Technische Hochschule
Brandenburg
University of
Applied Sciences

Masterarbeit (Masterthesis)

Parametrisierte Generierung diverser Planeten als Spielwelten für ein interplanetares Aufbaustrategiespiel

Masterarbeit im Fach
Medieninformatik

an der
Technischen Hochschule Brandenburg (THB)

eingereicht bei
Prof. Julia Schnitzer (THB)
und
Hans-Georg Reimer (Beuth Hochschule für Technik Berlin)

vorgelegt von
Sebastian Baier

Bremen, Oktober 2021

Vielen Dank an Franzi für die endlosen Gespräche, konstruktive Kritik und grenzenlose Geduld.

Vielen Dank an Herrn Reimer für die hilfreichen Besprechungen und Denkansätze.

Vielen Dank an Frau Schnitzer für die richtungsweisenden Ratschläge.

In Erinnerung an meine Mum, der ich alles erzählen konnte, selbst wenn sie das Thema nicht verstand.

Kurzfassung

Seit Jahrzehnten nutzen ComputergrafikerInnen prozedurale Methoden wie Simulationen, Noise-Funktionen und Verteilungsalgorithmen, um Terrain und im Speziellen erdähnliche Planeten zu erzeugen. Diese Forschung macht sich in den letzten fünfzehn Jahren auch die Videospielentwicklung verstärkt zunutze, mit dem Ziel spielerisch und visuell interessante Spielwelten zu generieren. Mit prozeduralen Techniken lässt sich eine große Menge Spielinhalte erzeugen und dadurch der Wiederspielwert eines Videospiels und das langfristige Interesse seiner SpielerInnen steigern. Das Ausmaß dieser positiven Auswirkungen hängen dabei stark von der Qualität der generierten Inhalte ab. Bieten sie zu wenig visuelle oder spielerische Vielfalt so schmälert das den Erkundungsdrang der SpielerInnen und ihr langfristiges Interesse am Spiel. Dies gilt insbesondere für Spiele mit Weltraumthematik, die Exploration als zentrales Spielelement beinhalten und Planeten als Spielwelten generieren. Diese Planeten sind im Vergleich zueinander meist sehr vielfältig, aber einzeln betrachtet visuell und spielerisch homogen. Im Rahmen dieser Arbeit werden Methoden zur Generierung in sich und untereinander diverser Planeten als Spielwelten untersucht und ausgewählte Techniken in Form eines Planetengenerators umgesetzt. Der Generator wird exemplarisch für ein interplanetares Aufbaustrategiespiel entwickelt.

Inhaltsverzeichnis

1 Einführung

In den letzten zwei Jahrzehnten hat die prozedurale Generierung von Inhalten bei der Entwicklung von Videospielen stark an Bedeutung gewonnen. Dies liegt zum Einen an der Beliebtheit von Spielen wie dem 2009 veröffentlichten *Minecraft* [S12], deren Spielwelten prozedural generiert werden. Zum Anderen hat sich die Rechenleistung von Heimcomputern und Spielekonsolen in den letzten zwanzig Jahren vervielfacht und somit auch aufwendige prozedurale Techniken relevant für Echtzeitanwendungen wie Videospiele gemacht. Die erste *Xbox*-Konsole von 2001 enthielt einen Intel Prozessor mit einem Kern bei einer Taktrate von 733 MHz [Ah01]. Im Vergleich verfügt die neueste Generation *XBox Series X* von 2020 über einen Prozessor mit acht Kernen bei einer Taktrate von 3,8 GHz sowie eine leistungsstarke Grafikkarte [Microsoft1]. Die zunehmende Rechenleistung von Prozessoren und Grafikkarten sowie die Verfügbarkeit von kostengünstigem Arbeits- und Festplattenspeicher haben zudem zu einem Anstieg der visuellen Komplexität von Videospielen geführt. Entsprechend sind auch die Ansprüche vieler SpielerInnen an die visuelle Darstellung der Spielwelten gestiegen. Prozedural erzeugte Inhalte müssen diesen Ansprüchen gerecht werden. Spiele wie das 2008 erschienene *Spore* [S20] oder *No Man's Sky* [S13] von 2016 versuchen, SpielerInnen mit ihren umfangreichen, prozedural generierten Spielwelten für lange Zeit zu motivieren. Hierbei nimmt die Erkundung der Spielwelt eine zentrale Rolle ein. Eine Vielzahl diverser und interessanter Inhalte, die diesen Erkundungsprozess spannend gestalten, ist somit ein wichtiger Faktor für den Unterhaltungswert dieser Videospiele. Prozedurale Generierung ermöglicht die Erstellung umfangreicher Spielwelten in einem realistischen Zeit- und Kostenrahmen, der bei einer manuellen Erstellung der Inhalte nicht eingehalten werden könnte.

Bei der Generierung prozeduraler Inhalte ist die Schaffung visueller und spielerischer Vielfalt eine Herausforderung. Es besteht die Gefahr, dass sich die generierten Inhalte zu ähnlich sind und vom Spieler nicht als unterschiedlich wahrgenommen werden. Dieses Problem beschreibt Kate Compton [Co17], eine leitende Entwicklerin von *Spore*, als "10,000 bowls of oatmeal"-Problem (dt. 10,000 Schüsseln Haferflocken). Hier zieht Compton den Vergleich mit 10,000 Schüsseln Haferflocken, die mathematisch verschieden sind, jedoch von Betrachtenden visuell nicht als unterschiedlich wahrgenommen werden. Rein mathematische Vielfalt führt somit nicht zwingend zu einem Mehrwert für die SpielerInnen. Am Beispiel von *No Man's Sky* wird dieses Problem deutlich. Zwar sind die generierten Planeten aufgrund unterschiedlicher Texturierung, Terrainstrukturen, Flora, Fauna und Ressourcenvorkommen im Vergleich zueinander sehr unterschiedlich. Doch sind diese Elemente auf der Planetenoberfläche gleichförmig verteilt, was dem Planeten insgesamt ein homogenes Erscheinungsbild und wenig abwechslungsreiches Spielverhalten verleiht. Dadurch wird der Erkundungsanreiz reduziert. Ähnliches lässt sich auch bei *Spore* und anderen Videospielen mit prozedural generierten Planeten wie *Dyson Sphere Program* [S05] oder *Elite Dangerous* [S07] beobachten. Die genannten Beispiele beinhalten zwar der Erde nachempfundene Planten, weisen jedoch nicht deren visuelle Komplexität auf. Dies ist unter anderem im Fehlen unterschiedlicher Biome begründet, welche das Erscheinungsbild der Erde prägen. Auch ist die Verteilung, der Umfang und die Form von visuell auffälligen Strukturen wie Gebirgen und Seen in Realität nicht zufällig, sondern das Resultat geologischer und klimatischer Prozesse.

Das vom Autor entwickelte *Exploit Inc.* möchte hier ansetzen und unter Einbezug der genannten Aspekte visuell und spielerisch spannende Planeten als Teil seiner Spielwelt bieten. *Exploit Inc.* ist das Konzept eines Aufbaustrategiespiel mit Weltraumthematik, dessen Spielwelt aus prozedural generierten Planeten aufgebaut ist. Die SpielerInnen beuten die Ressourcen dieser Planeten aus (engl. exploit) und verarbeiten diese zu weiterführenden Waren. Da jeder Planet nur eine begrenzte Auswahl und Anzahl an Ressourcen bietet, liegt der Fokus des Spiels auf interplanetaren Warenaustausch sowie der Entdeckung und Besiedlung neuer Planeten. Die erzeugten Waren nutzt die SpielerIn, um verschiedene Missionsziele zu erfüllen. Häufige Lieferengpässe führt zum Ende eines Spieldurchlaufs. Scheitern ist dabei fester Bestandteil des Spielkonzepts und soll die SpielerInnen zu einem erneuten Spieldurchlauf anregen. Damit *Exploit Inc.* spannend spannend bleibt, wird die Spielwelt mit jedem Spieldurchlauf neu generiert. Um dabei das „10,000 bowls of oatmeal"-Problem zu vermeiden, sollen die Planeten von *Exploit Inc.* nicht nur im Vergleich zueinander, sondern auch in sich eine visuelle und spielerische Vielfalt bieten.

Ziel dieser Arbeit ist die Entwicklung eines Planetengenerators, dessen Anforderungen aus dem Spielkonzept

von *Exploit Inc.* abgeleitet werden. Der Fokus liegt dabei auf der Erzeugung von erdähnlichen Planeten, da diese im Vergleich zu Gasriesen oder Gesteinsplaneten ohne Atmosphäre, Wasser oder Leben das größte Potential visueller und spielerischer Vielfalt aufweisen. Für die Implementierung des Generators werden unterschiedliche Generierungstechniken evaluiert. Die Computergrafik hat in den letzten Jahrzehnten viele Techniken zur Generierung von Terrains und Planeten hervorgebracht. Die Forschungsfelder umfassen beispielsweise die Simulation von Plattentektonik und Erosion, die prozedurale Generierung diverser Landschaften oder die natürlich wirkende Verteilung von Vegetation auf dem Terrain. Die Generierung innerhalb von Echtzeitanwendungen wie Videospielen steht jedoch bei wenigen Veröffentlichungen im Fokus.

Im ersten Kapitel dieser Arbeit werden die Anforderungen an den Planetengenerator, an den Generierungsprozess und an die erzeugten Planeten definiert. Dabei bildet das Spielkonzept *Exploit Inc.* die Grundlage. Daraufhin werden verschiedene prozedurale Techniken eingeführt, die bei der Entwicklung des Generators zum Einsatz kommen können. Das nächste Kapitel geht auf die Generierung von Terrains im Speziellen ein. Die folgenden zwei Kapitel beschäftigen sich mit der Erzeugung von Kugelmeshes als Grundkörper der Planeten. Dabei wird auf Unterteilungstechniken eingegangen, mit denen feinere Oberflächen erzeugt werden können. Anschließend wird der Planetengenerator mit seiner Generierungspipeline vorgestellt und im folgenden Kapitel auf die definierten Anforderungen geprüft. Im Ausblick werden darauf aufbauend Themengebiete für zukünftige Arbeiten vorgeschlagen.

2 Anforderungen an die Generierung von Planeten als Spielwelten

Die Anforderungen an den Planetengenerator und seine Erzeugnisse leiten sich in erster Linie aus dem Spielkonzept von *Exploit Inc.* sowie aus den dem Spiel zugeordneten Spielegenres ab. Um den Herleitungsprozess nachvollziehbar zu machen, wird im Folgenden das Spielprinzip und Szenario von *Exploit Inc.* erläutert.

Abbildung 2.1: Das Firmenlogo der Exploit Incorporation.

2.1 Exploit Inc.

Exploit Inc. ist der Name für das Konzept eines interplanetaren Aufbaustrategiespiels welches vom Autor im Rahmen des Kurses „Game Design" an der TH Brandenburg im Sommersemester 2020 entwickelt wurde.

2.1.1 Szenario

Das Spiel handelt von einer Welt, in der die stetig wachsenden Anforderungen der Menschen an ihre Lebensqualität, ihre wachsende Anzahl und die zunehmende Lebenserwartung die Kapazitäten der Erde ausgeschöpft haben. Doch waren mit dem Aufschwung der Raumfahrt die ungenutzten Ressourcen des Weltraums in greifbarer Nähe der Menschen und mächtige Firmen entwickelten Raumschiffe, um die Planeten des Universums auszubeuten.

Die *Exploit Incorporation* (siehe Abbildung 2.1) ist die größte Firma der Menschheit, die sich mit der Ausbeutung extraterrestrischen Ressourcen beschäftigt. Sie operiert von der Erde aus und kontrolliert eine Flotte tausender *Exploiter*, die mit dem Abbau dieser Ressourcen beauftragt werden. Die Führungsebene des Unternehmens ist skrupellos und fordert von seinen Untergebenen maximale Hingabe. Wer sie zu oft enttäuscht, wird ausgewechselt und ist fortan in den Weiten des Weltraums auf sich alleine gestellt. Im Gegensatz dazu haben erfolgreiche Exploiter gute Chancen, innerhalb der Firmenhierarchie aufzusteigen und in die Führungsebene aufgenommen zu werden.

2.1.2 Spielmechanik

In *Exploit Inc.* übernehmen die SpielerInnen die Rolle eines niederrangigen Exploiters in der Flotte der Exploit Incorporation. Sie erfüllen immer komplexer werdende Missionen des Unternehmens, gewinnen an Gunst und steigen in der Firmenhierarchie auf. Wird die SpielerIn in die Führungsebene aufgenommen, so gewinnt sie den aktuellen Spieldurchlauf. Misslingen ihr zu viele Missionen, so ist der Spieldurchlauf verloren. Beim Starten eines weiteren Spieldurchlaufs wird eine neue Spielwelt generiert und die SpielerIn fängt erneut als niederrangiger Exploiter an.

Der erfolgreiche Abschluss der Firmen-Missionen ist die Hauptaufgabe der SpielerInnen. Sie werden im Verlauf des Spiels komplexer und schwieriger zu erfüllen. Jede Mission muss in einem begrenzten Zeitfenster abgeschlossen werden. Wird dieses Zeitfenster überschritten, so wird dies als Scheitern gewertet und steigert die Missgunst der Führungsebene. Der erfolgreiche Abschluss einer Mission bringt dem Spieler zusätzliches Budget sowie erhöhte Gunst bei der Exploit Incorporation ein.

Das Missionsziel ist zumeist die Beschaffung einer bestimmten Anzahl Rohstoffe oder Waren. Während die ersten Missionen auf einem einzelnen Planeten erfüllt werden können, erfordern komplexere Missionen die Koordination von Produktions- und Lieferketten auf mehreren Planeten gleichzeitig. Die Ressourcen werden aus den endlichen Vorräten der Planeten gewonnen. Sie umfassen Erze (z.B. Eisen, Gold), Flüssigkeiten (z.B. Wasser, Erdöl), Gase (z.B. Sauerstoff, Wasserstoff), die Flora sowie die Fauna eines Planeten. Zusätzlich kann beispielsweise aus Wind oder Sterneneinstrahlung Energie gewonnen werden.

Die Ausbeutung eines Planeten bleibt nicht ohne Folgen. Baut die SpielerIn beispielsweise die Wasservorräte einem Planeten ab, so verschwinden Flora und Fauna. Nutzt sie einen Planeten als Endlager für Abfallprodukte, die bei der Warenproduktion entstehenden, so wird der Boden und die Atmosphäre des Planeten verseucht. Die SpielerIn muss Kosten und Nutzen ihrer Handlungen stetig abwägen.

Als Avatar der SpielerIn dient ein Raumschiff. Dieses kann unter Einsatz von Waren und Ressourcen mit Verbesserungen ausgestattet werden. Die SpielerInnen müssen abwägen, ob Waren zur Erfüllung der aktuellen Mission oder zum Ausbau des Schiffes eingesetzt werden sollen.

Niederlagen sind bei *Exploit Inc.* Teil des Konzepts. SpielerInnen sollen aus fehlgeschlagenen Versuchen lernen und für nachfolgende Durchläufe besser vorbereitet sein. Mit jedem Spieldurchlauf soll die SpielerIn weiter in der Hierarchie der Exploit Incorporation aufsteigen können. Die prozedural generierten Welten und Missionen haben das Ziel den Wiederspielwert von *Exploit Inc.* erhöhen und jeden Durchlauf interessant gestalten.

Die Planeten in *Exploit Inc.* sind nicht maßstabsgetreu, da ihr Terrain im Vergleich zum Planetendurchmesser deutlich größer ist als bei ihrem realen Vorbild. Dadurch werden der vorhandene Bauplatz und die Ressourcenmenge soweit eingegrenzt, dass die Besiedelung anderen Planeten im Verlauf des Spiels notwendig ist.

2.1.3 Einordnung des Spielkonzepts

Exploit Inc. verbindet Aspekte des *Rogue-like*-Genres mit *Aufbaustrategie* und *Wirtschaftssimulation*. Diese werden im Folgenden vorgestellt.

2.1.3.1 Rogue-likes

Der Name „Rogue-like" leitet sich von dem 1980 veröffentlichten Videospiel *Rogue* [S16] ab. In *Rogue* führt die SpielerIn eine virtuelle Spielfigur durch eine aus ASCII-Zeichen aufgebaute Spielwelt (Abbildung 2.2). Sie durchsucht dabei einen Dungeon mit mehreren Ebenen, die aus unterschiedlichen Räumen aufgebaut sind. In den Räumen bekämpft die Spielfigur Gegner mit steigendem Schwierigkeitsgrad und findet nützliche Gegenstände. Das Spielziel

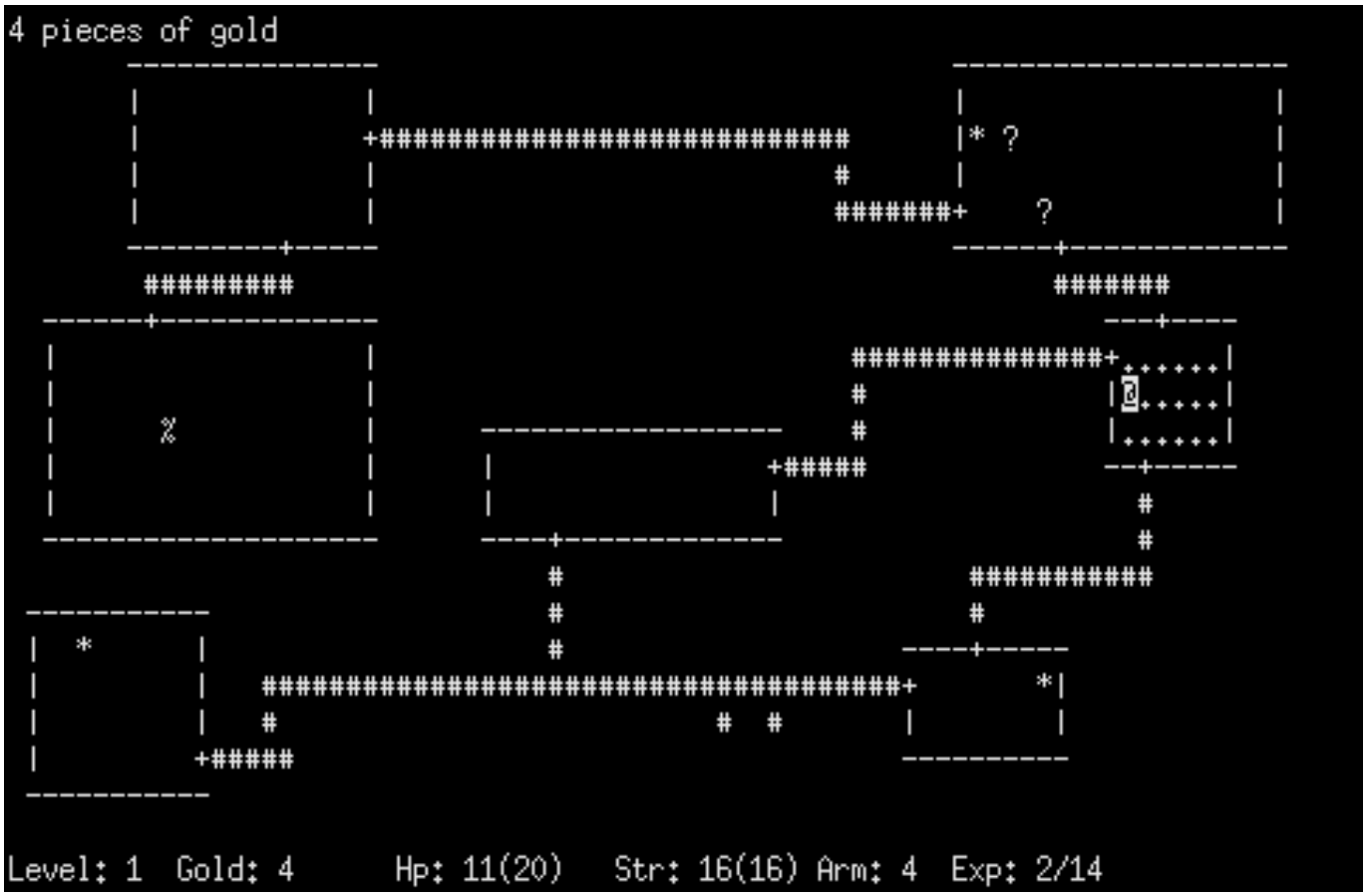


Abbildung 2.2: Das 1980 veröffentlichte Videospiel *Rogue* begründete das Genre der Rogue-like bzw. Rogue-lite Spiele. (Quelle: https://en.wikipedia.org/wiki/Roguelike#/media/File:Rogue_Screenshot.png [Stand: 26. August 2021])

ist das Erreichen der untersten Ebene und der erneute Aufstieg zur Oberfläche. Die Spielwelt von *Rogue* wird mit jedem Spieldurchlauf neu generiert. Der Tod der Spielfigur hat dabei das Ende eines Spieldurchlaufs zu Folge. Ein alter Spielstand kann nicht geladen werden. Spiele mit einem ähnlichen Konzept wie *Rogue* werden "Rogue-like" bzw., falls nur Teilaspekte erfüllt sind, "Rogue-lite" genannt [Cr21]. Vertreter des Genres sind unter anderem *FTL: Faster Than Light* [S09] und *Spelunky* [S19].

Aufgrund des permanenten Todes der Spielfigur ist ein hoher Wiederspielwert bei Rogue-like-Spielen von großer Bedeutung. Dies wird unter anderem mit einer spannenden Spielwelt und einem hohen explorativen Anreiz erreicht. Rogue-like-Spiele werden auf allen gängigen Plattformen veröffentlicht. Dies schließt Spielekonsolen, mobile Geräte und Desktop-Computer ein.

2.1.3.2 Aufbaustrategie und Wirtschaftssimulation

Bei Aufbaustrategie-Spielen errichtet die SpielerIn unter Einsatz von Ressourcen beispielsweise eine Stadt (*City Skylines* [S02]), einen Themenpark (*Planet Coaster* [S14]), einen Zoo (*Planet Zoo* [S15]) oder ein Fabrikgelände (*Factorio* [S08]), um die Ziele des Spiels zu erreichen. Anders als bei *Wirtschaftssimulationen* bzw. *Management*-Spielen liegt dabei der Fokus nicht primär auf wirtschaftlichen Entscheidungen, sondern auf dem Aspekt des Aufbauens und der Verwaltung [USK1, USK2]. Die *Anno*-Spielreihe [S01] vereint Aspekte beider Genres. Deren Spielwelten sind aus einer begrenzten Anzahl Inseln aufgebaut, welche von den SpielerInnen besiedelt werden. Die begrenzten Ressourcen dieser Inseln werden abgebaut, um daraus Baumaterial und Waren zur Versorgung der Inselbevölkerung zu gewinnen. Seltene Ressourcen sind spärlich verteilt. Die Produktionsketten werden mit zunehmendem Spielfortschritt komplexer (Abbildung 2.3). Da nicht jede Insel über die gleichen Ressourcen verfügt, müssen Transportlinien zwischen den Inseln eingerichtet werden. Hier baut sich Spannung mit den anderen Spieler- bzw. Nichtspielerfraktionen auf. Die Fraktionen liefern sich ein Wettrennen um die besten Inseln. Die Verteilung der manuell [Wo18] erstellten Inseln ist mit jedem Spieldurchlauf anders. Damit wird der Wiederspielwert gesteigert. Die SpielerIn kann den Schwierigkeitsgrad

Abbildung 2.3: Der Umfang der Produktionsketten bei Anno 1800 [S01] variiert von einfach (links) bis komplex (rechts). Die Komplexität nimmt im Laufe eines Spieldurlaufs zu.

der Spielwelt vor dem Spielstart mittels Parametern an die eigenen Vorlieben anpassen. Die Zielplattform der *Anno*-Spiele ist wie bei den meisten Aufbaustrategie-Spielen aufgrund seiner komplexen Steuerung der Desktop-Computer.

Sowohl bei Aufbaustrategie-Spielen als auch bei Spielen des Rogue-like-Genres ist eine interessante Spielwelt von großer Bedeutung für ihren Unterhaltungs- und ihren Wiederspielwert. Eine große visuelle Vielfalt steigert zudem den explorativen Reiz.

2.1.3 Zielplattform

Da es sich bei *Exploit Inc.* zu großen Teilen um ein Aufbaustrategiespiel handelt und komplex zu bedienen ist, wird das Spiel für den Desktop PC bzw. der Laptop entwickelt. Die dreidimensionale Darstellung macht zudem den Einsatz einer Grafikkarte erforderlich. Ein Port für Spielekonsolen ist denkbar, liegt aber nicht im Fokus der Entwicklungsarbeit.

2.2 Definition der Anforderungen

Ziel dieser Arbeit ist die Entwicklung eines Generators, der auf Basis sprechender Parameter pseudozufällig diverse und spielerisch interessante Planeten erzeugt. Diese sollen als Spielwelten für *Exploit Inc.* genutzt werden können. In diesem Kapitel werden die Anforderungen an den Generator, den Generierungsprozess und die resultierenden Planeten festgelegt.

2.2.1 Konkretisierung der Begrifflichkeiten

Ein klares Verständnis der Begrifflichkeiten ist für die Definition der Anforderungen notwendig. Im Folgenden werden die Begriffe „sprechende Parameter", „pseudozufällig", „Diversität", „spielerisch interessant" und „Spielwelten" konkretisiert.

Als „sprechende Parameter" werden im Kontext dieser Arbeit Eingabewerte für den Generator bezeichnet, die von fach- bzw. themenfremden EntwicklerInnen nachvollzogen und sinnvoll angepasst werden können. Die Parameter abstrahieren die technische Implementierung, sodass kein Expertenwissen bei der Nutzung des Planetengenerators erforderlich ist. Dies schließt mit ein, dass jede Kombination aus Eingabewerten ein valides Resultat erzeugt. Beispiele für sprechende Parameter sind unter anderem „Radius des Planeten", „Krümmung der Planetenachse" oder „Temperaturspektrum auf der Planetenoberfläche".

Im Gegensatz zu rein zufälliger Generierung ist ein „pseudozufälliger" Generierungsprozess deterministisch und erzeugt somit bei identischen Eingabeparametern das gleiche Resultat. Um dennoch Zufälligkeit zu simulieren, wird zumeist ein beliebig gewählter „Seed" verwendet. Der Seed ist ein zusätzlicher Parameter in Form einer Zahl oder einer Zeichenkette, der entweder manuell oder zufällig gewählt wird. Mit ihm wird der Zufallsgenerator der verwendeten Programmiersprache parametrisiert, sodass dieser Zahlen in festgelegter Reihenfolge generiert. Der gleiche Seed führt dabei zur identischen Reihe Zufallszahlen. Die Zufallszahlen werden anschließend für den weiteren Generierungsprozess verwendet.

„Diversität" bezeichnet die visuelle und auch spielerische Unterschiedlichkeit der generierten Planeten sowohl im Vergleich zueinander als auch für sich betrachtet. Dadurch wird die Erkundung der Spielwelt spannend gestaltet und spielerische Tiefe geschaffen. Diversität entsteht zum Beispiel durch unterschiedliche Biome mit eigener Farbgebung, Flora und Fauna sowie durch unterschiedliche Terrainstrukturen wie Kontinente, Ozeane, Gebirge, Schluchten oder Seen, die durch geologische und klimatische Phänomene geformt werden. Die generierten Planeten stellen keine Visualisierung mit wissenschaftlichem Anspruch dar, sondern sollen sich spielerisch und visuell in *Exploit Inc.* einfügen. Dennoch fließen astronomische Beobachtungen, geologisches Wissen und physikalische Gegebenheiten in den Generierungsprozess ein, um authentische Resultate zu erzielen.

"Spielerisch interessante" Planeten haben eine spannende Ressourcenverteilung und ein Terrain mit begrenztem Bauplatz, der unter Umständen durch Veränderung des Terrains geschaffen werden muss. Eine spannende Ressourcenverteilung bezeichnet beispielsweise die Platzierung von Ressourcen auf der Planetenoberfläche (Flora und Fauna) oder zwischen den Gesteinsschichten (Erze) des Planeten. Da dadurch einige Ressourcen verborgen sind und erst von der SpielerIn entdeckt werden müssen, wird zusätzliche Spieltiefe erzeugt und der explorative Reiz verstärkt. Auch können Ressourcen von Planet zu Planet unterschiedlich und durch Biome regional begrenzt sein, sodass sie die SpielerIn möglichst geschickt zusammenführen muss. Dabei sind die Vorkommen wertvoller Ressourcen kleiner als die von gewöhnlichen Rohstoffen.

Die generierten Planeten sollen als "Spielwelten" in *Exploit Inc.* zum Einsatz kommen. Der Planet ist daher keine reine Kulisse, sondern durch Interaktion der SpielerInnen und durch Spielprozesse veränderbar. Zusätzlich muss die Platzierung von Objekten auf der Planetenoberfläche möglich sein. Zudem soll ein Planet, alle an ihm vorgenommen Veränderungen und die auf Ihm platzierten Objekte im Rahmen eines Spielstand gespeichert und wiederhergestellt werden können.

2.2.2 Die Anforderungen

Auf Basis des Spielkonzepts von *Exploit Inc.* und der Begriffsdefinitionen lassen sich folgende Anforderungen an den Generator, an den Generierungsprozess und die resultierenden Planeten definieren. Die Anforderungen an den Generator sind mit absteigender Priorität folgende:

- Die Bedienung der Generators kann von fach- bzw. themenfremden Entwicklern erfolgen. Technische Details werden abstrahiert.
- Der Generator kann auf einfache Art und Weise in den Entwicklungsprozess eines Spiels, hier im Speziellen *Exploit Inc.*, integriert werden.

Die Anforderungen an den Prozess sind mit absteigender Priorität:

- Die Generierungsprozess erfolgt zur Laufzeit und nicht zur Entwicklungszeit des Spiels.
- Der Generierungsprozess ist deterministisch, sodass Planeten nicht explizit gespeichert werden müssen, sondern bei Bedarf erneut generiert werden können.
- Der Generierungsprozess kann durch sprechende Eingabeparameter gesteuert werden. Jedes Tupel aus Eingabewerten resultiert in einem validen Planeten.
- Die Speicherbelegung und die beanspruchte Rechenleistung des Generierungsprozesses sind auf durchschnittliche bis leistungsstarke Heimrechner mit Grafikkarte ausgelegt.
- Die Dauer des Generierungsprozesses liegt in einem Zeitrahmen, der für die SpielerInnen akzeptabel ist.

Die Anforderungen an die resultierenden Planeten sind mit absteigender Priorität:

- Die Generierung von erdähnlichen Planeten ist möglich.
- Die generierten Planeten haben Eigenschaften und Ressourcen, welche in die Spielmechanik eingebettet werden können. Sie sind durch Interaktionen der SpielerInnen und durch Spielprozesse veränderbar. So können beispielsweise Ressourcen abgebaut, Gebäude errichtet und das Terrain verformt werden.
- Die generierten Planeten sind im Vergleich zueinander visuell divers.
- Die generierten Planeten sind in sich visuell divers.
- Die Oberfläche der generierten Planeten kann verschiedene Biome mit unterschiedlichen Ressourcen aufweisen.
- Die Ressourcen des Planeten werden spielerisch interessant verteilt.
- Die generierten Planeten sind nicht maßstabsgetreu, da ihr Terrain im Vergleich zum Planetendurchmesser deutlich größer ist als bei ihrem realen Vorbild.
- Ein generierter Planet kann mit allen an ihm vorgenommenen Veränderungen im Rahmen eines Spielstands wiederhergestellt werden.

2.3 Prozedurale Problemstellungen bei der Generierung der Planeten

Aus den Anforderungen an den Generator, den Generierungsprozess und dessen Resultate lassen sich folgende Problemstellungen ableiten, die prozedural gelöst werden sollen:

- **Terrain.** Das Terrain der erdähnlichen Planeten muss prozedural generiert werden. Das schließt unter anderem die Erzeugung von Strukturen wie Gebirge oder Schluchten ein. Diese Strukturen müssen so auf dem Planeten verteilt werden, dass ein visuell diverses und spielerisch interessantes Terrain entsteht.
- **Unterteilung in Regionen.** Auch die Unterteilung des Terrains in verschiedene Regionen wie Biome, Kontinente oder Ozeane erfolgt prozedural. Für ein visuell diverses und ansehnliches Resultat sollten die Regionen dabei von unterschiedlicher Größe und Form sein und zudem eine interessante Verteilung aufweisen.
- **Ressourcenverteilung.** Die Ressourcen des Planeten sollen spielerisch interessant auf der Planetenoberfläche oder innerhalb der Hülle des Planeten verteilt werden. Im Falle von Flora und Fauna wird zudem eine natürlich wirkende Verteilung angestrebt. So sollen die Entitäten mit variabler Dichte verteilt werden und nur in bestimmten Biomen auftreten.

Um Lösungen für diese Problemstellungen zu entwickeln, wird in Kapitel 3 ein Überblick zu Methoden prozeduraler Generierung geschaffen.

3 Methoden Prozeduraler Generierung

Um Planeten spielerisch und visuell interessant zu gestalten, können *Methoden der prozeduralen Generierung* eingesetzt werden. Nach Lagae et al. [LLC*10] bezeichnet der Begriff „prozedural" Entitäten, die durch Programmcode und nicht durch Datenstrukturen beschrieben werden. Prozedurale Techniken sind in diesem Zusammenhang Codesegmente oder Algorithmen, die bestimmte Eigenschaften eines computergenerierten Modells oder Effekts spezifizieren.

Methoden prozeduraler Generierung können eingesetzt werden, wenn die manuelle Gestaltung der zu erzeugenden Elemente zu aufwendig oder zu teuer ist. So können Inhalte mit großer Vielfalt mit vergleichbar wenig Aufwand erzeugt werden. Lagae et al. argumentieren, dass auch rechenintensive Algorithmen an Relevanz gewinnen, da Rechenzeit immer günstiger wird. Das ist nach der Meinung der Autoren einer der Gründe, wieso prozedurale Techniken bereits vor zehn Jahren an Bedeutung gewannen.

Heutzutage setzen Videospielentwickler prozedurale Techniken ein, um den Wiederspielwert ihrer Spiele zu steigern und einen großen inhaltlichen Umfang zu generieren. Weitere Einsatzmöglichkeiten für Methoden prozeduraler Generierung nennt Compton [Co17]:

- Generierung von Terrain in Videospielen (z.B. *No Man's Sky* [S13] und *Minecraft* [S12])
- Generierung kompletter Videospiele (z.B. *Ultima Ratio Regum* [S22])
- Analoge Spiele mit generierten Spielregeln (z.B. *Yavalath*)
- Generierte Stoffe oder Mode
- Generierte Inhalte von Twitter Bots

Togelius et al. [TSD16] nennen des weiteren:

- Generierung von Vegetation mit der Middleware *Speedtree* [S18] als Beispiel
- Generierung von Spiellevel-Strukturen
- Generierung von Fraktalen

3.1 Übersicht

Compton [Co17] beschreibt sechs Kategorien, in die sich Methoden prozeduraler Generierung einordnen lassen: "Tiles" (dt. *Kacheln*), *Grammatiken*, *Verteilung*, *Parametrisch*, *Interpretation* und *Simulation*. In der Praxis werden diese Techniken häufig kombiniert, um das gewünschte Ergebnis zu erzielen.

Abbildung 3.1: Die Spielwelten von *Civilization 6* [S17] werden auf Basis von Parametern und einem Seed aus Hexagon-Kacheln zusammengesetzt.

3.1.1 Kacheln

Compton beschreibt *Kacheln* als atomare Elemente, die zu einem größeren Ganzen kombiniert werden. Als Beispiele für den Einsatz von prozeduraler Generierung auf Basis von Kacheln nennt die Autorin die Spielwelten der *Civilization*-Spielereihe [S17], die Dungeons im Videospiel *Diablo* [S03] oder die Level von *Spelunky* [S19]. Nach Compton ist der Einsatz von prozeduraler Generierung auf Basis von Kacheln insbesondere dann sinnvoll, wenn sich das gewünschte Resultat in gleich große Regionen unterteilen lässt und die Platzierung der Kacheln ohne Einschränkungen möglich ist. Ferner sollte durch die Anordnung der Kacheln interessantes Gameplay oder sonstiger Mehrwert entstehen. Die Spielwelten von *Civilization* (Abbildung 3.1) beispielsweise werden mit jedem Spieldurchlauf auf Basis von Parametern und einem Seed neu generiert. Durch die Vielfalt der Hexagon-Kacheln, ihrer Anordnung und den darauf verteilten Ressourcen entsteht eine große spielerische Vielfalt und ein hoher Wiederspielwert.

3.1.2 Grammatiken

Compton stellt Grammatiken als eine weite Kategorie prozeduraler Generierung vor. Ihre Verwendung sei dann sinnvoll, wenn ein Ergebnis rekursiv aus einem anderen erzeugt werden soll. Als Beispiele nennt sie *Lindenmayer-Systeme*, *Template-Systeme* und *Ersatzgrammatiken*, auf die in Kapitel 3.2 näher eingegangen wird.

Die Spielwelt, die Rätsel und die einzelnen Level des Rogue-lite-Spiels *Unexplored* [S23] werden mithilfe einer Grammatik generiert, welche vom leitenden Entwickler Joris Dormans [Do11] entwickelt wurde. Im Sinne des Genre-Vorbilds *Rogue* [S16] versuchen die SpielerInnen von *Unexplored* die unterste Ebene eines prozedural generierten Dungeons zu erreichen. Sie sammeln auf dem Weg Tränke, Zauber und sonstige Ausrüstung, um Gegner zu bekämpfen oder andere Hindernisse wie verschlossene Türen, Giftwolken oder Finsternis zu überwinden. Abbildung 3.2 zeigt den Aufbau einer Ebene eines Dungeons von *Unexplored* mit allen integrierbaren Objekten, Türen sowie Auf-

Abbildung 3.2: Die Levelstruktur von *Unexplored* [S23] wird prozedural mit Hilfe einer Ersatzgrammatik generiert.

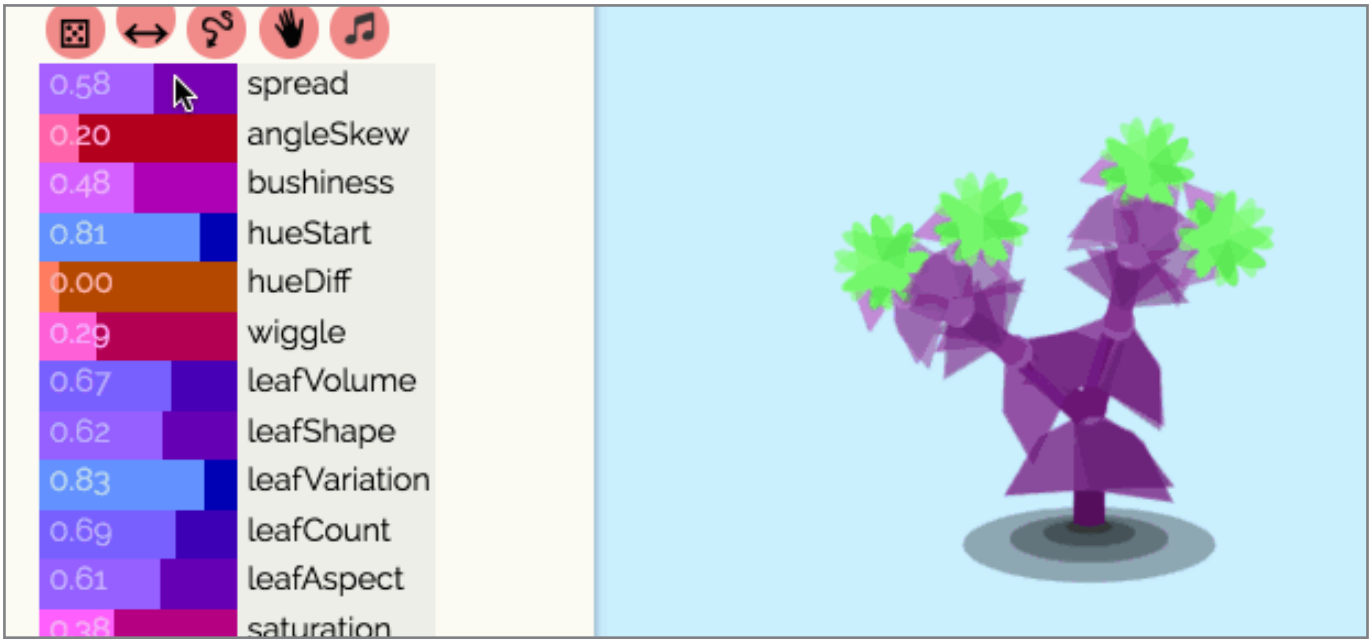


Abbildung 3.3: Ein von Compoton entwickelter Planzengenerator auf Basis von L-Systemen und 32 float-Parametern. (Quelle: https://www.galaxykate.com/img/projects-flowersparam.gif, [Stand: 28 August 2021])

und Abgängen. Die EntwicklerInnen von *Unexplored* haben durch den Einsatz von Grammatiken einen große Spieltiefe sowie hohen Wiederspielwert geschaffen.

3.1.3 Verteilung

Eine weitere von Compton [Co17] genannte Kategorie prozeduraler Generierung ist Verteilung (engl. *Distribution*). Diese Algorithmen und Techniken können beispielsweise verwendet werden, um Objekte in einer Spielwelt zu verteilen. Rein zufällige Verteilung führt dabei selten zu ansprechenden und zufriedenstellenden Ergebnissen, da teils große oder sehr kleine Abstände zwischen den Objekten entstehen. Natürlich wirkende Verteilungen sind nach Compton hierarchisch und gruppiert während die Objekte gleichzeitig einen Mindestabstand zueinander halten. Eine natürlich wirkende Verteilung kann mit unterschiedlichen Algorithmen und Techniken wie *Sample Elimination* oder der *Halton Sequenz* erreicht werden. In Kapitel 3.3 wird näher auf die genannten Algorithmen eingegangen.

3.1.4 Parametrisch

Bei *parametrisierter* prozeduraler Generierung wird eine Reihe von Einstellungen vorgenommen, um ein Artefakt zu erzeugen. Die Werte dieser Einstellungen können nach Compton als Kanten eines n-dimensionalen Würfels modelliert werden, in dem jede Position ein valides Artefakt ergibt. Daher ist es wichtig, die Skalen der Parameter einzuschränken, sodass nur Artefakte entstehen können, die von der AutorIn des Generators als gültig angesehen werden. Diese *Einschränkungen* (engl. Constraints) können auch temporär gesetzt werden, um Artefakte mit einigen bestimmten und unbestimmten Eigenschaften zu erzeugen. Bei einem Planetengenerator könnte beispielsweise die Anzahl der Kontinente und Ozeane festgelegt werden und die anderen Parameterwerte zufällig gewählt werden. Die Art der Parameter ist dabei nicht auf Zahlenwerte beschränkt. Auch andere Datentypen wie Boolean, Enumerations oder Strings sind möglich. Wird ein Seed zur pseudozufälligen Generierung der Artefakte verwendet, so ist dieser ebenfalls Teil der Parametrisierung.

Abbildung 3.3 zeigt beispielhaft einen von Compton entwickelten prozeduralen Pflanzengenerator auf Basis von L-Systemen und 32 Gleitkommazahl-Parametern mit Wertebereichen von 0 bis 1. Jedes Tupel dieser Parameterwerte generiert eine einzigartige virtuelle Pflanze.

3.1.5 Interpretation

Prozedurale Generierung auf Basis von *Interpretation* verarbeitet zuvor erzeugte Daten, um ein gewünschtes Resultat zu erzeugen [Co17]. Die Basis sind meist einfache Datensätze wie eine Verteilung von Punkten, ein Skelett oder eine Kurve, welche dann als komplexere Strukturen interpretiert werden. Hier nennt Compton die Interpretation der Benutzereingabe in *Spore* [S20] als Beispiel. Diese liegen als

Skelett vor und werden als grafische Repräsentation einer Kreatur interpretiert (Abbildung 3.4). Auch die Parameter eines prozeduralen Generators und Grammatiken werden interpretiert, um Artefakte zu erzeugen.

Compton ordnet die Auslegung von *Noise*-Werten und *Voronoi-Diagrammen* ebenfalls in die Kategorie der Interpretation ein. So können beispielsweise die evaluierten Werte von Noisefunktionen als Höhenangaben eines Terrains interpretiert oder eine Fläche durch ein Voronoi-Diagramm in konvexe Bereiche unterteilt werden. Auf Noise bzw. Voronoi-Diagramme wird in Kapitel 3.5 bzw. Kapitel 3.3.7 näher eingegangen.

3.1.6 Simulation

Simulation ist die letzte von Compton genannte Kategorie der prozeduralen Generierung. Diese umfasst unter Anderem die *Simulation von Partikeln* sowie *agenten-* oder *physikbasierte Simulationen*. Diese Simulationstechniken werden in Kapitel 3.4 näher behandelt.

3.2 Grammatiken

Nach Hoffmann [Ho15] ist *Grammatik* ein Begriff aus der Theorie formaler Sprachen. Mithilfe einer Grammatik lassen sich *Wörter* einer *Sprache* erzeugen. Diese *Wörter* werden aus Zeichen eines endlichen *Alphabets* gebildet. Jede Teilmenge dieser Wörter definiert eine *formale Sprache* über dem Alphabet wobei die Grammatik das Regelwerk zur Generierung von Wörtern einer Sprache bildet. Tavelle 3.1 zeigt eine Grammatik, die einen kleinen Auszug aus der deutschen Sprache erzeugt. Mit ihr können Sätze wie „Der flinke Eisbär isst Schokolade" bzw. „Die kleine Kröte mag Kekse" gebildet werden. Die in spitze Klammern gesetzten Platzhalter werden *Nonterminale* und die nicht weiter ersetzbaren Sprachbestandteile *Terminale* genannt. Das Nonterminal „<Satz>" ist in diesem Beispiel das *Startsymbol*, das Symbol mit dem eine Ableitung beginnt.

<Satz>	→	<Subjekt> <Prädikat> <Objekt>
<Subjekt>	→	<Artikel> <Adjektiv> <Substantiv>
<Artikel>	→	Der | Die | Das
<Adjektiv>	→	kleine | süße | flinke
<Substantiv>	→	Eisbär | Elch | Kröte | Maus | Nilpferd
<Prädikat>	→	mag | fängt | isst
<Objekt>	→	Kekse | Schokolade | Käsepizza

Tabelle 3.1: Beispiel einer Grammatik einer formalen Sprache. Die Wörter in spitzen Klammern sind Platzhalter und werden weiter ersetzt. (basierend auf [Ho15], S.163)

Jede Grammatik besteht aus folgenden vier Elementen:

- einer endlichen Variablenmenge (Nonterminale)
- einem endlichen Terminalalphabet
- einer endlichen Menge von Produktionsregeln
- sowie einer Startvariablen

Neben der Erzeugung von Textinhalten können Grammatiken sowohl zur Generierung grafischer Konstrukte als auch zur Definition von Level- und Missionsstrukturen verwendet werden. Togelius et al. [TSD16] unterscheiden zwischen

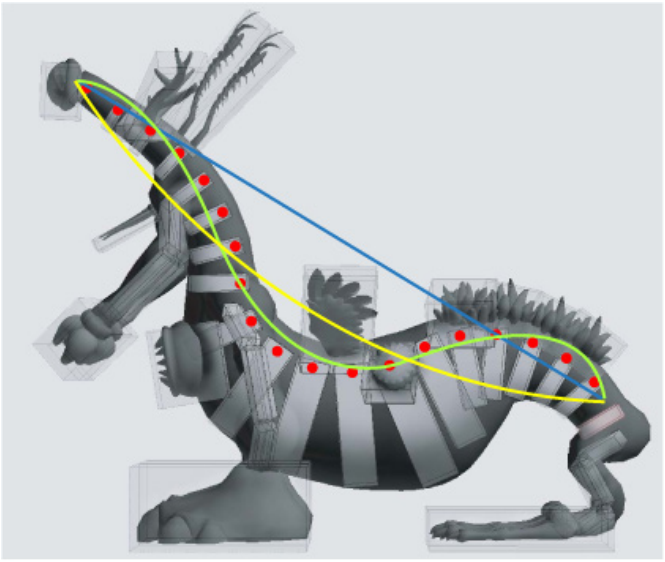


Abbildung 3.4: Das durch die Nutzereingabe generierte Skelett des *Spore*-Kreatureneditors wird grafisch interpretiert. (basierend auf: [HRE*08], S.8)

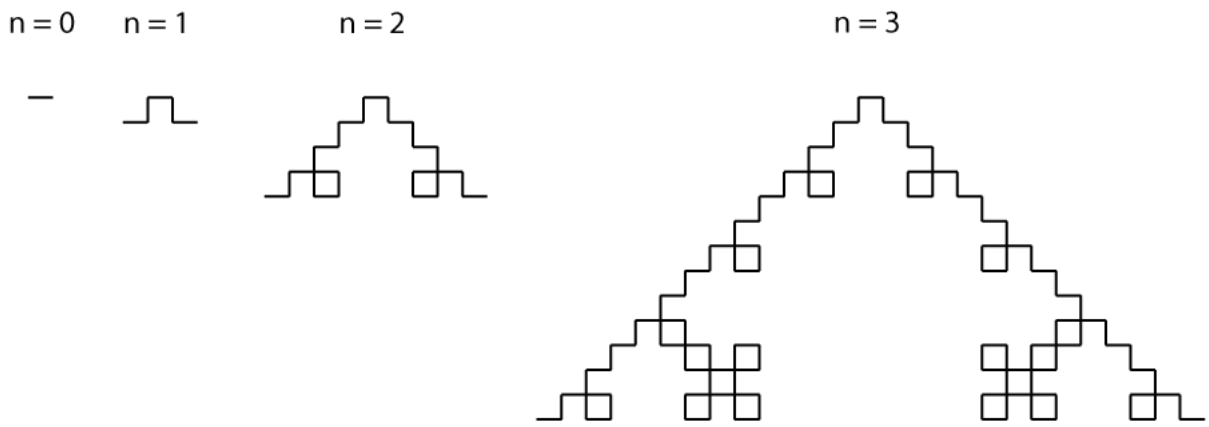


Abbildung 3.5: DIe Kochkurve erzeugt aus dem L-System F → F + F - F - F + F nach 0, 1, 2 und 3 Iterationen. (basierend auf: [TSD16], S.77)

Grammatiken auf String-Basis, zu denen die Lindenmayer- und Template-Systeme gehören, sowie Diagrammgrammatiken (engl. graph grammars) denen die von Compton [Co17] genannten Ersatzgrammatiken zuzuordnen sind.

3.2.1 String-Grammatiken

String-Grammatiken generieren Zeichenketten, die entweder, wie im Fall der *Template-Systeme*, eigenständig Sinn ergeben oder beispielsweise als grafische Strukturen interpretiert werden. Zu letzterer Kategorie zählen die *Lindenmayer-Systeme*.

3.2.1.1 Lindenmayer-Systeme

Lindenmayer- oder verkürzt *L-Systeme* wurden 1968 von Aristid Lindenmayer im Kontext der theoretischen Biologie vorgestellt [Li68]. Heutzutage finden L-Systeme insbesondere in der Computergrafik und Spielentwicklung Anwendung. Togelius et al. [TSD16] beschreiben L-Systeme als formale Grammatiken mit den dazugehörigen Produktionsregeln und einem vordefinierten Alphabet. Eine Zeichenfolge wird dabei unter Anwendung der Produktionsregeln in eine andere Zeichenfolge umgewandelt. Bei L-Systemen werden die Produktionsregeln parallel auf die Eingangs-Zeichenfolge angewendet und das Ergebnis in die resultierende Zeichenfolge geschrieben. Togelius et al. zeigen die Generierung einer Zeichenkette nach acht Iterationen anhand folgendem L-System:

Alphabet:
{A, B}

Startvariable:
A

Produktionsregeln:
A → AB
B → A

Für das Startsymbol "A" ergeben sich die folgenden ersten Iterationen:

0. A
1. AB
2. ABA
3. ABAAB
4. ABAABABA
5. ABAABABAABAAB
6. ABAABABAABAABABAABABA
7. ABAABABAABAABABAABABAABAABABAABAAB

Togelius et al. beschreiben die Möglichkeit, das Resultat von L-Systemen als Zeichenanweisungen zu interpretieren. Diese können sowohl im zweidimensionalen Raum als auch in der dritten Dimension umgesetzt werden. Die AutorInnen demonstrieren die visuelle Interpretation mit Hilfe folgender Produktionsregel:

F → F + F - F - F + F

Die Zeichen des Alphabets werden dabei wie folgt interpretiert:

- F: Bewege dich eine gewisse Distanz nach vorne
- +: Drehung um 90° nach links
- -: Drehung um 90° nach rechts

Wie Abbildung 3.5 zeigt, ergibt die grafische Interpretation der resultierenden Zeichenkette die Kochkurve.

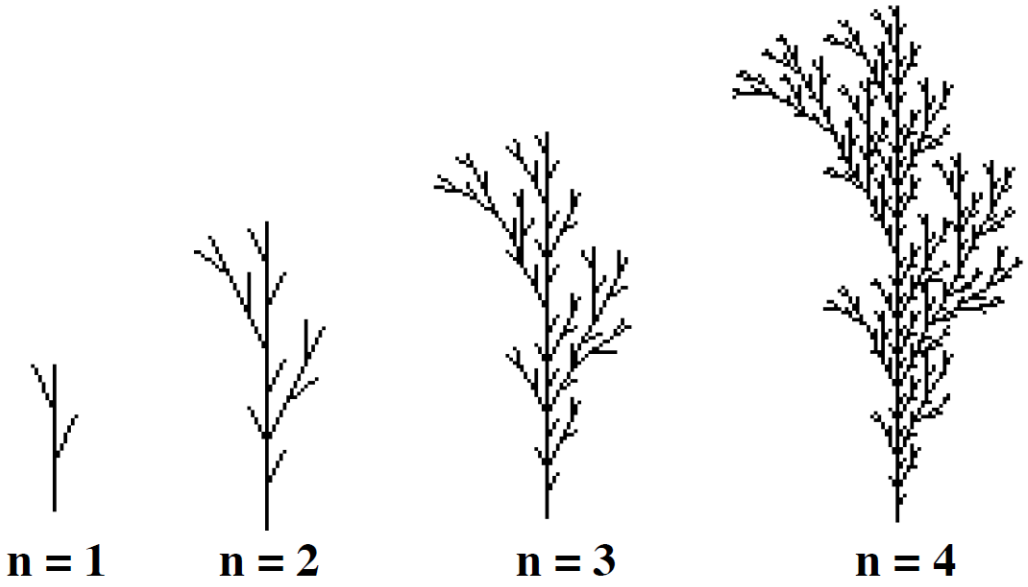


Abbildung 3.6: Die ersten vier Iterationen des eingeklammerten L-Systems F → F [- F] F [+ F] [F]. (Quelle: [TSD16], S.78)

Einfache L-Systeme ermöglichen das Zeichnen von Strukturen, die nur aus einer kontinuierlichen Linie bestehen. Bei komplexeren Strukturen müssen *eingeklammerte* (engl. bracketed) *L-Systeme* eingesetzt werden [TSD16]. Abbildung 3.6 zeigt die grafische Repräsentation von vier Entwicklungen eines eingeklammerten L-Systems. Bei dieser Art von Systemen kommen zwei zusätzliche Symbole „[" und „]" zum Einsatz. Sie fungieren bei der grafischen Interpretation der resultierenden Zeichenkette als „push"- und „pop"-Anweisungen für einen Stack. „[" speichert die aktuelle Position und Orientierung auf den Stack, wohingegen „]" die letzten gespeicherten Daten vom Stack nimmt. An dieser Position wird folglich weitergezeichnet.

Wie die Autoren mit den Abbildungen 3.7 und 3.8 zeigen, können durch einfache oder eingeklammerte L-Systeme komplexe Strukturen wie Fraktale und pflanzenähnliche Gebilde generiert werden.

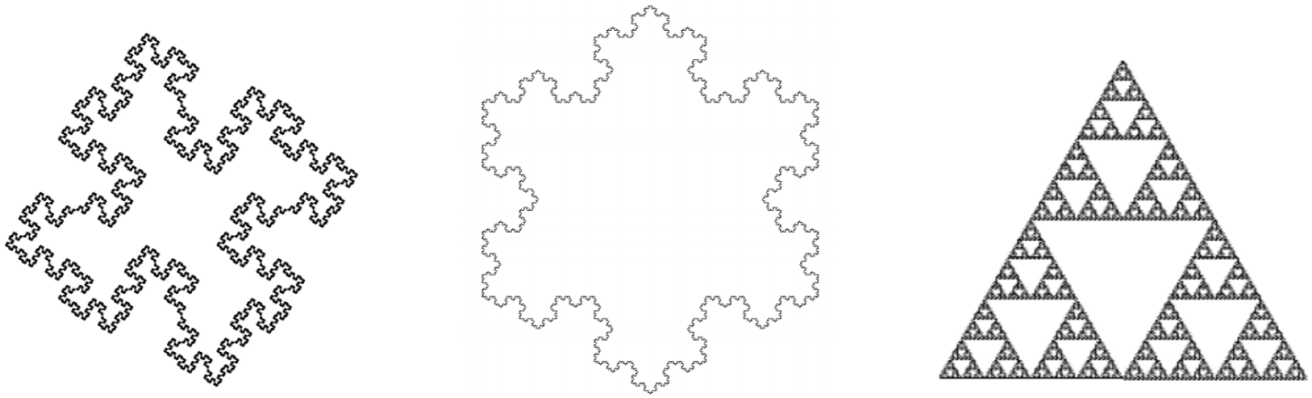


Abbildung 3.7: Mit einem L-System unter Verwendung verschiedener Produktionsregeln generierte Fraktale. (Quelle: [TSD16], S.97)

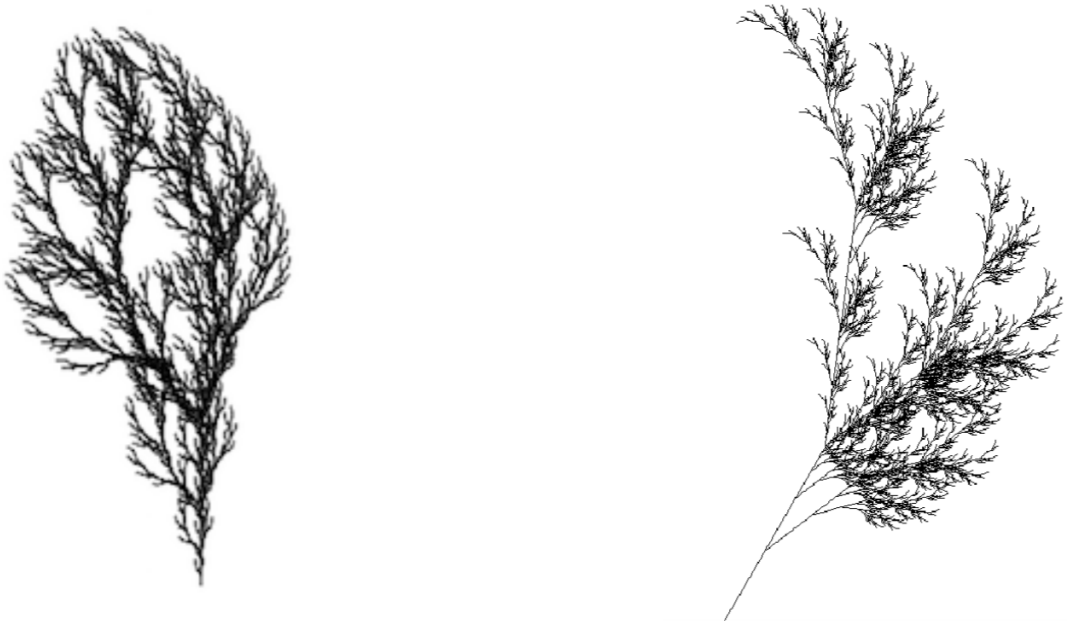


Abbildung 3.8: Pflanzenähnliche Strukturen, die mit einem L-System unter Verwendung verschiedener Instanziierungsparameter generiert wurden. (Quelle: [TSD16], S.96)

Trotz der Vielfalt an Strukturen, die mit den beschriebenen Varianten des L-Systems erzeugt werden können, ist ihre Modellierkraft beschränkt. So können kontinuierliche Phänomene wie Wachstum nur sehr schwer dargestellt werden. Als Lösung stellen Prusinkiewicz und Lindenmayer [PL90] *parametrisierte L-Systeme* vor. Diese operieren auf parametrisierten Wörtern, die aus Zeichen und mit ihnen assoziierten Parametern aufgebaut sind. Bei der Definition der Produktionsregeln können *arithmetische* (+, -, *, /), *relationale* (<, >, =) und *logische Operatoren* (!, &, |) genutzt werden, um den Wert eines Parameters mit jeder Iteration zu beeinflussen. Zudem können *Bedingungen* definiert werden, unter welchen eine Produktionsregel angewendet wird. Folgende Regel wird beispielsweise erst nach der 5. Iteration des L-Systems auf A angewendet. Die Zeitvariable t ermöglicht dabei die Simulation einer Modellstruktur über Zeit:

$$A(t) \ : \ t > 5 \rightarrow B(t+1)CD(t \wedge 0.5, t-2)$$

Bei der visuellen Interpretation einer Zeichenkette werden die Parameterwerte beispielsweise als Strichlänge oder Rotationswinkel interpretiert. Dies wird in Abbildung 3.9 deutlich. Der „Stamm" und die „Äste" der baumähnlichen Struktur verlängern sich mit jeder Iteration und simulieren dadurch Wachstum.

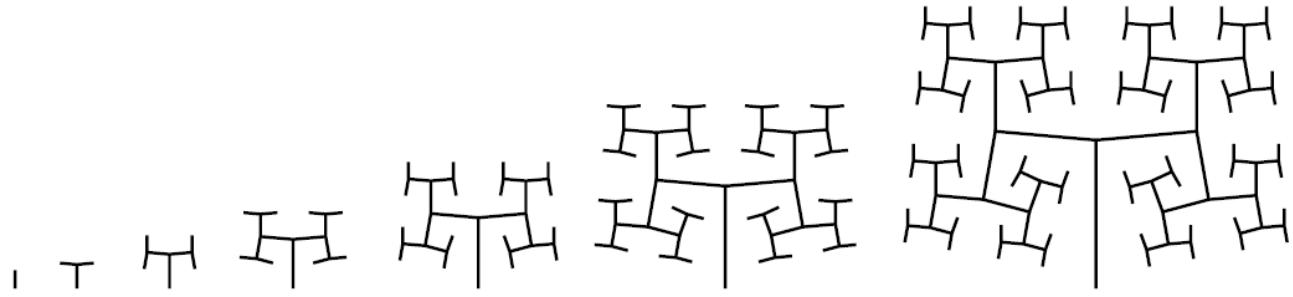


Abbildung 3.9: Die ersten Iterationen eines parametrisierten und eingeklammerten L-Systems. (Quelle: [PL90] S.50)

Das entsprechende L-System mit $R = \sqrt{2}$ ist wie folgt definiert:

Startwort ω:
A

Produktionsregeln:
$p_1 : A \quad \rightarrow F(1)[+A][-A]$
$p_2 : F(s) \rightarrow F(s*R)$

3.2.1.2 Template-Systeme

Anders als L-Systeme ermöglichen *Template-Systeme* die rekursive Generierung von Textinhalten, angefangen bei einzelnen Namen von Charakteren bis hin zu kompletten Geschichten. Sie richten sich dabei stark an die eingangs beschriebene Definition der Grammatik einer formalen Sprache. Allerdings werden die Elemente auf der rechten Seite einer Produktionsregel zufällig bestimmt. Dies zeigt auch der von Compton und Mitautoren [CKM15] entwickelte Webservice *Tracery* [S21]. Die Grammatik des Template-Systems wird in einer einfachen und lesbaren Syntax as JSON-Objekt geschrieben und von Tracery rekursiv zum finalen Text erweitert. Die Grammatik besteht aus einer Liste von Symbolen und Regeln zur Umschreibung. Fortgeschrittene NutzerInnen können neben erweiterbaren Symbolen die Persistenz von Textbausteinen steuern oder Features wie push-pop-Aktionen sowie Modifier wie „pluralize" bzw. „capitalize" nutzen. *Pluralize* bildet dabei den Plural eines Wortes und *capitalize* verändert das erste Zeichen eines Wortes zu seinem groß geschriebenden Pendant. Abbildung 3.10 zeigt einen einfachen Beispielsatz, der mit Tracery generiert wurde. Die dazugehörige Grammatik lautet:

```
{
    „origin“: [„[myPlace:#path#]#line#“],
    „line“: [„#mood.capitalize# and #mood#, the #myPlace#
    was #mood# with #substance#“,„#nearby.capitalize#
    #myPlace.a# #move.ed# through the #path#, filling me
    with #substance#“],
    „nearby“: [„beyond the #path#“,„far away“,„ahead“,„be-
    hind me“],
    „substance“: [„light“,„reflections“,„mist“,„shadow“,„dar-
    kness“,„brightness“,„gaiety“,„merriment“],
    „mood“: [„overcast“,„alight“,„clear“,„darkened“,„blue“,
    „shadowed“,„illuminated“,„silver“,„cool“,„warm“,„sum-
    mer-warmed“],
    „path“: [„stream“,„brook“,„path“,„ravine“,„forest“,„fence“,
    „stone wall“],
    „move“: [„spiral“,„twirl“,„curl“,„dance“,„twine“,„weave“,
    „meander“,„wander“,„flow“]
}
```

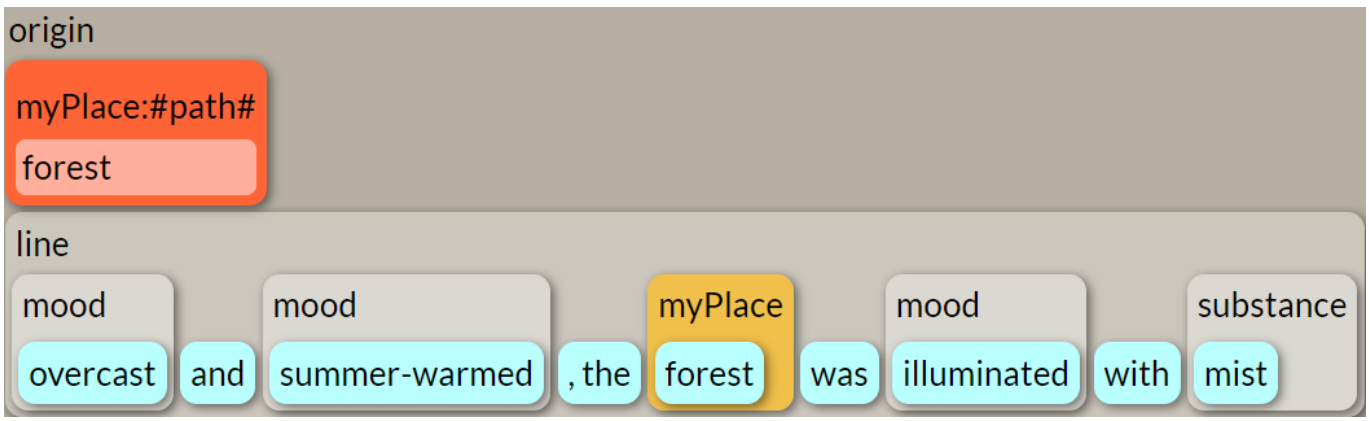


Abbildung 3.10: Ein mit dem Webservice *Tracery* [S21] generierter Satz.

Abbildung 3.11: Eine Diagrammgrammatik-Regel. (basieren auf [TSD16], S.81)

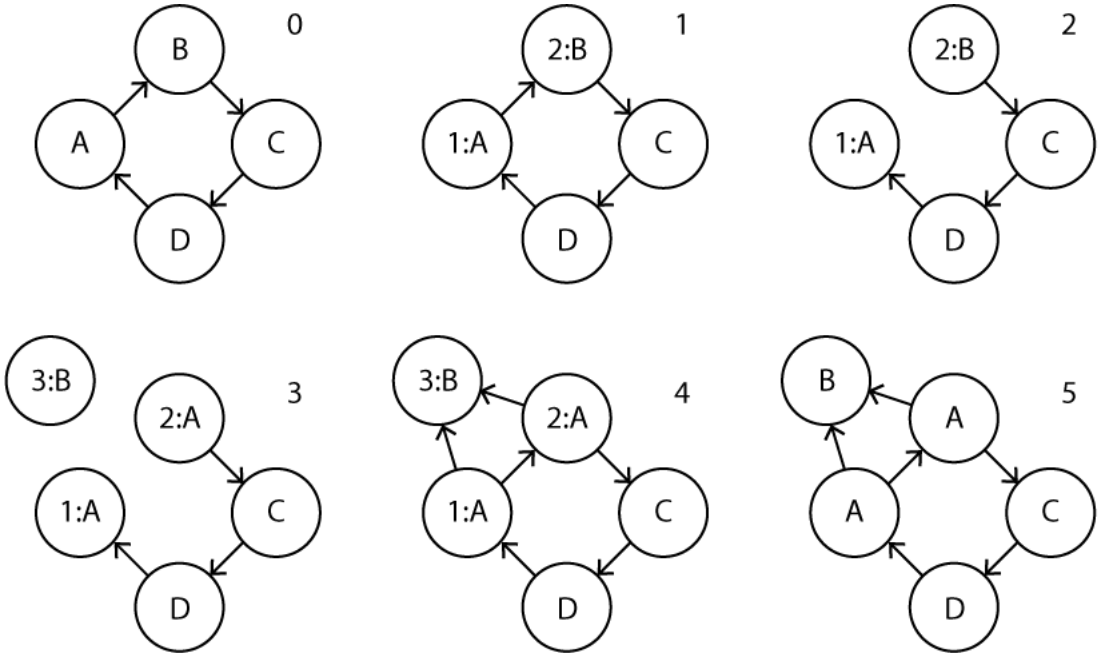


Abbildung 3.12: Eine Diagrammgrammatik Transformation. (basieren auf [TSD16], S.81)

3.2.2 Diagrammgrammatik

Togelius et al. [TSD16] merken an, dass generative Grammatiken neben Strings auch auf unterschiedliche Strukturtypen wie Graphen, aus Kacheln aufgebaute Karten oder zwei- bzw. dreidimensionale Formen wirken können. Diese Art von Strukturen seien sinnvoll, um Spielmissionen und Spielräume sowie, in Kombination, Spielelevels darzustellen. *Diagrammgrammatiken* funktionieren ähnlich wie String-Grammatiken. Die linke Seite ihrer Regeln identifiziert eine bestimmten Teilgraphen, der durch eine der Strukturen im rechten Teil der Regel ersetzt werden kann. Für die Transformation ist es jedoch wichtig, jeden Knoten auf der linken Seite einzeln zu identifizieren und mit einzelnen Knoten in jedem rechten Teil abzugleichen. Abbildung 3.11 zeigt eine Produktionsregel, die in Abbildung 3.12 angewendet wird. Togelius et al. nennen folgende Schritte im Transformationsablauf:

1. Einen Untergraphen im Zielgraphen suchen, der dem linken Teil der Regel entspricht. Anschließend durch Kopieren der Knoten-Bezeichner diesen Untergraphen markieren.
2. Alle Kanten zwischen den markierten Knoten entfernen.
3. Den Graphen durch Umwandeln der markierten Knoten in ihre entsprechenden Knoten auf der rechten Seite der Regel transformieren. Einen Knoten für jeden Knoten auf der rechten Seite hinzufügen, der keine Übereinstimmung im Zielgraphen hat. Anschließend alle Knoten entfernen, die keinen entsprechenden Knoten auf der rechte Seite haben.
4. Die Kanten wie auf der rechten Seite angegeben kopieren.
5. Alle Markierungen entfernen.

Die von Compton [Co17] genannten *Ersatzgrammatiken* fallen unter die Kategorie der Diagrammgrammatiken. Sie nennt das in Kapitel 3.1.2 vorgestellte Spiel *Unexplored* [S23] des Entwicklerstudios *Ludomotion* als Beispiel für diese Technik. Diese basiere auf der Arbeit von Joris Dormans [Do11], dem Mitbegründer von *Ludomotion* und Mitautor von Togelius et al. [TSD16]. Dormans entwickelte sogenannte "Rewrite Systems" und ein dazugehöriges Programm zur formalen Beschreibung und rekursiven Generierung von Spiellevels. Auf dieser Basis werden laut Compton die Spielwelten von *Unexplored* generiert. Abbildung 3.13 zeigt unterschiedliche Produktionsregeln, um Schlösser und Schlüssel in die Levelstruktur zu integrieren. Die folgende Legende erklärt dabei die Bedeutung der einzelnen Knotenpunkte:

- T: „Task" (Aufgabe)
- L: „Lock" (Schloss)
- K: „Key" (Schlüssel)
- ?: „Wild Card" (Platzhalter/Nonterminale)

3.3 Verteilungstechniken

Ziel von *Verteilungstechniken* ist die Platzierung von Entitäten innerhalb eines festgelegten Wertebereichs wie einer zweidimensionale Fläche oder einem dreidimensionalen Raum. Die Positionen der Entitäten werden dabei *Samples* [Co86, Br07, DH06] und der Wertebereich *Sampling-Domäne* [Yu15] genannt. Im Kontext von Videospielen wird meist versucht, eine natürlich wirkende bzw. spielerisch interessante Verteilung von Objekten wie Bäumen oder Ressourcen zu erreichen. Die Verteilung erfolgt häufig manuell während des Entwicklungsprozesses; bei prozedural generierten Welten hingegen muss auch die Verteilung von Objekten prozedural erfolgen.

3.3.1 Pseudozufälliges Sampling

Beim *pseudozufälligen Sampling* (Abbildung 3.14) einer bestimmten Menge von Punkten innerhalb der Sampling-Domäne wird die Position jedes Samples pseudozufällig bestimmt. Die entstehende Verteilung der Samples wirkt allerdings nicht natürlich, da die Samples Cluster bilden

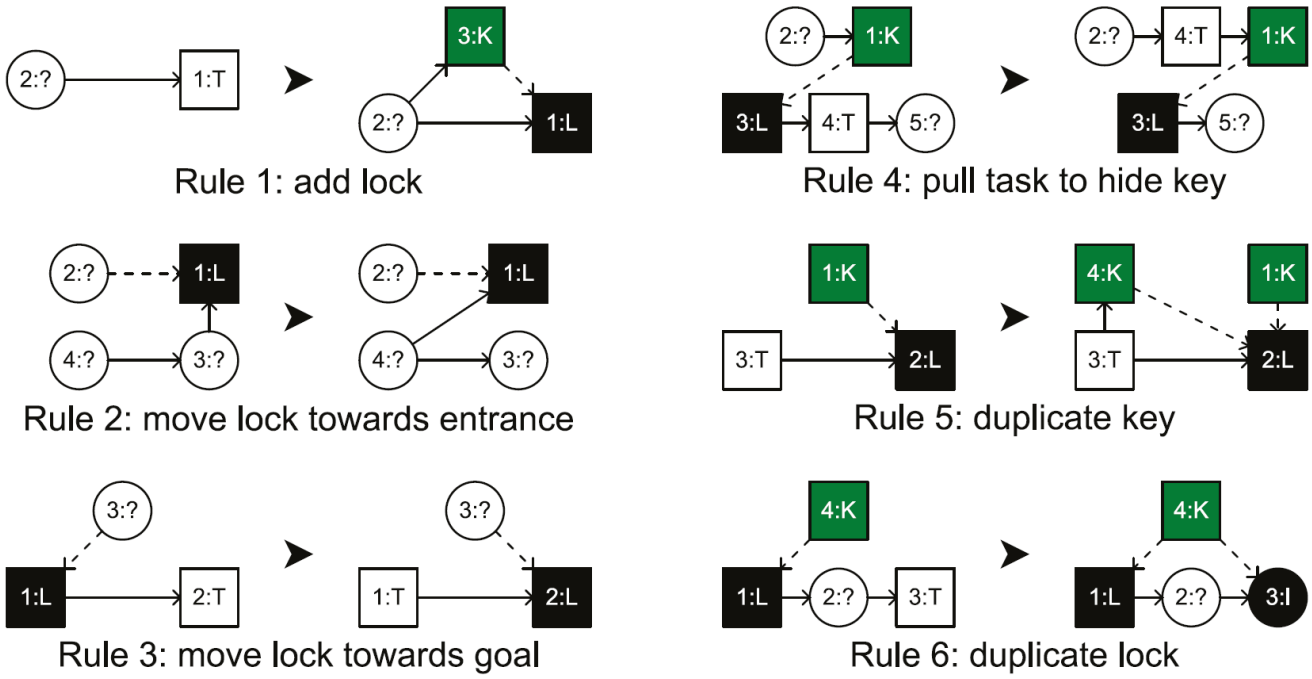


Abbildung 3.13: Ersatzregeln, welche die Transformationen bestimmen, die durch die Verwendung von Schlössern und Schlüsseln ermöglicht werden. (Quelle: [Do11], S.4)

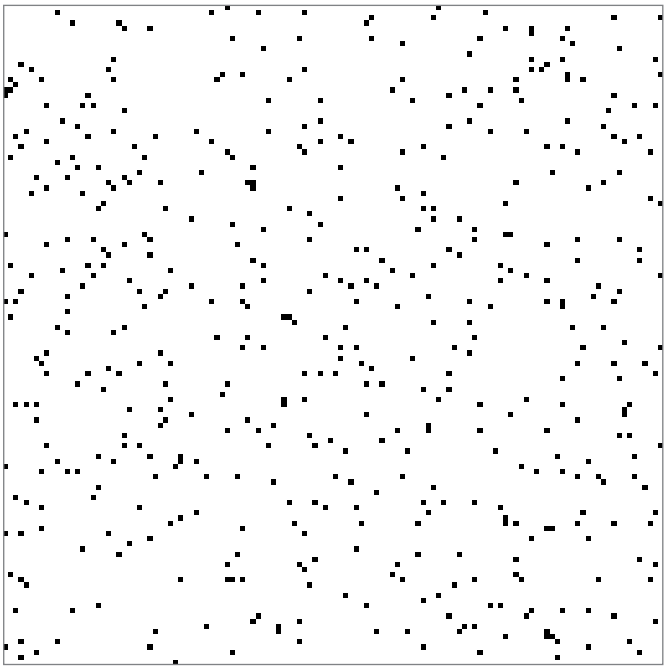


Abbildung 3.14: 500 pseudozufällig verteilte Samplepunkte auf einer 128x128 Pixel Textur.

oder sogar die gleiche Position innerhalb der Domäne einnehmen. Auch bilden sich durch die ungleichmäßige Abdeckung der Sample-Domäne viele Lücken unterschiedlicher Größe.

3.3.2 Gleichmäßiges Raster

Ungleichmäßige Lücken können bei einem *gleichmäßigen Raster* nicht entstehen. Bei dieser Verteilungstechnik wird die Sampling-Domäne durch ein gleichmäßiges Raster geteilt, wobei die Rasterschnittpunkte die Sampleposition bilden (Abbildung 3.15). Dadurch haben alle Samples den gleichen Abstand zueinander. Allerdings wirkt auch diese Verteilung aufgrund der Gleichmäßigkeit und fehlenden Zufälligkeit nicht natürlich. Zudem lässt sich mit einem gleichmäßigen Raster keine ungerade Anzahl Samples verteilen, ohne Lücken in der Sampling-Domäne zu erzeugen.

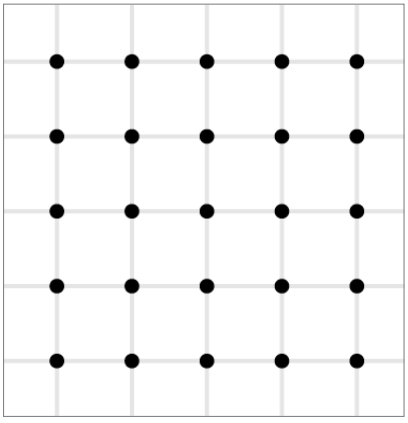


Abbildung 3.15: Ein zweidimensionales reguläres Raster über die Sampling-Domäne. Die Rasterschnittpunkte bilden die Samplepunkte.

3.3.3 Jittered Grid

Wie Cook [Co86] beschreibt, wird die Gleichmäßigkeit eines gleichmäßigen Rasters beim *Jittered Grid* (dt. aufgeregtes Raster) durch das Hinzufügen von *Noise* (dt. Rauschen) aufgebrochen, auf welches in Kapitel 3.5 näher eingegangen wird. Jeder Sampleposition wird ein pseudozufällig gewählter *Offset* hinzugefügt (Abbildung 3.16). Der maximale Offset wird so gewählt, dass sich die Varianzbereiche der einzelnen Samples nicht überschneiden.

Wie Abbildung 3.17 zeigt, wirkt die Verteilung von *Jittered Samples* zufällig, es treten keine großen Lücken zwischen ihnen auf und es lassen sich auch keine Raster Strukturen mehr erkennen. Jedoch kommen sich die Samples aufgrund eines fehlenden Mindestabstands zum Teil sehr nahe. Dieser Abstand wird beim Poisson Disk Sampling garantiert.

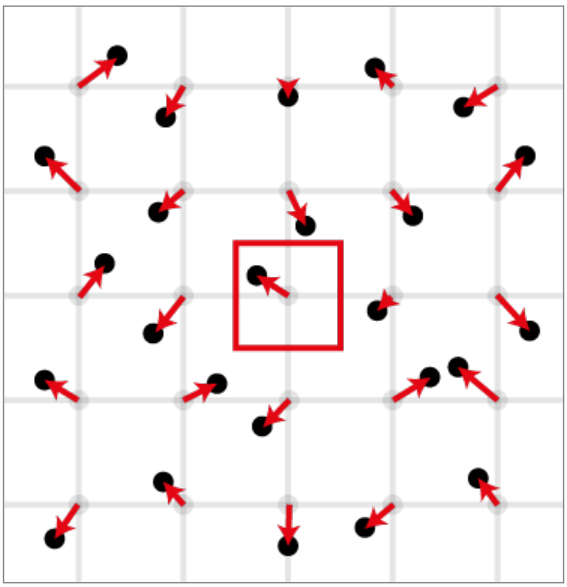


Abbildung 3.16: Beim Jittered Grid wird auf die Samples eines gleichmäßigen Rasters ein Offset gerechnet. Der rote Bereich stellt den Varianzbereich des mittleren Samples dar.

3.3.4 Poisson Disk Sampling

Poisson Disk Sampling ist eine von Cook [Co86] beschriebene Technik zur Erzeugung einer homogenen Verteilung der Samples. Diese Homogenität entsteht durch die *Blue Noise* Eigenschaften der entstehenden *Poisson Disk Verteilung*.

Lau et al. [LUA03] beschreiben *Blue Noise* als ein statistisches Modell, das die idealen räumlichen und spektralen Eigenschaften von verteilten Punkten beschreibt. Demnach werden gleich große Punkte möglichst homogen innerhalb einer Sampling Domäne verteilt. Ungewollte Rasterstrukturen werden durch eine stochastische Verteilung der Punkte verhindert.

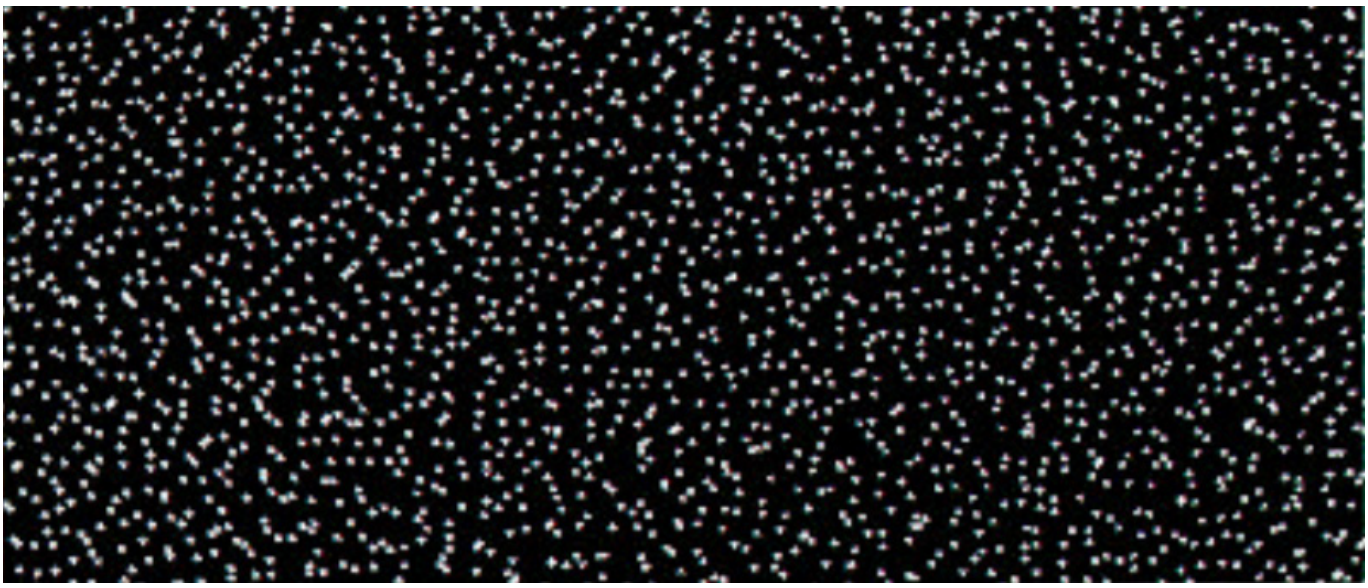


Abbildung 3.17: Verteilungsmuster von jittered Samples. (Quelle: [Co86], S.65)

Eine *Poisson Disk Verteilung* weist Blue Noise Eigenschaften auf [Br07, Yu15]. Die Samples halten einen Mindestabstand *r* zueinander (Abbildung 3.18), wodurch Clusterisierung von Samples vermieden wird. Gleichzeitig wird versucht, möglichst viele Samples innerhalb der Sampling-Domäne zu verteilen. Daher existieren bei einer idealen Poisson Disk Verteilung keine Positionen innerhalb der Sampling-Domäne, die mit einem weiteren validen Sample besetzt werden kann.

Cook [Co86] stellt eine sehr geradlinige, aber rechenintensive Implementierung des Poisson Disk Samplings vor, die *Dart Throwing* genannt wird [Br07, DH06]. Es wird eine Nachschlagetabelle erzeugt, die so lange mit zufälligen Samplepositionen gefüllt wird, bis die Sampling-Domäne voll ist. Positionen, die den Mindestabstand *r* zu anderen Samples nicht einhalten, werden verworfen. Cook hat das Jittered Grid dem Poisson Disk Sampling vorgezogen, da die von ihm vorgeschlagene Sampling-Technik vergleichsweise rechenintensiv ist und sich eine durch Jittered Grid generierte Verteilung Blue Noise Eigenschaften annähert. Nach der Veröffentlichung von Cooks Paper wurden allerdings Algorithmen mit geringerer Zeitkomplexität entwickelt, die ebenfalls eine Poisson Disk Verteilung erzielen.

Einer dieser Algorithmen wurde 2006 von Dunbar & Humphreys [DH06] vorgestellt und von Bridson [Br07] im folgenden Jahr auf eine beliebige Anzahl Dimensionen erweitert. Er ermöglicht die Berechnung von Poisson Disk verteilten Samples mit linearer Zeitkomplexität. Abbildung 3.19 zeigt eine Poisson Disk Verteilung von 17.593 Punkten, die mit dem Algorithmus von Dunbar & Humphreys generiert wurde.

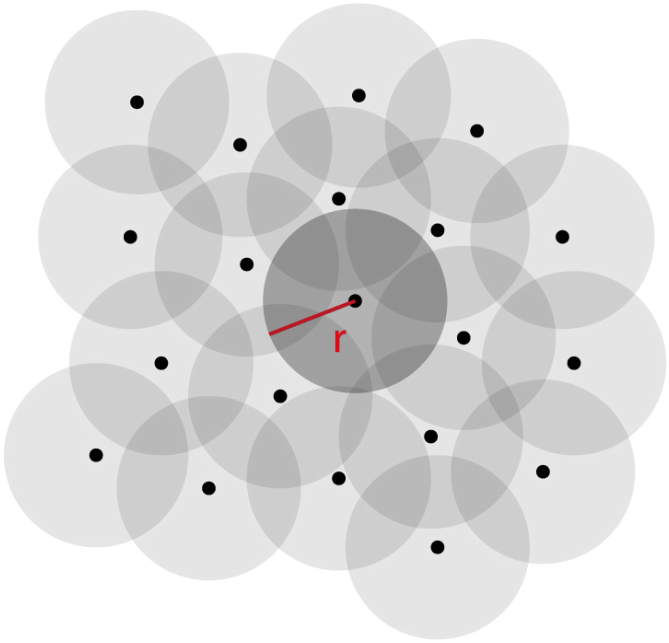


Abbildung 3.18: Zwanzig Punkte mit einer Poisson Disk Verteilung. Der graue Radius um die Punkte stellt den Mindestabstand dar, den die Punkte zueinander halten müssen.

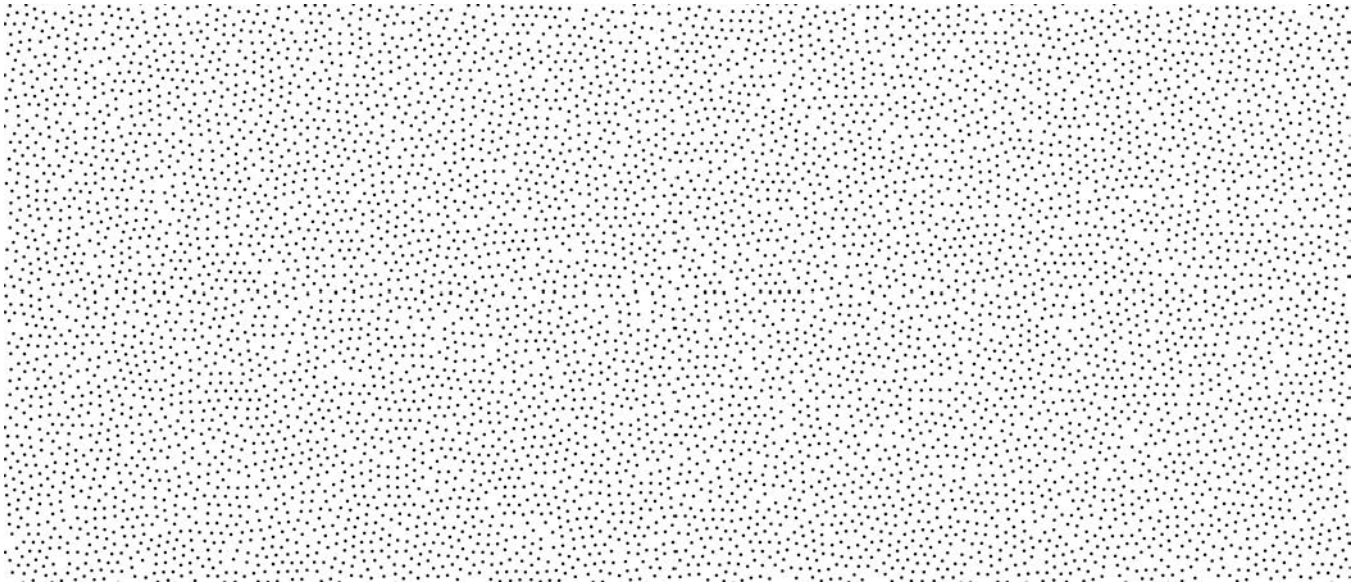


Abbildung 3.19: Eine in 80 ms generierte Poisson Disk Verteilung von 17.593 Punkten. (Quelle: [DU06], S. 503)

Der von Bridson [Br07] beschriebene Algorithmus nimmt die Ausdehnung der Sampling-Domäne R^n, eine minimale Distanz *r* sowie ein Zurückweisungslimit *k* als Eingabeparameter. *k* ist konstant und vom Autor standardmäßig auf einen Wert von 30 gesetzt. Abbildung 3.10 stellt den Ablauf des Algorithmus für die zweite Dimension dar. Vorbereitend wird ein *n*-dimensionales Hintergrundraster mit einer Zellengröße von $r/\sqrt{n}$ initialisiert (1). Dadurch enthält jede Zelle maximal einen Samplepunkt und kann durch ein *n*-dimensionales Array aus Integern repräsentiert werden. Ein positiver Integerwert gibt den Index eines Samples wieder, ein negativer Wert das Fehlen eines Samples. Im ersten Schritt wird ein beliebiges Start-Sample x_0 innerhalb der Sampling-Domäne ausgewählt (2). Sein Index wird dem Hintergrundraster sowie der sogenannten "aktiven Liste" hinzugefügt. Aus dieser Liste wird im zweiten Schritt so lange ein zufälliger Index *i* gewählt, bis sie keine Elemente mehr enthält. Im Bereich zwischen *r* und *2r* um den Samplepunkt x_i werden bis zu *k* Punkte p erzeugt und mit den existierenden Samples in Reichweite vergleichen (3). Ist die Distanz d zu einem Sample kleiner *r* (5), so wird der Punkt verworfen, andernfalls wird er dem Hintergrundraster und der aktiven Liste hinzugefügt (4). Wurde nach *k* Anläufen kein valider Punkt gefunden, so wird *i* aus der aktiven Liste entfernt. Bei einem hohen *k*-Wert kann mit dem Algorithmus von Bridson eine ideale Poisson Disk Verteilung erreicht werden (6). Bei einem niedrigen *k*-Wert sind Lücken innerhalb der Sampling-Domäne möglich.

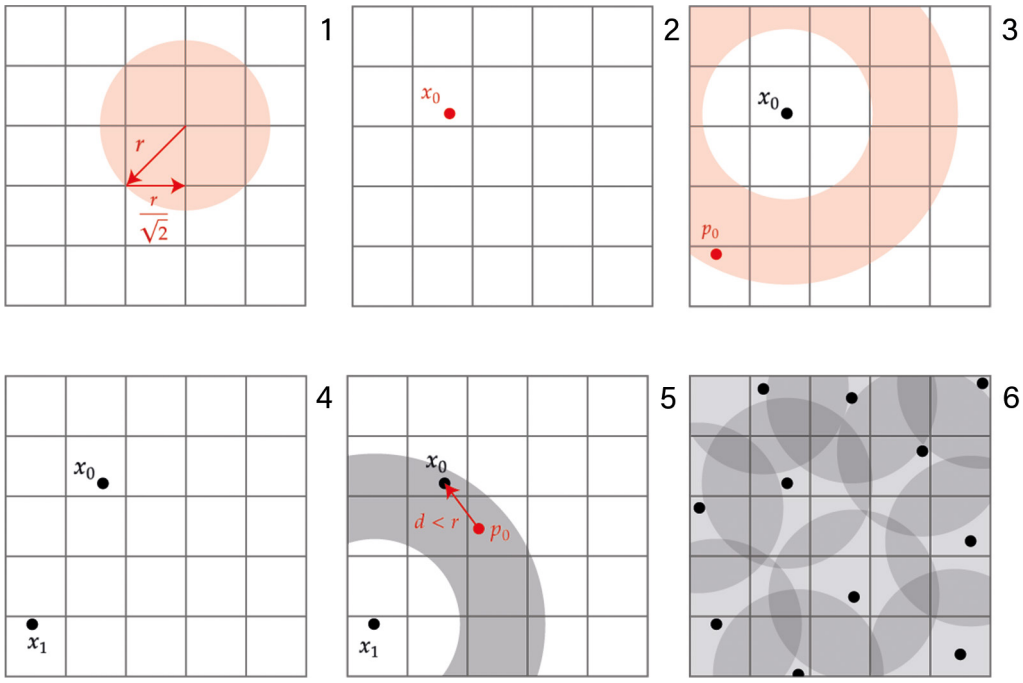


Abbildung 3.20: Darstellung des Poisson Disk Sampling Algorithmus von Bridson [Br07].

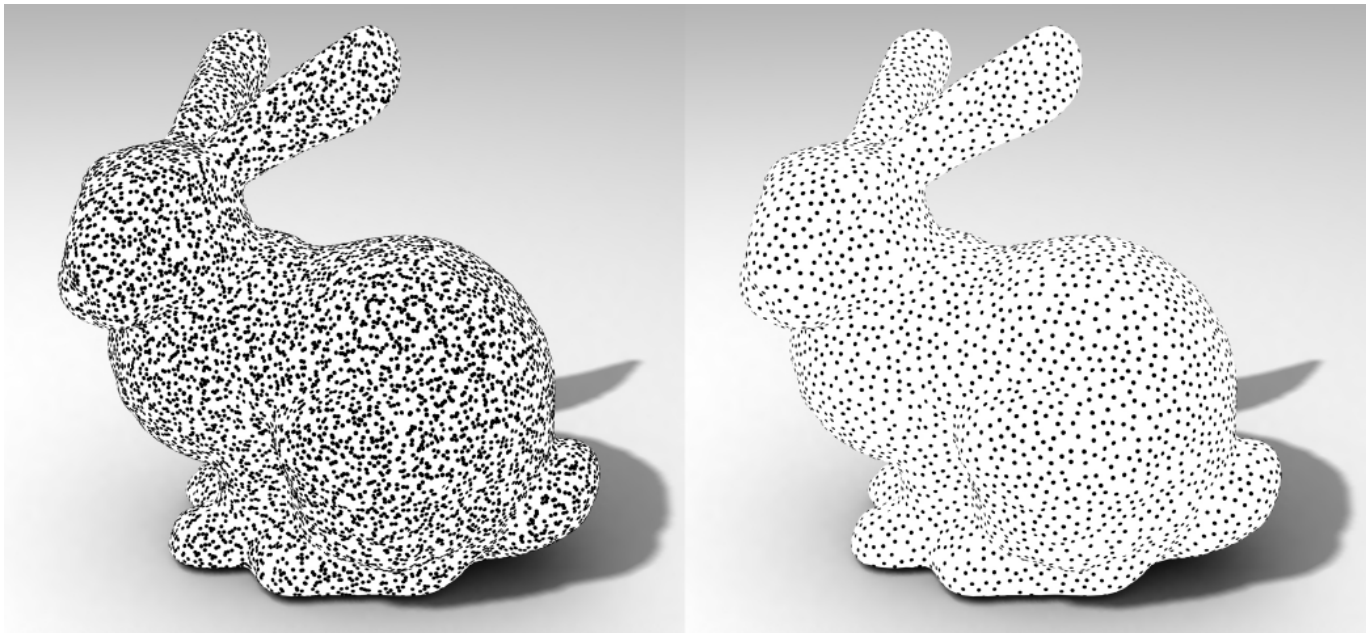


Abbildung 3.21: Bei einem gegebenen Eingabesatz von Samples wählt die Sample Elimination eine Untermenge mit der Poisson Disk Eigenschaften aus. Das linke Bild zeigt 15.000 pseudozufällig verteilte Punkte. Das rechte Bild zeigt 5.000 Poisson Disk Samples, die aus den 15.000 pseudozufälligen Punkten ausgewählt wurden. (Quelle: [Yu15], S.25)

3.3.5 Sample Elimination

Der von Cem Yuksel [Yu15] vorgestellten Algorithmus *Sample Elimination* ist ein weiterer Algorithmus zur Erzeugung einer Poisson Disk Verteilung. Mit ihm werden die Samples nicht mittels Dart Throwing oder auf Basis eines Rasters generiert. Vielmehr wird eine Basismenge *M* von zufällig verteilten Samples innerhalb der Sampling Domäne erzeugt. Gesucht wird anschließend eine Teilmenge *N* aus *M*, wobei *N* Poisson Disk Eigenschaften aufweisen soll. Bei der Sample Elimination werden fortlaufend Punkte aus *M* entfernt, bis die Zielmenge der Größe *N* erreicht ist. Je größer *M* im Vergleich zu *N* ist, desto zuverlässiger wird das Ergebnis, allerdings steigt dadurch die Berechnungszeit.

Anders als beim Poisson Disk Sampling wird eine Teilmenge des Sample Sets mit genau der gewünschten Größe ausgewählt, ohne explizit einen Radius der Poisson Disks anzugeben. Auch kann der Algorithmus auf beliebige Sampling-Domänen, z.B. einer Planetenoberfläche, angewendet werden.

Yuksel schlägt für die Sample Elimination einen Greedy-Algorithmus vor, der auf gängigen Datenstrukturen einer räumlichen Trennstruktur und einer Prioritätswarteschlange basiert. Durch die Greedy-Eigenschaft kann nicht garantiert werden, dass die Samples bestmöglich verteilt sind, also der Poisson Disk-Radius maximal ist. Dennoch werden, wie in Abbildung 3.21 zu erkennen ist, die gewünschten Poisson Disk Eigenschaften des Sample-Set erreicht.

Die Sample Elimination erfolgt gewichtet. Das bedeutet, abhängig vom Abstand zu seinen Nachbarn wird jedem Sample aus *M* ein Gewicht zugewiesen. Die Gewichtung ist abhängig von der Anzahl der Nachbarn in einem bestimmten Radius und von deren Abstand zum Sample. Je kleiner dem Abstand, desto größer die Gewichtung. Solange die Zielmenge *N* noch nicht erreicht ist, wird das Sample mit der größten Gewichtung aus *M* entfernt. Im Anschluss wird die Gewicht der Nachbarn des entfernten Samples neu berechnet. Als räumliche Trennstruktur hat Yuksel den *k-d-Baum* gewählt, als Prioritätenliste den *Heap*. Der gesamte Algorithmus ist in Abbildung 3.22 zu sehen.

Der Einsatz von Sample Elimination ist sinnvoll, wenn eine festgelegte Anzahl Samples mit Poisson Disk Eigenschaften innerhalb einer beliebigen Sample Domäne generiert werden soll.

```
function WeightedSampleElim( samples )
   Build a kd-tree for samples
   Assign weights w_i to each sample s_i
   Build a heap for s_i using weights w_i
   while number of samples > desired
      s_j ← pull the top sample from heap
      for each sample s_i around s_j
         Remove w_ij from w_i
         update the heap position of s_i
```

Abbildung 3.22: Pseudocode der gewichteten Sample Elimination. (Quelle: [Yu15], S. 3)

3.3.6 Halton Sequenz

Die nach ihrem Entwickler benannte Halton Sequenz ist ein Algorithmus zur Berechnung einer Sequenz von *n* quasi-zufälligen Punkten innerhalb eines *k*-dimensionalen Einheits-Hypercube [Ha64]. Abbildung 3.23 zeigt 100 zweidimensionale Halton Punkte im Vergleich zu 100 pseudozufällig verteilten Punkten.

Nach Pharr et al. [PJH18] bildet die *radikale Inverse* (engl. radical inverse) genannte Konstruktion den Kern der Halton Sequenz. Diese basiert auf dem Fakt, dass eine positive Integer-Zahl *a* mit Basis *b* als Sequenz als ihrer Ziffern $d_m(a) \ldots d_2(a)\, d_1(a)$ dargestellt werden kann, wobei $d_i(a)$ einen Wert zwischen *0* und *b-1* ergibt:

$$a = \sum_{i=1}^{m} d_i(a) b^{i-1}$$

Die radikale Inverse-Funktion Φ_b konvertiert diese Zahl in einen Bruchwert zwischen [0,1) indem die Ziffernwerte um den Radixpunkt gespiegelt werden:

$$\Phi_b(a) = 0.d_1(a)d_2(a) \cdots d_m(a)$$

Tabelle 3.2 zeigt die ersten sechs radikalen Inverse von Integer-Zahlen zur Basis *b*. Wie Pharr et al. anmerken, sind die erzeugten Werte weit von zuvor erzeugten Werten entfernt. Dieser Abstand schrumpft mit jedem generierten Sample, doch wird weiterhin ein Mindestabstand eingehalten.

Diese Eigenschaft nutzt die Halton Sequenz zur Generierung von annähernd Poisson Disk-verteilten Punkten in einem *n*-dimensionalen Raum. Dafür werden *n* radikale Inverse-Funktionen mit einer Basis benötigt, die zueinander Primzahlen sind, also keinen gemeinsamen Teiler außer 1 haben. Dies kann am einfachsten mit der Verwendung der ersten n Primzahlen $(p_1 \ldots p_n)$ erreicht werden. Damit kann ein Punkt der Halton Sequenz wie folgt bestimmt werden:

$$x_a = (\Phi_2(a), \Phi_3(a), \Phi_5(a), \cdots, \Phi_{p_n}(a))$$

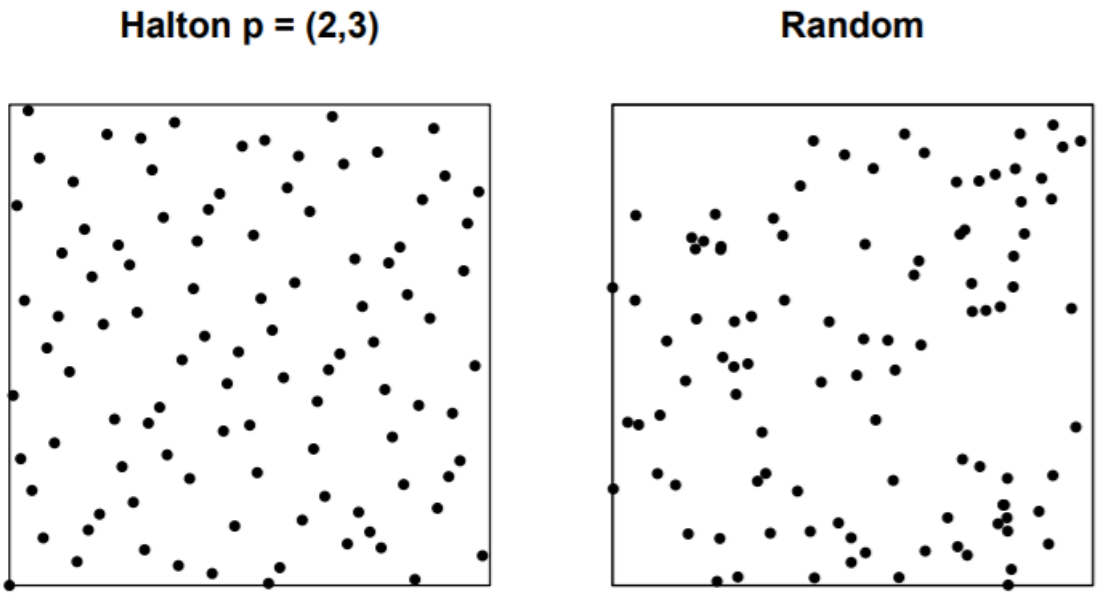


Abbildung 3.23: 100 Halton Punkte in $[0, 1]^2$ (links) im Vergleich zu 100 pseudozufälligen Punkten (rechts). (Quelle: [Ow17], S.8)

Willmott [Wi07], ein Entwickler von *Spore* [S20], beschreibt Vorteile, welche die Verwendung einer iterativen Variante der Halton Sequenz mit sich bringt. Zum Einen nutzt der Algorithmus keinen Hauptspeicher zum Speichern von Zwischenergebnissen bei der Erzeugung einer Liste von Samples. Zum Anderen ist der Prozess deterministisch, wodurch Samples nicht permanent gespeichert werden müssen, sondern bei Bedarf neu berechnet werden können. Wie die Sample Elimination kann auch die Halton Sequenz für eine beliebige Anzahl Samples eingesetzt werden.

3.3.7 Voronoi-Diagramm

Eine Sonderform der Verteilungstechniken ist die Unterteilung einer Fläche in Regionen. Diese kann beispielsweise durch ein *Voronoi-Diagramm* erfolgen. Nach Liebling und Pournin [LP10] unterteilt ein Voronoi-Diagramm eine Fläche auf Basis einer endlichen Menge *A* sogenannter *Sites* in möglicherweise unbegrenzte konvexe Polygone, die *Voronoi-Regionen* (engl. voronoi region) genannt werden. Eine Voronoi-Region besteht aus den Punkten, die einer bestimmten Site im Vergleich zu den anderen Sites am nächsten sind.

Die *Delaunay-Triangulierung* ist das untrennbare Gegenstück eines Voronoi-Diagramms. Sie wird mit denselben Sites *A* assoziiert wie das entsprechende Voronoi-Diagramm. Die Delaunay-Triangulierung kann aus einem Voronoi-Diagramm bestimmt werden, indem eine Dreieckskante zwischen jedem Paar Sites gezeichnet wird, deren dazugehörigen Voronoi-Regionen über diese Kante verbunden sind. Umgekehrt entstehen die Regionen eines Voronoi-Diagramms, wenn die Diagonalschnittpunkte der Dreiecke miteinander verbunden werden, die sich eine Site als Ecke teilen. Abbildung 3.24 zeigt die Delaunay-Triangulierung einer Fläche mit dem entsprechenden Voronoi-Diagramm.

Die Definitionen von Delaunay-Triangulierung und Voronoi-Diagrammen können geradeheraus auf höhere Dimensionen angewendet werden [LP10]. Dadurch kann die Oberfläche von dreidimensionalen Objekten wie Kugeln ebenfalls mittels Voronoi-Diagramm unterteilt werden.

a (Basis 10)	a (Basis 2)	$\Phi_2(a)$ (Basis 2)	$\Phi_2(a)$ (Basis 10)
0	0	0	0
1	1	0.1	1/2
2	10	0.01	1/4
3	11	0.11	3/4
4	100	0.001	1/8
5	101	0.101	5/8

Tabelle 3.2: Die radikale Inverse der ersten paar nichtnegativen ganzen Zahlen, berechnet zur Basis 2. (basierend auf [PJH18], Tabelle 7.3)

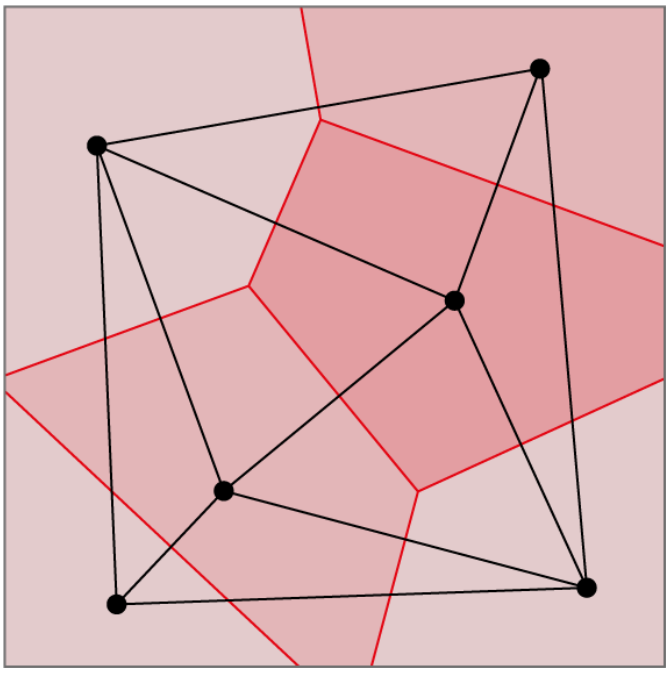


Abbildung 3.24: Eine Delaunay-Triangulierung (schwarz) von sechs Sites und dem entsprechenden Voronoi-Diagramm (rot).

3.4 Simulation

Nach Brockhaus [Brockhaus1] ist eine *(Computer-)Simulation* die (computergestützte) Nachbildung des Verhaltens eines Systems. Dafür werden die, in Realität zum Teil sehr komplexen, Systeme modelliert und die Entwicklung der Modelle simuliert. Darauf basiert im Bereich der Videospiele das Genre der *Simulatoren*, bei denen SpielerInnen komplexe Zusammenhänge nachvollziehen und durch Interaktion mit dem System selbst erleben können [USK3]. Das nachgebildete System kann dabei wie beim *Landwirtschafts Simulator* [S06] eine Berufsgruppe oder wie bei den *Sims* [S04] das Leben an sich darstellen. Doch auch im kleinen werden bei der Entwicklung von Spielen Simulationen wie Partikel-Systeme oder physikbasierte Simulation für beispielsweise die Erzeugung von Effekten oder die Animationen von Objekten verwendet.

3.4.1 Partikelsimulation

Partikel-Systeme wurden 1983 von William T. Reeves [Re83] entwickelt um verschwommene (engl. fuzzy) Objekte ohne klare Oberfläche wie Rauch, Feuer und Wasser darzustellen sowie deren Dynamik einzufangen. Ein Partikel-System stellt ein Objekt als Wolke primitiver *Partikel* dar, die ihre Form ändern und sich mit der Zeit bewegen. Auch ist ein Partikel-System nicht deterministisch. Stattdessen werden stochastische Prozesse genutzt, um die Form und das Auftreten eines Objektes zu bestimmen.

Heutzutage finden simulierte Partikel in der Computergrafik und bei der Spieleentwicklung häufig Anwendung, um beispielsweise Rauch, Gischt, Explosionen oder Flüssigkeiten darzustellen. So nutzen beispielsweise Iwasaki et al. [IUD*10] ein Partikel-System, um das Schmelzen von Eis zu simulieren (Abbildung 3.25). Partikelsimulation kann auch bei der prozeduralen Generierung von Objekten zum Einsatz kommen. Wie Compton [Co17] beschreibt und mit ihren Kollegen ausführt [CGG*07], wurden beispielsweise die Pfade von simulierten Partikeln verwendet, um bei der Planetengenerierung in *Spore* [S20] das Terrain der Planeten zu formen.

3.4.2 Agentenbasierte Modellierung und Simulation

Nach Weyer und Ross [WR17] ermöglicht *agentenbasierte*

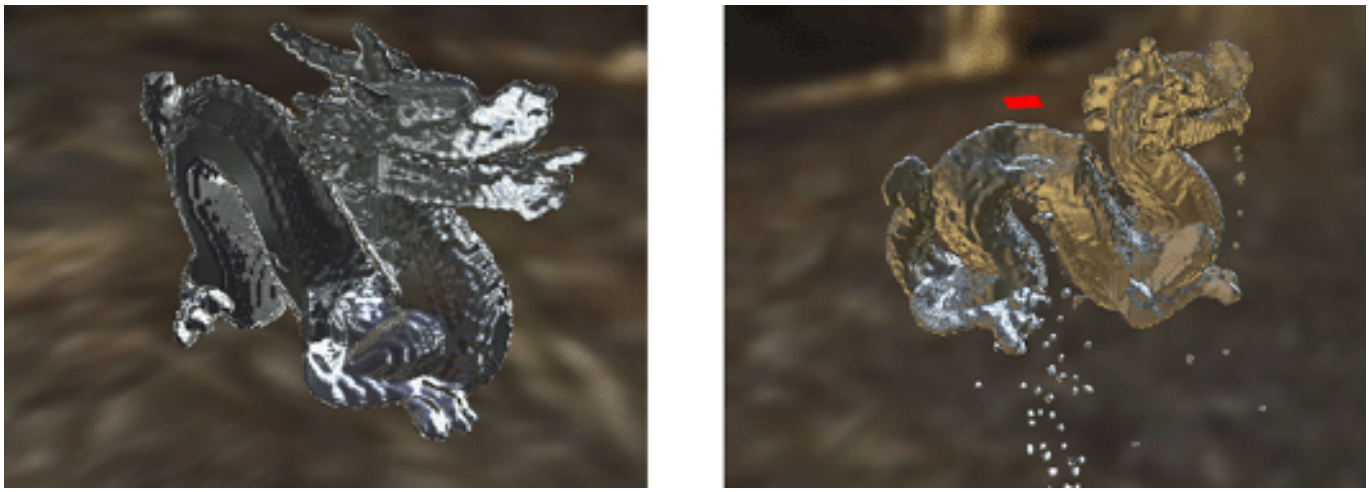


Abbildung 3.25: Interaktives Rendering und Simulation eines schmelzenden Eisdrachens aufgrund einer externen Wärmequelle, dargestellt durch ein rotes Quadrat. Die Simulation basiert auf einem Partikel-System (Quelle: [IUD*10], S.2222)

Modellierung und *Simulation*, kurz ABMS, komplexe Systeme computergesteuert zu modellieren und ihre Entwicklung zu simulieren. Dabei bestehen agentenbasierte Modelle aus den drei Komponenten *Agent*, der *Umwelt* in der dieser sich bewegt und *Regeln*, welche die Interaktionen des Agenten mit der Umwelt und anderen Agenten festlegen. Die Agenten unterscheiden sich durch ihre *Zustände* wie Alter und Geschlecht und ihre *Strategien* im Umgang mit ihrer Umwelt und anderen Agenten. Ihre individuellen Entscheidungsprozesse sowie die Art und Weise, mit der ein Agent Informationen beschafft, verarbeitet und sich mit anderen Agenten austauscht, werden durch den Regelsatz der Simulation bestimmt. Bei ihren Entscheidungen orientieren sich die Agenten an individuellen Präferenzen und versuchen, die aus ihrer Sicht optimale Lösung zu finden. Dadurch entstehen nach Weyer und Ross schwer vorhersehbare und überraschende Systeme, die sich nicht aus den Eigenschaften der Agenten und der Umwelt ableiten lassen, sondern das Ergebnis der Handlungen einer Vielzahl autonom agierender Agenten ist. Zwar wird agentenbasierte Simulation insbesondere von den Wirtschafts- und Sozialwissenschaften genutzt [WR17], doch kann sie auch bei der Entwicklung von Spielen z.B. bei der Simulation von Tierpopulation oder bei der Simulation von Verkehr auf virtuellen Straßen zum Einsatz kommen (Abbildung 3.26).

3.4.2.1 Zelluläre Automaten

Eine Unterart der agentenbasierten Modelle und Simulationen sind nach Scholz [Sc14] die zellulären Automaten. Dieser ermöglichen mit einem Spielfeld und einer großen Anzahl Spielern eine diskrete Modellierung komplexer Sachverhalte wie die Entstehung des Lebens oder Modelle im Verkehrswesen. Die Spieler agieren, anders als bei anderen agentenbasierten Modellen nach identischen Regeln in Abhängigkeit ihrer Nachbarschaft. Das Spielfeld kann dabei von beliebiger Dimension sein. Ein eindimensionales Spielfeld besteht aus einer Reihe Pixel. Jeder Pixel nimmt exakt einen Zustand aus einer vordefinierten Zustandsmenge ein. Wie Abbildung 3.27 verdeutlicht wird dieser Zustand mit jeder Generation durch ein vordefiniertes Regelwerk und die direkte Nachbarschaft einer Zelle bestimmt. In diesem Beispiel können die Zellen die Zustände schwarz und weiß annehmen.

Abbildung 3.26: Die Autos von *City Skylines* [S02] agieren als Agenten auf den virtuellen Straßen und simulieren Verkehr.

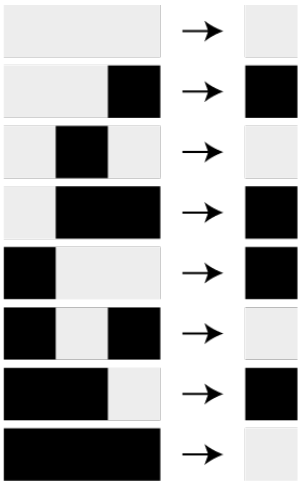


Abbildung 3.27: Die Grammatik "Regel 90" eines eindimensionalen Zellulären Automaten mit ihren acht Produktionsregeln. Grau bedeutet, es ist kein Pixel gesetzt; schwarz stellt ein Pixel dar. (basierend auf [Sc14] S.19)

Werden die einzelnen Iterationen des Automaten untereinander dargestellt, so entstehen grafische Strukturen. Abbildung 3.28 zeigt mehrerer Iterationen eines eindimensionalen Automaten mit Regel 90. Die Bezeichnung Regel 90 leitet sich aus der binären Schreibweise der Zahl neunzig (01011010) ab und ist eine Kurzschreibweise der Produktionsregeln aus Abbildung 3.27. Folglich existieren bei diese elementaren zellulären Automaten insgesamt 256 unterschiedliche Regeln. Eine „0" wird als „weißer Pixel" interpretiert; eine „1" bedeutet „schwarzer Pixel".

Werden Spielfelder der Iterationen eines zweidimensionalen zellulären Automaten hintereinander abgespielt, so entsteht eine Animation des Simulationsprozesses. Wie Scholz [Sc14] aufzeigt können so beispielsweise Waldbrände, das Räuber-Beute-Schema oder Verkehr simuliert werden.

3.4.3 Physikbasierte Simulation

Wie Baraff & Witkin [BW97] bereits 1997 anmerkten, sind physikbasierte Modellierung und Simulation zu einem bedeutenden Ansatz der Computergrafik und Computeranimation geworden. Das breite Einsatzfeld wird in dem State of The Art Report von Nealen et al. [NMK*05] deutlich. Sie nennen unter anderem die Simulation von Stoff, Haarsimulation, Kollisionserkennung, haptisches Force-Feedback sowie Flüssigkeitssimulation als Bereiche, in denen physikbasierte Simulation eingesetzt wird.

Zur physikbasierten Modellierung werden bestimmte physikalische Eigenschaften des nachzubildenden realen Systems verwendet und durch mathematische Berechnungen simuliert. Diese Eigenschaften sind beispielsweise Geschwindigkeit, Masse, Dichte oder Verformbarkeit einer Entität. Abbildung 3.29 zeigt die physikbasierte Simulation von Stoff, der sich über einige 3D-Objekte legt.

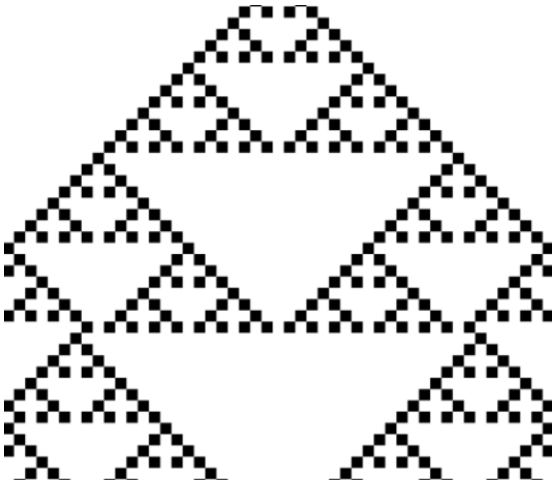


Abbildung 3.28: Eine simulierte Struktur, die mithilfe eines elementaren zellulären Automaten und Regel 90 erzeugt wurde. Hierfür wurde der Webservice Zellmemore [S26] genutzt.

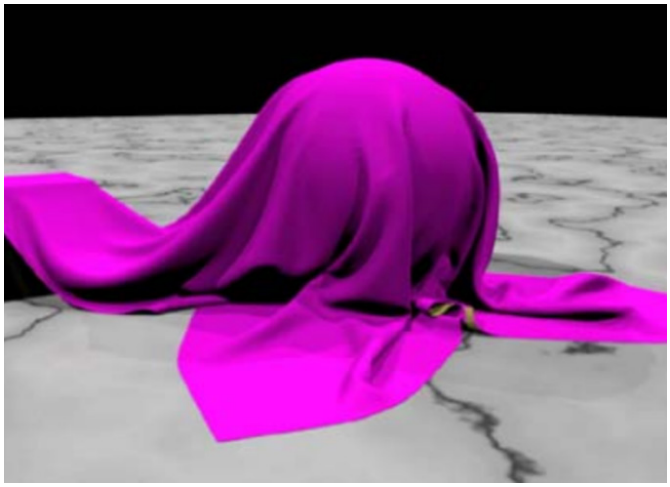


Abbildung 3.29: Ein Stück virtueller Stoff, dessen Verformung über einige 3D-Objekte mittels physikbasierter Simulation bestimmt wird. (Quelle [NMK*05], S.816)

3.5 Noise

Das Erzeugen von *Noise*, zu deutsch Rauschen, ist eine weitere Technik, die zur prozeduralen Generierung eingesetzt wird. Es kann dazu genutzt werden scheinbar zufällige Strukturen zu generieren. Lagae et al. [LLC*10] bezeichnen Noise als ein zufälliges und unstrukturiertes Muster, das überall dort nützlich ist, wo eine umfangreiche Detailquelle benötigt wird, der es jedoch an offensichtlicher Struktur mangelt. Als mögliche Einsatzgebiete nennen die Autoren unter anderem prozedurale Texturen, Wolken, Hitzewellen und zufällige Bewegung von animierten Charakteren. Noise werde sowohl bei Filmproduktionen als auch bei Videospielen weitläufig genutzt.

Lagae et al. [LLC*10] nennen folgende Vorteile von (prozeduralem) Noise:

- Noise ist meist schnell zu evaluieren
- Noise ermöglicht die Evaluation komplexer und verworrener Muster zur Laufzeit
- Noise hat einen geringen Speicherbedarf
- mit einem geeigneten Parametersatz lässt sich aus prozeduralem Noise auf einfache Weise eine Vielzahl unterschiedlicher Muster erzeugen
- Noise ist oft beliebig zugänglich und kann dadurch in konstanter Zeit unabhängig und parallel ausgewertet werden
- Eine prozedurale Noisefunktion ist von Natur aus kontinuierlich, mehrfach auflösend und basiert nicht auf diskret abgetasteten Daten

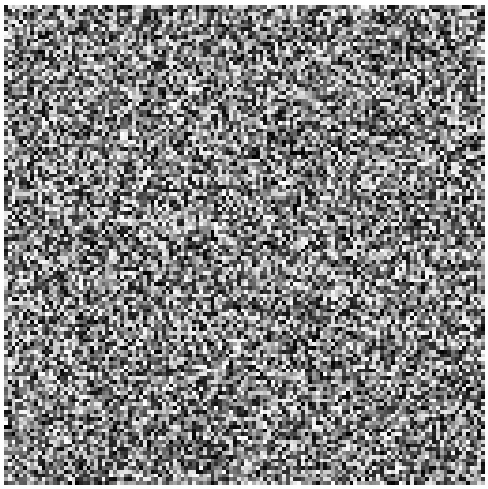


Abbildung 3.30: White Noise ist die Grundlage für viele Noise Techniken.

3.5.1 White Noise

Die Grundlage für viele Noise Techniken ist *White Noise*. Nach Lagae et al. [LLC*10] enthält White Noise alle Frequenzen in gleicher Mischung und mit zufälliger Phase, so dass es das Rohmaterial liefert, um unstrukturierte Signale mit jeder Kombination von Frequenzen zu erzeugen. Abbildung 3.30 zeigt beispielhaft einen zweidimensionalen White Noise dargestellt durch unterschiedliche Grauwerte. Für jeden Pixel wird ein Wert zwischen 0 und 1 zufällig gewählt und auf dieser Basis der RGB Grauwert berechnet.

3.5.2 Lattice Gradient Noise

Lagae et al. [LLC*10] bezeichnen *Lattice Gradient Noise* als eine Kategorie von Noisefunktionen, bei denen Noise als eine Interpolation zwischen zufälligen Werten eines *Integer-Gitters* evaluiert wird. In diese Kategorie fallen unter anderem Perlin Noise und Simplex Noise.

3.5.2.1 Perlin Noise

Perlin Noise wurde ursprünglich 1985 von Ken Perlin [Pe85] als Mittel der Textursynthese vorgestellt. Der Algorithmus kann neben der originalen zweidimensionalen Implementierung auch bei einer beliebigen Anzahl Dimensionen eingesetzt werden. Gustavson [Gu05] beschreibt den Evaluationsprozess auf anschauliche Weise. Nach dem Autor werden bei eindimensionalem Perlin Noise pseudozufällige Gradienten mit den Integer Koordinaten assoziiert und der Funktionswert an diesen Punkten auf 0 gesetzt. Für einen beliebigen Punkt x zwischen zwei Integer Koordinaten wird der Wert auf Basis der Gradienten interpoliert (Abbildung 3.31).

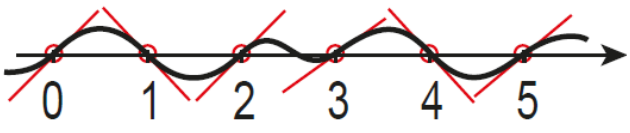


Abbildung 3.31: Eindimensionaler Perlin Noise (Quelle: [Gu*05] S.1)

Als Blendfunktion wird ein Polynom 5. Grades verwendet:

$$f(t) = 6t^5 - 15^4 + 10t^3$$

Diese führt zu ähnlichen Werten wie Hermetische Blendfunktion $f(t) = 3t^3 - 2t^2$, hat aber zusätzlich eine kontinuierliche zweite Ableitung an ihren Endpunkten. Nach Gustavson ist die Noise Funktion dadurch besser geeignet für typische Aufgaben der Computergrafik wie Oberflächenverschiebung und Bump Mapping.

Wie Abbildung 3.32 zeigt, formen die Integer-Punkte in der zweiten Dimension ein gleichmäßiges Gitter aus Quadraten. An jedem dieser Punkte wird ein pseudozufälliger 2D-Gradient gewählt. Für einen beliebigen Punkt *P* auf der Oberfläche wird der Noisewert aus den vier nächstgelegenen Gitterpunkten bestimmt und mittels Blendfunktion errechnet.

In der dritten Dimension wird der Raum durch das Integer Gitter in Würfel aufgeteilt. Die Gradienten sind ebenfalls dreidimensional. Die Interpolation erfolgt entlang der drei Achsen nacheinander. Für einen beliebigen Punkt *P* im Raum wird der Noisewert aus den acht nächstgelegenen Gitterpunkten bestimmt und mittels Blendfunktion verrechnet.

Bei der Wahl der Gradienten führt Gustavson aus, dass die Evaluierung einer Noise Funktion wiederholbar, also deterministisch, sein sollte. Aus diesem Grund sind die resultierenden Werte pseudozufällig und nicht rein zufällig. Die Pseudozufälligkeit wird bei Perlin Noise mit einem Permuationsarray aus zufällig verteilten Integerzahlen von 0 bis 255 sowie mit einer Sammlung aus zwölf festgelegten Gradienten gleicher Länge erreicht. Ein Gradient wird pro Gitterpunkt gewählt, indem mithilfe der drei Koordinaten Werte aus dem Permuationsarray ausgewählt und miteinander verrechnet werden. Mit Modulo 12 wird anschließend der Index für das Gradientenarray bestimmt. Abbildung 3.33 zeigt die von Ken Perlin ausgewählten Gradienten für das dreidimensionale Perlin Noise.

3.5.2.2 Simplex Noise

Simplex Noise ist als eine Weiterentwicklung von Perlin Noise zu verstehen. Der Algorithmus wurde 2001 ebenfalls von Ken Perlin vorgestellt [Pe01] und von Gustavson [Gu05] ausführlich beschrieben. Dieser nennt folgende Vorteile gegenüber Perlin Noise:

- geringere Rechenkomplexität mit weniger Multiplikationen
- Skalierung in höhere Dimensionen (größer als die dritte Dimension) mit weniger Rechenaufwand
- keine erkennbaren Richtungsartefakte
- ein wohldefinierter und kontinuierlicher Gradient, der recht günstig berechnet werden kann
- kann leicht auf Hardware implementiert werden

Nach Gustavson basiert Simplex Noise anders als Perlin Noise auf einem *Simplex-Gitter*. Ein *Simplex* ist dabei die einfachste und kompakteste Form, die wiederholt werden kann, um den gesamten Raum auszufüllen. In der zweiten Dimension ist der Simplex das gleichseitige Dreieck (Ab-

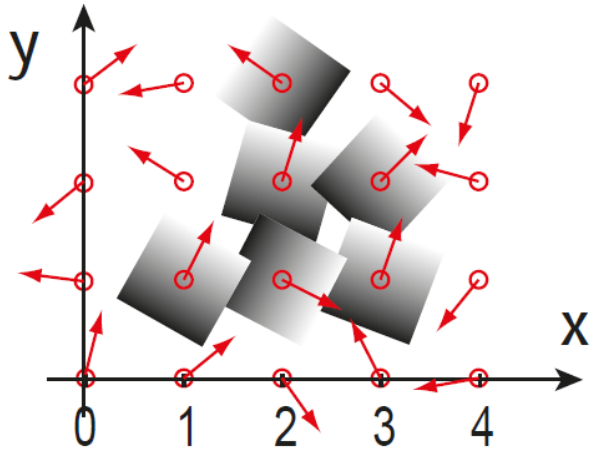


Abbildung 3.32: Pseudozufällig verteilte Gradienten auf dem Integer-Gitter eines zweidimensionalen Perlin Noise. (Quelle: [Gu*05], S.2)

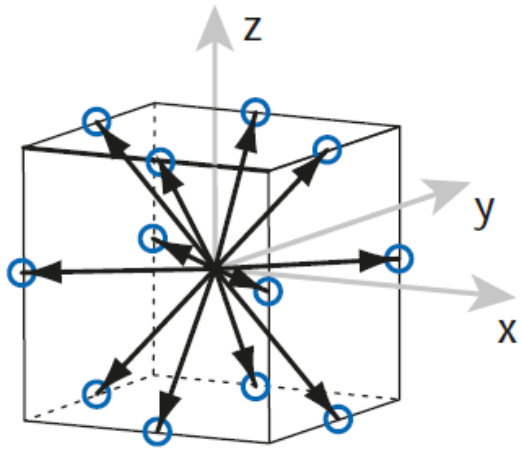


Abbildung 3.33: Die zwölf von Ken Perlin gewählten Gradienten für 3D Perlin Noise. (Quelle: [Gu*05], S.3)

bildung 3.34), in der dritten Dimension ein leicht verzerrter Tetraeder (Abbildung 3.35). Allgemein gesprochen hat eine Simplex-Form der *N*-ten Dimension *N + 1* Ecken im Gegensatz zu *2N* Ecken des entsprechenden Hypercubes. Dadurch müssen für die Berechnung des Noise-Wertes weniger Gradienten evaluiert und die Ergebnisse verrechnet werden, wodurch Rechenleistung gespart wird.

Simplex Noise berechnet den Noise Wert an einem Punkt *P* als Summe der Beiträge von jeder Ecke des Simplex, in dem sich *P* befinden. Ein Beitrag ergibt sich aus der Evaluierung des Gradienten multipliziert mit einer *radialsymmetrischen Dämpfungsfunktion*. Die radiale Dämpfung wird gewählt, so dass der Einfluss von jeder Ecke Null erreicht, bevor die Grenze zum nächsten Simplex überschritten wird. Dies bedeutet, dass Punkte innerhalb eines Simplex nur von den Beiträgen aus den Ecken dieses bestimmten Simplex beeinflusst werden (Abbildung 3.36).

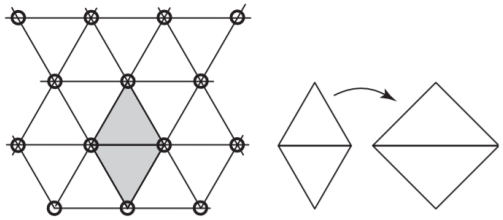


Abbildung 3.34: Das Simplex des zweidimensionalen Raums ist das gleichseitige Dreieck. Zwei dieser Dreiecke ergeben gestaucht ein Quadrat. (Quelle: [Gu05], S.4)

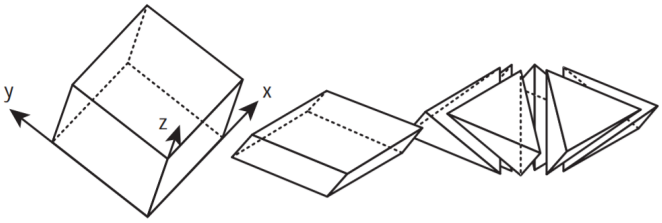


Abbildung 3.35: Das Simplex des dreidimensionalen Raums ist ein leicht gestauchter Tetraeder. Sechs dieser Tetraeder ergeben einen Würfel. (Quelle: [Gu05], S.4)

3.5.3 Alternativen

Lagae et al. [LLC*10] stellen einige Alternativen zu Lattice Gradient Noise vor. Zu diesen zählen *explizite Noises-Techniken* wie *Wavelet Noise* und *Anisotropes Noise*. Explizites Noises erzeugen Noise in einer Vorverarbeitung und speichern diese. Da explizite Noises im engeren Sinne nicht prozedural sind und einen endlichen Wertebereich haben, sind sie für den Planetengenerator ungeeignet.

Als weitere Alternative nennen Lagae et al. *Sparse Convolution Noise* (dt. spärliche Faltung) mit den Beispielen *Spot Noise* und *Gabor Noise*. Hier wird Noise als Summe zufällig positionierter und gewichteter *Kernels* generiert. Abbildung 3.38 zeigt eine "Haar"-Textur, die auf diese Art und Weise generiert wurde. Passende Kernels für eine Problemstellung zu identifizieren erfordert viel Übung, sowie ein externes Zeichenprogramm und rudimentäre gestalterische Fähigkeiten. Daher ist die Verwendung von Sparse Convolution-Techniken für den Planetengenerator ungeeignet.

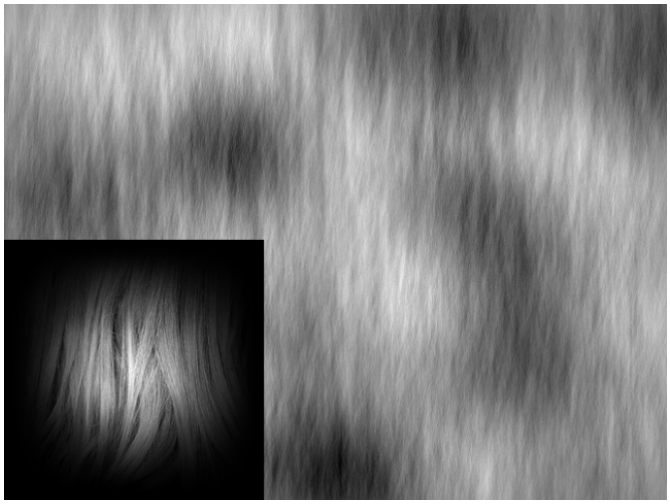


Abbildung 3.38: Sparse Convolution Noise. Mithilfe einer Fensterprobe aus einem Bild von Haaren (unten links) als Kernel wurde eine ungefähre "Haar"-Textur unter Verwendung von 2D Sparse Convolution erzeugt (rechts). (Quelle: [LLC*10], S.2585)

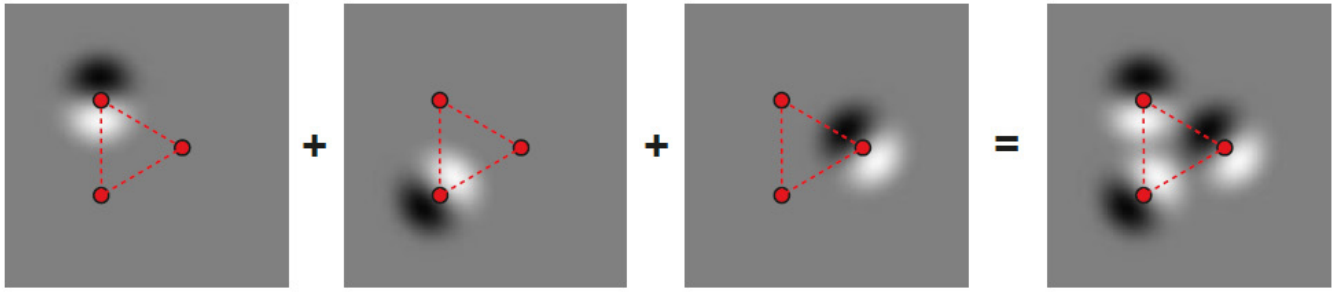


Abbildung 3.37: Die radialen Gradienten an den Ecken des Simplex tragen zum Noise Wert innerhalb der Simplex Form bei – hier beispielhaft in der zweiten Dimension. (Quelle: [Gu05], S.5)

3.5.4 Mathematische Operatoren

Auf Noisefunktionen lassen sich mathematische Operationen anwenden und zumeist frei miteinander kombinieren. Dadurch können unterschiedlichste Ergebnisse erzielt werden. Die beiden wichtigsten Parameter sind hierbei die *Frequenz* der Eingabewerte und die *Gewichtung* der Evaluierungsresultate.

3.5.4.1 Billow Noise

Billow Noise basiert auf den Werten einer Basisfunktion wie Simplex oder Perlin Noise welche zwischen -1 und 1 liegen. Wird von diesen Basiswerten die absolute Zahl errechnet, so bilden sich, wie in Abbildung 3.39 zu sehen ist, aufgeblähte (engl. billow) Strukturen. Daraus leitet sich der Name Billow Noise ab.

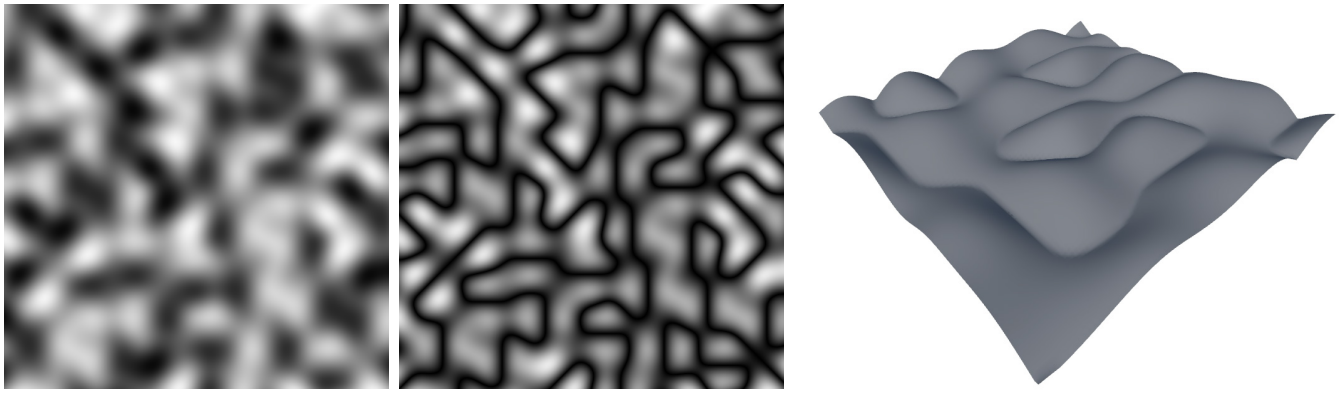


Abbildung 3.39: Von Simplex Noise (links) werden die absoluten Werte errechnet und dadurch Billow Noise erzeugt (mittig). Die rechte Abbildung stellt die Höhenkarte eines Billow Noise dar.

3.5.4.2 Ridge Noise

Galin et al. [GGP*19] beschreiben *Ridge Noise* als die Inverse der absoluten Werte einer Basis-Noisefunktion. Die absoluten Werte liefert das zuvor beschriebene Billow Noise. Der Name „Ridge Noise“ leitet sich von der Ähnlichkeit des aus dem Noise erzeugten Terrain mit Gebirgsgraten (engl. Ridge) ab (Abbildung 3.40).

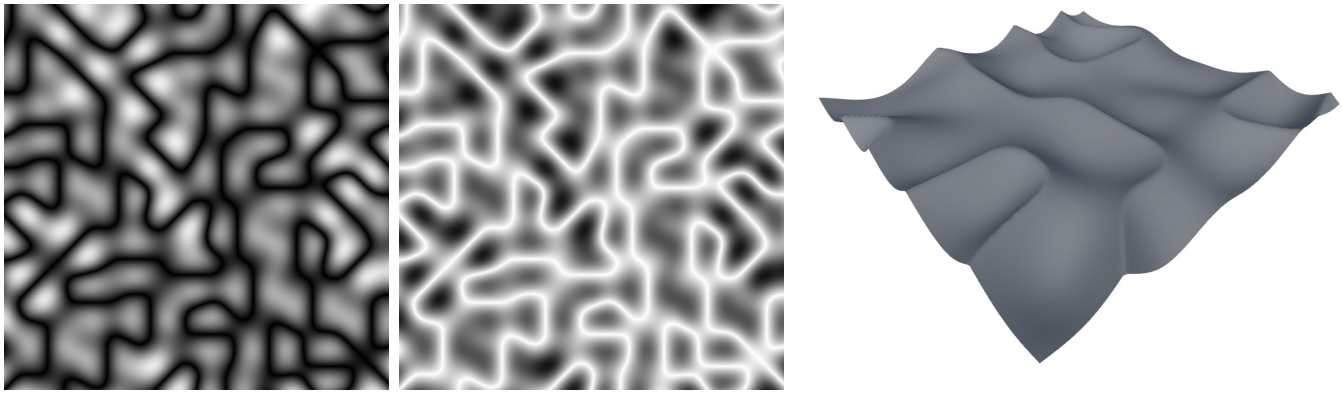


Abbildung 3.40: Die Werte des Billow Noise (links) werden invertiert und dadurch Ridge Noise erzeugt (mittig). Die rechte Abbildung stellt die Höhenkarte eines Ridge Noise dar.

3.5.4.3 Werte eingrenzen

Die Werte eines Basisnoise können *limitiert* werden, um beispielsweise Plateaus oder Ebenen im dreidimensionalen Raum zu generieren (Abbildung 3.41). Diese Art von Begrenzung ist vergleichbar mit der Clamp-Funktion gängiger Mathe-Bibliotheken.

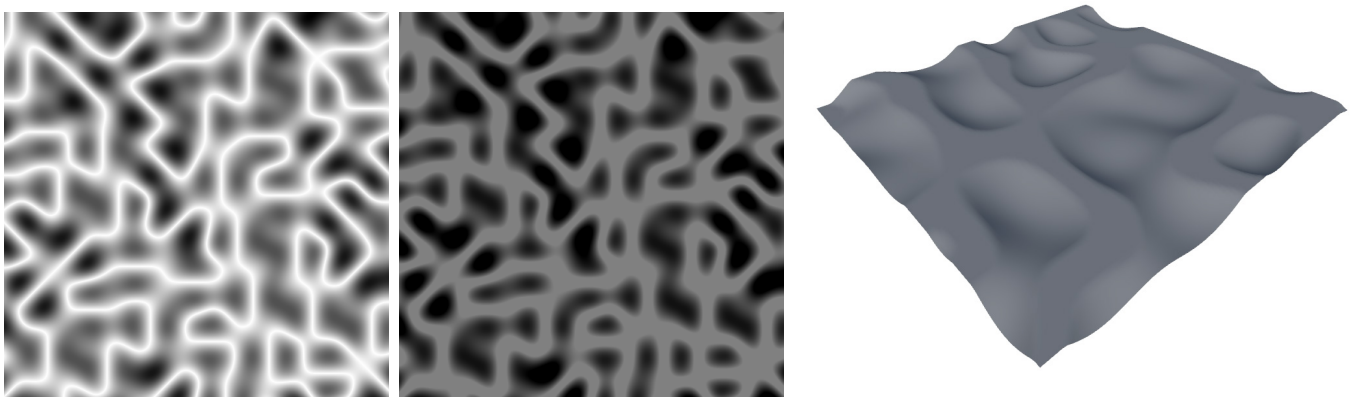


Abbildung 3.41: Die Werte des Ridge Noise (links) werden im oberen Bereich auf `0.5` und im unteren Bereich auf `-0.5` limitiert (mittig). Die rechte Abbildung stellt die Höhenkarte eines limitierten Ridge Noise dar.

3.5.4.4 Octave Noise

Galin et al. [GGP*19] beschrieben die Möglichkeit Werte einer Basis-Noisefunktion mit unterschiedlichen Frequenzen und Gewichtungen aufzusummieren, um feinere Noise Strukturen zu erzeugen. Die einzelnen Summanden werden hierbei *Oktaven* genannt. Die Anzahl der Oktaven kann frei gewählt werden. Mit steigender Oktave steigt die Frequenz der Eingabewerte und sinkt die Gewichtung. Abbildung 3.42 zeigt drei Oktaven eines Ridge Noise.

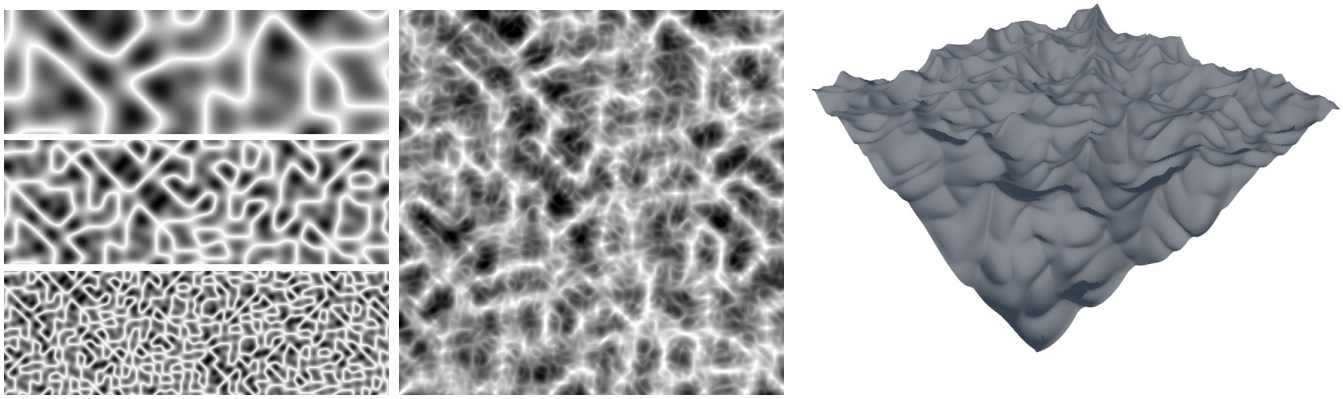


Abbildung 3.42: Drei Oktaven des Ridge Noise (links) werden summiert und dadurch Octave Noise generiert (mittig). Die rechte Abbildung stellt die Höhenkarte eines Ridge Noise mit drei Oktaven dar.

3.5.4.5 Noise mischen

Evaluationen von unterschiedlichen Noise Funktionen können mit einer *Blend Funktion* kombiniert werden. Dadurch wird ein neues Noise generiert. Abbildung 3.43 zeigt die Kombination von drei Oktaven Ridge Noise mit einfachem Simplex Noise.

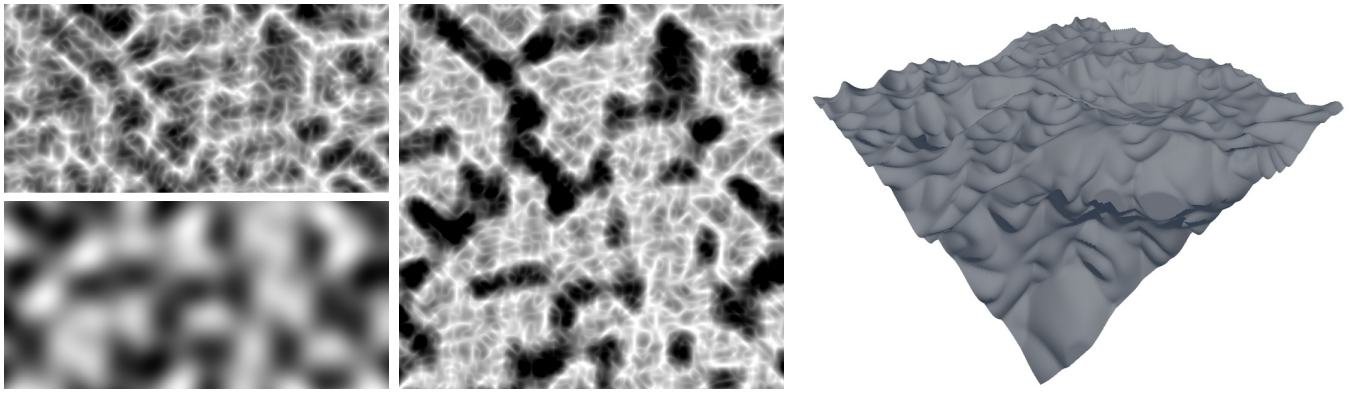


Abbildung 3.43: Zwei Eingabenoises (links) werden mit Hilfe einer Blendfunktion verrechnet und dadurch ein neues Noise erzeugt (mittig). Die rechte Abbildung stellt die Höhenkarte eines gemischten Noise dar.

3.5.4.6 Noise multiplizieren

Durch die *Multiplikation* von zwei Eingabenoises werden Werte nahe Null angeglichen, während Werte nahe 1 weitestgehend unverändert bleiben. Wie Abbildung 3.44 zeigt können dadurch beispielsweise die Täler des Ridge Noise geglättet werden, während gleichzeitig die Spitzen zerklüftet bleiben. Dieser Effekt kann durch mehrfache Multiplikation (quadratisch oder kubisch) verstärkt werden.

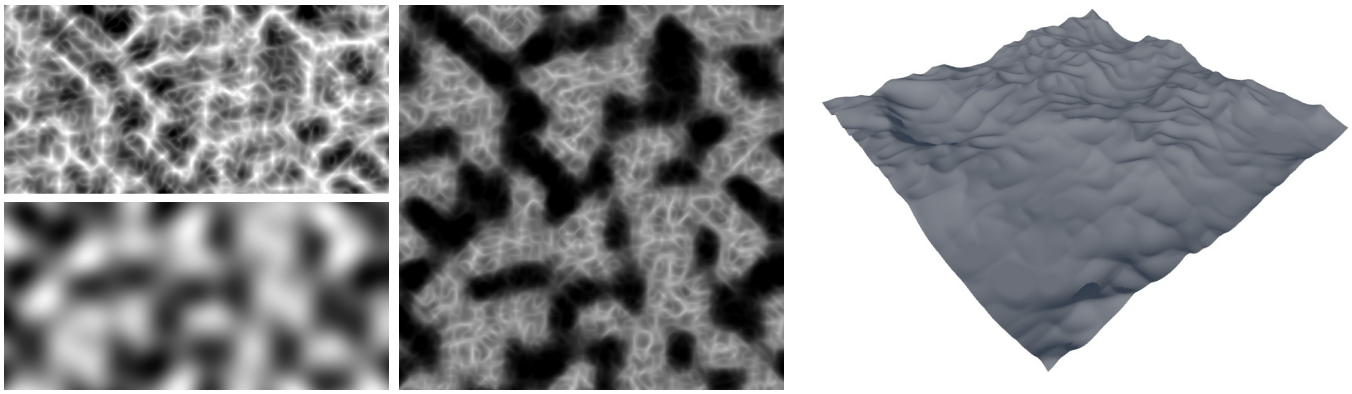


Abbildung 3.44: Zwei Eingabenoise (links) werden miteinander multipliziert. Dadurch entstehen natürlicher aussehendes Gebirge mit Tälern (mittig und rechts). Die rechte Abbildung stellt die Höhenkarte eines multiplizierten Noise dar.

3.5.5 Domain Warping

Bereits 1985 setzte Ken Perlin [Pe85] eine von ihm "turbulence" genannte Funktion ein, um eine Marmor-Textur zu erzeugen. Dafür addierte er die Texturkoordinaten mit einem *Turbulenz-Wert*. Dieser wird mit der *Turbulenz-Funktion* berechnet, welche die Texturkoordinaten als Eingabeparameter erhält und darauf eine Perlin Noise-Funktion aus mehreren Oktaven anwendet. Dadurch wird der ursprüngliche Evaluierungspunkt verzogen. Aus diesem Grund wird die von Perlin präsentierte Technik auch *Domain Warping* bzw. *Domain Distortion* genannt. Abbildung 3.45 zeigt eine Textur, die aus einfachem Simplex Noise erzeugt und auf dessen Eingabewerte Domain Warping angewendet wurde.

3.6 Fazit

Für die prozedurale Generierung von Inhalten stehen eine große Menge unterschiedlicher Techniken und Algorithmen zur Verfügung, die häufig kombiniert werden können, um das gewünschte Ergebnis zu erzielen. Zur Lösung der in Kapitel 2.3 beschrieben Problemstellungen sind folgende Ansätze denkbar:

Für die Unterteilung des Planeten in einzelne Regionen ist die Berechnung eines Voronoi-Diagrams auf Basis von Sites auf der Planetenoberfläche sinnvoll. Die Positionen dieser Sites können beispielsweise mithilfe von Sample Elimination bestimmt werden. Die Verwendung dieses Algorithmus ist sinnvoll, da sich eine Kugeloberfläche nicht mit einem gleichmäßigen Raster unterteilen lässt und eine festgelegte Anzahl Samples generiert werden soll.

Aus den gleichen Gründen kann Sample Elimination auch bei der Verteilung von Ressourcen auf der Planetenoberfläche zum Einsatz kommen.

Bei der Generierung des Terrains ist die Wahl der prozeduralen Methoden weniger eindeutig. Da die Planetenoberfläche nicht in gleich große Regionen unterteilt werden soll, ist der Einsatz von Kacheln nicht sinnvoll. Eher ist eine Mischung aus parametrischer, interpretierter und auf Simulation sowie Noise basierender Generierung denkbar. Aufgrund der Komplexität von Terrain-Generierung wird sie in Kapitel 4 gesondert betrachtet.

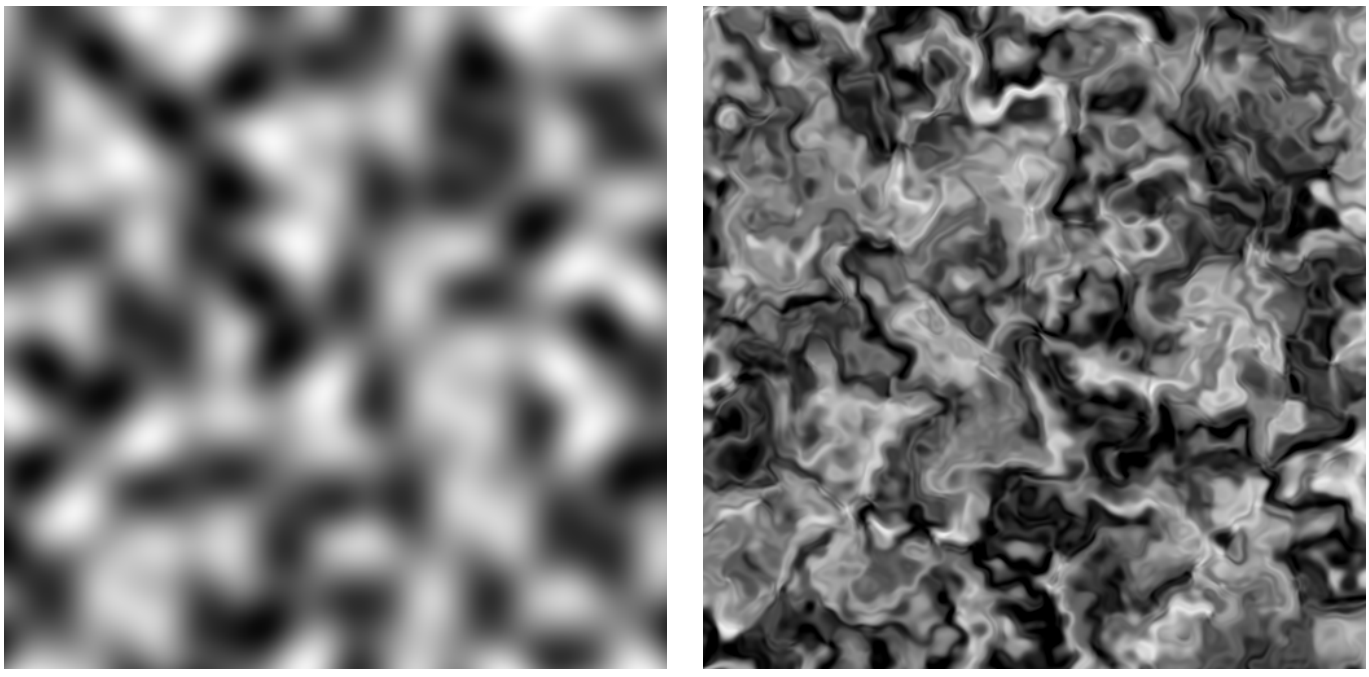


Abbildung 3.45: Einfaches Simplex Noise (links), dessen Eingabewerte mit Domain Warping verzogen werden (rechts).

4 Terrain-Generierung

4.1 Repräsentation

Galin et al. unterscheiden bei der Repräsentation des Terrains *Höhenmodelle*, *volumetrische Modelle* und das *hybride Modell*. Diese werden hier im Detail beleuchtet.

4.1.1 Höhenmodelle

Bei *Höhenmodellen* werden Terrains durch Höhendaten repräsentiert. Diese Höhendaten können entweder aus einer Funktion oder aus einem diskreten Höhenfeld gewonnen werden (siehe Abbildung 4.1). Diskrete Höhenfelder sind eine Sammlung von Höhen, die in einem regelmäßigen 2D-Raster angeordnet sind. Sie begrenzen den Detailgrad des Terrains durch die Auflösung dieses Rasters. Zwischen den einzelnen evaluierten Punkten wird interpoliert. Im Gegensatz zu Evaluierungsfunktionen haben sie einen großen Speicherbedarf von $O(n^2)$.

Terrains auf Basis einer Evaluierungsfunktion hingegen sind ihrem Detailgrad nicht begrenzt. Sie haben einen sehr geringen Speicherbedarf, jedoch kann die Evaluierung in Abhängigkeit der Funktion sehr aufwendig sein.

Ein weiteres Höhenmodell basiert auf unterschiedlichen *Schichten*, die verschiedene Sedimente repräsentieren. Hierbei können sowohl Evaluierungsfunktionen als auch Stapel diskreter Schickten verwendet werden. (siehe Abbildung 4.2). Im Gegensatz zum nachfolgend beschriebenen volumetrischen voxel-basierten Modell ist dieser Ansatz potentiell vertikal unendlich. Das Schichtenmodell findet breiten Einsatz bei Erosionssimulationen.

Ein großer Vorteil von Höhenmodellen ist der potentiell geringe Speicherbedarf durch Evaluierungsfunktionen. Allerdings lassen sich durch Höhenmodelle keine volumetrischen Strukturen wie Überhänge, Höhlen oder Gesteinsbögen darstellen, wodurch ihre Verwendbarkeit eingeschränkt ist.

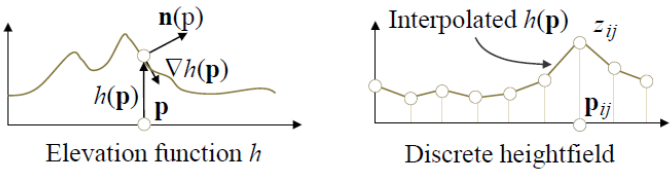


Abbildung 4.1: Die Höhe kann durch eine analytische oder prozedural definierte Funktion oder durch diskrete Höhendaten dargestellt werden, wobei in diesem Fall die Höhe an jedem Punkt durch Interpolation rekonstruiert wird. (Quelle: [GGP*19], S.554)

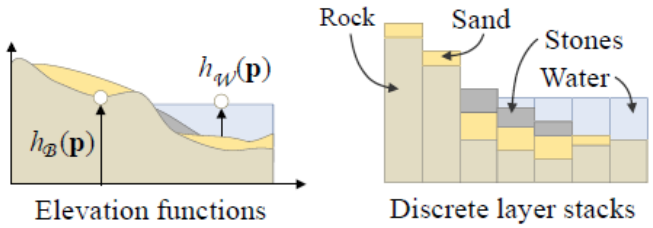


Abbildung 4.2: Schichtmodelle stellen verschiedene Arten von Materialien dar, die in einer vordefinierten Reihenfolge organisiert sind (Grundgestein, dann Sand und Gestein, gefolgt von Wasser). (Quelle: [GGP*19] S.555)

4.1.2 Volumetrische Modelle

Durch *volumetrische Modelle* lassen sich auch Strukturen wie Höhlen, Überhänge und Gesteinsbögen repräsentieren. Eine Form der Repräsentation volumetrischer Daten sind *Voxel*. Hier wird der Raum in ein 3D-Raster aufgeteilt und jeder Zelle ein Material-Index zugewiesen (siehe Abbildung 4.3, links). Dadurch entsteht ein hoher Speicherbedarf von $O(n^3)$, wodurch nicht unbegrenzt hohes Terrain durch Voxel repräsentiert werden kann. Allerdings kann das Terrain effizient berechnet werden.

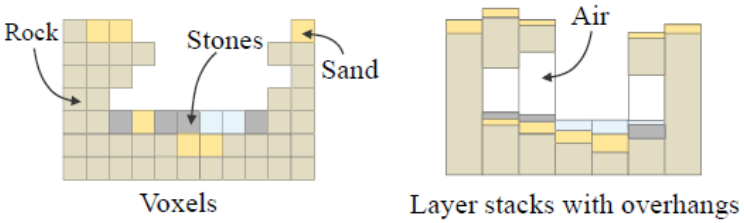


Abbildung 4.3: Voxeldarstellungen erlauben die Modellierung von Bögen, Höhlen oder Überhängen, sind jedoch durch ihre diskrete Natur eingeschränkt. (Quelle: [GGP*19], S.555)

Ein weiterer Ansatz ist die Definition des Terrains als implizite Oberfläche, welche aus einer Feldfunktion extrahiert wird. Ergibt diese Feldfunktion einen positiven Wert für einen Punkt *P*, so liegt *P* innerhalb des Terrains, ist der Wert negativ, so liegt *P* außerhalb. Ein Wert 0 impliziert die Oberfläche. Die durch die Feldfunktion evaluierten Daten müssen anschließend polygonisiert werden, um das Oberflächenmesh zu erhalten.

Das zuvor beschriebene schichtbasiere Höhenmodell kann mit einer Luftschicht erweitert werden, um die Darstellung von Höhlen, Überhängen und Bögen zu ermöglichen (siehe Abbildung 4.3, rechts). Auch hierbei muss anschließend polygonisiert werden. Das schichtbasierte Höhenmodell mit Luftschichten wird auch beim hybriden Modell verwendet.

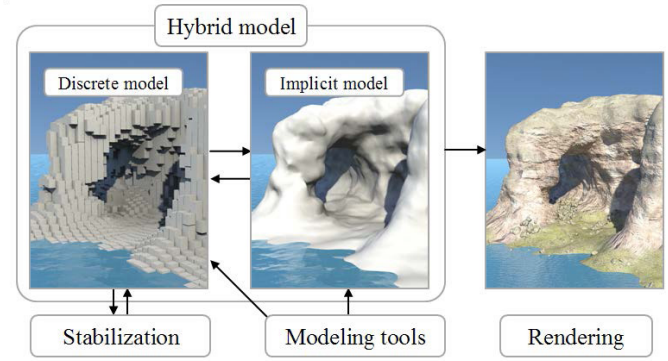


Abbildung 4.4: Übersicht der Architektur des hybriden Terrain Modeling Systems (Quelle: [PGG*09], S. 459)

4.1.3 Hybrides Modell

Das *hybride Modell* wurde von Peytavie et al. [PGG*09] vorgestellt und kombiniert eine diskrete volumetrische Datenstruktur in Form des vorgestellten Schichtenmodells mit einer impliziten Darstellung (Abbildung 4.4). Das erzeugte Terrain ist in die Materialien Grundgestein, Geröll, Sand und Wasser unterteilt, die jeweils eine unterschiedliche Dichte aufweisen. Luftschichten füllen Lücken im Terrain auf. Mit diesem Modell simulieren die Autoren Erosion, Material-Stabilisierung sowie Materialabtragung. Auf das Thema Erosion wird in Kapitel 4.3.2 näher eingegangen.

4.2 Prozedurale Generierungsmethoden

Galin et al. [GGP*19] bezeichnen *prozedurale Methoden zur Generierung von Terrain* als jede Technik, die nicht simuliert ist oder auf Basis von realen Daten arbeitet. Diese Techniken betrachten die reale Welt als Vorbild und versuchen ihre Phänomene nachzubilden. Die Autoren unterscheiden *großflächige Landgenerierung*, *prozedurale Landformmethoden* und *volumetrische prozedurale Terrains*.

4.2.1 Großflächige Landgenerierung

Techniken zur *großflächigen Landgenerierung* fokussieren sich auf die fraktalen und selbstähnlichen Eigenschaften des Reliefs in verschiedenen Maßstäben. Sie bieten im allgemeinen nur indirekte Kontrolle über das generierte Resultat. Galin et al. [GGP*19] beschreiben *Unterteilungsschemata*, *Verwerfungstechniken*, *Methoden auf Basis von Noise* und *Warping*.

4.2.1.1 Unterteilungsschemata

Wie die Autoren beschreiben, verfeinern Unterteilungsschemata mittels *Midpoint Displacement*-Algorithmen ein

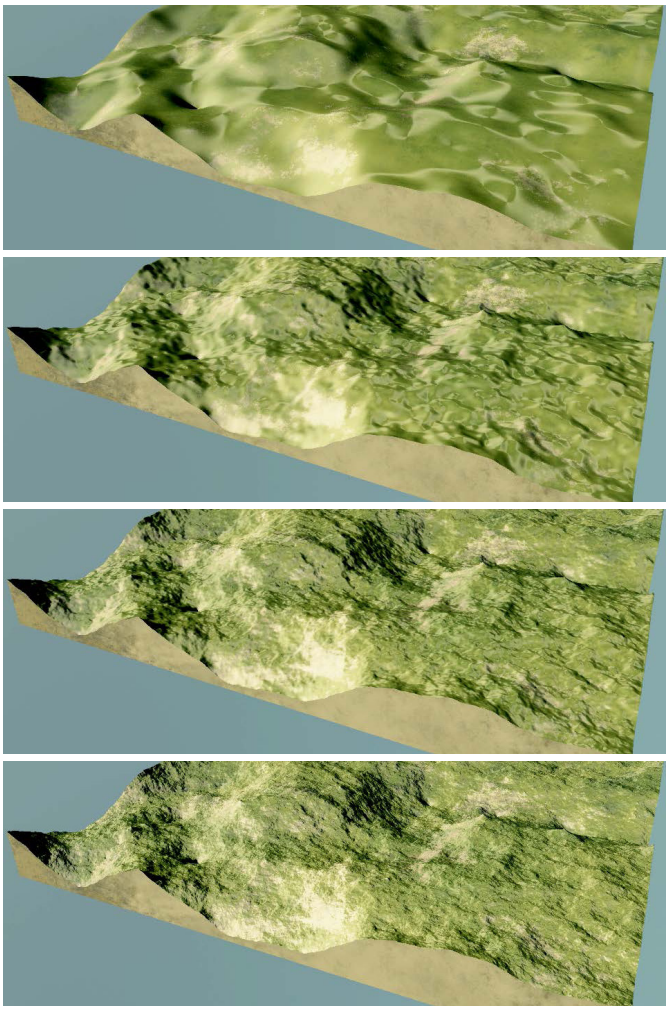


Abbildung 4.5: Mehrere Iterationen (2, 4, 6, 8) des Midpoint Displacement Algorithmus namens *diamond square*. (Quelle: [GGP*19], S.556)

Eingabegelände iterativ, um zunehmende Details zu erzeugen. Dafür werden mit jeder Iteration neue Vertices in der Mitte der Polygone einer Fläche generiert und vertikal um einen (pseudo)zufälligen Wert verschoben. Die Gewichtung der Verschiebung wird mit jeder Iteration reduziert, sodass ein Terrain mit fraktalen Eigenschaften entsteht (siehe Abbildung 4.5).

4.2.1.2 Verwerfungen

Techniken auf Basis von *Verwerfungen* (engl. faulting) gehen von einem flachen Gelände aus, auf dem iterativ zufällige vertikale Verwerfungen aus Linien generiert werden. Punkte auf beiden Seiten werden entsprechend der Entfernung zur Verwerfung nach oben oder unten verschoben. Durch Parameter wie Höhe oder Ausdehnung der Verwerfungen kann das Resultat beeinflusst werden (siehe Abbildung 4.6).

4.2.1.3 Noise und Warping

Durch eine Kombination unterschiedlicher Noisefunktionen mit unterschiedlicher Gewichtung und Amplituden kann nach Galin et al. eine Funktion erstellt werden, deren implizite Oberfläche lokal einem realen Gelände ähnelt. Eine Kombination aus mehreren Oktaven Ridge Noise auf Basis von Simplex Noise resultiert beispielsweise in überzeugende Gebirgszüge (siehe Abbildung 4.7). Mithilfe von Domain Warping kann die Monotonie und Regelmäßigkeit von Noise aufgebrochen und begrenzte Erosionseffekte erzielt werden.

4.2.1.4 Fazit

Wie Galin et al. deutlich machen, haben die genannten Techniken insgesamt betrachtet starke fraktale Eigenschaften, doch können sie keine „high-level" Strukturen von realen Terrains darstellen. Auf Noise basierte Geländemodellierung ist ein mächtiges Werkzeug, das jedoch sowohl technisches als auch künstlerisches Verständnis erfordert. Sie ermöglichen eine große Flexibilität und können eine Vielzahl von Effekten in Kombination mit Domain Warping und anderen Funktionen wie Clamping oder lineare Interpolation erzielen. Punkte können mit Noisefunktionen unabhängig voneinander evaluiert werden. Dadurch ist der Prozess parallelisierbar. Ein Nachteil funktionsbasierter Verfahren ist allerdings, dass sie komplexe Erosionsphänomene, vor allem den Materialtransport, nicht erfassen. Auch ist die Natur von Terrains nach Aussage der Autoren nicht streng fraktal und können daher nicht vollständig durch Algorithmen dargestellt werden, die fraktale Terrains erzeugen.

Für die Umsetzung des Planetengenerators wurde den Noise basierenden Techniken im Vergleich zu den Unterteilungsschemata und Verwerfungsalgorithmen aufgrund der großen Flexibilität der Vorzug gegeben. Das entwickelte modulare System zur Generierung von beliebigen Noisefunktionen wird in Kapitel 7.2 vorgestellt.

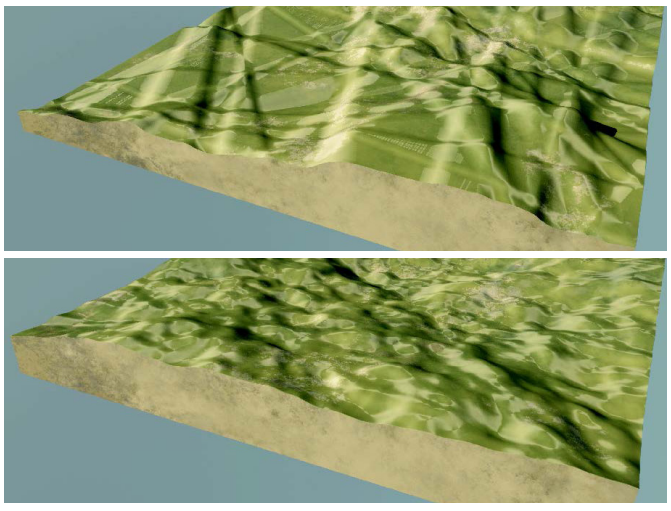


Abbildung 4.6: Der Verwerfungsalgorithmus erzeugt zufällige Verwerfungen als Linien und hebt oder senkt das Gelände auf beiden Seiten der Verwerfungslinie. (Quelle: [GGP*19], S.556)

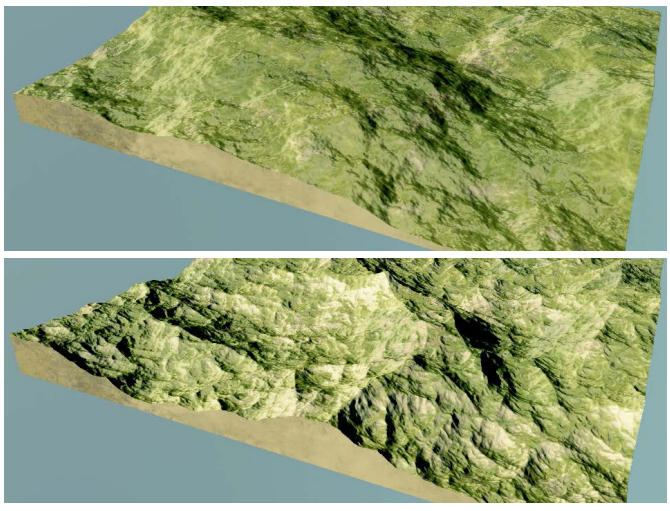


Abbildung 4.7: Eine Summe aus Simplex Noise n (links) und Ridge Noise r (rechts) mit 8 Oktaven: Ridge Noise erzeugt für junge Berge geeignete Kammlinien, während Simplex Noise eher für die Modellierung von Hügeln geeignet ist. (Quelle: [GGP*19], S.557)

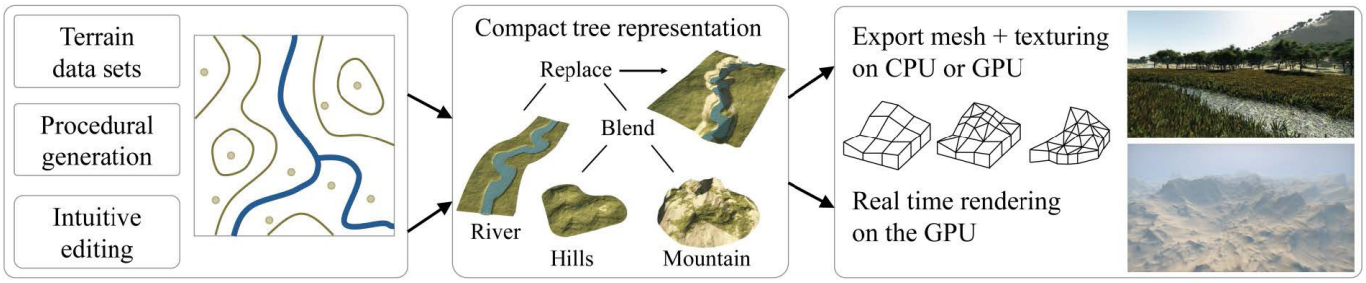


Abbildung 4.8: Die von Génevaux et al. [GGP*15] vorgestellte Generierungspipline auf Basis eines Konstruktionsbaums. Die Autoren kombinieren verschiedene Generierungstechniken zu einem diversen Resultat. (Quelle: [GGP*15], S. 199)

4.2.2 Prozedurale Methoden zur Landformung

Galin et al. stellen neben Techniken zur Generierung von großflächigen Terrains auch Methoden der *prozeduralen Landformung* vor. Diese Verfahrenstechniken zielen darauf ab, spezifische Landschaftsformen wie Flüsse, Klippen oder Canyons lokal zu formen, ohne auf Simulationen oder Synthesen aus Beispielen angewiesen zu sein. Sie arbeiten mit Hilfe von geometrischen Steuerparametern wie Merkmalskurven oder Ankerpunkten. Die Autoren unterscheiden zwischen *kontrollieren Unterteilungen* und *featurebasierten Konstruktionsmethoden*.

4.2.2.1 Kontrollierte Unterteilungen

Bei *kontrollierten Unterteilungen* (engl. controlled subdivisions) wird bei Terraingenerierung mittels Midpoint Displacement Algorithmen die Positionierung neuer Vertices lokal eingeschränkt, um größere Kontrolle über den Prozess zu erlangen. Dadurch können Geländemerkmale wie Gebirgskämme oder Flusstäler berücksichtigt werden.

4.2.2.2 Featurebasierte Konstruktion

Featurebasiere Konstruktion arbeitet entweder auf Basis von *Merkmalskurven* (engl. feature curve) oder auf Basis eines *Konstruktionsbaums*. Die von kurvenbasierten Techniken verwendeten Merkmalskurven spezifizieren bestimmte Landschaften wie Gebirge, Täler und Flussläufe. Sie werden von einer NutzerIn auf die Domäne gezeichnet und markieren beispielsweise Basislinien, Erhebungen oder Grenzen. Das Terrain wird anschließend auf Basis dieser Einschränkungen generiert.

Bei den Konstruktionsbäumen besteht der allgemeine Ansatz darin, die Geländehöhe lokal zu modifizieren, indem verschiedene Arten von Modellen verwendet werden. Génevaux et al. [GGP*15] stellen einen hierarchischen *Konstruktionsbaum* vor, der verschiedene *Primitive* kombiniert, welche eine Vielzahl an Landschaftsformen repräsentieren. Zur Generierung der Primitive verwenden die Autoren sowohl Landschaftsdaten als auch prozedurale Generierung und intuitive Nutzeranpassungen. Die Primitive werden anschließend mithilfe unterschiedlicher Operationen wie „Blend" oder „Replace" miteinander kombiniert. Abbildung 4.8 zeigt die von Génevaux et al. entwickelte Generierungspipeline.

Die prozeduralen Primitive werden von Génevaux et al. als „Skeletal primitives" (dt. Skelett-Primitive) bezeichnet. Sie werden durch ein geometrisches Skelett (Punkt, Segment, Kurve oder Kontur) und eine Reihe von Parametern definiert, welche die Höhen- und Gewichtsfunktionen in Abhängigkeit vom Abstand zum Skelett beschreiben. Daraus leiten sich die von den Autoren beschriebenen Primitive, *Scheiben-Primitiv* (Abbildung 4.9), *Kurven-Primitiv* (Abbildung 4.10) und *Kontur-Primitiv* (Abbildung 4.11) ab. Die Gewichtung nimmt ausgehend vom Zentrum des Primitives stetig ab, um eine glatte Verschmelzung einzelner Primitive zu ermöglichen.

Als Operatoren zur Verschmelzung einzelner Primitive beschreiben Génevaux et al. unter anderem *Blend* (Abbildung 4.12), *Replace* (Abbildung 4.13) und *Addition* (Abbildung 4.14). *Warping* Techniken brechen zusätzlich potentiell monotone Strukturen auf.

Mit diesen Operationen war es den Autoren möglich eine große Bandbreite unterschiedlicher Landschaften wie Kratergebiete, felsige Wüsten, Flusslandschaften und Gebirge zu erzeugen.

4.2.2.3 Fazit

Kontrollierte Unterteilung und kurvenbasierte Konstruktionsmethoden können aufgrund der erforderlichen Nutzereingabe nicht für den Planetengenerator eingesetzt werden. Die Konstruktionsbäume hingegen zeigen großes Potential, diverse und spielerisch interessante Terrains ohne Interaktion einer NutzerIn zu generieren. Dafür müssen die zweidimensionalen Methoden in den dreidimensionalen Raum übertragen werden.

Abbildung 4.9: Scheiben-Primitive ermöglichen die Erzeugung von Landschaftsformen in kleinen kreisförmigen Bereichen. (Quelle: [GGP*15], S. 201)

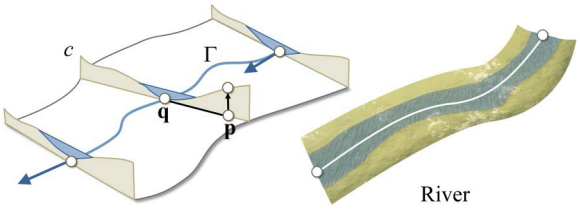


Abbildung 4.10: Kurven-Primitive ermöglichen es uns, auf einfache Weise Geländemerkmale zu erstellen, die um Trajektorien herum strukturiert sind, z. B. Straßen oder Flüsse. (Quelle: [GGP*15], S. 201)

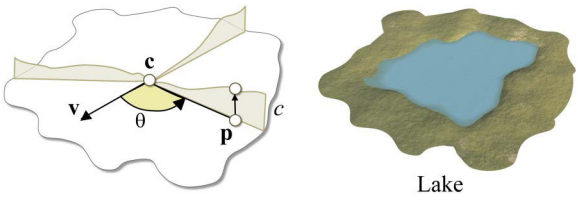


Abbildung 4.11: Komplexere Primitive können mit polygonalen Formen definiert werden. Ein See wird erzeugt, indem eine Reihe von zunehmenden Querschnitten definiert wird, die von der Mitte c ausstrahlen. (Quelle: [GGP*15], S. 201)

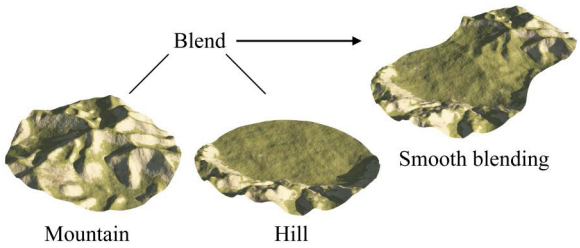


Abbildung 4.12: Der "Blend"-Operator ermöglicht es, mehrere Primitiven auf glatte und kontinuierliche Weise zusammenzufassen. (Quelle: [GGP*15], S. 202)

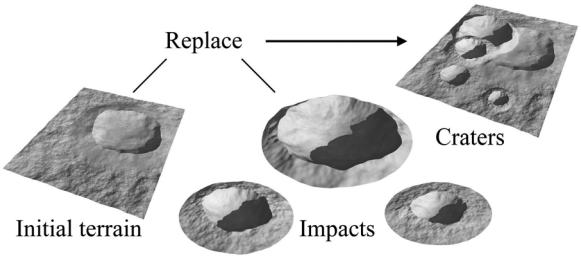


Abbildung 4.13: Der "Replace"-Operator ermöglicht es, die Landschaft auf leichte Art und Weise einzuschränken. Er repräsentiert auch einen Begriff von Zeitlichkeit: Hier entspricht der letzte Operand dem jüngsten Kratereinschlag. (Quelle: [GGP*15], S. 203)

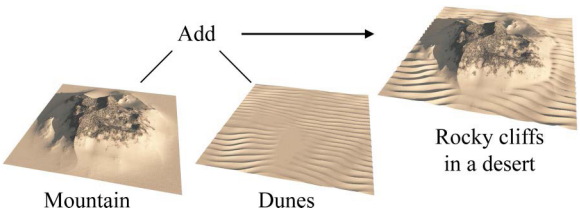


Abbildung 4.14: Der "Add"-Operator ermöglicht es, Wüstendünen mit einer bergigen Landschaft zu kombinieren. (Quelle: [GGP*15], S. 203)

4.2.3 Volumetrische prozedurale Terrains

Galin et al. [GGP*19] merken an, dass wenig Literatur zu *volumetrischen prozeduralen Methoden* existiert. Als eine Möglichkeit zur Erzeugung von Klippen mit Überhängen nennen die Autoren die Definition einer prozeduralen Verschiebungsfunktion, welche die Oberflächenpunkte des Terrains horizontal transformiert.

Eine weitere Möglichkeit präsentieren Peytavie et al. [PGG*09] mit ihrem hybriden Modell mit dem dreidimensionale Terrainstrukturen durch "Luft"-Schichten im Terrain modelliert werden. Durch die implizite Oberfläche können Überhänge, Gesteinsbögen und Höhlen mit glatten Wänder dargestellt werden (Abbildung 4.15).

Als weitere Technik um Terrain mit Überhängen und Gesteinsbögen zu erzeugen, nennen Galin et al. ein von Becher et al. [BKR*19] entwickeltes System. Dieses basiert auf Merkmalskurven im dreidimensionalen Raum und einem zugrunde liegenden Voxel-Raster. Die Kontrollparameter der Kurven geben zusammen mit einer Noise-Funktion die Oberflächenpunkte des Terrains an. Abbildung 4.16 zeigt ein Terrain, welches auf diese Weise erzeugt wurde.

Galin et al. [GGP*19] geben zu bedenken, dass implizit definierte dreidimesionale Terrains deutlich komplexer als Höhenfelder sind und einen rechenintensiven Polygonisierungsschritt erfordern.

4.3 Simulationen zur Formung eines Terrains

Nach Galin et al. [GGP*19] modellieren *Simulationstechniken* die Ursachen und Wirkungen, die sich aus einem Simulationsprozess ergeben. Damit stehen sie im Gegensatz zu den prozeduralen Ansätzen, die sich auf die Generierung eines bestimmten Phänomens fokussieren. Bei Simulationen werden Landschaften aus der Interaktion der simulierten Elemente geformt. Im Zusammenhang mit Terraingenerierung nennen die Autoren *Erosionssimulation* und *Simulation der Plattentektonik*. Bei der Erosionssimulation unterscheiden sie zwischen *thermaler*, *hydraulischer* und *äolischer* (durch Wind verursachte) Erosion. Hinzu kommt die *Simulation von Flussläufen*, wie sie beispielsweise von Peytavie et al. [PDG*19] beschrieben wird.

4.3.1 Simulation von Plattentektonik

Cordonnier et al. [CCB*18] generieren mithilfe von simulierter Plattentektonik Gebirgszüge. Die Platten und ihre Bewegungsrichtung werden durch Nutzereingaben definiert. An den Stellen, wo sich Platten aufeinander zubewegen, wird die Erdkruste gestaucht und angehoben. Die Autoren arbeiteten dabei mit unterschiedlichen Gesteinsschichten. Die tieferen Schichten kommen in den Gebirgsregionen durch Erosion der oberen Schichten zu Tage.

Abbildung 4.15: Gesteinsbögen, die mit dem Hybriden System von Peytavie et al. [PGG*09] generiert wurden. (Quelle: [PGG*09], S.467)

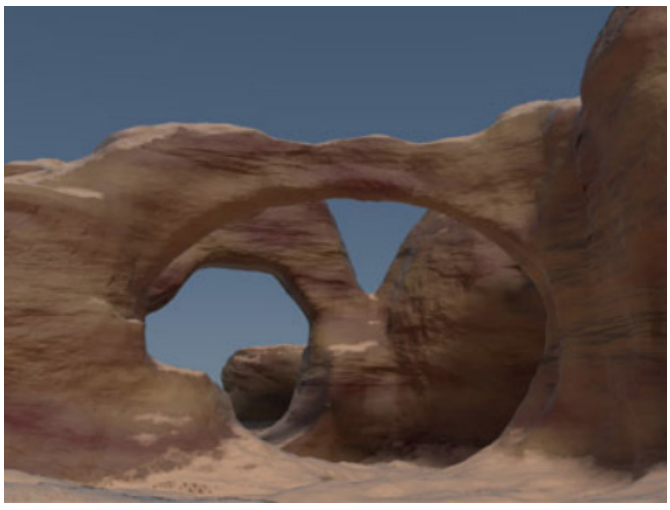


Abbildung 4.16: Ein Terrain mit dreidimensionalen Strukturen, welche durch 3D-Merkmalskurven erzeugt wurden. (Quelle: [BKR*19] S.1294)

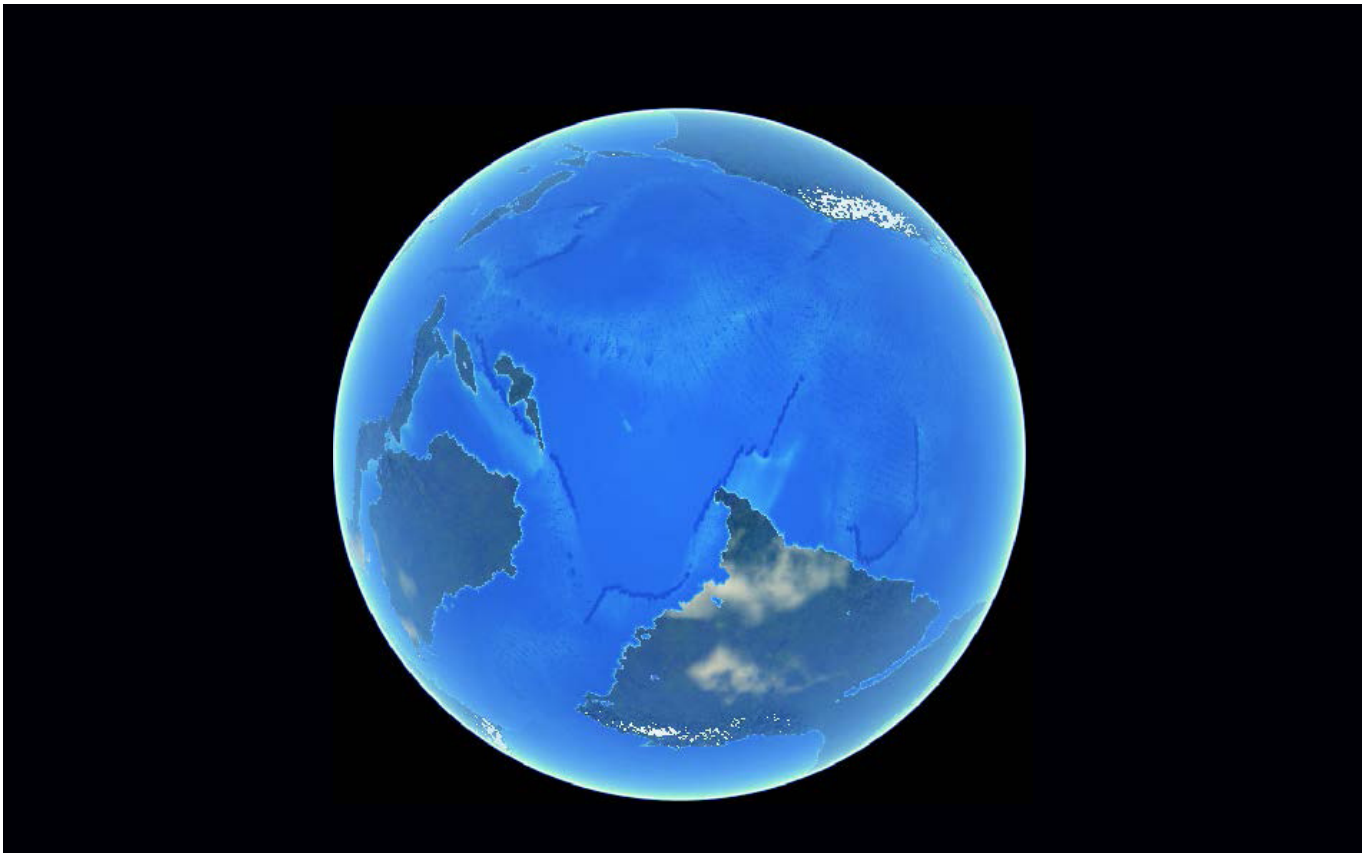


Abbildung 4.17: Ein Planet, welcher von Cortial et al. [CPG*19] mithilfe von simulierten Kontinentalplatten geformt wurde. (Quelle: [CPG*19], S.1)

Cortial et al. [CPG*19] erweitern dieses Prinzip und simulieren die Plattentektonik eines ganzen Planeten (Abbildung 4.17). Die Autoren nähern sich der Deformation der Planetenkruste unter tektonischen Kräften durch ein geologisch inspiriertes, prozedurales Modell an, um realistische Kontinente und Ozeane zu generieren. NutzerInnen können die Kontinentalplatten zu jedem Zeitpunkt der Simulation teilen oder ihre Bewegungsrichtung ändern. Dadurch können beispielsweise sehr alte Planeten simuliert werden. Nach Aussage der Autoren führen die konvergierenden und divergierenden Platten zu Phänomenen wie Kollision, Subduktion, Rifting und der Bildung ozeanischer Krusten. Dadurch entstehen Terrainstrukturen wie weitläufige Gebirge und Inselketten.

4.3.2 Erosionssimulation

Die *Erosion* von Gebirgshängen oder Klippen zusammen mit Materialabtrag ist ein wichtiges Phänomen, um generiertes Terrain realistisch aussehen zu lassen. Da sich Erosion nur sehr schwer mit Noise, Unterteilungsalgorithmik oder Verwerfung des Terrains nachbilden lässt, wurden Techniken entwickelt, um thermale, hydraulische oder äolische Erosion zu simulieren.

Nach Galin et al. [GGP*19] ist Erosion das Resultat von Oberflächenprozessen, bei denen Material an einer Stelle des Terrains abgetragen und und an eine andere Stelle transportiert wird. Die Autoren beschreiben Erosion in drei Schritten, die in Abbildung 4.18 zu sehen sind: *Abtragung*, *Transport* und *Ablage* des Materials. Der Transport erfolgt über einen *Agenten*, beispielsweise Wind oder Wasser. Erosion unterscheidet sich von *Verwitterung* dadurch, dass letzteres nicht durch Bewegung, sondern unter anderem durch Temperaturschwankungen verursacht wird.

4.3.2.1 Thermale Erosion

Wie Galin et al. beschreiben, erfolgt *thermale Erosion* als Kombination von thermaler Verwitterung und hauptsächlich durch die Erdanziehung verursachte Massenbewegung von Gestein und Sedimenten auf Abhängen. Thermale Erosion findet aufgrund von Wasser innerhalb von Rissen im Gestein statt, welches sich aufgrund schwankender Temperatur ausdehnt oder zusammenzieht, wodurch Material abbricht und fällt. Befindet sich das erodierte Material auf einem Abhang, so wird es weiter getragen, sodass ein "Talus" genannter *Schuttkegel* am Fuße des Abhangs entsteht. Dieser ist in Abbildung 4.19 zu erkennen.

Die Simulation von thermaler Erosion kann sowohl auf Höhenfelder aus auch auf volumetrische Terrainmodelle angewendet werden. Letzteres zeigen Peytavie et al. [PGG*09], die mit ihrem hybriden Modell Überhänge durch die Simulation von thermaler Erosion erzeugen. Sie arbeiten dabei mit den Materialien Gestein, Geröll und Sand. Erodiertes Gestein wird mit einem bestimmten Verhältnis in Geröll und Gestein umgewandelt, welches sich schließlich über den Boden verteilt.

Galin et al. [GGP*19] merken an, dass bei der Simulation von thermaler Erosion Probleme auftreten. Zum Einen wird die Simulation zumeist als iterativer Prozess implementiert und erfordert daher Zeitsprünge im Simulationsprozess. Auch kann erodiertes Material in der Realität durch den Fall und die Bewegung über das Terrain in weitere Einzelteile zerbrechen und dadurch den Simulationsprozess deutlich erschweren. Als dritte Einschränkung nennen die Autoren die wahrgenommene unnatürliche Regelmäßigkeit der aus der Simulation entstehenden Schuttkegel.

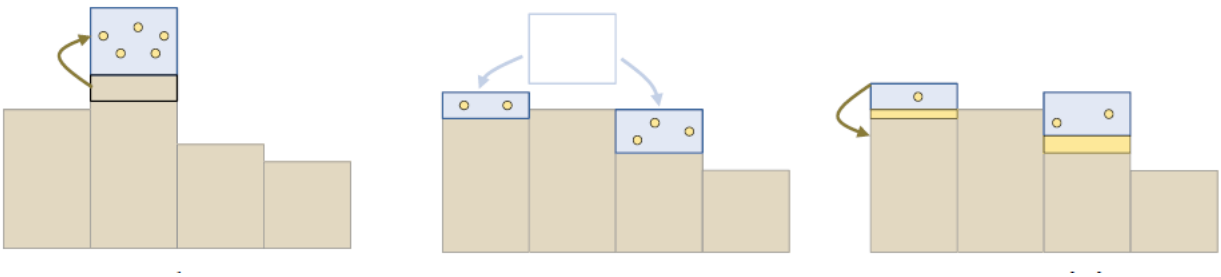


Abbildung 4.18: Die drei Schritte der Erosion: Abtragung (links), Transport (mittig) und Ablage (rechts). (Quelle: [GGP*19], S. 561)

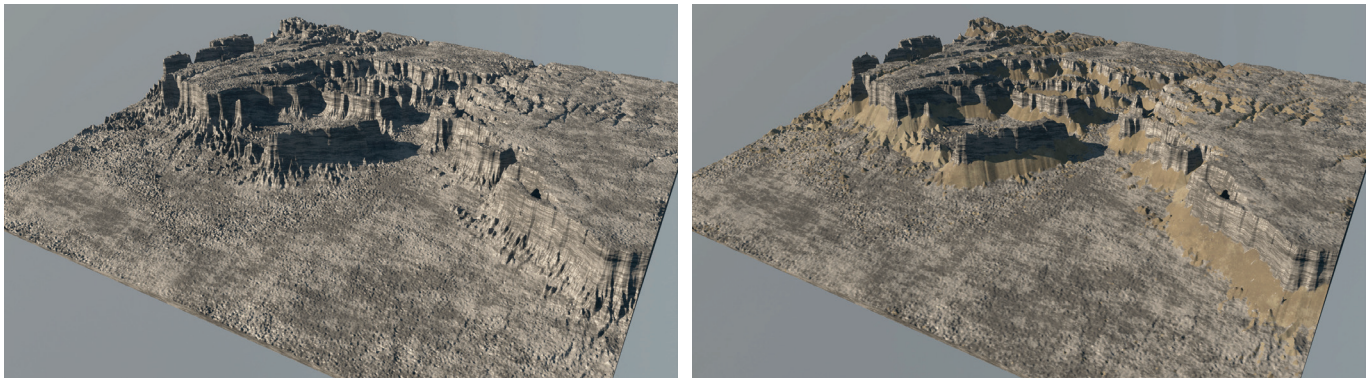


Abbildung 4.19: Ein Terrain aus Gestein (links) auf das ein Simulation thermaler Erosion angewendet wurde (rechts). Sedimente haben sich als Resultat an den Hängen des Terrains abgelagert. (Quelle: [GGP*19], S. 561)

4.3.2.2 Hydraulische Erosion

Nach Galin et al. [GGP*19] erfolgt *hydraulische Erosion* durch die Bewegung von Wasser gegen Grundgestein. Der durch diese Bewegung erzeugte Materialabtrag wird vom Wasser transportiert. Abbildung 4.20 zeigt ein Terrain, welches mittels Simulation von hydraulischer Erosion verändert wurde.

Die Autoren führen aus, dass auf die hydraulische Erosion sowohl mechanische Prozesse wie Abrieb oder Korrasion als auch chemische Prozesse, welche die Zusammensetzung des Gesteins verändern, Einfluss nehmen. Diese Prozesse werden entweder auf Basis eines diskreten rasterbasierten Modells und physikbasierter Simulation oder mithilfe von Partikeln simuliert. Beim ersten Ansatz wird der Druck, die Geschwindigkeit und die Flüssigkeitsmenge für jede einzelne Zelle des zugrundeliegenden Rasters berechnet. Bei partikelbasierten Ansätzen wird Strömung als Ansammlung von Partikeln unter Einfluss hydrodynamischer und Gravitationskräften simuliert. Durch eine gleichmäßige Verteilung von Partikeln über ein Terrain kann beispielsweise Niederschlag simuliert werden.

Wie Galin et al. anmerken, ist die Simulation von Flüssigkeiten insbesondere in Kombination mit hydraulischer Erosion sehr rechenaufwendig, wodurch entsprechende Techniken meist nur für Domänen mit kleinen Ausmaßen durchgeführt werden kann. Auch ist die Kontrolle über den Erosionsprozess begrenzt. Ist das Resultat einer Simulation nicht zufriedenstellend, so muss die Ausgangssituation verändert und der Prozess erneut gestartet werden.

4.3.2.3 Andere Erosionsphänomene

Nach Galin et al. [GGP*19] erfahren andere Erosionsphänomene neben thermaler und hydraulischer Erosion wenig Beachtung durch die Computergrafik. Zu diesen Phänomenen zählen *äolische*, *Küsten-*, *Blitz-* sowie *Gletschererosion*. Gletschererosion wirkt in einer kürzeren Zeitspanne als hydraulische Erosion und führt zur Bildung U-förmiger sowie hängende Täler. Küstenerosion erzeugt volumetrische Strukturen wie Überhänge und und Höhlen. Die äolische Erosion wird durch Winde verursacht, die Materialien abtragen, transportieren und ablagern. Insbesondere Wüsten

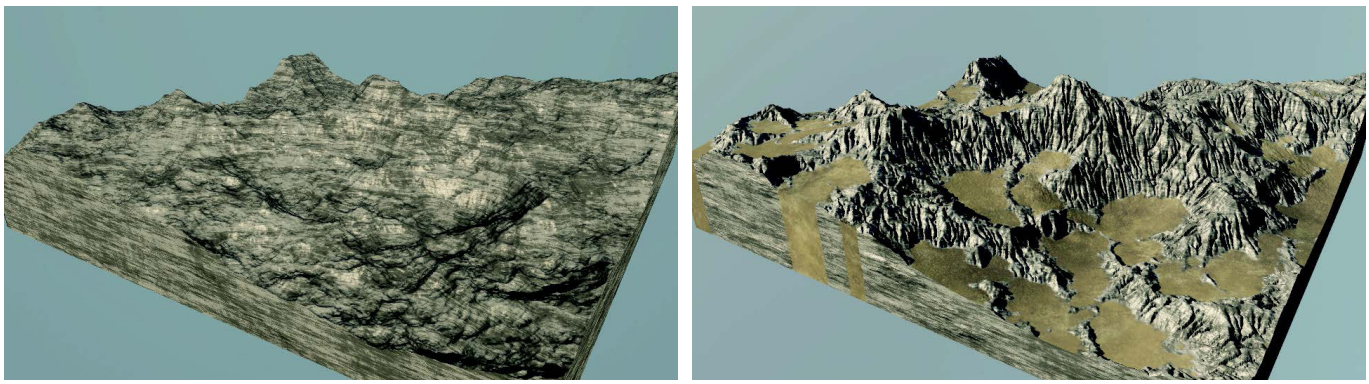


Abbildung 4.20: Ein fraktales, prozedural generiertes Terrain aus Gestein (links) auf das eine Simulation hydraulischen Erosion angewendet wurde (rechts). (Quelle: [GGP*19], S. 563)

mit Sanddünen sind davon betroffen, da durch Wind getragene Sandpartikel verstärkt Material von Gestein abtragen. Auch Blitze können Erosionseffekte hervorrufen, indem sie durch Einschläge eine große Menge Grundgestein zerstören und Felsen mehrere Meter weit schleudern.

4.3.3 Simulation von Flussläufen

Die *Simulation von Flussläufen* mit Wasserfällen, Zuflüssen und Deltas ist ein weiteres Forschungsgebiet der Computergrafik. Peytavie et al. [PDG*19] stellen in ihrer Arbeit eine Methode vor, entsprechende Flusslandschaften zu erzeugen. Als Eingabe nehmen die Autoren ein digitales Höhenmodell, werten seine hydrologischen Eigenschaften aus und bestimmen darauf aufbauend den Verlauf des Flussnetzes. Abbildung 4.21 zeigt eine durch die Simulation geformte, Landschaft.

Die Pipeline des Simulationsprozesses ist in Abbildung 4.22 zu sehen. Nach Peytavie et al. wird das Eingabeterrain zunächst analysiert und Daten zur Neigung, Entwässerungsgebieten und Stromstärke abgeleitet. Anschließend wird der Flusstyp und entsprechend die Flussgeometrie festgelegt. Dadurch entstehen beispielsweise schmale, schnelle Flüsse oder Flüsse mit vielen Schleifen und Wendungen. Anhand dieser Festlegung wird ein Flussnetzgraph mit Wasserflusswerten pro Zelle für Neigung, Volumen und Geschwindigkeit generiert. Das Flussbett wird entsprechend in das Terrain eingearbeitet. Die Autoren verfeinern das entstandene Flussnetz durch lokale animierte Primitive wie Wellen, Whirlpools oder Kaskaden auf Basis seiner Flussdaten und seiner Geometrie. Die Primitive werden als hierarchische Baumstruktur zusammengefasst, welche die animierte Wasseroberfläche als prozedurale Funktion definiert. Nach Aussage der Autoren kann diese Funktion ohne Simulation zu jeder Zeit und für jedem Punkt evaluiert werden. Im letzten Schritt wird das Terrain gerendert. Das Resultat der Pipeline ist eine Landschaft mit einem Flussnetz, welches aus der Flussbettgeometrie und einer animierten Wasseroberfläche zusammengesetzt ist.

Abbildung 4.21: Ein Eingabegelände (links) wird durch die Simulationstechnik von Peytavie et al. [PDG*19] geformt, sodass ein Flussnetzwerk entsteht (rechts). (Quelle: [PDG*19], S.36)

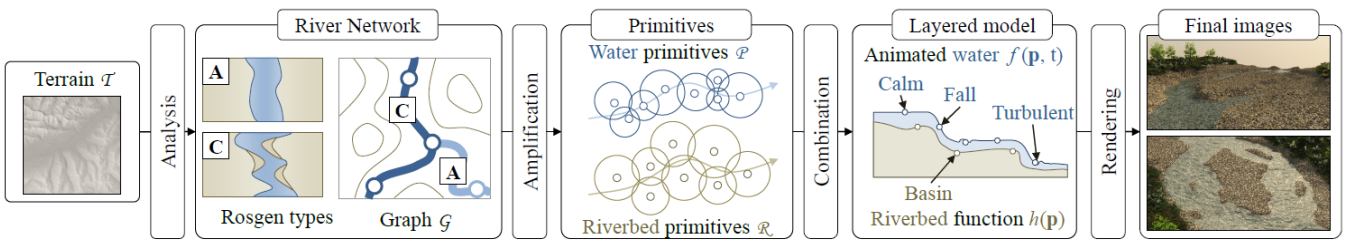


Abbildung 4.22.: Die Pipeline der von Peytavie et al. [PDG*19] präsentierten Technik zur Simulation von Flussnetzen. (Quelle: [PDG*19], S.37)

4.4 Terraingenerierung in Spielen

In den letzten fünfzehn Jahren werden prozedurale Methoden verstärkt für die Generierung von Terrain und im Speziellen von Planeten in Videospielen genutzt. Zum Einen erlangen Videospiele wie *Minecraft* [S12], welche mit jedem Durchlauf neue Spielwelten generieren, einen hohen Wiederspielwert. Zum Anderen macht eine große Menge an Inhalten die Anwendung von prozedurale Techniken erforderlich. In *Spore* [S20] oder *No Man's Sky* [S13] beispielsweise können die SpielerInnen eine ganze Galaxie mit einer entsprechend großen Anzahl Sonnensysteme und Planeten besuchen. Zudem hat die zunehmende Rechenleistung von Heimrechnern auch aufwendige prozedurale Techniken interessant für die Entwicklung von Videospielen gemacht.

4.4.1 Spore

Die Art und Weise, wie in Spielen prozedurales Terrain generiert wird, ist dabei zum Teil sehr unterschiedlich. Wie bereits in Kapitel 3.4.1 erwähnt, nutzten die EntwicklerInnen von *Spore* [S20] ein Partikelsystem, um die Planeten des Spiels zu generieren [CGG*07]. Die Datenbasis bilden Würfeloberflächen mit denen jeweils die Textur, die Normalen und die Höheninformationen des Terrains abgebildet werden (Abbildung 4.23). Durch Projektion eines Punktes auf der Kugeloberfläche wird die entsprechende Koordinate auf der Oberfläche der Würfel ermittelt. Die Höhenfelder repräsentieren die Oberfläche eines Planeten. Nach Compton et al. werden die Höhenfelder mit einem Pinselsystem bearbeitet, welches in Form texturierter Rechtecke die Höhenwerte anhebt oder senkt. Das bereits genannte Partikelsystem bewegt die Pinsel, gesteuert durch pseudozufällige Parameter, über die Planetenoberfläche. Die resultierenden Strukturen ähneln Flussläufen, Schluchten, Ozeanen und Plateaus. Mit diesem System ist es nach Angaben der Autoren möglich, bis zu vier Milliarden unterschiedlicher Planeten zu generieren. Eine Auswahl davon ist in Abbildung 4.24 zu sehen.

4.4.2 Minecraft

Minecraft [S12] hingegen verwendet ein volumetrisches Terrainmodell in Kombination mit mehreren Schichten Noise [Pe11]. Das Terrain ist aus unterschiedlichen, gleich

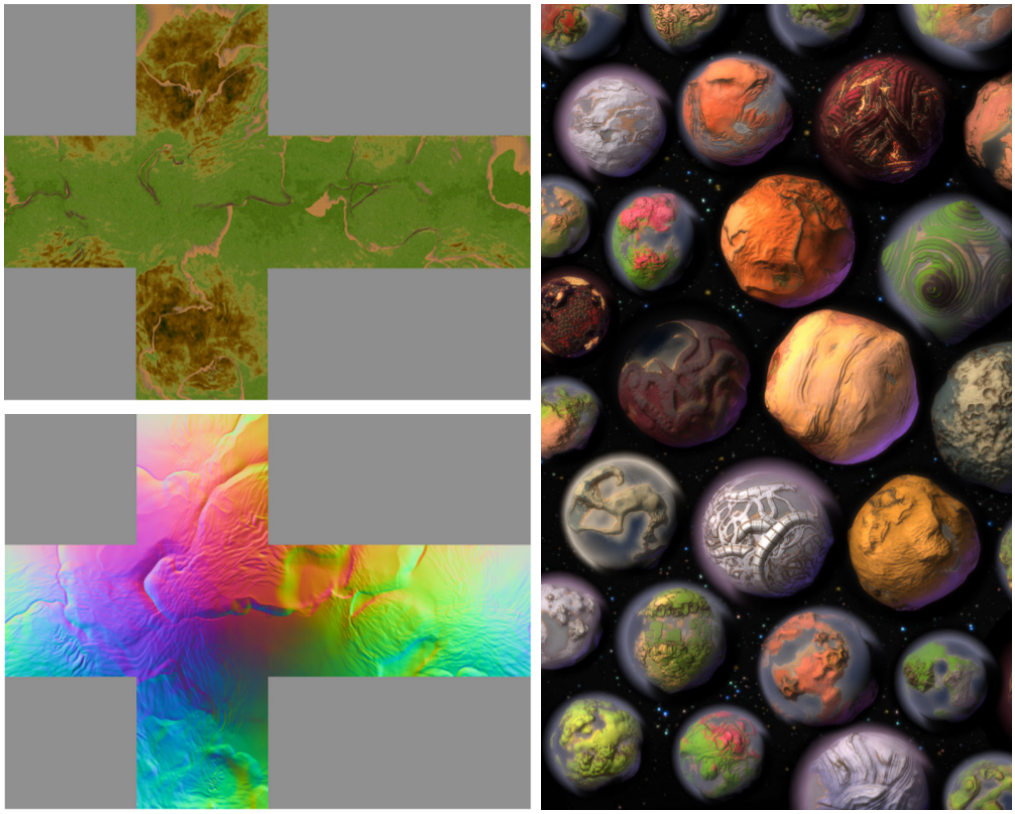


Abbildung 4.23: Compton et al. [CGG*07] verwendeten bei der Entwicklung von *Spore* [S20] Würfeloberflächen, um die Textur- (links), Normalen- (rechts), und Höheninformationen der Planeten zu speichern. (Quelle: [CGG*07], S.82)

Abbildung 4.24: Eine Auswahl von vier Milliarden möglicher Planeten in *Spore* [S20]. (Quelle: [CGG*07])

Abbildung 4.25: Die Spielwelten von Minecraft sind aus diversen Landschaften wie (von links nach rechts) Ebenen, Flussläufen, Schluchten, Wäldern und Höhlen aufgebaut.

großen Blöcken aufgebaut, sodass sich auch ein Kachel-basierter Ansatz erkennen lässt. Nach Persson [Pe11], dem leitenden Entwickler von *Minecraft*, ist eine Spielwelt von Videospiels nahezu unendlich groß und wird laufend um die Spielfigur herum generiert. Dafür verwendet *Minecraft* sogenannte *Chunks* aus 16x16x128 Blöcken, die das Terrain unterteilen und den von ihnen repräsentierten Abschnitt bei Bedarf durch die Evaluierung der Noisefunktionen erzeugen. Persson beschreibt die Verwendung von 3D Perlin Noise zur Bestimmung eines Dichtewerts pro Terrain-Block. Werte unter *0* werden als Luft interpretiert und Werte gleich oder über *0* repräsentieren Boden. Auf Basis dieses Systems werden in *Minecraft* Hügellandschaften, Schluchten, Höhlensysteme, Gebirge und Flussläufe erzeugt (Abbildung 4.25). Zusätzlich wird das Terrain in verschiedene Biome wie Wüste, Ebene, Sumpf und Wald unterteilt, welche jeweils eigene Flora, eigene Terrainstrukturen und unterschiedliche Materialien aufweisen.

4.4.3 No Man's Sky

Auch in *No Man's Sky* [S23] werden nach Sean Murray, dem leitenden Entwickler des Spiels, 3D Terrains mit Höhlen und Überhängen auf Basis von mehreren Noise-Schichten erzeugt [Mu17]. Das Terrain bildet die Oberfläche von Planeten mit Millionen km^2, welche mit Gebäuden, Wäldern und Kreaturen bevölkert und prozedural texturiert wird (Abbildung 4.26). Nach Murray wurden die Planeten in *No Man's Sky* mit dem Ziel generiert, die SpielerInnen zu überraschen, ihr Interesse zu wecken und gleichzeitig spielbar zu bleiben. Wie Innes McKendrick [Mc17] erklärt, wurde dafür folgendes System zur Generierung des Terrains entwickelt. Die Voxeldaten des Terrains werden in einer Würfelstruktur gespeichert und auf die Kugeloberfläche des Planeten projiziert. Die eigentliche Simulation erfolgt auf der gebogenen Planetenoberfläche. Der simulierte Bereich beträgt nach McKendricks Angaben 128 Höhenmeter über der Kugeloberfläche. Für diesen Bereich werden Voxeldaten erzeugt. Terrain unterhalb dieser Grenze ist unveränderliches Grundgestein. Um Höhen für Gebirge oder Tiefen für Ozeane über diese 128m hinaus zu erhalten, wird auf das Grundgestein ein zusätzlicher noisebasierter Höhenwert addiert. Dadurch können Höhenunterschiede von bis zu einem Kilometer entstehen. Jeder Voxel des Terrains besteht aus 6 Bytes, welche Informationen zur Dichte, zwei Materialtypen und einem Übergangswert zwischen diesen Materialien enthalten. Voxel werden zu Chunks von 36x36x36 zusammengefasst, die unabhängig und parallel geladen werden können. Die Chunks können im Zuge eines *Level of Detail*-Systems (kurz *LOD*) weiter unterteilt werden und so detailliertes Terrain erzeugen. Chunks mit einer höheren Voxel-Dichte werden insbesondere in der Nähe des Kamera generiert. Nach McKendrick [Mc17] erfolgt die Generierung des Terrains einer Voxel-Region zur Laufzeit des Spiels mit folgender Pipeline:

Abbildung 4.26: Das Terrain der Planeten von No Man's Sky wird mit einer eigens entwickelten Pipeline und verschiedenen Noise-Schichten generiert.

1. **Generierung**. Die Voxeldaten des Terrains werden mit mehreren Noise-Schichten, allgemeinen Planetendaten wie Art der Atmosphäre sowie einer Unterteilung der Planetenoberfläche in Regionen wie Gebirge oder Flusslauf generiert.
2. **Polygonisierung**. Das Mesh des Terrains wird mithilfe der Voxelposition und -dichte durch Polygonisierung erzeugt.
3. **Sphärifizierung**. Bei der Sphärifizierung werden die Voxel auf die gebogene Oberfläche des Planeten projiziert und somit die eigentlichen Positionen der Voxel bestimmt.
4. **Physik Konstruktion**. Nach der Sphärifizierung wird das Kollisionsmesh des Terrains erzeugt.
5. **KI-Wissen Konstruktion**. Anschließend werden auf Basis der Terrain- und Physik-Daten die Bewegungsmuster der Tiere des Planeten errechnet.
6. **Dekoration**. Im letzten Schritt werden Objekte mithilfe eines Oberflächengitters mit Versatz platziert (vgl. Kapitel 3.3.3). Zellen dieses Gitters werden pseudozufällig ausgewählt und mit einer bestimmten Anzahl, Pflanzen, Tieren oder Gebäuden gefüllt. Kleine, dichte Objekte wie Gräser oder Büsche werden mit einer Noisefunktion verteilt.

5 Generierung eines Kugelmesh

Für die Generierung von Planeten wird ein *sphärischer Grundkörper* in Form eines Kugelmesh benötigt. Kugelmesh mit gleichmäßig verteilten Vertices zu erzeugen ist keine triviale Aufgabe und ein Forschungsgebiet der Mathematik und Computergrafik. Es gibt nur wenige Körper wie der Würfel oder der Ikosaeder, deren Ecken sich mit gleichen Abständen auf eine sie umschließende Kugeloberfläche verteilen. Diese könne als Grundkörper für die Generierung einer Sphäre dienen. Sie werden so lange unterteilt, bis der gewünschte Detailgrad des Meshes erreicht ist. Im Anschluss werden die einzelnen Vertices auf die Kugeloberfläche projiziert.

5.1 Anforderungen an die Generierungsmethode

Ziel ist es, ein möglichst homogenes Mesh mit gleichmäßigen Vertex-Abständen zu erzeugen. Unregelmäßige Abstände und eine unregelmäßige Verteilung der Vertexpositionen führen zu einem variablen Detailgrad der Kugeloberfläche und damit des Planetenterrains. Auch sollte der Generierungsprozess möglichst geringe Leistungsanforderungen aufweisen. Somit ergeben sich folgende Anforderungen an die Generierungsmethode des Kugelmeshes:

- eine möglichst gleichmäßige Vertexdichte
- eine möglichst gute Performance der Generierungsprozesses
- eine regelmäßige Verteilung der Vertex Positionen

5.2 Kugelmesh auf Basis eines Würfels

Eine einfach Methode der Kugelmesh-Generierung stellt die Projektion der Polygone eines *Würfels* auf die Oberfläche einer Kugel dar. Dazu werden die sechs Flächen des Würfels geteilt, bis die gewünschte Auflösung erreicht ist. Das Thema Unterteilungen wird in Kapitel 6 detaillierter behandelt. Je feiner die Oberfläche unterteilt wird, desto gleichmäßiger wird die resultierende Kugeloberfläche. Jeder Vertex des Würfelmesh wird anschließend auf die Kugel mit gewünschtem Radius projiziert. Dafür wird der Ortsvektor des Vertex v_w normalisiert und mit dem Radius r der gewünschten Kugel multipliziert. Daraus resultiert die neue Position des Vektors v_k. Das Zentrum des Würfels befindet sich im Ursprung.

$$v_k = |v_w| * r$$

Die Abbildung 5.1 zeigt generierte Kugelmeshes auf Basis eines Würfels mit unterschiedlichen Unterteilungen der Oberfläche. Die gelben Linien markieren den maximalen Vertex Abstand, die violetten Linien stellen den minimalen Abstand dar (Diagonalen nicht mit eingeschlossen). Tabelle 5.1 zeigt die jeweils resultierenden Abstandsabweichungen der Vertexpositionen. Das relative Verhältnis zwischen minimalen und maximalen Abständen steigt mit zunehmenden Unterteilungen.

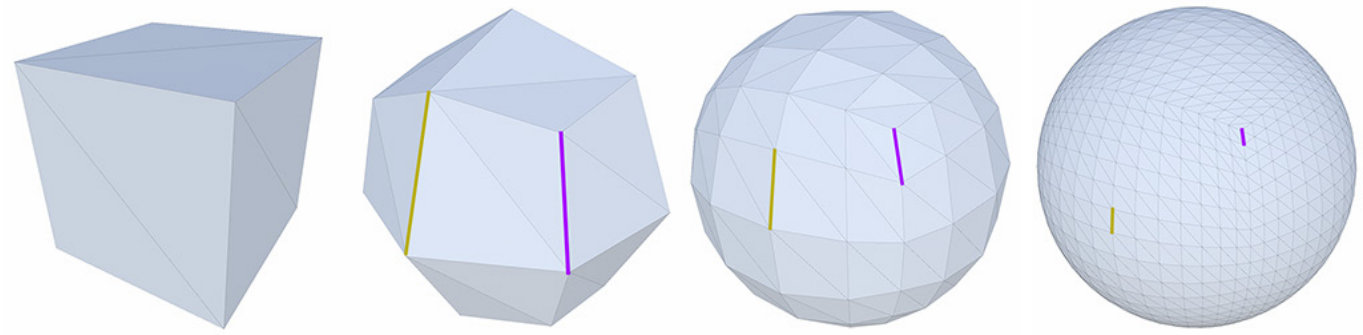


Abbildung 5.1: Ein Kugelmesh auf Basis eines Würfels mit 0, 1, 4 und 16 Unterteilungen. Die gelben Linien zeigen den größten Vertex-Abstand an, die violetten Linien den kleinsten.

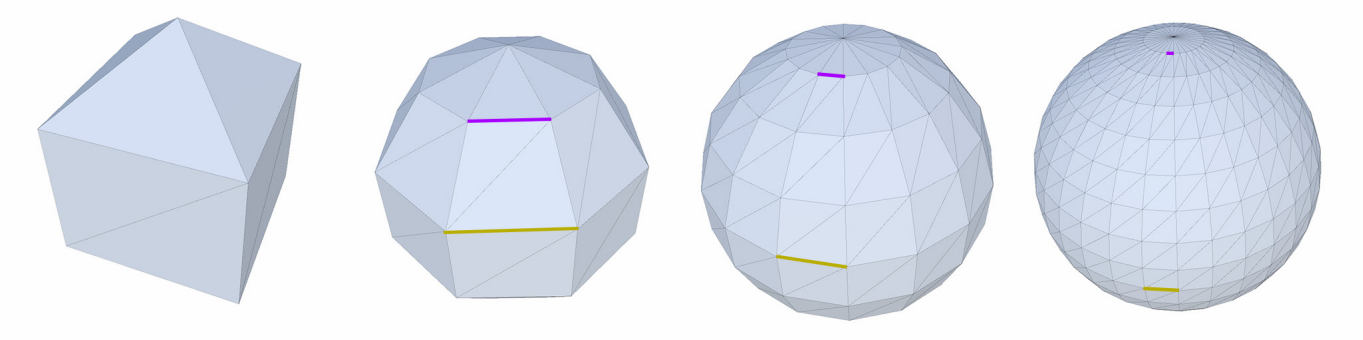


Abbildung 5.2: Eine UV-Sphäre mit [2 | 4], [4 | 8], [8 | 16] und [16 | 32] Breitengraden bzw. Längengraden. Die gelben Linien zeigen den größten Vertex-Abstand an, die violetten Linien den kleinsten.

Einige Programme, darunter die *Unity Engine* [S24], arbeiten nur mit 3D-Modellen, die aus Dreiecken aufgebaut sind. Darum werden die viereckigen Seiten des Würfels an einer Diagonale geteilt.

Unterteilungen	1	4	16
minimaler Abstand (Meter)	≈ 0,6058	≈ 0,2138	≈ 0,0577
maximaler Abstand (Meter)	≈ 0,9194	≈ 0,3849	≈ 0,1172
Verhältnis (max / min)	≈ 1,5176	≈ 1,8003	≈ 2,0312

Tabelle 5.1: Die minimalen und maximalen Vertex Abstände eines auf einem Würfel basierenden Kugelmeshes bei verschiedenen Unterteilungen.

5.3 UV-Kugelmesh

Die Mesh Struktur einer *UV-Sphäre* erinnert an Längen- bzw. Breitengrad-Linien auf einem Globus. Die Generierungsmethode verwendet dabei ein Längengrad-Breitengrad-Gitter [Go10]. Dadurch haben die Breitengrade regelmäßige Abstände auf der Kugeloberfläche. Die Längengrade hingegen laufen in den Polen zusammen, wodurch die Vertices um die Pole besonders dicht gepackt sind. Das zeigt auch Tabelle 5.2: Eine hohe Unterteilung des Meshes führt zu einem großen relativen Unterschied zwischen maximalem und minimalem Vertexabstand. Abbildung 5.2 zeigt UV-Sphären mit unterschiedlichen Breiten- und Längengraden.

Breiten-/Längengrade	4 | 8	8 | 16	16 | 32
minimaler Abstand (Meter)	≈ 0,4499	≈ 0,1334	≈ 0,0360
maximaler Abstand (Meter)	≈ 0,7279	≈ 0,3843	≈ 0,1952
Verhältnis (max / min)	≈ 1,6180	≈ 2,8794	≈ 5,4190

Tabelle 5.2: Die minimalen und maximalen Vertex Abstände eines UV-Sphären-Meshes bei verschiedenen Breiten- und Längengraden.

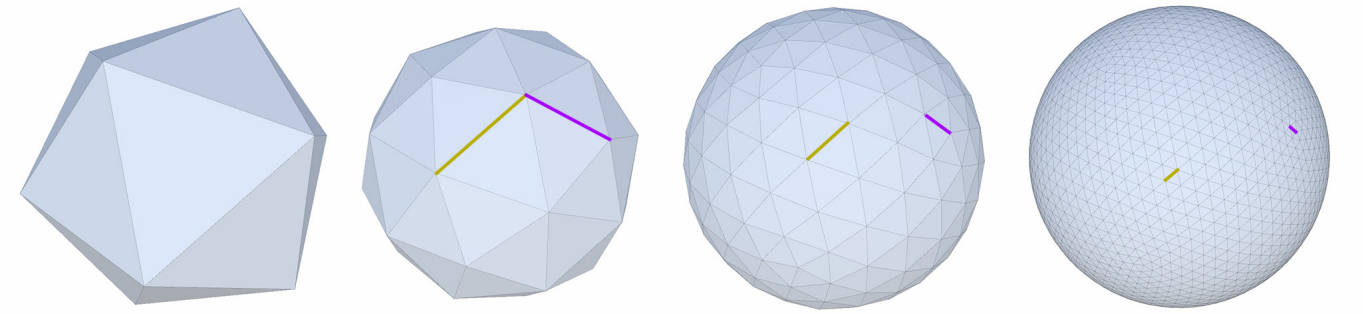


Abbildung 5.3: Eine Ikosphäre mit 0, 1, 4 und 16 Unterteilungen. Die gelben Linien zeigen den größten Vertex-Abstand an, die violetten Linien den kleinsten.

5.4 Kugelmesh auf Basis eines Ikosaeders

Eine weitere Variante der Generierung eines Kugelmeshes basiert auf dem *Ikosaeder* als Grundform. Dessen 20 Dreiecksflächen werden unterteilt, bis die gewünschte Auflösung erreicht wurde und anschließend auf die Kugeloberfläche projiziert. Wie Abbildung 5.3 und Tabelle 5.3 zeigen, ist die relative Abstandsdifferenz der Vertices bei der entstehenden *Ikosphäre* verglichen mit den zuvor präsentierten Methoden gering. Die Vertices sind sehr gleichmäßig auf der Kugeloberfläche verteilt.

Unterteilungen	1	4	16
minimaler Abstand (Meter)	≈ 0,5465	≈ 0,1981	≈ 0,0543
maximaler Abstand (Meter)	≈ 0,6180	≈ 0,2616	≈ 0,0778
Verhältnis (max / min)	≈ 1,1308	≈ 1,3202	≈ 1,4314

Tabelle 5.3: Die minimalen und maximalen Vertex Abstände eines auf einem Ikosaeder basierenden Kugelmeshes bei verschiedenen Unterteilungen.

5.5 Fibonacci-Sphäre

Ein weiterer Ansatz zur Erzeugung eines gleichmäßigen Kugelmeshes ist die *Fibonacci-Sphäre*. Hier wird ein Fibonacci-Gitter auf die Oberfläche einer Kugel projiziert. Dabei haben die Oberflächenpunkte im Vergleich zu einem Breitengrad-Längengrad-Gitter sehr ähnliche Abstände zueinander, wie Abbildung 5.4 veranschaulicht.

González [Go10] erklärt, dass die Punkte des *Fibonacci-Gitters* entlang einer eng gewundenen, generativen Spirale angeordnet sind, wobei jeder Punkt in die größte Lücke zwischen den vorherigen Punkten passt. Diese Spiralen sind in der resultierenden Sphäre nicht zu erkennen, da aufeinanderfolgende Punkte weit voneinander entfernt sind.

Das Fibonacci Gitter ist nach dem Fibonacci-Verhältnis benannt [Go10]. Dieses ergibt sich aus der *Fibonacci Sequenz*, bei der jeder Term ab dem dritten die Summe der vorherigen zwei ist. Daraus ergibt sich folgende Zahlensequenz: 0, 1, 1, 2, 3, 5, 8, 13, …. Bei zwei aufeinanderfolgenden Termen, F_i und F_{i+1}, beträgt das Fibonacci-Verhältnis F_i / F_{i+1}. Das Fibonacci-Verhältnis nähert sich für sehr große i dem Goldenen Schnitt an (≈ 1.1618). Das Fibonacci-Gitter unterscheidet sich von anderen Spiralgittern auf der Kugel dadurch, dass die Längsdrehung zwischen aufeinanderfolgenden Punkten entlang der generativen Spirale der *goldene Winkel* ($360° \Phi^{-2} \approx 137.5°$) oder sein Gegenstück ($360° \Phi^{-1} \approx 222.5°$) ist. González beschreibt, dass der goldene Winkel die Packungseffizienz von Elementen in Spieralgittern optimiert, wodurch Regelmäßigkeiten in der spiralförmigen Anordnung vermieden werden und es zu keiner Clusterisierung der Gitterpunkte kommt.

Das Spiralgitter basiert auf dem Goldenen Winkel und kann aus einer beliebigen ungerade Anzahl von Punkten P aufgebaut sein. Um das Gitter auszuarbeiten, sei N eine beliebige natürliche Zahl. Die ganze Zahl i soll von $-N$ bis $+N$ reichen. Die Punktzahl ist $P = 2N + 1$. Die Kugelkoordinaten im Bogenmaß des iten Punktes sind:

$$lat_i = \arcsin\left(\frac{2i}{2N+1}\right)$$

$$lon_i = 2\pi i \Phi^{-1}$$

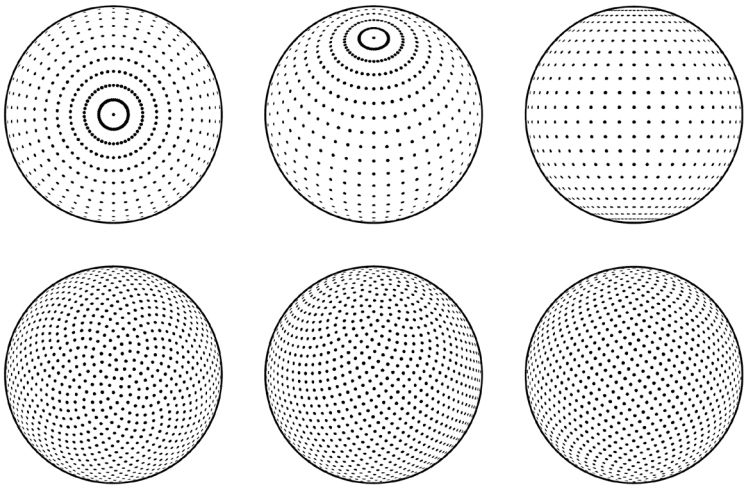


Abbildung 5.4: Ein Fibonacci-Gitter (unten) mit Blick auf dem Pol (links), um 45° gedreht (mittig) und mit Blick auf den Äquator (rechts) im Vergleich zu einem Breitengrad-Längengrad-Gitter (oben). Im Fibonacci-Gitter sind die Punkte wesentlich gleichmäßiger verteilt. (Quelle: [Go10] S.51)

Der Nachteil dieser Technik zur Generierung eines gleichmäßigen Kugelmeshes ist die Erfordernis einer komplexen Polygonisierung der Gitterpunkte. Die Dreiecke des Meshes lassen sich anders als bei den zuvor präsentierten Techniken nicht einfach erschließen, sondern müssen mit einem Polygonisierungverfahren ermittelt werden. Hier könnte bei beispielsweise die in Kapitel 3.3.7 beschriebene Delaunay Triangulierung zum Einsatz kommen.

5.6 Fazit

Die Generierung eines Kugelmeshes mit gleichmäßig verteilten Vertices ist kein triviales Problem. Die vorgestellten Methoden haben entweder den Nachteil unterschiedlicher Vertexdichte oder benötigen eine komplexe Triangulierung zur Erzeugung eines Kugelmeshes. Als Kompromiss wurde für den Planetengenerator die Ikosphäre als Grundkörper gewählt. Der zugrunde liegende Unterteilungsalgorithmus benötigt wenig Rechenleistung, das resultierende Mesh ist aus nahezu gleichseitigen Dreiecken aufgebaut und der Unterschied zwischen minimalen und maximalen Vertex-Abstand ist auch bei einer hohen Anzahl von Unterteilungen vergleichsweise gering.

6 Einfache Unterteilungsalgorithmik

Die Generierung von Kugelmeshes auf Basis eines Grundkörpers basiert auf *Unterteilungsalgorithmen* (engl. *Subdivision*). Die Flächen des Grundkörpers werden unterteilt, bis die gewünschte Granularität erreicht ist. Da das Zielmesh eine regelmäßige Kugel ist, kann ein einfacher Unterteilungsalgorithmus verwendet werden, der den Grundkörper nicht verändert. Anschließend werden die Vertices vom Zentrum der Kugel aus auf die Kugeloberfläche projiziert. Der Unterteilungsprozess soll möglichst wenig rechenintensiv sein. Zudem soll er keine überflüssigen Vertices erzeugen, da diese den Rechenaufwand bei der Darstellung des Objekts erhöhen und zudem zu doppelten Daten im weiteren Generierungsprozess des Planeten führen.

Da manche Programme wie die *Unity Engine* [S24] nur Meshes aus Dreieck-Polygonen akzeptiert, liegt auch bei der Wahl des Unterteilungsalgorithmus für den Planetengenerator der Fokus auf Polygonen mit drei Vertices und drei Kanten. Die einfache Unterteilung kann entweder rekursiv oder iterativ ausgeführt werden. Bei der rekursiven Variante wird das Ergebnis der vorherigen Unterteilungen wiederholend unterteilt, bis die gewünschte Rekursionstiefen erreicht ist (Abbildung 6.1). Die resultierende Vertexanzahl ist zwangsläufig eine Zweierpotenz. Auch entstehen an den Kanten zu anderen Polygonen zwangsläufig doppelte Vertices, die anschließend aufwendig identifiziert und entfernt werden müssen (Abbildung 6.2).

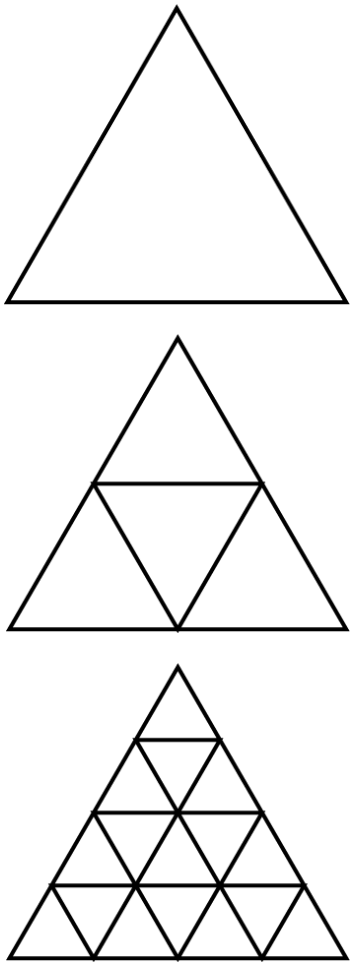


Abbildung 6.1: Rekursive Unterteilung eines Dreiecks. Die Ausgangsform (oben) wird Schrittweise unterteilt (mittig, unten) bis die gewünschte Rekursionstiefe erreicht wurde.

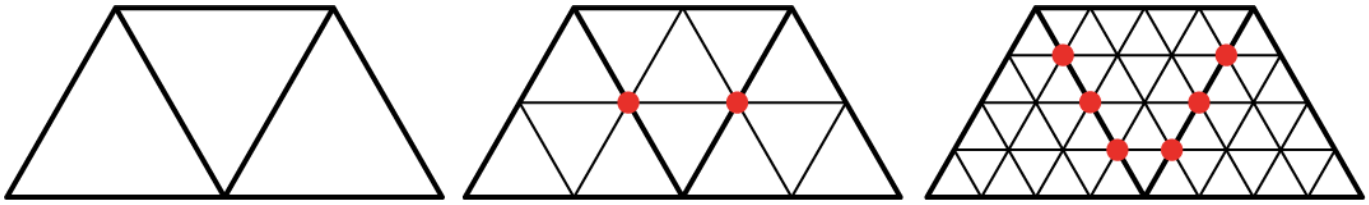


Abbildung 6.2: Beim Unterteilungsprozess entstehen doppelte Vertices an den Kanten zu den Nachbarpolygonen des Ausgangsdreiecks (rote Punkte). Diese müssen entweder aufwendig entfernt oder bei der Ausführung des Algorithmus vermieden werden.

Da die Ausgangspolygone ausschließlich Dreiecke sind, kann die Berechnung der Unterteilungen auch auf einfache Art und Weise iterativ erfolgen. Für zwei Kanten wird, abhängig von der Anzahl der Unterteilungen, ein Delta berechnet. Dieses wird zur Berechnung der Position der neuen Vertices verwendet. Anschließend werden die neuen Dreiecke bestimmt. Bei dreieckigen Polygonen wird im Vergleich zu Polygonen mit vier Ecken in jeder Zeile ein Vertex weniger erzeugt. Abbildung 6.3 zeigt die einzelnen Teilschritte des iterativen Unterteilungsalgorithmus. Anders als bei der rekursiven Berechnung kann eine beliebige Anzahl an gewünschten Unterteilungen gewählt werden. Auch müssen doppelte Vertices nicht erst nach Durchführung des Algorithmus entfernt werden, sondern können im Prozess vermieden werden. Dafür werden die Vertices an Kanten zu anderen Dreiecken gespeichert und bei der Unterteilung der benachbarten Dreiecke wiederverwendet.

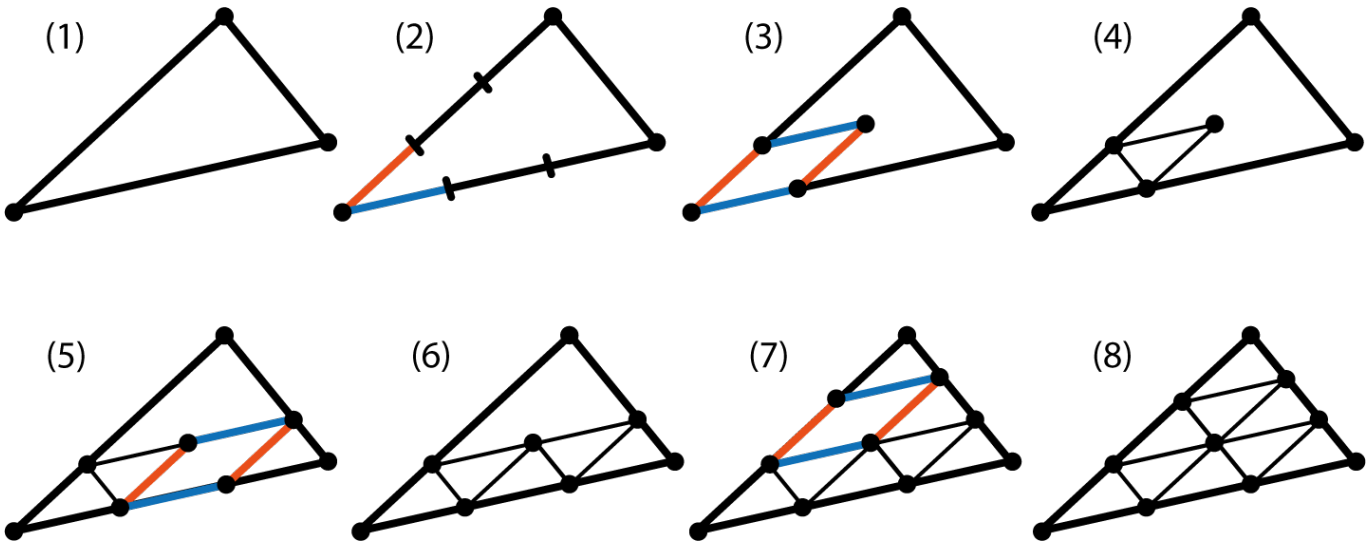


Abbildung 6.3: Bei der iterativen Unterteilung werden die neuen Vertices und Dreiecke Schrittweise erzeugt. Für zwei Kanten des Startpolygons (1) werden abhängig von der Anzahl der Unterteilungen Deltas berechnet (2). Diese werden zur Berechnung der neuen Vertexpunkte verwendet (3). Mit den Vertices werden anschließend neue Dreiecke gebildet (4). Schritte 3 und 4 werden solange durchgeführt bis keine neuen Vertices und Dreiecke generiert werden können (5-8).

7 Planetengenerator

Die vorgestellten Techniken zur prozeduralen Generierung, Terrainerzeugung und Kugelgenerierung bilden die theoretische Basis für die Implementierung des Planetengenerators. Für seine Umsetzung wurden passende Techniken und Algorithmen ausgewählt und miteinander verknüpft. Dadurch entstand eine Generierungspipeline, die in Kapitel 7.3 vorgestellt wird.

7.1 Eingesetzte Werkzeuge

Um den Planetengenerator optimal in den Entwicklungsprozess eines Spiels zu integrieren, wird er als Erweiterung einer Game Engine implementiert. Da der Autor über umfangreiche Kenntnisse bei der Entwicklung mittels *Unity Engine* [S24] verfügt, wird der Generator gezielt mit der und für die *Unity Engine* entwickelt. Dabei werden deren Datenstrukturen sowie Funktionen, wie beispielsweise das Job-System für Parallelisierung von Prozessen genutzt um den Generator optimal zu integrieren und zu optimieren. Die dem Generator zugrundeliegenden Techniken können allerdings auch mit einer anderen Engine wie *Unreal* [S25] oder *Godot* [S10] implementiert werden.

7.2 Noise System

Für den Planetengenerator wurde ein modulares System unterschiedlicher Noise Funktionen implementiert. Die Basis bilden Perlin und Simplex Noise. Wie von Perlin [Pe02] beschrieben, wird hierbei keine statische Permutationstabelle verwendet. Sie wird stattdessen auf Basis des Planeten-Seeds pseudozufällig errechnet. Dadurch ergibt sich mit jedem Seed eine unterschiedliche Noise Funktion. Eine Alternative zu dieser Vorgehensweise ist die Errechnung eines Offsets, der auf dem Planetenseed basiert und immer auf die Eingabevektoren des Noise summiert wird. Dieser Ansatz erfordert allerdings mehr Rechenaufwand als die einmalige Generierung der Permutationstabelle.

Die evaluierten Werte des Perlin oder Simplex Noise können anschließend mit den in Kapitel 3.5 genannten Operationen verändert werden. Die Operationen können in den meisten Fällen miteinander verschachtelt werden, wodurch ein modulares System entsteht. Mit diesem System können komplexe Terrainstrukturen wie Gebirgszüge und Canyons generiert werden. Anders als bei Noise üblich, liegt der Wertebereich der implementierten Noise Funktionen nicht zwischen -1 und 1 sondern zwischen 0 und 1. Dies erleichtert die Interpretation als Farb- oder Höhenwerte. Abbildung 7.1 zeigt mit diesem System erzeugte Schluchten. Die dazugehörige Noise Funktion mit *x* als dreidimensionaler oder zweidimensionaler Vektor lautet:

*f(x) = Clamp(Blend(Octave(Simplex(x * 0.5), 2) * 0.3, Ridge(Octave(Ridge(Billow(Simplex(x * 0.5))) , 3)*0.7)*0.5), 0, 0.5)*

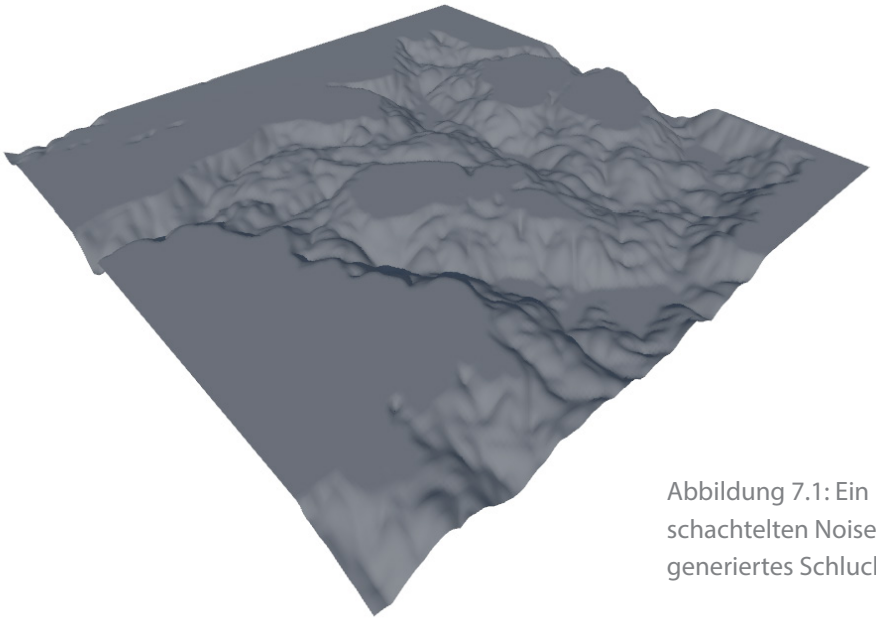


Abbildung 7.1: Ein mit verschachtelten Noise Funktionen generiertes Schluchten-Terrain.

7.2.1 Noise Planeten

Dieser Werkzeugkasten unterschiedlicher Noise Funktionen und Operationen ermöglicht bereits die Generierung von erdähnlichen Planeten mit Ozeanen, Kontinenten, Gebirgen und anderen Strukturen (Abbildung 7.2). Allerdings kann das Resultat mit dieser Technik nur begrenzt durch Parameter beeinflusst und die Planetenoberfläche schlecht in Zonen mit eigenen Biomen und Ressourcen unterteilt werden. Darum sind Planeten, die nur mit Noisefunktionen erzeugt wurden, nicht als Spielwelten für *Exploit Inc.* geeignet.

7.3 Generierungspipeline

Der Generierungsprozess des Planeten lässt sich in einzelne Teilabschnitte unterteilen, die sich wiederum als Pipeline zusammenfassen lassen (Abbildung 7.3). Die Elemente mit gestrichelter Linie waren als Teil der Pipeline angedacht, wurden aber aus zeitlichen Gründen nicht mehr für den Generator umgesetzt.

7.3.1 Parameter festlegen

Der Planetengenerator nimmt einige Eingabeparameter entgegen, welche den Generierungsprozess und dessen Resultat beeinflussen. Sie sind Teil der Anwenderschnittstelle und abstrahieren die technischen Details des Prozesses. Als Bezeichner für die Parameter wurden sprechende Namen gewählt, welche die auswirkungs- und nicht implementierungsbezogen sind.

Im Folgenden werden die einzelnen Eingabeparameter aufgeführt. Die kursiven Listenelemente sind dabei Parameter-Kategorien:

- **Seed (Integer).** Der Seed, eine beliebige Integer-Zahl, ist die Grundlage der pseudozufälligen Generierung. Der Generierungsprozess mit gleichen Parametern und gleichem Seed ist deterministisch.
- **Detailgrad (Float [0,1]).** Der Detailgrad bestimmt die Anzahl der Unterteilungen des Terrainmeshes und dadurch auch die Anzahl der Vertices und Evaluationspunkte.
- **Chaosfaktor (Float [0,1]).** Der Chaosfaktor bestimmt wie stark die Kanten der einzelnen Planetensegmente und die Terrainstrukturen verformt werden.
- **Chaosschichten (Integer [1,5])**. Beinflusst wie oft der Chaosfaktor angewendet wird.
- ***Ausmaße.*** Die Ausmaße-Parameter bestimmen die Größe des Planeten und die Dicke der einzelnen Schichten Grundgestein, Hülle und Atmosphäre.
  - **Radius des Grundgesteins r_g (Float [3,r_h]).** Der Radius des Grundgesteins bestimmt die Ausmaße des unveränderlichen Planetenkerns. Der Wert muss kleiner sein als der Hüllenradius.
  - **Radius der Hülle r_h (Float (r_g,r_a)).** Der Radius der Hülle des Planeten beeinflusst die Dicke der veränderlichen Außenhülle des Planeten. Der Wert muss kleiner sein als der Atmosphärenradius und größer als der Radius des Grundgesteins.
  - **Radius der Atmosphäre ra (Float (r_h,10]).** Der Radius der Atmosphäre ist die äußerste Ausdehnung

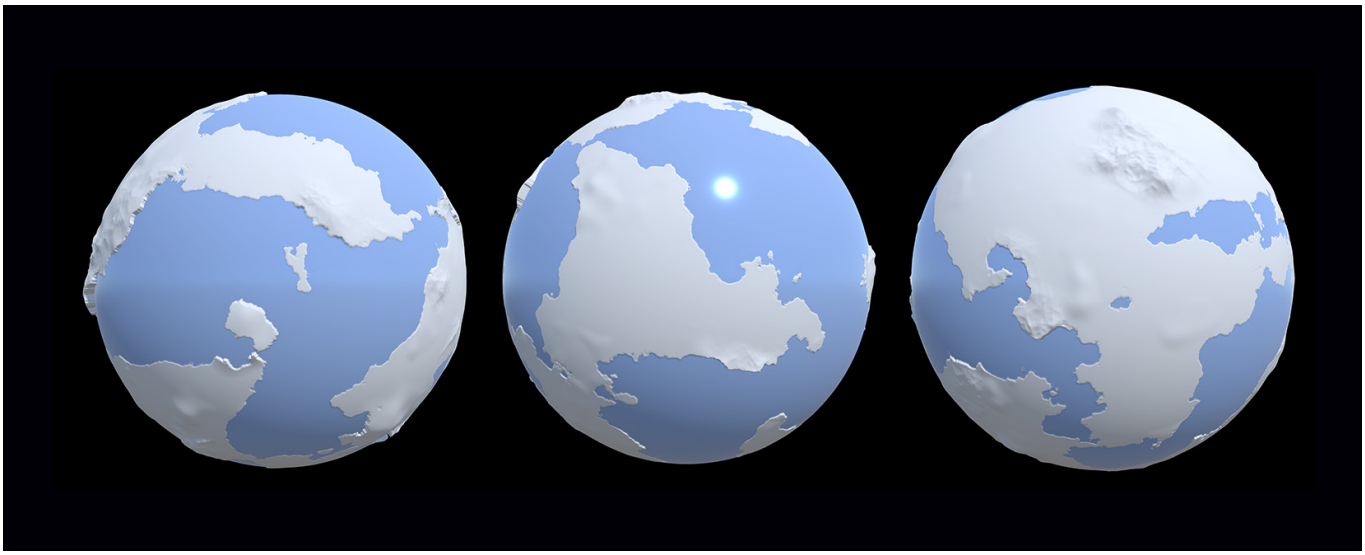


Abbildung 7.2: Drei mittels Noise generierte Planeten. Alleine auf Basis von Noise Funktionen können erdähnliche Planeten mit Kontinenten, Inseln, Gebirgen und anderen Strukturen generiert werden.

eines Planeten. Er ist immer größer als der Radius der Außenhülle.
    - **Relativer Meeresspiegel (Float [0,1]).** Der relative Meeresspiegel bestimmt die Höhe des Meeresspiegels innerhalb der Außenhülle des Planeten und dadurch auch die maximale Tiefe der Ozeane.
- ***Planetenachse.*** Die Planetenachse des Planeten bestimmt die Position seiner Pole und die Dauer eines Tages auf diesem Planeten.
    - **Neigungswinkel (Float [0, 90]).**
    - **Sekunden pro Umlauf (Float [1, 1000]).**
- ***Planetenregionen.*** Die Oberfläche des Planeten wird in Kontinente und Ozeane unterteilt, die aus mindestens einem Segment bestehen.
    - **Anzahl der Segmente *s* (Integer [20,1000]).** Die Oberfläche des Planeten ist in atomare Segmente unterteilt. Die Anzahl dieser Segmente ist nach unten hin durch ein technisches Minimum beschränkt. Die obere Grenze wurde aus Gründen der Performance gesetzt.
    - **Anzahl der Kontinente *k* (Integer [0,*s*–*o*]).** Ein Kontinent besteht aus mindestens einem Segment. Die Anzahl der Kontinente ist nach oben hin durch die Menge Segmente minus Anzahl der Ozeane beschränkt.
    - **Anzahl der Ozeane *o* (Integer [0,*s*–*k*]).** Ein Ozean besteht aus mindestens einem Segment. Die Anzahl der Ozeane ist nach oben hin durch die Menge Segmente minus Anzahl der Kontinente beschränkt.
    - **Regelmäßigkeit (Float [0,1]).** Der Regelmäßigkeit-Parameter bestimmt die Gleichmäßigkeit der Verteilung der einzelnen Segmente.
    - **Überblendung (Float [0,1]).** Der Überblendung-Parameter beeinflusst, wie stark Segmente und dadurch auch einzelne Biome ineinander überblenden.
- ***Kontinentalplatten.*** Die Einstellungen zu den Kontinentalplatten beeinflussen die Simulation der Plattentektonik. Gebirge und Schluchten entstehen dabei an den Kanten der Platten.
    - **Anzahl (Integer [1,*k*+*o*]).** Die Anzahl der Kontinentalplatten liegt zwischen 1 und der Summe der Anzahl Ozeane und der Anzahl Kontinente.
    - **Minimale Bewegungskraft (Float [0,1]).** Die minimale Bewegungskraft der Platten bestimmt die Ausprägung der Gebirge und Schluchten an ihren gemeinsamen Kanten.
- ***Terrainstrukturen.*** Das Terrain des Planeten wird durch Strukturen geformt. Der Planetengenerator unterstützt zum aktuellen Stand die Terrainstrukturen Gebirge und

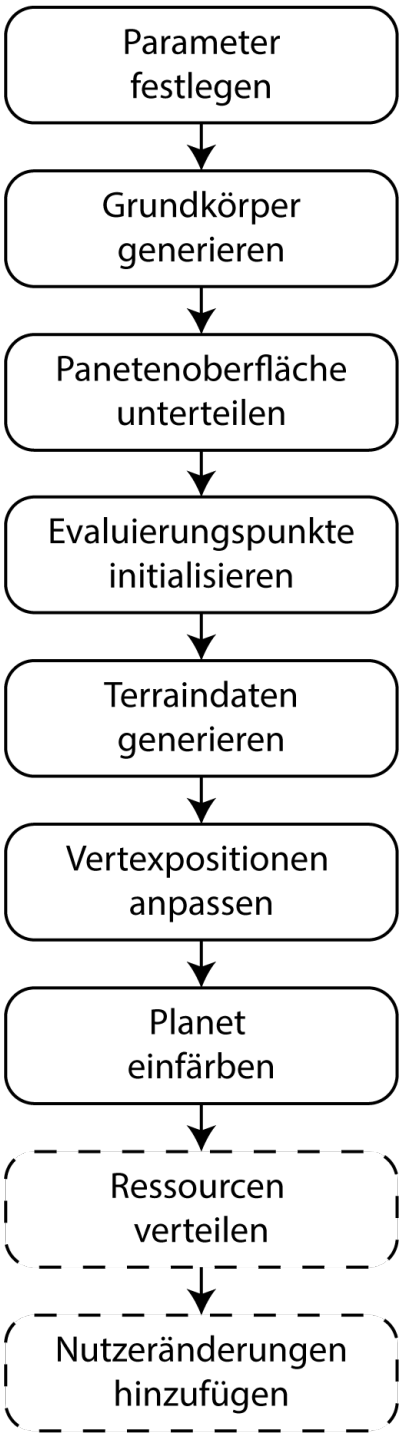


Abbildung 7.3: Die Generierungspipeline des Planetengenerators. Die gestrichelten Elemente symbolisieren einen Generierungsschritt, der angedacht war, aus Zeitgründen allerdings nicht implementiert wurde.

Schluchten.

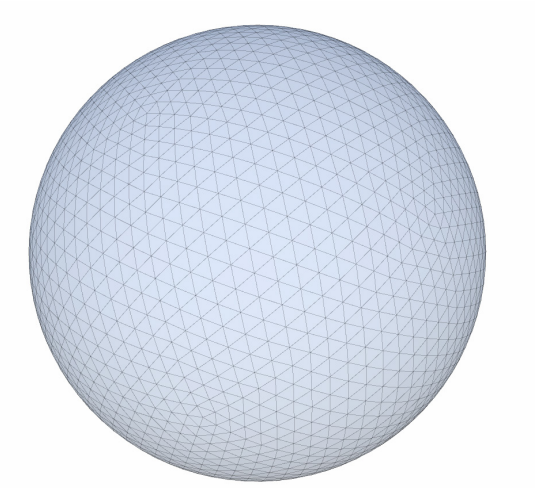
- ***Gebirge.*** Gebirge entstehen an Kanten von Kontinentalplatten, die sich aufeinander zubewegen. Sie können über die folgenden Parameter beeinflusst werden.
  - **Minimale Breite (Float [0,1]).**
  - **Maximale Breite (Float [0,1]).**
  - **Minimale Höhe (Float [0,1]).**
  - **Maximale Höhe (Float [0,1]).**
  - **Überblendung (Float [0,1]).**
- ***Schluchten.*** Schluchten entstehen an Kanten von Kontinentalplatten, die sich voneinander wegbewegen. Sie können über die folgenden Parameter beeinflusst werden.
  - **Minimale Breite (Float [0,1]).**
  - **Maximale Breite (Float [0,1]).**
  - **Minimale Tiefe (Float [0,1]).**
  - **Maximale Tiefe (Float [0,1]).**
  - **Überblendung (Float [0,1]).**
- ***Temperaturspektrum.*** Das Temperaturspektrum eines Planeten beeinflusst die Verteilung und die Art der Biome, die auf seine Oberfläche vorkommen können. Kalte Planeten haben keine Sandwüste und heiße Planeten keine vereisten Pole. Am Äquator herrscht die maximale und an den Polen die minimale Temperatur. Dazwischen wird der Wert überblendet.
  - **Minimal min (Float [-273.2, max]).** Die minimale Temperatur eines Planeten liegt zwischen dem absoluten Nullpunkt und der maximalen Temperatur.
  - **Maximal max (Float [min, ∞]).** Die maximale Temperatur eines Planeten liegt zwischen seiner minimalen Temperatur und dem maximalen Float-Wert.

7.3.2 Grundkörper generieren

Im ersten Schritt erzeugt der Generator den Grundkörper des Planeten in Form einer Kugel. Dazu wird wie in Kapitel 5.4 beschrieben ein Ikoasaeder-Mesh generiert und die Seiten des Körpers voneinander getrennt, sodass 20 separate Flächen entstehen. Jede dieser Flächen wird auf Basis des „Detailgrad"-Parameters iterativ unterteilt (siehe Kapitel 6). Anschließend werden die einzelnen Vertices, vom Zentrum des Körpers aus, auf die Kugeloberfläche projiziert. Wie in Abbildung 7.4 zu sehen ist, entsteht dadurch eine fein unterteilte Kugel.

7.3.3 Planetenoberfläche unterteilen

Im nächsten Schritt wird die Oberfläche des Kugelmeshes in einzelne Segmente unterteilt. Dazu werden mit Sample Elimination (Kapitel 3.3.5) basierend auf dem „Anzahl der Segmente"-Parameter eine bestimmte Menge Sites ermittelt. Diese Sites werden verwendet, um anschließend ein Voronoi-Diagramm (Kapitel 3.3.7) zu generieren (Abbildung 7.5). Die Linien des Diagramms bilden die Kanten der einzelnen Segmente. Auf Basis der "Chaosfaktor"-Einstellung werden die geraden Linien des Voronoi-Diagramms mittels Domain Warping (Kapitel 3.5.5) verformt. Dadurch erhalten die Segmente eine natürliche Form. Jedem Segment wird ein Kontinent bzw. ein Ozean sowie ein Biom zugeordnet. Die Kontinente und Ozeane wiederum sind Teil einer Kontinentalplatte. Die Wahl der Biome wird von der lokal herrschenden Temperatur und somit von der Position eines Segments in Abhängigkeit zur Planetenachse bestimmt. So treten Wüsten bei einem erdähnlichen Planeten in der Nähe des Äquators und Eisregionen an den beiden Polen auf.



bbildung 7.4: Ein Kugelmesh, welches auf Basis eines Ikosaeders generiert wurde.

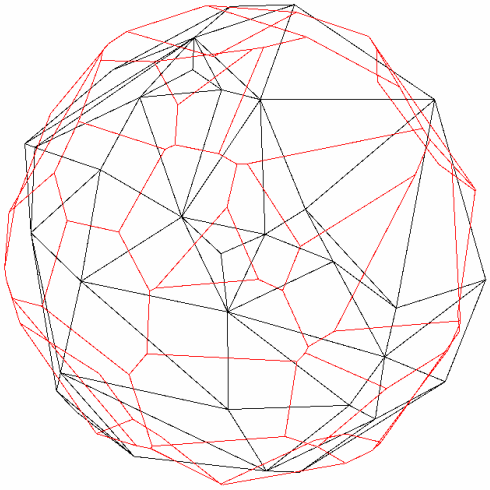


Abbildung 7.5: Die Delaunay-Triangulierung (schwarz) von 50 Sites auf der Oberfläche einer Kugel mit dem dazugehörigem Voronoi-Diagramm (rot).

7.3.4 Evaluierungspunkte initialisieren

Im dritten Schritt wird pro Vertexpunkt des Planeten ein Datensatz erzeugt, der das Terrain des Planeten an dieser Stelle abbildet. Der Datensatz enthält Informationen zu den dort vorzufindenden Biomen, den einzelnen Terrainschichten und dem Segment, in dem sich der Vertexpunkt befindet. An einem Evaluierungspunkt können mehrere Biome auftreten, da die Regionen fließend ineinander übergehen und an den Segment-Kanten dadurch eine Übergangszone entsteht. Die Terrainschichten werden im nächsten Schritt initialisiert.

7.3.5 Terraindaten generieren

Die Außenhülle der Planeten ist aus einzelnen Schichten aufgebaut (siehe Kapitel 4.2.2). Diese werden im vierten Schritt der Pipeline für jeden Evaluierungspunkt berechnet. Der Generator beschränkt sich in der aktuellen Version auf fünf Schichten: Luft, Vegetation, Flüssigkeit, Boden und Gestein. Abbildung 7.6 zeigt die einzelnen Schichten eines generierten Planeten.

Die Daten der Terrainschichten eines Evaluierungspunktes bestehen aus einem relativen Höhenwert sowie einer Liste mit Materialinformationen. Die Art dieser Materialien ist abhängig von den Biomen, die an dem Evaluierungspunkt vorkommen. Bei derzeitigem Stand kann jedem Biom maximal ein Material pro Schicht zugewiesen werden. Das „Ebene"-Biom beispielsweise ist zusammengesetzt aus den Materialien Granit, Erde, Wasser, Gras und Luft. Die Materialien bestimmen insbesondere den Farbwert der Planetenoberfläche, welcher im sechsten Schritt gesetzt wird.

Die Terrainstrukturen wie Gebirge und Schluchten bestimmen die Dicke der Gesteinsschicht. Die Position, Ausmaße und Überblendung dieser Strukturen wird, wie in Kapitel 4.1 beschrieben, durch einzelne Primitive repräsentiert, die als „Shaping-Layer" gebündelt werden. So werden beispielsweise einzelne Gebirgszüge als Gebirge-Layer zusammengefasst. Jedem Layer wird eine Noisefunktion zugewiesen, welche zuvor mit dem in Kapitel 7.2 beschriebenen Verfahren zusammengesetzt wurde. Das Primitiv in Form eines Kreises, einer Ellipse oder eines konvexen Polygons bestimmt, in welchem Bereich die Noisefunktion evaluiert werden soll. Befindet sich ein Evaluierungspunkt außerhalb des Primitives, so fließt es nicht in die Berechnung ein. Die Evaluationsergebnisse der einzelnen Shaping-Layer werden am Ende überblendet und ergeben so den Höhenwert der Gesteinsschicht am evaluierten Punkt. Als Verrechnungsoptionen können Überblendung, Addition, Mindestwert oder Maximalwert gewählt werden.

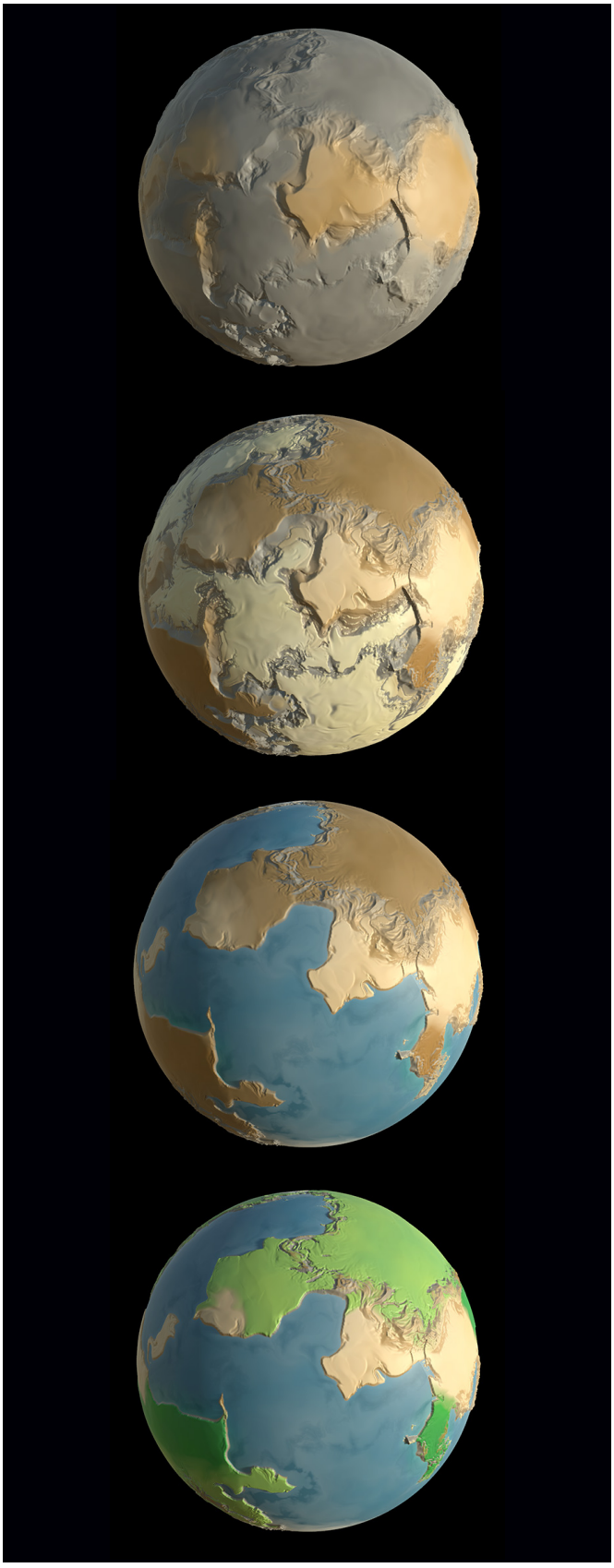


Abbildung 7.6: Das Terrain der Planeten ist aus verschiedenen Schichten aufgebaut (von oben nach unten: Gestein, Boden, Füssigkeiten, Vegetation).

Die Position und Form der Gebirge- und Schluchten-Primitive werden über eine vereinfachte Version der in Kapitel 4.3.1 beschriebenen Plattentektonik-Simulation bestimmt. Bewegen sich zwei Kontinentalplatten aufeinander zu, so entstehen an der Kante der beiden Platten Gebirge; bewegen sie sich voneinander weg, so bilden sich Schluchten. Die Ausprägung der Gebirgen bzw. Schluchten ist abhängig von den gesetzten Eingangsparametern und der Bewegungskraft der Planeten.

7.3.6 Vertexpositionen anpassen

Auf Basis der Terraindaten kann die Position der einzelnen Vertices bestimmt werden. Dafür wird die Luftschicht ignoriert und die Höhenwerte der anderen Schichten aufaddiert. Die neue Position liegt zwischen dem Radius des Planetenkerns und dem Radius seiner Hülle.

7.3.7 Planet einfärben

Auch bei der Bestimmung der Farbwerte der Planetenoberfläche kommen die Terraindaten zum Einsatz. Die Materialien der ersten Schicht unter der Luftschicht definieren die Vertexfarbe am entsprechenden Oberflächenpunkt (Abbildung 7.7). Die Farbwerte der einzelnen Materialien der Schicht werden in Abhängigkeit ihres prozentualen Anteils aufaddiert. Ist die oberste Schicht die Flüssigkeitsschicht, so wird zudem die Dicke der Schicht mit betrachtet, sodass tiefes Wasser dunkler dargestellt wird als flaches Gewässer. Desweiteren scheint bei einer dünnen Vegetations- bzw. Bodenschicht das Material der darunterliegenden Schicht durch.

7.3.8 Ressourcen verteilen

Die Verteilung von Ressourcen war als Schritt im Generierungsprozess angedacht, konnte aber aus zeitlichen Gründen nicht mehr implementiert werden. Geplant war die Verteilung von Pflanzen, Tieren und Rohstoffen auf der Planetenoberfläche mittels Sample Elimination. Da Ressourcen nur in festgelegten Biomen vorkommen, bildet sich die Sampling-Domäne aus den Segmenten mit entsprechenden Biom. Die Verteilungsdichte wird von einer Noisefunktion bestimmt. Erze hingegen wären als zusätzliche Terrainschicht in die Gesteinsschicht integriert worden. Sie bleiben der BetrachterIn verborgen, solange das Terrain nicht verändert oder die Terrainstruktur nicht analysiert wird.

7.3.9 Nutzeränderungen anwenden

Auch potentielle benutzerdefinierte Änderungen werden im Generierungsprozess bisher nicht berücksichtigt. Dennoch können die Terraindaten eines Planeten auch nach dem Generierungprozess leicht angepasst werden, was grundsätzlich die Übernahme von benutzerdefinierten Änderungen ermöglicht. Eine Anpassung der Terraindaten macht anschließend eine Neuberechnung der Vertexpositionen und Farbwerte erforderlich.

7.4 Planeten

Das Resultat des Generierungsprozesses ist ein GameObject der *Unity Engine* [S24] mit einer Planet-Komponente. Das Skript hält eine Referenz auf zwanzig *PlanetFace-*

Abbildung 7.7: Die Oberflächenfärbung eines Planeten wird durch die Materialien der obersten nicht-gasförmigen Terrainschicht bestimmt.

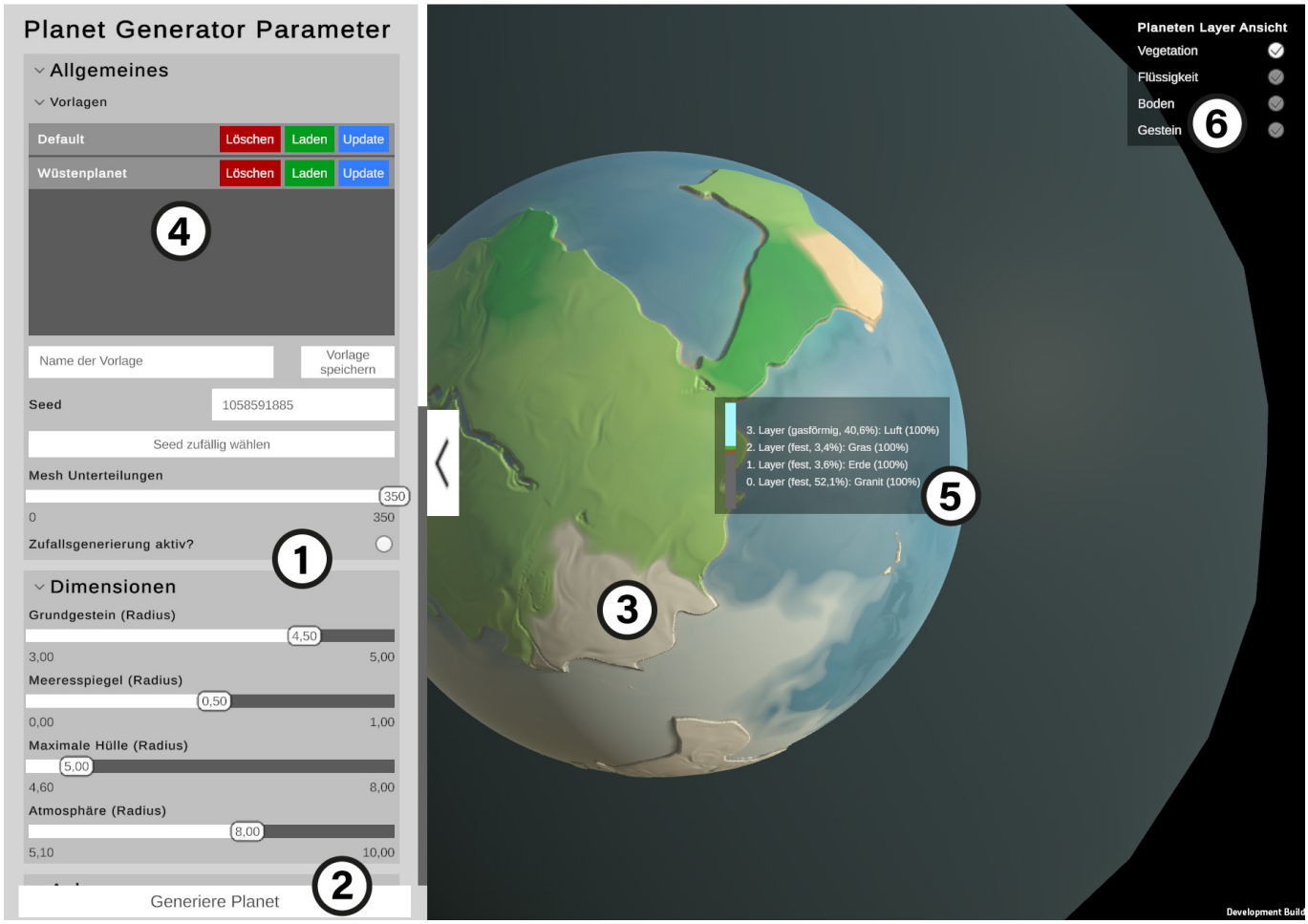


Abbildung 7.8: Die Demoanwendung zum Testen des Planetengenerators. Zu sehen sind folgende Elemente: (1) Parametermenü, (2) "Generiere Planet"-Button, (3) Planeten-Ansicht, (4) Vorlagenübersicht, (5) Übersicht der Materialschichten, (6) Ansichtmenü.

Komponenten sowie auf den *PlanetData*-Datensatz. Jede *PlanetFace*-Komponente repräsentiert mit einer Referenz auf sein Mesh und seine Evaluierungspunkte einen Teil der Planetenoberfläche. Der *PlanetData*-Datensatz enthält zudem die Informationen, welche dem Planetengenerator als Eingabeparameter übergeben wurden. Dazu zählen die Ausmaße des Planeten sowie die Biome und Materialien, die auf der Planetenoberfläche bzw. in den Terrainschichten auftreten können.

7.5 Demoprogramm

Im Zuge dieser Arbeit wurde mithilfe der *Unity Engine* [S24] eine Demoanwendung entwickelt, mit welcher der Planetengenerator getestet werden kann (Abbildung 7.8). Alle Eingangsparameter des Generators können über das UI angepasst werden (1). Dabei wurde darauf geachtet, dass die NutzerInnen nur valide Werte wählen können. So überschreitet beispielweise der Radius der Planetenkerns niemals den Radius der Atmosphäre.

Der Generierungsprozess wird über den „Generiere Planet"-Button gestartet (2). Die Parameterwerte werden von der Anwendung gesammelt und an den Generator übergeben. Der erzeugte Planet wird daraufhin in der Mitte der Bildschirms angezeigt (3). Mit einer einfachen Kamerasteuerung kann die NutzerIn um den Planeten rotieren und an die Planetenoberfläche heranzoomen. Der Generierungsprozess kann jederzeit mit angepassten Parametern erneut gestartet werden.

Wurde ein Set aus Parametern gefunden, welches interessante Planeten erzeugt, so kann dieses in Kombination mit einem benutzerdefinierten Namen als Vorlage abgespeichert werden. Die neue Vorlage erscheint anschließend in der Übersicht (4). Jede gespeicherte Vorlage kann geladen, mit den aktuellen Parametern überschrieben oder gelöscht werden.

Eine Übersicht der Schichten der Planetenaußenhülle kann an jedem Punkt des Planeten mit gedrückter rechter Maustaste aufgerufen werden (5). Mit dem Menü in der oberen rechten Ecke können die einzelnen Schichten aus- bzw. eingeblendet werden (6).

Das Demoprogramm arbeitet mit einer vordefinierten Anzahl von Biomen und Materialien, die der Erde nachempfunden sind. So können je nach gewähltem Temperaturspektrum Wüsten, Tropen, Tundra, Eisgebiete und gemäßigte Zonen auftreten. Auch die Art der Noisefunktionen zur Erzeugung von Gebirgen, Schluchten oder Hügeln sind in der Demoanwendung festgelegt.

Neben der *Unity Engine* wurde für die Umsetzung des Demoprogramms das Dependency Injection Framework *Zenject* [S27] genutzt, um die Abhängigkeiten der einzelnen Skripte aufzulösen. Außerdem wurde für die Serialisierung und Deserialisierung der Vorlagen als Json-Objekt das Framework *Json.NET* [S11] verwendet.

7.6 Ressourcenbedarf des Generierungsprozesses

Mithilfe des Demoprogramms und den Profiling-Werkzeugen der *Unity Engine* [S24] wurden der benötigte Arbeitsspeicher sowie die Dauer des Generierungsprozesses untersucht. Dafür wurden drei verschiedene Ausgangssituationen mit jeweils unterschiedlichem Detailgrad des Planeten und einer unterschiedlichen Anzahl von Oberflächensegmenten gewählt. Bei jedem Testdurchlauf wurden zehn Kontinente und zehn Ozeane mit sechs Kontinentalplatten erzeugt. Die Ergebnisse des Tests werden in Tabelle 7.1 aufgelistet.

Die Anzahl der Segmente wirkt sich weitestgehend konstant auf die Leistungsanforderungen des Generierungsprozesses aus. Der steigende Detailgrad hingegen erhöht sowohl die Dauer des Prozesses als auch den verwendeten Arbeitsspeicher stark. Dies liegt insbesondere an der hohen Anzahl Vertices, von denen jeder ein Evaluierungspunkt auf dem Planeten mit eigenem Datensatz ist.

Für den Test wurde ein Windows-Heimcomputer mit folgenden Spezifikationen verwendet:

- Betriebssystem: Windows 10 Home
- Prozessor: Intel Core i7 -6700 CPU mit 3,40 GHz und vier Kernen
- Arbeitsspeicher: 16 GB RAM
- Grafikkarte: NVIDIA GeForce GTX 1060 6GB

Mit diesen Werten befindet sich der Rechner im mittleren bis oberen Teil des möglichen Leistungsspektrums von spieletauglichen Heimrechnern.

7.7 Quelldateien

Die Quelldateien des Demoprogramms und des Planetengenerators können unter folgendem Github-Link aufgerufen werden:

https://github.com/BlackLambert/masterthesis

Die dazugehörige Version der *Unity Engine* [S24] ist 2021.1.9f.

Detailgrad	Segmente	resultierende Vertices	RAM in GB	Prozessdauer in Sekunden
10%	100	13.320	ca. 0,25	< 0,1
10%	800	13.320	ca. 0,32	< 0,5
50%	100	308.000	ca. 0,50	< 4
50%	800	308.000	ca. 0,54	< 5
100%	100	1.242.560	ca. 2,07	< 15
100%	800	1.242.560	ca. 2,13	< 16

Tabelle 7.1: Der benötigte Arbeitsspeicher sowie die Dauer des Generierungsprozesses bei unterschiedlichen Ausgangssituationen.

8 Fazit

Für die Generierung von Planeten als Spielwelten für ein interplanetares Aufbaustrategiespiel steht eine Vielzahl prozeduraler Techniken und Algorithmen zur Verfügung. Das Terrain der Planeten kann durch Noisefunktionen und Simulationen geformt werden und es können Ressourcen mit Verteilungsalgorithmik auf der Planetenoberfläche verteilt werden. Auf Basis dieser prozeduralen Techniken wurde ein Planetengenerator entwickelt, der Spielwelten für das Spiel *Exploit Inc.* erzeugen soll. In diesem Kapitel wird untersucht, ob der Generator und seine Resultate die in Kapitel 2.2.2 definierten Anforderungen erfüllen, wo Verbesserungspotential vorhanden ist und welche Anforderungen nicht erfüllt wurden.

Diese Anforderungen wurden erfüllt:

- Parameter mit sprechenden Namen abstrahieren die technischen Details. Dadurch können ihn auch fach- bzw. themenfremde EntwicklerInnen sinnvoll nutzen. Einstellungen zu den Biomen, Materialien und Noisefunktionen erfolgen über sogenannte *ScriptableObjects*, die auch von themen- oder fachfremden EntwicklerInnen leicht angepasst werden können.
- Wie die Demoanwendung deutlich macht, erfolgt der Generierungsprozess der Planeten wie gefordert zur Laufzeit der Anwendung und nicht zur Entwicklungszeit.
- Der Generierungsprozess der Planeten ist deterministisch. Gleiche Eingabeparameter inklusive Seed resultieren in den gleichen Planeten. Zur Optimierung könnten die Parameterwerte innerhalb des Seeds kodiert werden, wodurch kein Parameter-Tupel zur Erzeugung eines bestimmten Planeten nötig wäre, sondern ausschließlich der Seed.
- Die Generierung von erdähnlichen Planeten mit unterschiedlichen Biomen, Ozeanen und Kontinenten ist möglich.
- Wie Abbildung 7.9 zeigt, sind die generierten Planeten im Vergleich zueinander visuell divers. Die Varianz entsteht durch die verschiedenen Eingabeparameter. So können mit entsprechendem Temperaturspektrum Eis-, Wüsten- sowie erdähnliche Planeten erzeugt werden und durch eine hohe Anzahl Kontinentalplatten können Planeten mit vielen Gebirgen und Schluchten entstehen. Auch reine Wasserplaneten sind möglich.
- Die generierten Planeten sind wie gefordert nicht maßstabsgetreu und haben im Vergleich zum Planetendurchmesser ein deutlich größeres Terrain als ihr reales Vorbild. Dadurch behält die SpielerIn auch bei einer Totalansicht des Planeten die Übersicht.
- Die Oberfläche des Planeten wird wie gefordert in unterschiedliche Segmente mit eigenen Biomen unterteilt.

Diese Anforderungen wurden teilweise erfüllt:

- Da der Generator mit und für die *Unity Engine* [S24] entwickelt wurde, ist er sehr leicht in den Entwicklungsprozess eines mit *Unity* entwickelten Spiels zu integrieren. Um allerdings in anderen Engines verwendet werden zu können, muss der Generator zuvor generalisiert werden und von der Verwendung *Unity*-eigener Systeme wie dem Job-System abgesehen werden.
- Der Generierungsprozess der Planeten wird durch sprechende Parameter gesteuert. Allerdings führt nicht jedes Tupel aus Eingabewerten zu einem Planeten, der als Spielwelt für *Exploit Inc.* verwendet werden kann. Daher wurden die Option implementiert, Vorlagen und so interessante Werte-Tupel zu speichern.
- Wie in Kapitel 7.6 deutlich wird, kann der Generierungsprozess, sowohl die Speicherbelegung als auch die beanspruchte Rechenleistung betreffend, auf einem durchschnittlichen Heimrechner ausgeführt werden. Doch steigen die beanspruchten Ressourcen mit zunehmendem Detailgrad des zu generierenden Planeten stark an.
- Auch die Anforderung, die Dauer den Generierungsprozesses in einem Rahmen zu halten, welcher für SpielerInnen akzeptabel ist, wurde nur zum Teil erfüllt. Die Generierung eines Planeten mit niedrigem Detailgrad und einer geringen Anzahl Segmente erfolgt innerhalb weniger Sekunden. Die Erzeugung einen Planeten auf höchster Detailstufe hingegen dauert bis zu einer halben Minute. Für einen Prozess außerhalb der initialen Ladezeiten eines Spiels dauert das zu lange.
- Die Datenstruktur der generierten Planeten ermöglicht eine anschließende Manipulation durch einen Prozess im Spiel oder Nutzerinteraktion. So kann Terrain verändert werden, indem die Höhenwerte der Terrainschichten erhöht oder reduziert werden. Da der Generator

allerdings bei derzeitigem Stand keine Ressourcen verteilt, können diese nicht abgebaut werden. So wurde die Anforderung, Planeten mit veränderbarem Terrain und Ressourcen zu erzeugen, nur zum Teil erfüllt.
- Die generierten Planeten sind durch unterschiedliche Biome, Klimazonen sowie Terrainstrukturen wie Gebirgszüge und Schluchten zum Teil in sich divers. Dennoch entsteht auf der Planetenoberfläche eine gewisse Regelmäßigkeit. Dem könnte durch zusätzliche Simulationen wie der Berechnung von Flussläufen oder durch zusätzliche Terrainstrukturen wie Krater oder Plataugebirge entgegengewirkt werden.

Diese Anforderungen wurden nicht erfüllt:

- Die Anforderung, die Ressourcen der Planeten spielerisch interessant zu verteilen wurde nicht erfüllt. Die Verteilung von Ressourcen wurde aus zeitlichen Gründen nicht in die Generierungspipeline integriert. Dennoch schafft diese Arbeit die theoretische Grundlage für eine spielerisch interessante und natürlich wirkende Verteilung.
- Durch Spielprozesse oder Nutzerinteraktion entstandene Änderungen am Planeten werden vom Generator bei derzeitigem Stand nicht wiederhergestellt. Ein Spiel, welches den Generator verwendet, müsste diese Änderungen nachträglich anwenden.

Insgesamt erfüllen der Generator, der Generierungsprozess und die erzeugten Planeten die meisten Anforderungen mindestens zum Teil. Es können aufgrund unterschiedlicher Biome, Kontinente und Ozeane visuell diverse, erdähnliche Planeten generiert werden, Parameter ermöglichen eine leichte Anpassung der generierten Planeten und der Generator kann leicht in den Entwicklungsprozess eines Spiels integriert werden. Jedoch sind die Terrainstrukturen auf Gebirge, Hügel und Schluchten begrenzt, wodurch die Vielfältigkeit des Terrains eingeschränkt ist. Zudem ist die Anzahl der Terrainschichten festgelegt, sodass die Generierung unterschiedlicher Gesteinsschichten zum aktuellen Zeitpunkt nicht möglich ist. Diese müsste nachträglich hinzugefügt werden. Auch konnte aus zeitlichen Gründen die Ressourcenverteilung nicht in den Generierungsprozess integriert werden.

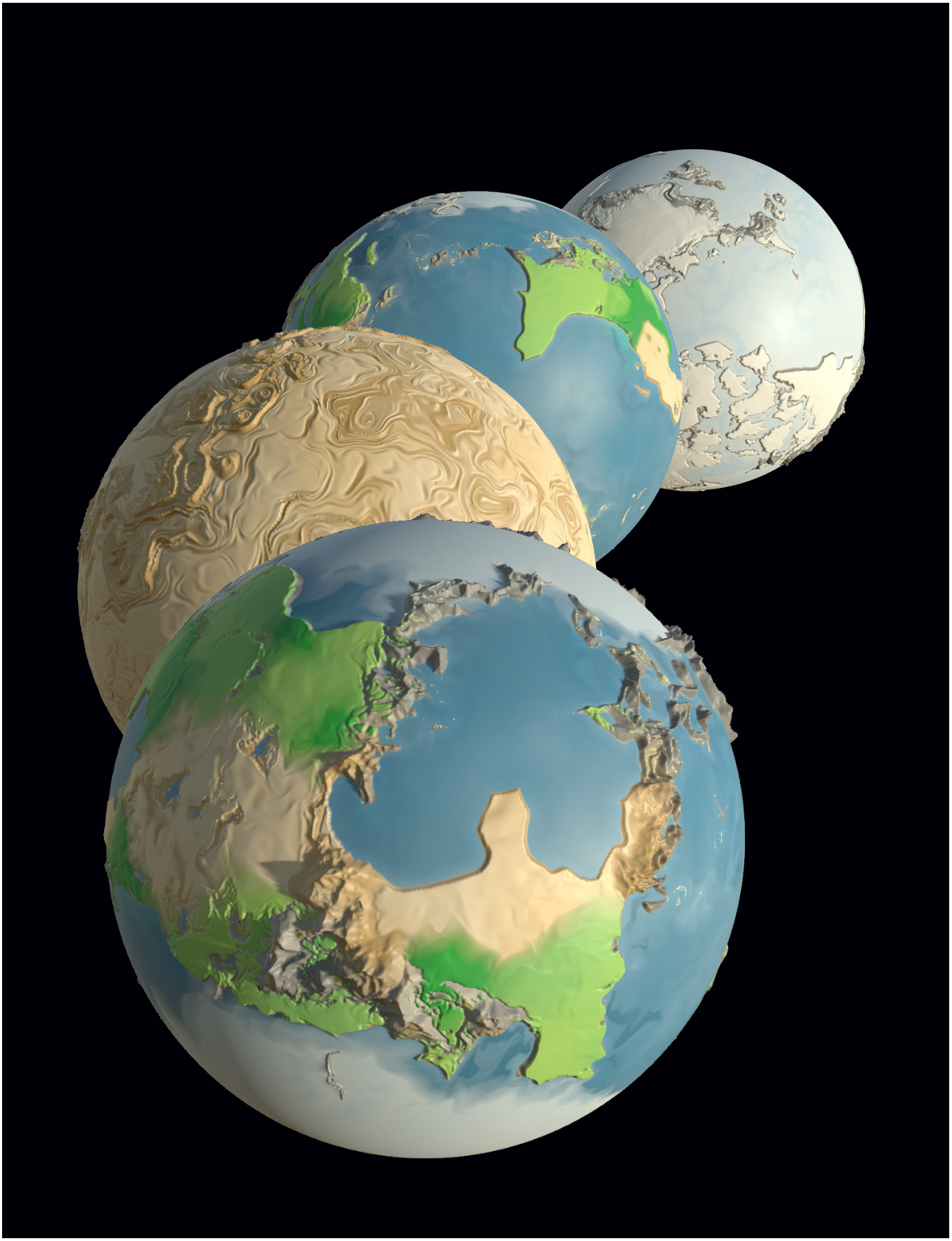


Abbildung 7.9: Vier mit dem Generator erzeugte Planeten. Es kann eine große Vielfalt unterschiedlicher Planeten generiert werden.

9 Ausblick

Der Planetengenerator bildet eine Basis zur Weiterentwicklung. Die folgenden Punkte können in zukünftigen Arbeiten angegangen werden:

- **Performanceoptimierung.** Der Generierungsprozess eines Planeten mit einem stark unterteilten Mesh und einer hohen Anzahl Evaluierungspunkte dauert bis zu einer halben Minute auf einem leistungsstarken Heimrechner. Im Rahmen eines Videospiels wie *Exploit Inc*. ist dieser Wert zu hoch. Um den Generierungsprozess zu beschleunigen, sind einige Ansätze möglich. Zum Einen könnte die Evaluierung von Noisefunktionen und andere parallelisierbare Algorithmen in Form von Compute Shadern auf die GPU verlagert werden. Der Grafikprozessor eines Computers ist für mathematische Berechnungen im Bereich der Computergrafik optimiert. Daher ist hier ein Performancegewinn zu erwarten. Zum Anderen wurde beim Profiling des Generierungsprozesses die Berechnung der einzelen Terrainschichten des Planeten als besonders rechenintensiv identifiziert. Hier könnte an einer alternativen oder optimierten Lösung gearbeitet werden.
- **Ressourcenverteilung.** Aus zeitlichen Gründen konnte die Verteilung von Ressourcen und Objekten wie Erzen, Öl, Flora und Fauna nicht mehr in den Generierungsprozess integriert werden und ist dadurch ein Aspekt für zukünftige Arbeiten.
- **Größere Vielfalt.** Durch die seedbasierte, pseudozufällige Generierung der Planeten sowie durch unterschiedliche Biome und Terrainstrukturen wie Gebirge und Schluchten können visuell diverse Planeten erzeugt werden. Dennoch gibt es auch hier Potential zur Verbesserung. So ist es denkbar, das Terrain beispielsweise um Flüsse, Seen, Plattaugebirge, Vulkane, Krater oder Klippen zu erweitern. Ferner kann eine größere Vielfalt durch die Verwendung von Texturen anstelle einer einfachen Einfärbung des Terrains basierend auf der Vertexfarbe erreicht werden.
- **Verbesserte Terrainqualität.** Die visuelle Qualität des Terrains kann beispielsweise mit der Anwendung von Erosionssimulation verbessert werden. Auch ist derzeit die maximale Anzahl der Vertices des Terrainmeshes durch die Performance des Generierungsprozesses und *Unity*-interne Begrenzung von 65.535 Vertices pro Mesh [Unity1] beschränkt. Dadurch wirkt die Planetenoberfläche aus der Nähe sehr verschwommen. Dem kann mit einer Unterteilung des Terrains in Chunks sowie der Einführung eines Level-of-Detail-Systems (LOD) entgegengewirkt werden.
- **Volumentisches Datenmodell.** Das Terrainmesh der generierten Planeten wird derzeit auf Basis von Höhendaten bestimmt. Jeder Vertex hat, ausgehend vom Zentrum des Planeten, eine bestimmte Höhe. Dadurch können volumetrische Strukturen wie Höhlen, Gesteinsbögen oder Überhänge nicht dargestellt werden. Dies schränkt wiederum die spielerische und visuelle Vielfalt der Planeten ein. Insbesondere in Kombination mit Terraforming durch Ressourcenabbau ist das Fehlen von volumetrischen Strukturen von Nachteil. Die Entwicklung eines solchen volumetrischen Datenmodells könnte das Thema einer zukünftigen Arbeit sein.
- **Generalisierung.** Der Planetengenerator wurde explizit für die *Unity Engine* [S24] entwickelt und verwendet deren Datenstrukturen und Systeme wie das Job-System zur Parallelisierung von Prozessen. Dennoch können die verwendeten Techniken und Algorithmen auch für andere Engines wie *Unreal* [S25] oder *Godot* [S10] implementiert werden. Diese Generalisierung des Generators ist eine weiteres Thema für zukünftige Arbeiten.

Abbildungsverzeichnis

Tabellenverzeichnis

Schriftquellen

[Ah01] Ahmad, S. (2001): »CES 2001: Microsoft Unveils the Xbox Console«. URL: https://www.gamespot.com/articles/ces-2001-microsoft-unveils-the-xbox-console/1100-2671751/ [Stand: 10. September 2021].

[BKR*19] Becher M. et al. (2019): »Feature-Based Volumetric Terrain Generation and Decoration«. In: *IEEE Transactions on Visualization and Computer Graphics* 25, H. 2, S. 1283–1296.

[Br07] Bridson, R. (2007): »Fast Poisson Disk Sampling in Arbitrary Dimensions«. In: Alexa, M. / Finkelstein, A. (Hg.): *SIGGRAPH ,07: ACM SIGGRAPH 2007 sketches*. New York: Association for Computing Machinery, S. 22.

[Brockhaus1] Brockhaus Enzyklopädie Online: »Computersimulation«. URL: https://brockhaus-de.ezproxy.th-brandenburg.de/ecs/permalink/6CE368978B-65B0AF300F2F5B61269598.pdf [Stand: 26. September 2021].

[BW97] Baraff, D. / Witkin, A. (1997): *Physically Based Modeling: Principles and Practice*. SIGGRAPH 1997 Course Notes. URL: https://www.cs.cmu.edu/~baraff/sigcourse/notesa.pdf [Stand: 26. September 2021].

[CCB*18] Cordonnier, G. et al. (2018): »Sculpting Mountains: Interactive Terrain Modeling Based on Subsurface Geology«. In: *IEEE Transactions on Visualization and Computer Graphics* 24, H. 5, S.1756–1769.

[CGG*07] Compton, K. et al. (2007): »Creating spherical worlds«. 2007. In: Alexa, M. / Finkelstein, A. (Hg.): *SIGGRAPH ,07: ACM SIGGRAPH 2007 sketches*. New York: Association for Computing Machinery, S. 82.

[CKM15] Compton, K. / Kybartas, B. / Mateas, M. (2015): »Tracery: An Author-Focused Generative Text Tool«. In: Schoenau-Fog, H. et al. (Hg.): *Interactive Storytelling*. Cham: Springer, S. 154–161.

[Co17] Compton, K. (2017): »Practical Procedural Generation for Everyone«. URL: https://www.youtube.com/watch?v=WumyfLEa6bU&t=1641s [Stand: 3. Juli 2021].

[Co86] Cook, R. L. (1986): »Stochastic Sampling in Computer Graphics«. In: *ACM Transactions on Grafics* 5, H. 1, S. 51–72.

[CPG*19] Cortial, Y. et al. (2019): »Procedural Tectonic Planets«. In: *Computer Graphics Forum* 38, H. 2, S. 1–11.

[Cr21] Creswell, J. (2021): »What Is the Difference Between a Roguelike and a Roguelite?«. URL: https://www.cbr.com/roguelike-roguelite-difference/ [Stand: 10. September 2021].

[DH06] Dunbar, D. / Humphreys, G. (2006): »A Spatial Data Structure for Fast Poisson-Disk Sample Generation«. In: Finnegan, J. / Dorsey, J. (Hg.): *SIGGRAPH ,06: ACM SIGGRAPH 2006 Papers*. New York: Association for Computing Machinery, S. 503–508.

[Do11] Dormans, J. (2011): »Level Design as Model Transformation: A Strategy for Automated Content Generation«. In: *PCGames ,11: Proceedings of the 2nd International Workshop on Procedural Content Generation in Games.* New York: Association for Computing Machinery, Artikel 2, S. 1–8.

[GGP*15] Génevaux, J.-D. et al. (2015): »Terrain Modelling from Feature Primitives«. In: Computer Graphics Forum 34, H. 6, S. 198–210.

[GGP*19] Galin, E. et al. (2019): »A Review of Digital Terrain Modeling«. In: *Computer Graphics Forum* 38, H. 2, S. 553–577.

[Go10] González, Á. (2010): »Measurement of Areas on a Sphere Using Fibonacci and Latitude–Longitude Lattices«. In: *Mathematical Geosciences* 42, S. 49–64.

[Gu05] Gustavson, S. (2005): »Simplex noise demystified«. Linköping, Universität.

[Ha64] Halton, J. H. (1964): »Algorithm 247: Radical-inverse quasi-random point sequence«. In: *Communications of the ACM* 7, H. 12, S. 701–702.

[Ho15] Hoffmann, D. W. (2015): *Theoretische Informatik*. 3., überarb. Aufl. München: Carl Hanser Verlag GmbH & Co. KG.

[HRE*08] Hecker, C. et al. (2008): »Real-time Motion Retargeting to Highly Varied User-Created Morphologies«. In: *ACM Transactions on Graphics* 27, H. 3, S. 1–11.

[IUD*10] Iwasaki, K. et al. (2010): »Fast Particle-based Visual Simulation of Ice Melting«. In: *Computer Graphics Forum* 29, H. 7, S. 2215–2223.

[Li68] Lindenmayer, A. (1968): »Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs«. In: *Journal of Theoretical Biology* 18, H. 3, S. 300–315.

[LLC*10] Lagae, A. et al. (2010): »A Survey of Procedural Noise Functions«. In: *Computer Graphics Forum* 29, H. 8, S. 2579–2600.

[LP12] Liebling, M. / Pournin, L. (2012): »Voronoi Diagrams and Delaunay Triangulations: Ubiquitous Siamese Twins«. In: Grötschel, M. (Hg.): *Documenta Mathematics. Optimization Stories*. Deutsche Mathematiker-Vereinigung (DMV), S. 419–431.

[LUA03] Lau, D. L. / Ulichney, R. / Arce, G. R. (2003): »Blue and green noise halftoning models«. In: *IEEE Signal Processing Magazine* 20, H. 4, S. 28–38.

[Mc17] McKendrick, I. (2017): »Continuous World Generation in No Man's Sky«. URL: https://www.youtube.com/watch?v=sCRzxEEcO2Y [Stand: 23. September 2021].

[Microsoft1] Microsoft: »XBox Series X«. URL: https://www.xbox.com/de-DE/consoles/xbox-series-x [Stand: 10. September 2021].

[Mu17] Murray, S. (2017): »Building Worlds in No Man's Sky Using Math(s)«. URL: https://www.youtube.com/watch?v=C9RyEiEzMiU [Stand: 23. September 2021].

[NMK*05] Nealen, A. et al. (2005): »Physically Based Deformable Models in Computer Graphics«. In: *Computer Graphics Forum* 25, H. 4, S. 809–836.

[Ow17] Owen, A. B. (2017): »A randomized Halton algorithm in R«. Stanford, Universität.

[Re83] Reeves, W. T. (1983): »Particle Systems – a Technique for Modeling a Class of Fuzzy Objects«. In: *ACM Transactions on Graphics* 2, H. 2, S. 91–108.

[PDG*19] Peytavie, A. et al. (2019): »Procedural Riverscapes«. In: *Computer Graphics Forum* 38. H. 7, S. 35–46.

[Pe85] Perlin, K. (1985): »An image synthesizer«. In: *ACM SIGGRAPH Computer Graphics* 19, H. 3, S. 287–296.

[Pe01] Perlin, K. (2001): »Noise Hardware«. In: *Real-Time Shading. SIGGRAPH 2001 Course 24 Notes*, Artikel 9, S. 1–24. URL: https://www.csee.umbc.edu/~olano/s2001c24/ch09.pdf [Stand: 26. September 2021].

[Pe02] Perlin, K. (2002): »Improving Noise«. In: *Appolloni, T. (Hg.): SIGGRAPH 2002, SIGGRAPH ,02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York: Association for Computing Machinery, S. 681–682.

[Pe11] Persson, "Notch" M. (2011): »Terrain generation, Part 1«. URL: https://web.archive.org/web/20210813083219/https://notch.tumblr.com/post/3746989361/terrain-generation-part-1 [Stand: 19. September 2021].

[PGG*09] Peytavie, A. et al. (2009): »Arches: a Framework for Modeling Complex Terrains«. In: *Computer Graphics Forum* 28, H. 2, S. 457–467.

[PDG*19] Peytavie, A. et al. (2019): »Procedural Riverscapes«. In: *Computer Graphics Forum* 38, H. 7, S. 35–46.

[PJH18] Pharr, M. / Jakob, W. / Humphreys, G. (2018): *Physically Based Rendering: From Theory To Implementation*. 3., überarb. Aufl. Eigenveröffentlichung. Open Source. URL: https://www.pbr-book.org/ [Stand: 26. September 2021].

[PL90] Prusinkiewicz, P. / Lindenmayer, A. (1990): *The Algorithmic Beauty of Plants*. New York: Springer.

[Sc14] Scholz, D. (2014): P*ixelspiele – Modellieren und Simulieren mit zellulären Automaten.* Berlin/Heidelberg: Springer Spektrum.

[TSD16] Togelius, J. / Shaker N. / Dormans J. (2016): »Grammars and L-systems with applications to vegetation and levels«. In: Shaker, N. / Togelius, J. / Nelson, M. J. (Hg.) (2016): *Procedural Content Generation in Games.* Cham: Springer, S.73–98.

[Unity1] Unity Dokumentation: »Mesh Index Format«. URL: https://docs.unity3d.com/ScriptReference/Mesh-indexFormat.html [Stand: 20. September 2021].

[USK1] USK: »Aufbau-Strategie«. URL: https://usk.de/alle-lexikonbegriffe/aufbau-strategie/ [Stand: 11. September 2021].

[USK2] USK: »Management-Spiel«. URL: https://usk.de/alle-lexikonbegriffe/management/ [Stand: 11. September 2021].

[USK3] USK: »Simulation«. URL: https://usk.de/alle-lexikon-begriffe/simulation/ [Stand: 18. September 2021].

[Wi07] Willmott, A. (2007): »Fast object distribution«. In: Alexa, M. / Finkelstein A. (Hg.) (2007): *SIGGRAPH ,07: ACM SIGGRAPH 2007 sketches*. New York: Association for Computing Machinery, S.80.

[Wo18] Wolf, S. (2018): »DevBlog: Erschaffung einer Insel«. URL: https://anno-union.com/de/devblog-erschaffung-einer-insel/ [Stand: 11. September 2021].

[WR17] Weyer, J. / Roos, M. (2017): »Agent-Based Modeling and Simulation: A Tool for Anticipatory Technological Impact Assessment«. In: *TATuP* 26, H. 3, S.11–16.

[Yu15] Yuksel, C. (2015): »Sample Elimination for Generating Poisson Disk Sample Sets«. In: *Computer Graphics Forum* 34, H. 2, S. 25–32.

Softwarequellen

[S01] *Anno 1800* (2019). Blue Byte Mainz. Ubisoft. [Videospiel].

[S02] *City Skylines* (2015). Colossal Order. Paradox Interactive. [Videospiel].

[S03] *Diablo* (1996). Blizzard North. Blizzard Entertainment. [Videospiel].

[S04] *Die Sims 4* (2014). Maxis. Electronic Arts [Videospiel].

[S05] *Dyson Sphere Program* (2020). Youthcat Studio. [Videospiel].

[S06] *Landwirtschafts Simulator 2020* (2019). Giants Software. [Videospiel].

[S07] *Elite Dangerous* (2014). Frontier Developments. [Videospiel].

[S08] *Factorio* (2013). Wube Software. [Videospiel].

[S09] *FTL: Faster Than Light* (2012). Subset Games. [Videospiel].

[S10] *Godot Engine* (2014-2021). Open-Source. Aktuelle Version: 3.3.3. [Game Engine].
URL: https://godotengine.org/

[S11] *Json.NET* (2021). Newtonsoft. [Framework].
URL: https://www.newtonsoft.com/json

[S12] *Minecraft* (2009). Mojang Studios. [Videospiel].

[S13] *No Man's Sky* (2016). Hello Games. [Videospiel].

[S14] *Planet Coaster* (2016). Frontier Developments. [Videospiel].

[S15] *Planet Zoo* (2019). Frontier Developments. [Videospiel].

[S16] *Rogue* (1980). Toy, M. et al.. [Videospiel].

[S17] *Sid Meier's Civilization 6* (2016). Firaxis Games. 2K Games. [Videospiel].

[S18] *SpeedTree* (2009). Interactive Data Visualization, Inc. [Middware].

[S19] *Spelunky* (2012). Mossmouth. [Videospiel].

[S20] *Spore* (2008). Maxis. EA Games. [Videospiel].

[S21] *Tracery* (2015). Compton K., Kybartas B., Mateas M. [Software].

[S22] *Ultima Ratio Regum* (2015). Johnson M. [Videospiel].
URL: https://www.markrjohnsongames.com/games/ultima-ratio-regum/

[S23] *Unexplored* (2017). Ludomotion. [Videospiel].

[S24] *Unity Game Engine* (2005–2021). Unity Technologies. Aktuelle Version: 2021.1.20f1. [Game Engine].
URL: https://unity3d.com/

[S25] *Unreal Engine* (1998–2021). Epic Games. Aktuelle Version: 4.27. [Game Engine].
URL: https://www.unrealengine.com/

[S26] *Zellmemore* (2019). Schneider, F. [Software].
URL: http://www.zellmemore.de/

[S27] *Zenject* (2019). Open Source (MIT-Lizenz). [Framework].
URL: https://github.com/modesttree/Zenject

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

Bremen, den 02.10.21

Unterschrift
(Sebastian Baier)