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

ausgegeben und betreut von
Prof. Dr. Stefan Schlechtweg

mit Unterstützung von
Prof. Dr. Alexander Carôt

Halle (Saale), den 11. November 2023

Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Max Mustermann)
Musterhausen, 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 Motivation

1.2 Verwandte Arbeiten

1.3 Zielstellung

~~In diesem Kapitel wird die Zielsetzung der Arbeit beschrieben.~~ Es soll ein Konzept für ein prozedurales System entworfen werden, das in der Lage ist, digitale Welten zu generieren. Dabei soll sichergestellt werden, dass das System und sein Output folgende Anforderungen bestmöglich erfüllen: Realismus, Flexibilität, Performance und Benutzbarkeit.

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.

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.

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.

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 [6] [5] 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 [6] 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 [20]. 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.

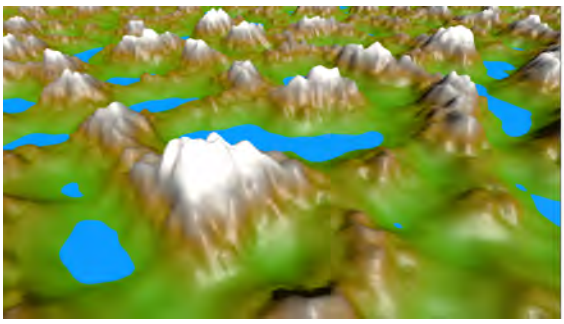
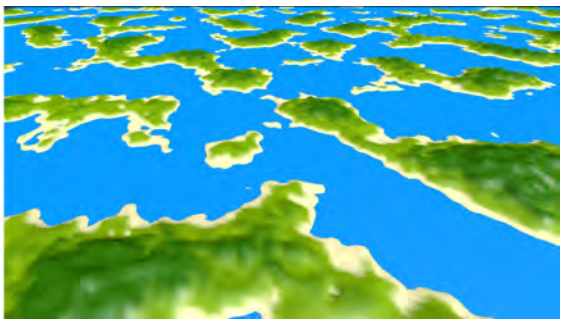

Abbildung 2.1: Verschiedene generierbare Höhenkarten
Quelle: Eigene Darstellung

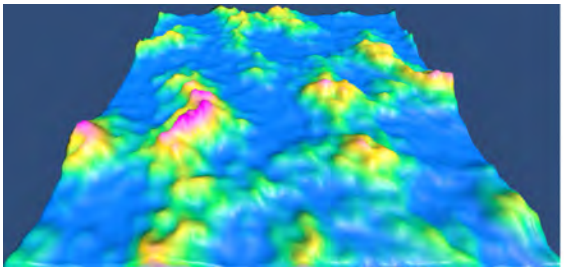
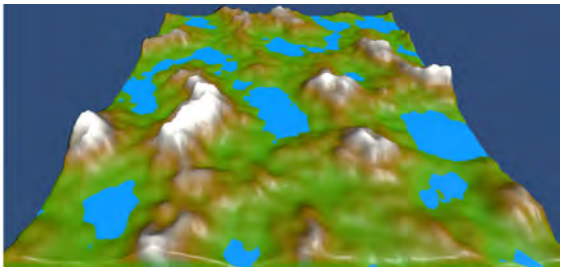

Abbildung 2.2: Standard Terrain mit unterschiedlichen Farbgradienten
Quelle: Eigene Darstellung

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 [5] 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 [27] 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

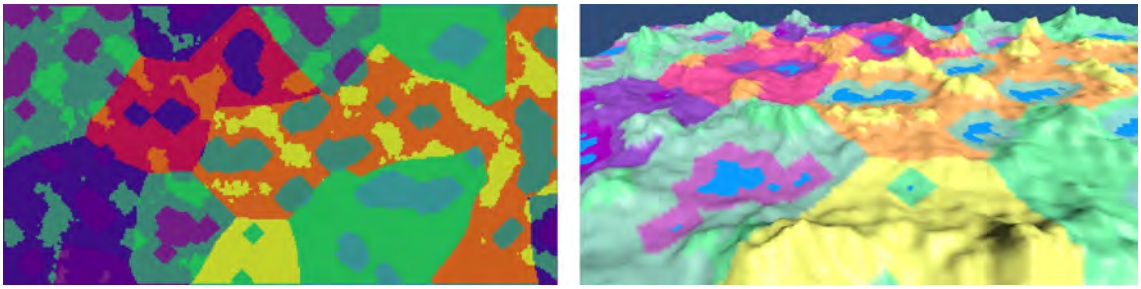

Abbildung 2.3: Eine Biom-Karte als Textur (links) und projiziert auf die zugehörige Höhenkarte (rechts)
Quelle: Eigene Darstellung

- 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

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. 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 Kapitel werden die Grundlagen zur Arbeit an dem Texturierungssystem vermittelt. Dabei werden ~~abschnittsweise~~ Konzepte und Techniken vorgestellt sowie auf wichtige Publikationen hingewiesen. Im ersten Abschnitt gibt es 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 und Texturen 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.

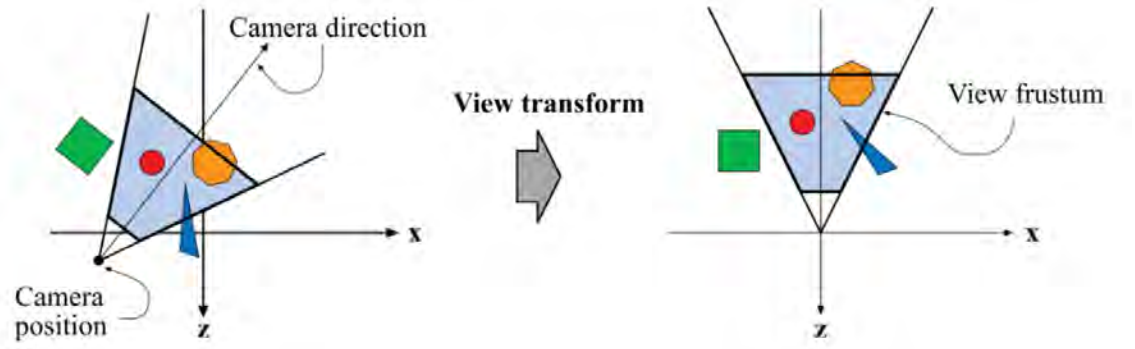

Abbildung 4.1: Szenentransformation der Geometriephase
Quelle: [26]

Die Architektur der Grafik-Pipeline

Die Basisarchitektur einer Grafik-Pipeline umfasst drei Hauptphasen: die Anwendungsphase, die Geometriephase und die Rasterisierungsphase. Jede dieser Phasen kann wiederum in Unterphasen aufgeteilt werden, welche teilweise als Pipeline, teilweise jedoch auch parallelisiert ausgeführt werden können [26] [8].

In der ersten Phase jeder Grafik-Pipeline, der Anwendungsphase, werden die genauen Spezifikationen der zu rendernden Grundelemente, wie Eckpunkte, Linien und Dreiecke, festgelegt. Diese Informationen werden dann an die Geometriephase weitergeleitet. Typische Aufgaben in dieser Phase umfassen Kollisionsberechnungen, die Verarbeitung von Nutzereingaben, Animationsberechnungen und mehr. Die Anwendungsphase wird vollständig auf der CPU ausgeführt, wodurch Entwickler die volle Kontrolle über die Abläufe in dieser Phase haben [26] [8].

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 (siehe Abbildung 4.1) und in Richtung der z-Achse zeigt. Dies vereinfacht spätere Berechnungen [26] [8].

Die Geometriephase setzt sich fort mit dem Vertex-Shading, bei dem die Auswirkungen des Lichteinfalls auf die Materialien in der Szene als "Texture Enhancement Factor"berechnet werden. Gefolgt wird das Vertex-Shading vom 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. ~~Aufgrund der genannten Eigenschaften findet die orthogonale Projektion überwiegend Anwendung in technischen Darstellungen.~~ Für realistische Ansichten aus Sicht eines Betrachters ist sie weitgehend unbrauchbar, da die dargestellten Objekte verzerrt wahrgenommen werden [26] [8].

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 [26] [8].

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 [26] [8].

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 [26] [8].

Die Rasterisierungsphase bildet den Abschluss der Grafik-Pipeline. Hier werden die geometrischen, kontinuierlichen Grundelemente in diskrete Fragmente umgewandelt, die als Pixel auf dem Bildschirm dargestellt werden. Die eingeschränkte Sichtbarkeit von Pixeln, bedingt durch die Überlappung von Polygonen, wird mithilfe eines Z-Buffer-Algorithmus bestimmt. Anschließend erfolgt die Berechnung der sichtbaren Pixel-Farben basierend auf Beleuchtungsdaten, Texturen und anderen Materialeigenschaften [26] [8].

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 ~~gewaltigen~~ 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 SScriptable Render Pipeline"(SRP) ein. Dieses Produkt ermöglicht es Entwicklern, maßgeschneiderte Render-Pipelines für spezifische Projekte zu erstellen. Auf der Grundlage der SRP wurden die Üniversal 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].

URP eignet sich hierbei für Anwendungen mit weniger anspruchsvollen grafischen Anforderungen und ist auf vielen verschiedenen Plattformen, einschließlich mobiler Geräte, nutzbar [25]. HDRP hingegen unterstützt viele Features zur Umsetzung von anspruchsvollen grafische Anwendungen und ist deshalb größtenteils auf leistungsstarken Endgeräten wie PCs und Konsolen einsetzbar [23].

4.1.2 Shader und Shader-Graph

Shading ist eine fundamentale Technik in der 3D-Computergrafik, die verwendet wird, um das visuelle Erscheinungsbild von Objekten zu bestimmen. Ein Shader ist ein benutzerdefiniertes Programm, das in einer high-level Programmiersprache geschrieben wird und unterschiedliche Code-Schnittstellen innerhalb der Render-Pipeline anspricht [19].

Die Entwicklung des Shaders

Die ersten Ansätze zur Shader-Entwicklung wurden in den 1980er Jahren im Rahmen von Software-Renderern entwickelt. Ein Beispiel dafür ist das SShade Trees"Modell von Robert L. Cook aus dem Jahr 1984. Dieses flexible Shading-Modell nutzte eine Baumstruktur, um benutzerdefinierte Skripte an vordefinierten Stellen in der Render-Pipeline einzufügen. Diese Idee erlaubt es Entwicklern, eine Vielzahl von grafischen Effekten zu erzeugen, ohne den Haupt-Renderer ständig direkt modifizieren zu müssen [4]. Ein weiterer wichtiger Meilenstein war der "Pixel Stream Editorßum Design realistischer CGI von Ken Perlin aus dem Jahr 1985. Dieser Software-Renderer baut auf den Ergebnissen von Cook auf und generierte einige zu diesem Zeitpunkt sehr aufwändige oder sogar unerreichte grafische Effekt [20]. Die Popularität von Shading-Techniken nahm 1990 mit der Einführung der RenderMan Shading Language [9] zu. Die RenderMan-Spezifikation ist bis heute ein weit verbreiteter Industriestandard. Sie ermöglicht es Software-Renderern, Oberflächenfarben, Verschiebungen, Lichtfarben, Lichtrichtungen und Volumeneffekte durch die

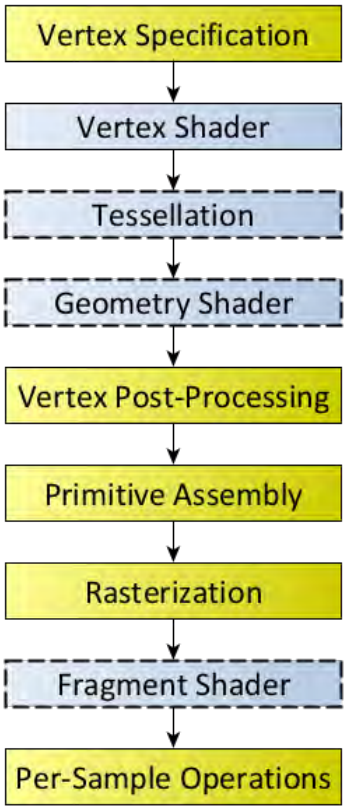

Abbildung 4.2: Aufbau der Grafik-Pipeline
Quelle: [18]

Verwendung von Shadern zu beeinflussen. Auf der Hardware-Ebene werden Shader genutzt, um Vertex-Transformationen, Pixelfarben und neuerdings sogar Geometrieänderungen zu spezifizieren. Vertex-Shader bearbeiten ausschließlich die Objektgeometrie (Position, Texturkoordinaten, Normalen, Tangenten, Binormalen und Vertex-Färbungen), während Pixel-Shader nur Farbwerte auf den auszugebenden Bildpunkten beeinflussen. Geometrie-Shader hingegen erweitern oder modifizieren die vorhandene Geometrie, indem sie neue Vertex-Gruppen berechnen [16] [17]. Einen Überblick über die Grafik-Pipeline und ihre Shader-Schnittstellen gibt Abbildung 4.2.

Shader-Graph: Shader ohne Programmcode

Viele der aktuellen Game Engines bieten ihren Nutzern seit einiger Zeit Authoring-Tools zur Erstellung von individuellen Shadern an, ohne dass dafür eigener Programmcode geschrieben werden muss. In der Unity Game Engine heißt dieses Tool

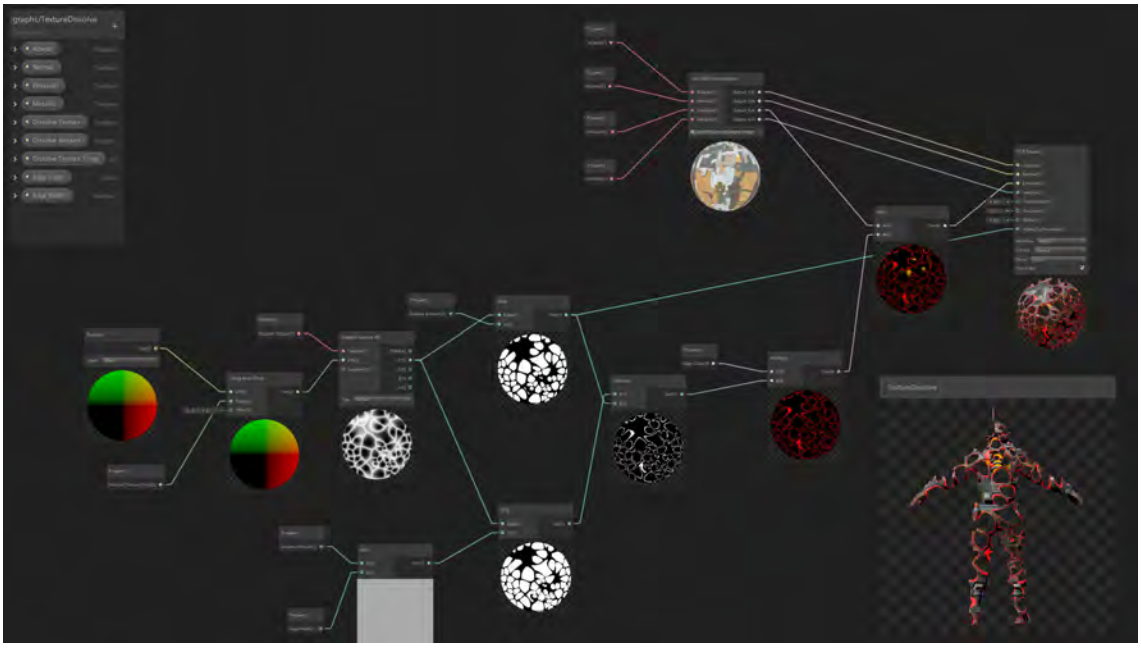

Abbildung 4.3: Beispiel eines Shader-Graph in Unity
Quelle: [14]

Shader-Graph 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. 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 [24]. Ein Beispiel für einen in Unity umgesetzten Shader-Graph ist in Abbildung 4.3 dargestellt.

4.1.3 Texture-Mapping und Texture Splatting

Der Begriff Texture wird von Paul S. Heckbert definiert als ein multidimensionales Bild, das auf einen multidimensionalen Raum abgebildet wird [13]. 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.

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 [13]. Ursprünglich wurden auf diese Weise lediglich transformierte Farbpixel auf eine Oberfläche gezeichnet. Diese Technik wird heute als Diffuse-Mapping bezeichnet und stellt eine Unterkategorie des Texture-Mappings dar. Darüber hinaus existieren weitere Variationen wie Normal-Mapping, Alpha-Mapping, Specular-Mapping, Illumination-Mapping und weitere. Alle diese Techniken haben gemein, dass bei deren Anwendung keine Veränderung der geometrischen Auflösung der Zielobjekte durchgeführt wird [11] [3]. Auf diese Weise kann einer Szene eine hohe Detailgenauigkeit hinzugefügt werden, während der zusätzliche Render-Aufwand moderat bleibt [12].

Ablauf des typischen Texture-Mappings

Beim typischen Texture-Mapping liegt das zu texturierende Modell im Objektraum vor, repräsentiert als eine Abfolge von Eckpunkten und zugehörigen UV-Koordinaten. Diese UV-Koordinaten folgen den Eckpunktdaten ihres Modells und geben dadurch die nötigen Informartionen vor, um Texturen präzise auf einem Objekt abzubilden, unabhängig von möglichen Transformation. Durch verschiedene Prozeduren werden die UV-Koordinaten verarbeitet und interpoliert, wodurch ein vollständiger UV-Raum mit einer UV-Textur des Modells einsteht. Dabei können sowohl automatische Softwareprozesse als auch Tools zur manuellen Nachbearbeitung der UV-Textur verwendet werden. Die Bildtextur wird abschließend auf die UV-Textur projiziert, wodurch das 3D-Objekt in eine 2D-Textur ëingewickelt"wird [11] [28]. Dieser Prozess wird in Abbildung 4.4 veranschaulicht.

Alpha-Mapping und RGBA-Farbpixel

Alpha-Mapping, auch als Transparency-Mapping bekannt, bezeichnet eine Technik aus dem Bereich der Computergrafik, die Transparenzwerte mithilfe von Texturen auf Objekte abbildet. Dabei werden häufig die Alpha-Kanäle der Farbpixel zur Aufbewahrung der Transparenzdaten verwendet, wodurch die Technik ihren Namen erhielt. Alpha-Mapping ermöglicht die Simulation von partiell transparenten Oberflächen (siehe Abbildung 4.5), ohne die aufwändigen Berechnungen für die Umsetzung der gewünschten Effekte in geometrischer Form durchführen zu müssen [22]. Catmull und Smith betonten bereits in ihrer Arbeit von 1995, dass Transparenz die visuelle Wirkung eines Objektes genauso fundamental beeinflusst wie dessen Farbkanäle [21]. Daraus resultierte die Einführung eines "RGBAFarbpixel mit drei Farbkanälen

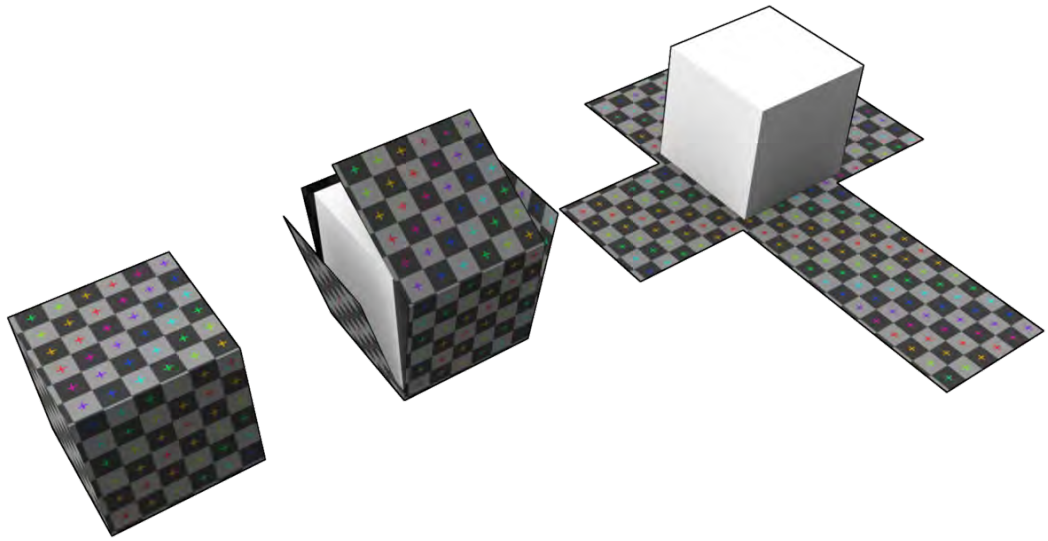

Abbildung 4.4: UV-Unwrapping dargestellt an einem Würfel
Quelle: [11]

Abbildung 4.5: Vergleich der linearen Interpolation zweier Texturen ohne (oben) und mit (unten) Alpha-Map
Quelle: [10]

und einem Alpha-Kanal, um Transparenzdaten als gleichwertigen Teil eines Bildes abspeichern zu können.

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 und anschließend übereinander geschichtet. Dabei 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] [7].

Transparenzwerte sind Zahlenwerte, die in einem mehrdimensionalen Array abgespeichert werden. Aus diesem Grund ist es theoretisch möglich, Alpha-Werte auch abseits der Alpha-Kanäle in beliebigen Kanälen (einschließlich RGB-Kanälen der Farbpixel) zu platzieren. Um den Ressourcenbedarf zu reduzieren und den Entwickler eine bessere Übersicht zu gewährleisten, werden in der Praxis eigene Splat-Texturen eingesetzt, die unter Verwendung aller Kanäle der Farbpixel bis zu vier

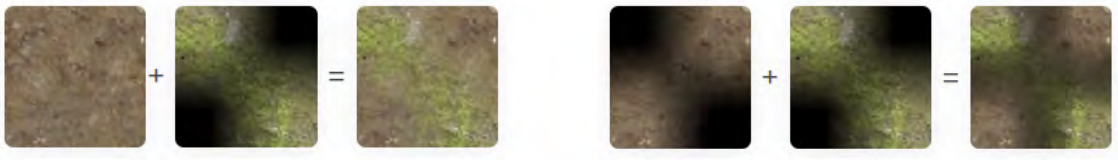

Abbildung 4.6: Vergleich Fehlerfreier Blend (links) und Blend mit Texturlöchern (rechts)
Quelle: [7]

Alpha Maps aufnehmen können. Andernfalls wäre für jede benötigte Alpha Map die Erstellung einer gesonderten Textur notwendig. Abschließend kann in Kombination mit den Objekt-Texturen, der Geometrie der Zieloberfläche und den Splat-Texturen ein gemischtes Texturbild mit kontinuierlichen Übergängen berechnet werden [1] [7].

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 Zielsetzung

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 Unterabschnitt 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 möglicher Ansätze

Die Umsetzung eines Texturierungssystems für den Weltengenerator erfordert eine fundierte Evaluierung verschiedener Ansätze, um die gesteckten Ziele effizient zu erreichen. In diesem Abschnitt werden potenzielle Ansätze zur Implementierung eines Texturierungssystems betrachtet und die Vor- und Nachteile dieser Ansätze diskutiert.

Wahl der Render Pipeline

Die Auswahl der Render Pipeline ist ein entscheidender Faktor bei der Gestaltung des Texturierungssystems. Verschiedene Optionen wie die Build-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. HDRP wird gut von Unity gewartet, weist jedoch eingeschränkte Plattformkompatibilität auf, was die Flexibilität der Anwendung beeinträchtigt.
- **Build-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 Build-in

Render Pipeline jederzeit möglich. Außerdem bietet sie ähnliche Funktionen wie 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 Build-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.

Texturart: Prozedurale oder vorgefertigte Texturen

Eine prozedurale Textur ist eine algorithmisch-mathematische Beschreibung der Charakteristiken einer Objektoberfläche. 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.

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. Die Textursynthese bietet jedoch so weitläufige Möglichkeiten, dass dessen Integration in das Projekt den Rahmen dieser Arbeit sprengen würde. Auch dieser Ansatz könnte in einem zukünftigen Projekt interessante Ergebnisse liefern.

Eine Alternative zu den prozeduralen Texturen bieten vorgefertigte Texturen, die zur Laufzeit aus dem Systemspeicher geladen werden. Hierbei kann zwischen den folgenden zwei Möglichkeiten 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, dessen 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 aus, was dem Ziel der Benutzbarkeit definitiv, aber auch dem Ziel des Realismus potenziell widerspricht.

Tiling-Texturen: Diese Art der Textur bildet ein relativ kleines Muster einer regelmäßigen Materialoberfläche wie Holz oder Backsteinwänden als Quadrat ab. Maßgeblich für Tiling-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 Tiling-Texturen angewendet wird. Da diese Art der Texturierung mithilfe der Grafik-Pipeline auch automatisch erfolgen kann, bieten Tiling-Texturen die Möglichkeit, die Texturierung einer Welt dynamisch an geänderte Umstände anzupassen. Ein Austausch der Tiling-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 bieten Tiling-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. Glücklicherweise 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).

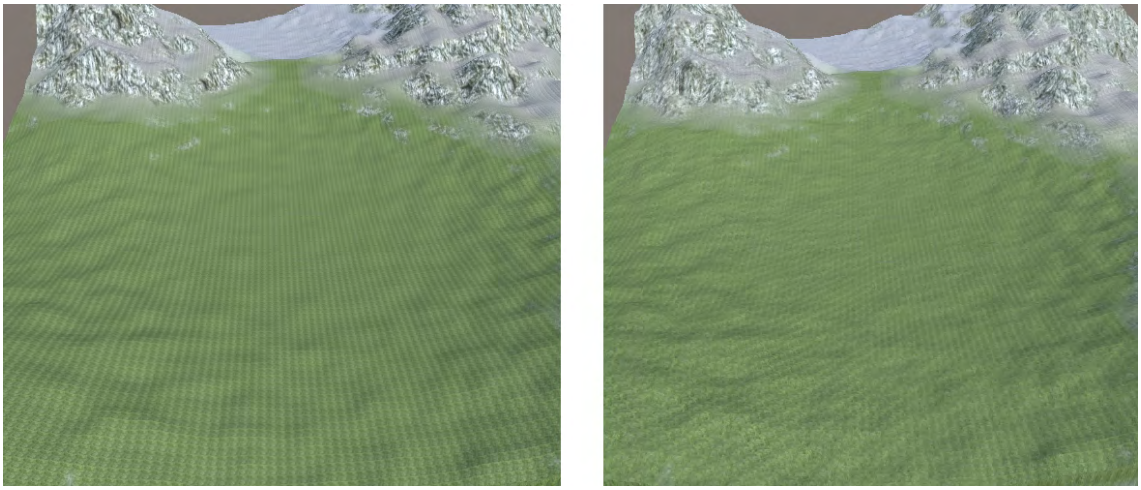

Abbildung 4.7: Vergleich von Repetitiven Tiles (links) und Transformierten Tiles (rechts)
Quelle: Eigene Darstellung

Für dieses Projekt ist die Verwendung externe Anwendungen zur Lösung des Problems problematisch, da das angestrebte Design des Weltengenerator 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. Shader-Graph 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 sowie zur Interpolation von Texturen als auch zur Berechnung von Noise Maps eingesetzt.

Der Vorteil dieses Ansatzes liegt auf der Hand: Sämtliche Berechnungen können innerhalb der Unity Game Engine durchgeführt werden. Dadurch ergibt sich die Möglichkeit, die nötigen Abläufe zu automatisieren und in die Pipeline zur Weltengenerierung 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.

Texturauswahl: Farben, Tiefe, Spiegelungen und Schattierungen

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.

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 projiziert. Durch diesen Prozess werden sowohl die Farbgebung als auch die Musterung 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 4.8). Möglich wird dies durch das Speichern von dreidimensionalen Raumvektoren 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 die Projektion der Normal Texture auf das Zielobjekt in einer niedrigen geometrischen Auflösung, werden die gespeicherten Schattierungsdetails 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 4.9), kann das sogenannte Specular-Mapping verwendet werden. Bei dieser Technik wird ein Specular-Werte für jeden Pixel einer Textur als Specular Map abgespeichert. Der Specular Wert gibt an, welcher Anteil an einfallendem Licht von einer Oberfläche reflektiert wird. ~~Ein hoher Wert hat zur Folge, dass das betroffene Objekt glänzt.~~ Die Specular Map wird auf eine Objektoberfläche projiziert, 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

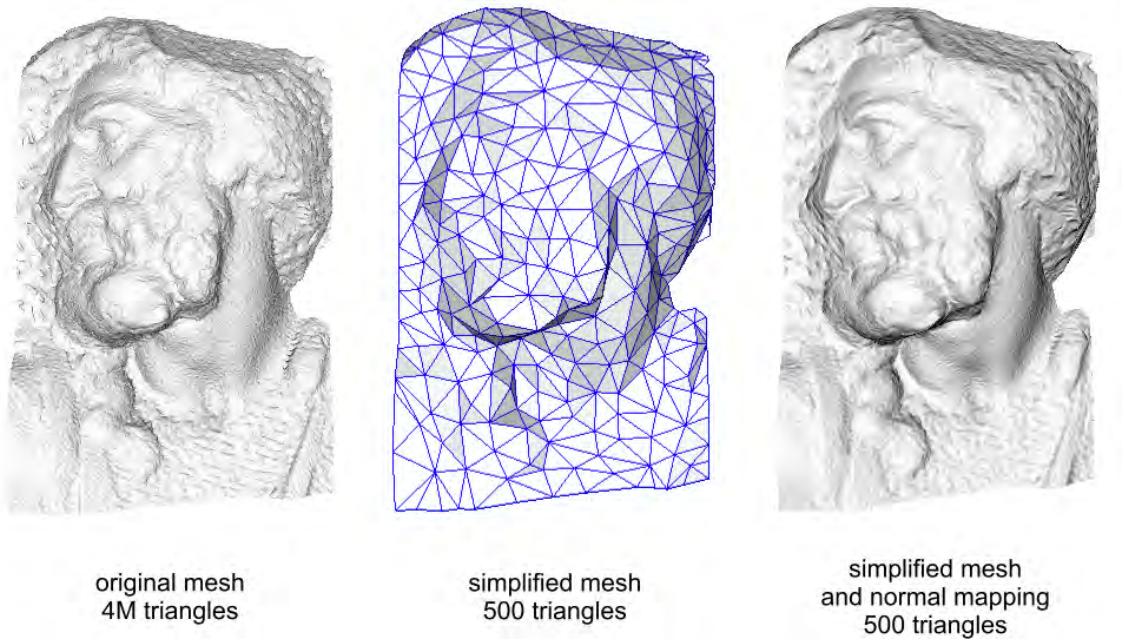

Abbildung 4.8: Normal-Mapping am Beispiel eines 3D-Büstenmodells
Quelle: [30]

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 projiziert werden (siehe Abbildung 4.10). 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 Positions- oder Rotationsänderungen 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 aufwändig und sollte nicht zur Laufzeit durchgeführt werden. Aus diesem Grund ist Light-Mapping nicht für dynamische Objekte geeignet.

Verteilung der Texturen auf dem Terrain

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.

Abbildung 4.9: Vergleich einer Kiste ohne (links) und mit (rechts) Specular-Mapping
Quelle: [15]

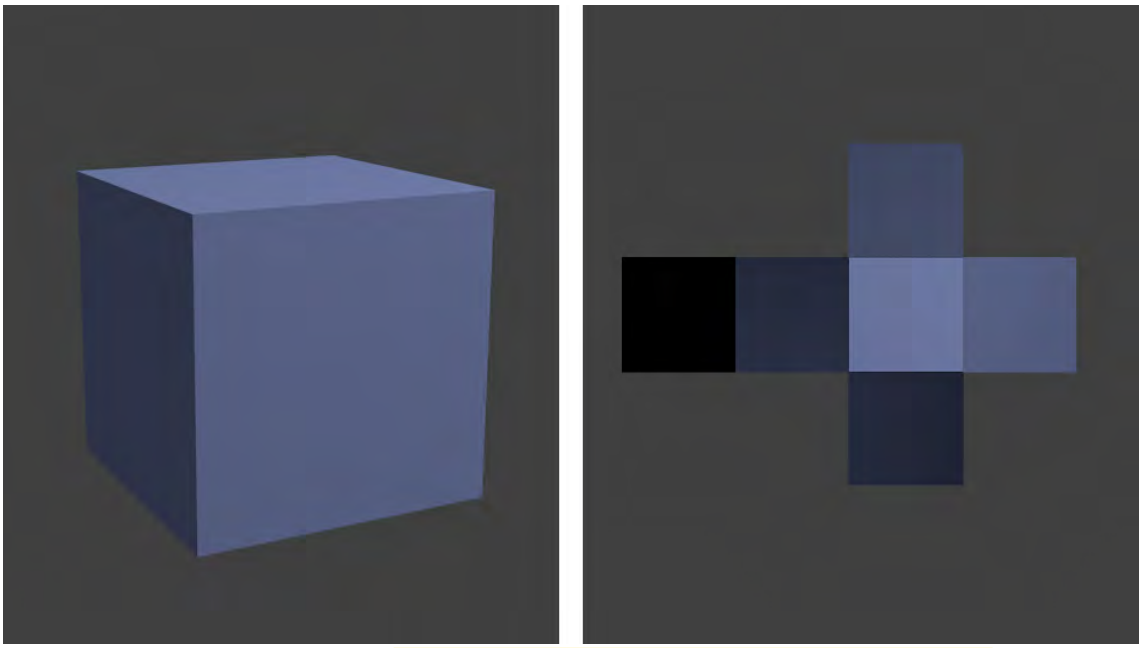

Abbildung 4.10: Ein Würfel und seine zugehörige Light Map
Quelle: [29]

Ein naiver Ansatz für dieses Problem stellt die manuelle Anfertigung einer Großtextur durch den Nutzer dar, die auf die vollständige Oberfläche des Terrains projiziert 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 mentalen Modelle 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 dem Mesh~~ 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 ein automatisierter Ansatz zur Verteilung der Texturen auf dem Mesh gewählt werden. Vielversprechend ist dafür die Technik des Texture Splatting (siehe Unterabschnitt 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. 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.

4.2.3 Das vollständige Konzept

Nachdem in den vorhergehenden Abschnitten die Ziele und mögliche Ansätze zur Umsetzung des Texturierungssystems diskutiert wurden, wird in diesem Abschnitt das finale Konzept des Systems vorgestellt und die getroffenen Entscheidungen kurz begründet.

Wahl der Render Pipeline

Bei der Wahl der Render Pipeline fiel die Entscheidung für die Universal Render Pipeline der Unity Game Engine aus. Diese bietet beinahe den gleichen Funktionsumfang wie die Build-in Render Pipeline und wird gleichzeitig besser vom Entwickler gewartet. Zusammen mit der Möglichkeit, benutzerdefinierte Shader zu programmieren, stellt die URP sowohl aktuell als auch in zukünftigen Projekten eine zuverlässige und flexible Grafik Pipeline dar. Die Wahl der High Definition Render Pipeline wurde verworfen, um eine breite 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.

Texturart: Prozedurale oder vorgefertigte Texturen

Die Idee der Entwicklung eines Verfahrens zur Textursynthese wurde verworfen. Der umfassenden Forschungsmöglichkeiten in diesem Bereich sprengen den Rahmen dieser Arbeit. 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 (Verweis auf verwandte Arbeiten: Spiele und Weltengeneratoren). Vorteile dieser Lösung sind eine zufriedenstellende Flexibilität und Realismus bei einer gleichzeitig hohen Benutzerfreundlichkeit.

Lösungsansatz zur Vermeidung von Tile-Repetition

Um die repetitiven Muster von Tiling-Texturen vor dem Nutzer zu verbergen, wird im Folgenden ein Shader konzipiert, der die verwendeten Tiling-Texturen mithilfe von Noise-Maps transformiert und die besagten Muster auflöst. Ein Diagramm zur Verdeutlichung der Funktionsweise des Shaders ist in Abbildung 4.11 Dargestellt.

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

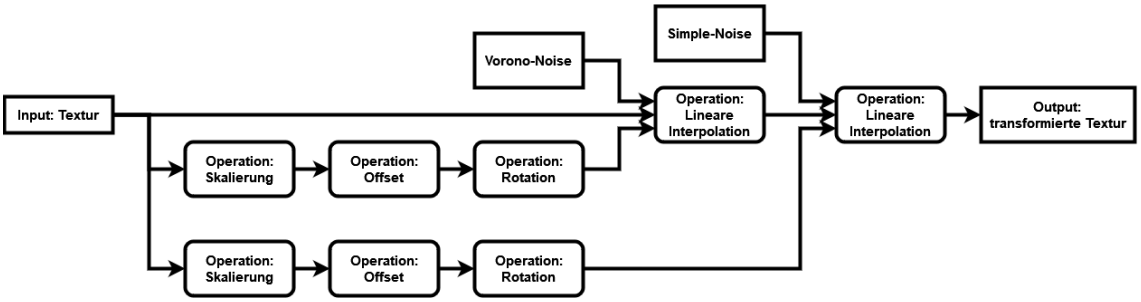

Abbildung 4.11: Funktionsweise des Texture Tile Splitter Shaders
Quelle: Eigene Darstellung

1. Die Textur wird um einen einige Prozent Hochskaliert. Dadurch scheint sie etwas näher am Betrachter zu sein.
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 die daraus entstehende transformierte Textur mit der Input-Textur linear interpoliert. Für diese Interpolation 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. Die Ergebnistextur wird im Anschluss mit 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 mithilfe einer 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 LERP-Operationen 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.

Texturauswahl

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 werden unter Berücksichtigung die 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 Form des Terrains zur Laufzeit ständigen Veränderungen ausgesetzt ist, müssten die zuständigen Light Maps in regelmäßigen Abständen während der Laufzeit 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. 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.

Verteilung der Texturen auf dem Terrain

Die Texturen des Terrains werden in diesem Projekt mithilfe der Texture-Splatting-Methode automatisch auf dem Mesh verteilt. Dadurch ist es möglich, die Texturierung dynamisch an ein verändertes Terrain anzupassen sowie die Übergänge zwischen den Biomen glaubhaft zu interpolieren.

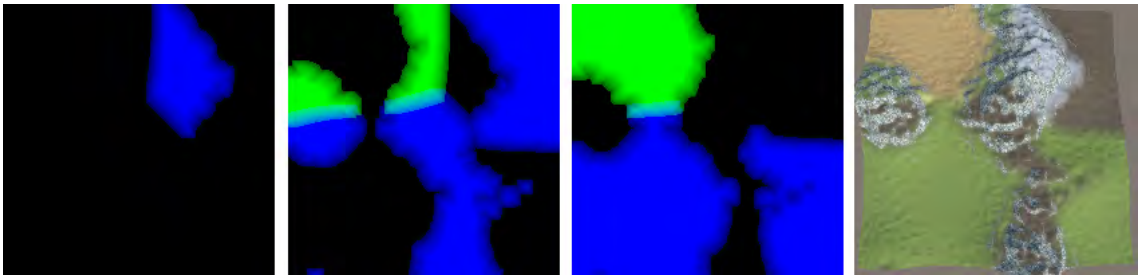

Abbildung 4.12: Beispiel einer möglichen Menge von Splat Maps und der resultierenden Textur
Quelle: Eigene Darstellung

Zu diesem Zweck 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, 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.

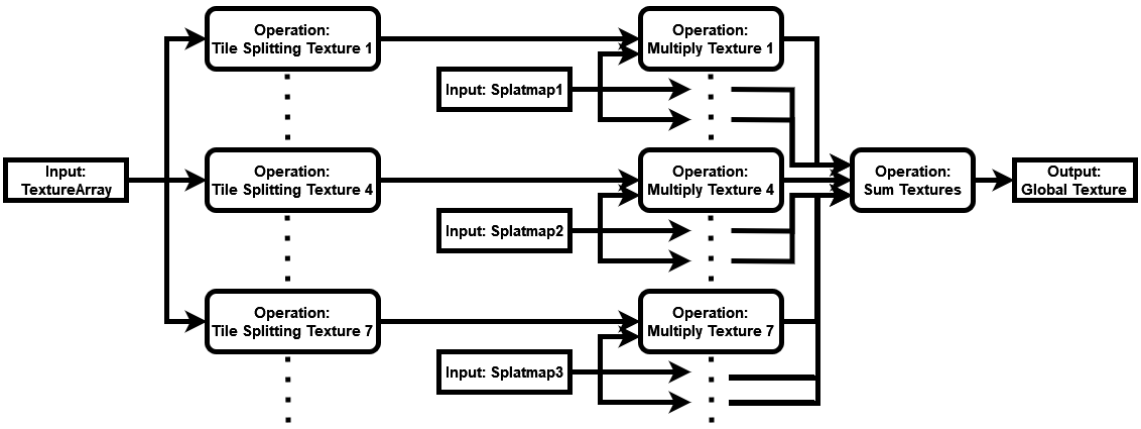

Abbildung 4.13: Funktionsweise des Texture Splat Shaders
Quelle: Eigene Darstellung

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.

Die Vorteile dieses Ansatzes liegen auf der Hand. Er nimmt eine große Arbeitslast von Nutzern, da diese die Texturen nicht manuell auf das Terrain malen müssen. 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 Aktualisierung der Texturen nur erfolgen kann, wenn auch die Biom-Karte aktualisiert wird. 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.

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

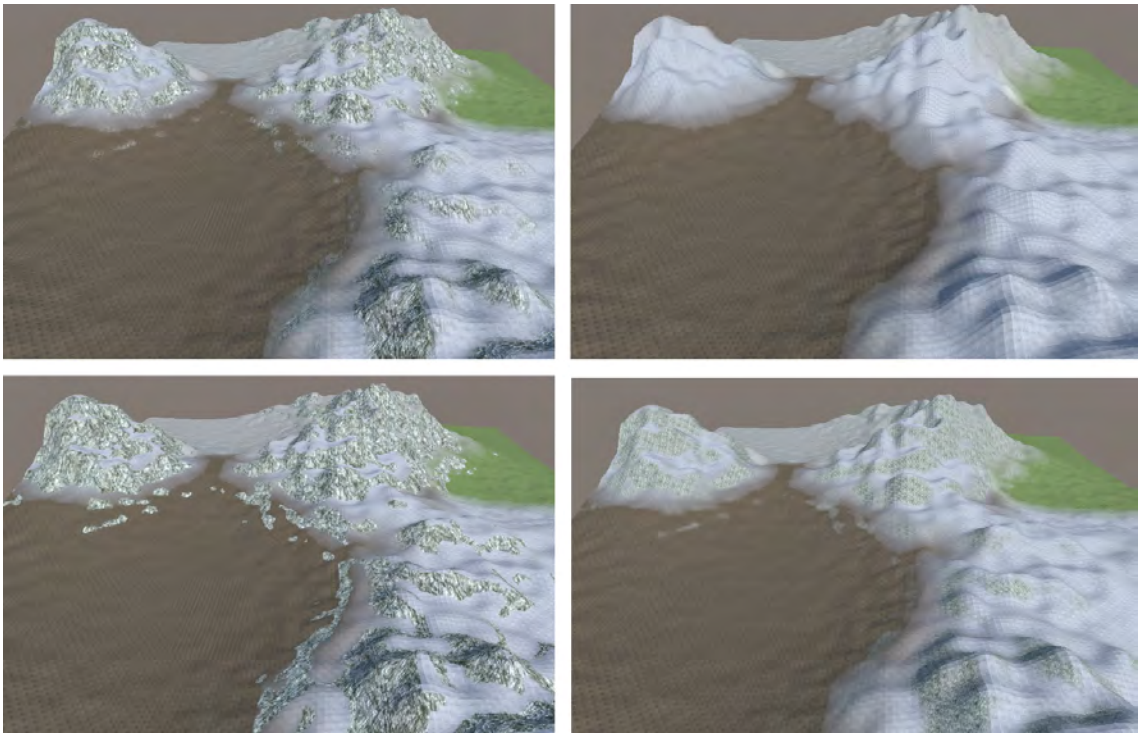

Abbildung 4.14: Vergleich von Steilhängen:
standardmäßig, fehlend, scharfe Übergänge, skalierte Textur
Quelle: Eigene Darstellung

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 Steilhangtextur 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 projiziert 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 projiziert 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 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.

Der Output des Terrain-Generators besteht aus einem landschaftsähnlichen Mesh. Dieses verwendet ein Material, das den benutzerdefinierten TerrainTextureShader 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
- TextureScaling_Street
- TextureScaling_Steepness
- NormalStrength
- SteepnessIntensity
- SteepnessBlending
- TerrainTextures
- TerrainNormalTextures

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

- Splatmap01
- Splatmap02

- Splatmap03
- Steepness-Map
- StreetMap

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 Unterabschnitt 4.2.3 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.
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.

```
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:
Funktion zur Berechnung der Steigung in einem Punkt im Mesh

3. Durch 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, ausgedrückt als Bogenmaß.
4. Die Obergrenze für den Winkel eines Steilhangs beträgt 90 Grad. Als Bogenmaß wären das 1.57 Radiant. 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 Abbildung 4.16 präsentiert.

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

4.3.2 Umsetzung der Shader

Der TerrainTextureShader arbeitet nach einem Pipeline-Prinzip, bei dem jeder Berechnungsschritt auf ähnliche Weise abläuft:

1. Einlesen der bisher berechneten Basistextur.

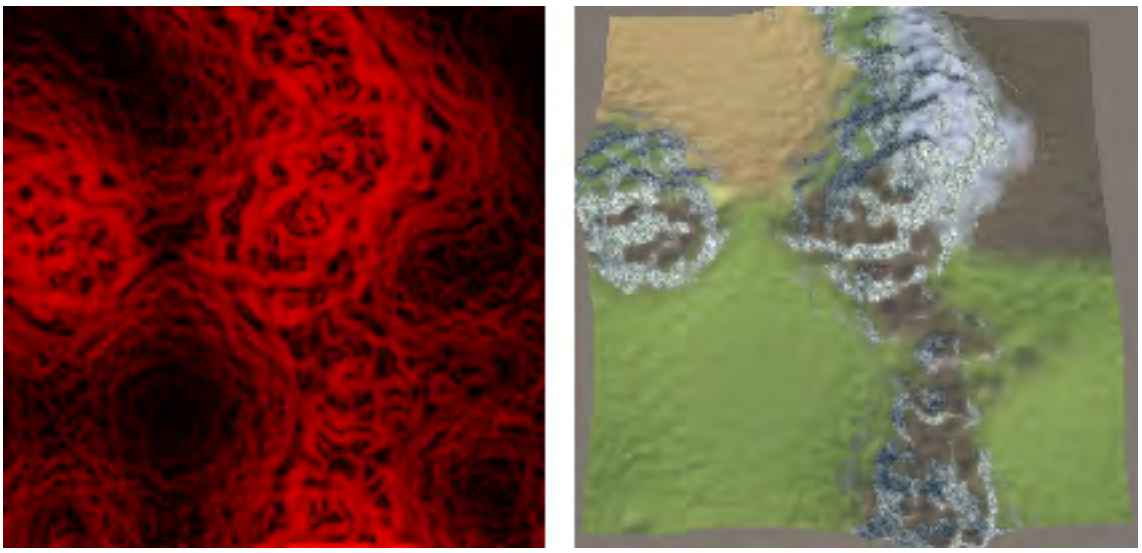

Abbildung 4.16: Vergleich Steepness-Map mit dem zugehörigen Terrain
Quelle: Eigene Darstellung

2. Einlesen der aufzutragenden Textur und ihrer Alpha Map.
3. Lineare Interpolation der aufzutragenden Textur mit der Basistextur über die Alpha Map.

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

Der TerrainTextureShader umfasst mehrere Sub-Shader, die in der folgenden Reihenfolge durchlaufen werden (siehe Abbildung 4.17):

1. Verteilung der Biom-Texturen mithilfe des BiomeTextureSplatter
2. Verteilung der Steilhangtextur mithilfe des SteepnessTextureSplatter
3. Verteilung der Straßentextur mithilfe des StreetTextureSplatter

Diese Textur-Pipeline 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 Sub-Shader des TerrainTextureShader werden im Folgenden näher erläutert:

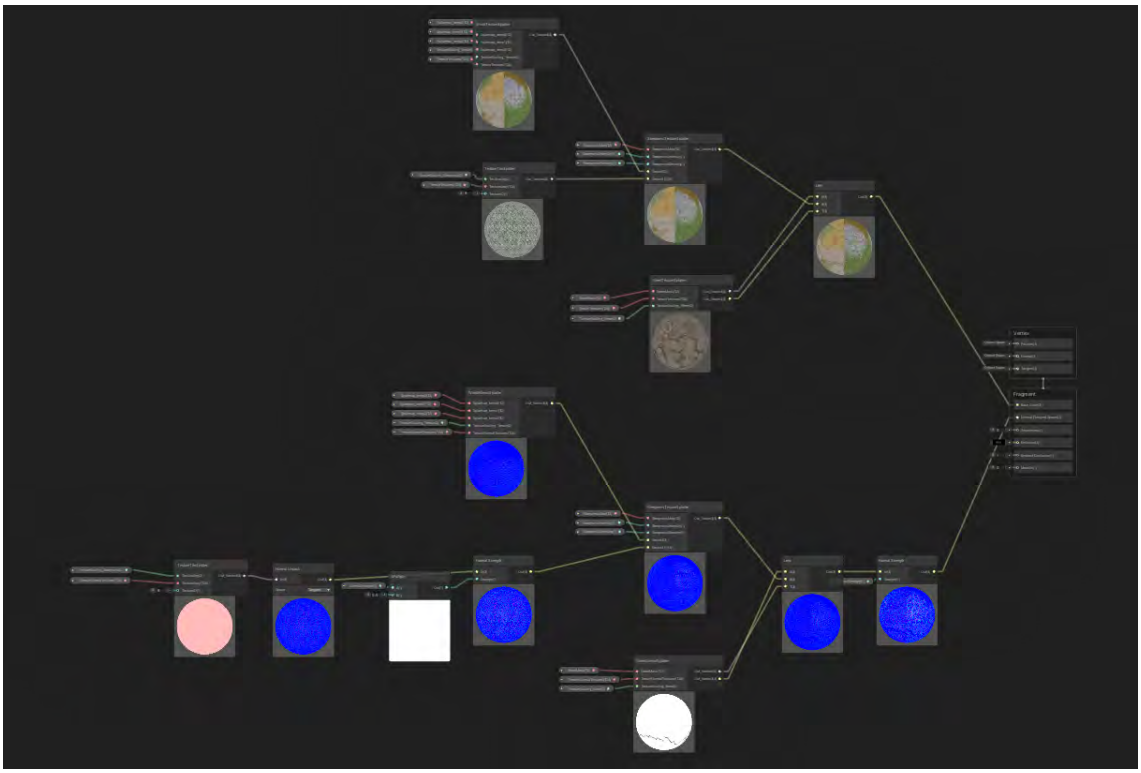

Abbildung 4.17: TerrainTextureShader umgesetzt in Shader-Graph
Quelle: Eigene Darstellung

TextureTileSplitter

Der TextureTileSplitter repräsentiert die Umsetzung der in Unterabschnitt 4.2.3 konzipierten Lösung zur Vermeidung von Tile-Repetition und wird in Abbildung 4.18 dargestellt. Dieser Sub-Shader wird auf sämtliche Tiling-Texturen direkt nach dem Einlesen aus dem Speicher angewendet. Den Input des Shaders bilden der jeweilige TextureScaling-Vektor sowie die zu transformierende Tiling-Textur. Folgende Schritte werden im Shader durchlaufen:

1. **Skalierung der Textur:** Die Texturskalierung wird, wie in Unterabschnitt 4.2.3 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 Unterabschnitt 4.2.3 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 Sub-Shader nutzt den Input der drei zuvor

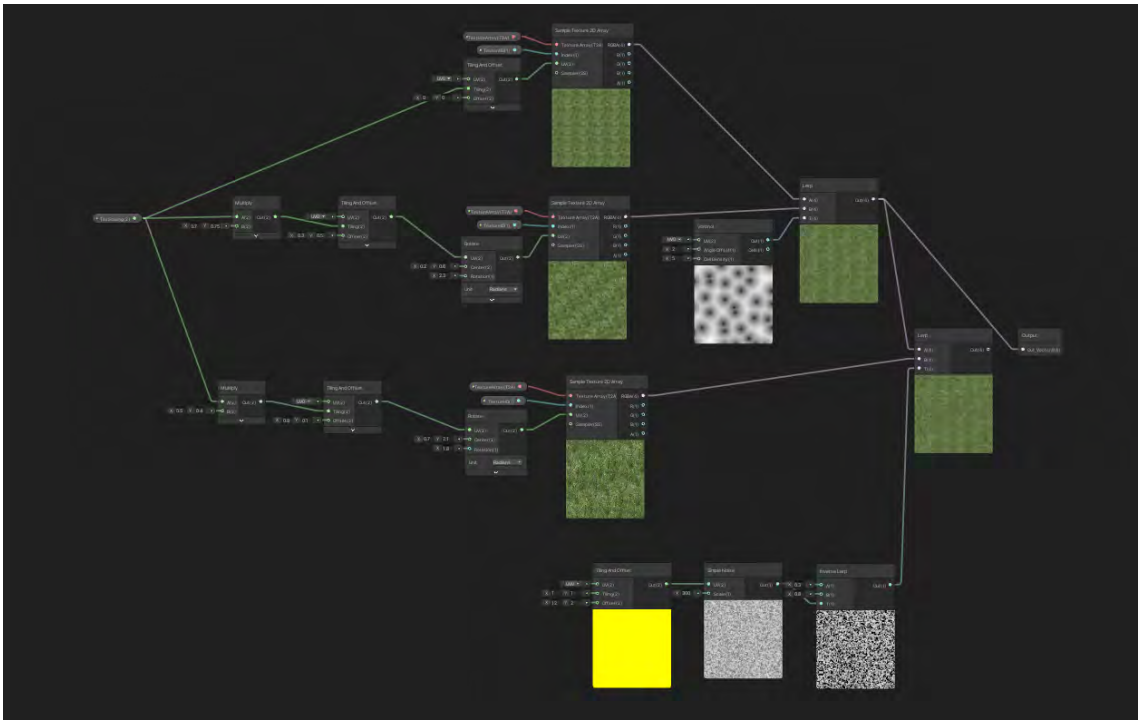

Abbildung 4.18: TextureTileSplitter umgesetzt in Shader-Graph
Quelle: Eigene Darstellung

berechneten Splatmaps, den TextureScaling_Terrain-Parameter und das TerrainTextures-Array, um sämtliche Biome, die vom Biomgenerator erstellt wurden (siehe Abschnitt 2.2), mit Texturinformationen zu versehen. Der Ablauf des BiomeTextureSplatter-Sub-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.
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.

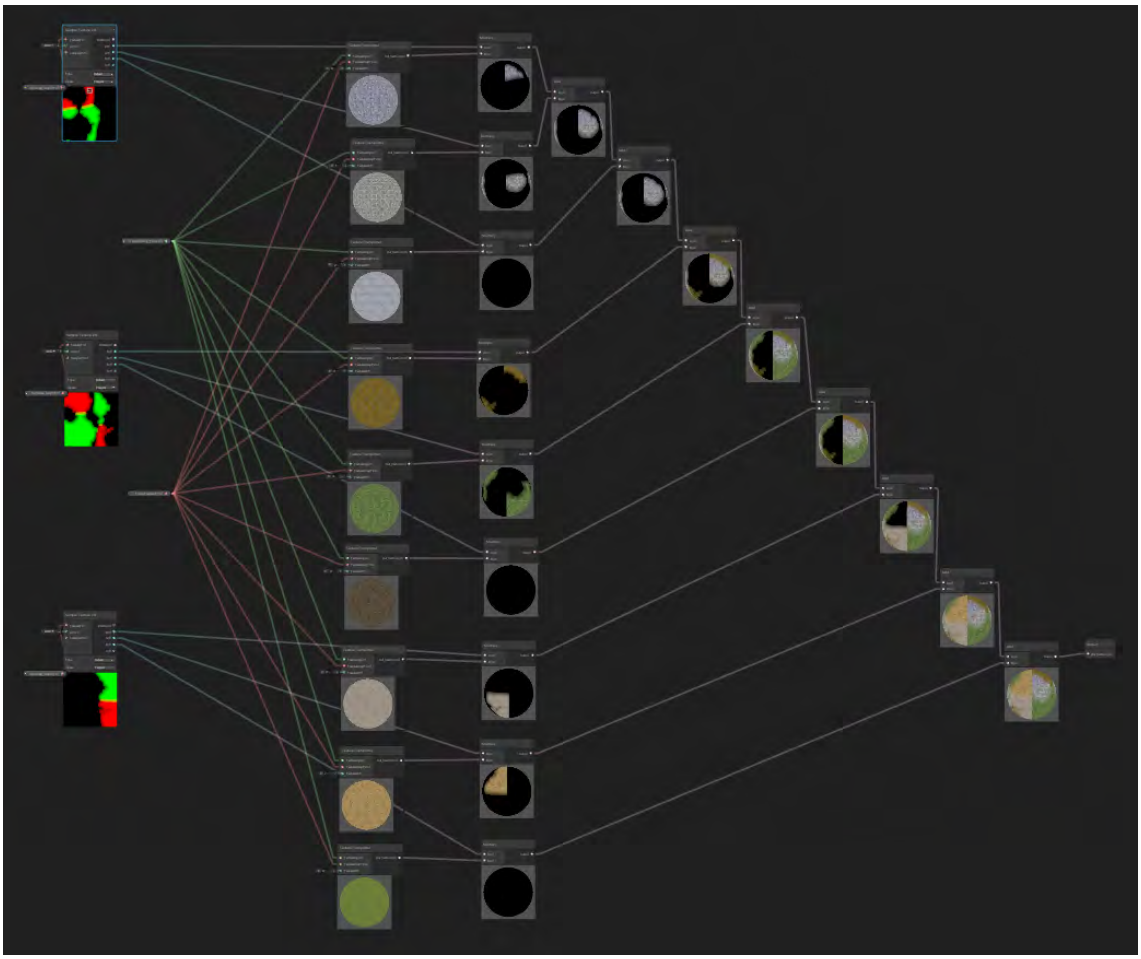

Abbildung 4.19: BiomeTextureSplatter umgesetzt in Shader-Graph
Quelle: Eigene Darstellung

SteepnessTextureSplatter

Der SteepnessTextureSplatter hat einen klaren Fokus: die Erzeugung einer Alpha-Map, die die steilen Bereiche des Terrains abbildet. Dieser Sub-Shader nimmt mehrere Inputs entgegen, darunter die zuvor berechnete Steepness-Map, den Output des BiomeTextureSplatter-Sub-Shaders, die Steilhangtextur, sowie die Parameter TextureScaling_Steepness, SteepnessIntensity und SteepnessBlending. Der Ablauf dieses Sub-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 Steilhangtextur 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 Steilhangtextur:** Nach den vorherigen Bearbeitungsschritten wird die berechnete Alpha-Map als Basis für das Splatting der Steilhangtextur auf die Großtextur verwendet. Vor diesem Schritt erfolgt jedoch noch die Skalierung der Steilhangtextur mithilfe des TextureScaling_Steepness-Parameters.

StreetTextureSplatter

Der StreetTextureSplatter-Sub-Shader fokussiert sich auf das Aufbringen der vom Nutzer ausgewählten Straßentextur auf die zu berechnende Großtextur. Hierfür wird die zuvor vorberechnete StreetMap 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 Rahmen dieser Präsentation 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:

- 100×100 Mesh-Knoten
- 200×200 Mesh-Knoten
- 400×400 Mesh-Knoten

Die Wahl dieser Auflösungen stellt sicher, dass die Anzahl der Knotenpunkte linear um einen Faktor von 4 ansteigt. Neben den gemessenen Werten werden auch sogenannte Zuwachsfaktoren angegeben. Diese Faktoren zeigen an, um welchen Faktor sich die Laufzeit im Vergleich zur vorherigen Messung erhöht hat. Diese Zuwachsfaktoren 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 Unterabschnitt 4.3.1). Die Speicherphase hingegen ist für das Schreiben der Daten der Steepness-Map in den Systemspeicher 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 diskrete Splatmaps mithilfe der Biom-Karte angefertigt, wie in Unterabschnitt 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 Unterabschnitt 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 Abbildung 4.20)
- **Maximale Laufzeitabweichung** als Differenz des höchsten und des geringsten Messwertes (siehe Abbildung 4.20)
- **Durchschnittslaufzeit der Generierungsphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Abbildung 4.21)
- **Durchschnittslaufzeit der Speicherphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Abbildung 4.21)

Erstellung der Splatmaps

- **Durchschnittliche Gesamtlaufzeit** über allen Messwerten (siehe Abbildung 4.22)
- **Maximale Laufzeitabweichung** als Differenz des höchsten und des geringsten Messwertes (siehe Abbildung 4.22)
- **Durchschnittslaufzeit der Initialisierungsphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Abbildung 4.23)
- **Durchschnittslaufzeit der Blending-Phase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Abbildung 4.23)
- **Durchschnittslaufzeit der Speicherphase** als absoluter Wert im Vergleich zur Gesamtlaufzeit (siehe Abbildung 4.23)

	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

Abbildung 4.20: Generierung Steepness-Map in steigenden Auflösungen: Durchschnittliche Laufzeit und maximale Laufzeitabweichung in 10 Durchläufen sowie deren Zuwachsfaktoren
Quelle: Eigene Darstellung

	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

Abbildung 4.21: Generierung Steepness-Map in steigenden Auflösungen: Durchschnittliche Laufzeit der einzelnen Berechnungsphasen in 10 Durchläufen sowie deren Zuwachsfaktoren
Quelle: Eigene Darstellung

	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

Abbildung 4.22: Generierung der 3 Splatmaps in steigenden Auflösungen: Durchschnittliche Laufzeit und maximale Laufzeitabweichung in 10 Durchläufen sowie deren Zuwachsfaktoren
Quelle: Eigene Darstellung

	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

Abbildung 4.23: Generierung der 3 Splatmaps in steigenden Auflösungen: Durchschnittliche Laufzeit der einzelnen Berechnungsphasen in 10 Durchläufen sowie deren Zuwachsfaktoren
Quelle: Eigene Darstellung

5 Vegetationsgenerator

5.1 Theoretische Grundlagen und Verwandte Arbeiten

5.2 Konzept

5.3 Implementierung

5.4 Performance

6 Siedlungsgenerator

6.1 Theoretische Grundlagen und Verwandte Arbeiten

6.2 Konzept

6.3 Implementierung

6.4 Performance

7 Ergebnisse und Evaluation

8 Zusammenfassung und Ausblick

Literatur

[1] Charles Bloom. *Terrain Texture Compositing by Blending in the Frame-Buffer (aka SSplatting"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] Robert L Cook. »Shade trees«. In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 1984, S. 223–231.

[5] Marcus Gagelmann. »Entwicklung eines Biom-Generators für landschaftsähnliche Höhenkarten«. 2023.

[6] Marcus Gagelmann. »Entwicklung eines prozeduralen Landschaftsgenerators«. 2022.

[7] 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.

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

[9] 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.

[10] 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.

[11] Mostafa Hassan. *Proposed workflow for UV mapping and texture painting*. 2016.

[12] Paul S Heckbert. »Fundamentals of texture mapping and image warping«. In: (1989).

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

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

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

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

[17] online-tutorials.net. *Tutorials - Shader Konzept*. http://www.online-tutorials.net/directx/shader-konzept/tutorials-t-7-77.html. [Online; Zugriff am 29.08.2023]. 2014.

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

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

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

[21] Alvy Ray Smith. *Alpha and the history of digital compositing*. Techn. Ber. Citeseer, 1995.

[22] 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.

[23] 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.

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

[25] 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.

[26] Naty Hoffman Tomas Akenine-Möller Eric Haines. *Real-Time Rendering*. 3. Auflage. CRC Press, 2019.

[27] Robert Harding Whittaker u. a. »Communities and ecosystems.« In: *Communities and ecosystems.* (1970).

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

[29] Wikipedia. *Lightmap*. https://en.wikipedia.org/wiki/Lightmap. [Online; Zugriff am 29.08.2023]. 2023.

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