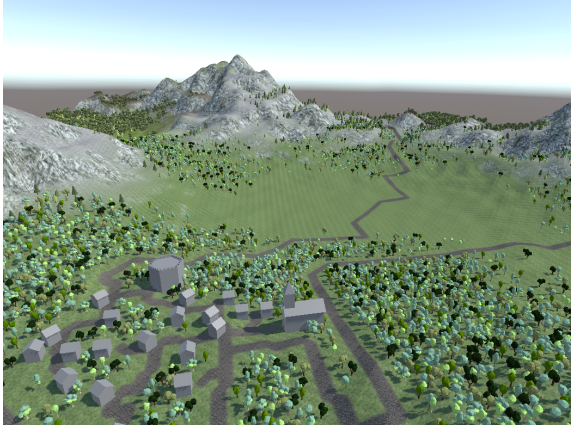
# Prozedurale Generierung offener Spielwelten

*Procedural Generation of Open Game World*



**Masterarbeit**

im Rahmen des Studiengangs
**Interaktive Medien**
der Hochschule Anhalt & der Martin-Luther-Universität Halle-Wittenberg

vorgelegt von
**Marcus Gagelmann**
Matrikelnummer: 5041161 / 217221436
E-Mail: marcus.gagelmann@student.uni-halle.de

Erstgutachter
**Prof. Dr. Stefan Schlechtweg**

Zweitgutachter
**Prof. Dr. Alexander Carôt**

Halle (Saale), den 11. November 2023

## Selbstständigkeitserklärung

Diese Arbeit mit dem Titel „Prozedurale Generierung offener Spielwelten“ wurde von mir selbständig verfasst und in gleicher oder ähnlicher Fassung noch nicht in einem anderen Studiengang als Prüfungsleistung vorgelegt. Ich habe keine anderen als die angegebenen Hilfsmittel und Quellen, einschließlich der angegebenen oder beschriebenen Software, verwendet.

(Marcus Gagelmann)
Halle (Saale), den 11. November 2023

**Kurzfassung** Die Kurzfassung sollte nicht länger als einige Absätze sein. Für die deutsche Kurzfassung muss eine englische Version existieren, die eine genaue Übersetzung der deutschen Fassung sein muss.

**Abstract** The abstract should not be longer than some paragraphs. There must be an English translation of the the German abstract, which has to be the exact translation of the German version.

# Inhaltsverzeichnis


**1 Einleitung** **1**
1.1 Motivation . . . 1
1.2 Verwandte Arbeiten . . . 1
1.2.1 Minecraft . . . 1
1.2.2 The Elder Scrolls IV: Oblivion . . . 2
1.2.3 Kommerzielle Landschaftsgeneratoren . . . 3
1.3 Zielstellung . . . 4
1.4 Gliederung . . . 5

**2 Vorangegangene Arbeiten** **7**
2.1 Terraingenerator . . . 7
2.1.1 Ziele . . . 7
2.1.2 Ergebnisse . . . 8
2.2 Biomgenerator . . . 10
2.2.1 Ziele . . . 10
2.2.2 Ergebnisse . . . 11

**3 Systemüberblick** **15**
3.1 Das Pipeline-Modell . . . 15
3.2 Texturen als Berechnungsgrundlage . . . 15

**4 Texturierungssystem** **17**
4.1 Theoretische Grundlagen und Verwandte Arbeiten . . . 17
4.1.1 Die Grafik-Pipeline . . . 17
4.1.2 Shader und Shader-Graph . . . 20
4.1.3 Texture-Mapping und Texture Splatting . . . 21
4.2 Konzeption des Texturierungssystems . . . 26
4.2.1 Zielsstellung . . . 26
4.2.2 Diskussion und Konzept . . . 27
4.3 Implementierung des Texturierungssystems . . . 40
4.3.1 Vorberechnung des Shader-Inputs . . . 41
4.3.2 Umsetzung der Shader . . . 43
4.4 Performance des Texturierungssystems . . . 49

**5 Vegetationsgenerator** **53**
5.1 Theoretische Grundlagen und Verwandte Arbeiten . . . . . 53
5.1.1 Verwandte Systeme zur Generierung von Vegetationsszenen . . 53
5.1.2 Ansätze zur prozeduralen Asset-Verteilung . . . . . 57
5.1.3 Performance in großen Vegetationsszenen . . . . . 59
5.2 Konzeption des Vegetationsgenerators . . . . . 60
5.2.1 Zielsstellung . . . . . 61
5.2.2 Diskussion und Konzept . . . . . 62
5.3 Implementierung des Vegetationsgenerators . . . . . 71
5.3.1 Vorbereitung der Vegetationsgenerierung . . . . . 73
5.3.2 Umsetzung der Vegetationsgenerierung . . . . . 73
5.4 Performance des Vegetationsgenerators . . . . . 79

**6 Siedlungsgenerator** **83**
6.1 Theoretische Grundlagen und Verwandte Arbeiten . . . . . 83
6.1.1 Generierung urbaner Regionen durch Inferenz . . . . . 83
6.1.2 Siedlungserstellung durch prozedurale Generierung . . . . . 84
6.2 Konzeption des Siedlungsgenerators . . . . . 87
6.2.1 Zielstellung . . . . . 87
6.2.2 Diskussion und Konzept . . . . . 88
6.3 Implementierung des Siedlungsgenerators . . . . . 99
6.3.1 Generierung der Siedlungskarte . . . . . 99
6.3.2 Instanziierung der Siedlungsmodelle . . . . . 104
6.4 Performance des Siedlungsgenerators . . . . . 108

**7 Ergebnisse und Evaluation** **109**

**8 Zusammenfassung und Ausblick** **111**

# 1 Einleitung

-intro + überblick

## 1.1 Motivation

## 1.2 Verwandte Arbeiten

Der folgende Abschnitt beschäftigt sich mit verwandten Anwendungen, deren Ansätze für die Durchführung der Arbeit interessant sind. Zuerst werden dabei zwei digitale Spiele betrachtet, die unterschiedliche Ansätze zur Erstellung ihrer Spielwelt umsetzen. Im Anschluss wird auf kommerzielle Lösungen zur Weltengenerierung eingegangen.

### 1.2.1 Minecraft

Minecraft ist ein Open-World-Sandbox-Survival-Spiel, das im Jahr 2009 veröffentlicht wurde, bis heute erfolgreich ist und aktiv gewartet wird. In Minecraft werden beliebig viele vollständig prozedurale 3D-Welten generiert, die aus Blockelementen gleicher Größe aufgebaut sind und mit denen die Spieler interagieren können. Diese Blockelemente repräsentieren verschiedene Materialien wie Erde, Holz und Erz.

Die Spielwelt von Minecraft besteht aus verschiedenen prozedural generierten Elementen, darunter:

- Terrain mit Höhlen und Gewässern
- unterschiedliche Vegetationsarten
- Siedlungen, in denen NPCs (Nicht-Spieler-Charaktere) leben und handeln

Eine Besonderheit von Minecraft ist, dass die Generierung der Welten durch einen sogenannten Seed beeinflusst wird. Ein Seed ist eine Zahl, die als Initialwert für den Zufallszahlengenerator verwendet wird. Da solche Generatoren deterministisch arbeiten, bedeutet dies, dass bei gleicher Seed-Eingabe immer das gleiche Ergebnis erzielt wird. Dies ermöglicht es den Spielern, bekannte Welten in Minecraft erneut zu generieren [27] [57].

Ein weiterer Vorteil dieser Technik besteht darin, dass gespeicherte Welten nicht vollständig auf der Festplatte abgelegt werden müssen. Stattdessen wird die umgebende Welt bei jedem Ladevorgang erneut prozedural generiert. Der verringerte Speicherbedarf trägt zur Verbesserung der Performance bei.

Die verwendung des Seed-Systems wirkt sich negativ auf die Flexibilität bei der Weltengenerierung in Minecraft aus. Spieler können zwar den Seed auswählen, aber die prozedurale Generierung erfolgt weitgehend unvorhersehbar auf dessen Grundlage [57].

### 1.2.2 The Elder Scrolls IV: Oblivion

Oblivion ist ein Open-World-Rollenspiel aus dem Jahr 2006. Bei jedem Start eines neuen Spiels bewegt sich der Spieler immer in derselben vorgefertigten 3D-Welt. Die Welten können sich lediglich in einigen wenigen Details von den Welten anderer Spielstände unterscheiden. Die Spielwelt von Oblivion besteht aus verschiedenen Modell-Assets unterschiedlicher Art und Größe.

Während der Entwicklung des Spiels wurde ein hybrider Ansatz zur Erstellung der Inhalte der Spielwelt angewendet. Dieser Ansatz umfasst einen ausbalancierten Mix aus prozeduralen Inhalten, manuell angefertigten Inhalten und prozedural assistierten Inhalten.

Prozedurale Inhalte sind kostengünstig zu generieren und können theoretisch in unbegrenzter Menge erstellt werden. Ein Beispiel dafür ist die Platzierung einzelner Bäume, Sträucher und Gräser in der Welt von Oblivion, die durch einen Algorithmus erfolgt. Diese Methode ist zwar günstig und zeitsparend, durch sie können jedoch nur schwierig Inhalte mit tieferer Bedeutung integriert werden. Prozedurale Inhalte können häufig repetitiv auf den betrachter wirken.

Manuell angefertigte Inhalte sind gut durchdacht, fügen sich nahtlos ins Spiel ein und verleihen den Spielinhalten eine individuelle Bedeutung. Diese Inhalte sind jedoch teurer, da Künstler für ihre Erstellung bezahlt werden müssen. Außerdem sind diese Art von Inhalten in der Regel fragiler, da eine erhöhte Wahrscheinlichkeit besteht, dass Fälle eintreten, die vom Autor nicht berücksichtigt wurden. Dies kann zu Fehlern in der Zielanwendung führen. Ein Beispiel für manuell angefertigte Inhalte

in Oblivion sind Städte, Gebäude und Straßen, die von Autoren in der Welt platziert wurden.

Prozedural assistierte Inhalte sind eine Kombination aus beiden Ansätzen. Der gesamte Inhalt hat im großen Zusammenhang Bedeutung und hebt die Prämisse des Spiels hervor, während kleinere Details prozedural generiert werden, um Kosten zu sparen und eine größere Spielwelt zu ermöglichen. Diese prozeduralen Inhalte können jedoch auch hier repetitiv und bedeutungslos wirken und sollten deshalb vorausschauend eingeplant werden. Ein Beispiel dafür ist das Terrain in Oblivion, das vom Autor gemalt wurde, während Algorithmen darauf verschiedene Effekte wie Erosion berechnen.

### 1.2.3 Kommerzielle Landschaftsgeneratoren

Neben Spielen existieren auch einige kommerzielle Produkte zur Erstellung von Landschaftsmodellen und Vegetation in hoher Qualität. Der Fokus dieser Anwendungen liegt hierbei auf fotorealistischen Szenen. Die Arbeitsabläufe zur Erstellung dieser Szenen werden oft prozedural assistiert durchgeführt. Das bedeutet, dass der Nutzer viel manuelle Arbeit durchführen und zahlreiche Designentscheidungen selbständig treffen muss, wobei er jedoch von prozeduralen Algorithmen unterstützt wird. Diese Algorithmen können beispielsweise Erosion auf dem Terrain berechnen. Einige Beispiele für solche kommerziellen Produkte sind:

- Terragen [44]
- World Machine [33]
- World Creator [17]
- Gaia [62]

Da der Quellcode solcher kommerziellen Anwendungen in der Regel nicht veröffentlicht wird, kann nicht weiter auf die verwendeten Methoden dieser Anwendungen eingegangen werden. Es fällt jedoch auf, dass keines dieser kommerziellen Produkte die prozedurale Generierung von Siedlungen unterstützt. Auch die prozedurale Verteilung von Biomen ist in keiner der genannten Anwendungen möglich. Daher sind die kommerziellen Landschaftsgeneratoren nicht für die Generierung einer endlosen Welt geeignet, wie es beispielsweise in Minecraft der Fall ist (siehe Abschnitt 1.2.1).

## 1.3 Zielstellung

In dieser Arbeit soll ein Konzept für ein prozedurales System entworfen werden, das in der Lage ist, digitale 3D-Welten zu generieren, die Terrain, Biome, Vegetation und Siedlungen enthalten. Das entwickelte System und dessen Ausgabe sollen dabei folgende Anforderungen bestmöglich erfüllen: Realismus, Flexibilität, Performance und Benutzbarkeit.

1. **Realismus** umfasst die Generierung von Welten, die den Eindruck realer Landschaften und Siedlungen der Erde vermitteln. Dabei wird angestrebt, dass die erzeugten digitalen Umgebungen für das menschliche Auge als vertraut und plausibel wahrgenommen werden.
2. **Flexibilität** soll den Nutzern ermöglichen, individuell Einfluss auf die Berechnungen und den Output zu nehmen. Hierbei wird ein breites Spektrum an einstellbaren Parametern bereitgestellt, um den Anwendern die Möglichkeit zu geben, die erzeugten Welten an ihre spezifischen Bedürfnisse anzupassen. Dadurch soll das prozedurale System universell für verschiedenste digitale Anwendungen, insbesondere digitale Spielen, einsetzbar gemacht werden.
3. **Performance** wird im Rahmen dieser Arbeit daran gemessen, ob die Erstellung der Welten und ihrer einzelnen Teilbereiche schnell genug erfolgt, um den Nutzern eine nahtlose Erfahrung zu bieten. Die Laufzeiten der einzelnen Programmroutinen sollten dabei vom Nutzer nicht als störende Unterbrechung wahrgenommen werden.
4. **Benutzbarkeit** beschreibt in dieser Arbeit zum einen, wie intuitiv die Anwendung von sowohl erfahrenen als auch neuen Nutzern bedient werden kann. Darüber hinaus gewährleistet eine hohe Benutzbarkeit, dass die Anwendung zuverlässig funktioniert, um eine konsistente und verlässliche Bedienung und Ausgabe zu erreichen. Zuletzt sollte der Arbeitsaufwand für den Nutzer möglichst gering gehalten werden.

Es ist jedoch wichtig anzumerken, dass die oben genannten Anforderungen nicht alle vollständig erfüllt werden können, da einige der Ziele miteinander konkurrieren. So führt ein höherer Grad an Realismus oft zu komplexeren Berechnungen und längeren Laufzeiten, was die Performance beeinträchtigt. Ebenso resultiert aus einem höherern Grad an Flexibilität, durch eine Vielzahl von einstellbaren Parametern, möglicherweise eine eingeschränkte Benutzbarkeit, weil die Anwendung für den Anwender unübersichtlicher wird.

Das Finden einer Balance zwischen den konkurrierenden Anforderungen ist entscheidend. Es ist klar, dass nicht alle Anforderungen in vollem Maße erfüllt werden

können. Daher werden im Rahmen dieser Arbeit die Kriterien Realismus und Flexibilität besonder priorisiert, um die Qualität des Outputs für erfahrene Nutzer zu maximieren. Gleichzeitig wird jedoch angestrebt, ein Mindestmaß an Performance und Benutzbarkeit zu gewährleisten, um eine insgesamt zufriedenstellende Nutzungserfahrung zu bieten.

## 1.4 Gliederung

# 2 Vorangegangene Arbeiten

Die vorliegende Masterarbeit baut in ihren Untersuchungen auf zwei vorhergehenden Projektarbeiten [14] [13] auf. Im Folgenden werden die Ziele und die Ergebnisse beider Projekte kurz zusammengefasst, da diese Vorarbeit eine zentrale Rolle in den folgenden Kapiteln spielen wird.

## 2.1 Terraingenerator

Im Rahmen des ersten Projektes [14] wurde eine Möglichkeit zur prozeduralen Generierung von Höhenkarten für die Verwendung in digitalen Landschaften diskutiert. Im praktischen Teil dieser Projektarbeit entstand eine Anwendung, die das besprochene Konzept zur Landschaftsgenerierung umsetzt.

### 2.1.1 Ziele

Das Ziel dieser Arbeit war es, ein Konzept für ein prozedurales System zu entwerfen, das einem Nutzer ermöglicht, möglichst vielfältige dreidimensionale Höhenkarten nach seinen Wünschen zu generieren. Die Höhenkarten sollten für die glaubwürdige Verwendung in digitalen Landschaften geeignet sein. Folgende Kriterien wurden dabei berücksichtigt:

1. **Die Generierung der Höhenkarten erfolgt prozedural**, wodurch die deterministische Eigenschaft der Anwendung sichergestellt wird und immer wieder die gleichen Inhalte unter denselben Ausgangsbedingungen erzeugt werden können. Dadurch können auch komplexere Inhalte zeit- und platzsparend entwickelt und weitergegeben werden.
2. **Die Generierung vielfältiger Höhenkarten wird ermöglicht.** Zu diesem Zweck ist eine starke Parametrisierung des Systems nötig. Mithilfe dieser Parameter ist es dem Nutzer möglich, die unterschiedlichen Aspekte der Höhenkarte zu beeinflussen.

3. **Nutzer können ihre mentalen Modelle von Höhenkarten in einer angemessener Zeit umsetzen.** Hierfür wurde eine ansprechende Visualisierung der Berechnungen als 3D-Modell sowie die Echtzeitberechnung der Höhenkarten angestrebt. Dadurch können die Auswirkungen einer Parameteränderung direkt für den Nutzer sichtbar gemacht werden, wodurch ein responsiver Workflow gewährleistet wird.

### 2.1.2 Ergebnisse

In der Projektarbeit zum Terraingenerator konnte das Konzept einer Anwendung entwickelt werden, die in der Lage ist, dreidimensionale landschaftsähnliche Höhenkarten zu berechnen und zu visualisieren. Dieses Konzept wurde im Anschluss als Prototyp in der Unity Game Engine implementiert.

Jede von der Anwendung generierte Höhenkarte besteht aus einer Kombination mehrerer Perlin-Noise-Abtastungen [41]. Insgesamt werden hierbei drei 2D-Noise-Maps in Frequenzen mit unterschiedlichen Größenordnungen überlagert. Die Noise-Map mit der niedrigsten Frequenz stellt dabei die Basis der Höhenkarte dar. Ihre Werte werden an jeder diskreten Koordinate der Karte mit den Werten der anderen beiden Noise-Maps zu einer einzelnen Höhenkarte summiert. Beeinflusst werden die Ausprägungen der Perlin-Noise-Maps durch 12 unterschiedliche benutzerdefinierte Parameter, die in die folgenden drei Funktionsgruppen unterteilt werden können:

1. **Die Basisparameter** bestimmen das Ausmaß der Höhenkarte in Länge und Breite und darüber hinaus den Offset der Basis-Noise-Map entlang der X- und der Y-Koordinate. Sie werden verwendet, um eine geeignete Basis für die weiteren Modellierungsschritte zu finden.

2. **Die Höhenparameter** beeinflussen die Frequenz der Basis-Noise-Map sowie das Intervall an potenziellen Höhenwerten, in denen die einzelnen Punkte der Karte liegen. Hiermit ist es möglich, die Häufigkeit des Wechsels von Bergen und Tälern, die Ausprägung von Höhenunterschieden oder den mittleren Steigungswinkel von Berghängen zu beeinflussen.

3. **Die Noise-Layer Parameter** fließen in die Berechnung der Noise-Layer, also der zusätzlichen Noise-Maps in höherer Frequenz als der Basis-Noise-Map, ein. Über die Parameter können die Frequenz sowie die Ausprägung der Höhenunterschiede angepasst werden. Niedrige und höhere Höhenlagen besitzen dabei gesonderte Parameter und werden auch gesondert mit den Basiswerten des Terrains verrechnet. Hierdurch wird das glatte Basis-Terrain mit realistischen zufälligen Unebenheiten versehen, die in den verschiedenen Höhenlagen variieren.

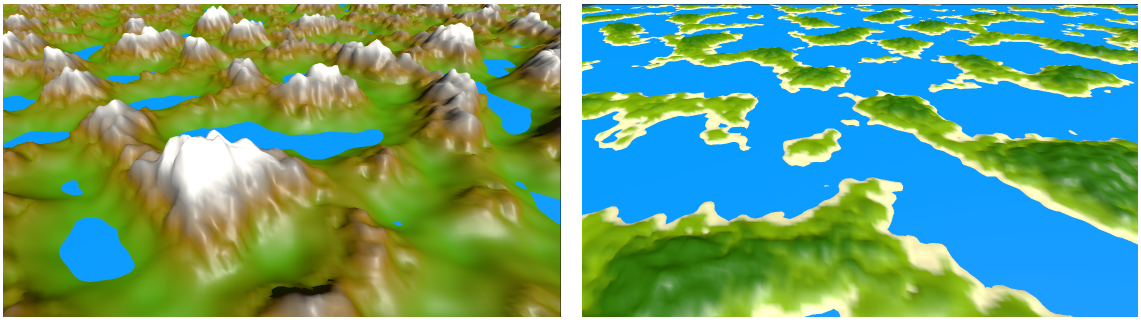


**Abbildung 2.1:** Verschiedene generierbare Höhenkarten
Quelle: Eigene Darstellung

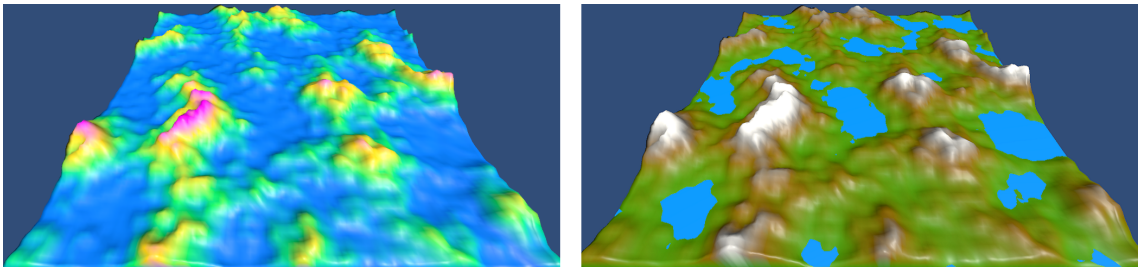


**Abbildung 2.2:** Standard Terrain mit unterschiedlichen Farbgradienten
Quelle: Eigene Darstellung

Sämtliche Parameter werden in der Anwendung zusammen mit den Perlin-Noise-Maps zu einer Höhenkarte mit endlich vielen Koordinaten und zugehörigen Höhenwerten verrechnet. Diese Daten werden im Anschluss in Form eines Meshs innerhalb der Unity-Game-Engine visualisiert. Dabei wird ein Farbgradient eingesetzt, der die Darstellung von unterschiedlichen Höhenlagen durch beispielsweise das Einzeichnen einer Schneegrenze ermöglicht. Zusätzlich wird eine blau eingefärbte Ebene in das Terrain gesetzt, welche den Wasserspiegel der Welt repräsentiert. Die Höhe des Wasserspiegels kann mittels eines Parameters angepasst werden.

Die Ziele des Projekts wurden weitestgehend erreicht. Mithilfe des Prototyps ist es möglich, eine Vielzahl an Höhenkarten zu generieren, die zur Verwendung als Basis einer Landschaft geeignet sind. Einige Beispiele dafür sind in Abbildung 2.1 dargestellt. Außerdem konnte die Generierung prozedural umgesetzt werden, wodurch auch größere Höhenkarten unter relativ geringem Zeit- und Platzaufwand entwickelt und verbreitet werden können. Zuletzt ist es gelungen, dem Nutzer seine Ergebnisse visuell und responsiv zu präsentieren. Dafür sorgt sowohl die Mesh-Berechnung in Unity mit Farbgradient (siehe Abbildung Abbildung 2.2) als auch die Aktualisierung der Berechnungen in Echtzeit.

Anzumerken ist, dass die Masse an einstellbaren Parametern im Prototypen zwar einen positiven Einfluss auf die potenzielle Vielfalt der Höhenkarten hat, aber sich

negativ auf die Nutzbarkeit auswirkt. Ohne eine Anleitung, genügend Einarbeitungszeit oder ein durchdachtes User-Interface könnten Nutzer Schwierigkeiten bei der Umsetzung ihrer mentalen Landschaftsmodelle haben.

## 2.2 Biomgenerator

Die zweite Projektarbeit [13] diskutiert Möglichkeiten zur Aufteilung von prozedural generiertem Terrain in realistische Biome. Dabei konnte ein Prototyp entwickelt werden, der einen der diskutierten Ansätze praktisch umsetzt. Als Input für diesen Algorithmus dienen Höhenkarten, die mithilfe des Prototyps aus der ersten Projektarbeit, dem Terraingenerator, erzeugt werden.

### 2.2.1 Ziele

Diese Arbeit zielte darauf ab, Möglichkeiten zur Realisierung eines Biom-Generators zu diskutieren und einen Prototyp zu implementieren, der eines der besprochenen Konzepte umsetzt. Dabei soll einem Nutzer mithilfe der Anwendung ermöglicht werden, dreidimensionale Höhenkarten in realistische Biom-Gebiete einzuteilen. Der Output des Programms soll für die zukünftige Verwendung in digitalen Landschaften geeignet sein. Die folgenden Ziele konnten für dieses Projekt spezifiziert werden:

1. **Die Einteilung der Höhenkarten in Biome erfolgt automatisch.** Da der Input der Anwendung prozedural generiert wird und dadurch beinahe beliebige Ausmaße annehmen kann, wäre ein Tool zur rein manuellen Einteilung der Höhenkarten in Biom-Gebiete nicht angebracht. Schlecht skalierende Bearbeitungszeiten für den Nutzer bei steigender Kartengröße wären die Folge.
2. **Die Einteilung durch den Nutzer ist individualisierbar.** Zu diesem Zweck ist die Parametrisierung der unterschiedlichen Berechnungsschritte des Systems notwendig. Über solche Parameter wird es dem Nutzer ermöglicht, die Berechnung der Biome nach seinen Bedürfnissen zu beeinflussen.
3. **Es werden Möglichkeiten zur Nachbearbeitung der Biome für den Nutzer bereitgestellt.** Aufgrund des automatisch-parametrisierten Ansatzes wird es dem Nutzer nicht möglich sein, die Berechnungsergebnisse bis auf das kleinste Detail nach seinen Wünschen zu gestalten. Diesem Problem wirken sowohl automatische als auch manuelle Tools zur Nachbearbeitung der Biom-Aufteilungen entgegen.

### 2.2.2 Ergebnisse

Im Rahmen der Projektarbeit zum Biom-Generator wurden verschiedene Ansätze zur Umsetzung eines Biom-Generators zusammengetragen. Der vielversprechendste dieser Ansätze ist anschließend erfolgreich als Anwendung in der Unity Game Engine implementiert worden.

Einleitend führt die Arbeit das Klassifikationsschema für die Biome der Erde nach Robert H. Whittaker [56] ein. Dieses Schema reduziert das Vorkommen der unterschiedlichen Biome der Erde auf die Ausprägung des vorherrschenden Jahresniederschlags sowie der Jahrestemperaturen. Da dieses Schema in der Lage ist, trotz geringer Komplexität Biome zuverlässig zu bestimmen, wurde es als Basis für die weitere Arbeit ausgewählt.

Den Hauptteil der Untersuchung bildet der Vergleich von Weltkarten am Beispiel aktueller digitaler Spiele. Dabei kristallisierten sich folgende Eigenschaften heraus, die bei der Einteilung von dreidimensionalem Terrain in Biome das Landschaftsbild grundlegend beeinflussen:

1. **Die Temperatureinflüsse auf die Biome** einer digitalen Landschaft können entweder nach dem realen Vorbild an den Breitengraden der zugehörigen Welt orientiert sein, oder aber diese realistische Eigenschaft zugunsten anderer Design-Aspekte ignorieren. Existieren große Höhenlagen im Terrain, so kann es ebenfalls zugunsten des Realismus sinnvoll sein, diese bei den Temperaturberechnungen zu berücksichtigen.

2. **Die Feuchtigkeitseinflüsse auf die Biome** einer digitalen Landschaft treten in der realen Welt durch räumliche Nähe zu Gewässern und Windrichtungen auf. Auch hierfür konnte beobachtet werden, dass einige Spiele diese Eigenschaften präzise Umgesetzt haben, um dem Spieler eine möglichst realistische Welt bieten zu können. Andere Spiele ignorieren higegen die Wirkung von Feuchtigkeitseinflüssen bewusst für besser ein abgestimmtes Game Design.

3. **Die Übergänge zwischen Biomen** in einer digitalen Landschaft können entweder stetig oder diskret ausgeprägt sein. Stetige Biome besitzen einen fließenden Übergang ineinander und stellen den realistischeren der beiden Ansätze dar. Im Gegensatz dazu besitzen diskrete Biome feste Grenzen in Form von Pfaden, die zwischen den Biomen verlaufen. Diese Art der Biome ist durch den Betrachter immer eindeutig identifizierbar, kann jedoch in dieser Form nicht in der realen Welt vorgefunden werden.

Auf Grundlage der Untersuchungen am Beispiel einiger Spiele wurde daraufhin ein Biom-Modell diskutiert, das als Grundlage für den implementierten Prototypen dienen kann. Oberste Priorität hatte bei diese Modellierung, eine Mindestkompatibi-

lität zu den geläufigsten digitalen Spielen mit offenen Biomlandschaften aufrechtzuerhalten. Solange dieses Kriterium erfüllt war, wurden die Modellierungsentscheidungen zugunsten des Realismus getroffen. Folgende Realismus-Einbußen mussten dabei hingenommen werden:

1. **Die Temperaturen werden nicht von Breitengraden beeinflusst**, da das Terrain sowohl planar anstatt kugelartig geformt ist, als auch eine dynamische Größe besitzt. Unter diesen Voraussetzungen ist das Konzept von Breitengraden nicht realistisch anwendbar.
2. **Der Wind nimmt keinen Einfluss auf den Niederschlag.** Aufgrund eines fehlenden Systems zur Berechnung von Windrichtungen können die Wasserkreisläufe der Welt nicht vollständig realistisch simuliert werden.
3. **Die Biome werden diskret realisiert**, weil das verwendete Biom-Schema nach Whittaker ebenfalls mit diskreten Biomen arbeitet. Dieser Ansatz fördert außerdem die universelle Verwendbarkeit der berechneten Biom-Karten in externen Anwendungen. Biom-Übergänge können bei Bedarf auch noch an späterer Stelle aus den zugrunde liegenden Daten berechnet werden.

Der entwickelte Prototyp auf Grundlage dieses Biom-Modells generiert sowohl eine Temperatur als auch eine Feuchtigkeitskarte, die im Anschluss zu einer Biome-Karte kombiniert werden. Die Temperatur – und Feuchtigkeitskarten werden mithilfe von Voronoi-Diagrammen erstellt und berücksichtigen sowohl den Feuchtigkeitseinfluss naher Gewässer als auch den Temperatureinfluss von großen Höhenlagen. Die zellenartige Struktur der Voronoi-Diagramme hilft dabei, eine diskrete Biom-Aufteilung des Terrains durch die Kombination der beiden Basiskarten herbeizuführen. Dabei wird das vorherrschende Biome an einem konkreten Punkt auf der Karte mithilfe einer abstrahierten Version des Klassifikationsschemas nach Whittaker bestimmt. Folgende neun Biome können in der aktuellen Version auftreten:

- Tundra
- Borealer Nadelwald
- Schneewüste
- Steppe
- Gemäßigter Wald
- Sumpf
- Subtropische Wüste
- Savanne

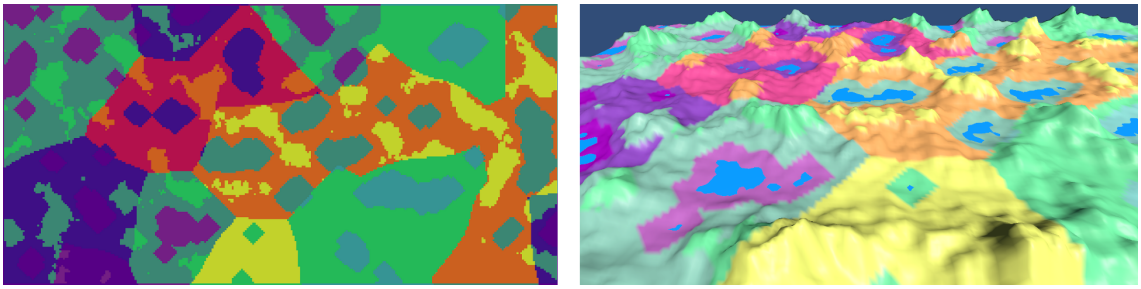


**Abbildung 2.3:** Eine Biom-Karte als Textur (links) und projiziert auf die zugehörige Höhenkarte (rechts)
Quelle: Eigene Darstellung

- Tropischer Regenwald

Um die in den Zielen spezifizierte Individualisierbarkeit der automatischen Berechnungen zu realisieren, wurden auch in diesem Projekt eine Reihe an benutzerdefinierten Parametern eingeführt. Diese ermöglichen es, auf folgende Berechnungen Einfluss zu nehmen:

- Die Verteilung der Basisregionen bei der Berechnung der Voronoi-Diagramme
- Die Krümmung der Grenzen der Basisregionen
- Die Durchnittstemperatur
- Den Durchschnittsniederschlag
- Die Höhenlage, ab der die Temperatur abfällt
- Den Einfluss von Gewässern auf die Feuchtigkeit

Dem Prototyp wurden einige Methoden zur Nachbearbeitung der generierten Biom-Karte hinzugefügt. Zum einen kann der Nutzer zwei unterschiedliche Programmroutinen auf Wunsch ausführen. Die eine Routine überschreibt zu klein geratene Biome auf der Karte, wohingegen die andere Routine eine Abwandlung der binären Open-Operation auf der Karte ausführt. Dadurch können störende Pixel-Artefakte entfernt werden. Zusätzlich wurde der Anwendung auch ein Zeichen-Tool hinzugefügt, womit der Nutzer die unterschiedlichen Karten auch manuell nachbearbeiten kann.

Abschließend ist erkennbar, dass die Ziele dieses Projektes zufriedenstellend erreicht werden konnten. Mithilfe des implementierten Prototyps ist es möglich, automatisch Biom-Karten auf Grundlage von dreidimensionalen Höhenkarten zu erzeugen (siehe Abbildung 2.3). Dabei ist der Berechnungsprozess durch den Nutzer individualisierbar und das Ergebnis kann nachbearbeitet werden.

Anzumerken ist, dass auch in diesem Projekt das Fehlen eines intuitiven User-Interface die Nutzerfreundlichkeit der Anwendung mindert. Dieser Umstand kann sich, bei unerfahrenen Nutzern, negativ auf die Qualität der erstellten Biom-Karten auswirken. Eine Einarbeitung in die Anwendung ist in solchen Fällen nötig.

# 3 Systemüberblick

-systemdiagramme -größenmaßstab -

## 3.1 Das Pipeline-Modell

## 3.2 Texturen als Berechnungsgrundlage

# 4 Texturierungssystem

Dieses Kapitel widmet sich der Entwicklung eines Texturierungssystems, das es ermöglicht, dreidimensionale Landschaften realistisch und detailreich darzustellen. Zu diesem Zeck werden unterschiedliche Texturen eingelesen, verarbeitet und auf das erzeugt Terrain (siehe Kapitel 2) gemappt. Im Folgenden wird ein umfassender Überblick über die einzelnen Komponenten des Texturierungssystems gegeben sowie deren Funktionsweisen und Zusammenspiel detailliert beleuchtet. Dabei werden die technischen Herausforderungen und Lösungsansätze hervorgehoben, um die festgelegten Ziele zu erreichen. Letztendlich soll das Texturierungssystem einen maßgeblichen Beitrag zur Realisierung einer immersiven und flexiblen Landschaftssimulation leisten.

## 4.1 Theoretische Grundlagen und Verwandte Arbeiten

In diesem Abschnitt werden die Grundlagen zur Arbeit an dem Texturierungssystem vermittelt. Im ersten Abschnitt erfolgt eine Einführung in Grafik-Pipelines, gefolgt von einer Einführung in Shader im zweiten Abschnitt. Abschließend werden die Techniken des Texture-Mappings und des darauf aufbauenden Texture Splattings näher betrachtet.

### 4.1.1 Die Grafik-Pipeline

Die Grafik-Pipeline, auch als Render-Pipeline bekannt, bildet das Herzstück der Echtzeitvisualisierung. Sie ist dafür verantwortlich, aus einer Vielzahl von 3D-Elementen, Lichtquellen, Texturen und einem Kamera-Objekt ein 2D-Bild zu generieren, das die modellierte Szene visuell darstellt. Die Position und Form der Objekte im finalen Bild werden durch ihre Geometrie, die Charakteristiken ihrer Umgebung und die Platzierung der Kamera in dieser Umgebung bestimmt. Die Oberflächeneigenschaften der Objekte werden hingegen durch die verwendeten Materialien, Lichtquellen, Texturen und Shader-Modelle beeinflusst.

### Die Architektur der Grafik-Pipeline

Die Basisarchitektur einer Grafik-Pipeline umfasst drei Hauptphasen: die Anwendungsphase, die Geometriephase und die Rasterisierungsphase (siehe Abbildung 4.1). Jede dieser Phasen kann wiederum in Unterphasen aufgeteilt werden, welche teilweise als Pipeline, teilweise jedoch auch parallelisiert ausgeführt werden können [50] [20].

In der Anwendungsphase werden die genauen Spezifikationen der zu rendernden Grundelemente, wie Eckpunkte, Kanten und Dreiecke, festgelegt. Diese Informationen werden dann an die Geometriephase weitergeleitet (siehe Abbildung 4.1). Typische Aufgaben in der Anwendungsphase umfassen Kollisionsberechnungen, die Verarbeitung von Nutzereingaben, Animationsberechnungen und mehr. Sie wird vollständig auf der CPU ausgeführt, wodurch Entwickler die volle Kontrolle über die Abläufe in dieser Phase haben [50] [20].

Die Geometriephase ist für die meisten Berechnungen im Zusammenhang mit Polygonen und Eckpunkten verantwortlich. Hierbei wird nicht nur die Geometrie der Objekte berücksichtigt, sondern auch die virtuelle Kamera der Szene definiert. Diese legt die Position und Blickrichtung fest, aus der die Szene gerendert wird. Die Geometriephase beginnt mit der Transformation der Szene, sodass sich die Kamera im Ursprung des Koordinatensystems befindet und in Richtung der z-Achse zeigt. Dies vereinfacht spätere Berechnungen [50] [20].

Die Geometriephase setzt sich fort mit dem programmierbaren Vertex-Shader (siehe Abbildung 4.1), bei dem die in lokalen Koordinaten angegebenen Eckpunkte in das globale Koordinatensystem der Szene umgerechnet werden. An diesen Eckpunktkoordinaten können an dieser Stelle bereits Beleuchtungsberechnungen durchgeführt werden.

Darauf folgt der Projektionsschritt, bei dem sämtliche Geometriekoordinaten aus dem Sichtvolumen in den Einheitswürfel transformiert werden. Das Sichtvolumen beschreibt den durch die Kamera einsehbaren Bereich. Dabei können orthogonale oder perspektivische Projektionen angewendet werden. Die orthogonale Projektion eignet sich gut für technische Darstellungen, weil die Parallelität im Objektraum bei dieser Art der Projektion auch im Bildraum erhalten bleibt. Für realistische Ansichten aus Sicht eines Betrachters ist die orthogonale Projektion weitgehend unbrauchbar, da die dargestellten Objekte verzerrt wahrgenommen werden [50] [20].

Die perspektivische Projektion imitiert hingegen eine realistische perspektivische Sicht, wodurch weiter entfernte Objekte kleiner erscheinen und sich Parallelen im Modell bei einer wachsenden Entfernung zum Betrachter im Bildraum annähern. Diese Projektionsart ist geeignet für die Darstellung von beispielsweise Landschaften [50] [20].

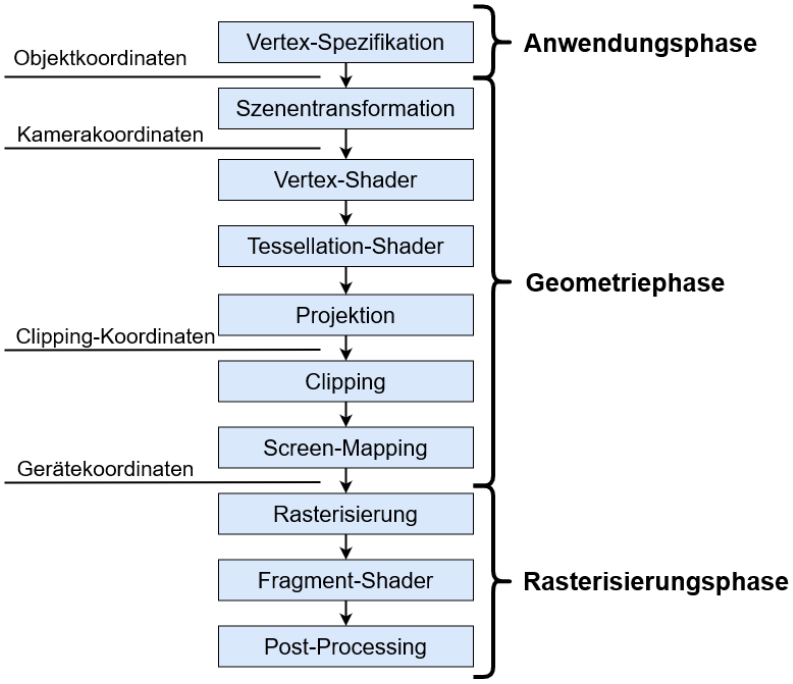



**Abbildung 4.1:** Aufbau der Grafik-Pipeline
Quelle: Eigene Darstellung nach [38]

Im Anschluss an die Projektion erfolgt das sogenannte Clipping, bei dem die Geometrie, die außerhalb des Sichtvolumens liegt, entfernt wird. Dies erfolgt durch das Ersetzen der Eckpunkte, die aus dem Volumen herausragen, durch Eckpunkte, die sich genau auf dem Randbereich des Volumens befinden. Hierdurch wird sichergestellt, dass nur die sichtbare Geometrie weiterverarbeitet wird [50] [20].

Den Abschluss der Geometriephase bildet das Screen-Mapping, bei dem die Koordinaten der Geometrie in Koordinaten auf dem Ausgabegerät transformiert werden. Zu diesem Zweck wird in diesem Schritt eine Verschiebung der Geometriekoordinaten gefolgt von einer Skalierungsoperation durchgeführt [50] [20].

Die Rasterisierungsphase bildet den Abschluss der Grafik-Pipeline (siehe Abbildung 4.1). Hier werden die geometrischen, kontinuierlichen Grundelemente in diskrete Fragmente umgewandelt, die als Pixel auf dem Bildschirm dargestellt werden. Diese Fragmente durchlaufen den programmierbaren Fragment-Shader (siehe Abbildung 4.1), der Veränderungen an den Fragmenten vornimmt, um bestimmte Effekte wie zum Beispiel Schattierungen, Spiegelungen, Blooming und Lens Flares zu erzeugen. Danach wird die eingeschränkte Sichtbarkeit von Pixeln, bedingt durch die Überlappung von Polygonen, wird mithilfe eines Z-Buffer-Algorithmus bestimmt. Abschließend erfolgt die Berechnung der sichtbaren Pixel-Farben basierend auf Beleuchtungsdaten, Texturen und anderen Materialeigenschaften [50] [20].

### Grafik-Pipelines in Unity

In der Unity Game Engine gab es in frühen Versionen nur eine einzige Render-Pipeline, die heute als „Built-In Render Pipeline“ bezeichnet wird. Diese Pipeline konnte für eine Vielzahl von Projekten genutzt werden, von einfachen 2D-Grafiken auf Mobilgeräten bis hin zu anspruchsvoller 3D-Grafik für den PC und Konsolen. Allerdings stellte sich dieser Ansatz als nicht ideal heraus. Die Verwendung einer allumfassenden Grafik-Pipeline bedeutete in der Praxis für den Entwickler viel Overhead und Zusatzarbeit, um die Pipeline auf die individuellen Anforderungen eines Projekts anzupassen [2].

Um diesem Problem entgegenzuwirken, führte Unity im Jahr 2018 die „Scriptable Render Pipeline“ (SRP) ein. Diese ermöglicht es Entwicklern, maßgeschneiderte Render-Pipelines für spezifische Projekte zu erstellen. Auf der Grundlage der SRP wurden die „Universal Render Pipeline“ (URP) und die „High Definition Render Pipeline“ (HDRP) von Unity entwickelt und bereitgestellt. Diese vorgefertigten Pipelines bieten eine erweiterbare Basis für Entwickler, die keine komplette Pipeline von Grund auf erstellen möchten [2].

Die URP eignet sich hierbei für Anwendungen mit weniger anspruchsvollen grafischen Anforderungen und ist auf vielen verschiedenen Plattformen, einschließlich mobiler Geräte, nutzbar [49]. Die HDRP hingegen unterstützt viele Features zur Umsetzung von anspruchsvollen grafische Anwendungen und ist deshalb größtenteils auf leistungsstarken Endgeräten wie PCs und Konsolen einsetzbar [47].

## 4.1.2 Shader und Shader-Graph

Die Render Pipeline gibt Entwicklern einige programmierbare Hardware-Abschnitte vor, die Shader-Stages genannt werden. Die folgenden Abschnitte der Render Pipeline fallen in diese Kategorie (siehe Abbildung 4.1):

- Vertex Shader
- Tessellation Shader
- Geometry Shader
- Fragment Shader

Ein Software-Shader ist ein benutzerdefiniertes Programm, das in einer high-level Programmiersprache geschrieben wird und diese Hardware-Schnittstellen innerhalb der Render-Pipeline anspricht [39]. Verwendet werden für diese Programme speziell dafür vorgesehene Sprachen wie „C for Grafics“ (Cg) [5], die OpenGL Shading Language (GLSL) [37] oder auch die High Level Shading Language (HLSL) [35].

Software-Shader ermöglichen es dem Entwickler Oberflächenfarben, Verschiebungen, Lichtfarben, Lichtrichtungen und Volumeneffekte zu beeinflussen. Weiterhin ist es möglich mithilfe von Shadern Vertex-Transformationen, Pixelfarben und sogar Geometrieänderungen zu spezifizieren [21]. Vertex-Shader bearbeiten dabei die vorhandene Objektgeometrie (Position, Texturkoordinaten, Normalen, Tangenten, Binormalen und Vertex-Färbungen), während Tessellation-Shader zur Zerlegung von Objektoberflächen in kleinere Einheiten eingesetzt werden. Geometrie-Shader hingegen erweitern oder modifizieren die bestehende Geometrie, indem sie neue Vertex-Gruppen berechnen wohingegen Fragment-Shader Farbwerte auf den auszugebenden Bildpunkten beeinflussen [36].

#### Shader-Graph: Ein GUI zur Shader-Erstellung

Viele der aktuellen Game Engines bieten ihren Nutzern Authoring-Tools zur Erstellung von individuellen Shadern an, ohne dass dafür Programmcode vom Nutzer geschrieben werden muss. In der Unity Game Engine heißt dieses Tool Shader-Graph [48] und wird über eine Benutzerschnittstelle gesteuert, die auf Graphen basiert. Nutzer können neue Shader erstellen, indem sie Funktionsknoten hinzufügen und miteinander verknüpfen, um gewünschte Shader-Funktionalitäten zu erreichen. Die auf diese Weise zusammengefügten Graphen werden anschließend von Unity in HLSL zur weiteren Verarbeitung übersetzt. Shader, die mithilfe dieses Tools erstellt werden, sind dynamisch, was bedeutet, dass Nutzer während der Entwicklung mit jeder Anpassung am Shader die resultierenden Änderungen am Ergebnisbild in Echtzeit beobachten können [48]. Ein Beispiel für einen in Unity umgesetzten Shader-Graph ist in Abbildung 4.2 dargestellt.

### 4.1.3 Texture-Mapping und Texture Splatting

Texture-Mapping ist eine Technik, um Details oder Farben auf einem Objekt-Modell festzulegen. Heckbert definierte den Prozess des Texture-Mappings bereits 1986 als das Abbilden einer Funktion auf eine dreidimensionale Oberfläche [25]. Ursprünglich wurden auf diese Weise lediglich transformierte Farbpixel auf eine Oberfläche gezeichnet. Diese Technik wird heute als Diffuse-Mapping bezeichnet und stellt eine Art des Texture-Mappings dar. Weitere Varianten des Texture-Mappings sind zum Beispiel Normal-Mapping, Alpha-Mapping, Specular-Mapping und Light-Mapping. Alle diese Techniken haben gemein, dass bei deren Anwendung keine Veränderung der geometrischen Auflösung der Zielobjekte durchgeführt wird [23] [3]. Auf diese Weise kann einer Szene eine hohe Detailgenauigkeit hinzugefügt werden, während der zusätzliche Render-Aufwand moderat bleibt [24].

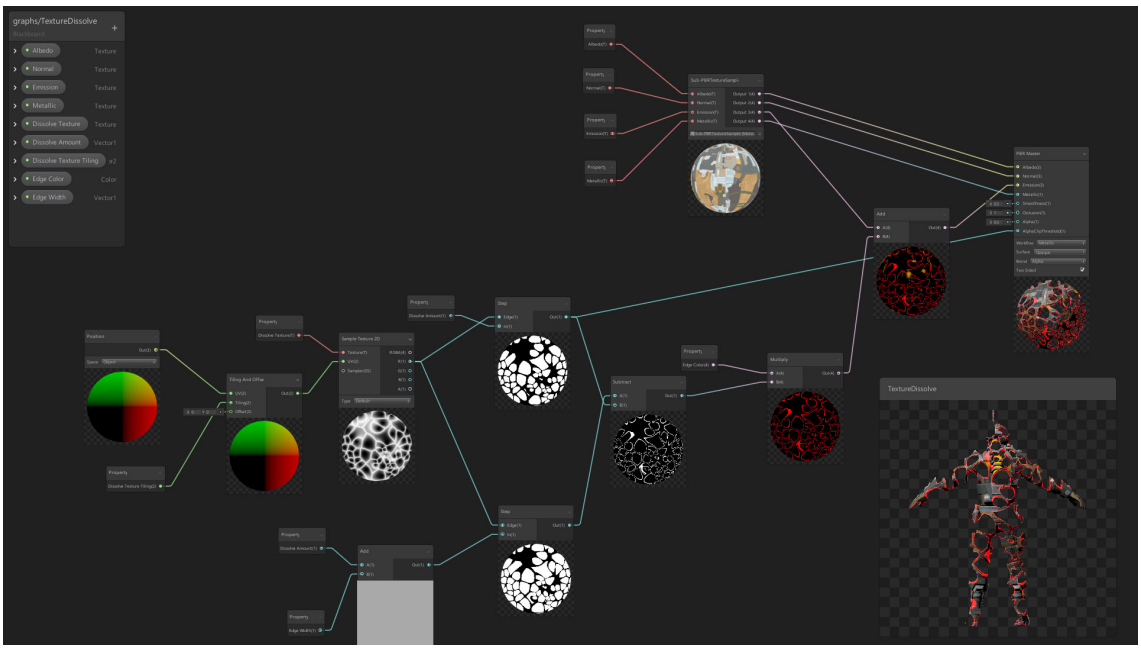



**Abbildung 4.2:** Beispiel eines Shader-Graph in Unity
Quelle: [28]

Der Begriff Texture wird von Paul S. Heckbert definiert als ein multidimensionales Bild, das auf einen multidimensionalen Raum abgebildet wird [25]. Hierbei können, für die Darstellung von Objekten im dreidimensionalen Raum, bis zu dreidimensionale Texturen verwendet werden. Eindimensionale Texturen können beispielsweise für die Simulation von übereinander lagernden Gesteinsschichten verwendet werden. Texturen mit zwei Dimensionen sind in der Lage unter anderem Wellen, Vegetation und Unebenheiten zu repräsentieren. Texturen der dritten Dimension finden bei der Wiedergabe von beispielsweise Wolken, Holz oder Marmor Anwendung. Die Definition von Heckbert schließt sowohl sich wiederholende Bildmuster wie Stoff, Gras und Ziegelsteinwände als auch nicht wiederholende Bilder wie Anzeigetafeln oder Gemälde ein.

### Ablauf des Texture-Mappings

Das typische Texture-Mapping besteht aus der Anwendung von zwei Funktionen, der Mapping-Funktion $M$ und der Sampling-Funktion $S$, auf eine Textur. Die Mapping-Funktion bildet dabei die dreidimensionalen Koordinaten eines Objekts auf zweidimensionale Texturkoordinaten, die sogenannten UV-Koordinaten, ab:

$$M(x, y, z) \mapsto (u, v)$$

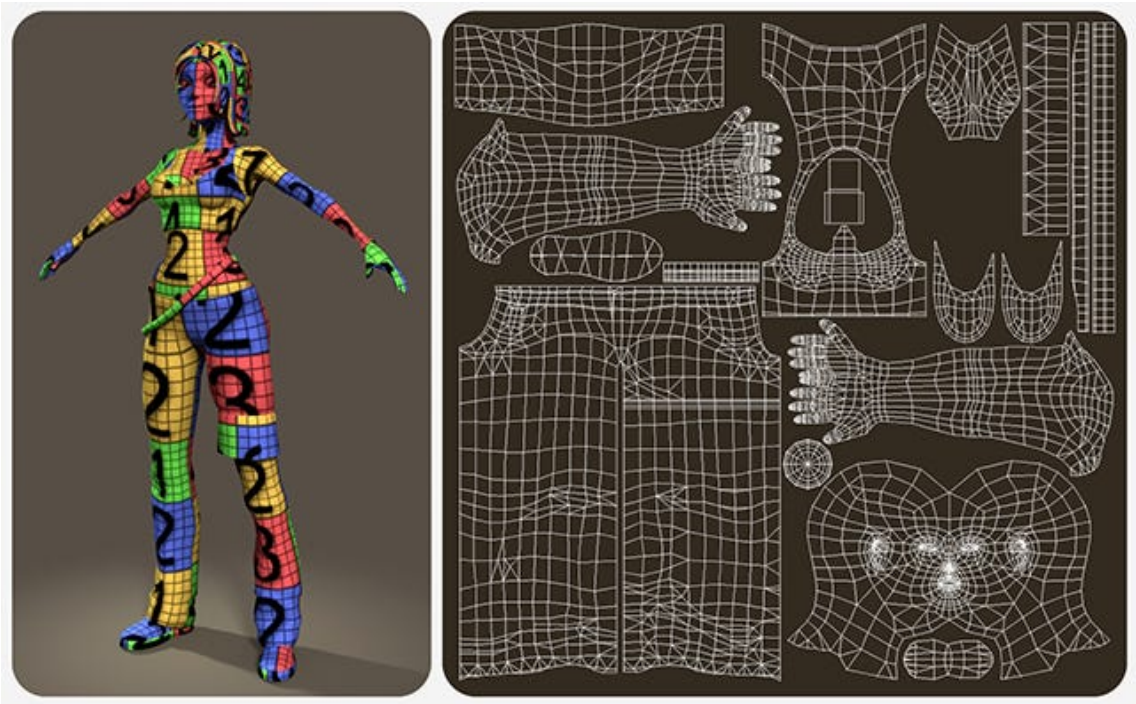


**Abbildung 4.3:** UV-Mapping dargestellt an einem Charakter-Modell
Quelle: [10]

Die UV-Koordinaten einer Textur ergeben sich typischer Weise durch die Zuweisung konkreter Koordinaten zu den Eckpunkten der Objektgeometrie (siehe Abbildung 4.3). Über diese wird interpoliert, um UV-Koordinaten für jeden beliebigen Punkt einer Oberfläche zu erhalten. Die Sampling-Funktion rechnet im Anschluss die UV-Koordinaten in Pixelpositionen auf der gewählten Textur um und bestimmt dadurch die Farbwerte, die auf das ursprüngliche Objekt gemappt werden [50] [59]:

$$S(u, v) \mapsto (r, g, b, a)$$

Alpha-Mapping, auch als Transparency-Mapping bekannt, bezeichnet eine Variante des Texture-Mappings, bei der Transparenzwerte von Texturen auf Objekte abgebildet werden. Dabei werden häufig die Alpha-Kanäle der RGBA-Farbpixel zur Aufbewahrung der Transparenzdaten verwendet, wodurch die Technik ihren Namen erhielt. Alpha-Mapping ermöglicht die Simulation von Oberflächen mit partieller Transparenz (siehe Abbildung 4.4), ohne die aufwändigen Berechnungen für die Umsetzung der gewünschten Effekte als geometrische Formen durchführen zu müssen [46].

**Abbildung 4.4:** Vergleich der linearen Interpolation zweier Texturen ohne (oben) und mit Alpha-Mapping (unten)
Quelle: [22]

### Texture Splatting zum Kombinieren von Texturen

Texture Splatting stellt eine Technik dar, um mehrere Texturen auf einer gemeinsamen Oberfläche mithilfe von Alpha-Maps abzubilden und kontinuierliche Übergänge zwischen den Texturen zu erzeugen. Um diese Technik anzuwenden, ist eine lineare Interpolation zwischen den Texturen erforderlich, die auf der Zieloberfläche aneinandergrenzen sollen. Hierbei werden die Texturen mit ihren entsprechenden Alpha-Maps multipliziert, wodurch Texturen mit partieller Transparenz entstehen. Diese werden im nächsten Schritt durch Addition miteinander zu einer gemeinsamen Textur verbunden (siehe Abbildung 4.5). Dieser Schritt wird als Blending bezeichnet. Beim Texture-Splatting kann es sinnvoll sein, die unterste Texturschicht als Basistextur vollständig undurchsichtig über die gesamte Oberfläche zu verteilen, um potenzielle Texturlücken (siehe Abbildung 4.6) in der finalen Darstellung zu vermeiden [1] [16].

Es ist üblich, zum Speichern einer Alpha-Map den Transparenz-Kanal einer RGBA-Textur zu verwenden und die übrigen drei Kanäle der Textur mit RGB-Farbwerten zu belegen. Um Ressourcen beim Splat-Mapping zu sparen und dem Entwickler eine bessere Übersicht zu gewährleisten können in der Praxis jedoch auch bis zu vier Alpha-Maps in einer gemeinsamen RGBA-Textur zusammengefasst werden. Dabei belegt jede Alpha-Map genau einen der Farb-Kanäle der Textur. Andernfalls wäre

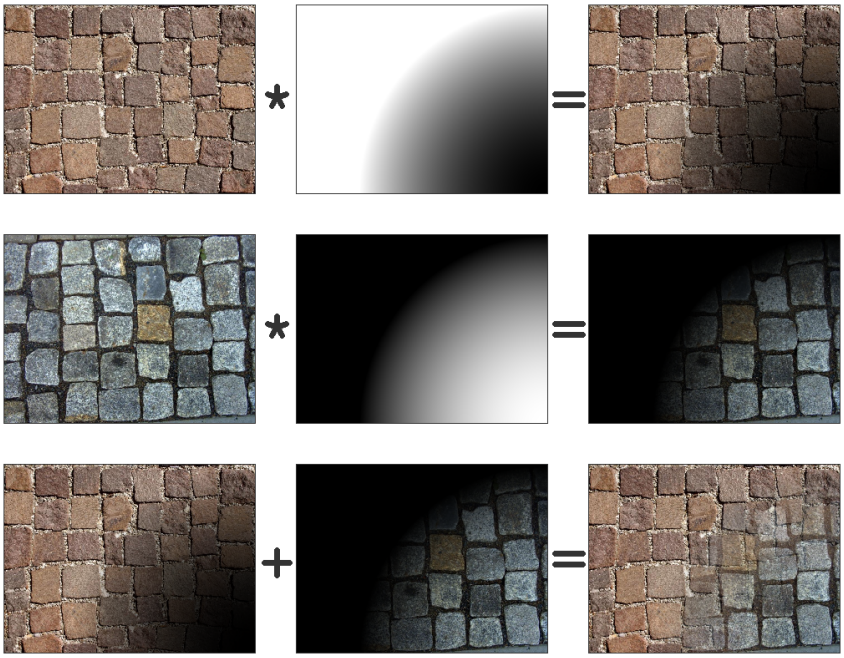


**Abbildung 4.5:** Texture-Splatting dargestellt am Beispiel
Quelle: [61]

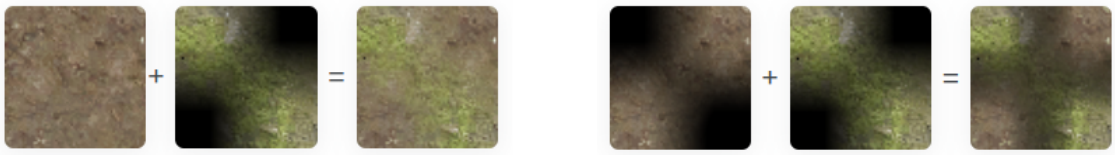


**Abbildung 4.6:** Vergleich Fehlerfreier Blend (links) und Blend mit Texturlöchern (rechts)
Quelle: [16]

für jede benötigte Alpha-Map die Erstellung einer gesonderten Textur notwendig [1] [16].

## 4.2 Konzeption des Texturierungssystems

Die effektive Erzeugung von realistischen und vielfältigen virtuellen Welten erfordert eine präzise Integration von unterschiedlichen Aspekten. Dieser Abschnitt widmet sich der umfassenden Betrachtung eines zentralen Bestandteils: dem Texturierungssystem. Im Zentrum stehen die präzise Definition der angestrebten Ziele, die Evaluierung möglicher Ansätze zur Zielerreichung und die Ausarbeitung eines Konzepts für das Texturierungssystem, das auf den vorhergehenden Überlegungen basiert. Als Eingabe erhält das System den Output des Terraingenerators und des Biomgenerators aus Kapitel 2. Es wird also auf einem dreidimensionalen landschaftsähnlichen Mesh gearbeitet, das bereits in verschiedene Biome eingeteilt ist.

### 4.2.1 Zielsstellung

Die zu erreichenden Ziele des Texturierungssystems werden im Folgenden mithilfe der in Abschnitt 1.3 definierten Anforderungen an das Projekt spezifiziert. Insgesamt sollen mithilfe des Systems alle neun unterschiedlichen Biomarten (siehe Abschnitt 2.2.2) auf dem Terrain texturiert werden können. Zusätzlich wird die Darstellung von Steigungsdaten und Straßen mithilfe von Texturen angestrebt. Dabei werden die folgende Kriterien berücksichtigt:

1. **Realismus:** Eine realistische Darstellung der Landschaft ist von zentraler Bedeutung für die Immersion des Nutzers. Ziel ist es, das Landschaftsmodell durch die Verwendung von Texturen visuell an ein reales Landschaftsbild anzunähern und dadurch eine für den Betrachter glaubwürdige Darstellung zu erzielen.

2. **Flexibilität:** Um dem Nutzer die volle kreative Kontrolle über die Gestaltung der Landschaft zu geben, muss das Texturierungssystem eine hohe Flexibilität bieten. Der Nutzer soll in der Lage sein, sowohl die verwendeten Texturen auszutauschen als auch individuelle Anpassungen an den Texturen und deren Übergängen vornehmen zu können.

3. **Performance:** Die Performance des Texturierungssystems ist von entscheidender Bedeutung, um eine reibungslose und unterbrechungsfreie Interaktion des Nutzers mit der virtuellen Landschaft zu gewährleisten. Die Texturberechnungen dürfen den Nutzer zu keinem Zeitpunkt in seinen Handlungen einschränken.

4. **Benutzbarkeit:** Es genügt, wenn das Texturierungssystems sein volles Potenzial bei der Verwendung durch einen erfahrenen Nutzer durchgehend entfaltet. Erfahren bedeutet hierbei, dass der Nutzer mindestens mit der Unity Game Engine umgehen kann und im besten Fall eine Einweisung in die Funktionalitäten des Systems erhalten hat. Zu keinem Zeitpunkt sollten Programmfehler, wie zum Beispiel untexturierte Bereiche im Mesh, auftreten. Der Arbeitsaufwand sollte für den Nutzer zu jedem Zeitpunkt möglichst gering gehalten werden.

### 4.2.2 Diskussion und Konzept

Die Umsetzung eines Texturierungssystems für den Weltengenerator erfordert eine fundierte Evaluierung verschiedener Ansätze, um die gesteckten Ziele effizient zu erreichen. Eingeleitet wird der Abschnitt mit der Diskussion einer geeigneten Render-Pipeline für das Projekt. Anschließend erfolgt eine Diskussion zu den Vor- und Nachteilen von prozeduralen Texturen gegenüber vorgefertigten Texturen. Abschließend werden geeignete Texture-Mapping Verfahren und Lösungsansätze für die realistische Verteilung von Texturen auf dem Terrain betrachtet.

#### Wahl der Render Pipeline

Die Auswahl der Render Pipeline ist ein entscheidender Faktor bei der Gestaltung des Texturierungssystems. Sie ermöglich die Spezialisierung der Grafik eines Projekts auf einen bestimmten Anwendungsbereich, was unterschiedliche Vor- und Nachteile mit sich bringt. Verschiedene Optionen wie die Built-in Render Pipeline, HDRP, URP oder eine benutzerdefinierte SRP stehen zur Verfügung.

- **HDRP:** Diese Pipeline bietet eine Fülle von Funktionen und eine einfache Erweiterbarkeit durch Code. Allerdings geht dies zu Lasten der Performance. Die HDRP wird gut von Unity gewartet, weist jedoch eingeschränkte Plattformkompatibilität auf, was die Flexibilität der Anwendung beeinträchtigt.
- **Built-in Render Pipeline:** Obwohl sie in Teilen der Unity-Community bereits als veraltet angesehen wird, ist sie für Anfänger und einfache Projekte ausreichend. Ein späteres Upgrade auf die URP ist nach Wahl der Built-in Render Pipeline jederzeit möglich. Außerdem bietet sie ähnliche Funktionen wie die URP und unterstützt sogar, aufgrund ihres höheren Alters, die größte Masse an Assets aus dem Unity Asset Store. Dennoch wird sie schlechter von Unity gewartet als die alternativen Grafik-Pipelines und ist nur schwierig durch benutzerdefinierte Shader in Form von Skripten erweiterbar.

- **URP:** Die URP weist eine breite Plattformkompatibilität auf und bietet einen ähnlichen Funktionsumfang wie die Built-in Render-Pipeline. Sie wird gut gewartet und ist leicht durch benutzerdefinierte Shader erweiterbar. Im Kriterium Performance kann diese Pipeline jedoch nicht mit der HDRP mithalten.
- **Benutzerdefinierte SRP:** Dieser Ansatz verspricht eine optimale Lösung, da die SRP als Basis dem Entwickler die Möglichkeit gibt, sämtliche Aspekte der Render Pipeline einzeln auf die gesetzten Ziele abgestimmt zu programmieren. Gleichzeitig erfordert die Konfiguration einer benutzerdefinierten SRP jedoch auch einen wesentlich höheren Arbeitsaufwand als ihre Alternativen. Hierbei müssen viele benötigte Features neu implementiert werden, anstatt auf vorgefertigte Lösungen zurückgreifen zu können. Eine effiziente Implementierung einer SRP für den Prototyp könnte eine interessante Forschungsmöglichkeit einer zukünftigen Arbeit sein, sie bildet jedoch nicht den Mittelpunkt dieses Projekts.

Unter Betrachtung der genannten Vor- und Nachteile im Kontext der Projektziele stellt die URP der Unity Game Engine die vielversprechendste Option für die Umsetzung des Prototyps dar. Diese bietet beinahe den gleichen Funktionsumfang wie die Built-in Render Pipeline und wird gleichzeitig besser vom Entwickler gewartet. Zusammen mit der Möglichkeit, benutzerdefinierte Shader zu programmieren, ist die URP sowohl aktuell als auch in zukünftigen Projekten eine zuverlässige und flexible Grafik Pipeline.

Die Wahl der HDRP wurde verworfen, um eine breitere Masse an Plattformen unterstützen und damit den Prototypen in zukünftigen Anwendungen flexibler einsetzen zu können. Diese Entscheidung wirkt sich auf dem aktuellen Stand nicht auf die Qualität der Grafik aus. Zukünftige Obergrenzen bei der Grafikqualität mit Auswirkungen auf den Realismus sind bei einer fortschreitenden Entwicklung des Projekts über diese Arbeit hinaus jedoch nicht auszuschließen.

### Diskussion von prozeduralen und vorgefertigten Texturen

Die Textursynthese ist ein Prozess zur automatisierten Erstellung von Texturen. Maßgeblich für prozedurale Texturen ist deren Eigenschaft, nicht aus dem Speicher geladen, sondern zur Laufzeit berechnet zu werden. Wie bei der Generierung anderer prozeduraler Inhalte fließt auch hierbei der Aspekt der Pseudo-Zufälligkeit mit in die Berechnungen ein [54].

Eine Alternative zu den prozeduralen Texturen bieten vorgefertigte Texturen. Diese werden, anders als bei der Textursynthese, häufig vor der Ausführung des Programms erstellt und zur Laufzeit zur weiteren Verwendung aus dem Systemspeicher

geladen werden. Üblicher Weise werden entweder Zeichnungen oder Forografien als vorgefertigte Texturen verwendet.

Die prozedurale Texturgenerierung könnte neue Möglichkeiten zur individuellen Anpassung aller Objekte innerhalb einer generierten Welt bieten. Dazu gehören unter anderem das Terrain, die Vegetation und Siedlungen. Sie bietet also Potenzial für eine bessere Flexibilität der Anwendung und eine realistischere Welt. Weitere Vorteile sind der niedrige Speicherbedarf, weil die Texturen nur innerhalb der Laufzeit vorliegen müssen und die Möglichkeit für eine nahezu beliebige Texturauflösung. Diese Tatsache bedeutet jedoch auch, dass die Texturen zur Laufzeit in jedem Frame neu berechnet werden müssen, was sich negativ auf die Performance auswirken kann.

Bei der Betrachtung der vorgefertigten Texturen kann zwischen folgenden Ansätzen unterschieden werden:

**Eine allumfassende Textur:** Es wäre denkbar, eine fertige Textur aus dem Speicher zu laden, die die gesamte Oberfläche des Terrains abdecken kann. Eine solche Großtextur könnte zwar alle Ansprüche an das Texturierungssystem mit einem Mal erfüllen, deren Erstellung und Aktualisierung würde jedoch zum Problem werden. Jede Aktualisierung des Terrains oder der Biome im Generator muss für eine fehlerfreie Darstellung auch eine Aktualisierung der Texturen herbeiführen. Eine vorgefertigte Großtextur müsste jedoch bei jeder Aktualisierung erneut außerhalb des Generators gezeichnet werden. Dadurch verlagert das System die Verantwortung für eine fehlerfreie Darstellung auf den Nutzer, was dem Ziel der Benutzbarkeit definitiv, aber auch dem Ziel des Realismus potenziell widerspricht.

**Kachelbare Texturen:** Diese Art der Textur bildet ein relativ kleines Muster einer regelmäßigen Materialoberfläche wie Holz oder Backsteinwänden als Quadrat ab. Maßgeblich für kachelbare Texturen ist die Eigenschaft, nahtlos ineinander überzugehen, wenn sie direkt nebeneinander platziert werden. Auf diese Weise kann ein Zielobjekt vollständig mit einer einzigen dieser Texturen „gefliest“ werden. Noch realistischer wird es, wenn eine Kombination mehrerer kachelbarer Texturen angewendet wird. Da diese Art der Texturierung mithilfe der Grafik-Pipeline auch automatisch erfolgen kann, bieten kachelbare Texturen die Möglichkeit, die Texturierung einer Welt dynamisch an geänderte Umstände anzupassen. Ein Austausch der kachelbaren Texturen kann durch den Nutzer jederzeit durchgeführt werden, was eine gewisse Flexibilität gewährleistet. Das Potenzial für eine realistische Welt ist mit diesem Ansatz gewährleistet. Die Qualität der Ergebnisse hängt jedoch letztendlich von der Verteilung der Texturen auf dem Mesh ab.

Ein Problem, das gelöst werden muss, um den Projektstandard für das Ziel Realismus zu erhalten, sind die stark repetitiven Muster, die sich auf den texturierten Oberflächen ergeben. Diese können für das menschliche Auge extrem auffallend und unnatürlich wirken (siehe Abbildung 4.7). Wird dieses Problem jedoch gelöst, dann

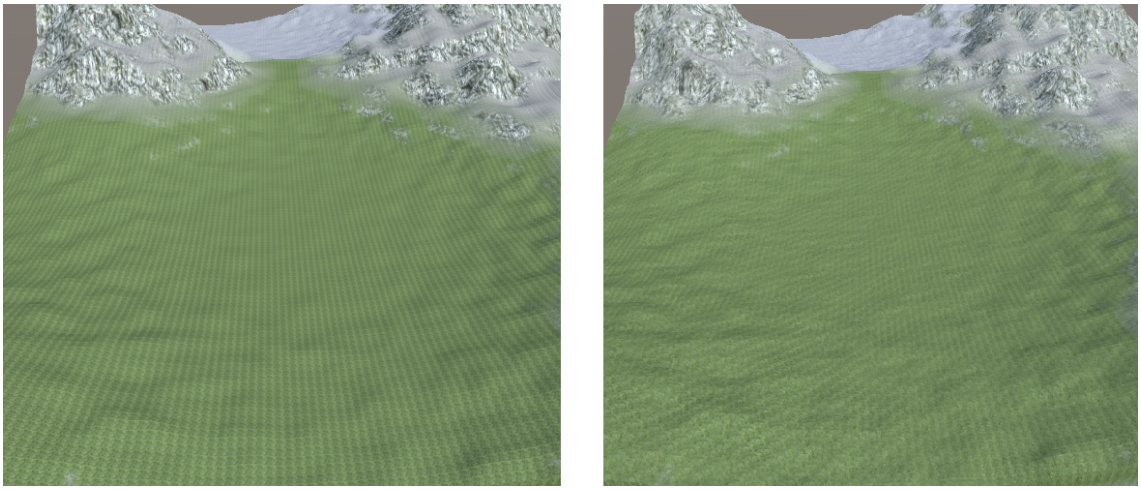


**Abbildung 4.7:** Vergleich von Repetitiven Tiles (links) und Transformierten Tiles (rechts)
Quelle: Eigene Darstellung

bieten kachelbare Texturen einen vielversprechenden Ansatz, um die Welten des dynamischen Prototyps zu texturieren.

### Lösungsansätze zur Vermeidung von Tile-Repetition

Das Problem der Tile-Repetition unter Verwendung von Tiling Texturen ist besonders auf weitläufigem Terrain so auffällig, dass es nicht ignoriert werden kann. Die Einbußen in der realistischen Darstellung der Welt wären zu groß. Es sind bereits Techniken zur Vermeidung von Tile-Repetition bekannt, die mithilfe von Grafikanwendungen wie Blender, Photoshop, aber auch mit Shadern in Game Engines umgesetzt werden können (siehe Abbildung 4.7).

Für dieses Projekt ist die Verwendung externer Anwendungen zur Lösung des Problems nicht möglich, da das angestrebte Design des Weltengenerators eine zusammenhängende Pipeline zur dynamischen Generierung vorsieht. Der Einsatz von Blender oder Photoshop zur Vermeidung von Tile-Repetition würde die Integration des Grafikprogramms in die Pipeline zur Weltengenerierung erfordern, was nur unter großem Aufwand sinnvoll zu bewerkstelligen ist. Es existieren keine offiziellen Werkzeuge, um die Prozesse eines Grafikprogramms nahtlos in die Prozesse von Unity übergehen zu lassen. Die Nutzer wären nach jeder Änderung des Terrains oder der Biome gezwungen, die Texturen der Welt manuell in einem Grafikprogramm anzupassen und anschließend erneut in den Weltengenerator einzubinden. Dadurch widerspricht dieser Ansatz dem Ziel der Benutzbarkeit.

Eine Alternative bietet die Umsetzung eines benutzerdefinierten Shaders in Unity. Die HLSL bietet Funktionen, die auf ähnliche Weise wie in externen Grafikprogrammen zur Verschleierung von Tile-Repetition genutzt werden können. Der Grundgedanke ist hierbei, Tiles unter Verwendung von Noise miteinander zu verschmelzen und ihre Anordnung abzuwandeln. Dadurch sollen die auffallenden Wiederholungsmuster unterdrückt werden. Zu diesem Zweck werden Shader-Graph Nodes zur Rotation, zur Verschiebung und zur Interpolation von Texturen sowie zur Berechnung von Noise Maps eingesetzt.

Dieser Ansatz führt dazu, dass sämtliche Berechnungen innerhalb der Unity Game Engine durchgeführt werden können. Dadurch ergibt sich die Möglichkeit, die nötigen Abläufe zu automatisieren und in die Pipeline zur Welt-Generierung zu integrieren. Änderungen an der Welt können ohne Umwege als Textur für den Nutzer sichtbar gemacht werden, was sowohl die Benutzbarkeit als auch die Flexibilität der Anwendung fördert.

### Zusammenfassung der Texturauswahl

Die Idee der Entwicklung eines Verfahrens zur Textursynthese wurde verworfen. Die umfassenden Forschungsmöglichkeiten in diesem Bereich sprengen den Rahmen dieser Arbeit und der erhöhte Berechnungsaufwand widerspricht dem Ziel der Performance. Dennoch könnte dieser Ansatz eine interessante Möglichkeit zur Erweiterung des Projekts im Rahmen einer zukünftigen Arbeit darstellen.

Stattdessen verwendet das Texturierungssystem vorgefertigte Tiling Texturen, die auf eine angemessene Größe skaliert und zur Texturierung beliebig großer Terrainabschnitte verwendet werden können. Dieser Ansatz ist eine Standardlösung in vielen Anwendungen, die mit dreidimensionalen texturierten Landschaften arbeiten Abschnitt 1.2. Vorteile dieser Lösung sind eine zufriedenstellende Flexibilität und Realismus bei einer gleichzeitig hohen Benutzerfreundlichkeit.

Um die repetitiven Muster von kachelbaren Texturen vor dem Nutzer zu verbergen, wird im Folgenden ein Shader konzipiert, der die verwendeten kachelbaren Texturen mithilfe von Noise-Maps transformiert und die besagten Muster auflöst. Ein Diagramm zur Verdeutlichung der Funktionsweise des Shaders ist in Abbildung 4.8 dargestellt.

Der Input des Shaders besteht aus einer einzigen kachelbare Textur. Auf dieser werden folgende Operationen ausgeführt, wodurch sie leicht abgewandelt wird, aber weiterhin eindeutig erkennbar bleibt:

1. Die Textur wird um einen einige Prozent vergrößert. Dadurch scheint sie etwas näher am Betrachter zu sein.

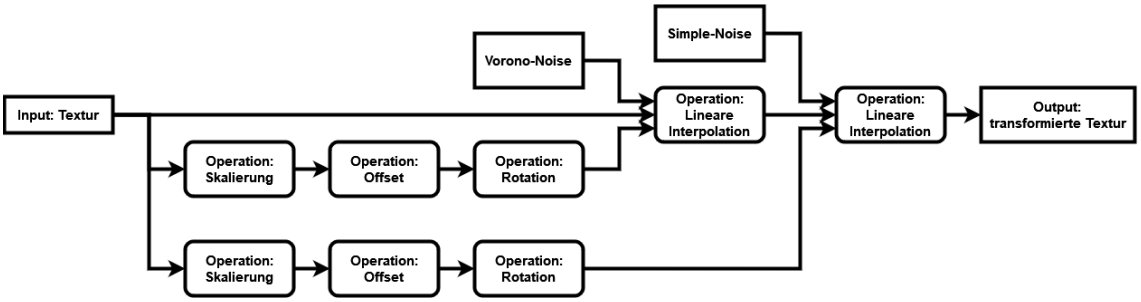



**Abbildung 4.8:** Funktionsweise des Texture Tile Splitter Shaders
Quelle: Eigene Darstellung

2. Die Textur wird durch Addition eines Offsets um einen kleinen Betrag versetzt.
3. Die Textur wird um einen Wert rotiert, der die Rotation deutlich erkennbar macht.

Im folgenden Schritt wird zwischen der daraus entstehenden transformierte Textur und der Input-Textur linear interpoliert. Als Interpolationsfaktor für diese Operation wird ein zweidimensionales Voronoi-Noise eingesetzt. Diese Art von Noise ist hierfür besonders geeignet aufgrund seiner unregelmäßigen Zellenstruktur mit kontinuierlichen verlaufenden Rändern. Dieser Aufbau ermöglicht es, die regelmäßigen Grenzen der einzelnen Tiles zu verschleiern. Im Anschluss wird zwischen der Ergebnistextur und einer weiteren Abwandlung der Input-Textur interpoliert. Diese zusätzliche Textur ist eine Input-Textur, die die gleichen drei oben beschriebenen Transformationsschritte durchläuft. Dabei werden jedoch wahrnehmbar unterschiedliche Parameterwerte bei jeder Operation eingesetzt.

Die zweite Interpolation erfolgt über eine Simple-Noise-Map. Diese Art von Noise definiert an jeder Pixelposition einen zufälligen Wert zwischen 0 und 1 und ist mit einem Fernsehrauschen vergleichbar. Auf diese Weise können die repetitiven Muster verschleiert werden, die bis jetzt noch eindeutig in den Zellen des Voronoi-Noise erkennbar waren. Anschließend werden keine weiteren linearen Interpolationen mehr auf der Textur ausgeführt. Zu viele Iterationen dieses Prozesses würden die Ausgangstextur immer weiter verschwimmen lassen, sodass irgendwann keine Details des ursprünglichen Musters mehr zu erkennen sind.

### Diskussion von Texture-Mapping-Verfahren

Die Auswahl der Texturen und Effekte, die dem Terrain durch Details ein realistischeres Aussehen verleihen, ist ein wichtiger Aspekt der Texturierung. Im Folgenden werden unterschiedliche Formen des Texture-Mappings und ihre Bedeutung für den Prototyp betrachtet.

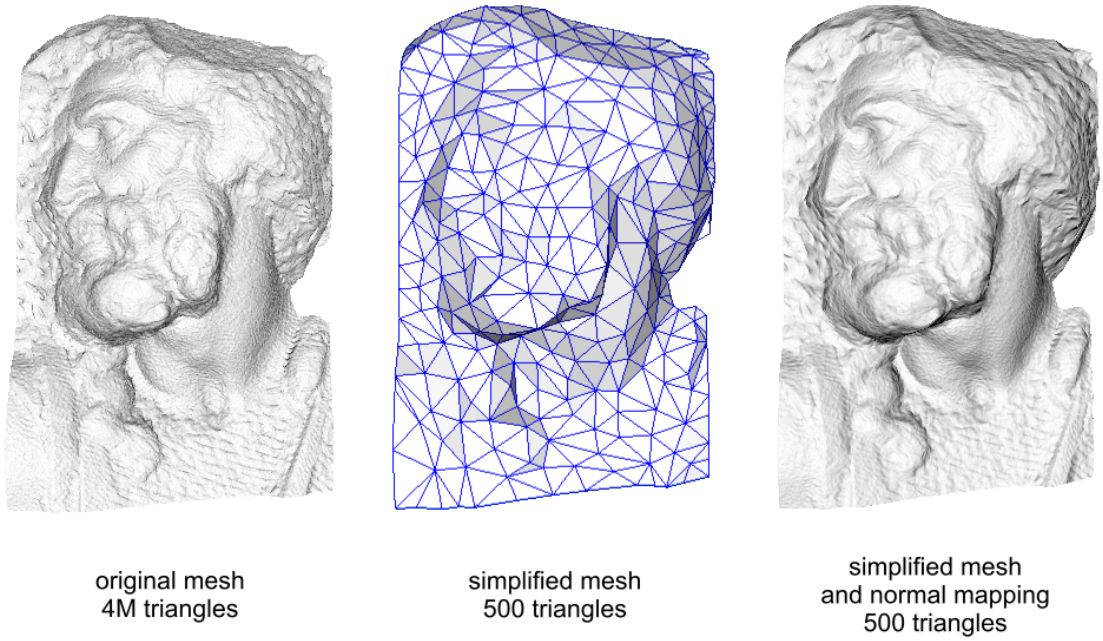



**Abbildung 4.9:** Normal-Mapping am Beispiel eines 3D-Büstenmodells
Quelle: [60]

**Diffuse-Mapping:** Das Diffuse-Mapping bildet den Standard unter den Texture-Mapping Verfahren. Dabei werden die RGB-Farbpixel einer Bilddatei auf die Oberfläche des Zielobjekts gemappt. Durch diesen Prozess wird die Farbgebung eines Objekts definiert, ohne die Geometrie des Zielobjekts zu beeinflussen. Die Anwendung von Diffuse-Mapping ist essenziell für einen performanten Realismus innerhalb der meisten aktuellen Anwendungen mit dreidimensionaler Grafik.

**Normal-Mapping:** Normal-Mapping ermöglicht die Darstellung von größerem Detailreichtum auf der Oberfläche eines Objekts durch feine Schattierungen, ohne eine komplexere Objektgeometrie herbeizuführen (siehe Abbildung 4.9). Möglich wird dies durch das Speichern der Oberflächennormalen eines Objekts innerhalb der RGB-Pixel einer Textur, der Normal-Texture. Diese Vektoren geben Auskunft über die Ausrichtung aller Normalen des Zielobjekts in einem Zustand hoher geometrischer Auflösung. Durch das Mapping der Normal-Texture auf das niedrig aufgelöste Zielobjekt, werden die gespeicherten Oberflächendetails übertragen und eine hohe Auflösung des Objekts simuliert. Auch diese Technik wird häufig angewendet, um einer Szene realistische Details hinzuzufügen, ohne die Performance maßgeblich zu beeinflussen.

**Specular-Mapping:** Um den Glanzwert einer Oberfläche abschnittweise zu definieren (siehe Abbildung 4.10), kann das sogenannte Specular-Mapping verwendet werden. Bei dieser Technik wird ein Specular-Werte für jeden Pixel einer Textur

**Abbildung 4.10:** Vergleich einer Kiste ohne (links) und mit (rechts) Specular-Mapping
Quelle: [31]

als Specular Map abgespeichert. Der Specular Wert gibt an, welcher Anteil an einfallendem Licht von einer Oberfläche glänzend reflektiert wird. Die Specular Map wird auf eine Objektoberfläche gemappt, wodurch auf diesem Objekt nicht nur eine, sondern unterschiedliche Glanzabstufungen definiert werden können.

Mithilfe des Specular-Mappings kann das Aussehen von Objekten demzufolge realistischer gestaltet werden, wenn diese in der realen Welt einen partiellen Glanz besitzen würden. Ist ein partieller Glanz nicht erwünscht, dann reicht es aus, den Specular Wert des dem Objekt zugrunde liegenden Materials anzupassen, anstatt Texture-Mapping durchzuführen. Wie bei allen Texture-Mapping Verfahren ist auch Specular-Mapping relativ performant, da sämtliche benötigten Daten zur Laufzeit vorberechnet im Speicher vorliegen.

**Light-Mapping:** Zuletzt wird das Light-Mapping betrachtet, bei dem vorberechnete Helligkeitswerte in einer Textur abgespeichert und auf die Oberflächen des Zielobjektes gemappt werden (siehe Abbildung 4.11). Statische Objekte erhalten durch diesen Vorgang saubere Schattierungen, ohne die Performance durch dynamische Lichtberechnungen zu beeinträchtigen. Sobald die involvierten Lichtquellen oder die betroffenen Objekte jedoch von Änderungen, zum Beispiel in ihrer Position oder Rotation betroffen sind, kann eine fehlerfreie Darstellung der zugehörigen Schatten nicht mehr gewährleistet werden. In diesem Fall müsste eine Neuberechnung der Light-Map, das sogenannte „baken“, erfolgen. Dieser Vorgang ist jedoch sehr zeitaufwändig und wird deshalb nicht zur Laufzeit, sondern vor dem Programmstart in der Anwendungsphase der Render Pipeline durchgeführt. Aus diesem Grund ist Light-Mapping nicht für dynamische Objekte geeignet.

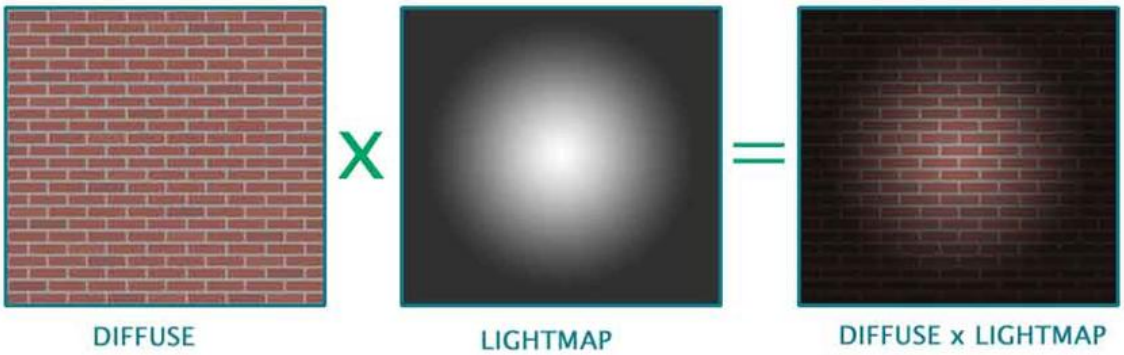



**Abbildung 4.11:** Light-Mapping am Beispiel einer Backsteinwand
Quelle: [4]

### Zusammenfassung der Texture-Mapping-Verfahren

Das Texture-Mapping ist eine performante Möglichkeit, unterschiedliche Arten von Details und Effekten auf einem Objekt darzustellen. Im Rahmen dieses Projekts wurden sowohl das Diffuse-Mapping als auch das Normal-Mapping als gewinnbringende Bestandteile eines Texturierungssystems identifiziert. Diese Verfahren versprechen eine Verbesserung des Realismus und werden unter Berücksichtigung der anderen Design-Entscheidungen mit in den Prototypen integriert.

Das Specular-Mapping und das Light-Mapping bieten hingegen in der aktuellen Version keinen besonderen Mehrwert für das Projekt. Die Definition von partiellem Glanz auf den Geländeoberflächen der generierten Welten ist nicht sinnvoll, da die ausgewählten natürlichen Oberflächen in der realen Welt keinen besonderen Glanz besitzen. Durch die Simulation von Regen und einem damit verbundenen befeuchteten Gelände würde sich dieser Umstand ändern. Einzelne feuchte Grashalme oder Steinoberflächen könnten mithilfe von Specular Maps einen eigenen Glanzwert zugewiesen bekommen. Dieses Feature sprengt jedoch den Rahmen des Projektes und bietet genügend Potenzial für eine gesonderte zukünftige Arbeit.

Die Vorberechnung von Lichtdaten des Terrains als Light-Maps kann im aktuellen Design des Weltengenerators ebenfalls nicht effektiv eingesetzt werden. Da die Geometrie des Terrains zur Laufzeit ständigen Veränderungen ausgesetzt ist, müssten die zuständigen Light-Maps in regelmäßigen Abständen neu berechnet werden, um eine realistische Beleuchtung der Welt zu gewährleisten. Dies würde regelmäßige Störungen des Arbeitsflusses und damit eine starke Beschränkung der Benutzbarkeit bedeuten. Gegen eine Anwendung von Light-Mapping auf einer finalen Ausgabe des Weltengenerators in einer externen Anwendung, beispielsweise einem Spiel, spricht jedoch nichts. Sobald sämtliche Editorprozesse abgeschlossen sind, könnte sich die Anwendung von Light-Mapping spürbarer positiv auf die Performance auswirken.

Als Alternative zu Light-Maps und als Kompromiss zwischen Realismus und Performance werden stattdessen dynamische Lichtberechnungen durchgeführt, die auf einen moderaten Bereich um den Betrachter begrenzt sind. Dadurch passen sich die Schatten von Bäumen, Häusern und Bergen dynamisch an Parameteränderungen durch den Nutzer an. Kleine Einbußen im Kriterium Realismus müssen jedoch hingenommen werden, da bei diesem Ansatz Schatten in großer Entfernung zur Wahrung einer angemessenen Performance nicht dargestellt werden können.

### Möglichkeiten zur Verteilung von Texturen

Die Art und Weise, wie die Texturen im System auf dem Terrain verteilt werden, beeinflusst maßgeblich sämtliche in Abschnitt 4.2 genannten Kriterien. Aus diesem Grund ist es wichtig, einen zufriedenstellenden Lösungsansatz für diesen Teilbereich des Texturierungssystems zu finden.

Ein naiver Ansatz für dieses Problem stellt die manuelle Anfertigung einer Großtextur durch den Nutzer dar, die auf die vollständige Oberfläche des Terrains gemappt wird. Für diesen Prozess könnte der Nutzer geeignete Texturierungs-Tools eines Grafikprogramms verwenden. Dabei könnte er sowohl auf prozedurale Texturen als auch auf Tiling Textures zurückgreifen.

Das händische Texturieren der Welt durch den Nutzer ist ein attraktiver Ansatz aufgrund der unbegrenzten Flexibilität, die dem Nutzer bei der Umsetzung seiner Ideen überlassen wird. Die Qualität der Ergebnisse schwankt hierbei jedoch stark, abhängig vom Talent und der Erfahrung des Nutzers. Ein zufriedenstellendes Maß an Realismus für künstlerisch weniger begabte Nutzer kann dabei nicht sichergestellt werden. Ein weiterer großer Kritikpunkt ist, dass die Verantwortung und die Arbeitslast für die Verteilung der Texturen auf den Nutzer ausgelagert wird. Aufgrund des Designs des Weltengenerators, mit seinen regelmäßigen Aktualisierungen des Terrains und der Biome, müssten hierdurch große Abstriche bei der Benutzbarkeit hingenommen werden. Eine dynamische Vorschau für den Nutzer wäre nicht mehr im Bereich des Möglichen.

Um die beschriebenen Probleme zu vermeiden, sollte daher ein automatisierter Ansatz zur Verteilung der Texturen auf dem Mesh gewählt werden. Vielversprechend ist dafür die Technik des Texture Splatting (siehe Abschnitt 4.1.3). Dieser Ansatz verspricht die Berechnung einer Texturkomposition aus einer Menge aus Verteilungsdaten, ohne dass der Nutzer in den Prozess eingreifen muss. Die Qualität der zusammengestellten Textur unterliegt bei diesem Ansatz, gemessen am Realismus, wesentlich geringeren Schwankungen als es beim manuellen Zeichnen durch den Nutzer der Fall ist. Das bedeutet jedoch auch, dass künstlerisch begabte Nutzer unter Umständen nicht ihr volles Potenzial bei der Weltengenerierung ausschöpfen können.

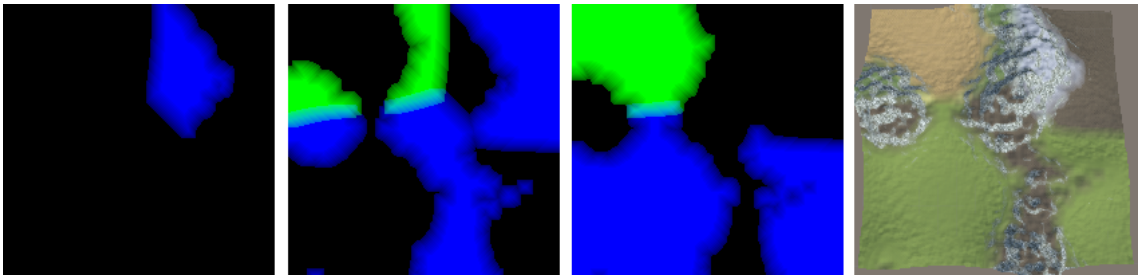


**Abbildung 4.12:** Beispiel einer möglichen Menge von Splat Maps und der resultierenden Textur
Quelle: Eigene Darstellung

Die Möglichkeiten für herausragenden Realismus und uneingeschränkte Flexibilität bei der Texturverteilung entfallen damit. Dennoch garantiert dieser Ansatz eine hohe Benutzbarkeit durch einen geringen Arbeitsaufwand und ermöglicht die weitere Umsetzung einer dynamischen Vorschau.

### Konzeption des Texture-Splatting-Moduls

Zur Umsetzung des Texture-Splatting-Ansatzes müssen zuerst die Splatmaps erstellt werden, die definieren, welche Texturen an welchen Punkten im Terrain abgebildet sein sollen. Grundlage für diese Splatmaps ist der Output des Biom-Generators. Diese Biom-Karte definiert, wie im Abschnitt 2.2 beschrieben, für jeden Knoten im Terrain Mesh eine Biomzugehörigkeit. Diese Biomzugehörigkeiten dienen als Richtwert, nach denen die Basistexturen auf das Terrain aufgetragen werden.

Geplant ist, die Sichtbarkeit der einzelnen Texturen durch Alpha-Maps zu beschränken und diese anschließend übereinander zu schichten. Insgesamt gibt es 9 unterschiedliche Biome, die durch die Splatmaps erfasst werden müssen (siehe Abschnitt 2.2). Da eine Textur bestehend aus RGBA-Farbpixel maximal 4 Alpha-Maps aufnehmen kann, ist es keine Option, alle Alpha-Maps in einer Textur unterzubringen. Stattdessen werden für die 9 nötigen Alpha-Maps insgesamt 3 Texturen berechnet, die jeweils 3 der Alpha-Maps aufnehmen. Geordnet werden die Alpha-Maps dabei nach dem Temperaturwert ihres zugehörigen Bioms: alle kalten, gemäßigten und heißen Biome werden gesondert als Alpha-Map in einer eigenen Splatmap gespeichert. Die Übergänge zwischen den Alpha-Maps werden dabei so interpoliert, dass sie miteinander von diskreten zu kontinuierlichen Übergängen verschmelzen (siehe Abbildung 4.12). Dies trägt zu einem realistischen Aussehen der Welt bei.

Die berechneten Splatmaps dienen im Anschluss als Input für einen Shader. Dieser Shader hat die Aufgabe, eine Großtextur für die vollständige aktuelle Terrainoberfläche zusammenzustellen. Zusätzlich zu den Splatmaps benötigt der Shader ein Array,

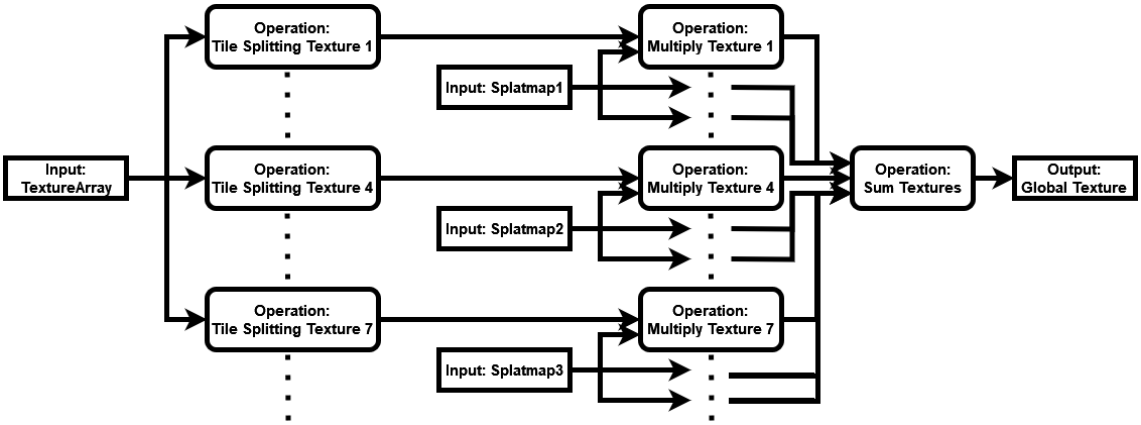



**Abbildung 4.13:** Funktionsweise des Texture Splat Shaders
Quelle: Eigene Darstellung

in dem alle Basistexturen als Texture Tiles abgespeichert sind sowie einen Float-Parameter, der die Skalierung der Geländetexturen auf dem Mesh vorgibt. Dieser Input wird durch den Nutzer vorgegeben. Jede der Basistexturen wird auf folgende Weise bearbeitet (siehe Abbildung 4.13):

1. Der oben beschriebene Shader zur Verschleierung von Tile-Repetition wird angewendet.
2. Die Textur wird mit ihrem zugehörigen Alpha-Map Kanal, welcher aus einer der Splatmaps stammt, multipliziert.
3. Die partiell transparenten Zwischenergebnisse werden zu einer gemeinsamen Textur addiert.

Der Output dieses Shaders bildet eine zusammenhängende Textur, die sich über das gesamte Terrain spannt. Das beschriebene Konzept zum Splatting der Welttexturen wird im Prototypen sowohl zur Verteilung der Diffuse Maps als auch zur Verteilung der zugehörigen Normal-Maps verwendet.

Dieser Ansatz entlastet den Nutzer, da dieser die Texturen nicht manuell auf das Terrain zeichnen muss. Zudem werden Standards für Realismus aufrechterhalten, und Entwicklern wird Flexibilität bei der Verteilung der Texturen mithilfe der Splatmaps durch die Austauschbarkeit der Texturen gewährt. Allerdings ergibt sich aus der Abhängigkeit der Splatmaps von der Biom-Karte, dass eine Veränderung der Texturverteilung nur über eine aktualisierte Biom-Karte herbeigeführt werden kann. Diese Einschränkung der Benutzbarkeit wurde bereits bei der Konzeption des Biom-Generators berücksichtigt und ist hinnehmbar, da der Nutzer eine solche Aktualisierung durch Betätigung eines einzelnen Knopfes durchführen kann. Eine vollständig dynamische Vorschau für den Nutzer muss dadurch jedoch verworfen werden.

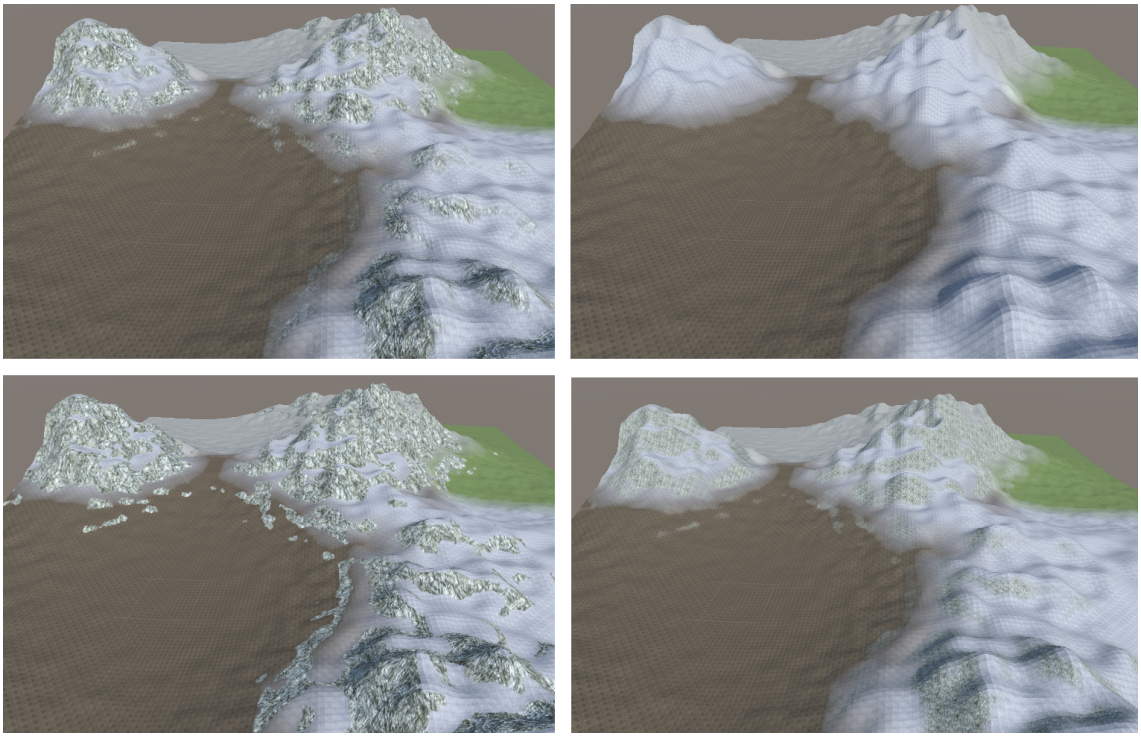


**Abbildung 4.14:** Vergleich von Steilhängen:
standardmäßig, fehlend, scharfe Übergänge, skalierte Textur
Quelle: Eigene Darstellung

Abschließend werden noch einige weitere Texturen auf die zuvor berechnete Basistextur des Terrains verteilt, um die generierte Welt realistischer zu gestalten. Den Anfang bilden dabei Felsentexturen, die auf allen Steilhängen des Terrains sichtbar sein sollen. Auch in der realen Welt sind steile Berghänge oft von kargem Fels geprägt. Dieser Zusatz verleiht der Welt ein heterogenes Aussehen und bricht mit dem Muster der Biome, wie in Abbildung 4.14 erkennbar ist. Für das Splatting der Felsentextur wird eine Alpha-Map benötigt, die sämtliche Steilhänge des Terrains abbildet. Zu diesem Zweck werden die Steigungswinkel des Terrains an jedem Knoten des Mesh bestimmt. Aus diesen Werten wird eine Steigungskarte erstellt, die schon beinahe als Alpha-Map verwendet werden kann. Aus der Steigungskarte müssen lediglich die Pixelwerte herausgefiltert werden, die eine geringe Steigung repräsentieren. Ab welchem Wert ein Berghang ausreichend steil ist, um die Alpha-Map zu beeinflussen, soll der Nutzer über einen Parameter festlegen können. Zusätzlich soll dem Nutzer über weitere Parameter die Kontrolle darüber gegeben werden, wie scharf die Grenzen der Steigungstextur in die angrenzenden Texturen übergehen und mit welchem Faktor die Größe der Felsentextur im Shader skaliert wird. Die Auswirkungen dieser Parameter werden in Abbildung 4.14 präsentiert.

Gleichzeitig sollen auch Straßentexturen auf das Gelände gemappt werden, um die in Kapitel 6 generierten Siedlungen realistischer zu gestalten. Hierzu wird mithilfe des Outputs des Siedlungsgenerators, einer Karte des Geländes mit den Koordinaten aller siedlungsbezogenen Strukturen, eine Alpha-Map erstellt. Diese Alpha-Map speichert die Koordinaten sämtlicher Straßenabschnitte.

Mithilfe der Steigungskarte und der Straßenkarte können nun die zugehörigen Felsen- und Straßentexturen auf die zuvor zusammengestellte Basistextur mithilfe von Texture Splatting gemappt werden. Auch hier gilt wieder, dass die zugehörigen Normal-Maps analog zu den Diffuse Maps verarbeitet und verteilt werden.

## 4.3 Implementierung des Texturierungssystems

Im folgenden Abschnitt wird die Umsetzung des Texturierungssystems im Detail erläutert. Die in Abschnitt 4.2 erarbeiteten Lösungsansätze konnten größtenteils mithilfe von Shadern im Unity Shader-Graph Tool realisiert werden. Einige Code-Abschnitte waren jedoch erforderlich, um Input-Texturen für die Shader vorzubereiten. Diese sind in der Sprache C# geschrieben. Die zugehörigen Klassen werden von der vordefinierten Unity-Klasse *MonoBehaviour* abgeleitet, wodurch die Skripte mit Unity-GameObject verknüpft werden können.

Der Output des Terrain-Generators besteht aus einem landschaftsähnlichen Mesh. Dieses verwendet ein Material, das den benutzerdefinierten `TerrainTexture-Shader` referenziert. Dieser Shader bildet die Grundlage der Implementierung und führt alle Berechnungen der einzelnen Sub-Systeme des Texturierungssystems zusammen. Er liest sowohl benutzerdefinierte als auch vorberechnete Texturen und Parameter ein. Hieraus werden alle erforderlichen Berechnungen durchgeführt, um aus den Input-Texturen eine zusammenhängende Großtextur zu erstellen, die sich über das gesamte Mesh erstreckt. Folgende benutzerdefinierte Parameter dienen als Input für das Texturierungssystem:

- TextureScaling_Terrain (Skalierungsfaktor der Terraint-Texturen)
- TextureScaling_Street (Skalierungsfaktor der Straßentextur)
- TextureScaling_Steepness (Skalierungsfaktor der Steigungstextur)
- NormalStrength (Intensität der Normalentexturen)
- SteepnessIntensity (Intensität der Steigungstextur)
- SteepnessBlending (Schärfe des Übergangs der Steigungstextur)
- TerrainTextures (Array mit allen kachelbaren Terrain-Texturen)

- TerrainNormalTextures (Array mit allen kachelbaren Normalen-Texturen)

Zusätzlich dazu werden vom `TerrainTextureShader` auch noch folgende vorberechnete Parameter eingelesen, die nicht direkt durch den Nutzer beeinflusst werden können:

- Splatmap01 (Alpha-Maps für Biome mit niedriger Temperatur)
- Splatmap02 (Alpha-Maps für Biome mit mittlerer Temperatur)
- Splatmap03 (Alpha-Maps für Biome mit hoher Temperatur)
- Steepness-Map (Steigungswerte des Terrains)
- Street-Map (Binäre Karte zur identifizierung aller Straßen)

Im folgenden wird der Begriff „Steigungstextur“ und „Straßentextur“ zur Bezeichnung der diffusen Texturen verwendet, die der Nutzer in der Ergebniswelt erkennen kann. Die Begriffe „Steepness-Map“ und „Street-Map“ bezeichnen hingegen die entsprechenden Alpha-Maps, die zur Verteilung dieser diffusen Texturen auf dem Terrain verwendet werden.

### 4.3.1 Vorberechnung des Shader-Inputs

Die Splatmaps basieren auf den Biomen, die durch den Biomgenerator im Voraus angelegt werden. Sie werden jedoch nicht vom Biomgenerator berechnet. Diese Aufgabe fällt dem Texturierungssystem zu und wird mithilfe des folgenden Algorithmus als Skript realisiert:

1. Initialisiere drei Splatmaps mit Farbkanälen, die vollständig auf 0 (schwarz) gesetzt sind.
2. Für jede der neun unterschiedlichen Biom-Arten wird über alle Pixel der aktuellen Biom-Textur iteriert.
3. Falls das gesuchte Biom gefunden wird, wird der RGB-Farbkanal des Pixels der dem Biom zugeordneten Splatmap auf 1 gesetzt.

Um die in Abschnitt Abschnitt 4.2.2 beschriebenen kontinuierlichen Übergänge zu erzeugen, durchläuft jede Splatmap einen weiteren Algorithmus. Das Auftreten von Texturlöchern (siehe Abbildung 4.6) kann hierbei durch die Wahl eines großzügigen Blending-Bereichs in den Übergängen vermieden werden. Die Berechnungen verlaufen folgendermaßen:

1. Es wird über jeden Farbkanal der Splatmap und jede Pixelreihe um den betrachtete Farbbereich innerhalb der Splatmap iteriert.

```
calculateSteepness(x,y) {
    slopeX = height(x-1,y) - height(x+1,y);
    slopeY = height(x,y-1) - height(x,y+1);

    normalVector = (slopeX, 1, slopeY);
    normalVector = normalize(normalVector);

    dotProduct = calculateDotProduct(normalVector, (0,1,0));
    radianSteepness = ArkusKosinus(dotProduct);

    transformedSteepness = InverseLerp(0, 1.6, radianSteepness);
    return transformedSteepness;
}
```

**Abbildung 4.15:**
Pseudocode zur Berechnung der Steigung in einem Punkt im Mesh
Quelle: Eigene Darstellung

2. Die betrachteten Pixel werden über den aktuellen Farbkanal eingefärbt. Der Farbwert wird mit jeder durchlaufenen Pixelreihe nach außen hin linear abgeschwächt.

Die Steepness-Map, die vom `TerrainTextureShader` einzulesen ist, muss ebenfalls mithilfe eines Skripts vorberechnet und gespeichert werden. Der zuständige Algorithmus iteriert über alle Knoten des Mesh und verarbeitet die Koordinaten wie folgt (siehe Abbildung 4.15):

1. Die Höhendifferenzen der Nachbarknoten in der Länge und Breite werden berechnet.
2. Diese werden verwendet, um den Normalenvektor des Mesh im betrachteten Punkt aufzustellen. Dieser wird anschließend normiert.
3. Über das Skalarprodukt und die Arkuskosinus-Berechnung wird der Winkel zwischen der Normalen und dem Vektor (0,1,0) bestimmt. Das Ergebnis ist ein Wert, der die Steigung des Mesh im betrachteten Punkt misst.
4. Die Obergrenze für den Winkel eines Steilhangs beträgt 90 Grad. Als Bogenmaß wären das 1.57. Um die zukünftige Arbeit mit den Steigungswerten zu vereinfachen, wird der Wertebereich des Outputs mit der InverseLerp-Funktion vom aufgerundeten Intervall $[0, 1.6]$ auf das Intervall $[0, 1]$ normiert.

Die Ergebniswerte der Steigungsberechnungen zwischen 0 und 1 können im Anschluss direkt in den Farbkanal einer Textur eingetragen und abgespeichert werden. Eine auf diese Weise berechnete Steigungstextur wird in Abbildung 4.16 präsentiert.

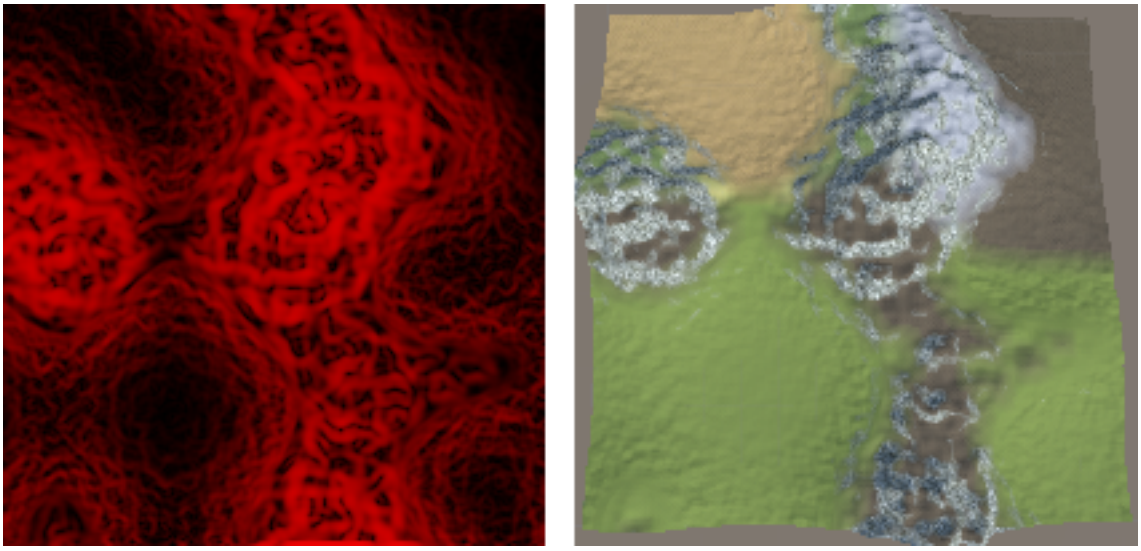


**Abbildung 4.16:** Vergleich Steepness-Map mit dem zugehörigen Terrain
Quelle: Eigene Darstellung

Die Street-Map, die als Input für den `TerrainTextureShader` dient, enthält Informationen darüber, wo im Mesh Straßen eingezeichnet werden müssen. Die Berechnung der Street-Map erfolgt als Teil des Siedlungsgenerators und wird in Kapitel 6 näher beschrieben.

### 4.3.2 Umsetzung der Shader

Der `TerrainTextureShader` durchläuft iterativ mehrere Berechnungsschritte, die alle nach dem selben Schema ablaufen:

1. Einlesen der bisher berechneten Basistextur.
2. Einlesen der aufzutragenden Textur und ihrer Alpha-Map.
3. Lineare Interpolation zwischen der aufzutragenden Textur und der Basistextur über die Alpha-Map.

Das Ergebnis eines solchen Berechnungsschritts ist eine erweiterte Basistextur mit einer neuen, teilweise transparenten obersten Texturschicht.

Der `TerrainTextureShader` ruft mehrere benutzerdefinierte Shader auf, die in der folgenden Reihenfolge durchlaufen werden (siehe Abbildung 4.17):

1. Verteilung der Biom-Texturen mithilfe des BiomeTextureSplatter
2. Verteilung der Steigungstextur mithilfe des SteepnessTextureSplatter

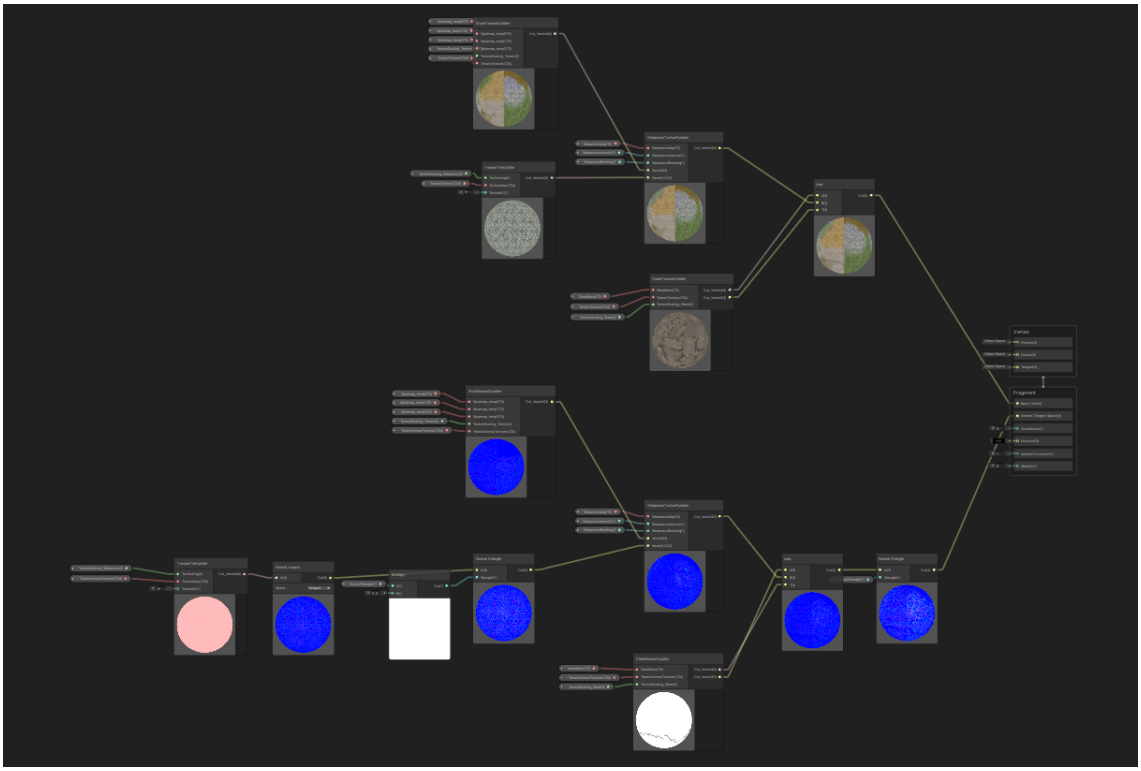


**Abbildung 4.17:** `TerrainTextureShader` umgesetzt in Shader-Graph
Quelle: Eigene Darstellung

3. Verteilung der Straßentextur mithilfe des StreetTextureSplatter

Dieser Berechnungsablauf wird im `TerrainTextureShader` sowohl für die Farbtexturen als auch für die Normalentexturen angewendet (siehe Abbildung 4.17), um beide Texturarten gleichwertig zu verarbeiten. In die Pipeline zur Verarbeitung der Normalentexturen wird zudem ein NormalStrength-Knoten eingefügt, der die berechnete Normalen-Großtextur sowie den NormalStrength-Parameter nutzt. Auf diese Weise kann der Nutzer die Intensität der Normalen in der Welt anpassen und so den visuellen Effekt steuern.

Die einzelnen Shader, die durch den `TerrainTextureShader` aufgerufen werden, werden im Folgenden näher erläutert:

### TextureTileSplitter

Der TextureTileSplitter repräsentiert die Umsetzung der in Abschnitt 4.2.2 konzipierten Lösung zur Vermeidung von Tile-Repetition und wird in Abbildung 4.18 dargestellt. Dieser Shader wird auf sämtliche kachelbaren Texturen direkt nach dem

Einlesen aus dem Speicher angewendet. Den Input des Shaders bilden der jeweilige TextureScaling-Vektor sowie die zu transformierende kachelbare Textur. Folgende Schritte werden im Shader durchlaufen:

1. **Skalierung der Textur:** Die Texturskalierung wird, wie in Abschnitt 4.2.2 beschrieben, durch die Multiplikation der einzelnen Komponenten des TextureScaling-Vektors mit vorbestimmten Konstanten realisiert. Dadurch werden die UV-Koordinaten der Textur beeinflusst und der gewünschte Zoom-Effekt erzielt.
2. **Verschiebung der Textur:** Mithilfe des TilingAndOffset-Knotens in Shader-Graph werden die skalierten UV-Koordinaten um gewisse konstante Offset-Werte verschoben. Dadurch entsteht eine Verschiebung in der Texturplatzierung.
3. **Rotation der Textur:** Die verschobenen und skalierten UV-Koordinaten werden unter Anwendung des Rotate-Knotens um einen vordefinierten Winkel und Mittelpunkt rotiert.
4. **Voronoi-Noise:** Für die Berechnung des benötigten Voronoi-Noise existiert in Shader-Graph ein für diese Aufgabe passender Voronoi-Knoten.
5. **Simple-Noise:** Das benötigte Simple-Noise kann ebenfalls mit einem Shader-Graph Knoten generiert werden. Dabei wird ein relativ großer Skalierungsfaktor von 300 angesetzt, um ein besonders feines Rauschen zu erhalten. Dadurch kann der gewünschte Effekt der Musterverschleierung verstärkt werden.
6. **Zusammenführen der Texturanteile:** Die aus den vorherigen Schritten resultierenden Texturanteile (Skalierung, Verschiebung, Rauschen) werden durch Lerp-Knoten, wie in Abschnitt 4.2.2 beschrieben, mithilfe der Noise-Maps zu einer einzelnen Textur zusammengeführt. Diese dient als Output des TextureTileSplitter.

### BiomeTextureSplatter

Der BiomeTextureSplatter (siehe Abbildung 4.19) spielt eine entscheidende Rolle in der Generierung der Welttextur. Dieser Shader nutzt den Input der drei zuvor berechneten Splatmaps, den TextureScaling_Terrain-Parameter und das Terrain-Textures-Array, um sämtliche Biome, die vom Biomgenerator erstellt wurden (siehe Abschnitt 2.2), mit Texturinformationen zu versehen. Der Ablauf des BiomeTextureSplatter-Shaders sieht für jedes der 9 Biome wie folgt aus:

1. **Textur einlesen:** Die passende Textur für das aktuelle Biom wird aus dem TerrainTextures-Array geladen.

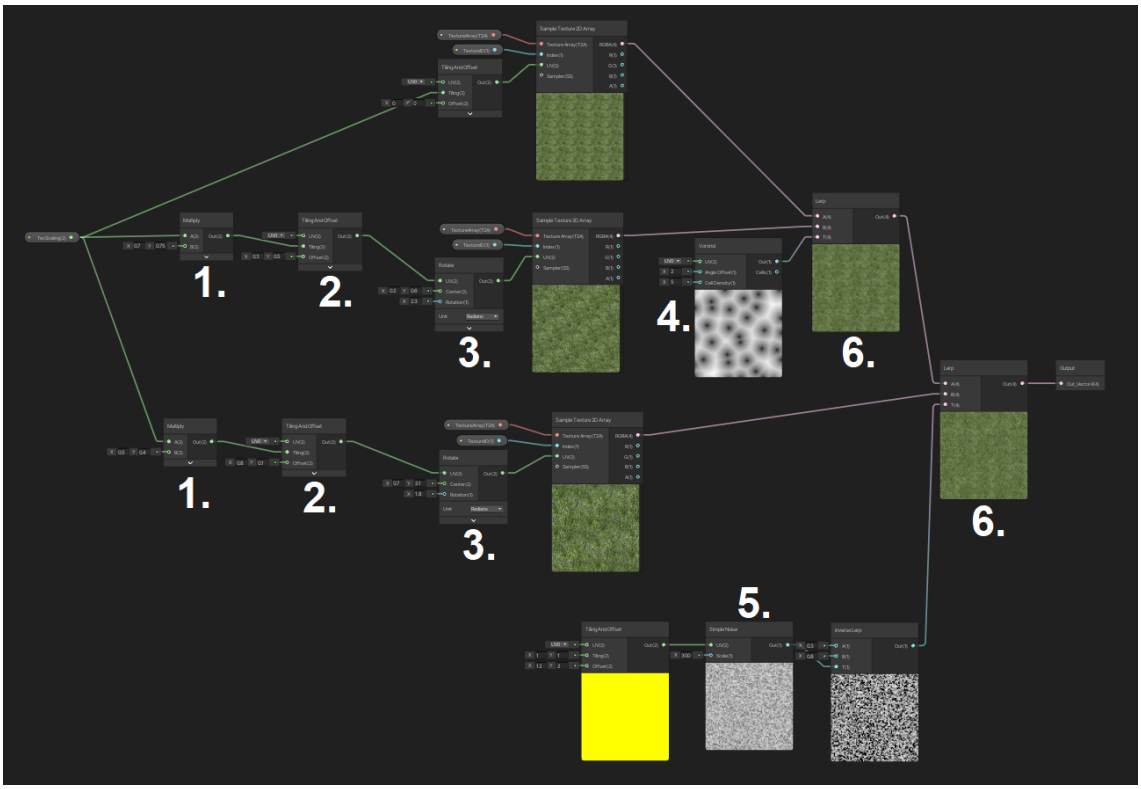



**Abbildung 4.18:** TextureTileSplitter umgesetzt in Shader-Graph
Die Nummerierung folgt Abschnitt 4.3.2
Quelle: Eigene Darstellung

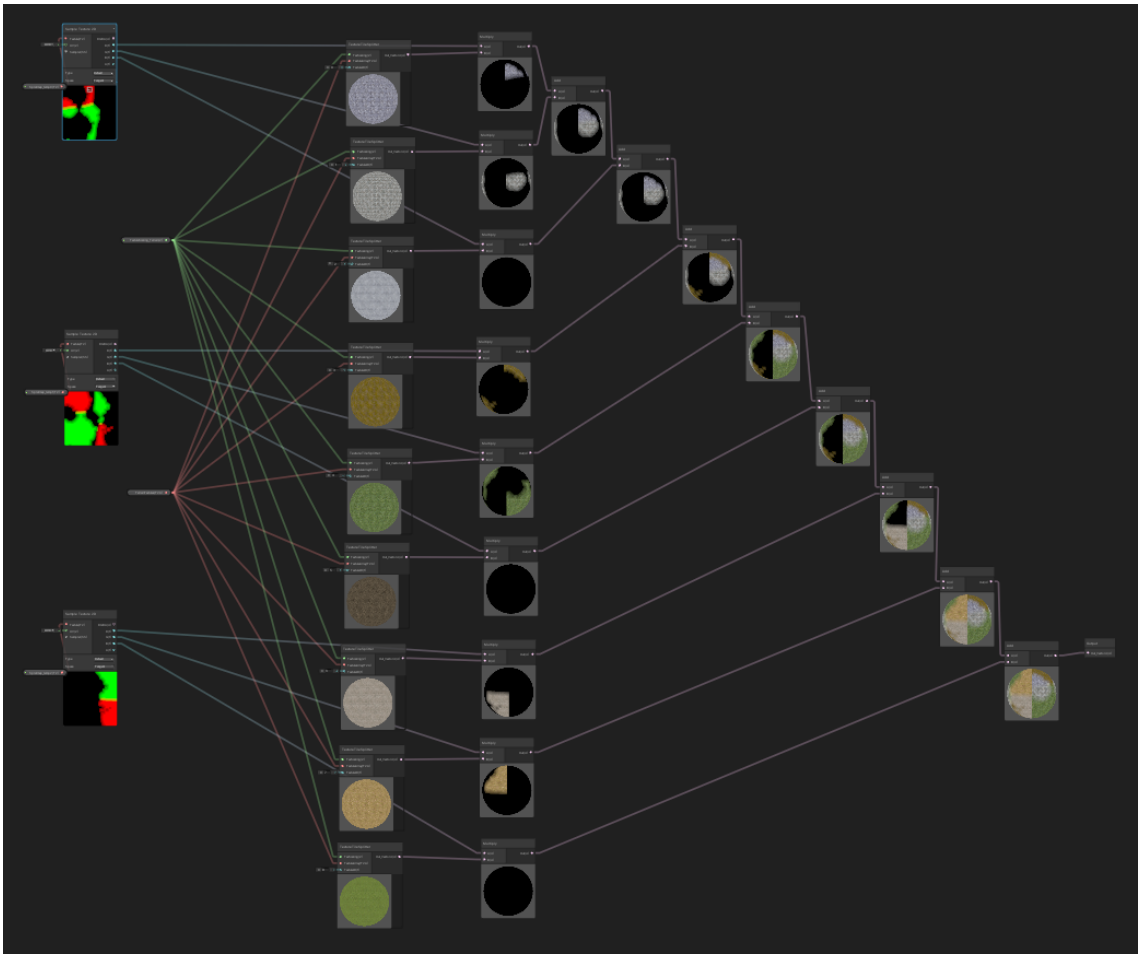


**Abbildung 4.19:** BiomeTextureSplatter umgesetzt in Shader-Graph
Quelle: Eigene Darstellung

2. **TextureTileSplitter anwenden:** Auf die eingelesene Textur wird der TextureTileSplitter angewendet.
3. **Alpha-Kanal-Multiplikation:** Die eingelesene und bearbeitete Textur wird mit dem zum Biom gehörigen Alpha-Kanal der entsprechenden Splatmap multipliziert. Dies stellt sicher, dass die Textur nur dort sichtbar sein wird, wo das Biom vorhanden ist.
4. **Farbwerte addieren:** Die RGBA-Werte der aktuellen Textur werden zur zuletzt berechneten Textur addiert. Dies füllt freie Bereiche auf der entstehenden Großtextur und ermöglicht die korrekte Darstellung der Biome.

### SteepnessTextureSplatter

Der SteepnessTextureSplatter erzeugt eine Alpha-Map, die die steilen Bereiche des Terrains abbildet. Dieser Shader nimmt mehrere Inputs entgegen, darunter die zuvor berechnete Steepness-Map, den Output des BiomeTextureSplatter-Shaders, die Steigungstextur, sowie die Parameter TextureScaling_Steepness, SteepnessIntensity und SteepnessBlending. Der Ablauf dieses Shaders ist wie folgt:

1. **Steigungswerte bearbeiten:** Der SteepnessIntensity-Parameter wird von jedem Wert in der Steepness-Map abgezogen. Dieser Schritt bewirkt, dass niedrige Steigungswerte entfernt werden und nur die steilen Bereiche des Terrains übrig bleiben. Je höher der Parameterwert angesetzt wird, desto größer wird die Fläche an flachem Terrain, das aus der Steepness-Map entfernt wird.
2. **Berechnung der Steilhang-Übergänge:** Alle Werte der Steepness-Map werden mit dem SteepnessBlending-Parameter multipliziert. Ja nach Wahl dieses Parameters werden dadurch die Graustufenwerte der resultierenden Alpha-Map abgeschwächt oder verstärkt, was wiederum kontinuierlich-verlaufende oder aber scharfe Grenzübergänge an den Rändern der Steigungstextur zur Folge haben kann (siehe Abbildung 4.14).
3. **Wertebereich begrenzen:** Die erzeugten Alpha-Werte werden auf den Wertebereich [0,1] begrenzt, um konsistente Werte für eine saubere Interpolation sicherzustellen.
4. **Splatting der Steigungstextur:** Nach den vorherigen Bearbeitungsschritten wird die berechnete Alpha-Map als Basis für das Splatting der Steigungstextur auf die Großtextur verwendet. Vor diesem Schritt erfolgt jedoch noch die Skalierung der Steigungstextur mithilfe des TextureScaling_Steepness-Parameters.

### StreetTextureSplatter

Der StreetTextureSplatter-Shader fokussiert sich auf das Aufbringen der vom Nutzer ausgewählten Straßentextur auf die zu berechnende Großtextur. Hierfür wird die zuvor vorberechnete Street-Map als Alpha-Map genutzt. Die Straßentextur wird dabei mit dem TextureScaling_Street-Parameter skaliert, um die Straßen realistisch auf dem Terrain abzubilden.

## 4.4 Performance des Texturierungssystems

Im folgenden werden die Ergebnisse von Performance-Messungen für ausgewählte Abschnitte des Texturierungssystems vorgestellt. Diese Messungen wurden mithilfe des internen Profiling-Tools der Unity Engine durchgeführt. Die Tests wurden auf einem Desktop-PC mit folgender Hardwarespezifikation ausgeführt:

- Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- 16 GB DDR4 RAM
- NVIDIA GeForce GTX 1070
- Samsung SSD 870 EVO 2TB

Die Laufzeitmessungen wurden in 10 vollständig unabhängigen Durchläufen im Editor bei den folgenden drei unterschiedlichen Terrain-Auflösungen durchgeführt:

- $100 \times 100$ Mesh-Knoten
- $200 \times 200$ Mesh-Knoten
- $400 \times 400$ Mesh-Knoten

Die Wahl dieser Auflösungen stellt sicher, dass die Anzahl der Knotenpunkte linear um einen Faktor von 4 ansteigt. Ein linearer Anstieg der Auflösung stellt sicher, dass die Zuwachsfaktoren, die sich aus den Messwerten ergeben, miteinander verglichen werden können. Diese Zuwachsfaktoren zeigen an, um welchen Faktor sich die Laufzeit im Vergleich zur vorherigen Messung erhöht hat. Sie beziehen sich immer auf die vorhergehende, nächstkleinere Auflösung.

Der erste Prozess, für den Messwerte ermittelt wurden, ist die Erstellung der Steepness-Map. Dabei wurde zwischen der Generierungsphase und der Speicherphase unterschieden. Die Generierungsphase umfasst sämtliche Prozesse, die zur Berechnung der Steepness-Map notwendig sind (siehe Abschnitt 4.3.1). Die Speicherphase hingegen ist für das Schreiben der Daten der Steepness-Map in den Festplattenspeicher verantwortlich.

Weiterhin wurde für die Erstellung aller drei Splatmaps eine gemeinsame Laufzeitmessungen angesetzt. Die Berechnungszeiten der einzelnen Splatmaps sind, abhängig von der Biom-Verteilung, in jedem Durchlauf unterschiedlich verteilt. In Summe muss jedoch, bei einer gleichen Ausgangslage, immer die gleiche Anzahl an Berechnungsschritten für alle drei Splatmaps durchgeführt werden. Aus diesem Grund sollte die Splatmap-Ermittlung als zusammenhängender Prozess behandelt werden. Dieser Prozess kann in die Phasen Initialisierung, Blending und Speichern unterteilt werden. Zur Initialisierung werden Splatmaps mit diskreten Zonen mithilfe der

Biom-Karte angefertigt, wie in Abschnitt 4.3.1 beschrieben wird. Die Blending-Phase umfasst die Berechnung der Blending-Bereiche an den Rändern der diskreten Splatmap-Zonen, die die Grenzen zwischen den Biomen verschwimmen lassen (siehe Abschnitt 4.3.1). Abschließend werden, ähnlich wie auch für die Steepness-Map, sämtliche berechneten Daten in der Speicherphase auf die Festplatte geschrieben.

Die folgenden Messwerte wurden für alle geplanten Terrain-Auflösungen erhoben:

### Erstellung der Steepness-Map

- **Durchschnittliche Gesamtlaufzeit** über allen Messwerten (siehe Tabelle 4.1)
- **Maximale Laufzeitabweichung** als Differenz des höchsten und des geringsten Messwertes (siehe Tabelle 4.1)
- **Durchschnittslaufzeit der Generierungsphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 4.2)
- **Durchschnittslaufzeit der Speicherphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 4.2)

### Erstellung der Splatmaps

- **Durchschnittliche Gesamtlaufzeit** über allen Messwerten (siehe Tabelle 4.3)
- **Maximale Laufzeitabweichung** als Differenz des höchsten und des geringsten Messwertes (siehe Tabelle 4.3)
- **Durchschnittslaufzeit der Initialisierungsphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 4.4)
- **Durchschnittslaufzeit der Blending-Phase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 4.4)
- **Durchschnittslaufzeit der Speicherphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 4.4)

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | 100 × 100 | 200 × 200 | | 400 × 400 | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Durchschn. | 4 | 17 | 4,25 | 70 | 4,12 |
| Max-Abw. | 1 | 3 | 3,00 | 11 | 3,67 |

**Tabelle 4.1:** Generierung Steepness-Map in steigenden Auflösungen: Durchschnittliche Laufzeit und maximale Laufzeitabweichung in 10 Durchläufen sowie deren Zuwachsfaktoren

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | 100 × 100 | 200 × 200 | | 400 × 400 | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Gesamt | 4 | 17 | 4,25 | 70 | 4,12 |
| Generierung | 1,78 | 13,41 | 7,53 | 56 | 4,18 |
| Speichern | 2,22 | 3,59 | 1,62 | 14 | 3,90 |

**Tabelle 4.2:** Generierung Steepness-Map in steigenden Auflösungen: Durchschnittliche Laufzeit der einzelnen Berechnungsphasen in 10 Durchläufen sowie deren Zuwachsfaktoren

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | 100 × 100 | 200 × 200 | | 400 × 400 | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Durchschn. | 907 | 4160 | 4,59 | 14725 | 3,54 |
| Max-Abw. | 32 | 204 | 6,38 | 3647 | 17,88 |

**Tabelle 4.3:** Generierung der 3 Splatmaps in steigenden Auflösungen: Durchschnittliche Laufzeit und maximale Laufzeitabweichung in 10 Durchläufen sowie deren Zuwachsfaktoren

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | 100 × 100 | 200 × 200 | | 400 × 400 | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Gesamt | 907 | 4160 | 4,59 | 14725 | 3,54 |
| Initialisierung | 21 | 104 | 4,95 | 368 | 3,54 |
| Blending | 885 | 4044 | 4,57 | 14343 | 3,55 |
| Speichern | 1 | 12 | 12,00 | 14 | 1,17 |

**Tabelle 4.4:** Generierung der 3 Splatmaps in steigenden Auflösungen: Durchschnittliche Laufzeit der einzelnen Berechnungsphasen in 10 Durchläufen sowie deren Zuwachsfaktoren

# 5 Vegetationsgenerator

Dieses Kapitel beschreibt die Entwicklung eines Algorithmus zur Generierung von Vegetation im dreidimensionalen Terrain. Zu diesem Zweck werden Pflanzenmodelle ausgewählt und auf Grundlage der Biome (siehe Abschnitt 2.2) auf dem Terrain verteilt. In den folgenden Abschnitten wird ein Überblick über die theoretischen Grundlagen der Vegetationsgenerierung und damit verwandte Arbeiten gegeben. Anschließend werden Ziele und mögliche Ansätze zur Realisierung diskutiert, auf deren Grundlage dann ein Konzept für den Vegetationsgenerator erstellt wird. Den Abschluss des Kapitels bilden Ausführungen zur Implementierung des Konzepts.

## 5.1 Theoretische Grundlagen und Verwandte Arbeiten

Dieser Abschnitt beschäftigt sich anfangs mit einigen verwandten Arbeiten, die jeweils unterschiedliche Ansätze zur Vegetationsgenerierung umsetzen. Im Anschluss wird näher auf bestehende Ansätze zur prozeduralen Vegetationsdistribution eingegangen. Zuletzt werden bekannte Möglichkeiten vorgestellt, die Performance in Szenen mit einer hohen Anzahl an Pflanzenmodellen zu verbessern.

### 5.1.1 Verwandte Systeme zur Generierung von Vegetationsszenen

Im folgenden Abschnitt werden verwandte Systeme zur Generierung von Vegetationsszenen betrachtet, die für diese Arbeit von Interesse sein könnten. Zuerst werden die Ansätze von zwei erfolgreichen digitalen Spielen zur Umsetzung ihrer Spielwelten untersucht. Anschließend wird der Ansatz einer wissenschaftlichen Arbeit näher betrachtet.

### Minecraft

Im Spiel *Minecraft* (siehe Abschnitt 1.2) wird Vegetation innerhalb der Spielwelt erzeugt, um dem Spieler Interaktionsmöglichkeiten zu bieten und die Welt anschaulicher zu gestalten. Dabei verfolgt das Spiel den Ansatz der vollständig prozeduralen Synthese. Dem Spieler ist es dabei nicht möglich, Einfluss auf unterschiedliche Aspekte der Vegetationsgenerierung zu nehmen. Sämtliche Entscheidungen bezüglich der Verteilung und der Modellierung von Pflanzen werden durch einen deterministischen Algorithmus getroffen.

Die Vegetationsdichte und die verwendeten Pflanzenmodelle basieren in *Minecraft* auf den Biomen der Spielwelt. Zur Klassifizierung dieser Biome werden während der Generierung der Welt Metadaten in Form von Noise-Maps angelegt, die neben einigen anderen Faktoren auch die Temperatur und den Niederschlag von Regionen definieren [7] [63]. Damit folgt *Minecraft* dem Klassifikationsschema für Biome von Whittaker [56] und ist somit auch mit dem Ansatz zur Generierung von Biomen dieses Projekts verwandt (siehe Abschnitt 2.2). Ein Vergleich der Schemas zur Klassifikation von Biomen nach Whittaker, in Minecraft und in diesem Projekt ist in Abbildung 5.1 dargestellt.

Der Algorithmus zur Weltengenerierung in *Minecraft* verwendet Perlin-Noise-Maps, um die Zugehörigkeitsbereiche einzelner Biome festzulegen. Diese Methode steht im Kontrast zum Ansatz des Biom-Generators (siehe Abschnitt 2.2), bei dem stattdessen Voronoi-Noise verwendet wurde. Sobald sämtliche Biome vom prozeduralen Algorithmus in *Minecraft* festgelegt wurden, wird die Vegetation durch ein Regelsystem erzeugt. Dabei werden die Wachstumsprozesse individueller Pflanzen auf Grundlage ihrer Umgebungsparameter wie Lichtintensität, Untergrund, Freiraum und Spielernähe simuliert. Aus dem Zusammenspiel der regelbasierten Individuen ergibt sich letztendlich ein realistisches Gesamtbild. Die Flora in *Minecraft* wächst zur Laufzeit des Spiels auf Grundlage der Umgebungsparameter dynamisch weiter [34] [58].

### The Elder Scrolls IV: Oblivion

Der hybride Ansatz zur Weltengenerierung des digitalen Rollenspiels *The Elder Scrolls IV: Oblivion* (siehe Abschnitt 1.2) setzte sich auch bei der Erzeugung der Vegetation der Spielwelt durch. Der Prozess der Vegetationserzeugung bestand bei der Entwicklung des Spiels aus zwei Phasen: der Autorenphase und der Generierungsphase. In der Autorenphase haben Künstler die unterschiedlichen Waldarten der Welt manuell in eine Karte gezeichnet. Die dadurch entstandenen Vegetationszonen sind in ihrer Funktion mit den Biomen aus *Minecraft* gleichzusetzen. Diese bestimmen, welche Pflanzenmodelle in dem betrachteten Bereich verwendet werden und

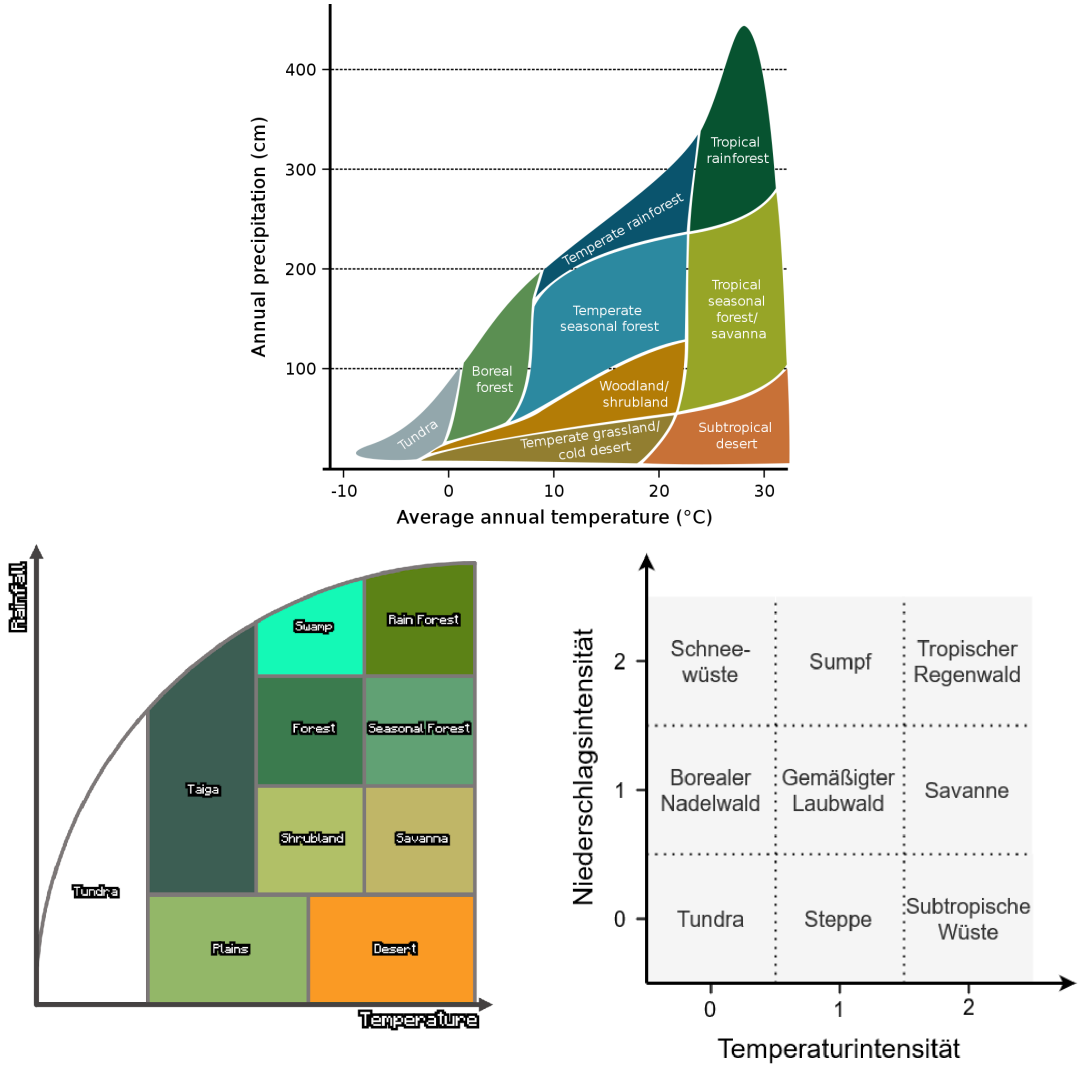



**Abbildung 5.1:** Vergleich der Schemas zur Klassifikation von Biomen nach Whittaker (oben), in Minecraft (links) und in diesem Projekt (rechts) Quellen: [56], [63], nach [13]

welche Vegetationsdichte vorherrschen soll. In der Generierungsphase platziert ein prozeduraler Algorithmus anschließend individuelle Bäume, Sträucher und Gräser in den Grenzen der Vegetationszonen [43].

### Wissenschaftlicher Ansatz

Ein weiteres System zur Generierung von Vegetationsszenen wurde im Rahmen des Papers „Realistic modeling and rendering of plant ecosystems“ [9] entwickelt und beschrieben. Das Ergebnisprogramm namens *EcoSys* implementiert eine grafische Benutzeroberfläche, die es dem Nutzer ermöglicht, Vegetationsszenen individuell anzupassen. Im Rahmen der Arbeit wurden zwei Ansätze verfolgt, die die Generierung von Vegetation auf unterschiedliche Weise umsetzen. Der erste Ansatz befasst sich mit der interaktiven Spezifikation von Pflanzenpopulationen. Dafür wird dem Nutzer ein Zeichen-Tool zur Verfügung gestellt, mit dem Graustufenbilder angefertigt werden können. Mithilfe dieser Graustufenbilder werden anschließend von EcoSys die räumliche Verteilung der Vegetation und die Charakteristiken jeder individuellen Pflanze berechnet.

Der zweite Ansatz der Arbeit befasst sich mit einem Modell, das die Simulation eines wachsenden Ökosystems basierend auf den Interaktionen zwischen individuellen Pflanzen beschreibt. Dafür werden nacheinander Individuen in einer abstrakten Repräsentation als Kreis in einer zweidimensionalen Umgebungskarte erzeugt. Die Pflanze bildet hierbei den Mittelpunkt des Kreises. Die Fläche des Kreises stellt die ökologische Nachbarschaft der Pflanze dar, auch Interaktionsradius genannt. Ein Regelsystem bestimmt ab diesem Zeitpunkt den Ausgang der Interaktion von Pflanzen, deren Interaktionsradien sich überschneiden. In einem Testversuch ist es Deussen et al. [9] gelungen, die Wirkung der sogenannte Selbstausdünnungsregel (englisch: self-thinning rule) für diesen Ansatz der Vegetationsgenerierung nachzuweisen.

Bei der Selbstausdünnungsregel handelt es sich um einen natürlichen Prozess, bei dem eine erhöhte Pflanzensterberate in einem festgelegten Bereich aufgrund des Wettbewerbs um Ressourcen zwischen den Pflanzen erfolgt. Auf gleiche Weise sinkt die Sterberate von Pflanzen in dem Bereich, sobald wieder mehr Ressourcen verfügbar werden. Diese beiden Effekte gleichen einander aus, sodass sich eine Obergrenze an Pflanzen-Biomasse ergibt, die gleichzeitig in dem spezifizierten Bereich existieren kann. Ersichtlich wird der Prozess anhand eines Graphen (siehe Abbildung 5.3), der die logarithmierte Biomasse der logarithmierten Pflanzendichte gegenüberstellt [55].

Der Versuch von Deussen et al. [9] wurde auf einer initialen Menge von Kreisen mit zufälligen Radien aus einem festgelegten Intervall durchgeführt, die zufällig über

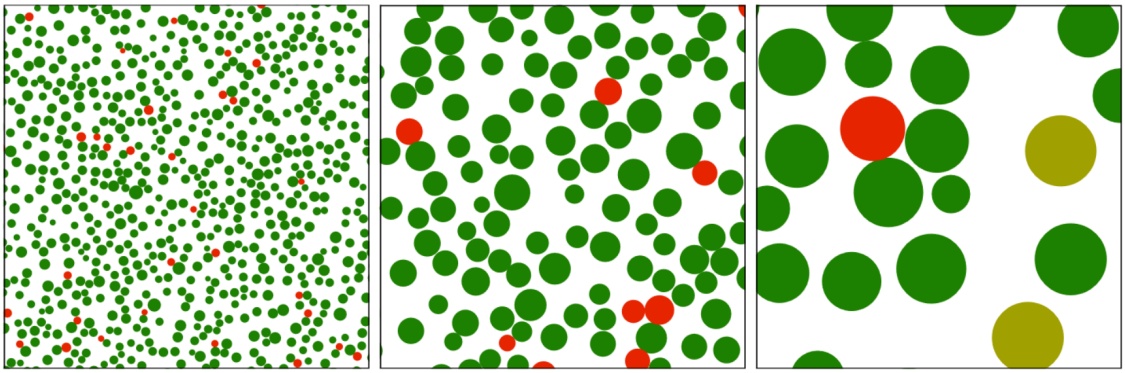


**Abbildung 5.2:** Ausschnitte der Simulation des Selbstausdünnungsprozesses. Unberührte Pflanzen (grün), dominierte Pflanzen (rot) und ausgewachsene Pflanzen (gelb) sind gekennzeichnet. Quelle: [9]

die festgelegte Fläche verteilt wurden. Sämtliche Kreise unterlagen anschließend einem konstanten Wachstum mit fortschreitender Zeit. Einzelne Individuen wurden aus der Simulation entfernt, sobald ihr Interaktionsradius mit einem anderen Interaktionsradius kollidierte und dieses Individuum den kleineren Interaktionsradius besaß. Zusätzlich wurden Individuen entfernt, sobald deren Interaktionsradien die Maximalgröße erreichten (siehe Abbildung 5.2 und Abbildung 5.3) [9].

### 5.1.2 Ansätze zur prozeduralen Asset-Verteilung

Auf Grundlage des kombinierten Ansatzes von Deussen et al. [9] teilt die Arbeit von Lane et al. [29] die gängigen Methoden zur Modellierung von Vegetationsdistributionen in zwei Kategorien ein. Die Autoren unterscheiden zwischen dem Local-to-Global-Ansatz und dem Global-to-Local-Ansatz. Methoden der Local-to-Global-Modellierung werden charakterisiert durch die Entstehung globaler Distributionseigenschaften durch die Simulation von lokalen Interaktionen zwischen individuellen Pflanzen. Beim Global-to-Local-Ansatz hingegen werden die Positionen individueller Pflanzen von globalen Vegetationsdichte-Karten abgeleitet [29].

Beide von Lane et al. [29] definierten Ansätze zur prozeduralen Asset-Verteilung tendieren dazu, unterschiedliche Vegetationsarten gleichmäßig in den vorgegebenen Bereichen zu verteilen. In der Realität tendieren Pflanzenarten jedoch häufig dazu, lokale Cluster einer gemeinsamen Vegetationsart zu bilden [6]. Dieses Phänomen kann unter anderem durch lokale Umweltfaktoren ausgelöst werden, die Vegetation mit bestimmten Eigenschaften beim Wettbewerb um Ressourcen wie Licht und Wasser bevorteilen. Ein weiterer Grund ist, dass die Samen einer Pflanze in vielen Fällen in der Nähe des Elter zu Boden gehen. Solche lokalen Cluster wirken sich auf das

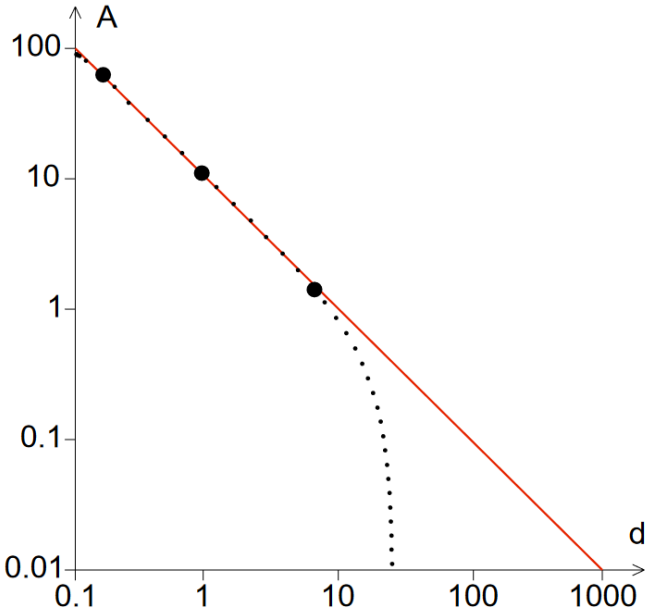



**Abbildung 5.3:** Pflanzenoberfläche in Relation zur Pflanzendichte. Die drei gekennzeichneten Punkte stellen die in Abbildung 5.2 gezeigten Stadien der Simulation dar
Quelle: [9]

visuelle Erscheinungsbild einer Vegetationsszene aus. Um dieses Phänomen berücksichtigen zu können, nutzen Lane et al. [29] die statistische Methode der Hopkins Indexe [26], um das Ausmaß der Gruppenbildung innerhalb von Vegetationsverteilungen zu messen. Auf Grundlage dieser Methode zur Quantifizierung von Clustern war es Lane et al. [29] möglich, Maßnahmen zu implementieren, um die Cluster-Bildung innerhalb von Vegetationsszenen zu kontrollieren.

### 5.1.3 Performance in großen Vegetationsszenen

Deussen et al. präsentieren in „Interactive visualization of complex plant ecosystems“ [8] Ansätze, um große Vegetationsszenen effizient berechnen zu können. Die Autoren ziehen den Ansatz von Weber et al. [53] zur Abstraktion von einzelnen Pflanzenmodelle mithilfe von Punkten und Linien in LOD-Abstufungen als Grundlage für ihre Arbeit heran und weiten das Prinzip auf ganze Pflanzenpopulationen aus. LOD steht hierbei für „Level of Detail“ und bezeichnet Detailabstufungen bei der Darstellung von Objekt-Modellen. Abhängig von den ausgewählten LOD-Distanzabstufungen kann der Wechsel zwischen unterschiedlichen Modellen für einen Betrachter einer Szene sichtbar sein. Dieser Effekt wird „Popping“ genannt und wird in realistischen Szenen als unnatürlich und störend empfunden. Deussen et al. [8] entwickelten eine Methode, die zwischen den Pflanzenmodellen unterschiedlicher Komplexität interpoliert, um Popping zu vermeiden. Dies wird erreicht durch ein kontinuierlich fortschreitendes Ersetzen der originalen Geometrie eines Modells durch Punkte und Linien mit wachsendem Abstand zum Betrachter.

Eine weitere Möglichkeit zur Performance-Optimierung von großen Vegetationsszenen, die von Deussen et al. [8] identifiziert wurde, ist die Verwendung eines LOD-Systems innerhalb von komplexen, weitreichenden Pflanzenmodellen. Befindet sich ein Betrachter in einer geringen Distanz vor einer relativ großen Pflanze, dann sollte die nahe Geometrie der Pflanze in möglichst hoher Qualität gerendert werden. Die Geometrie, die sich jedoch auf der Rückseite der Pflanze in einer größeren Distanz zum Betrachter befindet, muss nicht zwingend in der höchstmöglichen Auflösung gerendert werden, um eine realistische Darstellung zu erzielen. In standardmäßigen LOD-Systemen werden Modelle zusammenhängend bearbeitet, wodurch immer sämtliche Teile eines Modells in der gleichen Auflösung dargestellt werden. Der Ansatz von Deussen et al. [8] teilt die Geometrie von relativ großen, komplexen Modellen in Octrees auf. Die Zellen eines solchen Octrees werden dann vom LOD-System einzeln bei der Darstellung der Szene bearbeitet, wodurch sich Detailabstufungen innerhalb eines Pflanzenmodells ergeben können und die Masse an zu rendernder Geometrie verringert werden kann.

Als Erweiterung dieses Ansatzes können LOD-Systeme mit einer Methode namens „Importance Reduction“ weiter optimiert werden. Hierbei wird die Geometrie sämt-

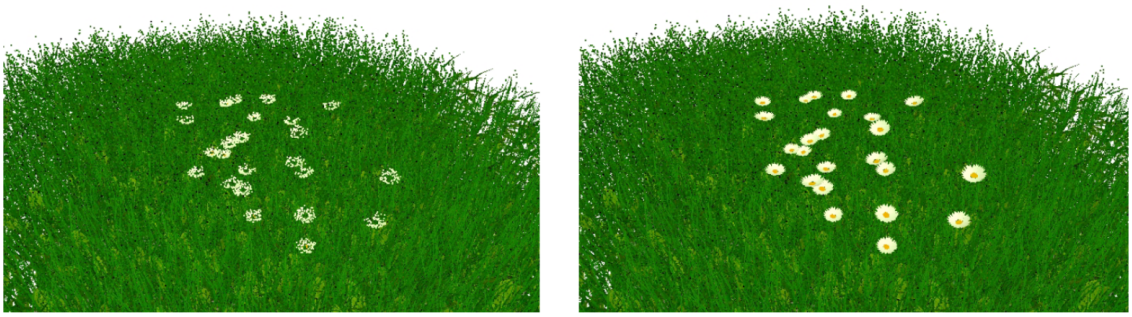


**Abbildung 5.4:** Gänseblümchen in einheitlich reduzierter Auflösung (links) und unter Anwendung von Importance Reduction (rechts) Quelle: [8]

licher Pflanzenmodelle innerhalb des LOD-Octrees bestimmt, die von besonderer Relevanz für die visuelle Darstellung ist. Diese Teilmodelle, die in Octree-Zellen mit höherer Priorität gespeichert sind, werden dann bei der Bearbeitung durch das LOD-System weniger stark in ihrer geometrischen Auflösung herabgestuft, sobald sich die Distanz zum Betrachter vergrößert. Dadurch verlieren Pflanzenmodelle, trotz intensivem Einsatz von LOD-Systemen, nicht ihre primären visuellen Merkmale [8]. Ein Beispiel für die Anwendung dieser Methode ist in Abbildung 5.4 dargestellt.

Ein weiterer Performance-Faktor, der bei der Generierung großer Vegetationsszenen Beachtung finden muss, ist der Speicherplatzverbrauch der Pflanzenmodelle, die zur Laufzeit geladen werden. Es existiert die Möglichkeit, Pflanzenmodelle prozedural mithilfe von sogenannten L-Systemen zur Laufzeit auf Grundlage einiger Parameter zu berechnen. L-Systeme sind mathematische Ersetzungsformalismen, die unter anderem zur realistischen Modellierung von Pflanzen verwendet werden können [42]. Durch die Berechnung der Pflanzenmodelle zur Laufzeit ist es möglich, den Speicherbedarf der zugrundeliegenden Anwendung zu senken. Anstatt der Modell-Dateien müssten lediglich die Parameterwerte des Algorithmus auf der Festplatte vorliegen. Dabei sollte jedoch berücksichtigt werden, dass dieser Ansatz zu einer erhöhten Berechnungslast führt und unter Umständen die Laufzeit-Performance verringert. Die Verwendung dieser Technik muss deshalb unter Berücksichtigung des jeweiligen Anwendungsfalls abgewogen werden [8].

## 5.2 Konzeption des Vegetationsgenerators

Im ersten Abschnitt dieses Kapitels werden die Anforderungen an den Vegetationsgenerator definiert. Auf deren Grundlage wird dann im Anschluss ein mögliches

Konzept diskutiert und erstellt.

### 5.2.1 Zielsstellung

Im Folgenden werden die Anforderungen an den Vegetationsgenerator mithilfe der in Abschnitt 1.3 definierten Ziele des Projekts spezifiziert. Die Aufgabe des Vegetationsgenerators ist es, Pflanzenmodelle auf dem vorgegebenen Terrain (siehe Abschnitt 2.1) zu erzeugen. Die Basis für diese Berechnungen bildet die Biomeinteilung (siehe Abschnitt 2.2) des Terrains. Bei der Generierung der Vegetation werden folgende Kriterien berücksichtigt:

1. **Realismus:** Eine realistische Darstellung von Vegetation in dreidimensionalen, offenen Welten kann, abhängig vom vorherrschenden Ökosystem und dem Grad der Urbanisierung, einen signifikanten Beitrag zur Immersion des Nutzers leisten. Das Ziel ist es, bekannte Muster aus der realen Flora der Erde zu modellieren, um eine für den Betrachter glaubwürdige Darstellung der Welt zu erzielen.

2. **Flexibilität:** Das Projekt soll für ein breites Spektrum an Anwendungen einsetzbar gemacht werden. Zu diesem Zweck ist es notwendig, dass der Vegetationsgenerator Funktionen unterstützt, die dem Nutzer eine individuelle Anpassung der Vegetation auf seine Bedürfnisse ermöglichen. Dabei stehen besonders die Vegetationsdistribution und die Pflanzenmodelle im Fokus.

3. **Performance:** Üblicherweise umfasst die Menge der Pflanzenmodelle einen Großteil der Geometrie von digitalen offenen Welten. Um eine reibungslose Nutzererfahrung gewährleisten zu können, muss sowohl die Performance des Algorithmus zur Generierung der Vegetation, als auch die Performance der Anwendung mit existierender Vegetation zur Laufzeit beachtet werden. Der Nutzer sollte sich hierbei zu keinem Zeitpunkt in seiner Interaktion mit dem System eingeschränkt fühlen.

4. **Benutzbarkeit:** Die Implementierung einer intuitiven Benutzerschnittstelle ist kein zentraler Bestandteil dieser Arbeit. Es genügt deshalb, wenn der Vegetationsgenerator sein volles Potenzial bei der Verwendung durch einen Nutzer entfaltet, der sowohl Erfahrung mit der Unity Game Engine besitzt, als auch in das System eingewiesen wurde. Von größerer Relevanz ist, die Arbeitsabläufe des Nutzers zur Generierung der Vegetation möglichst gering zu halten. Außerdem gilt es, Programmfehler bei der Bedienung des Vegetationsgenerators so gut es geht zu vermeiden. Dazu zählen beispielsweise Parameter, die nicht ihren angegebenen Zweck erfüllen und fehlplatzierte Pflanzenmodelle.

### 5.2.2 Diskussion und Konzept

Im Folgenden werden Ansätze zur Verteilung von Pflanzenmodellen im Terrain diskutiert. Auf Grundlage dieser Diskussion wird einer der Ansätze zur weiteren Umsetzung ausgewählt. Anhand des gewählten Ansatzes werden im Anschluss Konzepte für Methoden und Parameter zur Platzierung von Pflanzenmodellen erstellt. Den Abschluss des Kapitels bildet ein Abschnitt, der sich damit beschäftigt, wie die zu platzierenden Pflanzenmodelle ausgewählt werden.

#### Diskussion von Ansätzen zur Vegetationsdistribution

Die Methode zur Asset-Verteilung spielt eine entscheidende Rolle bei der Konzipierung des Vegetationsgenerators. Der gewählte Ansatz bildet die Basis für alle weiteren Überlegungen dieses Abschnitts. Die unterschiedlichen Ansätze bringen verschiedene Vorteile und Nachteile mit sich, die sorgfältig unter Berücksichtigung der in Abschnitt 5.2.1 festgelegten Ziele abgewogen werden müssen. Betrachtet werden der Local-to-Global Ansatz und der Global-to-Local Ansatz (siehe Abschnitt 5.1.2).

**Local-to-Global:** Die Simulation realer Wachstumsprozesse beim Local-to-Global Ansatz bietet Potenzial für einen hohen Grad an Realismus. Beispielsweise können mit dieser Methode natürliche Phänomene, wie die Selbstausdünnung von Vegetation auf dicht bewachsenen Flächen, nachgestellt werden (siehe Abschnitt 5.1.1). Ein weiterer Vorteil ist, dass die Wachstumssimulation, falls gewünscht, auch zur Laufzeit der jeweiligen Zielanwendung weiter durchgeführt werden kann, um eine dynamische Flora zu schaffen. Die Qualität der Ergebnisse wird beim Local-to-Global Ansatz dadurch bestimmt, wie ausgereift der Algorithmus für die Simulation ist.

Flexibilität muss bei diesem Ansatz über die Eingabe von Parametern herbeigeführt werden, die den Ablauf der Wachstumssimulation beeinflussen. Die Vegetationsdistribution könnte dabei sowohl durch Umweltparameter wie Höhe, Steigung und Gewässer als auch durch Interaktionsparameter wie Interaktionsradius, Interaktionsart und Interaktionsintensität beeinflusst werden.

Im Kriterium Performance schneidet der Local-to-Global Ansatz tendenziell schlechter ab als der Global-to-Local Ansatz. Beim Global-to-Local Ansatz muss lediglich ein einziges Mal über die Anzahl $N$ aller potenziellen Vegetationsstandorte des Terrains iteriert werden, um alle nötigen Pflanzenmodelle zu platzieren. Der Local-to-Global Ansatz führt seine Berechnungen ebenfalls über der Anzahl $N$ aller potenziellen Vegetationsstandorte aus, wiederholt diese Berechnungen aber in $K$ vielen Simulationsschritten.

$$N \leq NK$$

Zusätzlich muss beachtet werden, dass eine Wachstumssimulation während der Laufzeit einer Zielanwendung weitere Herausforderungen an die Performance mit sich bringt. Die Simulationsschritte müssen hierbei in regelmäßigen Abständen und beliebig häufig parallel zur Anwendung ausgeführt werden. Zwischen den einzelnen Simulationsschritten kann sich jedoch ein größeres Zeitfenster befinden, in dem andere nötige Berechnungen durchgeführt werden können. Hier könnten Maßnahmen zur Laufzeitoptimierung nötig sein, falls der Nutzer durch die zusätzliche Berechnungslast eingeschränkt werden sollte.

Der Arbeitsaufwand für den Nutzer kann beim Local-to-Global Ansatz relativ gering gehalten werden. Eventuell werden mehrere Simulationsdurchläufe mit nachfolgenden Parameteranpassungen notwendig sein, bis der Nutzer eine gewünschte Vegetationsverteilung erhält. Dieser Prozess ist jedoch auch abhängig von der Erfahrung des jeweiligen Nutzers. In jedem Fall werden die Anforderungen dieses Projekts an die Benutzerfreundlichkeit bei diesem Ansatz eingehalten.

**Global-to-Local:** Wie realistisch die Ergebnisse beim Global-to-Local Ansatz ausfallen, hängt von der Qualität der verwendeten Vegetationsdichte-Karten ab. Betrachtet man einzelne Pflanzen und deren direkte Nachbarschaft im Detail, dann wird klar, dass dort nicht die gleiche Genauigkeit bei der Modellierung erreicht werden kann, wie beim Local-to-Global Ansatz. Hier sind weitere Berechnungen, zusätzlich zu globalen Vegetationsdistribution nötig, um im Kriterium Realismus mit dem Local-to-Global Ansatz konkurrieren zu können.

Die Flexibilität der Vegetationsverteilung kann beim Global-to-Local Ansatz über die interaktive Anpassung der Vegetationsdichte-Karten herbeigeführt werden. Diese Karten und die daraus resultierenden Ergebnisse sind vorhersehbarer, als die Wachstumssimulation des Local-to-Global Ansatzes. Der Nutzer erhält dadurch einen höheren Grad an Kontrolle über die Vegetationsverteilung als beim Local-to-Global Ansatz.

Das Kriterium Performance fällt für den Global-to-Local Ansatz zufriedenstellend aus. Die einmalige Iteration über alle potenziellen Vegetationsstandorte genügt, um diese mit der Vegetationsdichte-Karte abzugleichen und die passenden Pflanzenmodelle in der Szene zu platzieren. Es fallen keine weiteren Berechnungen zur Laufzeit einer potenziellen Zielanwendung an.

Der Arbeitsaufwand für den Nutzer hängt beim Global-to-Local Ansatz davon ab, wie der Prozess zur Erstellung der Vegetationsdichte-Karten abläuft. Die Generierung der Karte unter zuhilfenahme benutzerdefinierter Parameter hält den Aufwand für den Nutzer gering. Sollen Nutzer die Vegetationsdichte-Karten manuell mithilfe von Zeichen-Tools anfertigen, dann erhöht das zwar die Flexibilität der Anwendung

und bietet Potenzial für hohen Realismus, der Nutzer müsste jedoch viel Arbeitszeit in den Schritt der Kartenerstellung investieren.

### Zusammenfassung der Ansätze zur Vegetationsdistribution

Unter Betrachtung der genannten Vorteile und Nachteile wird klar, dass sowohl der Local-to-Global Ansatz als auch der Global-to-Local Ansatz das Potenzial bieten, die Vegetationsszenen des Projektes zufriedenstellend umzusetzen. In das Projekt integriert wird der Global-to-Local Ansatz, da dieser dem Nutzer einen höheren Grad an Kontrolle über die Vegetationsverteilung bei einer besseren Performance bietet. Der gewählte Ansatz muss jedoch mit Verfahren zur realistischen Platzierung einzelner Pflanzen unter Beachtung von Nachbarschaftsbeziehungen kombiniert werden, um die Anforderungen an den Realismus zu erfüllen. Diese Methoden werden im folgenden Abschnitt genauer beschrieben.

Zur Erstellung der Vegetationsdichte-Karten wird ein Noise-basierter Ansatz verwendet werden, um dem Nutzer die Arbeitslast der manuellen Kartenanfertigung abzunehmen. Das manuelle Zeichnen der Karten würde den Arbeitsfluss innerhalb der Anwendung zu sehr stören. Die Vegetationsdichte-Karten werden prozedural mithilfe eines Perlin-Noise auf Grundlage einiger benutzerdefinierter Parameter generiert. Dadurch ergeben sich jedoch Einbußen bei der Flexibilität, da der Nutzer nicht mehr in der Lage sein wird, jedes Pixel der Vegetationsdichte-Karten selbst zu bestimmen.

### Methoden und Parameter zur Vegetationsplatzierung

Ob ein Pflanzenmodell an einem Punkt im Terrain platziert werden soll, ist nach dem gewählten Global-to-Local Ansatz grundsätzlich von einer Vegetationsdichte-Karte abhängig. Durch zusätzliche Berechnungen auf dieser Karte und der Berücksichtigung der vorhandenen Umwelt können der Realismus und die Flexibilität der Anwendung verbessert werden. Im folgenden Abschnitt wird ein algorithmischer Entscheidungsprozess entworfen, der in der fertiggestellten Anwendung bestimmt, ob ein Pflanzenmodell an einem bestimmten Standort platziert wird. Dafür werden folgende benutzerdefinierte Parameter benötigt, die nachfolgend näher erläutert werden:

- NoiseDensity (Frequenz des Perlin-Noise)
- NoiseOffset (Verschiebung des Perlin-Noise)
- ForestSize (Schwellwert: Berücksichtigung des Perlin-Noise)

- ForestSpread (Streuung der Vegetation)
- ForestDensity (Dichte der Vegetation)
- SteepnessInfluence (Schwellwert: Berücksichtigung von Steigungen)
- TreeHeightLimit (Baumgrenze in Höhenlagen)
- ModelScaling (Größenskalierung der Pflanzenmodelle)

Zuerst muss eine Vegetationsdichte-Karte erstellt werden, um den Global-to-Local Ansatz umsetzen zu können. Dafür wird die von Unity vorgegebene Perlin-Noise-Funktion verwendet. Diese benötigt 2D-Koordinaten als Eingabe und liefert einen Rückgabewert zwischen 0 und 1. Um dem Nutzer Kontrolle über die Erstellung der Vegetationsdichte-Karte zu verleihen, werden die beiden Parameter *NoiseDensity* und *NoiseOffset* an dieser Stelle mit den jeweiligen Koordinaten verrechnet, um die Frequenz und die Verschiebung des Perlin-Noise beeinflussen zu können. Die Auswirkungen dieser Parameter werden beim Vergleich von Abbildung 5.5 mit Abbildung 5.6 und Abbildung 5.7 deutlich.

Die Perlin-Noise-Funktion $P(x, y)$, wird im Folgenden als Eingabe für eine Fitness-Funktion $F(P(x, y))$ verwendet. Diese Funktion entscheidet, ob ein Pflanzenmodell platziert werden soll. Es ist jedoch noch unklar, ab welchem Wert von F alle Anforderungen an die Vegetationsplatzierung erfüllt sind. Um diese Unklarheit zu beseitigen, wird der *ForestSize*-Parameter eingeführt. Dieser Parameter fungiert als Schwellenwert. Ein Wertevergleich zwischen $F(P(x, y))$ und dem *ForestSize*-Parameter entscheidet, ob mit der Platzierung des Pflanzenmodells an dem aktuellen Standort fortgefahren wird. Diese Funktionsweise wird durch einen Vergleich der Abbildung 5.5 und Abbildung 5.8 veranschaulicht.

Die Berechnung von $F(P(x, y))$ erfolgt mithilfe der Parameter *ForestSpread*, *SteepnessInfluence* und *TreeHeightLimit*. Der *ForestSpread*-Parameter bewirkt eine gleichmäßige Streuung an den Rändern der Vegetationsgebiete. Dafür wird ein gleichverteilter zufälliger Wert S aus dem Intervall $[-ForestSpread, ForestSpread]$ ermittelt. Anschließend erfolgt die folgende Berechnung:

$$F_1(P(x, y)) = P(x, y) + S$$

Die Auswirkungen dieser Berechnung werden beim Vergleich der Abbildung 5.5 und Abbildung 5.9 ersichtlich.

Anschließend erfolgt die Ermittlung, ob das Terrain am betrachteten Standort zu steil für die Vegetationsplatzierung ist. Hierbei wird eine Karte mit Steigungsdaten (siehe Abschnitt 4.3) zusammen mit dem *SteepnessInfluence*-Parameter als Referenz

herangezogen. Ein einfacher Wertevergleich genügt, um diese Entscheidung zu treffen. Falls die Steigung zu hoch ist, wird die Vegetationsplatzierung nicht vollständig ausgeschlossen, aber der Fitness-Wert wird durch Subtraktion einer signifikanten Konstante $k_1$ gesenkt:

$$F_2(F_1(x)) = F_1(x) - (k_1 * isPointTooSteep)$$

Durch dieses Vorgehen ist es einzelnen Ausreißern immer noch möglich, unter ansonsten sehr guten Bedingungen an steilen Klippen platziert zu werden. Ein Vergleich der Abbildung 5.5 und Abbildung 5.11 zeigt die Relevanz dieser Berechnung für ein realistisches Gesamtbild. Ohne diese Berechnungen würde die Vegetation sämtliche Steilklippen überwuchert.

Eine Baumgrenze wird mithilfe des *TreeHeightLimit*-Parameters über die Fitness-Funktion festgelegt. Hierzu muss dieser Parameter lediglich mit der Höhenlage des potenziellen Vegetationsstandortes verglichen werden. Liegt der Standort in zu großer Höhe, dann wird auch hier der Fitness-Wert durch Subtraktion einer signifikanten Konstante $k_2$ gesenkt:

$$F(P(x, y)) = F_3(F_2(x)) = F_2(x) - (k_2 * isPointTooHigh)$$

Die Auswirkungen einer niedrigen Baumgrenze werden durch einen Vergleich der Abbildung 5.5 und Abbildung 5.12 deutlich.

Die letzten beiden Parameter, die Einfluss auf die Vegetation haben, sind die Parameter *ForestDensity* und der *ModelScaling*. Der *ForestDensity*-Parameter legt fest, in welcher Frequenz die Fitness-Funktion abgetastet wird. Bei einer niedrigeren Frequenz verringert sich die Anzahl potenzieller Vegetationsstandorte, wie beim Vergleich der Abbildung 5.5 mit Abbildung 5.10 ersichtlich wird. Der *ModelScaling*-Parameter beeinflusst hingegen die Größe der Pflanzenmodelle, die im Terrain platziert werden. Ein Vergleich der Abbildung 5.5 und Abbildung 5.13 zeigt, dass der Nutzer ein passendes Maß für diesen Parameter finden muss, da die Pflanzenmodelle ansonsten ungewollt ineinander übergehen können.

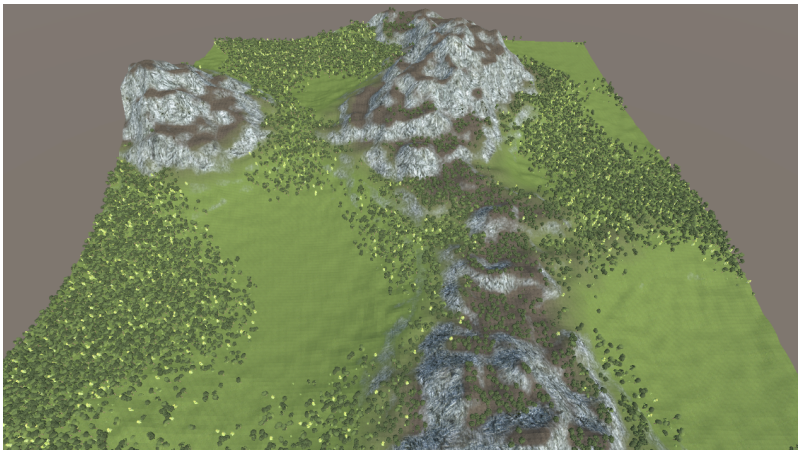


**Abbildung 5.5:** Standardmäßige Vegetation
Quelle: Eigene Darstellung

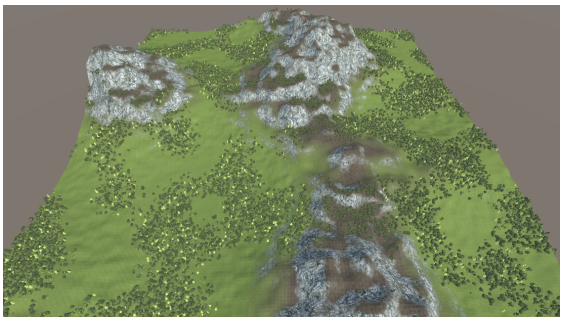


**Abbildung 5.6:**
Doppelte NoiseDensity
Quelle: Eigene Darstellung

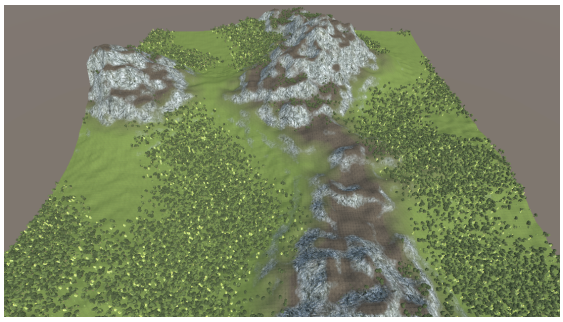


**Abbildung 5.7:**
NoiseOffset in X-Richtung
Quelle: Eigene Darstellung

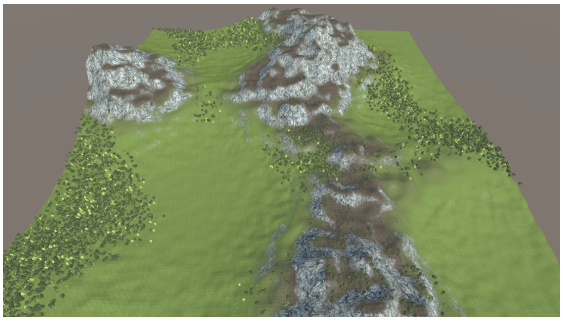


**Abbildung 5.8:**
Niedrige ForestSize
Quelle: Eigene Darstellung

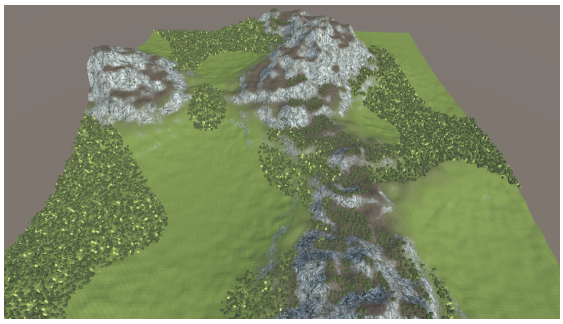


**Abbildung 5.9:**
Niedrige ForestSpread
Quelle: Eigene Darstellung

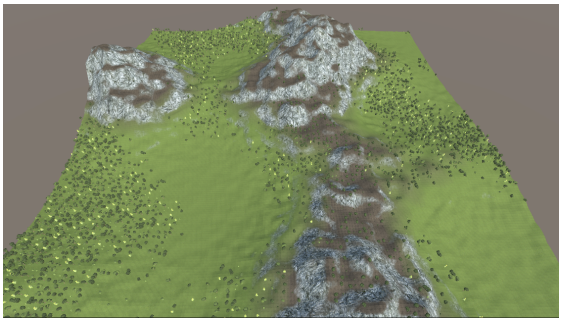


**Abbildung 5.10:**
Niedrige ForestDensity
Quelle: Eigene Darstellung

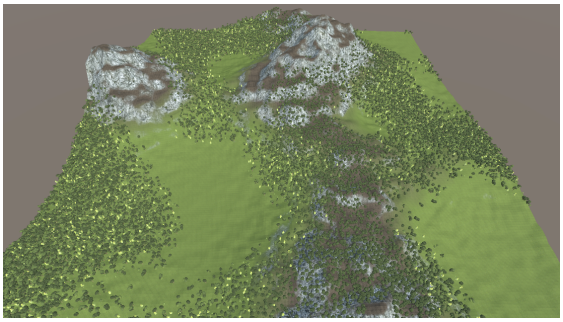


**Abbildung 5.11:**
Niedrige SteepnessInfluence
Quelle: Eigene Darstellung

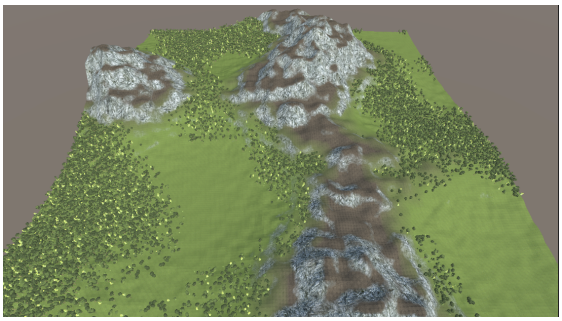


**Abbildung 5.12:**
Niedriges TreeHeightLimit
Quelle: Eigene Darstellung

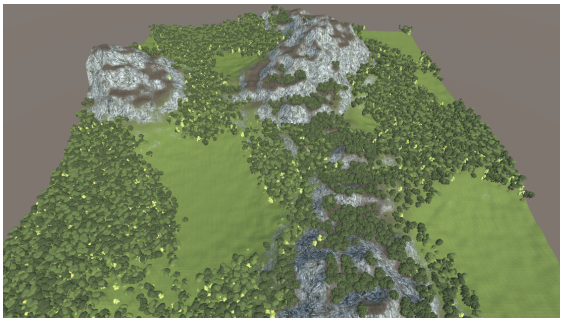


**Abbildung 5.13:**
Doppeltes ModelScaling
Quelle: Eigene Darstellung

Unabhängig von den benutzerdefinierten Parametern ist anzumerken, dass Vegetationsstandorte, die sich unterhalb der globalen Wassergrenze (siehe Abschnitt 2.1.2) befinden oder sich mit Siedlungsstrukturen (siehe Kapitel 6) überschneiden, in dieser Version des Weltengenerators von der Vegetationsplatzierung ausgeschlossen werden. Die Generierung von Unterwasserpflanzen ist ein interessanter Ansatz für zukünftige Arbeiten, in dieser Arbeit liegt jedoch der Fokus auf der terrestrischen Vegetation.

Die Abtastung des Geländes zur Ermittlung der potenziellen Vegetationsstandorte erfolgt in gleichmäßigen Abständen. Nachdem die Entscheidung getroffen wurde, eine Pflanze an einem bestimmten Vegetationsstandort zu platzieren, wird den Koordinaten des ausgewählten Pflanzenmodells ein zufälliger lokaler Offset hinzugefügt. Dies geschieht, um ein unnatürliches repetitives Muster in der Vegetationsszene zu vermeiden. Der Abstand zwischen den einzelnen Abtastungen des Geländes kann als $D$ beschrieben werden. In diesem Fall ergibt sich sowohl für die X-Koordinate (Breitenkoordinate) als auch die Z-Koordinate (Längenkoordinate) des Pflanzenmodells ein gleichverteilt zufälliger Offset aus dem Intervall $[-0.5D, 0.5D]$. Die Y-

Koordinate (Höhenkoordinate) wird von diesem Vorgang nicht beeinflusst. Abschließend wird das zu platzierende Modell um einen gleichverteilt zufälligen Winkel aus dem Intervall $[0, 360]$ um die Y-Achse (Höhenachse) rotiert, um repetitive Muster zu vermeiden, die aufgrund gleicher Modellausrichtungen entstehen könnten.

### Auswahl der Pflanzenmodelle

Es existieren unterschiedliche Ansätze, um zu bestimmen, welche Modelle in einer Vegetationsszene verwendet werden. Im Folgenden werden die Vorteile und Nachteile der prozeduralen Synthese (siehe Abschnitt 5.1.3) von Pflanzenmodellen mit dem klassischen Ansatz verglichen, bei dem sämtliche Modelle vorgefertigt sind und aus dem Festplattenspeicher geladen werden. Dabei werden die in Abschnitt 5.2.1 festgelegten Anforderungen als Qualitätsmaßstab herangezogen.

In der realen Welt gleicht keine Pflanze genau einer anderen, selbst wenn diese derselben Art angehören. Durch prozedurale Synthese von Pflanzenmodellen kann eine beliebige Anzahl an unterschiedlichen Modellen erzeugt werden. Aus diesem Grund besitzt der Ansatz das Potenzial, einen höheren Grad an Realismus in einer Vegetationsszene zu erreichen, als es mit einer endlichen Anzahl vorgefertigter Modelle möglich wäre. Dieser Vorteil ist jedoch abhängig von der Qualität des Algorithmus zur Generierung der Modelle. Die Berechnungen müssten auf einem komplexen Umweltmodell aufbauen, um Umweltparameter der realen Welt wie Wind, Regen und Sonneneinstrahlung berücksichtigen zu können. Falls die Entwicklung eines ausgereiften Algorithmus zur Modellsynthese nicht möglich ist, bietet die manuelle Anfertigung realistischer Modelle durch Künstler und deren Einsatz in der Vegetationsszene eine konkurrenzfähige Alternative zur prozeduralen Modellsynthese.

Im Kriterium der Flexibilität bietet der Einsatz vorgefertigter Modelle einen Vorteil gegenüber der Modellsynthese. Der Nutzer hat die Möglichkeit, eine beliebige Anzahl an vorgefertigten Modellen festzulegen, die in den Vegetationsszenen verwendet werden sollen. Diese Modelle können ohne Einschränkungen nach den individuellen Bedürfnissen und Vorstellungen des Nutzers gestaltet werden. Im Gegensatz dazu ermöglicht die Modellsynthese dem Nutzer lediglich die Anpassung von einstellbaren Parametern, die die Generierung der Modelle beeinflussen. Die volle Kontrolle über die eingesetzten Modelle kann der Nutzer dadurch nicht erhalten.

Wie in Abschnitt 5.1.3 beschrieben, ist die Entscheidung zwischen den beiden Ansätzen unter Berücksichtigung der Performance gleichzusetzen mit der Entscheidung zwischen einem erhöhten Berechnungsaufwand während der Platzierung der Pflanzenmodelle oder einem erhöhten Bedarf der Anwendung an Festplattenspeicher. Da Festplattenspeicher in den letzten Jahren relativ günstig geworden ist, stellt das Abspeichern einer hohen Anzahl an Modellen aktuell kein großes Problem dar. Dieser

Trend scheint sich auch in den folgenden Jahren fortzusetzen. Aus diesem Grund ist im Kriterium Performance der Ansatz der vorgefertigten Pflanzenmodelle bevorteilt.

Der Ansatz, bei dem der Arbeitsaufwand, den der Nutzer investieren muss, um individualisierte Vegetationsszenen zu erzeugen, geringer ist, hängt von der Implementierung und der Erfahrung des Nutzers ab. Bei der prozeduralen Synthese muss der Nutzer mehrere Parameter so aufeinander abstimmen, dass die von ihm gewünschten Vegetationsarten erzeugt werden können. Mit steigender Anzahl an Parametern wird dieser Prozess komplexer und damit für den Nutzer erschwert. Dadurch sind unter Umständen viele Durchläufe mit erneuten Parameteranpassungen notwendig. Bei der Verwendung vorgefertigter Modelle liegt es hingegen in der Verantwortung des Nutzers, die benötigten Modelle zu beschaffen und in die Anwendung zu integrieren. Der Nutzer könnte die Modelle selbständig anfertigen, was zu einem großen Arbeitsaufwand führen würde. Alternativ ist die Nutzung der Assets von Drittanbietern eine Möglichkeit, den Arbeitsaufwand für den Nutzer zu verringern.

Unter Berücksichtigung der diskutierten Vorteile und Nachteile wird deutlich, dass die prozedurale Synthese von Pflanzenmodellen das Potenzial bietet, den Realismus einer Szene zu optimieren. Bis jetzt verwendet der Prototyp jedoch noch kein Umweltmodell, das umfassend genug ist, um als Basis für einen qualitativ hochwertigen Algorithmus zur Vegetationssynthese zu dienen. Außerdem übersteigt der benötigte Aufwand für Nachforschungen, Konzeption und Implementierung eines solchen Algorithmus die Möglichkeiten dieser Arbeit. Die Synthese von Pflanzenmodellen bleibt ein interessanter Ansatz für zukünftige Forschungen. Stattdessen wird dieses Projekt mithilfe von vorgefertigten Pflanzenmodellen umgesetzt, die sowohl bei der Flexibilität als auch bei der Performance Vorteile bieten.

Die Art des platzierten Pflanzenmodells an einem spezifischen Standort soll abhängig vom vorherrschenden Biom entschieden werden. Dadurch werden Gebiete innerhalb der Welt geschaffen, deren Eigenschaften (Relief, Texturen, Vegetation) dem Nutzer aus realen Ökosystemen bekannt sind, was zur Wahrnehmung einer realistischen Welt beiträgt. Dem Nutzer soll es dafür ermöglicht werden, seine vorgefertigten Pflanzenmodelle auf alle existierenden Biome-Arten aufzuteilen. Die einzelnen Modelle werden dann während des Prozesses der Vegetationsgenerierung nur in den ihnen zugeordneten Biome-Arten instanziiert.

Die Auswahl eines Pflanzenmodells für einen bestimmten Standort aus der Vegetationsmenge des zugehörigen Bioms wird durch eine Wahrscheinlichkeitsverteilung bestimmt. Der Nutzer hat dabei die Möglichkeit, die Vorkommenswahrscheinlichkeit jedes Modells in der Vegetationsmenge frei mithilfe von Parametern festzulegen. In diesem Zusammenhang stellt sich jedoch die Frage, wie Pflanzenmodelle in den Übergangsbereichen zwischen den Biomen (siehe Abschnitt 4.2.2) ausgewählt

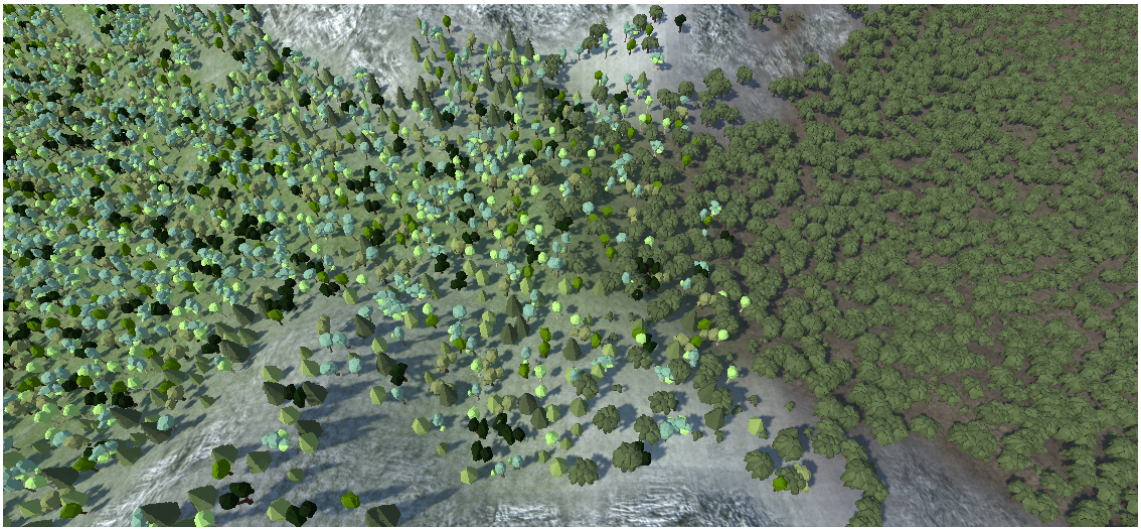


**Abbildung 5.14:**
Übergang zwischen gemäßigtem Wald und Sumpfvegetation
Quelle: Eigene Darstellung

werden sollen. Hierfür werden die interpolierten Texturwerte aus Abschnitt 4.2.2 verwendet, um auf ihrer Grundlage zunächst eines der am Standort vorliegenden Biome mithilfe einer weiteren Wahrscheinlichkeitsverteilung auszuwählen. Der Wahrscheinlichkeitswert für ein Biom ergibt sich dabei aus dem Anteil seines Farbwerts in der Splatmap an der Summe aller Farbwerte der Biome an dem betrachteten Standort. Dieses Verfahren führt zu einem fließenden Übergang der Vegetation zwischen den Biomen (siehe Abbildung 5.14) und unterstützt damit den Realismus.

## 5.3 Implementierung des Vegetationsgenerators

Der folgende Abschnitt beschreibt die Umsetzung des Konzepts aus Abschnitt 5.2.2. Zunächst werden die Abläufe erläutert, die notwendig sind, um die Vegetationsgenerierung vorzubereiten. Im Anschluss wird ein Überblick über die Umsetzung des Algorithmus zur Vegetationsgenerierung vermittelt. Für die Umsetzung dieses Projektabschnitts wurde die Programmiersprache C# unter Verwendung der Pakete *System.Collection* und *UnityEngine* eingesetzt. Die gezeigten Pflanzenmodelle stammen aus dem Unity Asset Store [45] [15] [52] [30]. Ein Überblick über den Algorithmus zur Vegetationsgenerierung ist in Abbildung 5.15 dargestellt.

```
GenerateVegetation() {
    vegetationDensityMap = calcPerlinNoise(noiseDensity, noiseOffset);
    removeCurrentVegetation();

    foreach(pixel (x,y) in vegetationDensityMap) {
        biomeID = determineBiome(x,y);
        prefabSet = determinePrefabSet(biomeID);
        locationCoords = calcLocationCoords(x, y, forestDensity);
        fitness = calcFitness(locationCoords);

        if(isFitnessEnough(fitness, forestSize)) {
            modelCoords = calcLocalOffset(locationCoords);
            modelHeight = approximateOffsetHeight(modelCoords);

            if( modelHeight > waterHeight &&
                !isLocationBlocked(modelCoords)
            ) {
                prefab = determinePrefab(prefabSet)
                modelRotation = randomRotationY();
                instantiatePlant(
                    prefab,
                    modelCoords,
                    modelHeight,
                    modelRotation,
                    modelScaling
                );
            }
        }
    }
}
```

**Abbildung 5.15:**
Pseudocode des Vegetationsgenerators
Quelle: Eigene Darstellung

### 5.3.1 Vorbereitung der Vegetationsgenerierung

Der Vegetationsgenerator benötigt umfassende Eingabedaten, um sämtliche Berechnungen, die in Abschnitt 5.2 konzipiert wurden, umsetzen zu können. Die folgenden Daten gehören zur Eingabe des Vegetationsgenerators:

- ForestSize (Schwellwert: Berücksichtigung des Perlin-Noise)
- ForestSpread (Streuung der Vegetation)
- ForestDensity (Dichte der Vegetation)
- SteepnessInfluence (Schwellwert zur Berücksichtigung von Steigungen)
- TreeHeightLimit (Baumgrenze in Höhenlagen)
- ModelScaling (Größenskalierung der Pflanzenmodelle)
- VegetationDensityMap (Perlin-Noise zur Vegetationsdistribution)
- PrefabSets (Mengen von Pflanzenmodellen nach Biomen)
- VegetationInstances (Mengen an instanziierten Modellen)

Die ersten sechs Parameter wurden in Abschnitt 5.2.2 diskutiert und werden bei der Bestimmung des Vegetationsstandorts und der Platzierung von Pflanzenmodellen benötigt. Die *VegetationDensityMap* wird als zweidimensionales Perlin-Noise aus den benutzerdefinierten Parametern *NoiseDensity* und *NoiseOffset* berechnet (siehe Abschnitt 5.2.2 und Abbildung 5.15, Zeile 2). Dazu wird über alle Knoten des Terrain-Mesh iteriert, wobei für jeden Knoten ein Perlin-Noise-Wert berechnet und in der *VegetationDensityMap* abgespeichert wird. Um die Frequenz und Verschiebung des Perlin-Noise wie geplant (siehe Abschnitt 5.2.2) umzusetzen, werden die in Abbildung 5.16 in zeile 3 und 4 dargestellten Operationen mithilfe des *NoiseDensity*-Parameters und des *NoiseOffset*-Parameters durchgeführt.

### 5.3.2 Umsetzung der Vegetationsgenerierung

Im Folgenden wird beschrieben, wie innerhalb des Algorithmus die zu platzierenden Pflanzenmodelle ausgewählt werden. Anschließend wird näher auf die Standortbestimmung zur Vegetationsplatzierung eingegangen. Zuletzt werden die Prozesse beschrieben, die während der Platzierung eines Modells an einem bestimmten Standort ablaufen.

```
GenerateVegetationDensityMap() {
    foreach(node (x,y) in mesh) {
        xCoord = x * noiseDensity + noiseOffset;
        yCoord = y * noiseDensity + noiseOffset;
        noiseValue = Mathf.PerlinNoise(xCoord, yCoord);
        vegetationDensityMap[x,y] = noiseValue;
    }
    return vegetationDensityMap;
}
```

**Abbildung 5.16:**
Pseudocode zur Generierung der *VegetationDensityMap*
Quelle: Eigene Darstellung

### Auswahl von Pflanzenmodellen

Der erste Schritt zur Ermittlung eines passenden Pflanzenmodells an einer bestimmten Koordinate im Mesh ist die Bestimmung des zugrundeliegenden Bioms mithilfe der `DetermineBiome`-Funktion (siehe Abbildung 5.17). Biome werden innerhalb des Projekts mithilfe eines ID-Tupels $(T, F)$ identifiziert. Die erste Stelle $T$ des Tupels zeigt dabei die vorliegende Temperaturintensität an, während die zweite Stelle $F$ des Tupels die Feuchtigkeitsintensität darstellt (siehe Abschnitt 2.2). Während der Input der `DetermineBiome`-Funktion 2D-Koordinaten des aktuellen Standorts sind, stellt der Output die ermittelte *BiomeID* $(T, F)$ dar.

Die `DetermineBiome`-Funktion kann in zwei getrennte Phasen unterteilt werden. In der ersten Phase werden die Farbintensitätswerte sämtlicher Biome aus ihren korrespondierenden Splatmaps ausgelesen (siehe Abbildung 5.17, Zeile 7–9). Parallel dazu werden diese Farbwerte in einer Variable summiert (siehe Abbildung 5.17, Zeile 10–12). Die zweite Phase umfasst die zufällige Bestimmung eines Bioms. Dabei werden die Anteile einzelner Intensitätswerte von Biomen an der Gesamtsumme aller Intensitätswerte als Wahrscheinlichkeitsverteilung für diese Ermittlung herangezogen (siehe Abbildung 5.17, Zeile 17–27).

Nachdem das Biom für den betrachteten Standort ermittelt wurde, können mithilfe der zugehörigen *BiomeID* die benötigten Pflanzenmodelle ausgewählt werden. Diese Modelle sind innerhalb der Anwendung in einer eigens angelegten Datenstruktur namens *PrefabSet* organisiert (siehe Abbildung 5.18). Ein *PrefabSet* umfasst sämtliche unterschiedlichen Pflanzenmodelle, die in einem bestimmten Biom instanziiert werden sollen, sowie einige Metadaten. Alle diese Modelle und Daten können beliebig vom Nutzer bearbeitet werden. Die Variable *name* erhält den Namen des Bioms, um das *PrefabSet* zuordnen zu können. Der *PrefabDensityPenalty*-Parameter beeinflusst die Vegetationsdichte in jedem separaten Biom und steht im Zusammenhang

```
DetermineBiome(x,y) {

    //determine biome ratios
    biomeIntensities = new float[3,3];
    tempID = 0;
    foreach(splatmap) {
        biomeIntensities[tempID,0] = splatmap.GetPixel(x,y).r
        biomeIntensities[tempID,1] = splatmap.GetPixel(x,y).g
        biomeIntensities[tempID,2] = splatmap.GetPixel(x,y).b
        pixelValueSum += biomeIntensities[tempID,0];
        pixelValueSum += biomeIntensities[tempID,1];
        pixelValueSum += biomeIntensities[tempID,2];
        tempID++;
    }

    //determine biome at random based on ratios
    randomValue = Random(0,pixelValueSum);
    lowerBound = 0;
    foreach(biomeID (tempID, precID)) {
        biomeIntensity = biomeIntensities[tempID, precID]
        upperBound = lowerBound + biomeIntensity;
        if(lowerBound <= randomValue <= upperBound) {
            return (tempID, precID);
        } else {
            lowerBound = upperBound;
        }
    }
}
```

**Abbildung 5.17:**
Pseudocode zur Bestimmung des Bioms an einem Standort
Quelle: Eigene Darstellung

```
public class PrefabSet {
    public string name;
    public int prefabDensityPenalty;
    public PrefabData[] prefabData;
}

public class PrefabData {
    public GameObject prefab;
    public int placementFrequency;
}
```

**Abbildung 5.18:**
Pseudocode der Definition der *PrefabSet*-Datenstruktur
Quelle: Eigene Darstellung

mit dem *ForestDensity*-Parameter. Die genauen Berechnungen zur Vegetationsdichte werden später im Abschnitt 5.3.2 näher erklärt. Zuletzt implementiert die *PrefabSet*-Datenstruktur noch ein Array von *PrefabData*-Objekten. Jedes *PrefabData*-Objekt speichert hierbei ein einzelnes Pflanzenmodell und die gewünschte Vorkommenswahrscheinlichkeit dieses Modells im zugehörigen Biom als *PlacementFrequency*-Parameter.

Die Auswahl eines bestimmten Pflanzenmodells $m$ innerhalb des ermittelten Bioms erfolgt zufällig aus der Menge $M$ aller *PrefabData*-Objekte innerhalb des zum Biom gehörenden *PrefabSet*-Objekts (siehe Abbildung 5.15, Zeile 18). Die Wahrscheinlichkeitsverteilung dafür ergibt sich aus den Anteilen der einzelnen *PlacementFrequency*-Parametern $F$ an der Summe aller *PlacementFrequency*-Parameter und wird durch folgende Formel beschrieben:

$$P(m) = \frac{F_m}{\sum_{n=1}^{|M|} F_n}$$

### Ermittlung potenzielle Vegetationsstandorte

Der erste Schritt zur Ermittlung potenzieller Vegetationsstandorte im Gelände ist die Berechnung der Frequenz (siehe Abbildung 5.15, Zeile 8), mit der dieses abgetastet wird. Diese Frequenz ergibt sich aus einer Kombination des *ForestDensity*-Parameters $P$ als globaler Wert und dem *PrefabDensityPenalty*-Parameter $P_b$ aus dem *PrefabSet* des zugrundeliegenden Bioms $b$. Grundsätzlich wird das Mesh in gleichmäßigen Abständen $D_V$ entlang der X-Achse und der Z-Achse abgetastet. Dabei ergibt sich $D_V$ aus der Distanz zwischen zwei Mesh-Knoten $D_K$ über folgende Formel:

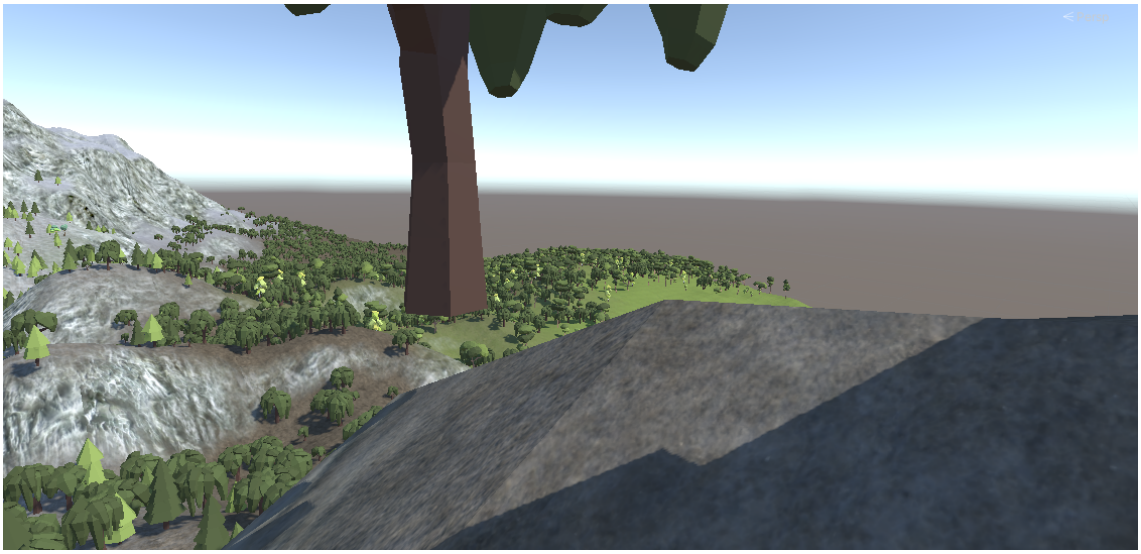


**Abbildung 5.19:** Modell wurde versetzt, ohne die Höhenlage anzupassen
Quelle: Eigene Darstellung

$$D_V = D_K * (P + P_b)$$

Dadurch kann die Abtastrate als Basis der Vegetationsdichte sowohl in der gesamten Karte als auch in einzelnen Biomen über die Parameter beeinflusst werden. Der in Abschnitt 5.2.2 besprochene lokale Offset jeder Pflanze wird mithilfe des Unity-Core-Moduls *Random* ermittelt (siehe Abbildung 5.15, Zeile 12).

Aufgrund des lokalen Offset in der X-Z-Ebene des Terrains kann es passieren, dass einige potenzielle Vegetationsstandorte in Y-Richtung (Höhenlage) nicht mehr korrekt auf dem Mesh aufliegen. Dadurch könnte eine Fehlplatzierung von Pflanzenmodellen erfolgen (siehe Abbildung 5.19), was die Benutzbarkeit der Anwendung negativ beeinflusst. Um dieses Problem zu lösen, wird die neue Höhenlage der Standorte approximiert (siehe Abbildung 5.15, Zeile 13). Hierzu findet eine lineare Interpolation zwischen den Höhenwerten statt, die an den Koordinaten des ursprünglichen Abtastwerts und des zum neuen Pflanzenstandort nächstliegenden benachbarten Abtastwerts vorliegen (siehe Abbildung 5.20).

### Platzierung von Pflanzenmodellen

Die Entscheidung, ob ein Pflanzenmodell an einem der potenziellen Vegetationsstandorte platziert werden soll, wird mithilfe der in Abschnitt 5.2.2 konzipierten Fitness-Funktion getroffen (siehe Abbildung 5.21). Der Basiswert der Funktion wird durch die *VegetationDensityMap* vorgegeben, die über den *ForestSpread*-Parameter

```
ApproximateOffsetHeight(sampleCoords, offsetCoords, offset) {
    neighborCoords = closestNeighborSample(offsetCoords);
    lerpValue = Max(|offset.x|, |offset.y|);
    newHeight =
        Lerp(   Height(sampleCoords),
                Height(neighborCoords),
                lerpValue);
    return newHeight;
}
```

**Abbildung 5.20:**
Pseudocode der Definition der *PrefabSet*-Datenstruktur
Quelle: Eigene Darstellung

```
1  CalcFitness(x,y) {
2      fitness = vegetationDensityMap[x,y];
3      fitness += Random(-forestSpread, forestSpread);
4      if(steepness(x,y) > 1-steepnessInfluence) {
5          fitness -= 0.7;
6      }
7      if(height(x,y) > maxTreeHeight) {
8          fitness -= 0.7;
9      }
10     return fitness;
11 }
```

**Abbildung 5.21:**
Pseudocode der Fitness-Funktion
Quelle: Eigene Darstellung

angepasst wird (siehe Abbildung 5.21, Zeile 3), der sich im Intervall $[0, 0.5]$ bewegt. Weiterhin wird der Fitness-Wert aufgrund der *SteepnessInfluence* und der *MaxTreeHeight* signifikant verringert, falls die zugehörigen Umweltvariablen gegen eine Vegetationsplatzierung sprechen (siehe Abbildung 5.21, Zeilen 4–9). Ob der ermittelte Fitness-Wert die Vegetationsplatzierung zulässt, wird durch einen Wertevergleich entschieden (siehe Abbildung 5.15, Zeile 11). Der Vergleichswert für diese Operation ist der maximale Fitness-Wert von 1.5, der sich aus dem maximalen Wert der *VegetationDensityMap* addiert mit dem maximalen *ForestSpread*-Parameter ergibt. Dieser Vergleichswert kann durch den Nutzer mithilfe des *ForestSize*-Parameters verringert werden, um die Vegetationsausbreitung zu steuern (siehe Abschnitt 5.2.2).

Zusätzlich dazu finden Überprüfungen statt, ob sich der aktuelle Standort unterhalb der globalen Wassergrenze befindet und ob sich auf dem aktuellen Standort bereits Siedlungsstrukturen befinden (siehe Kapitel 6). Folgende Aussagen müssen also erfüllt sein, damit ein Pflanzenmodell in der Landschaft platziert wird (siehe

Abbildung 5.15, Zeilen 11 und 15–16):

- $Fitness(x, y) > 1.5 - forestSize$
- $Height(x, y) > waterHeight$
- $isSettlement(x, y) == false$

Abschließend wird die Instanziierung des Pflanzenmodells an den ausgewählten Koordinaten durchgeführt (siehe Abbildung 5.15, Zeile 19–25). Dafür wird das ausgewählte Prefab, die Standortkoordinaten, der approximierte Höhenwert, der *ModelScaling*-Parameter sowie ein zufälliger Rotationswinkel um die Y-Achse (Höhenachse) aus dem Intervall $[0, 360]$ verwendet.

Diese durch den Algorithmus erzeugten Modellinstanzen werden in der *VegetationInstances*-ArrayList als Unity-GameObjects für die zukünftige Verwaltung abgespeichert. Um die Vegetationsszene auf Basis veränderter Parameter aktualisieren zu können, ist es notwendig, vorher die Menge aller bestehenden Instanzen von Pflanzenmodellen vorheriger Durchläufe zu leeren (siehe Abbildung 5.15, Zeile 3). Im Anschluss kann dann mit der Berechnung einer neuen Vegetationsmenge begonnen werden.

## 5.4 Performance des Vegetationsgenerators

Im Folgenden werden die Ergebnisse von Performance-Messungen zur Generierung der Vegetations vorgestellt. Diese Messungen wurden mithilfe des internen Profiling-Tools der Unity Engine durchgeführt. Die Tests wurden auf einem Desktop-PC mit folgender Hardwarespezifikation ausgeführt:

- Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- 16 GB DDR4 RAM
- NVIDIA GeForce GTX 1070
- Samsung SSD 870 EVO 2TB

Die Laufzeitmessungen wurden in 10 vollständig unabhängigen Durchläufen im Editor bei den folgenden drei unterschiedlichen Terrain-Auflösungen durchgeführt:

- $100 \times 100$ Mesh-Knoten
- $200 \times 200$ Mesh-Knoten
- $400 \times 400$ Mesh-Knoten

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | 100 × 100 | 200 × 200 | | 400 × 400 | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Durchschn. | 82,5 | 270,1 | 3,27 | 1258,5 | 4,66 |
| Max-Abw. | 12.6 | 28.6 | 2,27 | 330,6 | 11,56 |

**Tabelle 5.1:** Generierung der Vegetation in steigenden Terraingrößen: Durchschnittliche Laufzeit und maximale Laufzeitabweichung in 10 Durchläufen sowie deren Zuwachsfaktoren

Die Wahl dieser Auflösungen stellt sicher, dass die Anzahl der Knotenpunkte linear um einen Faktor von 4 ansteigt. Ein linearer Anstieg der Auflösung stellt sicher, dass die Zuwachsfaktoren, die sich aus den Messwerten ergeben, miteinander verglichen werden können. Diese Zuwachsfaktoren zeigen an, um welchen Faktor sich die Laufzeit im Vergleich zur vorherigen Messung erhöht hat. Sie beziehen sich immer auf die vorhergehende, nächstkleinere Auflösung.

Dargestellt sind die Laufzeiten der `GenerateVegetation`-Funktion (siehe Abbildung 5.15). Dabei wird zwischen drei Phasen unterschieden. In der ersten Phase findet die Berechnung des Perlin-Noise für die Vegetationsdichte-Karte statt. Die zweite Phase umfasst das Entfernen sämtlicher Pflanzenmodelle, die noch im Gelände vorhanden sind. Zuletzt, in der dritten Phase, wird die neue Vegetation generiert.

Die folgenden Messwerte wurden erhoben:

- **Durchschnittliche Gesamtlaufzeit** über allen Messwerten (siehe Tabelle 5.1)
- **Maximale Laufzeitabweichung** als Differenz des höchsten und des geringsten Messwertes (siehe Tabelle 5.1)
- **Durchschnittslaufzeit der Perlin-Noise-Phase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 5.2)
- **Durchschnittslaufzeit der CleanUp-Phase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 5.2)
- **Durchschnittslaufzeit der Generierungsphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Tabelle 5.2)

| | Auflösung des Terrains | | | | |
|---|---|---|---|---|---|
| | $100 \times 100$ | $200 \times 200$ | | $400 \times 400$ | |
| | Zeit (ms) | Zeit (ms) | Zuwachsfaktor | Zeit (ms) | Zuwachsfaktor |
| Gesamt | 82,5 | 270,1 | 3,27 | 1258,5 | 4,66 |
| Perlin Noise | 3,2 | 11,8 | 3,69 | 45,7 | 3,87 |
| CleanUp | 2,5 | 6,9 | 2,76 | 32,6 | 4,72 |
| Generierung | 76,8 | 251,4 | 3,27 | 1180,2 | 4,69 |

**Tabelle 5.2:** Generierung der Vegetation in steigenden Terraingrößen: Durchschnittliche Laufzeit der einzelnen Berechnungsphasen in 10 Durchläufen sowie deren Zuwachsfaktoren

# 6 Siedlungsgenerator

Dieses Kapitel beschreibt die Entwicklung eines Algorithmus zur Generierung von Siedlungen im dreidimensionalen Terrain. In den folgenden Abschnitten wird ein Überblick über die theoretischen Grundlagen der Siedlungsgenerierung und einigen damit verwandte Arbeiten gegeben. Anschließend werden Ziele und mögliche Ansätze zur Realisierung diskutiert, auf deren Grundlage dann ein Konzept für den Siedlungsgenerator erstellt wird. Den Abschluss des Kapitels bilden Ausführungen zur Implementierung des Konzepts.

## 6.1 Theoretische Grundlagen und Verwandte Arbeiten

Dieser Abschnitt beschäftigt sich mit einigen verwandten Arbeiten, die jeweils unterschiedliche Ansätze zur Siedlungsgenerierung umsetzen. Zuerst wird die Siedlungsgenerierung durch Inferenz aus realen Daten betrachtet. Im Anschluss wird näher auf die prozedurale Siedlungsgenerierung eingegangen.

### 6.1.1 Generierung urbaner Regionen durch Inferenz

In „Visualization of Simulated Urban Spaces“ von Vanegas et al. [51] wird ein Ansatz zur Generierung von Stadtkarten vorgestellt, der auf der Simulation urbaner Strukturen basiert und Daten aus der realen Welt verwendet. Die Eingabedaten für den Algorithmus umfassen:

- Georeferenzierte Luftbilder
- Straßen- und Parzellendaten aus Geoinformationssystemen
- Sozioökonomische Daten

Mit diesen Daten prognostiziert eine Simulation die zukünftigen Werte verschiedener Variablen in der urbanen Region. Diese Variablen im Simulationsmodell können beispielsweise Immobilienpreise sein, die sich aus der Interaktion von Agenten ergeben, die um Immobilien konkurrieren. Der Algorithmus, der von Vanegas et al. [51]

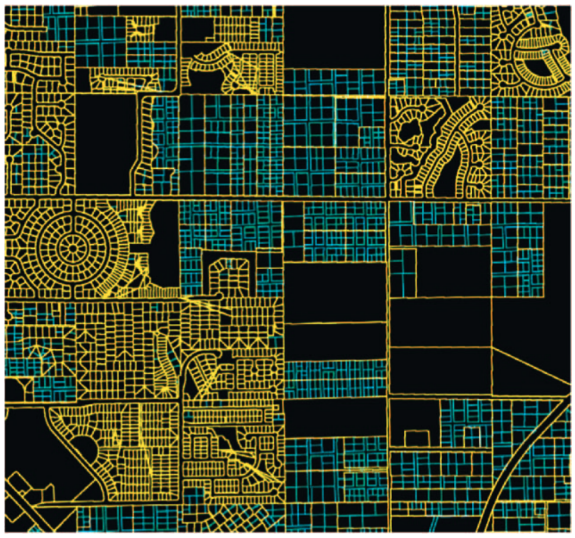


**Abbildung 6.1:** Aus realen Daten inferierte Stadtkarte
Quelle: [51]

präsentiert wird, sammelt stochastische Daten über die ursprüngliche Stadtstruktur und nutzt die Variablen aus verschiedenen Simulationsstadien anschließend, um plausible zukünftige Stadtkarten abzuleiten. Über ein gesondertes Tool können diese Stadtkarten zu beliebigen Zeitpunkten der Simulation für den Nutzer visualisiert werden (siehe Abbildung 6.1).

### 6.1.2 Siedlungserstellung durch prozedurale Generierung

In „Procedural generation of villages on arbitrary terrains“ von Emilien et al. [11] wird eine weitere Möglichkeit zur Erstellung von Siedlungsstrukturen untersucht, wobei der vorgestellte Ansatz prozedurale Modellierungstechniken verwendet. Im Gegensatz zu Arbeiten wie die von Parish et al. [40] und Greuter et al. [19], die die Generierung großer Städte behandeln, konzentriert sich die Arbeit von Emilien et al. [11] auf den weniger erforschten Bereich der Generierung von ländlichen Siedlungen. Der Fokus liegt hierbei auf kleinen europäischen Dörfern, die sich aufgrund vorteilhafter Geländeeigenschaften entwickelt haben. In dieser Arbeit wird das natürliche Wachstum solcher Siedlungen in mehreren Schritten simuliert.

Die Siedlungsgenerierung mit dem vorgestellten Algorithmus findet in einem eingeschränkten Gebiet innerhalb eines dreidimensionalen Terrains statt. Als Eingabe dienen sowohl Umweltdaten als auch ein Wachstumsszenario, das vom Nutzer ausgewählt werden muss. Dieses Wachstumsszenario erlaubt es dem Nutzer, die Entwicklung der Siedlung mithilfe von Parametern zu beeinflussen. Zu Beginn der

Siedlungsgenerierung werden sogenannte Interest-Maps auf Grundlage von Umweltdaten, Siedlungsdaten und dem Wachstumsszenario angelegt, die zu einer gemeinsamen Interest-Funktion aggregiert werden. Mithilfe der Interest-Funktion können im Anschluss über einen stochastischen Positionierungsprozess die Koordinaten von Gebäuden berechnet werden. Die Interest-Maps werden auf Basis folgender Funktionen in berechnet [11]:

- **Sociability:** Diese Funktion beschreibt die Tendenz der Gruppenbildung von Häusern in einer Siedlung.
- **Worship:** Diese Funktion modelliert die Tendenz, dass Häuser in der Nähe von religiösen Elementen wie Kirchen oder Statuen errichten werden.
- **Accessibility:** Mithilfe dieser Funktion wird die Nähe zu Straßen einberechnet. Damit wird berücksicht, dass Siedler ihre Häuser bevorzugt mit einer einfachen Anbindung an die lokale Infrastruktur bauen.
- **Slope:** Je nach Gebäudetyp können unterschiedliche Steigungen bei der Wahl des Standortes eines Gebäudes bevorzugt werden. Diese Funktion bildet das Interesse an einem Standort basierend auf der vorliegenden Steigung ab.
- **Water:** Siedlungsbildung findet häufig in der Nähe einer Wasserquelle statt. Diese Funktion modelliert dieses Zusammenhang unter Berücksichtigung von Minimal- und Maximaldistanzen zum Gewässer.
- **Fortification:** In Kriegszeiten bringt der Häuserbau in der Nähe von Verteidigungsanlagen den Siedlern einen Vorteil. Diese Funktion ermöglicht die berücksichtigung dieses Zusammenhangs.
- **Geographical domination:** Häuser in Siedlungen werden häufig an erhöhten Standorten als soziales Statussymbol oder als Vorteil bei der Verteidigung errichtet. Außerdem werden sogenannte Points-of-Interest (POI) (hier Kirchen und Befestigungen) oft erhöht errichtet, damit diese schon von weitem erblickt werden können. Diese Funktion bevorteilt einen erhöhten Standort beim Häuserbau umso mehr, je Näher dieser Standort an einem POI liegt.

Anschließend erfolgt der Anschluss der berechneten Gebäudestandorte an ein Straßennetz. Dafür ist es notwendig, eine Menge von Verbindungspunkten zwischen dem eingeschränkten Gebiet zur Siedlungsgenerierung und der Außenwelt zu definieren. Daraufhin wird jedes neue Gebäude mithilfe eines Shortest-Path-Algorithmus mit einem dieser Verbindungspunkte durch eine Straße verknüpft. Die Kosten des Shortest-Path-Algorithmus pro zurückgelegter Kante werden dabei über folgende Variablen bestimmt:

- Distanz zum nächsten Mesh-Knoten

**Abbildung 6.2:** Straßengenerierung ohne Straßen-Wiederverwendung (links), ohne Straßen-Rückführung (mittig), mit beiden Techniken (rechts) Quelle: [11]

- Steigung im Gelände
- Grad der Krümmung
- Vorhandenes Gewässer
- Vorhandenes Gebäude
- Vorhandener Straßenabschnitt

In der realen Welt ist es wichtig, starke Steigungen und Kurven im Straßenverlauf sowie Gewässer und Gebäude beim Straßenbau zu vermeiden. Daher werden die Kosten für die Generierung von Straßenabschnitten mit diesen Eigenschaften entsprechend erhöht. Außerdem werden bestehende Straßenabschnitte in der Realität oft wiederverwendet, anstatt zusätzliche parallele Straßen anzulegen, um die Zusammenführung von Straßenabschnitten zu fördern. Die Kosten für Abschnitte des Shortest-Path-Algorithmus, die bestehende Straßen wiederverwenden, werden daher verringert (siehe Abbildung 6.2).

Zusätzlich wurde eine Methode entwickelt, um Straßen, die in einer Sackgasse enden, über einen anderen Weg ins Straßennetz zurückzuführen (siehe Abbildung 6.2). Dies dient dazu, die ausschließliche Entstehung von Sackgassen zu verhindern. Dafür wird vom Endpunkt der Sackgasse aus versucht, die Straße vorwärts fortzusetzen. Dabei wird der nächstliegende existierende Straßenabschnitt in einem kegelförmigen Gebiet vor der Sackgasse gesucht. An diesen wird der Endpunkt der Sackgasse dann mithilfe des regulären Algorithmus zur Straßengenerierung angeschlossen.

Nach der Generierung von Gebäuden und Straßen erfolgt eine Unterteilung dieses Grundgerüsts in Parzellen. Eine Parzelle repräsentiert das Grundstück, auf dem ein Gebäude platziert ist, und wird beispeilsweise zur Berechnung von Gärten, Zäunen und Feldern verwendet. Abschließend wird ein Algorithmus zur prozeduralen Generierung dreidimensionaler Geometrie für die zuvor generierte Siedlungsaufteilung vorgestellt [11].

Die Berechnungszeiten des Ansatzes können variieren. Für die Generierung des Grundgerüsts einer Siedlung haben Emilien et al. [11] etwa 4 bis 7 Minuten gemessen, während die Unterteilung des Grundgerüsts in Parzellen etwa 7 bis 13 Minuten in Anspruch nimmt. Die prozedurale Generierung der Geometrie für die Siedlungsaufteilung nimmt mit einer Berechnungsdauer von 20 bis 30 Sekunden vergleichsweise wenig Zeit in Anspruch. Im besten Fall konnte eine Gesamtlaufzeit von 11 Minuten und 20 Sekunden für die Generierung einer vollständigen Siedlung gemessen werden [11].

## 6.2 Konzeption des Siedlungsgenerators

Im ersten Abschnitt dieses Kapitels werden die Anforderungen an den Siedlungsgenerator definiert. Auf deren Grundlage wird dann im Anschluss ein mögliches Konzept für den Prototyp diskutiert und erstellt.

### 6.2.1 Zielstellung

Im Folgenden werden die Anforderungen an den Siedlungsgenerator mithilfe der in Abschnitt 1.3 definierten Ziele des Projekts spezifiziert. Die Aufgabe des Siedlungsgenerators ist es, Standorte von Siedlungsstrukturen im bestehenden Terrain (siehe Abschnitt 2.1) zu berechnen und die Geometrie dieser Siedlungsstrukturen zu erzeugen. Zusätzlich sollen Straßen sowohl im Siedlungsgebiet als auch außerorts umgesetzt werden. Bei der Generierung der Siedlungen werden folgende Kriterien berücksichtigt:

1. **Realismus:** Die realistische Anordnung und Darstellung von Siedlungsmodellen in dreidimensionalen, offenen Welten kann einen signifikanten Beitrag zur Immersion des Nutzers leisten. Das Ziel ist es, bekannte Strukturen aus realen Siedlungen zu modellieren, um die Welt um eine für den Betrachter glaubwürdige komponente zu erweitern.

2. **Flexibilität:** Das Projekt soll für ein breites Spektrum an Anwendungen einsetzbar gemacht werden. Zu diesem Zweck ist es notwendig, dass der Siedlungsgenerator Funktionen unterstützt, die dem Nutzer eine individuelle Anpassung der Siedlungsstrukturen auf seine Bedürfnisse ermöglichen. Dabei befindt sich besonders die Distribution von Gebäuden und Straßen im Fokus.

3. **Performance:** Der Prozess zur Generierung von Siedlungen kann sehr berechnungsintensiv ausfallen (siehe Abschnitt 6.1.2). Um den Anforderungen dieses

Projekts an die Performance zu erfüllen, muss der Leistungsaufwand des Algorithmus möglichst gering gehalten werden, sodass für den Nutzer eine möglichst flüssige Verwendung des Editors garantiert werden kann. Zusätzlich sollte beachtet werden, dass die generierte Geometrie der Siedlungsstrukturen ebenfalls die Performance beschränken kann. Dieser Effekt ist von der Siedlungsgröße und der Komplexität der verwendeten Modelle abhängig. Hier muss gegebenenfalls optimiert werden.

4. **Benutzbarkeit:** Die Implementierung einer intuitiven Benutzerschnittstelle ist kein zentraler Bestandteil dieser Arbeit. Es genügt deshalb, wenn der Siedlungsgenerator sein volles Potenzial bei der Verwendung durch einen Nutzer entfaltet, der sowohl Erfahrung mit der Unity Game Engine besitzt, als auch in das System eingewiesen wurde. Von größerer Relevanz ist, die Arbeitsabläufe des Nutzers zur Generierung von Siedlungen möglichst gering zu halten. Außerdem gilt es, Programmfehler bei der Bedienung des Siedlungsgenerators so gut es geht zu vermeiden. Dazu zählen beispielsweise Parameter, die nicht ihren angegebenen Zweck erfüllen und fehlplatzierte Modelle.

### 6.2.2 Diskussion und Konzept

Im folgenden Abschnitt werden mögliche Ansätze zur Generierung von Siedlungen diskutiert. Auf dieser Grundlage findet die begründete Auswahl eines Ansatzes für den weiteren Verlauf der Arbeit statt. Danach findet die Konzeption von Methoden und Parametern statt, die zur Umsetzung des Ansatzes benötigt werden.

#### Diskussion von Ansätzen zur Siedlungsgenerierung

In diesem Abschnitt werden die Vorteile und Nachteile der Siedlungsgenerierung durch Inferenz sowie der prozeduralen Siedlungsgenerierung unter Berücksichtigung der gesetzten Anforderungen (siehe Abschnitt 6.2.1) diskutiert.

**Generierung mithilfe von Inferenz:** Die Generierung von Siedlungskarten durch Inferenz aus realen Ortschaftsdaten wie in der Arbeit von Vanegas et al. [51] stellt eine Möglichkeit dar, die Ziele dieses Projekts (siehe Abschnitt 6.1.1) umzusetzen. Allerdings muss dieser Ansatz noch um einen Algorithmus erweitert werden, der die Platzierung der Siedlungsgeometrie auf Basis der generierten Karten ermöglicht, um die Anforderungen dieses Projekts zu erfüllen. Ein Vorteil der Inferenz von Siedlungskarten liegt darin, dass die Siedlungsstrukturen, wie zum Beispiel die Grundstücksverteilung, Straßenverläufe und Standorte von Kirchen, das Abbild realer Ortschaften sind. Dies eröffnet das Potenzial für einen hohen Grad an Realismus in der generierten Umgebung.

Die Flexibilität dieses Ansatzes zur Generierung von Siedlungen hängt von verschiedenen Faktoren ab. Zum einen wird die sie maßgeblich durch die Menge der verfügbaren Siedlungsdaten bestimmt, da diese die Möglichkeiten beeinflussen, die dem Nutzer zur Erstellung einer Siedlung zur Verfügung stehen. Wenn der Nutzer beispielsweise eine Siedlung in einem von ihm festgelegten Terrain generieren möchte, dann benötigt er die Daten eines realen Siedlungs-Layouts, das sich in dieses Gelände einfügt. Wenn solche Daten nicht verfügbar sind, schränkt das die Flexibilität des Nutzers ein. Ein weiteres Problem besteht darin, wie der Nutzer bestimmte Aspekte der Siedlungsgenerierung steuern kann. Es ist unmöglich, für jede denkbare Anordnung von Siedlungsstrukturen einen realen Datensatz bereitzustellen. Eine mögliche Lösung könnte darin bestehen, die von Vanegas et al. [51] vorgestellte Siedlungssimulation so zu parametrisieren, dass der Nutzer mehr Kontrolle über den Generierungsprozess hat. Dies würde es dem Nutzer ermöglichen, bestimmte Parameter anzupassen, um die gewünschten Ergebnisse zu erzielen und die Flexibilität des Ansatzes zu erhöhen.

Die Performance dieses Ansatzes zur Generierung von Siedlungen basiert auf verschiedenen Faktoren und hängt stark von der spezifischen Implementierung ab, die für den Algorithmus entwickelt wird. Da der genaue Ablauf des Algorithmus noch nicht ausreichend spezifiziert ist und in der Arbeit von Vanegas et al. [51] keine Laufzeitmessungen durchgeführt wurden, ist es schwer, eine allgemeine Aussage zur Performance zu treffen. Es ist jedoch anzumerken, dass das Einlesen und die Umwandlung von realen Siedlungsdaten in Siedlungskarten wahrscheinlich den geringsten Teil der Berechnungen ausmachen wird. Der Hauptteil der Berechnungen wird voraussichtlich bei der Simulation zukünftiger Stadien der Siedlung stattfinden. Die Performance wird daher maßgeblich von der benötigten Anzahl der Simulationsschritte und der Komplexität dieser Schritte bestimmt. Die Effizienz der Implementierung und die Wahl geeigneter Algorithmen zur Simulation werden entscheidend sein, um akzeptable Berechnungszeiten sicherzustellen.

Die Benutzbarkeit dieses Ansatzes zur Generierung von Siedlungen hängt stark von den vom Benutzer benötigten Eingaben ab. Diese Eingaben setzen sich aus realen Siedlungsdaten und möglicherweise vorgefertigten Modellen von Siedlungsstrukturen zusammen, die der Benutzer selbst beschaffen muss. Wenn die Siedlungsmodelle prozedural generiert werden können, dann müssen möglicherweise nur Parameter für diesen Prozess angepasst werden. Die Eingabe von realen Siedlungsdaten kann jedoch ein Problem darstellen, da nicht vom Nutzer erwartet werden kann, dass er diese Daten eigenständig beschafft. Eine mögliche Lösung für dieses Problem wäre es, dem Benutzer eine Datenmenge oder eine Schnittstelle zu einer Datenbank bereitzustellen, aus der er passende Daten auswählen kann. Potenzielle Datenquellen könnten laut Vanegas et al. [51] Google Maps [18], MapQuest [32] und ArcGis [12] sein. Zusammenfassend ist es möglich, die Benutzbarkeit dieses Ansatzes im Rahmen der gesetzten Anforderungen zu gestalten. Der Prozess zur Auswahl realer

Daten durch den Benutzer kann jedoch, abhängig von der Menge aller Daten und dem erforderlichen Input, sehr aufwändig werden. Daher wäre es wichtig, eine benutzerfreundliche Schnittstelle mit Filtermethoden für die Auswahl und Verwendung dieser Daten bereitzustellen.

**Prozedurale Generierung:** Ein weiterer Ansatz zur Erstellung von Siedlungsgebieten besteht in der prozeduralen Generierung von Siedlungskarten nach Emilien et al. [11], auf deren Grundlage im Anschluss Siedlungsmodelle in der Landschaft platziert werden können (siehe Abschnitt 6.1.2). Bei diesem Ansatz wird versucht, bekannte Siedlungsstrukturen aus der realen Welt nachzubilden, um ein möglichst realistisches Erlebnis für den Benutzer zu schaffen. Im Gegensatz zum Ansatz von Vanegas et al. [51] (siehe Abschnitt 6.1.1) beinhaltet dieser Ansatz keine Verarbeitung von realen Siedlungsdaten, wodurch das Potenzial für einen hohen Grad an Realismus hier geringer ausfällt. Dennoch ist es möglich, auch durch prozedurale Generierung eine ausreichende Ergebnisqualität zu erreichen, um die Anforderungen dieses Projekts angemessen zu erfüllen.

Ein wesentlicher Vorteil dieses Ansatzes liegt in den Möglichkeiten zur Individualisierung der erstellten Siedlungen, die dem Benutzer zur Verfügung stehen. Der Benutzer ist nicht auf ein vorgefertigtes Siedlungs-Layout beschränkt, sondern kann mithilfe benutzerdefinierter Parameter eigene Ideen umsetzen. Im Vergleich zum Ansatz der Siedlungsinferenz, bei dem nicht jedes Gelände zu den ausgewählten Siedlungsdaten passt, wird der Benutzer unter Verwendung dieser Methode nicht bei der Erstellung der anderen Komponenten seiner Welt beeinträchtigt. Die Flexibilität dieses Ansatzes ist jedoch nicht unbegrenzt. Wie bei anderen prozeduralen Verfahren wird es dem Benutzer nicht möglich sein, die Position jedes Pixels in der Siedlungskarte und jedes Siedlungsmodells in der Welt genau zu bestimmen. Wenn ein solcher Grad an Flexibilität erforderlich ist, muss der Benutzer auf manuelle Modellierung zurückgreifen.

Die in der Arbeit von Emilien et al. [11] gemessenen Laufzeiten für den Ansatz der prozeduralen Siedlungsgenerierung sind mit über zehn Minuten nicht für ein Echtzeitsystem geeignet. Daher ist es ausgeschlossen, die Methode von Emilien et al. [11] in dieser Form für dieses Projekt zu übernehmen. Allerdings übersteigt der Realismus der implementierten Methode die Anforderungen dieses Projekts. Daher könnte eine Vereinfachung der präsentierten Berechnungen eine Möglichkeit im Rahmen dieser Arbeit darstellen. Dies würde zu einem verringerten Grad an Realismus führen, aber gleichzeitig die Laufzeit verbessern, wodurch die Einhaltung der Anforderungen aus Abschnitt 6.2.1 möglich werden könnte.

Der Arbeitsaufwand für den Nutzer bei diesem Ansatz ergibt sich aus der Anzahl und Komplexität der verfügbaren Parameter, die individuell angepasst werden müssen, um eine benutzerdefinierte Siedlung zu generieren. Darüber hinaus muss der Nutzer

Modelle beschaffen, die zur Instanziierung der Siedlungsgeometrie verwendet werden, falls diese nicht prozedural generiert werden. Dieser Aufwand für den Nutzer ist im Vergleich zur Inferenz von Siedlungskarten geringer, da die Beschaffung und Auswahl passender realer Siedlungsdaten entfällt. Somit können die Anforderungen an die Benutzbarkeit bei diesem Ansatz eingehalten werden.

### Zusammenfassung der Ansätze zur Siedlungsgenerierung

Unter Betrachtung der genannten Vorteile und Nachteile wird klar, dass die Generierung von Siedlungskarten durch Inferenz, trotz des Potenzials für einen hohen Grad an Realismus, nicht alle Anforderungen an das Projekt zufriedenstellend erfüllen kann. Das größte Problem liegt in der benutzerdefinierten Erstellung von Siedlungen. Hierbei existieren insgesamt zu wenige umsetzbare Anpassungsmöglichkeiten, mithilfe derer die realen Siedlungsdaten individualisiert werden können. Darüber hinaus besteht die Möglichkeit, dass die verwendeten Realdaten im Widerspruch zu den bereits existierenden Komponenten der Welt, wie dem Terrain, den Biomen oder der Vegetation, stehen könnten. Dies könnte die Integration der Siedlungskarten erschweren.

Stattdessen wird in den folgenden Abschnitten das Konzept der prozeduralen Siedlungsgenerierung umgesetzt. Dieser Ansatz bietet viele Möglichkeiten zur Individualisierung bei einem zufriedenstellenden Grad an Realismus und Benutzbarkeit. Allerdings sollte die Performance des Algorithmus zur Generierung von Siedlungen berücksichtigt werden. Diese könnte, abhängig von der Komplexität der generierten Siedlungen, unzureichend für dieses Projekt ausfallen, wie die Messungen zeigen [11]. Hierbei muss daher zwischen reduziertem Realismus zugunsten einer verbesserten Laufzeit abgewogen werden.

### Methoden und Parameter zur Siedlungsgenerierung

Die Positionen sämtlicher zu platzierenden Siedlungsstrukturen werden im Folgenden mithilfe einer Siedlungskarte ermittelt. Diese Siedlungskarte wird nach dem gewählten Ansatz (siehe Abschnitt 6.2.2) auf Grundlage des vorliegenden Terrains und einiger benutzerdefinierter Parameter prozedural generiert. Der Aufbau der Siedlungen wird in diesem Prototyp relativ einfach gehalten, um einen geringeren Berechnungsaufwand im Vergleich zur Methode von Emilien et al. [11] zu erreichen. Insgesamt bestehen die Siedlungen aus Kirchen, Befestigungen, Häusern und Straßen. Die Parzellenermittlung, die in der Arbeit von Emilien et al. [11] die höchste Berechnungszeit benötigte, wird aus den genannten Performance-Gründen ausgelassen. Dadurch entfallen Grundstücke mit Gärten, Zäunen und Feldern in der finalen Darstellung.

**Lage der Siedlung:** Der erste Schritt bei der Generierung einer Siedlung ist die Ermittlung einer passenden geografischen Lage im Terrain. Um diese Berechnungen durchführen zu können, müssen folgende Parameter vom Nutzer an den Algorithmus übergeben werden:

- VillageRadius (Radius des Siedlungsbereichs)
- ValidSteepness (Intervall an Steigungswerten, die bebaubar sind)
- ValidWaterDistance (Intervall an möglichen Distanzen zum Wasser)
- MinBuildableShare (Anteil bebaubarer Fläche am Siedlungsbereich)

Zu Beginn wird ein sogenannter Siedlungs-Seed an einer zufälligen Position im Terrain ermittelt. Sowohl der Steigungswert als auch die Distanz zum nächsten Gewässer an diesen Seed-Koordinaten müssen dabei innerhalb des *ValidSteepness*-Intervalls beziehungsweise des *ValidWaterDistance*-Intervalls liegen. Sind diese Bedingungen erfüllt, wird anschließend ein annähernd kreisförmiges Siedlungsgebiet mit dem Radius *VillageRadius* und dem Siedlungs-Seed als Ursprung berechnet. Sämtliche Mesh-Knoten, die bei der Ermittlung dieses Siedlungsgebiets nicht die durch *ValidSteepness* und *ValidWaterDistance* vorgegebenen Bedingungen erfüllen, werden ausgelassen.

Zuletzt findet noch eine Überprüfung statt, wie viel der potenziell möglichen Siedlungsfläche tatsächlich zum Siedlungsgebiet hinzugefügt werden konnte. Ist dieser Anteil geringer als der *MinBuildableShare*-Parameter vorgibt, dann muss das Siedlungsgebiet an einem passenderen Standort neu berechnet werden. Dies kann beispielsweise passieren, wenn der Großteil des ausgewählten Siedlungsgebiets aus steilen Abhängen besteht, die in der Realität nicht bebaut werden könnten. Ein mögliches Ergebnis dieses Berechnungsabschnitts ist in Abbildung 6.4 dargestellt.

**Siedlungsstrukturen:** Im Anschluss an die Berechnung der Grundfläche der Siedlung erfolgt die Generierung sämtlicher Strukturen innerhalb des Siedlungsgebiets. Für den ersten Teil dieses Berechnungsabschnitt werden folgende benutzerdefinierte Parameter nach Emilien et al. [11] verwendet:

- EntryCount (Anzahl der Straßenzugänge zum Siedlungsgebiet)
- MinEntryDistance (Mindestabstand zwischen zwei Straßenzugängen)
- ChurchCount (Anzahl der benötigten Kirchen)
- MinChurchDistance (Mindestabstand zwischen zwei Kirchen)
- FortCount (Anzahl der benötigten Befestigungen)
- MinFortDistance (Mindestabstand zwischen zwei Befestigungen)

- MinHeightPOI (Mindesthöhe sämtlicher Points-of-Interest

Um mit der Generierung der Strukturen beginnen zu können, müssen zuerst so viele Straßenzugänge zum Siedlungsgebiet ermittelt, wie der *VillageEntryCount*-Parameter spezifiziert. Straßenzugänge bilden eine Schnittstelle vom Inneren der Siedlung zur Außenwelt und erfüllen damit den gleichen Zweck wie die Verbindungspunkte in der Arbeit von Emilien et al. [11]. Die Zugänge werden zufällig über den äußeren Rand des Siedlungsgebiets verteilt (siehe Abbildung 6.5), wobei alle Straßenzugänge den im *MinEntryDistance*-Parameter spezifizierten Abstand zueinander einhalten müssen.

Darauf folgt die Berechnung der Standorte sämtlicher Kirchen und deren Verbindung mit den zuvor berechneten Siedlungszugängen über Straßen. Hierfür werden alle Mesh-Knoten des Siedlungsgebiets bestimmt, die der geforderten Mindesthöhe entsprechen, die durch den *MinHeightPOI*-Parameter festgelegt ist. Anschließend werden zufällige Standorte aus dieser Menge an hoch gelegenen Knoten für die Kirchen ausgewählt, wobei der Mindestabstand zwischen ihnen dem im *MinChurchDistance*-Parameter festgelegten Wert entsprechen muss. Die Anzahl der zu generierenden Kirchenstandorte wird durch den *ChurchCount*-Parameter vorgegeben. Um diese Standorte über Straßen mit den Siedlungszugängen zu verbinden ein Shortest-Path-Algorithmus verwendet. Die Kosten dieses Algorithmus erhöhen sich mit größerer Distanz und Steigung und werden durch die Verwendung bestehender Straßenabschnitte verringert. Dadurch wird steiles Terrain beim platzieren der Straßenabschnitte vermieden und existierende Straßenabschnitte wiederverwendet.

Die Standorte der Befestigungsanlagen und deren Straßenverbindungen werden auf die gleiche Weise berechnet wie die Kirchen. Hierbei werden jedoch die Parameter *FortCount* und *MinFortDistance* anstelle der Kirchenparameter verwendet. Anschließend werden sämtliche Straßenzugänge, die noch nicht mit dem Straßennetz verbunden sind, an dieses angeschlossen. Ein möglicher Zwischenstand der Berechnungen ist in Abbildung 6.5 dargestellt.

Im Anschluss an die Berechnung der Kirchen und Befestigungen erfolgt die Generierung von Wohnhäusern, um die Siedlung zu füllen. Um diese Berechnungen durchführen zu können, müssen folgende Parameter vom Nutzer spezifiziert werden:

- HouseCount (Anzahl der benötigten Wohnhäuser)
- Sociability (Abstand zu anderen Wohnhäusern)
- Worship (Abstand zur nächsten Kirche)
- Accessibility (Abstand zum nächsten Straßenabschnitt)
- Fortification (Abstand zur nächsten Befestigung)

- GeoDomination (Interesse an Höhenlage)

Zur Generierung eines einzelnen Hauses wird zuerst ein zufälliger Standort im Siedlungsgebiet ausgewählt, an dem noch keine andere Siedlungsstruktur platziert ist. Die Eignung dieses Standorts $(x, y)$ für die Platzierung eines Hauses wird mithilfe des ganzheitlichen Interest-Wertes $I_{ges}(x, y)$ bestimmt. Um diesen Wert zu berechnen, wird zunächst aus jedem Abstandsparameter (*Sociability*, *Worship*, *Accessibility* und *Fortification*) ein parameterspezifischer Interest-Wert ermittelt. Die Abstandsparameter definieren jeweils einen Minimalwert $D_{min}$, einen Optimalwert $D_{opt}$ und einen Maximalwert $D_{max}$, wobei folgende Ungleichung zu jedem Zeitpunkt erfüllt sein muss:

$$0 \leq D_{min} \leq D_{opt} \leq D_{max}$$

Diese Variablen beziehen sich auf den Abstand $D_p(x, y)$ des betrachteten Standorts zum nächsten Element der Siedlungsstruktur, das dem jeweiligen Parameter entspricht. Mithilfe dieser Variablen wird für jeden Abstandsparameter $p$ eine individuelle Dreiecksfunktion $I_p$ beschrieben, aus der die parameterspezifischen Interest-Werte $I_p(D_p(x, y))$ berechnet werden. Der Aufbau dieser Funktion ist in Abbildung 6.3 dargestellt.

Um die geografische Dominanz (siehe Abschnitt 6.1.2), die durch den *GeoDomination*-Parameter festgelegt wird, in den ganzheitlichen Interest-Wert $I_{ges}(x, y)$ einzubeziehen, wird ein Modifizierungswert $m$ gebildet. Diese Variable hat initial den Wert einer Konstanten aus dem Intervall $[0, 1]$ inne. Anschließend wird $m$ aufgrund von hoher Distanz zum nächsten Point-of-Interest oder einer geringeren Höhenlage relativ zum nächsten Point-of-Interest verringert, wobei 0 den Minimalwert darstellt. Der Modifizierungswert $m$ wird zum Durchschnitt der parameterspezifischen Interest-Werte $I_p(D_p(x, y))$ addiert, um geografische Dominanz von bei der Wohnhausplatzierung zu belohnen. Dadurch ergibt sich die folgende Formel zur Berechnung des ganzheitlichen Interest-Werts:

$$I_{ges}(x, y) = \frac{\sum_{p=1}^{4} I_p(D_p(x,y))}{4} + m$$

Liegt der Interest-Wert über einem festgelegten Schwellwert, wird auf der Siedlungskarte an der betrachteten Position $(x, y)$ eine Markierung für ein Wohnhaus platziert, welches dann mithilfe des zuvor eingeführten Algorithmus zur Straßenberechnung an das bestehende Straßennetz angeschlossen wird. Die Anzahl der zu platzierenden Wohnhäuser legt der Parameter *HouseCount* fest. Der beschriebene Algorithmus zur Berechnung der Wohnhäuser wird so oft durchgeführt, bis die benötigte Anzahl an

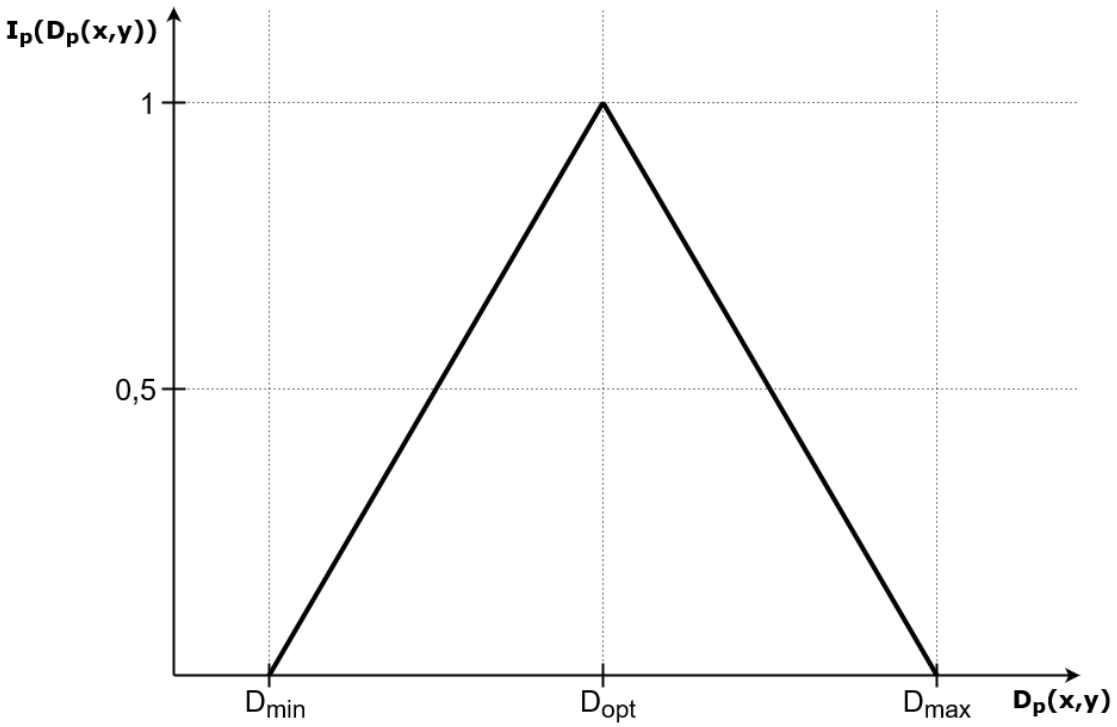



**Abbildung 6.3:** Funktion zur Berechnung der Interest-Werte
Quelle: Eigene Darstellung

Wohnhäusern auf der Siedlungskarte erreicht ist. Ein Beispiel für eine Siedlungskarte, wie sie nach den beschriebenen Berechnungen aussehen könnte, ist in Abbildung 6.6 dargestellt.

**Überlandstraßen** Abschließend muss die Verknüpfung der Siedlung mit dem Rest der Welt erfolgen. Denkbar wären dabei Straßenverbindungen zu anderen Siedlungen oder abgelegenen Siedlungsstrukturen wie beispielsweise Klöstern. Alle Methoden, die für solche Funktionalitäten benötigt werden, sind bereits im Algorithmus zur Berechnung von Siedlungskarten in dieser Arbeit konzipiert worden. Damit sollte es zukünftigen Arbeiten leicht möglich sein, die Weltengenerierung um zusätzliche Siedlungen zu erweitern. Um den Rahmen dieser Arbeit nicht zu sprengen und die Laufzeit den Anforderungen entsprechend zu gestalten, wird in dieser Arbeit lediglich eine einzige Siedlung generiert, die anschließend über Straßen mit den Kartenrändern verbunden wird.

Zu diesem Zweck wird zu jedem Straßenzugang der Siedlung ein zugehöriger Straßenzugang am Kartenrand ermittelt. Diese Kartenzugänge können, ähnlich wie die Siedlungszugänge, als Schnittstelle zwischen zwei mit dem Prototyp generierten Welten fungieren. Zukünftige Versionen des Weltengenerators könnten diese Schnittstellen nutzen, um mehrere Weltabschnitte miteinander über Straßen zu verbinden. Die Positionen der Kartenzugänge werden zufällig im Randbereich der Siedlungskarte platziert. Dabei wird darauf geachtet, dass sämtliche Kartenzugänge möglichst gleichmäßig auf alle vier Kartenränder aufgeteilt werden und in der näheren Umgebung eines Kartenzugangs kein Gelände vorhanden ist, das zu steil für die Platzierung von Straßen ist. Dadurch wird sichergestellt, dass die Kartenzugänge mit den Siedlungszugängen verbunden werden können. Ist die Verbindung der Siedlung mit den Kartenrändern abgeschlossen, dann ist die Siedlungsgenerierung vollständig. Aus den Straßenverläufen der vollständigen Siedlungskarte wird zuletzt noch die Street-Map generiert, die vom Texturierungssystem (siehe Abschnitt 4.3) als Alpha-Map zur Darstellung der Straßen benötigt wird.

Eine vollständig generierte Siedlungskarte kann in Abbildung 6.7 betrachtet werden. Beim Vergleich mit der zur Berechnung verwendeten SteepnessMap in Abbildung 6.8 fallen die Straßenverläufe auf, die den steilen Regionen der Karte ausweichen. Außerdem ist erkennbar, dass die Siedlung in einer Region realisiert wurde, die flach genug ist, um Siedlungsstrukturen darauf realistisch platzieren zu können.

### Auswahl der Siedlungsmodelle

Die Bestimmung der Siedlungsmodelle, die nach den Vorgaben der Siedlungskarte platziert werden sollen, kann entweder durch die prozedurale Synthese von Modellen

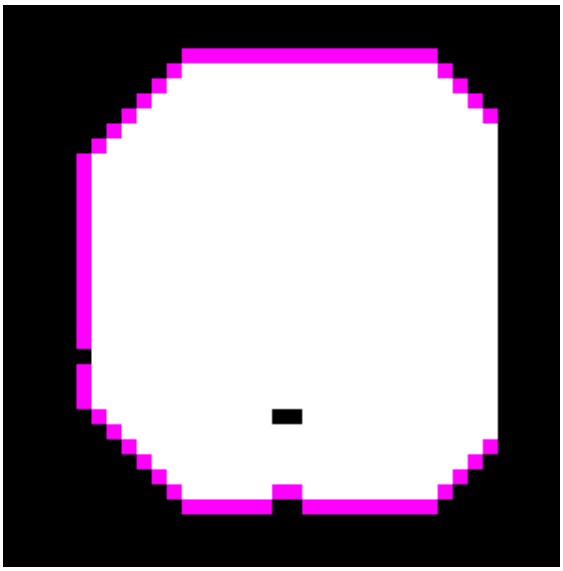


**Abbildung 6.4:**
Bestimmung Siedlungsgebiet
Quelle: Eigene Darstellung

**Abbildung 6.5:**
Bestimmung POI
Quelle: Eigene Darstellung

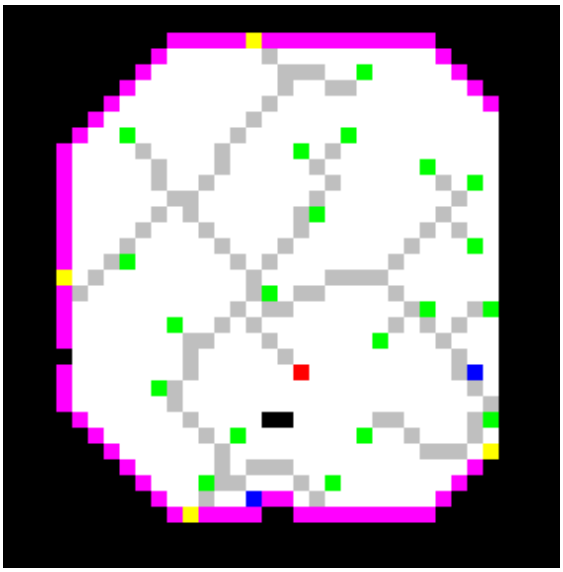


**Abbildung 6.6:**
Bestimmung Wohnhäuser
Quelle: Eigene Darstellung

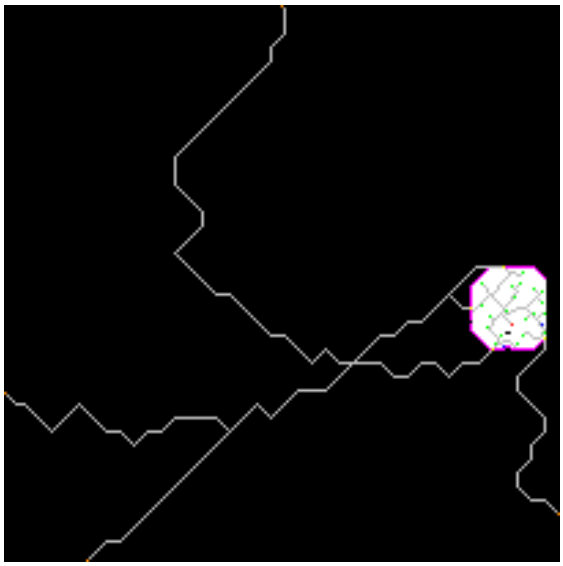


**Abbildung 6.7:**
Vollständige Siedlungskarte
Quelle: Eigene Darstellung

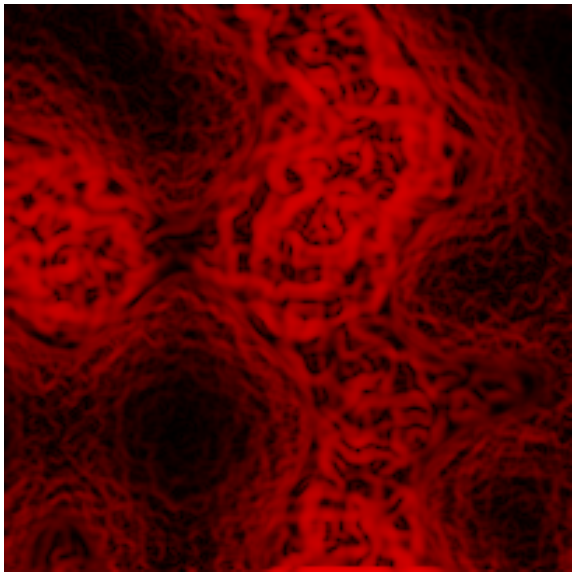


**Abbildung 6.8:** SteepnessMap zur Siedlungsgenerierung
Quelle: Eigene Darstellung

oder durch die benutzerdefinierte Auswahl vorgefertigter Modelle erfolgen. Diese Ansätze wurden bereits in Abschnitt 5.2.2 bei der Vegetationsgenerierung untersucht, und die dort herausgearbeiteten Argumente können auch in diesem Fall angewendet werden. Die prozedurale Synthese von Siedlungsmodellen wurde außerdem in der Arbeit von Emilien et al. [11] umgesetzt.

Bei der Betrachtung der Untersuchungen in Abschnitt 5.2.2 und der Arbeit von Emilien et al. [11] wird klar, dass die prozedurale Synthese von Siedlungsmodellen das Potenzial für eine beinahe beliebig große Anzahl unterschiedlicher Gebäude bietet, die an Umweltfaktoren wie das Terrain oder das Klima angepasst werden können. Im Vergleich dazu trägt der Nutzer beim zweiten Ansatz, der individuellen Auswahl vorgefertigter Modelle, eine größere Eigenverantwortung bei der Umsetzung einer realistischen Siedlung. Der Realismus fällt bei der prozeduralen Synthese aus diesem Grund tendenziell höher aus.

Abgesehen davon gilt hier, wie auch schon in Abschnitt 5.2.2 für die Pflanzenmodelle, dass der Nutzer durch die Auswahl vorgefertigter Modelle eine größere kreative Freiheit bei der Erstellung der Siedlung erhält. Ebenfalls kann das Performance-Argument aus Abschnitt 5.2.2 herangezogen werden. Dies besagt, dass die prozedurale Synthese einen höheren Berechnungsaufwand zur Laufzeit benötigt, während die Modellauswahl mehr Festplattenspeicher belegt. Da der Prototyp möglichst in Echtzeit arbeiten soll und Festplattenspeicher in den letzten Jahren relativ günstig geworden ist, besitzt auch hier die Modellauswahl einen Vorteil gegenüber der Synthese.

Um einen Algorithmus zu konstruieren, der die Anforderungen des Projekts an Flexibilität und Performance erfüllt, wird in dieser Version des Weltengenerators der Ansatz der Auswahl von Siedlungsmodellen durch den Nutzer verwendet. Im Prototypen werden für die Darstellung der Ergebnisse dieses Projekts eigenständig angefertigte Modelle für die Kirchen, Befestigungen und Wohnhäuser verwendet. Diese dienen lediglich der Veranschaulichung der Berechnungsergebnisse und können in zukünftigen Versionen gegen realistischere Modelle ausgetauscht werden.

## 6.3 Implementierung des Siedlungsgenerators

Der folgende Abschnitt beschreibt die Umsetzung des Konzepts aus Abschnitt **??** Zunächst wird die Erstellung der Siedlungskarte erläutert. Im Anschluss wird die Instanziierung der Siedlungsmodelle erläutert. Für die Umsetzung dieses Projektabschnitts wurde die Programmiersprache C# unter Verwendung der Pakete System.Collections.Generic, System.Linq, UnityEngine und UnityEditor eingesetzt. Die gezeigten Modelle wurden eigenständig erstellt. Ein Überblick über den Algorithmus zur Siedlunsggenerierung ist in Abbildung 6.9 dargestellt.

### 6.3.1 Generierung der Siedlungskarte

-der algorithmus zur generierung von siedlungskarten benötigt eine reihe an parametern als eingabe, um sämtliche berechnungen geplanten umsetzen zu können -das konzept des algorithmus ist zusammen mit den zugrundeliegenden parametern in **??** genauer erklärt -zuerst berechnet der algorithmus das basis“=gebiet der siedlung im terrain -im anschluss daran werden die standorte von siedlungstrukturen innerhalb des siedlungsgebietes und überlandstraßen außerhalb des siedlungsgebietes anhand der übrigen parameter prozedural ermittelt und in die siedlungskarte eingetragen

#### Generierung des Siedlungsgebiets

-um einen passenden standort für das siedlungsgebiet zu ermitteln, wird zuerst ein siedlungs“=seed ermittelt und in die siedlungskarte eingetragen, der die benutzerdefinierten anforderungen an den grad der steigung sowie die distanz zum nächsten gewässer erfüllt -ist ein solcher startpunkt gefunden, werden im anschluss sämtliche nachbar“=pixel des siedlungs“=seeds zum potenziellen siedlungsgebiet hinzugefügt, falls sie die anforderungen ebenfalls erfüllen -dieser prozess setzt sich in einer anzahl von iterationen fort, die durch den *villageRadius*-parameter vorgegeben ist -dabei werden in den ersten 50% der iterationen alle pixel in achter“=nachbarschaft

```
GenerateVillage() {
    villageMap = GenerateVillageMap();
    foreach(pixel (x,y) in villageMap) {
        switch(villageMap[x,y]) {
            case HousePixel:
                prefab = housePrefab;
            case ChurchPixel:
                prefab = churchPrefab;
            case FortPixel:
                prefab = fortPrefab;
            default:
                continue;
        }

        prefab.rotateTowards(closestStreetPixel);
        prefab.scale(modelScaling);
        worldPosition = getWorldPosition(x,y);
        buildingArray = prefab.instantiate(worldPosition);
    }
}

GenerateVillageMap() {
    villageMap = initVillageMap();
    villageMap.calcVillageArea();
    villageMap.calcVillageEntries();
    villageMap.calcMapEntries();
    villageMap.calcPOI(Churches);
    villageMap.calcPOI(Forts);
    villageMap.calcHouses();
    villageMap.calcCountryRoads();
    return villageMap;
}
```

**Abbildung 6.9:**
Überblick über die Siedlungsgenerierung in Pseudocode
Quelle: Eigene Darstellung

der bestehenden siedlungspixel betrachtet, wohingegen die folgenden iterationen mit vierer“=nachbarschaftsbetrachtungen durchgeführt werden -dadurch wird für eine gleichmäßigere ausbreitung des siedlungsgebietes in der form eines oktagons gesorgt (siehe **??**) -die pixel der letzten iteration werden gesondert als randbereich des gebietes markiert (siehe abb), für die folgende berechnung der straßenzugänge (siehe code)

-als schnittstellen zwischen den straßen werden sowohl siedlungszugänge als auch kartenzugänge in form von pixel“=markierungen in der siedlungskarte ermittelt -die benötigte anzahl an siedlungszugänge wird dabei zufällig im randbereich des siedlungsgebietes verteilt, wobei der vom nutzer festgelegte mindestabstand eingehalten werden muss (siehe abb, zeile) -die benötigten kartenzugänge werden hingegen gleichmäßig auf die vier randbereiche der karte aufgeteilt (siehe abb, zeile) -dabei werden nur standorte ausgewählt, die einen niedrigen durchschnittlichen steigungswert für ihre nähere umgebung aufweisen, um die chance zu erhöhen, dass straßen zu den jeweiligen kartenzugängen errichtet werden können

### Generierung der Siedlungsstrukturen

-die berechnung der siedlungstrukturen beginnt mit der platzierung von Points“=of“=Interest im siedlungsbereich, zu denen sowohl kirchen als auch befestigungsanlagen gehören (siehe abb) -um standorte in der vom nutzer gewählten höhenlage zu finden, wird zuerst eine menge an pixeln in der siedlungskarte gebildet, deren zugehörige mesh“=knoten oberhalb des durch den *minHeightPOI*“=parameter spezifizierten wertes liegen -für die erstellung eines POI wird jeweil ein zufälliges pixel aus dieser menge zu einer kirche beziehungsweise einer befestigung abgeändert

-den neuen POI wird dann jeweils ein siedlungseingang zugeordnet, über den sie mit dem straßennetz verbunden werden können -Für den erstplatzierten POI wird hierbei der am weitesten entfernte siedlungseingang ausgewählt, wohingegen alle nachfolgenden POI ihrem nächstliegenden zugang zugeordnet werden (siehe abb) -dadurch wird sichergestellt, dass sich die wohnhäuser im folgenden generierungsschritt weitreichender im siedlungsgebiet verteilen -sind alle POI mit zugängen verbunden, dann werden sämtliche Siedlungseingänge, die noch nicht an das bestehende straßennetzwerk angeschlossen sind über eine straße mit dem nächsten straßenabschnitt verbunden

-zur verbindung zweier wegpunkte durch eine straße wird die siedlungskarte als graph mit pixeln als knoten und nachbarschaften als kanten betrachtet -darauf agiert ein dijkstra algorithmus, der die kosten einer kante anhand des steigungswertes des dadurch erreichten knotens berechnet -dadurch wird ein gleichgewicht zwischen relativ kurzen straßen und einem möglichst geringen steigungswiderstand gehalten

```
calcVillageArea(villageMap) {
    possibleVillageSize = calcPossibleVillageSize(villageRadius);
    minVillageSize = possibleVillageSize * minBuildableShare;
    while(villageSize < minVillageSize) {
        villageMap.calcVillageSeed();
        villageMap.growVillageArea();
        villageSize = villageMap.countVillagePixel();
    }
}

calcVillageSeed() {
    while( not(isSteepnessValid && isWaterDistanceValid) ) {
        seed = Random(villageMap);
        isSteepnessValid =
            checkSteepness(seed.steepness, validSteepness);
        isWaterDistanceValid =
            waterDistance(seed.waterDist, validWaterDistance);
    }
    villageMap[seed] = villagePixel;
}

growVillageArea() {
    for(i=0; i<villageRadius; i++) {
        foreach(neighbor of villagePixel) {
            if( checkSteepness(neighbor.steepness, validSteepness) &&
                waterDistance(neighbor.waterDist, validWaterDistance)
                ) {
                villageMap[neighbor] = villagePixel;
            }
        }
    }
}
```

**Abbildung 6.10:**
Berechnung des Siedlungsbereichs in Pseudocode
Quelle: Eigene Darstellung

```
calcVillageEntries() {
    borderPixelSet = getVillageBorderPixel(villageMap);
    while(currentEntryCount < entryCount) {
        entry = RandomElement(borderPixelSet);
        if(distToNextEntry(entry) > minEntryDist) {
            villageMap[entry] = entryPixel;
            currentEntryCount++;
        }
    }
}

calcMapEntries() {
    mapEdgePixel = getMapEdgePixel();
    while(currentEntryCount < entryCount) {
        nextMapEdge = getEdgeWithLeastEntries();
        pixelSet = getPixelOfEdge(nextMapEdge, mapEdgePixel);
        entry = RandomElement(pixelSet);
        entrySteepness = calcLocalAvgSteepness(entry);
        if(checkSteepness(entrySteepness, validSteepness)) {
            villageMap[entry] = entryPixel;
        }
    }
}
```

**Abbildung 6.11:**
Berechnung des Straßenzugänge in Pseudocode
Quelle: Eigene Darstellung

-kanten, deren steigungswerte außerhalb des vom nutzer spezifizierten intervalls fallen, werden von der straßenberechnung ausgeschlossen -die wiederverwendung von straßen wird bei der straßenberechnung ebenfalls mithilfe der kostenfunktion des dijkstra“=algorithmus umgesetzt -falls ein knoten bereits als straße deklariert ist, dann besitzen sämtliche kanten, die zu diesem knoten führen, die kosten 0.

-die ermittlung von wohnhausstandorten erfolgt über einen interest“=wert und ist mithilfe der **??** ersichtlich -der dabei verwendete wert für geografische dominanz (siehe konzept erklärung) ergibt sich aus den beiden variablen *DistanceImpact* und *Heightimpact* -diese verringern den belohnungswert für geografische dominanz bei wachsender distanz oder einer wachsenden höhendifferenz zwischen dem potenziellen wohnhausstandort und dem nächsten POI -berechnet werden die variablen als InverseLerp zwischen vom nutzer festgelegten minimal“= und maximalwert über die distanz beziehungsweise die höhendifferenz

-die letzten zu generierenden siedlungsstrukturen sind die überlandstraßen, die außerhalb des siedlungsgebietes liegen sollen -diese verlaufen in der regel durch weite teile der weltkarte, wodurch sich eine hohe berechnungslast für den algorithmus zur berechnung der kürzesten wege ergeben kann -um diese berechnungslast zu senken, wird jede ermittlung einer überlandstraße in kleinere teilabschnitte aufgeteilt, die in summe leichter für den dijkstra“=algorithmus zu berechnen sind als ein zusammenhängender straßenzug -zu diesem zweck verringert der algorithmus die auflösung der originalen siedlungskarte um einen vom nutzer angegebenen wert und berechnet dort die überlandstraßen mithilfe der eingeführten methode zur ermittlung von straßenverbindungen -die koordinaten der ermittelten straßenabschnitte werden im anschluss in koordinaten der originalen siedlungskarte transformiert und eingetragen als wegpunkte eingetragen -diese wegpunkte werden nun nacheinander verbunden, um die überlandstraßen abzubilden, die vorher in reduzierter auflösung errechnet wurden

### 6.3.2 Instanziierung der Siedlungsmodelle

-iterate villageMap -choose prefab -rotate towards clostest street pixel -scale -instantiate at corresponding world pos in mesh -> put in array of game objects

-generate village map -generate water distance map -calc min village size with params -find village area that is big enough -> max 500 tries: -calc random village seed + checks: is seed alright? -calc village area -> grow village area around seed, first in 8-Neighborhood, then in 4-neighborhood -> leave out pixel that not in steepness-params or in waterDistance-params -> set the most outer ring as village border -calc road entry count with params + coords in villageBorder -> randomly distribute x pixels of type roadEntry into set of villageBorder-Pixels -calc map entries -map entry

```
calcPOI(typeOfPOI) {
    if(typeOfPOI == Church) {
        POICount = churchCount;
        minPOIDistance = churchDistance;
        POIPixel = churchPixel;
    } else {
        POICount = fortCount;
        minPOIDistance = fortDistance;
        POIPixel = fortPixel;
    }

    villagePixelSet = getVillagePixel(villageMap);
    pixelSet = getHighestPixel(villagePixelSet, minHeightPOI);
    while(currentPOICount < POICount) {
        POI = RandomElement(pixelSet);
        if(distToNextPOI(POI) > minPOIDistance) {
            villageMap[POI] = POIPixel;
            currentPOICount++;
            villageEntry = getVillageEntry(POI);
            villageMap.connectWithRoad(POI, villageEntry);
        }
    }
}
```

**Abbildung 6.12:**
Berechnung der Points-of-Interest in Pseudocode
Quelle: Eigene Darstellung

```
calcHouses() {
    villagePixelSet = getVillagePixel(villageMap);
    while(currentHouseCount < houseCount) {
        house = RandomElement(villagePixelSet);
        interest = calcInterest();
        threshold = Random(0,1);
        if(interest > threshold) {
            villageMap[house] = housePixel;
            currentHouseCount++;
            closestStreet = getClosestStreet(house);
            villageMap.connectWithRoad(house, closestStreet);
        }
    }
}

calcInterest(currentPixel) {
    //distance interest parameter
    distParam = [sociability, worship, accessibility, fortification]
    foreach(distParam p) {
        pixelOfInterest = getClosestPixel(p);
        distToPOI = distance(currentPixel, pixelOfInterest);
        if(distToPOI < p.best) {
            interest +=
                InverseLerp(p.min, p.best, distToPOI);
        } else {
            interest +=
                InverseLerp(p.best, p.max, distToPOI);
        }
    }
    interest = interest/distParam.count;

    //geographical domination
    distanceImpact = calcDistanceImpact();
    heightImpact = calcHeightImpact();
    interest = 0.3 * distanceImpact * heightImpact;

    return interest;
}
```

**Abbildung 6.13:**
Berechnung der Wohnhäuser in Pseudocode
Quelle: Eigene Darstellung

```
calcCountryRoads() {
    //calculate roads in low resolution
    lowResVillageMap = reduceResolution(villageMap, resolution);
    entryPairs = getEntryPairs(lowResVillageMap);
    foreach(entryPair (mapEntry, villageEntry) in entryPairs) {
        road = lowResVillageMap.connectWithRoad(
            mapEntry, villageEntry
        );
        countryRoads.add(road);
    }

    //transform low resolution roads to normal resolution
    foreach(road in countryRoads) {
        foreach(pixel in road) {
            waypoint = normalResCoordinates(pixel)
            waypoints.add(waypoint);
        }
        for(i=0; i < waypoints.count-1; i++) {
            villageMap.connectWithRoad(
                waypoints[i], waypoints[i+1]
            );
        }
    }
}
```

**Abbildung 6.14:**
Berechnung der Überlandstraßen in Pseudocode
Quelle: Eigene Darstellung

count = roadEntry count -evenly distribute map entries on map edges -avoid entries in steep areas -> calc avg steepness around potential map entry and check with maxSteepness-param if location fits -calc religious building seeds + roads to village entries -calc fortification building seeds + roads to village entries -connect unconnected villageEntries with street network -calc house seeds + connect to street network -ansatz zur berechnung von road cycles -> ergebnisse unzureichend -> verworfen

-generate country roads -calc low res steepness map -calc low res village map -> shortest path in low res maps from village entries to map entries -> transform low res roads into roads on high res map -use low res map to specify way points in high res map -> connect way points with shortest path algo

-calculate street map from village map -> used for texturing

-place village prefabs -prefab cleanUp -choose prefab and coords based on village map -rotate towards closest street pixel (describe vector calcs) -instantiate

## 6.4 Performance des Siedlungsgenerators

# 7 Ergebnisse und Evaluation

# 8 Zusammenfassung und Ausblick

# Literatur


[1] Charles Bloom. *Terrain Texture Compositing by Blending in the Frame-Buffer (aka „Splatting“ Textures)*. https://www.cbloom.com/3d/techdocs/splatting.txt. [Online; Zugriff am 29.08.2023]. 2000.

[2] Nicolas Alejandro Borromeo. *Hands-On Unity 2020 Game Development*. 1. Auflage. Packt Publishing, 2020.

[3] Richard J Carey und Donald P Greenberg. „Textures for realistic image synthesis“. In: *Computers & graphics* 9.2 (1985), S. 125–138.

[4] Keshav Channa. *Light Mapping - Theory and Implementation*. https://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml. [Online; Zugriff am 01.09.2023]. 2003.

[5] NVIDIA Corporation. *Cg Toolkit*. https://developer.nvidia.com/cg-toolkit. [Online; Zugriff am 12.09.2023]. 2023.

[6] Mark RT Dale und Mark Randall Thomas Dale. *Spatial pattern analysis in plant ecology*. Cambridge university press, 2000.

[7] Dawnosaur. *How Minecraft generates Worlds you want to explore*. https://dawnosaur.substack.com/p/how-minecraft-generates-worlds-you. [Online; Zugriff am 17.09.2023]. 2023.

[8] O. Deussen u. a. „Interactive visualization of complex plant ecosystems“. In: *IEEE Visualization, 2002. VIS 2002.* 2002, S. 219–226. DOI: 10.1109/VISUAL.2002.1183778.

[9] Oliver Deussen u. a. „Realistic Modeling and Rendering of Plant Ecosystems“. In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: Association for Computing Machinery, 1998, S. 275–286. ISBN: 0897919998. DOI: 10.1145/280814.280898. URL: https://doi.org/10.1145/280814.280898.

[10] Sean Dixon. *UV mapping, texturing and shaders, rigging and animation*. https://medium.com/@sdixon3/uv-mapping-texturing-and-shaders-rigging-and-animation-be9b4ddf0d48. [Online; Zugriff am 01.09.2023]. 2016.

[11] Arnaud Emilien u. a. „Procedural generation of villages on arbitrary terrains“. In: *The Visual Computer* 28 (2012), S. 809–818.

[12] esri. *ArcGIS*. https://www.esri.com/en-us/arcgis/about-arcgis/overview. [Online; Zugriff am 28.09.2023]. 2023.

[13] Marcus Gagelmann. „Entwicklung eines Biom-Generators für landschaftsähnliche Höhenkarten“. 2023.

[14] Marcus Gagelmann. „Entwicklung eines prozeduralen Landschaftsgenerators“. 2022.

[15] Proxy Games. *Low Poly Free Vegetation Kit*. https://assetstore.unity.com/packages/3d/environments/low-poly-free-vegetation-kit-176906. [Online; Zugriff am 23.09.2023]. 2022.

[16] Nate Glasser. *Texture Splatting in Direct3D*. https://www.gamedev.net/articles/programming/general-and-gameplay-programming/texturesplatting-in-direct3d-r2238/. [Online; Zugriff am 29.08.2023]. 2005.

[17] BiteTheBytes GmbH. *World Creator*. https://www.world-creator.com/. [Online; Zugriff am 24.09.2023]. 2023.

[18] google. *Google Maps*. https://www.google.de/maps/. [Online; Zugriff am 28.09.2023]. 2023.

[19] Stefan Greuter u. a. „Real-Time Procedural Generation of ‘pseudo Infinite’ Cities“. In: *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. GRAPHITE ’03. Melbourne, Australia: Association for Computing Machinery, 2003, 87–ff. ISBN: 1581135785. DOI: 10.1145/604471.604490. URL: https://doi.org/10.1145/604471.604490.

[20] Eric Haines. „An introductory tour of interactive rendering“. In: *IEEE Computer Graphics and Applications* 26.1 (2006), S. 76–87.

[21] Pat Hanrahan und Jim Lawson. „A language for shading and lighting calculations“. In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990, S. 289–298.

[22] Alexandre Hardy und Duncan Andrew Keith Mc Roberts. „Blend maps: enhanced terrain texturing“. In: *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. 2006, S. 61–70.

[23] Mostafa Hassan. *Proposed workflow for UV mapping and texture painting*. Bachelor’s Thesis. 2016.

[24] Paul S Heckbert. *Fundamentals of texture mapping and image warping*. Master’s thesis. 1989.

[25] Paul S Heckbert. „Survey of texture mapping“. In: *IEEE computer graphics and applications* 6.11 (1986), S. 56–67.

[26] BRIAN HOPKINS und J. G. SKELLAM. „A New Method for determining the Type of Distribution of Plant Individuals“. In: *Annals of Botany* 18.2 (Apr. 1954), S. 213–227. ISSN: 0305-7364. DOI: 10.1093/oxfordjournals.aob.a083391. eprint: https://academic.oup.com/aob/article-pdf/18/2/213/778952/18-2-213.pdf. URL: https://doi.org/10.1093/oxfordjournals.aob.a083391.

[27] The Big Procedural Game Journal. *Generating Seeds*. https://www.gamedev.net/blogs/entry/2259964-generating-seeds/. [Online; Zugriff am 24.09.2023]. 2014.

[28] Kyle Kukshtel. *Depthkit + Unity Shader Graph*. https://forums.depthkit.tv/t/depthkit-unity-shader-graph/32. [Online; Zugriff am 29.08.2023]. 2018.

[29] Brendan Lane, Przemyslaw Prusinkiewicz u. a. „Generating spatial distributions for multilevel models of plant communities“. In: *Graphics interface*. Bd. 2002. Citeseer. 2002, S. 69–87.

[30] Laufes. *Lowpoly Forest Pack*. https://assetstore.unity.com/packages/3d/vegetation/trees/lowpoly-forest-pack-21180. [Online; Zugriff am 23.09.2023]. 2017.

[31] LearnOpenGL. *Specular maps*. https://learnopengl.com/Lighting/Lighting-maps. [Online; Zugriff am 29.08.2023].

[32] Mapquest Holdings LLC. *MapQuest*. https://www.mapquest.com/. [Online; Zugriff am 28.09.2023]. 2023.

[33] World Machine Software LLC. *World Machine*. https://www.world-machine.com/. [Online; Zugriff am 24.09.2023]. 2022.

[34] Marglyph. *Tree Science: definitive guide up-to-date for 1.2_02*. https://www.minecraftforum.net/forums/minecraft-java-edition/survival-mode/223343-tree-science-definitive-guide-up-to-date-for-1. [Online; Zugriff am 17.09.2023]. 2011.

[35] Microsoft. *Reference for HLSL*. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference. [Online; Zugriff am 12.09.2023]. 2023.

[36] Steve Cunningham Mike Bailey. *Graphics Shaders: Theory and Practice, Second Edition*. 2. Auflage. CRC Press, 2016.

[37] OpenGL. *Core Language (GLSL)*. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL). [Online; Zugriff am 12.09.2023]. 2021.

[38] OpenGL. *Rendering Pipeline Overview*. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. [Online; Zugriff am 29.08.2023]. 2022.

[39] OpenGL. *Shader*. https://www.khronos.org/opengl/wiki/Shader. [Online; Zugriff am 29.08.2023]. 2019.

[40] Yoav I. H. Parish und Pascal Müller. „Procedural Modeling of Cities“. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, S. 301–308. ISBN: 158113374X. DOI: 10.1145/383259.383292. URL: https://doi.org/10.1145/383259.383292.

[41] Ken Perlin. „An image synthesizer“. In: *ACM Siggraph Computer Graphics* 19.3 (1985), S. 287–296.

[42] P. Prusinkiewicz u. a. *Lindenmayer Systems, Fractals, and Plants*. Lecture Notes in Biomathematics. Springer New York, 2013. ISBN: 9781475714296. URL: https://books.google.de/books?id=CRwOswEACAAJ.

[43] Mike Rozak. *Oblivion: Full spectrum content, from hand-generated to procedural*. https://web.archive.org/web/20110709160749/http://www.mxac.com.au/drt/OblivionProcedural.htm. [Online; Zugriff am 17.09.2023]. 2006.

[44] Planetside Software. *What's New in Terragen 4*. https://planetside.co.uk/whats-new-in-terragen-4/. [Online; Zugriff am 24.09.2023]. 2022.

[45] GameBatata Studio. *POLY STYLE - Vegetation Pack*. https://assetstore.unity.com/packages/3d/vegetation/poly-style-vegetation-pack-104112. [Online; Zugriff am 23.09.2023]. 2017.

[46] Jun Sun, Limei Xu und Liangping Zhu. „Alpha Mapping in Scene Simulation of Launch Vehicle“. In: *2006 International Conference on Mechatronics and Automation*. IEEE. 2006, S. 1757–1762.

[47] Unity Technologies. *High Definition Render Pipeline overview*. https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@16.0/manual/index.html. [Online; Zugriff am 29.08.2023]. 2023.

[48] Unity Technologies. *Shader Graph*. https://unity.com/de/features/shader-graph. [Online; Zugriff am 29.08.2023]. 2023.

[49] Unity Technologies. *Universal Render Pipeline overview*. https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@16.0/manual/index.html. [Online; Zugriff am 29.08.2023]. 2023.

[50] Naty Hoffman Tomas Akenine-Möller Eric Haines. *Real-Time Rendering*. 4. Auflage. CRC Press, 2018.

[51] Carlos A. Vanegas u. a. „Visualization of Simulated Urban Spaces: Inferring Parameterized Generation of Streets, Parcels, and Aerial Imagery“. In: *IEEE Transactions on Visualization and Computer Graphics* 15.3 (2009), S. 424–435. DOI: 10.1109/TVCG.2008.193.

[52] Broken Vector. *Low Poly Tree Pack*. https://assetstore.unity.com/packages/3d/vegetation/trees/low-poly-tree-pack-57866. [Online; Zugriff am 23.09.2023]. 2018.

[53] Jason Weber und Joseph Penn. „Creation and rendering of realistic trees“. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, S. 119–128.

[54] Li-Yi Wei und Marc Levoy. *Texture Analysis and Synthesis*. https://graphics.stanford.edu/projects/texture/. [Online; Zugriff am 01.09.2023].

[55] Mark Westoby. „The Self-Thinning Rule“. In: Hrsg. von A. MacFadyen und E.D. Ford. Bd. 14. Advances in Ecological Research. Academic Press, 1984, S. 167–225. DOI: https://doi.org/10.1016/S0065-2504(08)60171-3. URL: https://www.sciencedirect.com/science/article/pii/S0065250408601713.

[56] Robert Harding Whittaker. *Communities and ecosystems*. 1. Auflage. Macmillan, 1970.

[57] Minecraft Wiki. *Seed (level generation)*. https://minecraft.fandom.com/wiki/Seed_(level_generation). [Online; Zugriff am 17.09.2023]. 2023.

[58] Minecraft Wiki. *Tree*. https://minecraft.fandom.com/wiki/Tree. [Online; Zugriff am 17.09.2023]. 2023.

[59] WikiBooks. *Blender Dokumentation: UV-Mapping*. https://de.wikibooks.org/wiki/Blender_Dokumentation:_UV-Mapping. [Online; Zugriff am 29.08.2023]. 2018.

[60] Wikipedia. *Normal Mapping*. https://de.wikipedia.org/wiki/Normal_Mapping. [Online; Zugriff am 29.08.2023]. 2017.

[61] Wikipedia. *Texture splatting*. https://en.wikipedia.org/wiki/Texture_splatting. [Online; Zugriff am 01.09.2023]. 2021.

[62] Procedural Worlds. *Gaia*. https://www.procedural-worlds.com/products/indie/gaia/. [Online; Zugriff am 24.09.2023]. 2023.

[63] Alan Zucconi. *The World Generation of Minecraft*. https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/. [Online; Zugriff am 17.09.2023]. 2022.