

Fachhochschule Wedel

Studiengang Medieninformatik

Eine Einführung in die prozedurale Landschaftsgenerierung

Seminararbeit

Tjark Smalla
Matrikel-Nummer 100554

Betreuer Prof. Bohn

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
1 Einleitung	1
1.1 Datenstrukturen zur Landschaftsspeicherung: Höhenfelder vs. Voxel	2
1.1.1 Höhenfelder	3
1.1.2 Voxel	4
1.2 Implizite vs. explizite Funktionen	5
2 Diamond-Square Algorithmus	6
2.0.1 Der Algorithmus	6
2.0.2 Flexibilität	7
2.0.3 Bewertung im Rahmen der Fragestellung	7
3 Fourier-Spektralsynthese	9
3.0.1 Der Algorithmus	9
3.0.2 Flexibilität	10
3.0.3 Bewertung im Rahmen der Fragestellung	11
4 Noise	12
4.1 Der Algorithmus	13
4.1.1 Lattice-Function	13
4.1.2 Interpolation und Fade-Funktion	13
4.1.3 Value-Noise	15
4.1.4 Gradient-Noise	16
4.1.5 Fractal-Noise	17
4.2 Flexibilität	19
4.2.1 Ridged-Noise	19

Inhaltsverzeichnis

4.2.2	Multifraktal/heterogenes Terrain	21
4.2.3	Hybrid-Multifractal	23
4.2.4	Domain/Range Mapping	25
4.3	Bewertung im Rahmen der Fragestellung	28
5	Fazit	29
	Literaturverzeichnis	30

Abbildungsverzeichnis

1.1.1 Gegenüberstellung einer Heightfield als Textur(links) und der resultierenden Landschaft. Bildquelle: http://tinyurl.com/zu7gond	3
1.1.2 Gerenderte Voxellandschaft mit Überhängen erzeugt aus mehreren Rauschfunktionen. Bildquelle: http://tinyurl.com/jq8vta8	4
2.0.1 Diamond Square Algorithmus bei einem quadratischen Höhenfeld mit der Auflösung 5x5. Bildquelle: http://tinyurl.com/h29sv48	6
3.0.1 Durch Spektralsynthese erzeugte Höhenfelder in 3D gerendert. Bilder entnommen von: http://tinyurl.com/z6lfzmv	10
4.1.1 Fade-Function	14
4.1.2 Screenshot aus dem Beispielprogramm, "NoiseCPU" Szene. Pink: Wert der Lattice-Funktion, Rot: Value-Noise Funktionswert	15
4.1.3 Screenshot aus dem Beispielprogramm, "NoiseCPU" Szene. Pink: Wert der Lattice-Funktion, Rot: Gradient-Noise Funktionswert	16
4.1.4 Texturierte Landschaft im Beispielprogramm die aus fraktalem Perlin-Noise entstanden ist.	18
4.2.1 Texturierte Landschaft im Beispielprogramm die aus Ridged Noise entstanden ist	20
4.2.2 Texturierte Landschaft im Beispielprogramm die aus Ridged-Multifractal-Noise mit $\sigma = 0.2$ entstanden ist.	22
4.2.3 Texturierte Landschaft im Beispielprogramm die aus Hybrid-Multifractal-Noise mit $\sigma = 0.5$ und $w = 1.15$ entstanden ist.	24
4.2.4 Bias-Funktion	26
4.2.5 Verhalten der Gain-Funktion bei verschiedenen Werten für g . Bilder entnommen von: http://tinyurl.com/zgb6jaj	26

Abbildungsverzeichnis

1 Einleitung

Während die künstliche Erzeugung von Landschaften¹ in der Computergrafik schon seit den 80ern ein Thema ist, beschäftigte sich auch Hersteller von Computerspielen immer wieder mit dieser Technik. Insbesondere in den letzten 10 Jahren begannen immer mehr Entwickler diese Technik anzuwenden.

Die prozedurale Generierung ihrer Landschaften bietet ihnen viele Vorteile. Durch die mittlerweile sehr großen Spielwelten ist es nicht unüblich bis zu 300 Leute über 2 Jahre an nur einem großen Projekt zu beschäftigen. Ein großer Teil dieser Zeit wird in die Ausgestaltung der Spielwelt gesteckt. Prozedurale Techniken erlauben es ihnen realistische Spielwelten zu erschaffen die nahezu unendlich groß sind. Als bekanntestes Beispiel ist hier sicherlich das Spiel Minecraft vom Studio Mojang zu nennen, welches einen Großteil seiner Faszination aus der komplett prozedural erzeugten veränderbaren Landschaft zieht.

Im folgenden werden 3 bewährte Algorithmen zur Erzeugung von Landschaften vorgestellt und daraufhin untersucht, in wie weit sie zur Speicherung einer nahezu unendlich großen Landschaft wie in Minecraft geeignet sind.

Zuerst wird der *Diamond-Square* Algorithmus[FFC82] vorgestellt, welchen man verallgemeinert auch als einen Polygon unterteilungs Algorithmus bezeichnen kann. Danach wird kurz die *Spektalesynthese* mit der Fourier-Methode erläutert bevor es eine Einführung in die Welt der Rauschfunktionen gibt. Neben der Synthese von Landschaften sind diese Algorithmen vielseitig einsetzbar. Insbesondere bei der bereits erwähnten Kategorie der Rauschfunktionen ist davon auszugehen, dass sie auch in proprietärer 3D-Software wie Computerspielen trotz ihres Alters noch als wichtiger Algorithmus genutzt werden. Dies liegt vor allem an ihrer Flexibilität sowie Skalierbarkeit in mehreren Dimensionen durch die auch Wolken, Feuer, Rauchverwirbelung und sogar Fell auf einer Oberfläche

¹ Landschafts-Synthese

1 Einleitung

erfolgreich dargestellt wurden [DSE03]². Neben der Bewertung soll dabei insbesondere die Erklärung der Grundlagen dieser Algorithmen Zielsetzung sein. Zu diesem Zwecke wird neben der Erklärung im Unterkapitel "*Der Algorithmus*" jedes Verfahren auf seine Erweiter- bzw. Anpassbarkeit("Flexibilität") hin untersucht, bevor es eine abschließende "*Bewertung im Rahmen der Fragestellung*". In diesem letzten Schritt wird das Verfahren daraufhin untersucht, in wie weit es effektiv zur Speicherung und Auswertung einer nahezu unendlich großen Landschaft geeignet ist.

Der Diamond-Square Algorithmus sowie verschiedene Rauschfunktionen sind in dem dieser Arbeit beiliegendem Unity-Projekt in C# bzw. HLSL³ implementiert und lassen sich durch Unity auf Windows, Unix und Mac OS basierten Betriebssystemen kompilieren.

1.1 Datenstrukturen zur Landschaftsspeicherung: Höhenfelder vs. Voxel

Landschaften werden in der Regel entweder in einem Höhenfeld oder in Voxeln gespeichert. Der wesentliche Unterschied der beiden Techniken liegt in ihren Dimensionen. Das Höhenfeld speichert in einem ganzzahligen, zweidimensionalen Koordinatensystem zu jedem Punkt den dazugehörigen Höhenwert der Landschaft, während ein Voxel den Dichtewert eines Punktes in einem dreidimensionalem Koordinatensystem darstellt. Beide Datenstrukturen haben den Vorteil, dass sich die Position jedes Wertes in der Landschaft implizit aus der Position zu seinen Nachbarn bestimmen lässt. Eine zusätzliche Speicherung seiner Koordinate ist also nicht notwendig. Anders wäre es bei Polygonen, wo jeder Vertex eine Koordinate im Raum hat. Sie sind daher gut für Szenen mit viel leerraum geeignet, welcher allerdings in den hier vorgestellten Datenstrukturen zu unnötig gespeicherten Daten führe würde

2 Wolken: Kapitel 9, Feuer(S,125) & Rauchverwirbelung: Kapitel 12, Rauch, Wolken, Fell: S.297-302

3 High Level Shading Language

1 Einleitung

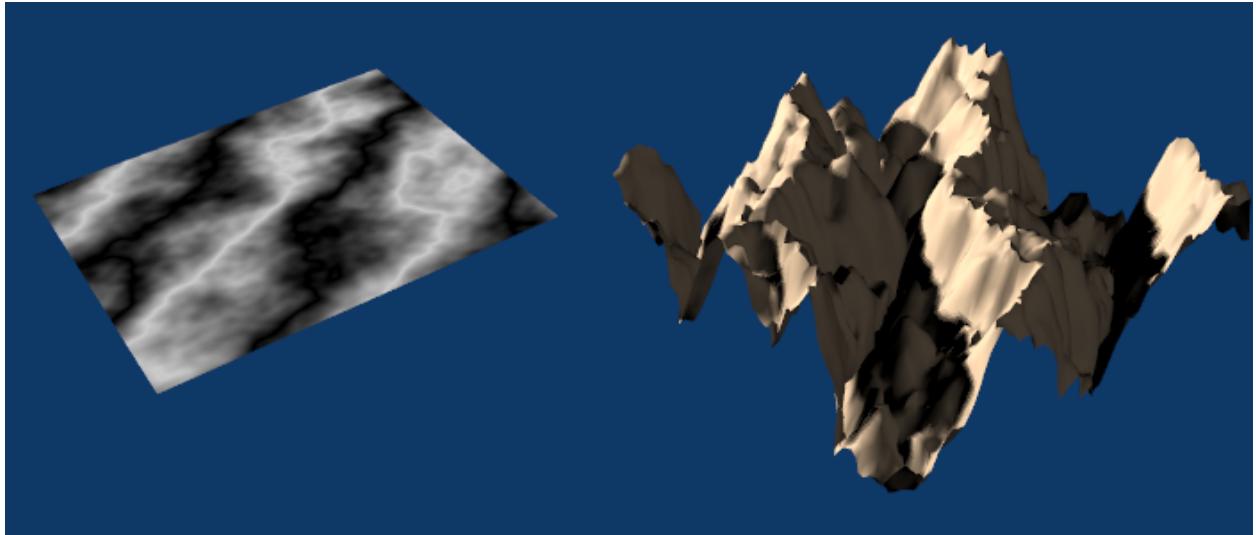


Abbildung 1.1.1: Gegenüberstellung einer Heightfield als Textur(links) und der resultierenden Landschaft. Bildquelle: <http://tinyurl.com/zu7gond>

1.1.1 Höhenfelder

Die meisten Algorithmen beziehen sich auf die Erzeugung von Landschaften mit Hilfe von Höhenfeldern. Dies ist vermutlich vor allem dem geschuldet, dass viele Algorithmen recht alt sind und sich Voxel basierte Verfahren aufgrund des Speicherplatzmangels nicht lohnten. Auch wenn Höhenfelder bereits zu sehr realistischen Ergebnissen großer Landschaftsstriche führen, haben sie einen entscheidenden Nachteil. Da pro Punkt im Koordinatensystem nur ein Höhenwert gespeichert werden kann sind Höhlen oder Überhänge nicht möglich.

Aufgrund der vereinfachten Algorithmen und der großen Verbreitung wird sich im folgendem auf Höhenfelder beschränkt. Eine Erweiterung in die dritte Dimension ist mit dem Diamond-Square Algorithmus oder den Rauschfunktionen jedoch problemlos möglich. Ein weiterer Vorteil von Höhenfeldern ist, dass sich durch die simple Verbindung nebeneinanderliegender Punkte sehr einfach ein Polygon erzeugen lässt, welches Hardwrebeschleunigt gerendert werden kann.

1 Einleitung

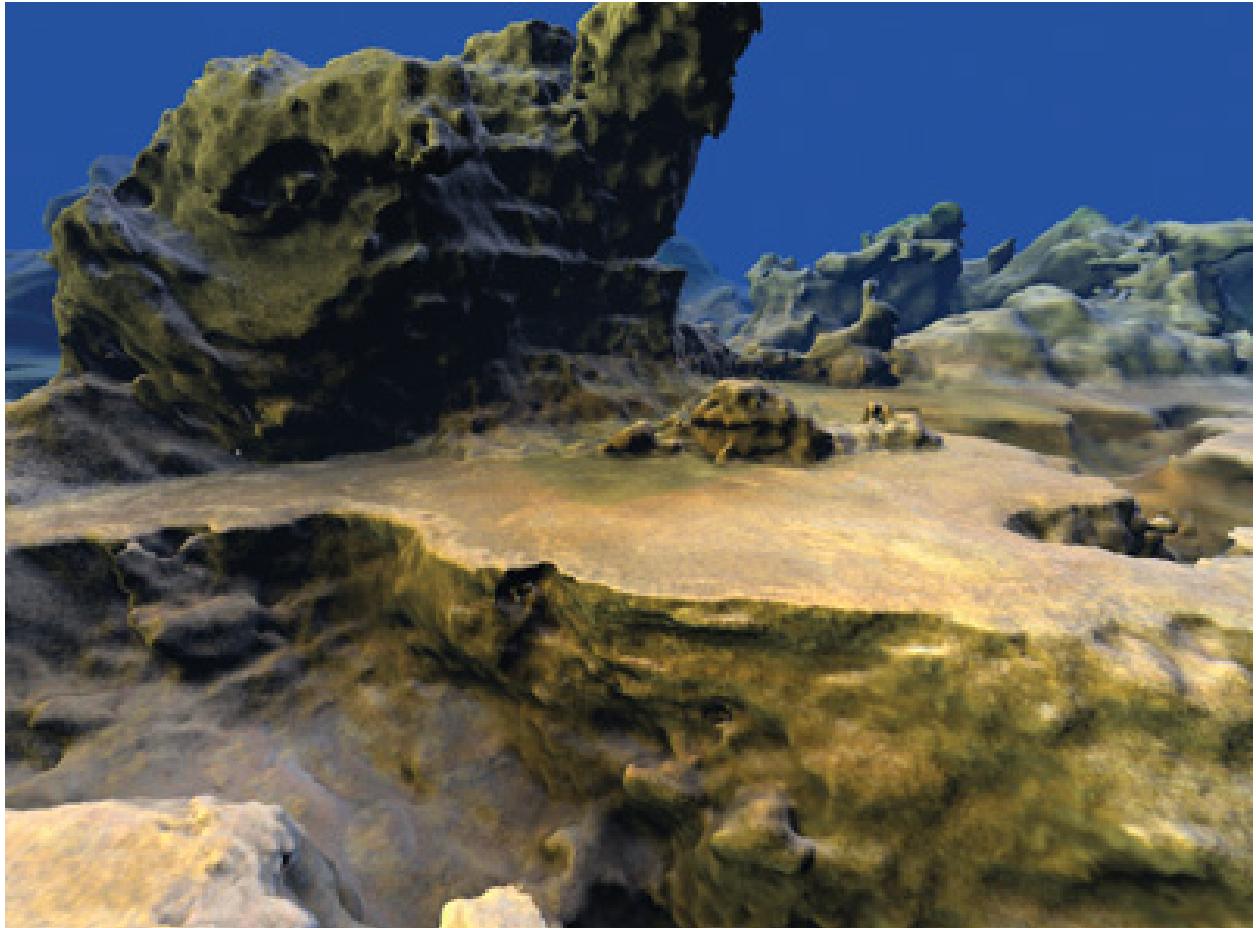


Abbildung 1.1.2: Gerenderte Voxellandschaft mit Überhängen erzeugt aus mehreren Rauschfunktionen. Bildquelle: <http://tinyurl.com/jq8vta8>

1.1.2 Voxel

Der Begriff Voxel leitet sich aus den Begriffen Volumen und Pixel ab. Durch die Speicherung in einem dreidimensionalem Koordinatensystem wird nun auch die, bei den Höhenfeldern explizit gespeicherte, Höhe eines Punktes implizit gespeichert. Dies ermöglicht es eine weitere Information für jeden Punkt explizit zu speichern. In der Regel ist dies ein Dichtewert der es ermöglicht verschiedene Arten von Materialien oder die Opazität zu simulieren.

Um Voxel hardwarebeschleunigt zu rendern müssen auch diese in ein Polygon umgewandelt werden. Dies funktioniert normalerweise über Raycasting oder den Marching Cube Algorithmus.

1.2 Implizite vs. explizite Funktionen

Alle hier vorgestellten Methoden lassen sich in 2 Gruppen einteilen: implizite und explizite Funktionen. Während eine explizite Funktion alle Höhenpunkte auf einmal berechnet lässt sich die implizite Funktion für jeden Punkt, also jede Koordinate, isoliert auswerten.

Durch die Unabhängigkeit der Berechnung für jeden einzelnen Punkt lassen sich implizite Algorithmen sehr effizient parallel auf einer GPU berechnen⁴. Dies ermöglicht die Ausführung zur Laufzeit, während explizite Methoden in der Regel vor oder bei Programmstart einmalig berechnet werden und deren Ergebnisse in einer Textur gespeichert werden. Dies hat den Vorteil, dass der Speicherbedarf enorm sinkt.

Ein Einsatzgebiet für diese Technik ist das Bump-Mapping bzw. Displacement Mapping bei der zusätzlichen Höhenwerte auf ein Objekt durch Shading oder neue Vertices auf der Objektoberfläche hinzugefügt werden[Rus]. Da moderne Echtzeitspiele immer mehr und immer größere Texturen verwenden steigt der Bedarf an Speicher enorm wenn für jede Textur noch eine Normal/Bump/Displacement Map gespeichert werden muss. Implizite Methoden erlauben es, anstatt der Texturen einige Parameter in Form von Floats und Integern zu speichern. Auch eine Anpassung des Detailgrades ist zur Laufzeit ohne Probleme möglich, während bei der Detailgrad bei expliziten Methoden von der Auflösung der Textur abhängt⁵.

Der Diamond-Square Algorithmus sowie die Spektralsynthese gehören zu der Gruppe der expliziten Algorithmen, während die Rauschfunktionen implizit auswertbar sind.

4 Siehe Beispielimplementierung in einem Vertex-Shader

5 Diese Eigenschaften lassen sich zwar auch durch die Berechnung von expliziten Methoden zur Laufzeit erreichen, jedoch lassen diese sich wie erwähnt nicht effektiv durch die GPU beschleunigen wodurch die Berechnung innerhalb eines Frames unperformant ist.

2 Diamond-Square Algorithmus

Die Vorteile des Diamond-Square Algorithmus liegen insbesondere in seinen, trotz seiner einfachen Implementierung, optisch plausiblen Ergebnissen. Der Algorithmus unterteilt sich pro Rekursionsschritt in 2 Phasen¹: der Diamond und der Square Step. Das zu berechnende Höhenfeld muss dabei in der Auflösung eine Breite bzw. Höhe von $2n + 1$ wobei $n \in \mathbb{N}$ aufweisen.

2.0.1 Der Algorithmus

Zur Initialisierung werden den vier Eckpunkte des zu berechnenden Höhenfeldes jeweils zufällige Höhenwerte zugewiesen. Außerdem wird eine von der Rekursionstiefe abhängige Offset-Funktion $O(k) = (\text{rnd}() * 2 - 1) / 2^k$ definiert, wobei rnd() einen Zufallswert zwischen 0-1(inklusive) zurückgibt und k der Rekursionsschritt ist.

Im Diamond Step wird nun der Mittelpunkt des Höhenfeld gesucht und mittels einer beliebigen Interpolation² zwischen den 4 Höhenwerten der Eckpunkte ein Wert ermittelt, welcher dann mit dem Ergebnis der Offset-Funktion addiert wird. Bei dem Square Step

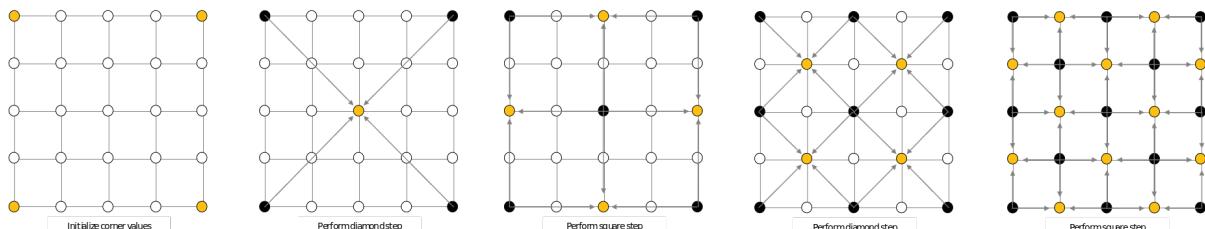


Abbildung 2.0.1: Diamond Square Algorithmus bei einem quadratischen Höhenfeld mit der Auflösung 5x5. Bildquelle: <http://tinyurl.com/h29sv48>

1 Auch *Steps* genannt

2 In der Beispieldarstellung wurde eine bilineare Interpolation gewählt

2 Diamond-Square Algorithmus

werden nun die jeweiligen mittleren Randpunkte zwischen den bereits initialisierten Eckpunkten durch eine einfache Interpolation zwischen den 2 nächstliegendsten Eckpunkten berechnet. Wie in Abbildung 2.0.1 zu sehen ergeben sich daraus 4 weitere Vierecke mit berechneten Eckpunkten, auf die der Algorithmus erneut angewandt wird. Dies wird solange wiederholt, bis alle Vertices berechnet sind.

2.0.2 Flexibilität

Das resultierende Höhenfeld lässt sich durch eine Veränderung der Offset-Funktion relativ flexibel anpassen. So besteht zum Beispiel die Möglichkeit, die Offset-Funktion von dem interpolierten Höhenwert des aktuellen Punktes abhängig so machen und dadurch ein heterogenes Landschaftsbild zu erzeugen.

2.0.3 Bewertung im Rahmen der Fragestellung

Zur Speicherung des resultierenden Höhenfeldes ist es ausreichend den Seed³ des Pseudozufallsgenerators zu speichern. Eine nahezu unendliche Landschaft wie sie etwa Minecraft bietet wäre mit diesem Algorithmus aus 2 Gründen allerdings nicht möglich. Wie bereits besprochen verwendet Minecraft Voxel, welche mit dem Diamond Square Algorithmus allerdings nicht berechnet werden können⁴. Außerdem würde die Generierung der Spielwelt besonders für Spieler mit Leistungsschwachen Computern sehr lange dauern. Um das zu vermeiden könnte man die Landschaft in rechteckige Patches⁵ aufteilen, die erst berechnet werden wenn der Spieler auf sie trifft. Da der Pseudozufallsgenerator jedoch seine Werte immer in der gleichen Reihenfolge ausgibt wäre die Anordnung der Patches zueinander abhängig von der Reihenfolge in der sie geladen werden.

3 Eingangswert für einen Pseudozufallsgenerator. https://www.wikiwand.com/en/Random_seed

4 Der Diamond-Square Algorithmus ist für Höhenfelder konzipiert. Für Voxel sollte sich das Prinzip allerdings auch anpassen lassen

5 Kleine Abschnitte der Landschaft, einzeln berechnen und aneinanderfügen

2 Diamond-Square Algorithmus

```
1  /**
2  *
3  * @param int          left    left coordinate of current rectangle
4  * @param int          top     top coordinate of current rectangle
5  * @param int          right   right coordinate of current rectangle
6  * @param int          bottom  bottom coordinate of current rectangle
7  * @param PseudoRandomFunction rnd    PRNG with values between -1 and 1
8  * @param float         r      Lacunarity, factor with which offset will be decreased each recursion step
9  * @param int          recStep Number of the current recursion step
10 */
11 public void DiamondSquare(int left, int top, int right, int bottom, PseudoRandomFunction rnd, float r, int recStep) {
12     Assert.IsTrue(Heightmap.GetLength(0) % 2 == 1 && Heightmap.GetLength(1) % 2 == 1);
13
14     // Diamond step
15     int xCenter = left + (right - left)/2,
16         yCenter = top + (bottom - top)/2;
17
18     Heightmap[xCenter, yCenter] = BillinearInterpolation(Heightmap[left, top],
19                                                       Heightmap[right, top],
20                                                       Heightmap[left, bottom],
21                                                       Heightmap[right, bottom], 0.5f, 0.5f) + rnd() / Mathf.Pow(r, recStep+1);
22
23     // Square step
24     Heightmap[left, yCenter] = LinearInterpolation(Heightmap[left, top], Heightmap[left, bottom], 0.5f) + rnd() / ,
25     Mathf.Pow(r, recStep+1);
26     Heightmap[xCenter, top] = LinearInterpolation(Heightmap[left, top], Heightmap[right, top], 0.5f) + rnd() / ,
27     Mathf.Pow(r, recStep+1);
28     Heightmap[right, yCenter] = LinearInterpolation(Heightmap[right, top], Heightmap[right, bottom], 0.5f) + rnd() / ,
29     Mathf.Pow(r, recStep+1);
30     Heightmap[xCenter, bottom] = LinearInterpolation(Heightmap[left, bottom], Heightmap[right, bottom], 0.5f) + rnd() / ,
31     Mathf.Pow(r, recStep+1);
32
33     // Recursion if uncalculated pixel left
34     if (right - left >= 2) {
35         yield return DiamondSquare(left, top, xCenter, yCenter, rnd, r, recStep+1); // Top left
36         yield return DiamondSquare(xCenter, top, right, yCenter, rnd, r, recStep+1); // Top right
37         yield return DiamondSquare(left, yCenter, xCenter, bottom, rnd, r, recStep+1); // bottom left
38         yield return DiamondSquare(xCenter, yCenter, right, bottom, rnd, r, recStep+1); // bottom right
39         if (!UseSteps)
40             UpdateTerrainHeightmap();
41     }
42 }
```

Diamond-Square Implementierung C#

3 Fourier-Spektralsynthese

Bei der Spektralsynthese wird das Höhenfeld in der Frequenz-Domäne modelliert und durch die inverse Fourier Transformation in ein Höhenfeld in der Raum-Domäne umgewandelt.

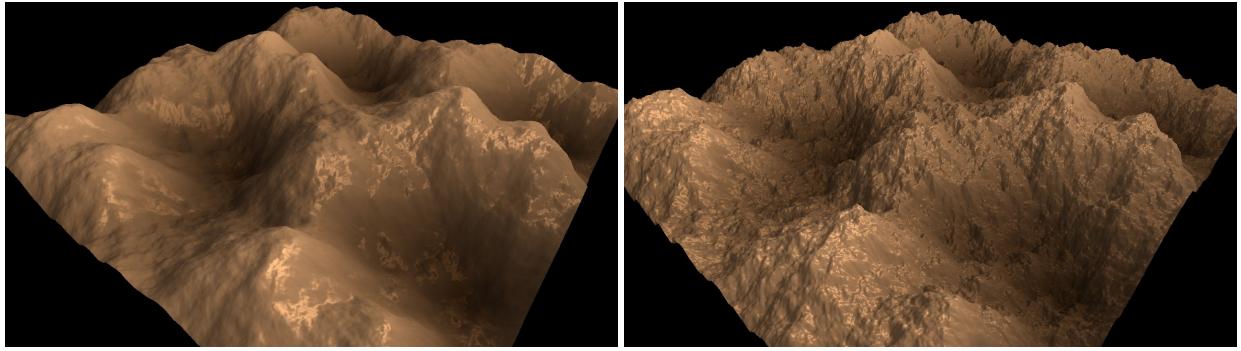
3.0.1 Der Algorithmus

Als ersten Schritt sollten sich Gedanken darüber gemacht werden, welche Frequenzanteile für einen plausiblen Eindruck notwendig sind und wie stark diese ausgeprägt sein sollten. Die niederfrequenten Anteile einer Landschaft formen mit großer Amplitude die Hügel und Täler, während höherfrequente Anteile mit niedriger Amplitude die Unregelmäßigkeiten und die Rauheit des Bodens simulieren.

Um das Höhenfeld zu modellieren wird weißes Rauschen erzeugt welches im Anschluss durch einen beliebigen Filter die unerwünschten Frequenzanteile entfernt bzw. deren Amplitude verringert. Weißes Rauschen bezeichnet ein Signal welches in der Frequenz-Domäne allen Frequenzen eine konstante Amplitude zuordnet[whi]. Für den Zweck der Höhenfeldsynthese ist eine Annäherung vollkommen ausreichend. Zur Erzeugung des weißen Rauschens wird in der Raum-Domäne ein Bild erzeugt in dem der Farbwert eines Pixels durch einen einfachen, C-ähnlichen, Pseudozufallsgenerator erzeugt wird. Dieses Bild verhält sich - in die Frequenz-Domäne transformiert - ähnlich wie weißes Rauschen.

Der Filter den wir nun auf dieses Rauschen anwenden bestimmt entscheidend das Aussehen des späteren Höhenfeldes. Plausible Ergebnisse lassen sich mit dem Filter $f = 1/f^\beta$ erzeugen. Das resultierende Frequenz Bild entspricht bei der richtigen Wahl des Filters dem von gebrochener Brownscher Bewegung[VOS89]. Diese ist geeignet eine Vielzahl von natürlichen Phänomenen die auf Selbstähnlichkeit basieren passend zu beschreiben[BMA].

3 Fourier-Spektralsynthese



(a) Spektralsynthese mit Filter $f = 1/f^{2.4}$

(b) Spektralsynthese mit Filter $f = 1/f^{2.0}$

Abbildung 3.0.1: Durch Spektralsynthese erzeugte Höhenfelder in 3D gerendert.
Bilder entnommen von: <http://tinyurl.com/z6lfzmv>

In Abbildung 3.0.1 sieht man das in 3D visualisierte Ergebnis des resultierenden Höhenfeldes. Besonders zu bemerken sind der deutlich höhere Detailgrad der Landschaft in Abbildung 3.0.1b welcher aus den höheren Amplituden der Hochfrequenten Anteile folgt.

3.0.2 Flexibilität

Die Spektralsynthese ist sehr flexibel anpassbar, da sich mit der Änderung des Filters beliebig viele Effekte mit wenig Implementierungsaufwand erzeugen lassen. Klare Nachteile ergeben sich jedoch aus der genutzten Fourier bzw. der inversen Fourier Transformation, welche eine Auswertung innerhalb eines Frames nahezu unmöglich macht. Da die Frequenzen an jedem Ort des Höhenfeldes wirken hat das resultierende Gelände einen konstanten Detailgrad. Wünschenswert wäre es jedoch, die höherfrequenten Anteile in den tiefer liegenden Regionen nicht anzuzeigen, da Täler eher von lang geschwungenen Hügeln als von kantigen Felsformationen geprägt sind.¹. Im Vergleich zum Diamond-Square Algorithmus ist die Auflösung des Höhenfeldes jedoch irrelevant für die Funktionalität.

¹ Man spricht dann auch von heterogenem Terrain

3 Fourier-Spektralsynthese

3.0.3 Bewertung im Rahmen der Fragestellung

Die Speicherung großer Geländeabschnitte erweist sich als einfach. Gespeichert werden muss nur das zugehörige Rauschbild 3.0.1 sowie der verwendete Filter werden. Da das Rauschbild aus einem Pseudozufallsgenerator kommt, reicht hier auch der Seed aus womit sich der Speicheraufwand auf einige Bytes verringert. Die Erzeugung einer nahezu unendlich großen Landschaft würde jedoch sehr viel Rechenzeit in Anspruch nehmen und müsste komplett im temporären Speicher gehalten werden. Eine Alternative wäre die Verwendung wie in 2.0.3 vorgeschlagen, allerdings ist dies nicht möglich, da es keine Möglichkeit gibt einen glatten Übergang zwischen zwei Patches zu gestalten.

4 Noise

Synthetisch erzeugtes Rauschen (*engl. Noise*) erweist sich als hilfreiches Mittel zur Erzeugung von zufällig erscheinenden Strukturen. Als wohl bekannteste Implementierung ist hier die Implementierung von Ken Perlin[K85] zur Erzeugung einer Marmortextur auf einer Vase zu nennen¹.

Neben umfangreichen Anpassungsmöglichkeiten durch verschiedene Parameter ist die Performance dieses Verfahrens ein entscheidender Grund für die Nutzung. Noise verbraucht extrem wenig Speicher, ist relativ einfach zu berechnen und ist zu jeder Zeit an einer beliebigen Stelle auswertbar, was es auch für Echtzeitanwendungen geeignet macht.[HH]

Dieses Kapitel soll ein grundlegendes Verständnis über Noise-Funktionen bieten. Dazu werden zuerst grundlegende Komponenten, welche jeder Implementierung zugrunde liegen, erläutert. Anschließend werden *Value*-4.1.3, *Gradient-Noise*4.1.4 sowie *Fractal-Noise*4.1.5 erklärt, bevor es einen Ausblick auf verschiedenen Abwandlungen von *Fractal Noise* gibt. Da mit Höhenfeldern gearbeitet wird und das Verständnis im Vordergrund liegt wird in den Beispiel nur auf Zweidimensionale Rauschfunktionen eingegangen. Die Begriffe Noise und Rauschfunktion werden synonym verwendet.

Weitere ausführliche Beschreibung in [Bur08] (Gradient-Noise) und in [Gus05] zu *Simplex-Noise* welches eine, besonders in höheren Dimensionen, performantere Implementierung von Perlin Noise darstellt.

¹ Auch als *Perlin-Noise* bezeichnete Implementierung von Gradient Noise in 3D

4.1 Der Algorithmus

4.1.1 Lattice-Function

Der erste Schritt zur Erzeugung von Noise ist in der Regel eine sogenannte *Lattice(Gitter)-Funktion*[AC] der Form $l(\vec{k}) : \mathbb{Z}^n \mapsto [-1 - 1]$. Diese dient zur Beschreibung eines Gitters, welches die Form unserer zukünftigen Noise-Funktion bestimmen wird und jeder Integer-Koordinate eine Wert im Intervall -1 - 1 zuweist. Die Funktion muss dabei unbedingt deterministisch sein². Perlin verwendet bei seiner Implementierung eine Hashfunktion die auf ein, mit zufälligen Werten gefülltes, Array zugreift. Ein Zugriff auf dieses Array mittels Modulo Operation hätte zur Folge, dass sich im Höhenfeld wiederkehrende Strukturen zeigen würden sobald die Funktion über den Rahmen des Arrays hinausgreift.

4.1.2 Interpolation und Fade-Funktion

Um aufbauend auf der Lattice-Funktion 4.1.1 eine Funktion $S(\vec{x}) : \mathbb{R}^n \mapsto \mathbb{R}, \vec{x} \in \mathbb{Z}^n$ zu definieren wird zwischen benachbarten Gitterpunkten lokal interpoliert. Dafür wird eine sogenannte Fade-Function[Per] der Form $f(t) : [0 - 1] \mapsto [0 - 1]$ definiert, welche den Übergang zwischen den Gitterpunkten steuert.

Um überhaupt eine stetige Noise-Funktion zu ermöglichen, muss

$$f(0) = 0 \wedge f(1) = 1 \quad (4.1)$$

gelten. Damit der Übergang zwischen den Gitterpunkten möglichst glatt und damit natürlich wirkt, sollte jedoch eine Stetigkeit von C^2 an den Übergängen und damit die Eigenschaften

$$f'(0) = f'(1) = 0 = f''(0) = f''(1) \quad (4.2)$$

gelten.

Dafür wird im folgenden das Polynom $f(t) = 6t^5 - 15t^4 + 10t^3$ verwendet, welches auch in Perlins Referenzimplementierung Verwendung findet[Bur08] und alle geforderten

² Siehe 4.1.5, für jede Koordinate eines Gitterpunktes muss sie also immer denselben Funktionswert liefern.

4 Noise

Eigenschaften erfüllt. Die Wahl der Fade-Funktion ist entscheidend für die Optik der resultierenden Funktion. In Abbildung 4.1.3 sieht man, wie die Fade Funktion zwischen jedem Lattice-Funktionswert wiederzuerkennen ist³.

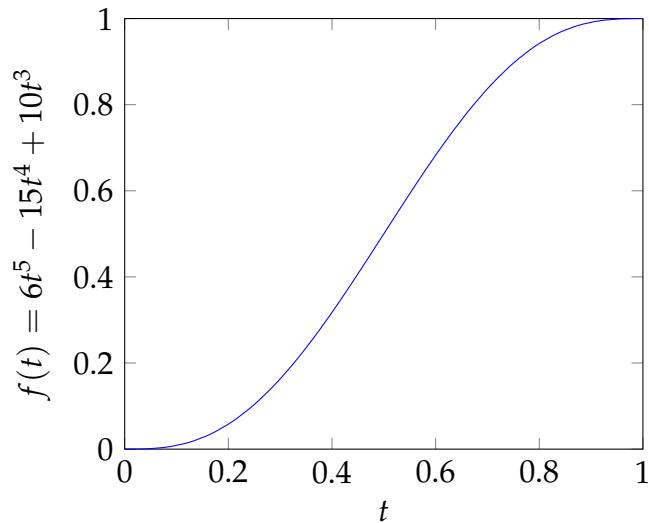


Abbildung 4.1.1: Fade-Function

³ Besonders auffällig zwischen dem 4. und dem 5. Wert

4 Noise

4.1.3 Value-Noise

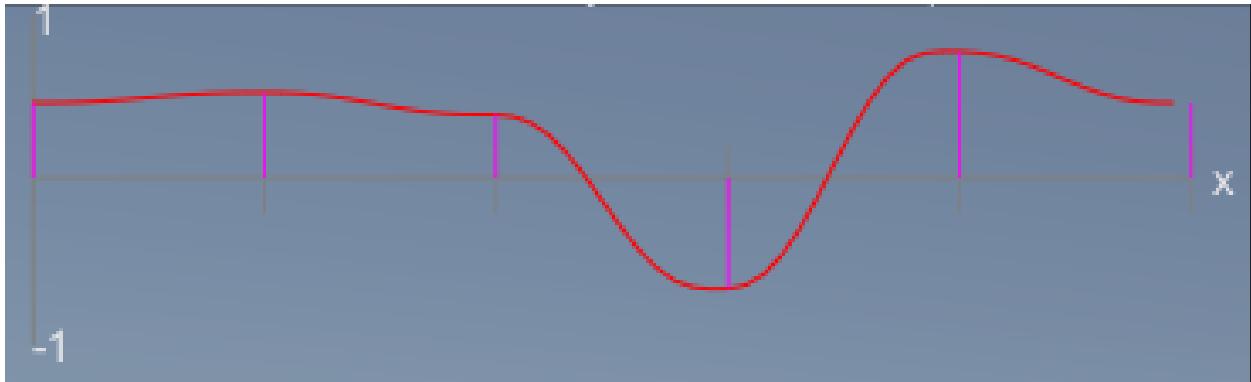


Abbildung 4.1.2: Screenshot aus dem Beispielprogramm, "NoiseCPU" Szene. Pink: Wert der Lattice-Funktion, Rot: Value-Noise Funktionswert

Value-Noise ist die wohl naivste Implementierung einer Noise-Funktion. Bei ihr werden die Gitterpunktswerte, welche durch die *Lattice-Funktion* 4.1.1 erzeugt wurden, als Höhenwerte interpretiert.

Die mit $f(t)$ gebildete, interpolierende Funktion $vnoise(\vec{x}) = noise(\vec{x}) = S(\vec{x})$ definiert nun die Value-Noise-Funktion.

Untenstehend ist eine Implementierung in C# für eine Zweidimensionale Rauschfunktion zu sehen. Noise lässt sich problemlos in mehrere Dimensionen skalieren. Einzig die Interpolation der Werte muss hier angepasst werden. In der Implementierung ist zu sehen, wie die Gitterpunktswerte - ähnlich einer *billinearen Interpolation* - mit der *Fade-Function* interpoliert werden.

```
1 public float ValueNoise(float x, float y) {
2     int floorX = Mathf.FloorToInt(x),
3         ceilX = Mathf.CeilToInt(x),
4         floorY = Mathf.FloorToInt(y),
5         ceilY = Mathf.CeilToInt(y);
6     float tx = x - floorX,
7         ty = y - floorY;
8
9     return FadeFunction(1 - ty) *
10        (LatticeFunc(floorX, floorY) * FadeFunction(1 - tx) + FadeFunction(tx) * LatticeFunc(ceilX, floorY))
11        +
12        FadeFunction(ty) *
13        (LatticeFunc(floorX, ceilY)*FadeFunction(1 - tx) + FadeFunction(tx)*LatticeFunc(ceilX, ceilY));
14 }
```

Value-Noise Implementierung C#

4.1.4 Gradient-Noise

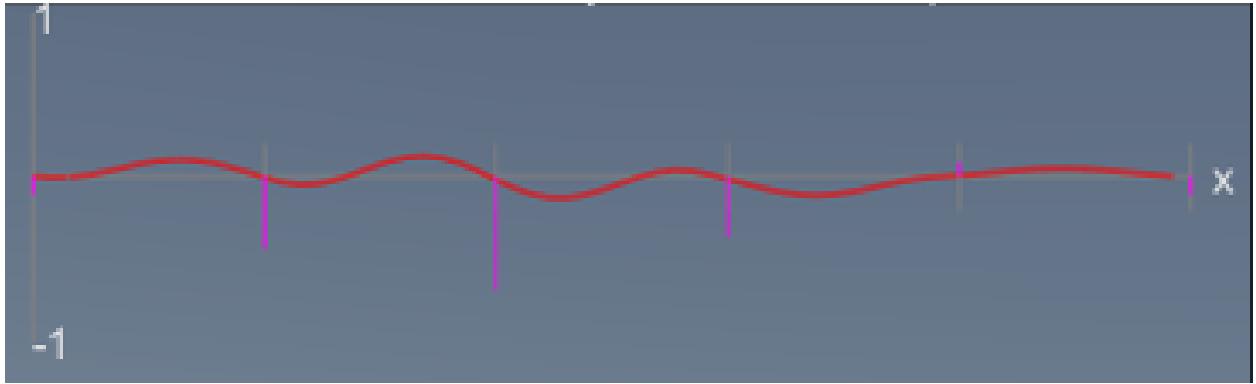


Abbildung 4.1.3: Screenshot aus dem Beispielprogramm, "NoiseCPU" Szene. Pink: Wert der Lattice-Funktion, Rot: Gradient-Noise Funktionswert

Der vorher erwähnte Value-Noise kann, je nach Parameterwahl, noch ein unruhiges Rauschen erzeugen. Um die Übergänge zwischen den Gitterpunktswerten noch sanfter und damit natürlicher aussehen zu lassen wurde der Gradient-Noise erfunden. Hier werden die Gitterpunktswerte nicht als Höhenwerte, sondern als Gradienten an den Nullstellen der Noise-Funktion gesehen. Es ergibt sich also:

$$\text{noise}(\vec{k}) = 0 \wedge \text{noise}'(\vec{k}) = g\text{noise}(\vec{k}), k \in \mathbb{Z}^n. \quad (4.3)$$

In der untenstehenden 2D C# Implementierung ist zu sehen, wie ein Höhenwert für den aktuellen Punkt \vec{x} über das Skalarprodukt⁴ zwischen dem Gradienten und der relativen Position $\begin{pmatrix} tx \\ ty \end{pmatrix}$ $tx, ty \in [0 - 1]$ und anschließend über die bekannte Interpolation berechnet wird.

```

1  public float GradientNoise(float x, float y) {
2      int floorX = Mathf.FloorToInt(x),
3          ceilX = Mathf.CeilToInt(x),
4          floorY = Mathf.FloorToInt(y),
5          ceilY = Mathf.CeilToInt(y);
6      float tx = x - floorX,
7          ty = y - floorY,
8          //Calc slopes
9      n00 = Vector2.Dot(LatticeFunc(floorX, floorY), new Vector2(tx, ty)),
10     n10 = Vector2.Dot(LatticeFunc(ceilX, floorY), new Vector2(tx-1, ty)),
11     n01 = Vector2.Dot(LatticeFunc(floorX, ceilY), new Vector2(tx, ty-1)),
12     n11 = Vector2.Dot(LatticeFunc(ceilX, ceilY), new Vector2(tx-1, ty-1));
13
14     return FadeFunction(1 - ty) *
15         (n00 * FadeFunction(1 - tx) + n10 * FadeFunction(tx))
16         +

```

⁴ engl. Dot-Product

4 Noise

```

17     FadeFunction(ty) *
18     (n01 * FadeFunction(1 - tx) + n11 * FadeFunction(tx));
19 }
```

Gradient-Noise Implementierung C#

4.1.5 Fractal-Noise

Die bisher behandelten Noise-Funktionen erzeugen zwar natürlich erscheinende Zufallswerte, wenn man diese jedoch auf eine Heightmap überträgt wird deutlich, dass es ihr an Details fehlt um realistisch zu wirken.

Um den Detailgrad der Noise-Funktion beliebig zu erhöhen, wird sie mit einer gestauchten, in der Amplitude verringerten, Version ihrer selbst addiert. Dieses Verfahren lässt sich beliebig oft anwenden, was einen hohen Detailgrad erlaubt. Die verschiedenen Frequenzen die dadurch entstehen werden wegen der Halbierung bzw. Verdopplung bei Perlin's Implementierung⁵ auch als *Oktaven* bezeichnet.

In [Sau88] wird daher folgende Formel definiert:

$$fractal(\vec{x}) = \sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} gnoise(r^k \vec{x}). \quad (4.4)$$

Der Exponent H wird je nach Quelle unterschiedlich bezeichnet. In [JS06] etwas wird er als Hurst-Exponent $H = 3 - D$ definiert, wobei D die fraktale Dimensionalität beschreibt. In der Praxis ist er als Exponent anzusehen, welcher die Abnahmgeschwindigkeit der Amplituden der höheren Frequenzen maßgeblich beeinflusst.

In Gleichung 4.4 beschreibt r die sogenannte Lacunarity. Mit ihr lässt sich steuern, wie stark die Amplitude bei jeder Oktave abnimmt bzw. sich die Frequenz erhöht. Durch die Anpassung dieser Parameter lässt sich das Verhalten der Noise-Funktion gezielt steuern. Eine Implementierung in C# findet sich unten. In Abbildung 4.1.4 ist eine gerenderte Variante von Perlin-Noise zu sehen.

⁵ $L = 2$

4 Noise

```
1 public float FractalValueNoise(float x, float y, float lacunarity, float H, int k) {
2     if (k <= 0) {
3         return 0;
4     }
5
6     return FractalValueNoise(x, y, lacunarity, H, -k)
7         + ValueNoise(x * Mathf.pow(lacunarity, k), y * Mathf.pow(lacunarity, k)) / Mathf.pow(lacunarity, k*H);
8 }
```

Simple Fractal-Noise Implementierung C#

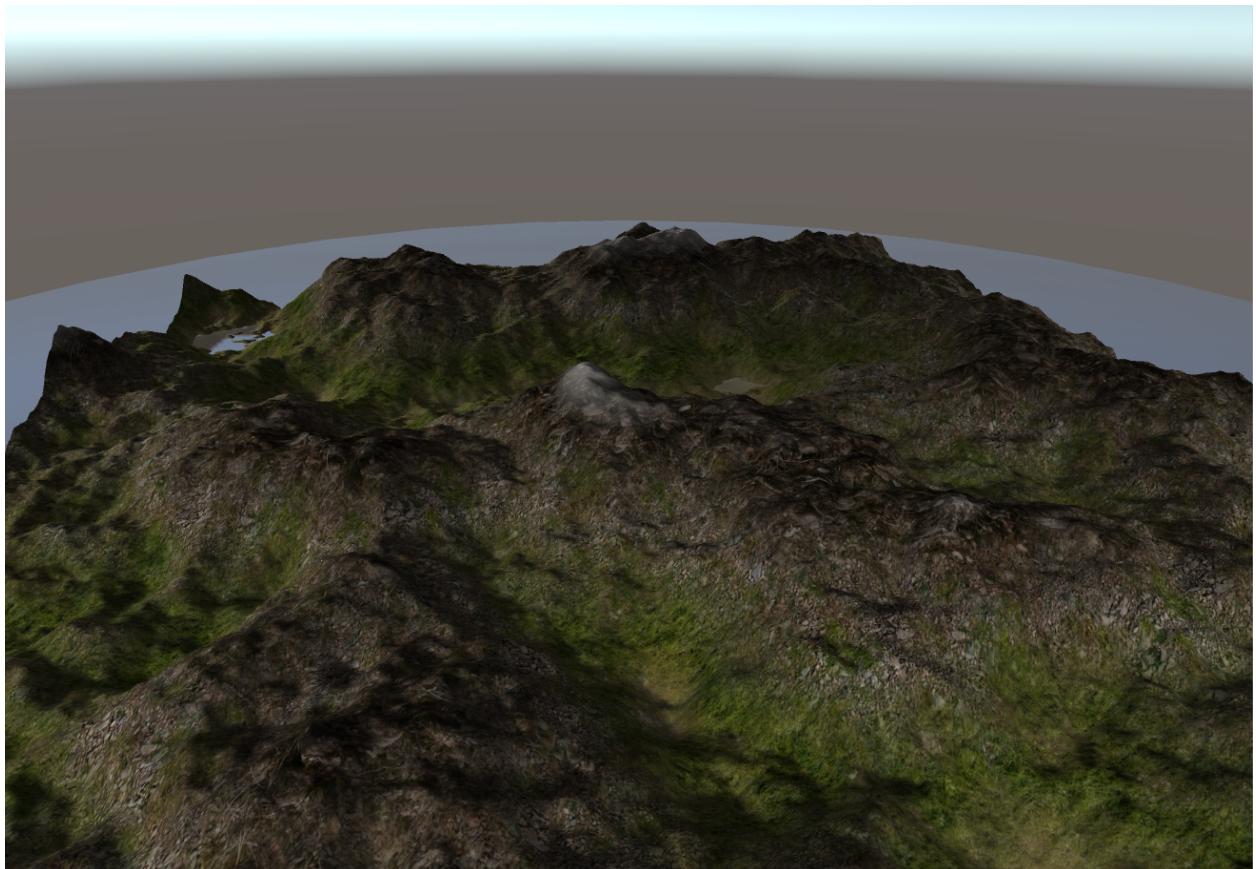


Abbildung 4.1.4: Texturierte Landschaft im Beispielprogramm die aus fraktalem Perlin-Noise entstanden ist.

4.2 Flexibilität

Die in 4.4 beschriebene Funktion lässt sich beliebig erweitern und anpassen um bestimmte Effekte zu erzielen. Im folgenden sollen daher einige bekannte und für Landschaften passende Erweiterungen erläutert werden. Diese Auflistung ist nur als eine Auswahl zu verstehen. Es gibt noch zahlreiche weitere Implementierungen wie z.B. Convolution oder auch Sparse-Convolution-Noise[DSE03].

4.2.1 Ridged-Noise

In der Natur lassen sich in einigen Gebirgen sehr steile und gezähnte Kämme feststellen. Aufgrund der kontinuierlichen Natur einer Noise-Funktion sind die Kämme jedoch eher abgerundet. Um diesen Effekt aufzuheben wird der Betrag der Noise-Funktion von 1 subtrahiert.

$$\text{ridged}(\vec{x}) = \sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} (1 - |gnoise(r^k \vec{x})|). \quad (4.5)$$

Anschaulich modelliert die Funktion nun Material aus einem Quader heraus anstatt die Landschaft auf eine Ebene herauf zu modellieren. Durch den Betrag verliert die Funktion außerdem ihre Stetigkeit in der ersten Ableitung, was zu einem abrupten Richtungswechseln der Höhenfunktion führt. An diesen Punkten entstehen nun, wie in Abbildung 4.2.1 zu sehen, die erwünschten scharfen Kämme.

4 Noise

```
1 #include "FractalNoiseBase.cginc"
2
3 /**
4  * Ridged Noise Function
5  * @param float x           X Coordinate
6  * @param float y           Y Coordinate
7  * @param int kmax          max Frequency
8  * @param float lacunarity   Lacunarity
9  * @param float h            H, fractal dimension
10 */
11 inline float RidgedNoise(float x, float y, int kmax, float lacunarity, float h) {
12
13     float currHeight = 0;
14
15     for (int k = 0; k < kmax; k++) {
16         currHeight += (1-abs(GradientNoise(x * pow(lacunarity, k),y * pow(lacunarity, k)))) / pow(lacunarity, k * h);
17     }
18
19     return currHeight;
20 }
```

Ridged-Noise Implementierung HLSL

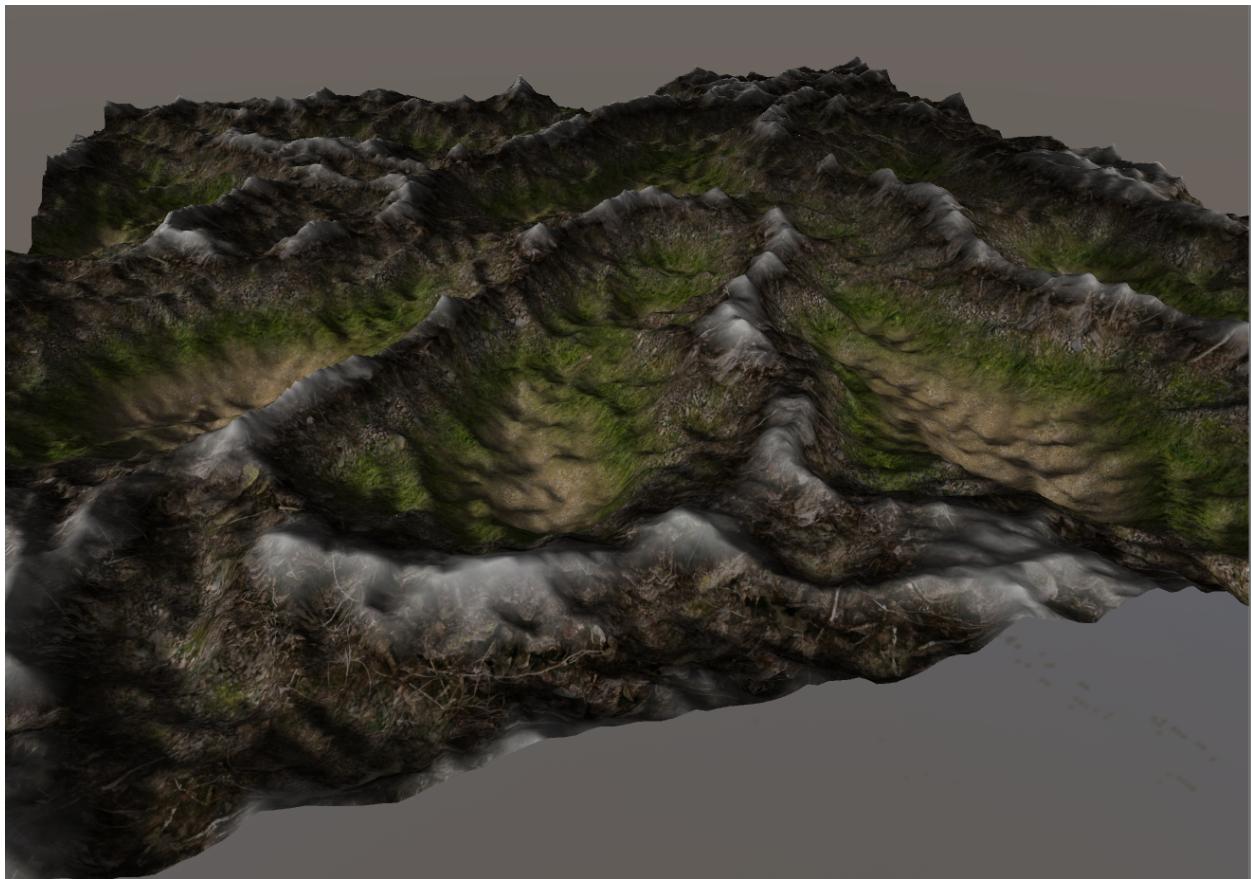


Abbildung 4.2.1: Texturierte Landschaft im Beispielprogramm die aus Ridged Noise entstanden ist

4 Noise

4.2.2 Multifraktal/heterogenes Terrain

Wie auch in 3.0.2 erwähnt kommen hohe Frequenzen momentan sowohl in den Tälern als auch auf den Bergen unserer Landschaft vor. Um diese homogenität zu verhindern wurde eine Noise Funktion entwickelt die durch Multiplikation der Oktaven zu einem Multifraktal führt⁶.

$$multi(\vec{x}) = \prod_{k=k_0}^{k_1} \frac{1}{r^{kH}} (gnoise(r^k \vec{x}) + o). \quad (4.6)$$

Es ist hierbei anzumerken, dass das Intervall des Ergebnisses dieser Rauschfunktion stark variiert. Um ein Ergebnis in einem konstanten Intervall zu verarbeiten muss im Nachhinein der höchste bzw. tiefste Punkt gesucht werden und entsprechend skaliert werden. Dadurch geht der Vorteil der parallelen Berechnung zur Echtzeit verloren. Eine andere Möglichkeit ist den Parameter o entsprechend anzupassen um das Ergebnis in eine bestimmte Richtung zu beeinflussen. In Abbildung 4.2.2 ist zu sehen, wie die tiefer gelegenen Gebiete deutlich weniger Anteile von hohen Frequenzen aufweisen und dadurch natürlicher wirken.

```

1 #include "FractalNoiseBase.cginc"
2
3 /**
4 * Ridged Noise Function
5 * @param float x           X Coordinate
6 * @param float y           Y Coordinate
7 * @param int kmax          max Frequency
8 * @param float lacunarity Lacunarity
9 * @param float h            H, fractal dimension
10 */
11 inline float RidgedMultiNoise(float x, float y, int kmax, float lacunarity, float h, float o) {
12
13     float currHeight = 1;
14
15     for (int k = 0; k < kmax; k++) {
16         currHeight *= (1-abs(GradientNoise(x * pow(lacunarity, k), y * pow(lacunarity, k)) + o)) / pow(lacunarity, k * h);
17     }
18
19     return currHeight;
20 }
```

Ridged-Multifractal Noise Implementierung HLSL

6 [DSE03] p.440

4 Noise

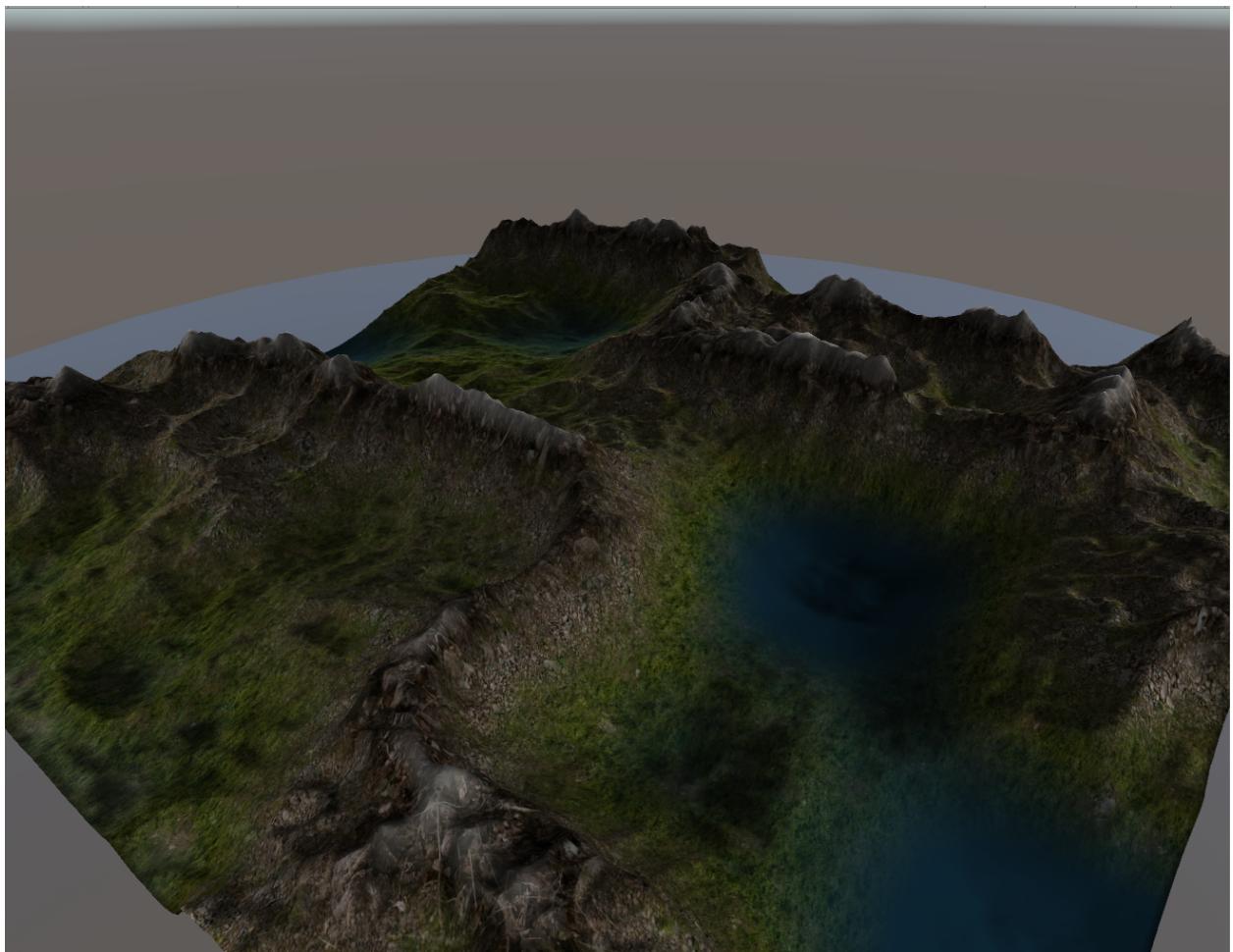


Abbildung 4.2.2: Texturierte Landschaft im Beispielprogramm die aus Ridged-Multifractal-Noise mit $o = 0.2$ entstanden ist.

4 Noise

4.2.3 Hybrid-Multifractal

Um das Problem des schwer zu bestimmenden Intervalls einer Multifraktalen Rauschfunktion zu umgehen und trotzdem ein heterogenes Ergebnis zu erzielen lassen sich die normale und die Multifraktale Rauschfunktion auch kombinieren. Die Heterogenität lässt sich dadurch erreichen, dass jede Oktave vor der Addition mit einem Gewicht multipliziert wird, welches das Produkt der vorherigen Oktaven ist. Dabei wird ein Faktor w definiert, welcher das zu schnelle abfallen des Gewichtwertes verhindert.

$$hybridnoise_{k1}(\vec{x}) = \sum_{k=k_0}^{k_1} s_k * octave_k \quad (4.7)$$

$$s_i = \prod_{l=k_0}^{k_{i-1}} octave_l * w \quad (4.8)$$

$$octave_k = \frac{1}{r^{kH}} gnoise(r^k \vec{x}) \quad (4.9)$$

In Abbildung 4.2.3 ist zu sehen, dass hohe Frequenzen, wie beim Multifraktalem Rauschen, kaum noch Einfluss in den Tälern haben.

```

1 #include "FractalNoiseBase.cginc"
2
3 /**
4  * Hybrid Noise Function
5  * @param float x           X Coordinate
6  * @param float y           Y Coordinate
7  * @param int kmax          max Frequency
8  * @param float lacunarity Lacunarity
9  * @param float h            H, fractal dimension
10 */
11 inline float HybridMultiNoise(float x, float y, int kmax, float lacunarity, float h, float o, float w) {
12
13     // calc first octave
14     float currHeight = GradientNoise(x * pow(lacunarity, k), y * pow(lacunarity, k)) + o,
15         weight = currHeight;
16
17     for (int k = 1; k < kmax; k++) {
18         // clamp weight
19         weight = min(1, weight);
20
21         float val = (GradientNoise(x * pow(lacunarity, k), y * pow(lacunarity, k)) + o) / pow(lacunarity, k * h);
22         currHeight += weight * val;
23
24         // update weight
25         weight *= val * w;
26     }
27
28     return currHeight;
29 }
```

4 Noise

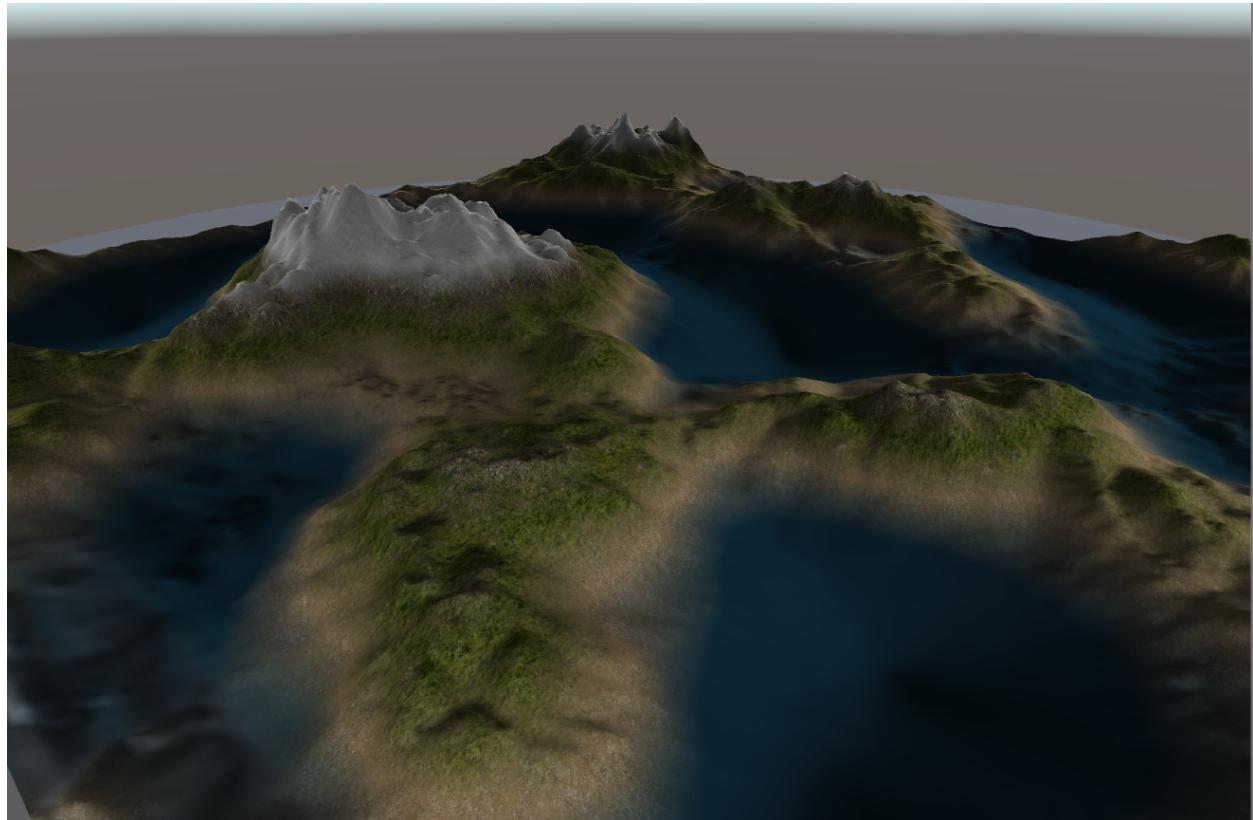


Abbildung 4.2.3: Texturierte Landschaft im Beispielprogramm die aus Hybrid-Multifractal-Noise mit $o = 0.5$ und $w = 1.15$ entstanden ist.

4.2.4 Domain/Range Mapping

Um die Rauschfunktion mit weiteren Effekten zu versehen lässt sich Domain bzw. Range Mapping anwendenden.

Beim Domain Mapping handelt es sich um eine Modifikation der Definitionsmenge⁷. So wurde etwa in Abbildung 4.2.6 der Eingangsvektor \vec{x} proportional zur Höhe rotiert.

$$rangedomain(\vec{x}) = r \left(\sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} (gnoise(d(r^k \vec{x}))) \right) \quad (4.10)$$

Range Mapping bezeichnet die nachträgliche Transformation des Ergebnisses der Rauschfunktion durch eine Funktion der Form $r(x) : [-1 - 1] \rightarrow [-1 - 1]$. Zum modellieren einer geeigneten Funktion r bietet sich die von Ken Perlin in [DSE03]⁸ vorgestellten Bias und Gain Funktion an.

Bias-Funktion

Die Bias-Funktion ist im Intervall $[0 - 1]$ definiert und bei einem Funktionswert von $x = 0.5$ seine Basis b zurück. Durch die Veränderung dieser Basis lässt sich das Krümmungsverhalten der Funktion anpassen.

$$bias_b(t) = t^{\frac{\ln(b)}{\ln(0.5)}} \quad (4.11)$$

⁷ englisch: Domain

⁸ Seiten 339-400

4 Noise

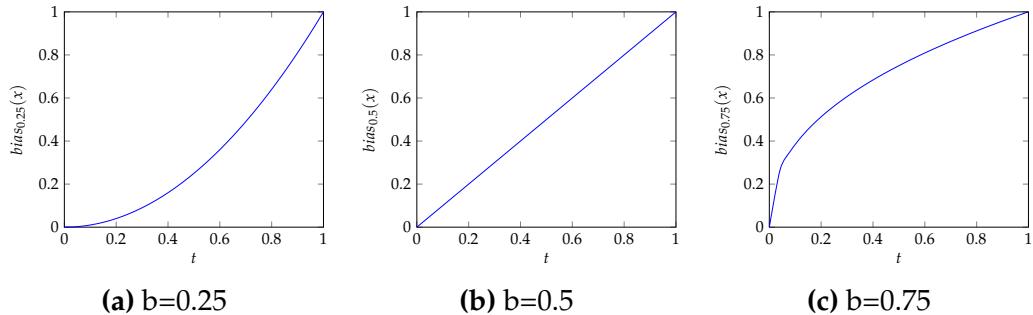


Abbildung 4.2.4: Bias-Funktion

Gain-Funktion

Im gleichen Intervall wie die Bias Funktion definiert, lässt sich mit Basis g der Gain-Funktion kontrollieren, ob die Funktionswerte sich, wie in Abbildung 4.2.5 zu sehen, die meiste Zeit im Mittel oder an den Extremwerten(0 und 1) befinden.

$$gain(t) = \begin{cases} \frac{bias_{1-g}(2t)}{2}, & \text{if } t > 0.5 \\ \frac{2-bias_{1-g}(2-2t)}{2}, & \text{if } t \leq 0.5 \end{cases}$$

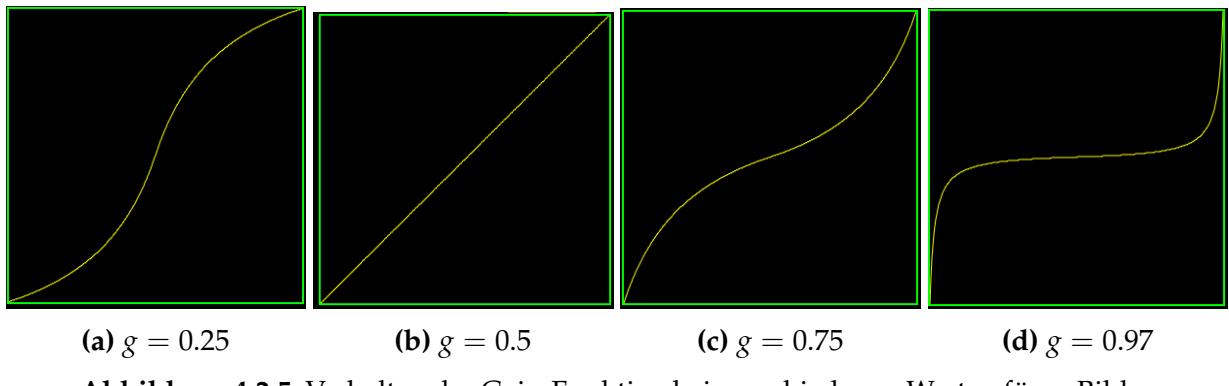


Abbildung 4.2.5: Verhalten der Gain-Funktion bei verschiedenen Werten für g . Bilder entnommen von: <http://tinyurl.com/zgb6jaj>

4 Noise

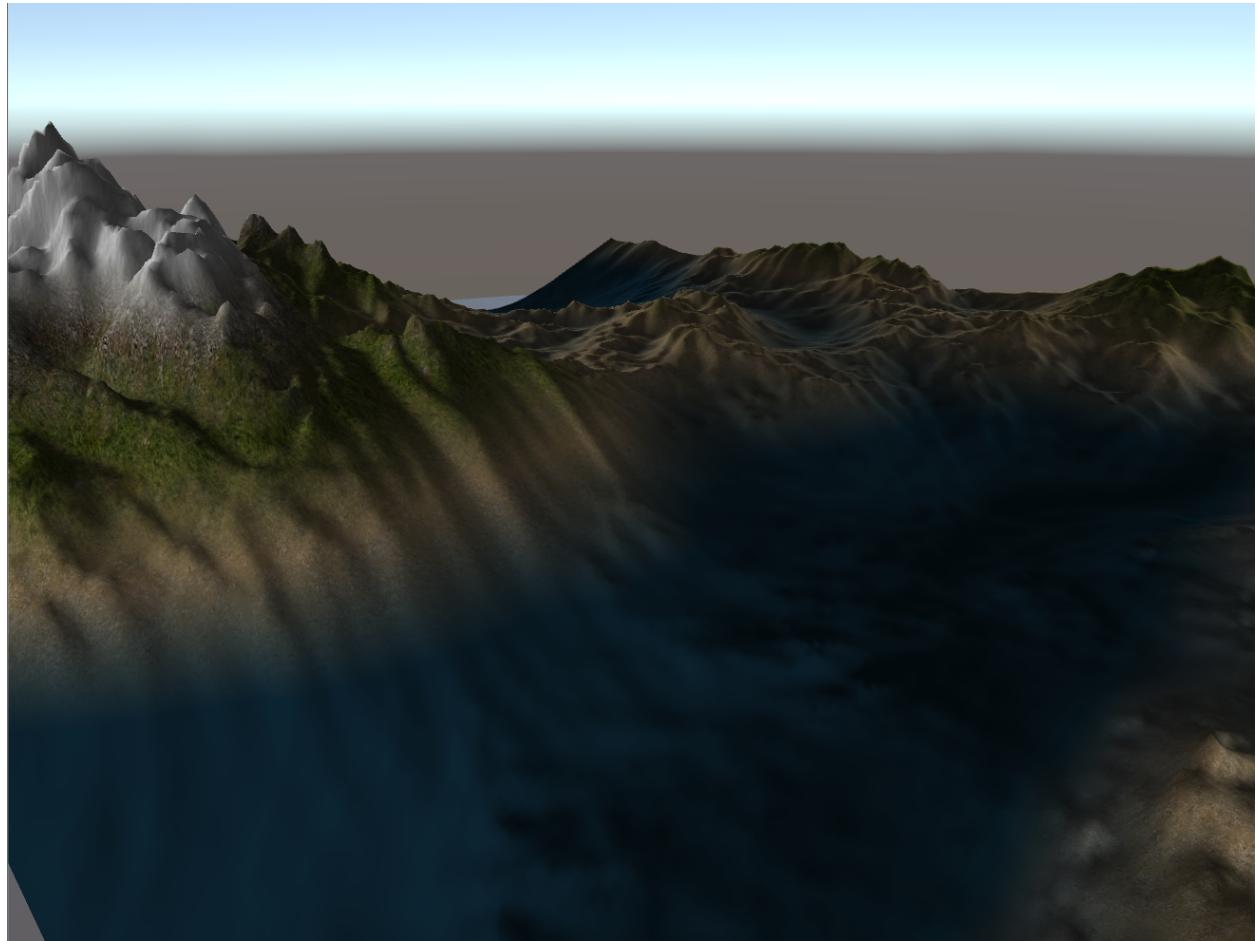


Abbildung 4.2.6: Texturierte Landschaft im Beispielprogramm die aus Domain Mapped Multifractal-Noise mit $\sigma = 0.48$ und $\alpha = 20$ entstanden ist.

4.3 Bewertung im Rahmen der Fragestellung

Das deterministische Verhalten von Rauschfunktionen erlaubt es einzig den Seed zur Erzeugung der Gitterwerte zu speichern. Durch die punktuelle Auswertung der Funktion lässt sich immer nur der Teil der Landschaft im Speicher halten, welcher gerade durch die virtuelle Kamera sichtbar ist. Eine Aufteilung des Geländes wie beim Diamond-Square Algorithmus vorgeschlagen^{2.0.3} ist nicht notwendig, wenn auch möglich um Ladezeiten zu vermeiden.

5 Fazit

Wenn es um die Erzeugung von realistisch wirkenden Höhenfeldern vor Programmstart geht ist der Diamon-Square Algorithmus die einfachste Lösung. Die Spektralsynthese ist zwar aufgrund ihrer Nutzung von beliebigen Filtern sehr flexibel, durch den Einsatz der Fourier Transformation gibt es allerdings keine Möglichkeit zur Erzeugung von heterogenem Terrain. Die Noise bzw. Rauschfunktionen zeigen ihren Nutzen besonders in ihrer einfachen Implementierung in höheren Dimension und ihrem daraus resultierenden vielseitigen Einsatzgebieten.

Für die Erzeugung von nahezu unendlich großen Landschaften eignen sich insbesondere die Rauschfunktionen. Neben dem geringen Speicherverbrauch, lässt sich die Funktion nur an der Stelle auswerten, die auch gerendert wird. Während bei den anderen Methoden die Auflösung zum Zeitpunkt der Höhenfeldsynthese feststehen muss ist es mit Rauschfunktionen möglich, die Auflösung zum Renderzeitpunkt zu bestimmen. Dadurch ist eine Veränderung des LOD¹ der Landschaft, abhängig von der Kameraposition, möglich.

¹ Level of detail - Beschreibt die Auflösung eines 3D-Modells

Literaturverzeichnis

- [AC] A.J. Crilly, R.A. Earnshaw H.: *Fractals and Chaos*
- [BMA] Brownian Motions, Fractional N.; Applications: Benoit B. Mandelbrot, John W. Van Ness.
- [Bur08] Burger, Wilhelm: Gradientenbasierte Rauschfunktionen und Perlin Noise / School of Informatics, Communications and Media, Upper Austria University of Applied Sciences. Version: November 2008. <http://staff.fh-hagenberg.at/burger/>. Hagenberg, Austria, November 2008 (HGBTR08-02). – Forschungsbericht
- [DSE03] David S. Ebert, Darwyn Peachey Ken Perlin Steven W. F. Kenton Musgrave M. F. Kenton Musgrave: *Texturing & Modeling: A Procedural Approach, Third Edition*. 2003
- [FFC82] Fournier, Alain; Fussell, Don; Carpenter, Loren: Computer Rendering of Stochastic Models. In: *Commun. ACM* 25 (1982), Juni, Nr. 6, 371–384. <http://dx.doi.org/10.1145/358523.358553>. – DOI 10.1145/358523.358553. – ISSN 0001-0782
- [fra] *Fraktale Dimension*. https://www.wikiwand.com/de/Fraktale_Dimension#/C3.84hnlichkeits-Dimension
- [Gus05] Gustavson, Stefan: *Simplex noise demystified*. <http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>. Version: 2005
- [HH] H. Hauser, E. R.: *State of the Art in Procedural Noise Functions*. <https://www-sop.inria.fr/reves/Basilic/2010/LLCDDELPZ10/LLCDDELPZ10STARPNF.pdf>

Literaturverzeichnis

- [JS06] Jens Schneider, Rudiger W. Tobias Boldte B. Tobias Boldte: *Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs.* http://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Research/Publications/2006/vmv06.pdf. Version: 2006
- [K85] In: K, PERLIN: *An image synthesizer.* vol. 19. 1985, S. 287–296
- [Per] In: Perlin, K.: *Improving noise*, S. 681–682
- [Rus] Russell, Eddie: *Elliminate Texture Confusion: Bump, Normal and Displacement Maps.* <http://blog.digitaltutors.com/bump-normal-and-displacement-maps/>
- [Sau88] Saupe, D: Point evaluation of multi-variable random fractals. In: *Visualisierung in Mathematik und Naturwissenschaft, Bremer Computergraphik Tage*, 1988
- [VOS89] VOSS, R.F.: *Random fractals: self-affinity in noise, music, mountains, and clouds.* 1989
- [whi] Weißes Rauschen - white noise. <http://www.itwissen.info/definition/lexikon/Weisses-Rauschen-white-noise.html>