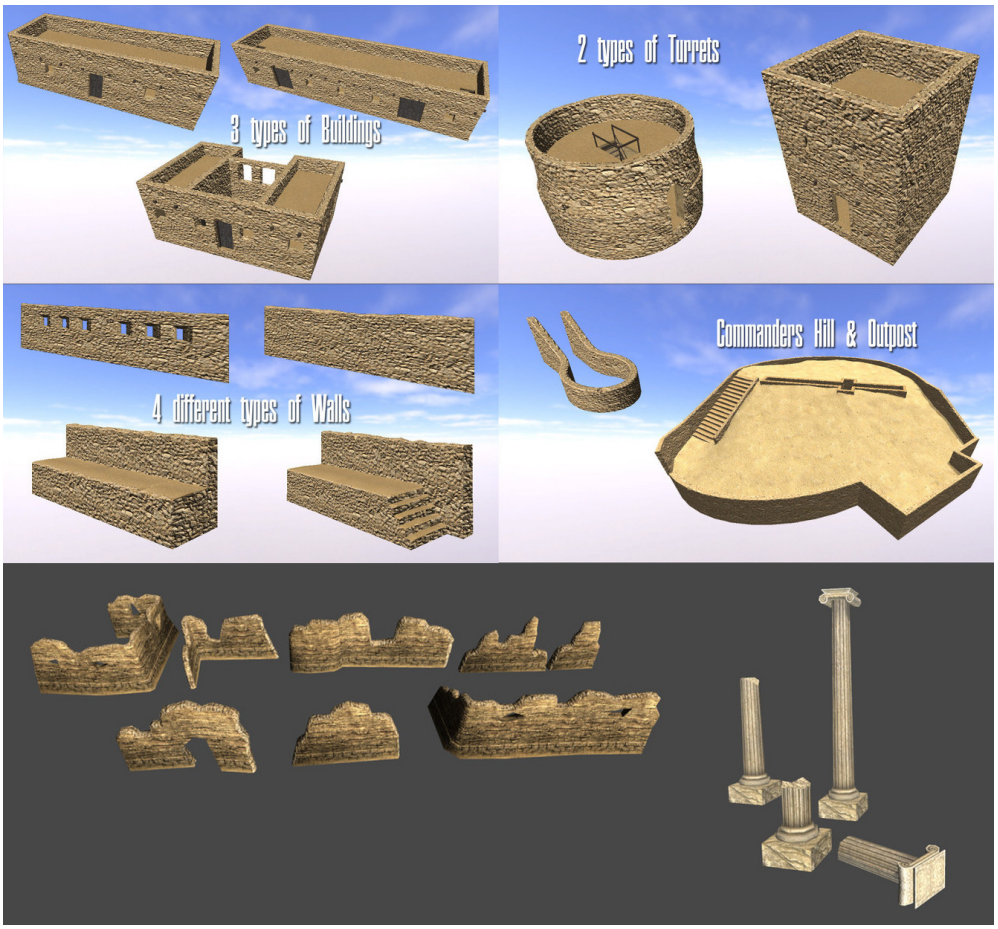


Abbildung A.16.: Strukturen

- Vegetation
  - Free Desert Plants von Lemuria[Lem15]
  - Realistic Tree 9 [Rainbow Tree] von Rakshi Games[Gam16d]
  - Tree (Mediterranean) von Immortal Factory[Fac16]
  - Realistic Tree Pack Vol.1 von PolyFix[Pol15]

- Nature Starter Kit 2 von Snapes[Sna16]
- Yughues Free Plam Trees von Nobiax and Yughues[NY15g]
- Cactus Pack von Sandro Tatinashvili[Tat15]
- Mobile Tree Package von Laxer[Lax16]
- Realistic Tree1 von Rakshi Games[Gam15]
- Realistic Tree 10 von Rakshi Games[Gam16c]
- Yughues Free Bushes von Nobiax and Yughues[NY15b]
- Free SpeedTrees Package von SpeedTree®[Spe16]

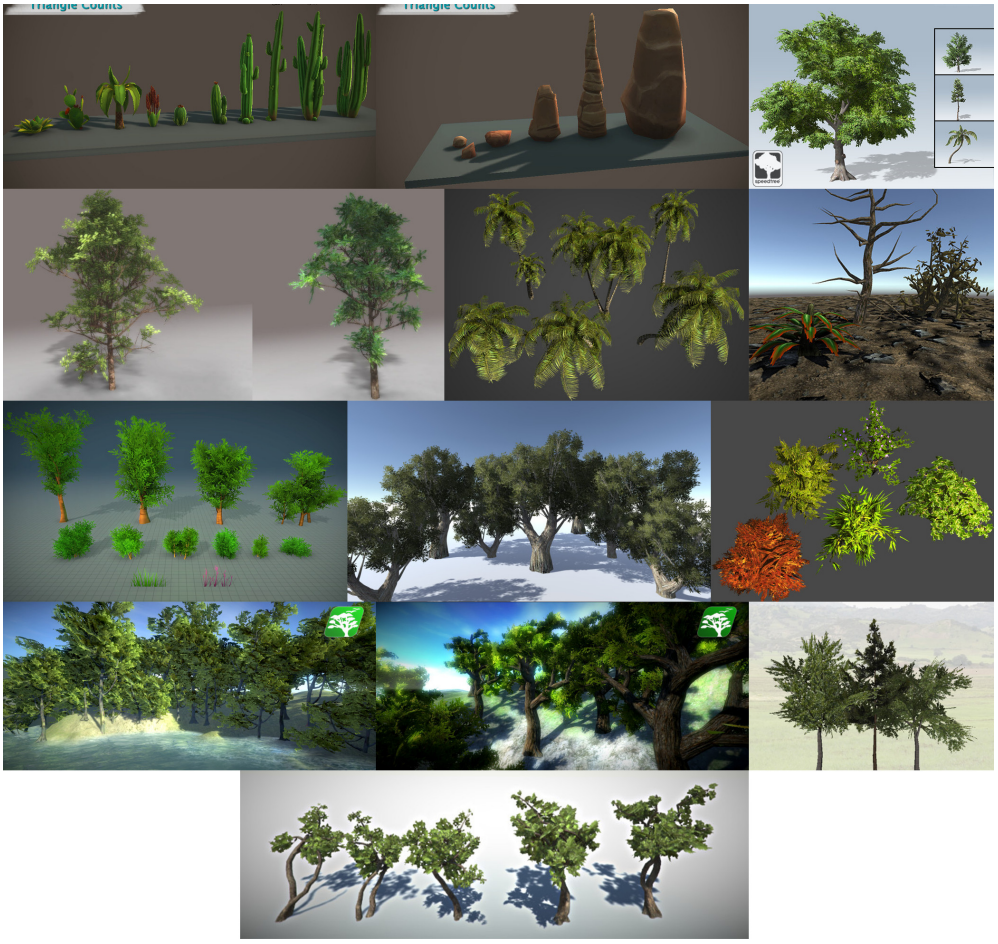


Abbildung A.17.: Vegetation

Literatur

[Ait13] Alastair Aitchison. *Procedural Terrain Splatmapping*. Nov. 2013. URL: https://alastaira.wordpress.com/2013/11/14/procedural-terrain-splatmapping/.

[AJO13] AJOpinnaytetyo. *World Builder*. 25. Sep. 2013. URL: https://www.assetstore.unity3d.com/en/#!/content/11333.

[Ant14] Luis Anton. *Verbose A* pathfinding algorithm in C# for Unity3D*. Okt. 2014. URL: http://playmedusa.com/blog/verbose-astar-pathfinding-algorithm-in-c-for-unity3d/.

[Arm06] Jim Armstrong. *Catmull-Rom Splines*. Jan. 2006. URL: http://algorithmist.net/docs/catmullrom.pdf.

[Bèg] Rémi Bèges. *ZNoise*. URL: https://github.com/Overdrivr/ZNoise.

[Ben12] Stephan Bender. *Die Geometrie der Natur - Fraktale*. 1. Sep. 2012. URL: http://ben.design/blog/fraktale-in-der-natur.

[Bev07] Jason Bevins. *libnoise*. 2007. URL: http://libnoise.sourceforge.net/index.html.

[Bia14] Adrian Biagioli. *Understanding Perlin Noise*. Aug. 2014. URL: http://flafla2.github.io/2014/08/09/perlinnoise.html.

[Bie16] Kai Bielenberg. "Einsatz eines Eyetracker basierten Miningverfahrens für ein Companionsystem". Masterarbeit. HAW Hamburg, Mai 2016. URL: http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/bielenberg.pdf.

[BK] Werner Brilon und Ray Krammes. *Die neuen Entwurfsstandards für Außerortsstraßen im internationalen Vergleich*. URL: http://www.ruhr-uni-bochum.de/verkehrswesen/download/literatur/Bri_KRAM97__cit.pdf.

[Bli12] Brain Blinks. *Forest Grounds Texture Pack*. 21. Dez. 2012. URL: https://www.assetstore.unity3d.com/en/#!/content/303.

[Buc10] Jamis Buck. *Maze Generation: Recursive Backtracking*. Dez. 2010. URL: http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking.html.

[Buc11a] Jamis Buck. *Maze Generation: Binary Tree algorithm*. Feb. 2011. URL: http://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm.html.

[Buc11b] Jamis Buck. *Maze Generation: Kruskal's Algorithm*. Jan. 2011. URL: http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm.html.

[Buc11c] Jamis Buck. *Maze Generation: Prim's Algorithm*. Jan. 2011. URL: http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm.html.

[Buc11d] Jamis Buck. *Maze Generation: Recursive Division*. Jan. 2011. URL: http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm.html.

[Chi15] Jashan Chittesh. *Das Unity-Buch. 2D- und 3D-Spiele entwickeln mit Unity 5*. 2015. ISBN: 978-3-86490-232-1. URL: http://unity-buch.de/.

[Die] Ton Dieker. *Simulation of fractional Brownian motion*. URL: http://www.columbia.edu/~ad3217/fbm/thesisold.pdf.

[Dun05a] Robert Dunlop. *Introduction to Catmull-Rom Slines*. Juli 2005. URL: http://www.mvps.org/directx/articles/catmull/.

[Dun05b] Robert Dunlop. *Introduction to Catmull-Rom Slines*. Juli 2005. URL: http://www.mvps.org/directx/articles/shadeland.old/images/spline1.gif.

[Dze16] DzenGames. *Small Stones*. 9. Jan. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/47791.

[Eib15] David Eibensteiner. "Prozedurale Generierung von endlosen Landschaften mit softwarebasierten Agenten". Masterarbeit. Fachhochschule Oberösterreich, Juni 2015. URL: http://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&cad=rja&uact=8&ved=0ahUKEwi1nNLci

8bOAhXMI8AKHfqXCJIQFghGMAY&url=http%3A%2F%2Ftheses.fh-hagenberg.at%2Fsystem%2Ffiles%2Fpdf%2FEibensteiner15.pdf&usg=AFQjCNHgdjHtOoYGze6U5ANAP5zK-UiWiA.

[Ent14a] NEKCOM Entertainment. *Ancient Ruins in the desert - Part1*. 2. Juli 2014. URL: https://www.assetstore.unity3d.com/en/#!/content/19175.

[Ent14b] NEKCOM Entertainment. *Ancient Ruins in the desert - Part2*. 2. Juli 2014. URL: https://www.assetstore.unity3d.com/en/#!/content/19178.

[Fac16] Immortal Factory. *Tree (Mediterranean)*. 4. Mai 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/61874.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 3. Okt. 2010. ISBN: 978-0321712943. URL: http://martinfowler.com/books/dsl.html.

[Gam15] Rakshi Games. *Realistic Tree1*. 28. Sep. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/44628.

[Gam16a] Gamepedia. *Chunk*. 9. Aug. 2016. URL: http://minecraft-de.gamepedia.com/Chunk.

[Gam16b] Rakshi Games. *Realistic Rock*. 10. Feb. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/51251.

[Gam16c] Rakshi Games. *Realistic Tree 10*. 2. Feb. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/54724.

[Gam16d] Rakshi Games. *Realistic Tree 9 [Rainbow Tree]*. 23. Feb. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/54622.

[GK16] Ulrich Gebhard und Thomas Kistemann. *Landschaft, Identität und Gesundheit. Zum Konzept der Therapeutischen Landschaften*. Springer VS, 2016. ISBN: 978-3-531-19722-7.

[Hag06] Jürgen Hagler. *Bézier-Kurven*. Okt. 2006. URL: http://www.dma.ufg.ac.at/assets/13079/intern/polynomialekurven.gif.

[HAW] Department Informatik der HAW Hamburg. *Living Place Hamburg*. URL: http://livingplace.informatik.haw-hamburg.de/blog/.

[Hel13] Torsten Heldmann. *Desert Fortress Pack*. 21. März 2013. URL: https://www.assetstore.unity3d.com/en/#!/content/6298.

[Hön12] Maximilian Hönig. "Prozedurale Spielweltgenerierung: Dungeons". Masterarbeit. Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, Juli 2012. URL: http://dungen.squarefox.net/downloads/Max_Hoenig%20-%20Masterarbeit.pdf.

[Igr15] Igrodelsky. *Free pack with stones*. 23. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/28296.

[Inc] Firsthand Technology Inc. *Controlling Pain Without Drugs*. URL: http://www.firsthand.com/portfolio/pain.html.

[ITW] ITWissen.info. *DSL (domain specific language)*. URL: http://www.itwissen.info/definition/lexikon/DSL-domain-specific-language-Domaenenspezifische-Sprache.html.

[Jen] Philipp Jenke. *Bachelorarbeit: Parametrisierbare Generierung einer Dynamischen Spielewelt*.

[JL05] Sanjay Jena und Nils Liebelt. *Heuristische Algorithmen am Beispiel des A*-Algorithmus / 8-Puzzle*. 2005. URL: http://www.gm.fh-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/E_gelb/ALA-HeuristischeAlgorithmen-Jena-Liebelt.pdf.

[K415] Manufactura K4. *Rock & Boulders*. 3. Sep. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/2452.

[K416] Manufactura K4. *Rock & Boulders 2*. 29. Aug. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/6947.

[Knö14] Stephan Knödler. "Moderne Techniken zur Generierung prozeduraler Landschaften". Bachelorarbeit. Technische Hochschule Ingolstadt, Aug. 2014. URL: http://www.stephan-knoedler.de/pdf/MTGPL.pdf.

[Kul15] Kulesz. *Infinite terrain generation in Unity 3D - Part 1*. Juli 2015. URL: http://code-phi.com/infinite-terrain-generation-in-unity-3d/.

[Lax16] Laxer. *Mobile Tree Package*. 5. Feb. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/18866.

[Lem15] Lemuria. *Free Desert Plants*. 20. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/32410.

[Les06] Patrick Lester. *A* Pfadfindung für Anfänger*. März 2006. URL: http://www.policyalmanac.org/games/aStarTutorial_de.html.

[Lin15] LinkFang. *Gebrochene Brownsche Bewegung*. Nov. 2015. URL: http://www.linkfang.de/wiki/Gebrochene_Brownsche_Bewegung.

[Mah] Christian Maher. *Working with Simplex Noise*. URL: https://cmaher.github.io/posts/working-with-simplex-noise/.

[Man16] Unity Manual. 2016. URL: https://docs.unity3d.com/Manual/index.html.

[Mat15] Mathepedia. *Bézier-Kurven*. Aug. 2015. URL: http://www.mathepedia.de/Bezierkurven.aspx.

[Mén14] Ricardo J. Méndez. *LibNoise.Unity*. 27. Juli 2014. URL: https://github.com/ricardojmendez/LibNoise.Unity.

[Mic] Microsoft. *Kinect für Xbox One*. URL: http://www.xbox.com/de-DE/xbox-one/accessories/kinect-for-xbox-one.

[mik12] mikera7. *Perlin Noise vs. Simplex Noise*. Aug. 2012. URL: https://clojurefun.wordpress.com/2012/08/05/perlin-noise-vs-simplex-noise/.

[Moj] Mojang. *Minecraft*. URL: https://minecraft.net/en/.

[Mül+] Larissa Müller u. a. *EmotionBike: A Study of Provoking Emotions in Cycling Exergames*.

[Mus] F. Kenton Musgrave. *Procedural Fractal Terrains*. URL: https://www.classes.cs.uchicago.edu/archive/2015/fall/23700-1/final-project/MusgraveTerrain00.pdf.

[Not10] Notch. *Hrmpth. Trade-offs*. 23. Juni 2010. URL: http://notch.tumblr.com/post/729098863/hrmpth-trade-offs.

[NOT15] NOT_Lonely. *HQ Big Rock*. 5. Dez. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/50759.

[NY15a] Nobiax und Yughues. *Yughues Free Architectural Materials*. 1. Apr. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/13234.

[NY15b] Nobiax und Yughues. *Yughues Free Bushes*. 27. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/13168.

[NY15c] Nobiax und Yughues. *Yughues Free Cobble Materials*. 1. Apr. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/12957.

[NY15d] Nobiax und Yughues. *Yughues Free Ground Materials*. 31. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/13001.

[NY15e] Nobiax und Yughues. *Yughues Free Nature Materials*. 31. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/13237.

[NY15f] Nobiax und Yughues. *Yughues Free Pavement Materials*. 1. Apr. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/12952.

[NY15g] Nobiax und Yughues. *Yughues Free Plam Trees*. 27. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/13540.

[NY15h] Nobiax und Yughues. *Yughues Free Sand Materials*. 31. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/12964.

[NY15i] Nobiax und Yughues. *Yughues Free Stone Materials*. 31. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/12962.

[Per85] Ken Perlin. *An Image Synthesizer*. Juli 1985. URL: https://pdfs.semanticscholar.org/e04d/7772b91a83a901408eb0876bbb7814b1d4b5.pdf.

[Pig14] Anthony Pigeot. *Experiments: Perlin noise 2D and 3D worlds*. Jan. 2014. URL: http://anthonypigeot.com/experiments-perlin-noise-2d-and-3d-worlds/.

[Pol15] PolyFix. *Realistic Tree Pack Vol.1*. 2. Dez. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/50418.

[Pro16] ProAssets. *Free HDR Sky*. 5. Mai 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/61217.

[Pul15] Walter D. Pullen. *Perfect Maze Creation Algorithms*. Nov. 2015. URL: http://www.astrolog.org/labyrnth/algrithm.htm.

[Qua] Niedersächsisches Landesinstitut für schulische Qualitätsentwicklung. *Bézierkurven*. URL: http://nibis.ni.schule.de/~lbs-gym/AnalysisTeil4pdf/Bezierkurven.pdf.

[RRF16] RRFreelance. *Scanned Rocks*. 19. Jan. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/22070.

[Scha] Jörg Schade. *Radfahren macht Spaß*. URL: http://www.radfahren-macht-spass.de/startseite/wissenswertes/.

[Schb] Nico Scheiffert. *Splines-Was ist das?* URL: http://mathematik.de/spudema/spudema_beitraege/beitraege/scheiffert/abschnitt1.htm.

[Sch10] Katrin Scharnowski. *Prozedurale Modellierung: Terrains*. Juli 2010. URL: http://www.vis.uni-stuttgart.de/plain/seminare/computerspiele/ProzMod/ausarbeitung.pdf.

[sel16] selfhtml. *XML*. 15. Juli 2016. URL: https://wiki.selfhtml.org/wiki/XML.

[SF] Thorsten Schmidt und David Fuchs. *A-Stern Algorithmus*. URL: http://www.geosimulation.de/methoden/a_stern_algorithmus.htm.

[Sme+09] Ruben M. Smelik u. a. *A Survey of Procedural Methods for Terrain Modelling*. 2009. URL: http://www.cg.its.tudelft.nl/Publications-new/2009/SDGTB09a/SDGTB09a.pdf.

[Sna16] Snapes. *Nature Starter Kit 2*. 14. Jan. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/52977.

[Spe16] SpeedTree®. *Free SpeedTrees Package*. 13. Jan. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/29170.

[Stu16a] Infinita Studio. *Mountains Canyons Cliffs*. 29. Jan. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/53984.

[Stu16b] Quadrante Studio. *QS Materials Nature - Pack Grass vol.1*. 3. Aug. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/21113.

[Stu16c] Quadrante Studio. *QS Materials Nature - Pack Grass vol.2*. 11. Juli 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/32020.

[sty15a] styanton. *Maze Generator*. Sep. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/38689.

[sty15b] styanton. *Maze Generator*. 21. Sep. 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/38689.

[Tat15] Sandro Tatinashvili. *Cactus Pack*. 26. März 2015. URL: https://www.assetstore.unity3d.com/en/#!/content/11547.

[Tea] MathWorld Team. *B-Spline*. URL: http://mathworld.wolfram.com/B-Spline.html.

[Tec] Unity Technologies. *Height Tools*. URL: https://docs.unity3d.com/Manual/terrain-Height.html.

[Tec16a] Unity Technologies. 2016. URL: https://unity3d.com/.

[Tec16b] Unity Technologies. 2016. URL: https://www.assetstore.unity3d.com/en/.

[Til12] Stefan Tilkov. *Domain Specific Languages*. 10. Juli 2012. URL: https://www.innoq.com/de/articles/2012/07/domain-specific-languages/.

[Tra15] Travelbook. *Das sind die steilsten Straßen der Welt*. 30. Sep. 2015. URL: http://www.travelbook.de/welt/Bis-zu-37-Prozent-Steigung-Die-steilsten-Strassen-der-Welt-598883.html.

[Tri14] TripleBrick. *Free Rocks*. 7. Juli 2014. URL: https://www.assetstore.unity3d.com/en/#!/content/19288.

[tut] tutorialspoint.com. *Computer Graphics Curves*. URL: http://www.tutorialspoint.com/computer_graphics/computer_graphics_curves.htm.

[Twe] Twexa. *What is Unity and what can I do with it?* URL: http://twexa.com/what-is-unity3d.

[Twi03] Christopher Twigg. *Catmull-Rom splines*. März 2003. URL: https://www.cs.cmu.edu/~462/projects/assn2/assn2/catmullRom.pdf.

[Ull13] Holger Ullmann. *Simplex - multidimensionale Tetraeder, 4. Dimension*. Juli 2013. URL: http://tetraktys.de/geometrie-4.html.

[UMN09] Universität Ulm, Otto-von-Guericke Universität Magdeburg und Leibniz-Institut für Neurobiologie. *sfb transregio 62 Companion Technology*. 1. Jan. 2009. URL: http://www.sfb-trr-62.de/.

[Vuy04] Pascal Vuylsteker. *B Spline vs Bezier curves*. Apr. 2004. URL: http://escience.anu.edu.au/lecture/cg/Spline/bSplineVsBezier.en.html.

[W3S] W3Schools. *XML Tutorial*. URL: http://www.w3schools.com/xml/.

[w3s] w3schools.com. *DTD Tutorial*. URL: http://www.w3schools.com/xml/xml_dtd_intro.asp.

[Wik12] DGL Wiki. *Perlin Noise*. März 2012. URL: https://wiki.delphigl.com/index.php/Perlin_Noise.

[Wik16] WikiPedalia. *Anfahren und Anhalten*. 6. Juli 2016. URL: https://wikipedalia.com/index.php?title=Anfahren_und_Anhalten.

[Win] Brian Winn. *Lecture 6 - Intro to Unity3D*. URL: https://www.coursera.org/learn/game-development/lecture/IQDE7/intro-to-unity3d.

[Wor16] Terramorph Workshop. *Natural Tiling Textures*. 29. Feb. 2016. URL: https://www.assetstore.unity3d.com/en/#!/content/35173.

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

Hamburg, 17. November 2016 Katharina Mulack