Prozedurale Content-Generierung

mit dem Schwerpunkt Terrain

Bachelorarbeit im Studiengang Medieninformatik

vorgelegt von Jakob Schaal

an der Hochschule der Medien Stuttgart

am 14.02.2016

zur Erlangung des akademischen Grades eines

Bachelor of Science

Jakob.Schaal@hotmail.com

Erstprüfer:	Prof. Walter Kriha
Zweitprüfer:	Prof. Dr. Oliver Korn

Veröffentlichte ePub Version

Ehrenwörtliche/eidesstattliche Versicherung

Hiermit versichere ich, Jakob Schaal, ehrenwörtlich/eidesstattlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Prozedurale Content-Generierung mit dem Schwerpunkt Terrain“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen/eidesstattlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO, Ausgabe WS 2015/2016 (7 Semester) / § 23 Abs. 2 Bachelor-SPO, Ausgabe SS 2012 (7 Semester) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen/eidesstattlichen Versicherung zur Kenntnis genommen.

Jakob Schaal

Danksagung

Ich möchte mich bei all jenen bedanken, die mich bei meiner Bachelor-Thesis begleitet haben.

Mein besonderer Dank gilt meinen beiden Prüfern Herrn Prof. Walter Kriha und Herrn Prof. Dr. Oliver Korn, welche mir stets mit wertvollen Tipps und Feedback zur Seite standen.

Weiter möchte ich mich bei meinen Eltern Eva Schaal und Dr. Martin E. Schaal bedanken, welche mir durch ihre Unterstützung das Studium ermöglicht haben und auf die ich mich auch in schweren Zeiten immer verlassen kann.

Abstract (EN)

Manual content creation for computer games is getting more expensive and time-consuming. This thesis presents a way to limit this problem: procedural content generation. Based on a game project, a self-written algorithm is presented which generates island landscapes automatically.

In chapter 1 a brief introduction on procedural content generation is provided.

Chapter 2 discusses the current state of research and presents well known techniques, which will be used in later chapters of this thesis. Interpolations, a technique to connect discrete points, are explained. After that Value and Perlin Noise are described, two noise generators which create “cloudy pictures”. At the end of chapter 2 these noise patterns are edited.

Chapter 3 reflects on both of these noise generators. Additionally, a new algorithm which creates pictures is presented at the end of this Chapter.

In chapter 4 various techniques, shown in chapter 2 are implemented. This will illustrate how an island landscape is created in Unity. These islands are generated with volcanos, vegetation, and so on.

In chapter 5 Value and Perlin Noise are compared. It will be shown that both algorithms are not visibly distinguishable, when using these algorithms for island generation as described in this thesis. At the end of chapter 5 the performance differences and a problem with procedural content generations are explained.

In chapter 6 this thesis ends with a conclusion and an outlook on future research opportunities.

Abstract (DE)

Die manuelle Content-Erzeugung für Computerspiele wird immer teurer und zeitaufwändiger. In dieser Thesis wird eine Möglichkeit vorgestellt, dieses Problem einzuschränken – die prozedurale Content-Generierung. Anhand eines Spielebeispiels wird ein selbstgeschriebener Algorithmus vorgestellt, welcher Insellandschaften vollautomatisch generiert.

In Kapitel 1 folgt eine kurze Einführung in das Thema der prozeduralen Content-Generierung.

In Kapitel 2 wird auf den Stand der Forschung eingegangen. Es werden bekannte Techniken vorgestellt, welche im weiteren Verlauf dieser Thesis benutzt werden. Begonnen wird dabei mit der Interpolation, einer Möglichkeit, diskrete Punkte zu verbinden. Weiter werden der Value- und der Perlin-Noise genauer betrachtet, zwei Noisevarianten, welche „Wolkenbilder“ erstellen. Zum Schluss von Kapitel 2 werden diese Wolkenbilder nachträglich noch bearbeitet.

In Kapitel 3 folgen eigene Überlegungen über die beiden Arten der Noiseerstellung. Außerdem wird am Ende von Kapitel 3 ein neuer Algorithmus zur Erstellung von Bildern vorgestellt.

In Kapitel 4 werden die in Kapitel 2 vorgestellten Techniken implementiert und damit gezeigt, wie in Unity eine Insellandschaft erstellt werden kann. Diese Inseln verfügen über Vulkane, Vegetation usw.

In Kapitel 5 werden der Value- und Perlin-Noise verglichen. Es wird gezeigt, dass es zwischen den vorgestellten Noisevarianten, im konkreten Beispiel der Inselgenerierung, keinen sichtbaren Unterschied gibt. Weiter wird auf die Performance eingegangen und einige Probleme der prozeduralen Generierung werden beleuchtet.

In Kapitel 6 folgt das Fazit und eine Liste der weiteren Forschungsmöglichkeiten, die sich aus dieser Thesis ergeben.

Inhalt

1. Einführung

Computerspiele werden von sehr vielen Menschen als Freizeitmedium genutzt. Laut dem BIU (2016) (Bundesverband Interaktive Unterhaltungssoftware) wurden im ersten Halbjahr 2015 in Deutschland 863 Millionen Euro Umsatz durch Computerspiele generiert, was einem Wachstum von 8% gegenüber dem ersten Halbjahr 2014 entspricht. 2014 spielten 13,5 Millionen Deutsche täglich Computerspiele. Zahlen für 2015 sind noch nicht bekannt, jedoch ist auch hier mit einem Wachstum zu rechnen. 46% aller Deutschen spielten ab und zu Computerspiele (Golem, 2014).

Bei den Entwicklungskosten hat sich vor allem die Content-Erzeugung als teuer und zeitaufwändig herausgestellt. Ca. 55% des Gesamtbudgets gehen in die Content-Erzeugung (Irish, 2005, S. 230). Die Entwicklungen der letzten Jahre zeigen, dass sich dieser Wert vermutlich auch in Zukunft noch weiter erhöhen wird (Hendrikx u. a., 2013).

Unter Content-Erzeugung versteht man das Erstellen von Texturen, Sounds, Modellen, Levels etc. Eine Möglichkeit, den Gewinn durch Computerspiele weiter zu steigern, ist es die Kosten zu senken. Dies ist bei der Content-Erzeugung durch prozeduralen Content möglich. Unter prozeduralem Content versteht man Content, der vom Computer durch verschiede Algorithmen automatisch generiert wird. Wurde ein solcher Algorithmus gefunden, so kann dieser Algorithmus ein spezielles Problem (z.B: das Erzeugen einer Holztextur) viel schneller und kostengünstiger lösen als ein Mensch. Ein solcher Algorithmus kann also nicht jedes beliebige Problem lösen, sondern muss für jedes Teilproblem neu geschrieben werden. Einen prozeduralen Algorithmus, welcher prozedurale Algorithmen schreibt, gibt es momentan noch nicht – dies ist großer Teil der aktuellen Forschung. Dennoch kann sich dieser Ansatz zur Lösung von Problemen lohnen. Das Spiel Minecraft (Mojang Synergies AB, 2016) erzeugt zum Beispiel die komplette Spielwelt prozedural. Dies hat die Vorteile, dass die Welt theoretisch unendlich groß werden kann, der Spieler also niemals ans Ende der Welt kommt und der Spieler immer wieder für ihn neue Levelstrukturen vorfinden kann. In der Praxis gibt es bei der unendlichen Welt Probleme durch die Genauigkeit von Fließkommazahlen, wodurch der Spieler durch das ununterbrochene Laufen nach einigen Tagen doch an das Ende einer Minecraftwelt gelangen kann. Diese Limitation spielt für den normalen Spieler aber keine Rolle. Es wird dennoch das Gefühl einer endlosen Welt erzeugt.

Diese Thesis soll eine Einführung in die prozedurale Content-Generierung liefern, mit dem Schwerpunkt auf der Terrainerstellung. Es gibt viele Methoden der prozeduralen Content-Generierung. Hier wird vor allem der Value- und Perlin-Noise vorgestellt. Andere Varianten sind zum Beispiel Fraktale (Mandelbrot, 1991), das Benutzen von Kurven, um zum Beispiel Flüsse zu erzeugen (Huijser u. a., 2010), KI-Algorithmen (Roberts u. a., 2015) oder zelluläre

Automaten (Linden u. a., 2014). Auf eine Vorstellung dieser wird hier allerdings bewusst verzichtet, um den Value- und Perlin-Noise ausführlich beschreiben zu können.

Die Terrainerstellung wird sich im Rahmen dieser Thesis darauf konzentrieren, eine sowohl glaubwürdige als auch gameplaytechnisch ausgewogene Insellandschaft zu erzeugen. Andere Terrains sind ebenso erzeugbar, zum Beispiel Rennstrecken für Computerspiele (Yannakakis u. a., 2011) oder das Erzeugen von ganzen Galaxien inklusive Planetenoberflächen und Vegetation (Hello Games, 2016). In Togelius u. a. (2011) wird außerdem beschrieben, wie mit prozeduralen Methoden Mariolevel erzeugt werden können, welche mit größer werdender Spielzeit an Komplexität zunehmen.

Obwohl sich diese Thesis auf die Terrainerstellung konzentriert, können die hier vorgestellten Algorithmen auch für andere Zwecke genutzt werden. Die Noisegeneratoren eignen sich zum Beispiel auch sehr gut, um natürlich wirkende Texturen zu erstellen (Lewis, 1989) oder unebene Oberflächen zu texturieren (Lagea u. a., 2009) sowie das Modellieren von voluminösem Glas (Barnard u. a., 2005).

In Abschnitt *3.5. 4-Ray* wird ein eigener Algorithmus zur Erstellung von Bildern beschrieben. Dieser wird vorgestellt, um zu zeigen, dass es noch andere Arten der prozeduralen Content-Erzeugung gibt.

2. Stand der Forschung

2.1. Interpolation

Viele Methoden zur Noiseerzeugung basieren auf der Interpolation diskreter Werte.

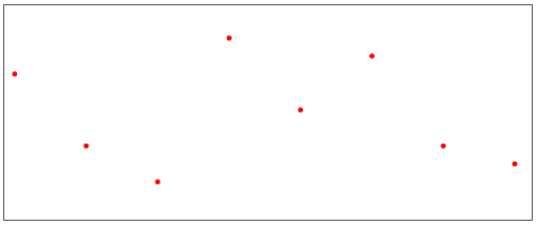

Abbildung 1 Zwischen diesen Punkten wird in den folgenden Abbildungen auf verschiedene Weisen interpoliert

Unter Interpolation (Salomon, 2006, S. 49 ff) versteht man das Ausrechnen von Werten zwischen diskreten Daten. Per Definition wird dies über eine stetige Funktion gelöst, welche durch die diskreten Punkte verläuft. Im Rahmen dieser Arbeit werden wir aber auch zwischen Punkten „interpolieren“ ohne eine stetige Funktion zu nutzen (Sprungfunktion). Streng genommen ist dies dann keine Interpolation mehr, da aber auch dabei Zwischenwerte errechnet werden, bezeichne ich auch dies als Interpolation.

Eine gute Implementierung der folgenden Interpolationen kann auf Bourke (1999) gefunden werden, außer der Sprunginterpolation, deren Implementierung allerdings trivial ist.

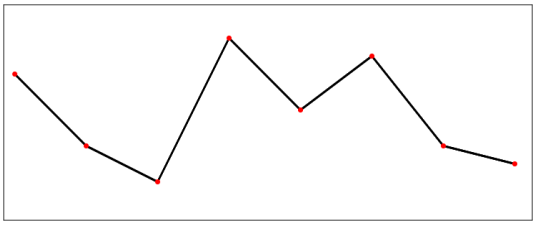

Abbildung 2 Die Punkte wurden linear interpoliert

Die einfachste Interpolation ist die lineare Interpolation. Dabei werden die zu interpolierenden Punkte einfach mit geraden Linien miteinander verbunden. Eine Eigenschaft dieser Interpolation ist, dass die Funktion an den zu interpolierenden Punkten im Allgemeinen plötzlich ihre Steigung ändert, sie also nicht differenzierbar ist. Je nach Anwendungsgebiet kann dies ein Problem darstellen.

Mathematisch kann die lineare Interpolation folgendermaßen als Formel dargestellt werden:

$$f(a, b, t) = a * (1 - t) + b * t$$

Formel 1 Lineare Interpolation

Dabei gilt:

- a und b = zwei benachbarte diskrete Werte
- t = Wert zwischen 0 und 1 (jeweils inklusive)

Wird t = 0 gesetzt, so muss die Interpolation den Wert a ergeben, bei t = 1 hingegen muss sie b ergeben. Werte zwischen 0 und 1 interpolieren zwischen a und b, in diesem Fall linear.

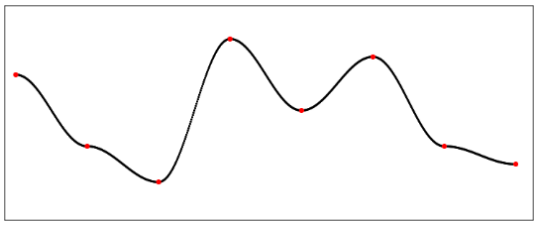

Abbildung 3 Die Punkte wurden trigonometrisch interpoliert

Ist die Differenzierbarkeit gewünscht, so bietet sich entweder die trigonometrische, kubische oder Polynominterpolation an.

Bei der trigonometrischen Interpolation wird zwischen den Punkten mithilfe der Sinus- bzw. Cosinusfunktion interpoliert. Dies wird berechnet, indem die Werte zwischen $\cos(0)$ und $cos(\pi)$ mit der Differenz zweier nebeneinanderliegender Punkte multipliziert werden. Eine interessante Eigenschaft dieser Interpolation ist, dass die Steigung an allen zu interpolierenden Punkten immer 0 ist und die Steigung zu den Punkten hin sich an 0 annähert. Dadurch ist diese Interpolation also nicht nur stetig, sondern auch differenzierbar.

Die trigonometrische Interpolation als Formel:

$$f(a,b,t) = a * \big(1 - g(t)\big) + b * g(t)$$

Formel 2 Trigonometrische Interpolation

mit:

$$g(t) = 1 - \frac{\cos(t * \pi)}{2}$$

a, b und t sind identisch definiert wie oben.

Der Wert eines beliebigen, nicht diskreten Punktes auf dieser Kurve hängt immer ausschließlich von den benachbarten diskreten Werten ab. Dies ist nicht immer gewünscht, manchmal will man, dass noch weitere Punkte die jeweiligen Werte

manipulieren. Für solche Anwendungen bietet sich die kubische Interpolation an. Dabei werden bei jeder Interpolationsberechnung nicht nur 2 Werte betrachtet, sondern noch zusätzlich deren Nachbarn, also insgesamt 4. Dadurch bildet sich eine noch sauberere Kurve, da diese je nach zusätzlichen Werten die Möglichkeit hat über- bzw. unterzuschwingen. Über- bzw. Unterschwingen bedeutet in diesem Zusammenhang, dass die Interpolation größere bzw. kleinere Werte als a und b annehmen kann.

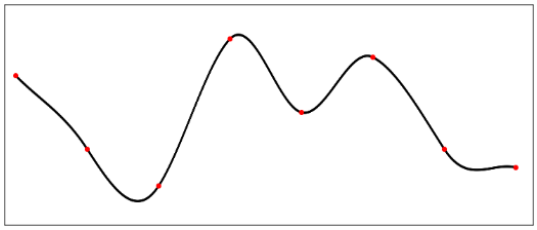

Abbildung 4 Die Punkte wurden kubisch interpoliert

Die kubische Interpolation als Formel:

$$f(preA, a, b, postB, t) = (postB - b - preA + a) * t^3 + (2 * preA - 2 * a - postB + b) * t^2 + (b - preA) * t + a$$

Formel 3 Kubische Interpolation

a, b und t sind identisch definiert wie oben. preA ist der Wert vor a und postB der Wert nach b.

Eine weitere Interpolationsmöglichkeit ist die Sprungfunktion. Es sei nochmals ausdrücklich erwähnt, dass es sich dabei streng genommen um keine Interpolation handelt, da es keine stetige Funktion darstellt. Dennoch können damit je nach Anwendungsgebiet interessante Ergebnisse erzielt werden. Jeder Punkt auf der Kurve nimmt dabei den Wert des am nächsten liegenden diskreten Wertes an. Liegt ein Punkt genau zwischen 2 diskreten Werten, so ist es abhängig von der Anwendung, welchen Wert die Kurve an diesem Punkt haben soll. In den später gezeigten Anwendungen macht es keinen sichtbaren Unterschied, welchem der beiden Punkte man den Wert zuordnet oder ob man den Durchschnitt bildet.

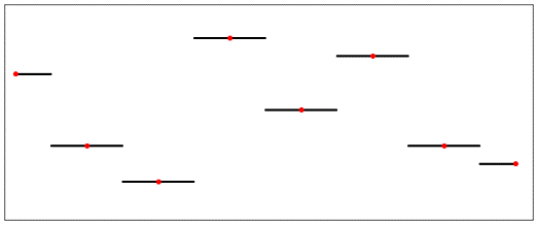

Abbildung 5 Die Punkte wurden mit einer Sprungfunktion "interpoliert"

Die Sprungfunktion als Formel:

$$f(a,b,t) = \begin{cases} a, & t < 0.5 \\ b, & t \geq 0.5 \end{cases}$$

Formel 4 Sprungfunktion

a, b und t sind identisch definiert wie oben.

Die Liste der hier genannten Interpolationsmöglichkeiten ist nicht vollständig. Im Abschnitt über Gradienten-Noises (*2.4. Gradient-Noise*) werden wir noch eine Möglichkeit sehen, wie über Polynome interpoliert werden kann. Streng genommen handelt es sich jedoch bereits bei der kubischen Interpolation um ein Polynom.

2.2. Vergleich: Value- und Gradient-Noise

Ein Großteil der Noiseerstellungsalgorithmen unterteilt sich in zwei Kategorien: Value-Noise und Gradient-Noise (Burger, 2008). Bei dieser Einteilung gibt es aber durchaus auch Ausreißer, wie zum Beispiel das sehr bekannte weiße Rauschen.

Der Unterschied zwischen diesen Noisearten besteht darin, dass bei Value-Noises zwischen diskreten Werten interpoliert wird, bei Gradient-Noises zwischen diskreten Gradienten, also Steigungen. Diese diskreten Werte bzw. diskreten Gradienten werden im Folgenden als „Stützstellen“ bezeichnet. In Abschnitt *2.3* wird der Value-Noise erklärt, in Abschnitt *2.4* ein sehr bekannter Vertreter der Gradient-Noises, der Perlin-Noise (Perlin, 1999).

2.3. Value-Noise

Die Generierung eines Value-Noise wird in verschiedene „Oktaven“ (Burger, 2008) zerlegt. In jeder Oktave werden die gleichen Berechnungsschritte vorgenommen, jedoch mit einer unterschiedlichen Anzahl an Stützstellen sowie unterschiedlichen Werten.

In der ersten Oktave werden eine Anzahl an *k* zufälligen Werten zwischen -α und α erzeugt. Dabei können *k* und α frei und anwendungsspezifisch gewählt werden. All die zufällig generierten Werte werden in einem begrenzten n-dimensionalen Raum gleichmäßig verteilt. Durch Wahl einer beliebigen Interpolationsfunktion wird allen Punkten zwischen den ursprünglichen Werten ebenfalls ein Wert zugeordnet. Wie dies genau geschieht, wird in Abschnitt *2.3.1* bis *2.3.3* genauer erklärt.

Nachdem in dieser ersten Oktave jedem Punkt im n-dimensionalen Raum ein Wert zugewiesen wurde, wird die zweite Oktave gebildet. Bei dieser Oktave wird *k* mit einem

beliebigen Wert β multipliziert (in der Literatur häufig als Persistenzwert bezeichnet) und α wird durch einen anderen ebenfalls frei wählbaren Wert μ geteilt. Häufig wird β = μ gesetzt, wobei β und μ in der Regel größer als 1 sind. Mit anderen Worten: Es wird die Anzahl der Stützstellen pro Oktave erhöht, ihr zufälliger Zahlenbereich aber verringert. Dann wird jedem Punkt im begrenzten n-dimensionalen Raum analog zur ersten Oktave ein Wert zugewiesen und auf die Werte der ersten Oktave addiert.

Dies wird mit beliebig vielen Oktaven (in der Regel einstellig, z.B. 5) wiederholt, jeweils mit den Werten *k* und α der vorherigen Oktave, multipliziert mit β und μ. β und μ ändern sich über die Oktaven hinweg nicht. Um zu verdeutlichen, zu welcher Oktave der Wert *k* und α gehört, wird im Folgenden die jeweilige Oktave als Zahl dahinter tiefgeschrieben, k_3 und α_3 gehören also jeweils zur dritten Oktave.

Als Formel ausgedrückt gilt also:

$$k_j = k_{j-1} * \beta$$

$$\alpha_j = \frac{\alpha_{j-1}}{\mu}$$

Formel 5 Werteentwicklung von k und α

mit:

- j = Nummer der Oktave

Fehlen die tiefgestellten Zahlen, so ist immer der Wert der 1. Oktave gemeint.

Der Value-Noise wurde, im Vergleich zum Perlin-Noise, bisher in wissenschaftlichen Artikeln selten benutzt. Dies liegt daran, dass für den Value-Noise in der Regel die lineare Interpolation als Interpolationsfunktion genutzt wird und die Ergebnisse dadurch schlechter werden als die Ergebnisse des Perlin-Noise. Allerdings habe ich im Rahmen meiner Arbeit mit dem Value-Noise im Zusammenhang mit anderen Interpolationsfunktionen vergleichbare Ergebnisse, jedoch mit besserer Performance erzielt als mit dem Perlin-Noise (mehr dazu in Kapitel 5).

2.3.1 Value-Noise 1D

Nachfolgend wird beispielhaft ein 1D-Value-Noise mit folgenden Werten erzeugt:

- k_1 = 8 (Anzahl Stützstellen auf 1. Oktave)
- α_1 = 1 (Zufallswertebereich)
- n = 1 (Dimension)
- $\beta = \mu = 2$ (Verdopplung der Anzahl der Stützstellen pro Oktave, Halbierung des Zufallswertebereichs)
- Anzahl Oktaven = 5
- Interpolationsfunktion: Linear

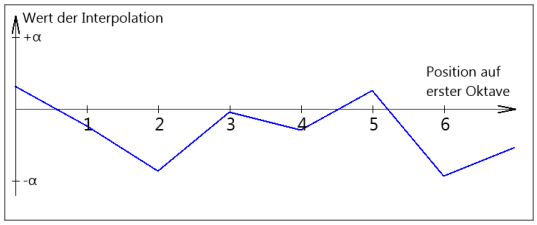

Abbildung 6 Beispielhafte erste Oktave mit den Einstellungen, die links zu sehen sind.

Nachdem die 8 Stützstellen zwischen -1 und +1 zufällig erzeugt wurden, wird zwischen ihnen abschnittsweise interpoliert, so wie in Abbildung 6 zu sehen. Wurde dies gemacht, so wird die zweite Oktave erzeugt.

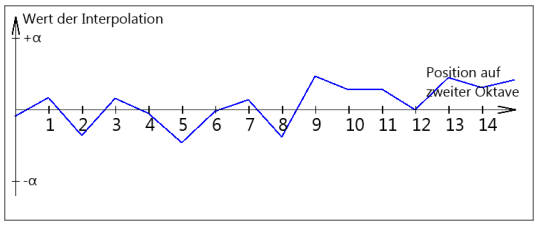

Abbildung 7 Zweite Oktave mit den Einstellungen, die links zu sehen sind.

Für die zweite Oktave gilt:

- k_2 = 16 da $k_1 * \beta = 8 * 2$
- α_2 = 0.5 da $\alpha_1 * \mu = 1 / 2$

Wie in Abbildung 7 zu sehen, hat sich der Wertebereich verringert, dafür wurden jedoch doppelt so viele Stützstellen generiert.

Für die dritte Oktave gilt:

- k_3 = 32 da $k_2 * \beta = 16 * 2$
- α_3 = 0.25 da $\alpha_2 * \mu = 0.5 / 2$

Für die vierte Oktave gilt:

- k_4 = 64 da $k_3 * \beta = 32 * 2$
- α_4 = 0.125 da $\alpha_3 * \mu = 0.25 / 2$

Für die fünfte Oktave gilt:

- k_5 = 128 da $k_4 * \beta = 64 * 2$
- α_5 = 0.0625 da $\alpha_4 * \mu = 0.125 / 2$

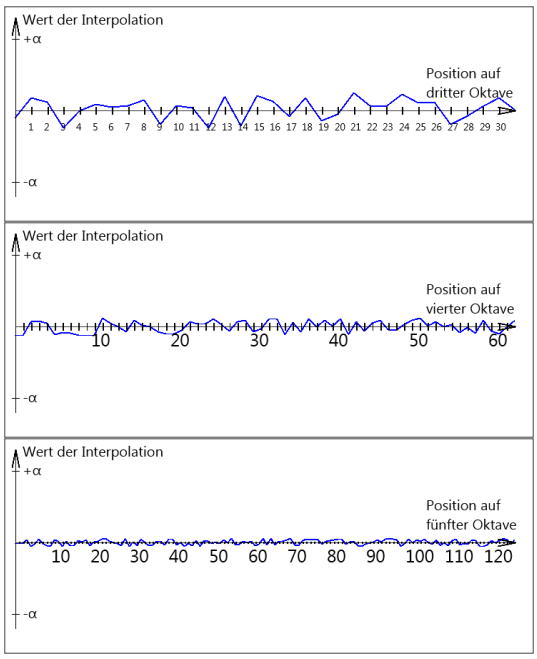

Abbildung 8 Dritte bis fünfte Oktave

Nachdem die fünf Oktaven generiert wurden, wurden die jeweiligen Ergebnisse addiert. Dadurch hat man den Value-Noise erfolgreich erzeugt.

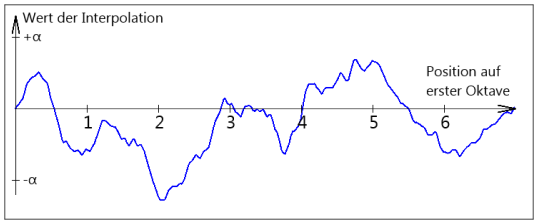

Abbildung 9 Der fertige Value-Noise: Oktaven eins bis fünf wurden addiert.

Anschaulich formuliert kann man sagen, dass die erste Oktave das Endergebnis am stärksten manipuliert und die nachfolgenden Oktaven dann immer mehr Details liefern.

Nutzt man andere Interpolationsfunktionen als die lineare Interpolation, so sehen die mathematischen Ergebnisse zunächst relativ ähnlich aus, wie in Abbildung 10 zu sehen ist. Doch obwohl die mathematischen Ergebnisse zunächst nicht signifikant unterschiedlich aussehen, so werden wir später, sobald ich zeige, wie ich den Value-Noise zur Landschaftsgenerierung benutze, sehen, dass die praktischen Ergebnisse sehr wohl signifikante Unterschiede aufweisen.

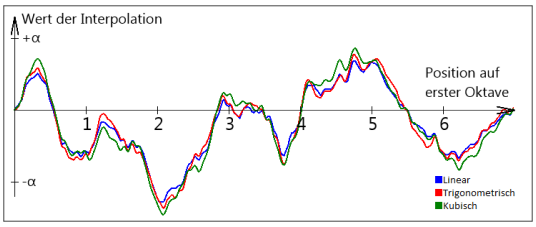

Abbildung 10 Unterschiedliche Interpolationsfunktionen im Vergleich

2.3.2 Value-Noise 2D

Für die Erzeugung von Value-Noise in mehr als einer Dimension muss erst geklärt werden wie interpoliert wird. Beim Value-Noise ist es üblich, bilinear zu interpolieren (Salomon, 2006, S.225).

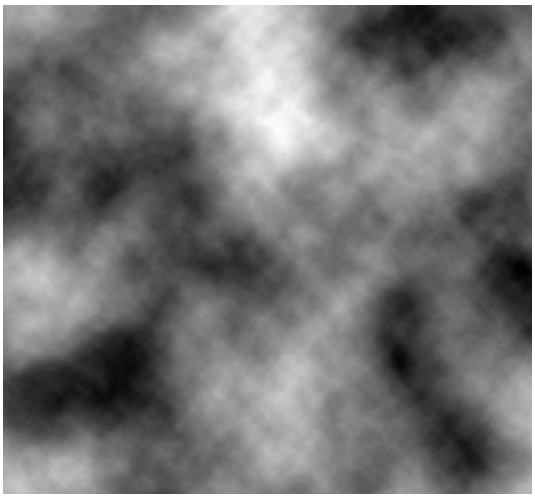

Abbildung 11 Beispielhafter 2D-Value-Noise

Bei der bilinearen Interpolation wird ein Punkt auf der begrenzten Ebene interpoliert, indem erst die 4 nächstgelegenen Stützstellen ermittelt werden. Anschließend wird erst in der Breite oder Höhe und dann in der jeweils anderen Richtung interpoliert. In Abbildung 12 wird zuerst in der Breite interpoliert. Gesucht ist letztendlich der Punkt „Interp“, dafür werden erst die Punkte AB und CD gebildet, indem zwischen A und B sowie zwischen C und D interpoliert wird und dann zwischen AB und CD. Die 2D-Interpolation wird also mit anderen Worten mit drei 1D-Interpolationen umgesetzt.

Im 2-dimensionalen Fall gibt es für *k* und β jeweils immer 2 Werte, ein Wert für jede Dimension. In diesem Abschnitt wird aber davon ausgegangen, dass beide Werte immer den gleichen Wert haben, also immer quadratische Value-Noises erzeugt werden.

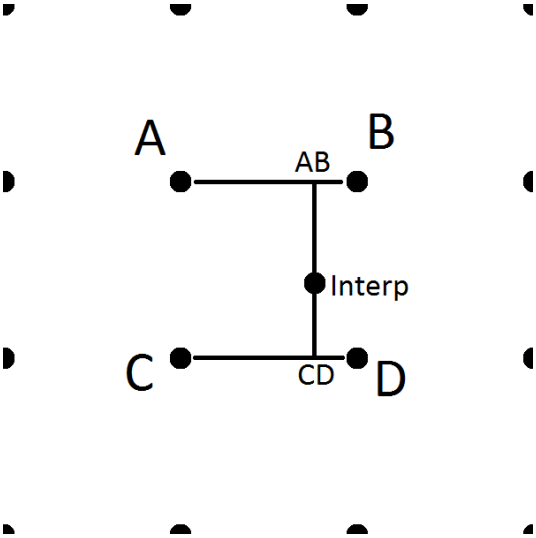

Abbildung 12 Beispiel einer bilinearen Interpolation

Alle Grafiken wurden nach Abschnitt *2.5.2 Standardisierung* standardisiert. Das bedeutet, dass die Grafiken der einzelnen Oktaven heller sind als ihre eigentlichen Werte. Im unten folgenden Beispiel wäre Oktave 6 fast nicht mehr zu sehen, weil die Werte so dicht bei 0 sind, vgl. Abbildung 15. Nach der Standardisierung befinden sich alle Werte zwischen 0 und 1. Werte gleich 1 werden weiß dargestellt, Werte mit 0 schwarz und Werte dazwischen bekommen entsprechende Graustufen.

Nachfolgend wird beispielhaft ein 2D-Value-Noise mit folgenden Werten erzeugt:

- k_1 = 5 (Anzahl Stützstellen auf 1. Oktave)
- α_1 = 1 (Zufallswertebereich)
- n = 1 (Dimension)
- $\beta = \mu = 2$ (Verdopplung der Anzahl der Stützstellen pro Oktave, Halbierung des Zufallswertebereichs)
- Anzahl Oktaven = 6
- Interpolationsfunktion: Linear

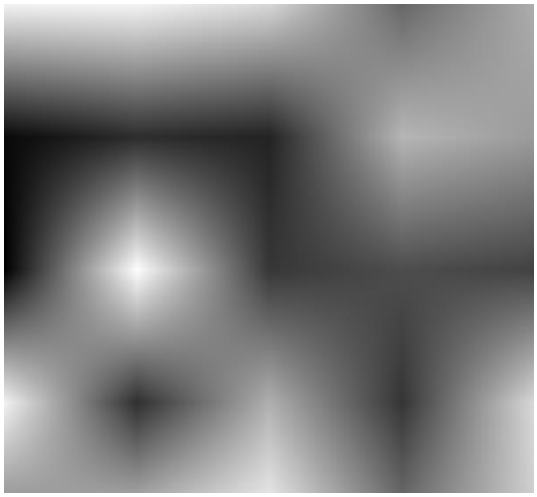

Abbildung 13 Die erste Oktave eines beispielhaften 2D-Value-Noise mit linearer Interpolationsfunktion

Für die zweite Oktave gilt:

- k_2 = 10 da $k_1 * \beta = 5 * 2$
- $\alpha_2 = 0.5$ da $\alpha_1 * \mu = 1 / 2$

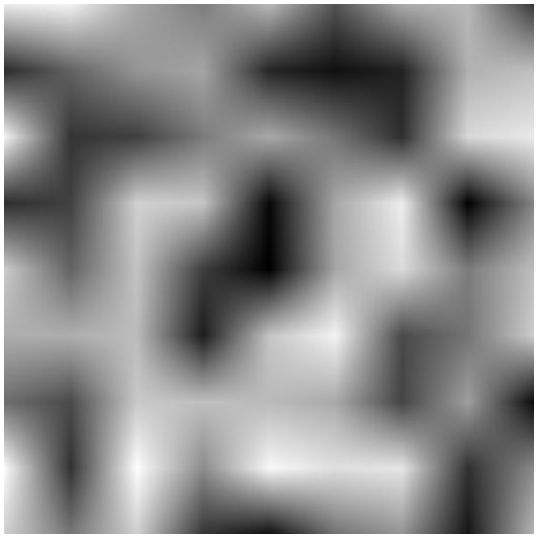

Abbildung 14 Die zweite Oktave

Für die dritte Oktave gilt:

- k_3 = 20 da $k_2 * \beta = 10 * 2$
- $\alpha_3 = 0.25$ da $\alpha_2 * \mu = 0.5 / 2$

Für die vierte Oktave gilt:

- k_4 = 40 da $k_3 * \beta = 20 * 2$
- $\alpha_4 = 0.125$ da $\alpha_3 * \mu = 0.25 / 2$

Für die fünfte Oktave gilt:

- k_5 = 80 da $k_4 * \beta = 40 * 2$
- $\alpha_5 = 0.0625$ da $\alpha_4 * \mu = 0.125 / 2$

Für die sechste Oktave gilt:

- k_6 = 160 da $k_5 * \beta = 80 * 2$
- $\alpha_6 = 0.03125$ da $\alpha_5 * \mu = 0.0625 / 2$

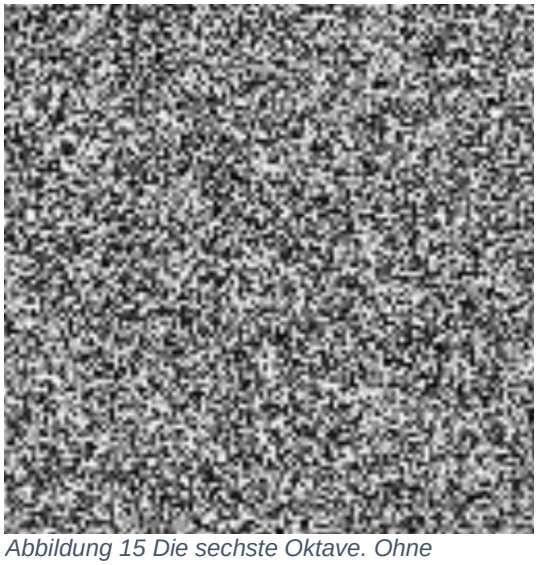

Abbildung 15 Die sechste Oktave. Ohne Standardisierung wäre sie hier kaum sichtbar, da viel zu dunkel.

Addiert man die 6 Oktaven nun analog zum 1D-Fall erhält man das fertige Value-Noise-Bild. Bei diesem sind aber noch deutlich lineare Artefakte sichtbar, wie in Abbildung 16 zu sehen.

Dies lässt sich beheben, indem man entweder die trigonometrische oder die kubische Interpolation statt der linearen zur Berechnung benutzt.

Mit nur einer Oktave liefert die kubische Interpolation bereits deutlich bessere Ergebnisse, wie in Abbildung 17 zu sehen. Die Übergänge sind weicher und wirken deutlich sauberer als beim linearen Ansatz.

Wenn man die Ergebnisse von Abbildung 16 mit den Ergebnissen von Abbildung 18 vergleicht, so fällt deutlich auf, dass die linearen Artefakte, wie beispielsweise in der rechten oberen Ecke, beim helleren Fleck verschwunden sind.

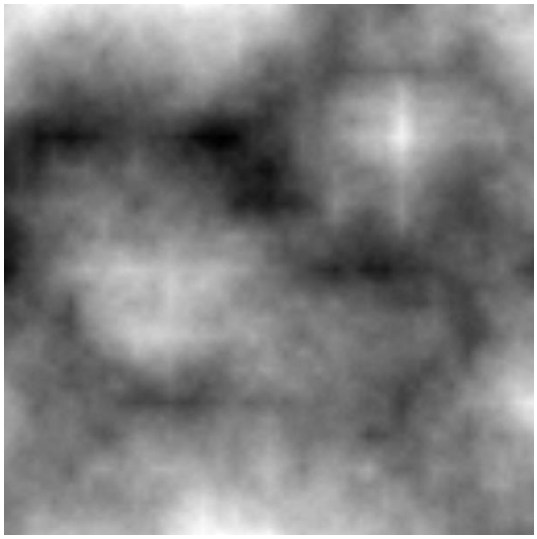

Abbildung 16 Ein 2D-Value-Noise-Bild mit linearer Interpolation

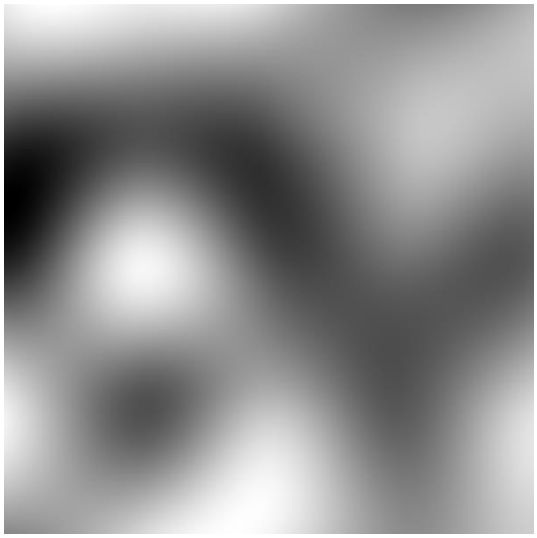

Abbildung 17 Erste Oktave eines Value-Noise mit kubischer Interpolation

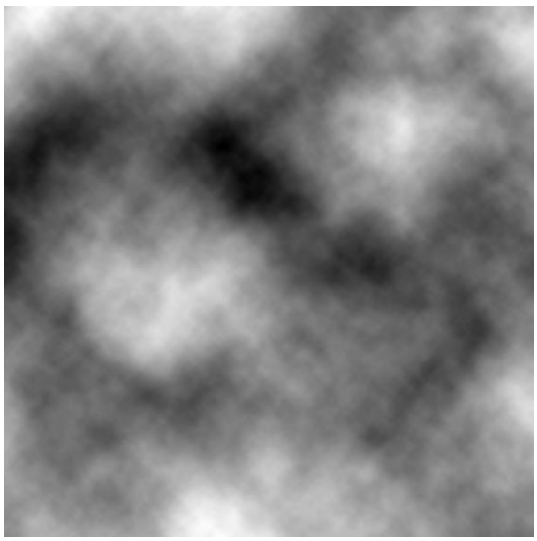

Abbildung 18 Ein Value-Noise mit sechs Oktaven und kubischer Interpolation

Die Ergebnisse der trigonometrischen Interpolation sind mathematisch nicht signifikant unterschiedlich, jedoch werden mit ihr deutlich sichtbar andere Landschaften generiert als mit der kubischen Interpolation.

Erhöht man die Anzahl der diskreten Werte auf der ersten Oktave, also *k*, so wird das Bild deutlich „hügeliger“. Das bedeutet, dass es mehr Schwarz-Weiß-Unterschiede gibt, wie in Abbildung 19 zu sehen

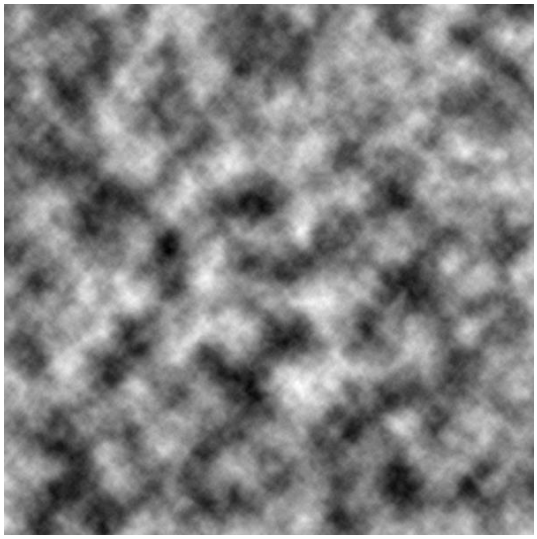

Abbildung 19 Höherer k-Wert. In diesem Bild wurde k = 16 gesetzt.

2.3.3 Value-Noise ND

Ein Value-Noise kann in beliebig vielen, ganzzahligen, positiven Dimensionen berechnet werden. Dabei wird das Problem immer auf N-1 Dimensionen verringert, bis man im 1-dimensionalen Fall angekommen ist. Im Fall von 3 Dimensionen entspricht dies der trilinearen Interpolation (Wagner, n.d.). Dabei wird das Problem erst auf 2 Dimensionen reduziert und dann wie in *2.3.2 Value-Noise 2D* gelöst. Dadurch ergeben sich nach Abbildung 20 die 2 Punkte ABCD und EFGH, zwischen denen anschließend noch ein letztes Mal 1-dimensional interpoliert wird, um den Punkt Interp zu berechnen. Im 4-dimensionalen Fall kann man sich vorstellen, dass das Problem auf zwei 3D-Würfel reduziert wird, in denen wie eben beschrieben das Ergebnis errechnet und anschließend interpoliert wird.

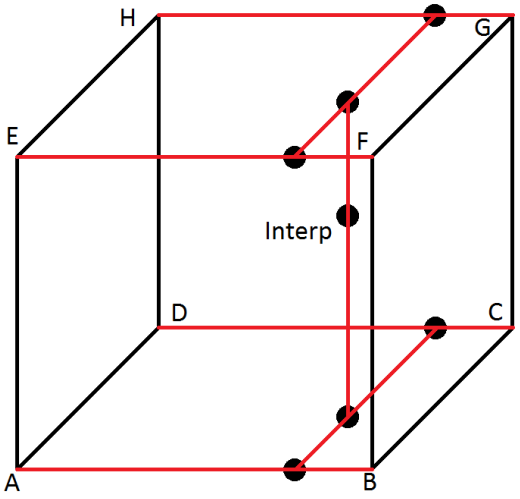

Abbildung 20 Darstellung der trilinearen Interpolation

Dieser Schritt kann für beliebig viele Dimensionen fortgeführt werden.

Einen 3D-Value-Noise kann man z.B. benutzen, um Wasseroberflächen zu animieren. Dabei wird die dritte Dimension als Zeit angesehen und die Wasseroberfläche wird als Ebene durch den 3D-Würfel „geschoben“.

Die Anzahl der 1D-Interpolationen ist:

$$f(n) = 2^n - 1$$

Formel 6 Anzahl der Interpolationen im n-dimensionalen Raum

mit:

- n = Anzahl Dimensionen

Der Beweis hierzu folgt in *3.4. Beweis von 2.3.3 Value-Noise ND.*

2.4. Gradient-Noise

Bei Gradient-Noises wird an den Stützstellen kein diskreter Wert, sondern eine diskrete Steigung (Gradienten) erzeugt, womit die Werte der restlichen Kurve berechnet werden können (Ebert u. a., 2003). In der Regel ist der Wert an den einzelnen Stützstellen auf 0 festgelegt.

Ein sehr bekannter Vertreter der Gradient-Noises ist der Perlin-Noise (Perlin, 1999), benannt nach seinem Erfinder Ken Perlin. Bei diesem wird über Polynome interpoliert. Die Funktionsweise des Perlin-Noise ist ähnlich dem des in *2.3. Value-Noise* vorgestellten Noise. Er wird ebenso in Oktaven unterteilt, die anschließend summiert werden. Unterschiede sind jedoch, dass die Interpolationsfunktion anders funktioniert und an den Stützstellen keine zufälligen Werte, sondern Steigungen erzeugt werden.

Eine Weiterentwicklung des Perlin-Noise ist der Simplex-Noise (Perlin, 2001). Dieser ist in höheren Dimensionen performanter als der Perlin-Noise. Da diese höheren Dimensionen im Rahmen meiner Arbeit aber nicht nötig waren, habe ich auf diesen verzichtet. Daher wird hier nicht weiter auf den Simplex-Noise eingegangen.

2.4.1 Perlin-Noise 1D

Beim Perlin-Noise wird über Polynome interpoliert. Die Interpolationsfunktion f nimmt dabei 3 Zahlen entgegen und gibt den Wert der Funktion zurück (nicht die Steigung, sondern den Wert).

Also muss die Funktion folgende Form haben:

$$f(a, b, t)$$

mit:

- a und b = zwei benachbarte Stützstellen

- t = Prozentwert zwischen a und b, siehe *2.1. Interpolation*

Weiter sind folgende Eigenschaften dieses Polynoms bekannt:

- $f(a,b,0) = 0$, da der Wert an den Stützstellen = 0 ist
- $f(a,b,1) = 0$, da der Wert an den Stützstellen = 0 ist
- $f'(a,b,0) = a$
- $f'(a,b,1) = b$

Dabei beschreibt f' die erste Ableitung von f nach t.

Um diese Eigenschaften in jedem Fall zu erfüllen, wird mindestens ein Polynom dritten Grades benötigt (Burger, 2008). Perlin hat stattdessen ein Polynom vierten Grades gewählt, also der Form:

$$f(a,b,t) = w_4 * t^4 + w_3 * t^3 + w_2 * t^2 + w_1 * t^1 + w_0$$

Formel 7 Perlins ursprüngliche Interpolationsform

wobei die Gewichte w_0 bis w_4 folgende Werte haben:

$$\begin{aligned} w_0 &= 0 \\ w_1 &= a \\ w_3 &= -2 * w_2 - 3 * a - b \\ w_4 &= w_2 + 2 * a + b \end{aligned}$$

w_2 kann frei gewählt werden. Burger (2008) wählt z.B. den Wert $w_2 = -3 * b$.

Eingesetzt in die Funktion oben ergibt sich damit:

$$f(a,b,t) = 2 * (a-b) * t^4 - (3a - 5b) * t^3 - 3b * t^2 + a * t$$

Formel 8 Ursprüngliche Perlin-Noise-Interpolation

Wie in Burger (2008) angemerkt, hat diese Funktion aber die Eigenschaft, dass sich die Krümmung, also die zweite Ableitung nach t, f'', an den Stützstellen plötzlich ändert, also die Charakteristik einer Sprungfunktion aufweist. Ist dies nicht gewünscht, so muss ein anderes Polynom gewählt werden. In Burger (2008) kann dazu mehr nachgelesen werden. Da es bei meinen Tests allerdings keine sichtbaren Unterschiede gab, verzichte ich hier darauf, diesen Ansatz weiter zu erläutern.

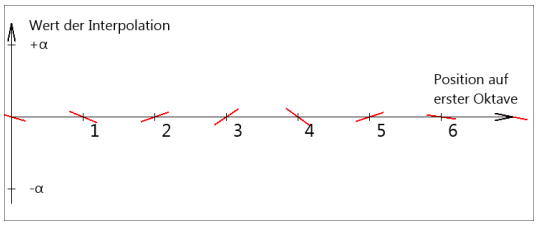

Abbildung 22 Zufällige Steigungen an den Stützstellen für Perlin-Noise

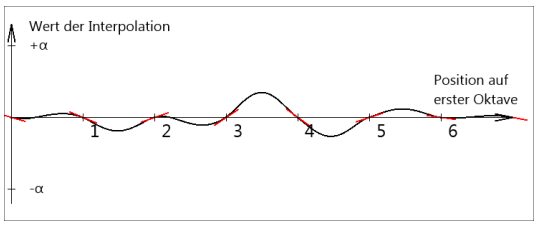

Abbildung 21 Interpolationsfunktion durch die Stützstellen mit den angezeigten Steigungen

In der ersten Oktave werden an den Stützstellen zunächst die Steigungen zwischen +α und –α zufällig gewählt. Diese Steigungen wurden in Abbildung 22 als rote Linien eingezeichnet. Die gewünschte Interpolation soll also durch die Stützstellen laufen (mit anderen Worten dort ihre Nullstellen haben) und dort die angezeigten Steigungen besitzen. Die folgenden Variablen-bezeichnungen sind identisch zu den in *2.3. Value-Noise* vorgestellten.

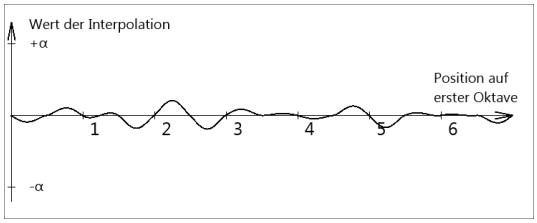

Abbildung 23 Die zweite Oktave des beispielhaften Perlin-Noise

In diesem Beispiel gilt:

- $k = 7$
- $\alpha = 1$
- $n = 1$
- $\beta = \mu = 2$
- Anzahl Oktaven = 5

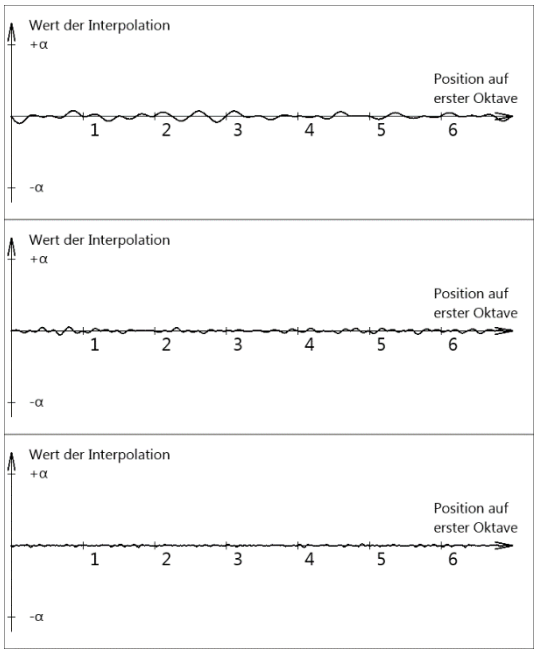

Abbildung 24 Oktaven drei bis fünf des Perlin-Noise-Beispiels

Wie in Abbildung 23 zu sehen, hat sich der Wertebereich halbiert und es gibt nun doppelt so viele Stützstellen. Dadurch gibt es nun auch (im Schnitt) doppelt so viele Nullstellen (Nullstellen können auch an Nicht-Stützstellen auftreten), jeweils immer noch an den Positionen x,5 - also 0,5; 1,5; 2,5 etc.

Dies wiederholt sich bei den Oktaven drei bis fünf, wie in Abbildung 24 zu sehen. Der Wertebereich wird immer kleiner und es gibt immer mehr Stützstellen. Addiert man nun die Werte von Oktave eins bis fünf, so hat man das 1D-Perlin-Noise aus Abbildung 25 erzeugt.

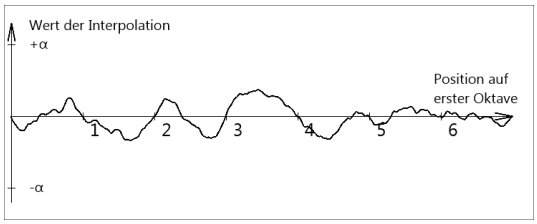

Abbildung 25 Das fertige Perlin-Noise-Beispiel

Wie in Burger (2008) beschrieben, kann Formel 8 auch als Splineinterpolation interpretiert werden. Dabei hat die Interpolation an den Stellen t=0 und t=1 jeweils die Tangenten:

$$h0(t) = a * t$$
$$h1(t) = b * (t - 1)$$

Damit lässt sich die Gleichung umstellen zu:

$$f(a, b, t) = h0(t) + s(t) * (h1(t) - h0(t))$$

Formel 9 Modifizierte Perlin-Noise-Interpolationsfunktion

Dabei kann die Überblendungsfunktion s(t) frei gewählt werden. Um die ursprüngliche Gleichung nicht zu verändern, muss sie zur kubischen Überblendungsfunktion

$$s(t) = -2t^3 + 3t^2$$

Formel 10 Die in dieser Thesis verwendete Überblendungsfunktion

gesetzt werden. Diese Darstellung wird für Dimensionen > 1 hilfreich.

2.4.2 Perlin-Noise 2D

Beim Value-Noise hat es auch im höherdimensionalen Fall gereicht, pro Stützstelle immer nur einen zufälligen Wert zu generieren. Bei Gradient-Noises ist dies nicht der Fall. Da die Steigung auch eine Richtung hat, muss ein Vektor mit n-Elementen gebildet werden, wobei n die Anzahl der Dimensionen ist.

Das bedeutet, dass im 2-dimensionalen Fall für jede Stützstelle ein Vektor folgender Form zufällig gewählt wird:

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

wobei x und y zwischen –α und +α liegen. Jedoch könnte man für x und y auch andere Wertebereiche nehmen, also 2 unterschiedliche α-Werte. Dies wird in meiner Arbeit allerdings nicht weiter untersucht, daher wird davon ausgegangen, dass α in allen Dimensionen identisch ist.

Der Wert t muss nun aber für beide Dimensionen getrennt betrachtet werden. tx sei dabei der Prozentwert in x-Richtung zwischen 0 und 1 und ty der Prozentwert in y-Richtung, jeweils analog zu den bisher beschriebenen Fällen.

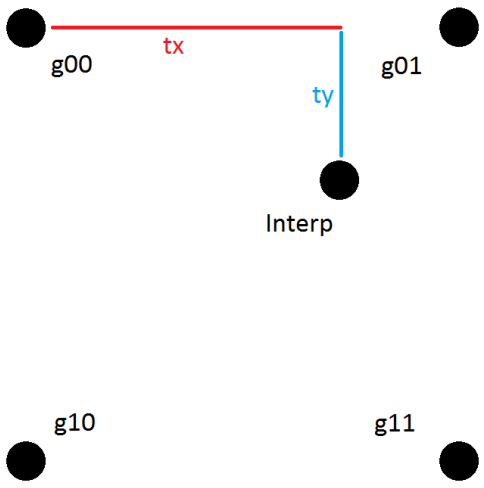

Abbildung 26 Perlin-Noise im 2-dimensionalen Fall, Punktbezeichnungen und tx, ty

Um die Interpolation beginnen zu können, müssen für jeden zu interpolierenden Punkt erst die 4 am nächsten liegenden Punkte bestimmt werden. Im Folgenden werden diese Punkte als g00 bis g11 bezeichnet, so wie in Abbildung 26 zu sehen.

Wurden diese bestimmt, so werden die Hilfswerte w00 bis w11 über folgende Formeln bestimmt:

$$w00 = g00 * \left(\begin{pmatrix} tx \\ ty \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)$$

$$w10 = g10 * \left(\begin{pmatrix} tx \\ ty \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)$$

$$w01 = g01 * \left(\begin{pmatrix} tx \\ ty \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)$$

$$w11 = g11 * \left(\begin{pmatrix} tx \\ ty \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right)$$

Formel 11 Gewichte der Stützstellen im 2D-Perlin-Noise

wobei * das Skalarprodukt bezeichnet.

Nun wird, genau wie bereits beim Value-Noise gezeigt, bilinear interpoliert. Dies funktioniert über folgende Formeln:

$$w0 = \big(1 - s(tx)\big) * w00 + s(tx) * w10$$
$$w1 = \big(1 - s(tx)\big) * w01 + s(tx) * w11$$

und:

$$w = \big(1 - s(ty)\big) * w0 + s(ty) * w1$$

mit:

- w = Ergebnis des Perlin-Noise an der gesuchten Stelle
- s(t) = kubische Interpolationsfunktion Formel 10 aus *2.4.1 Perlin-Noise 1D*

Alle in diesem Abschnitt nun folgenden Abbildungen wurden wie in *2.5.2 Standardisierung* gezeigt, bearbeitet, um sie deutlicher zu machen.

Nachdem die erste Oktave erzeugt wurde, wurden nun analog zum Value-Noise die restlichen Oktaven erzeugt und dann addiert. Ein ausführliches Beispiel spare ich mir an dieser Stelle, da es sich bei der Werteentwicklung von α und *k* gleich verhält wie in *2.3.2 Value-Noise 2D*.

Ein fertiges Perlin-Noise-Beispiel ist in Abbildung 27 zu sehen.

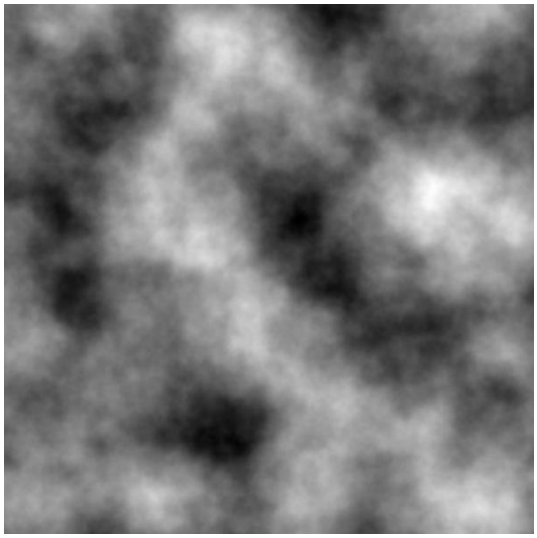

Abbildung 27 Fertiges 2D-Perlin-Noise-Beispiel

2.4.3 Perlin-Noise ND

Die Generierung von einem n-dimensionalen Perlin-Noise verläuft analog zu *2.3.3 Value-Noise ND*, jedoch müssen die Gewichte w0…0 bis w1…1 noch bestimmt werden. Jede der Zahlen 0/1 beschreibt dabei eine Achse im n-dimensionalen Raum. Im 3-dimensionalen Raum gilt also: wxyz, wobei die erste Zahl die x-Achse beschreibt, die zweite die y-Achse und die dritte die z-Achse. w101 wäre dann also auf der x- und z-Achse um 1 verschoben, auf der y-Achse nicht verschoben. Entsprechend verhalten sich die Gradienten g0…0 bis g1…1.

Im Allgemeinen gilt:

$$wx1x2 \dots xn = gx1x2 \dots xn * \left(\begin{pmatrix} tx1 \\ tx2 \\ \vdots \\ txn \end{pmatrix} - \begin{pmatrix} x1 \\ x2 \\ \vdots \\ xn \end{pmatrix} \right)$$

Formel 12 Gewichte der Stützstellen im ND-Perlin-Noise

Wurden diese Gewichte gebildet, kann nun, wie bereits erklärt, Schritt für Schritt eine Dimension nach der anderen eliminiert werden, bis man im 1-dimensionalen Fall ankommt und das Perlin-Noise-Ergebnis an der jeweiligen Stelle errechnen kann.

2.4.4 Blockartefakte

Beim Perlin-Noise können sogenannte Blockartefakte (Tatarchuk, n.d.) auftreten. Diese sind in einem extremen Fall in Abbildung 28 zu sehen. Blockartefakte treten auf, wenn benachbarte Gradienten das gleiche Vorzeichen besitzen, also beide positiv oder beide negativ sind.

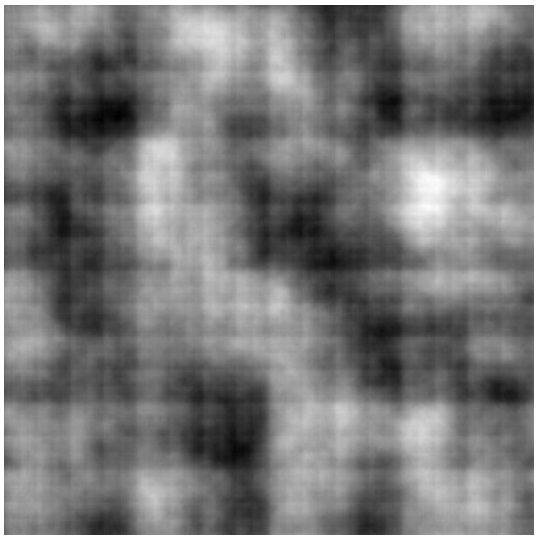

Abbildung 28 Ein extremer Fall von Blockartefakten. Alle Gradienten sind bei diesem Perlin-Noise positiv.

In der Regel ist dies kein wünschenswerter Effekt. Beim Value-Noise kann dies nicht auftreten (zumindest nicht, wenn die in *2.1. Interpolation*en gezeigten Funktionen verwendet werden).

Will man diesen Effekt verhindern, so kann man den Perlin-Noise mehrfach generieren und anschließend den Durchschnitt bilden, welcher dann wieder standardisiert wird (siehe *2.5.2 Standardisierung*). Dies steigert jedoch den Performanceaufwand drastisch.

2.5 Nachträgliche Noisebearbeitung

Wurde ein Noise generiert, so kann man diesen nachträglich noch manipulieren und so für den gewünschten Anwendungsfall besser verwendbar machen. Im Folgenden werden die beiden bekanntesten nachträglichen Bearbeitungen vorgestellt. Diese Liste ist aber keinesfalls vollständig – auf einen Noise lässt sich jede beliebige mathematische Operation anwenden, wodurch sein Verhalten teilweise stark manipuliert wird. In Michelon de Carli u. a. (2014) werden zum Beispiel verschiedene Techniken vorgestellt, um Noises so aufzubereiten, dass sie für die Erzeugung von 3D-Canyons benutzt werden können.

2.5.1 Smoothing

Beim Smoothing (Kuan u. a., 2009) wird auf den Noise der Gaußsche Weichzeichner (Beschorner u. a., 2016, S. 178) angewendet, um ihn auf diese Weise weicher zu machen. Beim Gaußschen Weichzeichner wird für jeden Pixel (im 2D-Fall) ein gewichteter Durchschnittswert seiner Nachbarn gebildet und dann dieser Wert dem Pixel neu zugewiesen.

Es können auch andere Weichzeichner-Algorithmen als der Gaußsche Weichzeichner verwendet werden, jedoch erzeugte der Gaußsche Weichzeichner bei Tests die besten Ergebnisse.

In der Regel lässt sich ein Smoothing aber auch performanter mit weniger Oktaven realisieren.

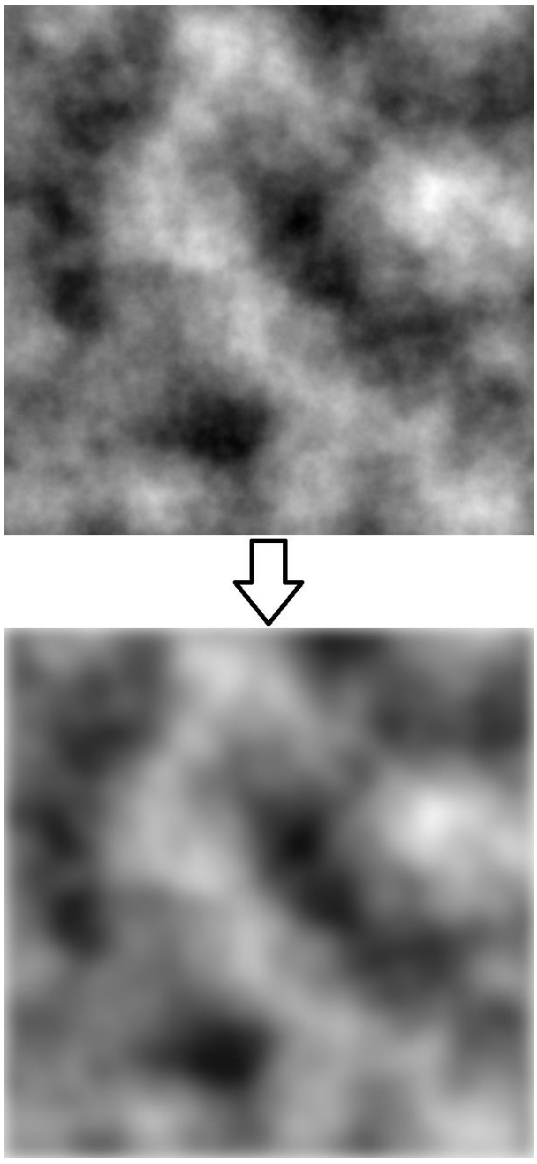

*Abbildung 29 Smoothing-Beispiel für Perlin-Noise. Verwendet wurde der Gaußsche Weichzeichner mit Range 30 Pixel (das Orginal ist 600*600 Pixel groß).*

2.5.2 Standardisierung

Alle Bilder von 2D-Noises in dieser Thesis wurden, wenn nicht anders angegeben, nachträglich standardisiert, um die Erkennbarkeit zu erhöhen.

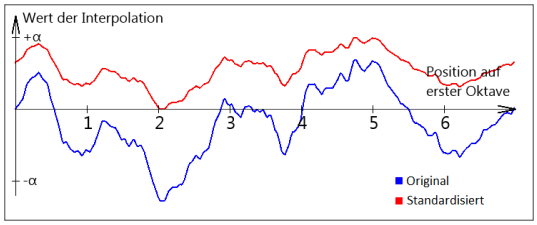

Abbildung 30 Beispiel einer 1D-Value-Noise-Standardisierung. Der Wert α ist in diesem Noise gleich eins.

Bei der Standardisierung wird der Wert jedes Pixels (im 2D-Fall) zwischen 2 Werte gebracht, in der Regel zwischen 1 und 0 oder zwischen 1 und -1. Im Folgenden wird von dem Range 1 bis 0 ausgegangen. Dabei hat der höchste Wert des ursprünglichen Noise nach der Standardisierung den Wert 1, der niedrigste den Wert 0. Alle Verhältnisse zwischen 2 Werten des Noise bleiben nach der Standardisierung unverändert. Ziel der Standardisierung ist es, den Noise besser kontrollieren zu können.

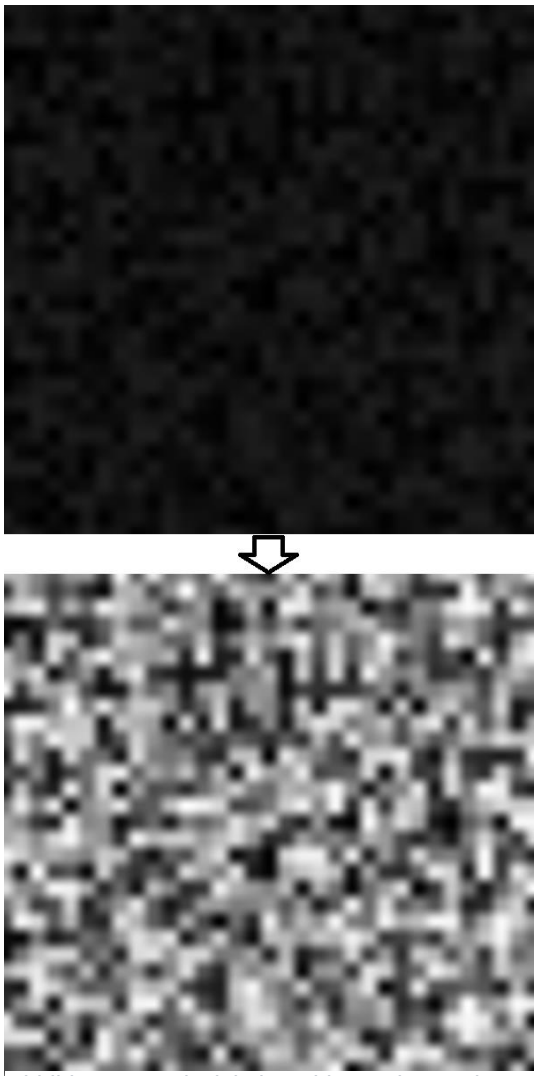

Abbildung 31 Beispiel einer dritten Oktave eines Value-Noise. Die Oktave wurde zur besseren Erkennbarkeit standardisiert und anschließend mit 255 multipliziert.

Um zwischen 1 und 0 zu standardisieren, muss zuerst der niedrigste Wert des Noise ermittelt werden, im Folgenden als min bezeichnet. Dann wird von jedem Wert dieser min-Wert abgezogen. Als Formel ausgedrückt:

$$f(noise) = noise - min$$

Dadurch ist der niedrigste Wert nun gleich 0.

Wenn man nun den maximalen Wert des Noise ermittelt, im Folgenden als max bezeichnet, kann man alle Werte des Noise durch max dividieren und erhält dadurch den standardisierten Wert zwischen 0 und 1. Als Formel ausgedrückt:

$$standardisiert(noise) = \frac{f(noise)}{max}$$

Diese beiden Schritte können auch zusammengefasst werden. Wird im originalen Noise zuerst der min und max errechnet, so gelangt man direkt zur Standardisierung über folgende Formel:

$$standardisiert(noise) = \frac{noise - min}{\max - min}$$

Formel 13 Standardisierung

3. Modelle und Konzepte

3.1. Weitere Überlegungen zu Value-Noise

Bei dem dargestellten Ergebnis in Abbildung 32 kann man bereits um den Punkt 2 sehen, dass der erzeugte Wert kleiner ist als –α. Dies ist eine wichtige Erkenntnis: das Ergebnis ist also nicht direkt durch α begrenzt, sondern der Maximal- und Minimalwert hängen von α und μ ab.

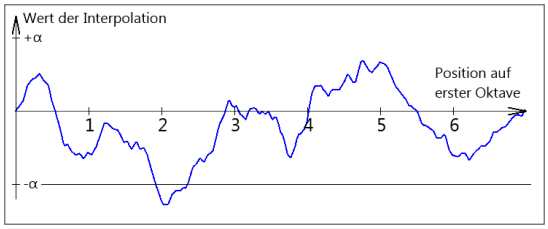

Abbildung 32 Der fertige Value-Noise ist nicht direkt durch α begrenzt.

Angenommen, wir hätten unendlich viele Oktaven und angenommen μ = 2, so gilt, wenn wir weiter annehmen, dass die Zufallspunkte überall den Wert α_j haben, dass der Wert überall gleich 2α ist.

Beweis: Der Wert der ersten Oktave ist überall α, auf der zweiten Oktave überall α/2, auf der dritten Oktave überall α/4 usw. Dies lässt sich als folgende Formel ausdrücken:

$$\sum_{m=0}^{\infty} \frac{\alpha}{2^m} = \alpha + \frac{\alpha}{2} + \frac{\alpha}{4} + \frac{\alpha}{8} + \cdots$$

Diese kann umgestellt werden zu:

$$\alpha * \sum_{m=0}^{\infty} \frac{1}{2^m} = \alpha * (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots)$$

Die unendliche Summe 1 + ½ + ¼ + … konvergiert bekannterweise gegen 2. Also:

$$\alpha * \sum_{m=0}^{\infty} \frac{1}{2^m} = 2\alpha$$

q.e.d.

Formel 14 Beweis, dass der Value-Noise den Wertebereich zwischen -α und α verlassen kann

Dies gilt jedoch nur für Interpolationen, die nicht über- bzw. unterschwingen, also z.B. lineare und trigonometrische Interpolation sowie die Sprungfunktion. Für Interpolationsfunktionen wie die kubische Interpolation gilt dies nicht. Durch das Über- bzw. Unterschwingen können größere Werte als α_j angenommen werden. Für nicht über- bzw. unterschwingende Interpolationsfunktionen gilt also im Fall von μ = 2, dass sich das Ergebnis immer im Bereich zwischen -2α und +2α befindet. In der Praxis, wenn man also nicht unendlich viele Oktaven errechnet, sind die Werte entsprechend etwas näher an 0.

Im allgemeinen Fall, für μ > 1 (μ darf nicht gleich 1 sein, da sonst der Zufallsbereich nicht kleiner werden würde und daher der Value-Noise bei unendlich vielen Oktaven nicht konvergieren würde) gilt, dass das Ergebnis bei nicht schwingenden Interpolationsfunktionen sich immer im Bereich zwischen $-\frac{\alpha*\mu}{\mu-1}$ und $+\frac{\alpha*\mu}{\mu-1}$ befindet.

Beweis: Formel von oben jedoch mit μ anstelle von 2:

$$\alpha * \sum_{m=0}^{\infty} \frac{1}{\mu^m} = \alpha * (1 + \frac{1}{\mu} + \frac{1}{\mu^2} + \frac{1}{\mu^3} + \cdots)$$

Die unendliche Summe ist:

$$\sum_{m=0}^{\infty} \frac{1}{\mu^m} = \frac{\mu}{\mu - 1}$$

weil:

$$\begin{aligned}
(\mu - 1) * \sum_{m=0}^{\infty} \frac{1}{\mu^m} &= \mu \\
\mu * \sum_{m=0}^{\infty} \frac{1}{\mu^m} - \sum_{m=0}^{\infty} \frac{1}{\mu^m} &= \mu \\
\left(\mu + 1 + \frac{1}{\mu} + \frac{1}{\mu^2} + \frac{1}{\mu^3} + \cdots\right) - \left(1 + \frac{1}{\mu} + \frac{1}{\mu^2} + \frac{1}{\mu^3} + \cdots\right) &= \mu \\
\mu + (1 - 1) + \left(\frac{1}{\mu} - \frac{1}{\mu}\right) + \left(\frac{1}{\mu^2} - \frac{1}{\mu^2}\right) + \left(\frac{1}{\mu^3} - \frac{1}{\mu^3}\right) + \cdots &= \mu \\
\mu + 0 + 0 + 0 + 0 + \cdots &= \mu \\
\mu &= \mu
\end{aligned}$$

Damit ergibt sich:

$$\alpha * \sum_{m=0}^{\infty} \frac{1}{\mu^m} = \alpha * \frac{\mu}{\mu - 1} = \frac{\alpha * \mu}{\mu - 1}$$

q.e.d.

Formel 15 Beweis für den exakten Wertebereich eines Value-Noise

3.2. Weitere Überlegungen zu Perlin-Noise

Auffällig bei Abbildung 33 ist, dass dieses Perlin-Noise-Beispiel niemals größer als +α oder kleiner als –α ist. Tatsächlich ist Perlin-Noise im Fall von μ = 2 ausschließlich direkt durch α begrenzt.

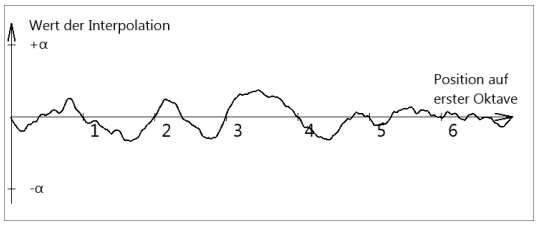

Abbildung 33 1D-Perlin-Noise-Beispiel

Beweis:

Der maximal mögliche Wert einer Oktave j liegt bei $\frac{\alpha_j}{2}$. Dies liegt daran, dass der maximale Wert nur erreicht werden kann, wenn eine Stützstelle exakt die Steigung $+\alpha_j$ hat und die nachfolgende Stützstelle die Steigung $-\alpha_j$. Damit ergibt sich für die Interpolationsfunktion f folgende Gleichung:

$$f(\alpha_j, -\alpha_j, t) = 2 * (\alpha_j - -\alpha_j) * t^4 - (3\alpha_j - -5\alpha_j) * t^3 - -3\alpha_j * t^2 + \alpha_j * t$$
$$f(\alpha_j, -\alpha_j, t) = 4\alpha_j * t^4 - 8\alpha_j * t^3 + 3\alpha_j * t^2 + \alpha_j * t$$

Der maximale Wert liegt bei t=0,5 und hat den Wert $\frac{\alpha_j}{2}$.

$$f(\alpha_j, -\alpha_j, 0{,}5) = 4\alpha_j * 0{,}5^4 - 8\alpha_j * 0{,}5^3 + 3\alpha_j * 0{,}5^2 + \alpha_j * 0{,}5$$
$$f(\alpha_j, -\alpha_j, 0{,}5) = 0{,}25\alpha_j - \alpha_j + 0.75\alpha_j + 0{,}5\alpha_j$$
$$f(\alpha_j, -\alpha_j, 0{,}5) = 0{,}5\alpha_j$$

Da μ = 2 halbiert sich α mit jeder Oktave. Das heißt, auf der ersten Oktave ist der maximale Wert = $\frac{\alpha}{2}$, auf der zweiten $\frac{\alpha}{4}$, auf der dritten $\frac{\alpha}{8}$ usw. Der Wert des Perlin-Noise ist die Summe dieser Werte.

Das lässt sich als Summe schreiben:

$$\sum_{m=1}^{\infty} \frac{\alpha}{2^m} = \frac{\alpha}{2} + \frac{\alpha}{4} + \frac{\alpha}{8} + \cdots$$

Diese Summe hat den Wert α.

q.e.d.

Formel 16 Beweis, dass der Perlin-Noise zwischen -α und α begrenzt ist.

Im allgemeinen Fall ist der Perlin-Noise, genau wie der Value-Noise durch α und μ begrenzt mit $\frac{\alpha}{\mu-1}$.

Beweis:

Laut *3.1. Weitere Überlegungen zu Value-Noise* gilt:

$$\sum_{m=0}^{\infty} \frac{1}{\mu^m} = \frac{\mu}{\mu - 1}$$

Damit gilt:

$$\sum_{m=1}^{\infty} \frac{1}{\mu^m} = \frac{\mu}{\mu - 1} - 1$$
$$\sum_{m=1}^{\infty} \frac{1}{\mu^m} = \frac{\mu - (\mu - 1)}{\mu - 1}$$
$$\sum_{m=1}^{\infty} \frac{1}{\mu^m} = \frac{1}{\mu - 1}$$

Dadurch ergibt sich:

$$\sum_{m=1}^{\infty} \frac{\alpha}{\mu^m} = \frac{\alpha}{\mu^1} + \frac{\alpha}{\mu^2} + \frac{\alpha}{\mu^3} + \cdots$$
$$\alpha \sum_{m=1}^{\infty} \frac{1}{\mu^m} = \alpha * \left(\frac{1}{\mu^1} + \frac{1}{\mu^2} + \frac{1}{\mu^3} + \cdots\right)$$
$$\alpha \sum_{m=1}^{\infty} \frac{1}{\mu^m} = \frac{\alpha}{\mu - 1}$$

q.e.d

Formel 17 Beweis für den exakten Wertebereich eines Perlin-Noise

3.3 Sprungfunktion als Interpolation in Value-Noise

Wie in Kapitel *2.1. Interpolation* bereits erwähnt, handelt es sich bei der Sprungfunktion streng mathematisch um keine Interpolation, da die Funktion beim eigentlichen Sprung nicht stetig ist. Im Rahmen dieser Thesis sei die Sprungfunktion aber auch als Interpolation definiert, da sie für die vorliegenden Probleme alle nötigen Eigenschaften erfüllt – sie verbindet 2 diskrete Werte miteinander. Mehr wird von einer Interpolation im Rahmen dieser Thesis nicht gefordert.

Alle bisherigen Interpolationen im Value-Noise haben im 1D-Fall noch keine signifikanten mathematisch unterschiedlichen Ergebnisse geliefert.

Dies gilt jedoch nicht für die Sprungfunktion. Bei der Sprungfunktion sind bereits in der mathematischen Darstellung (Abbildung 34) signifikante Unterschiede zu sehen. Bei dieser Darstellung wurden die eigentlichen Sprünge ebenfalls mit Linien verbunden. Mathematisch ist dies nicht korrekt, da die Sprungfunktion wirklich Sprünge macht, jedoch entspricht diese Darstellung eher dem späteren Praxisbezug.

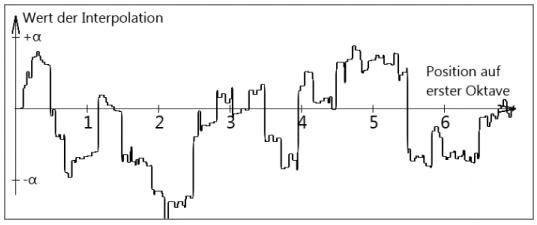

Abbildung 34 Sprungfunktion als Interpolationsfunktion

Noch offensichtlicher wird der Unterschied im 2D-Fall, wie in Abbildung 35 zu sehen. Je nach Anwendungsfall kann auch die Sprungfunktion sehr gute Ergebnisse liefern.

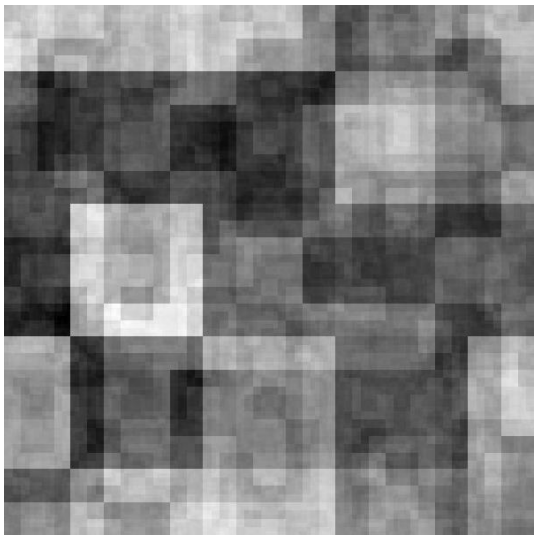

Abbildung 35 Die Sprungfunktion als Interpolation im 2D-Fall

3.4. Beweis von 2.3.3 Value-Noise ND

In *2.3.3 Value-Noise ND* wurde behauptet, dass sich die Anzahl der 1-dimensionalen Interpolationen über folgende Formel berechnen lassen würde:

$$f(n) = 2^n - 1$$

mit:

- n = Anzahl Dimensionen

Diese Formel habe ich schon in anderen wissenschaftlichen Arbeiten gesehen, zum Beispiel Perlin (1999), jedoch nie mit Beweis.

Dies wird in diesem Abschnitt bewiesen.

Beweis:

Die Anzahl der 1-dimensionalen Interpolationen verdoppelt sich mit jeder Dimension und es kommt noch eine zusätzliche Interpolation hinzu. Als Formel ausgedrückt bedeutet dies:

$$f(n) = f(n-1) * 2 + 1$$

mit:

$$f(1) = 1$$

da im 1-dimensionalen Fall nur eine Interpolation notwendig ist.

Vollständige Induktion:

A(2):

$$f(2-1) * 2 + 1 = 2^2 - 1$$
$$f(1) * 2 + 1 = 4 - 1$$
$$1 * 2 + 1 = 3$$
$$3 = 3$$

A(n):

$$f(n-1) * 2 + 1 = 2^n - 1$$
$$f(1) = 1$$

A(n+1):

$$f(n) * 2 + 1 = 2^{n+1} - 1$$
$$f(n) * 2 = 2^{n+1} - 2$$
$$f(n) = \frac{2^{n+1} - 2}{2}$$
$$f(n) = 2^n - 1$$

q.e.d.

3.5. 4-Ray

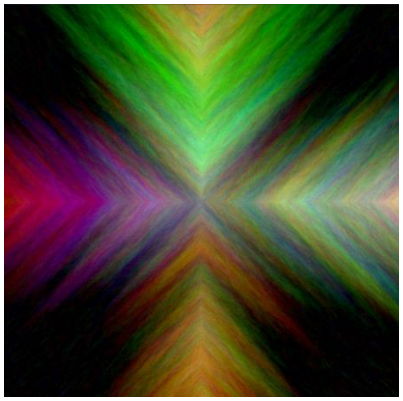

Abbildung 36 Ein typisches 4-Ray-Ergebnis

Der 4-Ray-Algorithmus ist ein von mir selbst entwickelter Algorithmus zum Erstellen von Bildern, die aussehen, als wären sie durch Pinselstriche gezeichnet worden. Charakteristisch für die Bilder des 4-Ray-Algorithmus ist, dass sich horizontal, nach links und rechts sowie vertikal nach oben und unten jeweils 4 „Strahlen“, „Rays“, bilden, die sich in der Regel bis zum Bildende erstrecken.

Während der Algorithmus läuft und das Bild errechnet wird, hat jedes Pixel einen RGB (Rot-Grün-Blau) -Wert, jeweils zwischen 0 und 255, wobei 0/0/0 Schwarz ist und 255/255/255 Weiß. Jeder der 3 RGB-Werte muss zu jedem Zeitpunkt eine ganze Zahl sein. Ist das Ergebnis einer Berechnung eine Kommazahl, so wird abgerundet. Außerdem befindet sich jedes Pixel in einem der 3 Stadien „inaktiv“, „aktiv“ und „gesetzt“. Nur im Status „gesetzt“ ist der RGB-Wert wichtig, die anderen beiden Werte beschreiben, dass dieser Wert noch ausgerechnet werden muss.

Am Anfang werden alle Pixel des Bildes auf „inaktiv“ gestellt, außer dem Pixel in der Mitte des Bildes, dieses wird auf „gesetzt“ gestellt und sein RGB-Wert wird auf den Grauwert in der Mitte zwischen Weiß und Schwarz gestellt, also 127/127/127. Nun werden folgende Schritte durchlaufen:

1. Durchlaufe jede Zeile bis ein inaktives Pixel gefunden wurde, das einen gesetzten Nachbarn hat. 2 Pixel sind benachbart, wenn sie vertikal oder horizontal direkt nebeneinanderliegen, diagonale Pixel sind NICHT benachbart. Im Weiteren wird angenommen, dass das Bild von oben links nach unten rechts durchlaufen wird, jeweils immer zuerst die Zeilen und dann die Spalten.
2. Setze das Pixel auf „aktiv“. Das bedeutet, dass seine Berechnung begonnen hat.
3. Berechne das arithmetische Mittel der RGB-Werte aller gesetzten Nachbarn dieses Pixels.
4. Addiere auf jeden der 3 RGB-Werte jeweils einen zufälligen Wert zwischen **-α** und **+α** (Definition von **α** siehe unten).
5. Runde alle 3 RGB-Werte des Pixels ab.

6. Setze das Pixel auf „gesetzt“ und weise ihm die errechneten RGB-Werte aus 5. zu.
7. Gehe zu 1., setzte die Suche aber bei dem in 6. gesetzten Pixel fort bis alle Pixel durchlaufen wurden.
8. Gehe zu 1. und fange wieder oben links im Bild an, bis alle Pixel als „gesetzt“ markiert wurden.

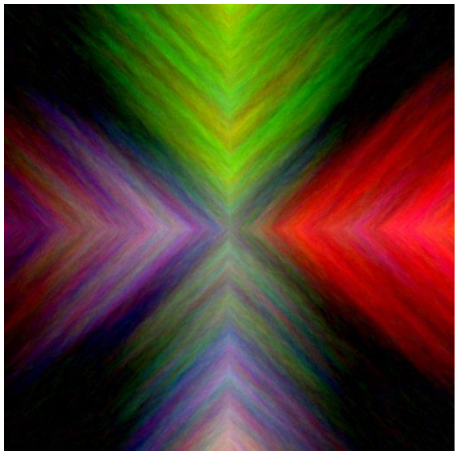

Abbildung 37 Mögliches Ergebnis mit α = 5

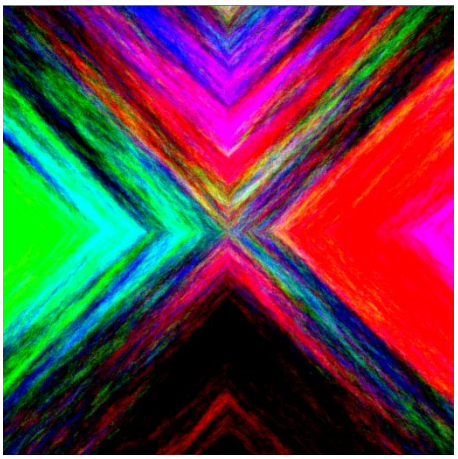

Abbildung 38 Mögliches Ergebnis mit α = 25

Der Wert **α** kann dabei frei gewählt werden, in Tests hat sich allerdings ein Wert von 5 als guter Richtwert herausgestellt.

Die charakteristische Eigenschaft der 4 Rays (Strahlen) erhält das Bild dadurch, dass beim Durchlaufen des Bildes in der Mitte der 4 Strahlen jeweils nur ein Nachbarpixel bereits gesetzt wurde. Beim Errechnen des arithmetischen Mittels und dem anschließenden Abrunden nimmt also kein Wert ab, sondern bleibt gleich. Bei allen anderen Pixeln des Bildes müssen immer 2 Nachbarpixel zur Berechnung herangezogen werden und deren arithmetisches Mittel gebildet und abgerundet werden, was in 50% der Fälle zu einem Werteverlust führt, in den anderen 50% zu keinem Wertegewinn. Daher neigen die Pixel dazu, mit größerer Entfernung der Rays immer dunkler zu werden, bis sie komplett schwarz eingefärbt werden.

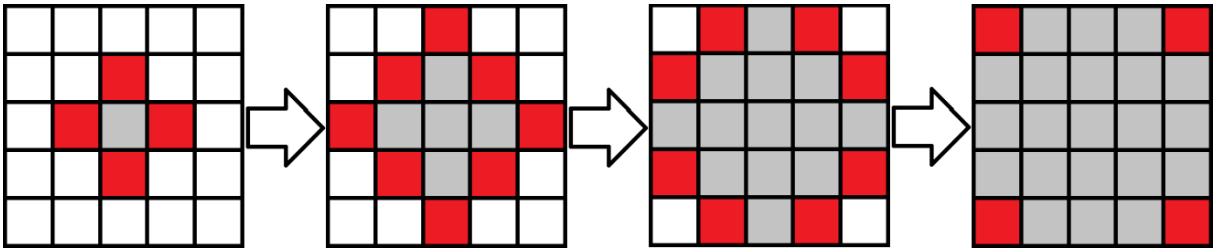

Abbildung 39 Optimierungsmöglichkeit: Nur die roten, aktiven Pixel müssen untersucht werden, die grauen wurden bereits gesetzt und die weißen sind inaktiv.

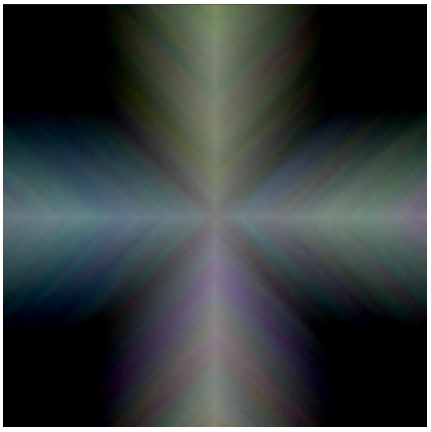

Abbildung 40 Mögliches Ergebnis mit α = 1

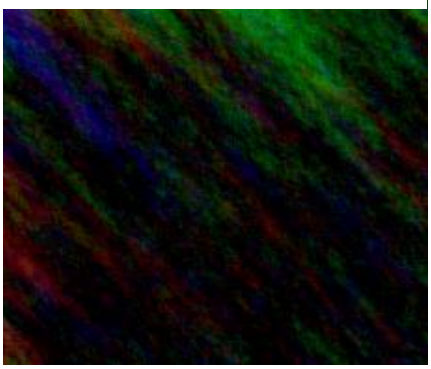

Abbildung 41 Nachträglich aufgehellte Abbildung: Sind nur Werte zwischen 0 und 255 zulässig, so werden die schwarzen Ecken des Bildes "schmutzig".

Die Algorithmusbeschreibung von oben kann deutlich performanter umgesetzt werden, indem immer nur die in Abbildung 39 rot gefärbten Pixel untersucht werden. Dadurch hat der Algorithmus lediglich eine Laufzeit von O(Breite*Höhe) anstatt wie vorher eine Laufzeit von O((Breite*Höhe)²), wobei Breite und Höhe jeweils die Breite und Höhe des Bildes in Pixeln repräsentieren.

Dadurch wird noch ein anderes Problem behoben: Der Ray, der nach rechts zeigt, ist beim ursprünglichen Algorithmus 1 Pixel höher als der Ray nach links. Dies liegt daran, dass der Ray nach rechts der erste Ray ist, der generiert wird und zwar aus dem ersten Pixel, das vom ursprünglichen Pixel in der Mitte nach oben geht.

Beschränkt man die Werte von RGB jeweils, so wie oben beschrieben, zu jedem Zeitpunkt zwischen 0 und 255, so werden die schwarzen Ecken des Bildes unsauber und werden nie komplett in größeren Gebieten schwarz. Dies nenne ich im Folgenden „schmutzig“. Dies liegt daran, dass durch den zufälligen Wert zwischen **-α** und **+α** die Wahrscheinlichkeit hoch ist (50%), aus einem Nullwert keinen Nullwert zu machen, also, dass z.B. aus einem Nullwert im Rotkanal plötzlich wieder ein Rotwert > 0 entsteht.

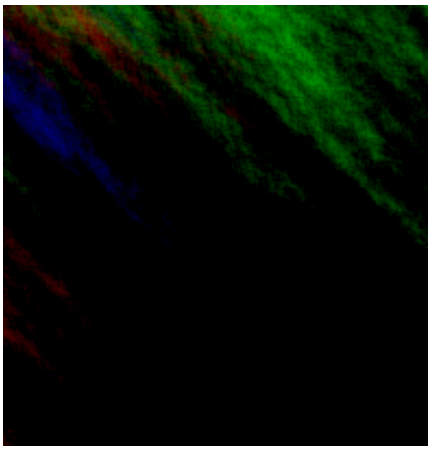

Abbildung 42 Durch das Zulassen beliebiger Werte werden die Ecken "schmutzfrei"

Dieses Problem lässt sich beheben, indem man bei der Berechnung beliebige Werte, auch abseits des Bereichs von 0 bis 255, zulässt, den Wert beim Rendern dann aber clampt, das heißt, zu hohe und zu niedrige Werte jeweils auf 0 bzw. 255 stellt. Dadurch bekommen die Pixel in den Ecken im Durchschnitt immer negativere Werte und die Wahrscheinlichkeit einer Farbentstehung aus einem schwarzen Pixel sinkt mit zunehmender Distanz von den Rays immer weiter auf 0.

4. Implementierung

Abbildung 43 Beispielinsel. Das folgende Kapitel erklärt u.a. wie eine solche Insel in Unity generiert werden kann.

Im nun Folgenden werden die in den Kapiteln 2 und 3 vorgestellten Konzepte in Unity (2016) implementiert (Ausnahme: *3.5. 4-Ray* und beide ND-Noise-Abschnitte). Unity ist eine 3D-Spieleengine mit eingebauter Terrainengine. Dieser Terrainengine kann man eine sogenannte Heightmap übergeben, welche dann die Höhe des Terrains festlegt. Eine Heightmap ist ein 2-dimensionales Floatarray, in dem jeder Index für eine Höhe im Terrain steht. Die Terrainengine interpoliert dann zwischen diesen diskreten Werten die jeweiligen Höhen der Landschaft. Dies ist wichtig zu erwähnen, da die Sprungfunktion wirklich Sprünge macht. Ohne diese unityinterne Interpolation wären im Terrain Löcher zu sehen.

Die hier gezeigten Terrainbeispiele sind im Rahmen eines Computerspiels mit dem Arbeitstitel NHP (New Horizon Procedural) entstanden. Sie wurden von mir entwickelt und generiert. NHP spielt in einer prozedural generierten Karibiklandschaft, welche ausschließlich aus Inseln besteht. Diese Inseln wurden mit Value-Noise und Perlin-Noise erstellt.

4.1 Erzeugung der Inselplätze

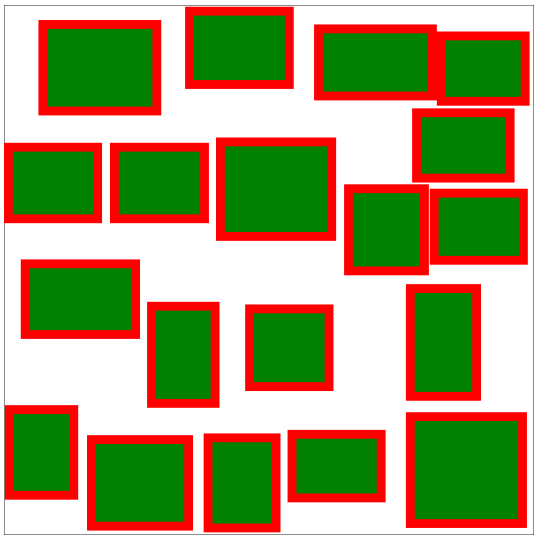

Abbildung 44 Beispielhafte Platzierung der Inselplätze. Weiß ist freier Bereich (in diesem Spiel also Meer), rot ist der Sicherheitsabstand und grün sind die tatsächlichen Inseln.

Jedes Spielelevel in NHP besteht aus einer rechteckigen Karte, auf welcher sich mehrere Inseln befinden. Diese Inseln wurden zufällig auf der Karte platziert und werden zunächst nur als AABBs (Bergen, 1997) (Axis-Aligned-Bounding-Box) dargestellt, also nicht rotierte Rechtecke. Die Breite und Höhe dieser Inseln wird zufällig im Wertbereich zwischen $\frac{Kartenbreite}{20}$ und $\frac{Kartenbreite}{10}$ sowie $\frac{Kartenhöhe}{20}$ und $\frac{Kartenhöhe}{10}$ gewählt. Nachdem x- und y-Koordinaten zufällig gewählt wurden, wird überprüft, ob die Insel mit einer anderen bereits platzierten Insel kollidiert. Um zu verhindern, dass sich 2 Inseln zu nahekommen, wird zusätzlich noch ein Sicherheitsabstand von den einzelnen Inseln

zueinander mit einberechnet. Ist dies der Fall, so werden neue zufällige Koordinaten gewählt. Es werden so lange neue Inseln platziert, bis die neue Insel 1024-mal mit anderen Inseln kollidiert ist und mindestens 5 Inseln platziert wurden. Alternativ wird abgebrochen, wenn eine maximale Anzahl an Inseln erreicht wurde (z.B. 10).

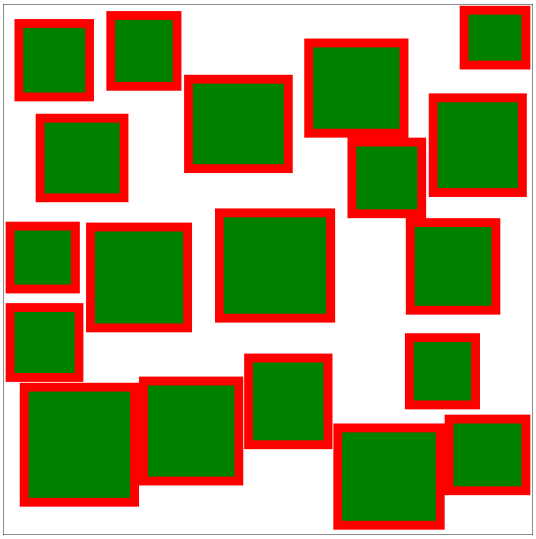

Abbildung 45 Modifizierte Inselplatzberechnung

Wurden diese Inselplätze platziert, so ist sichergestellt, dass sich die Inseln gegenseitig nicht beeinflussen. Im Folgenden müssen also ausschließlich die in Abbildung 44 grünen Bereiche berücksichtigt werden, der Rest kann bei der Generierung jeder einzelnen Insel ignoriert werden. Bei der weiteren Entwicklung von NHP hat sich herausgestellt, dass die Inseln „schöner“ aussehen, wenn sie sich der quadratischen Grundform annähern. Daher wurden die ursprünglichen Breiten- und Höhengrenzen modifiziert. Die Breitenberechnung blieb identisch, die Höhenberechnung hat aber eine Range zwischen Inselbreite - 10% und Inselbreite + 10%. Als Formeln ausgedrückt:

$$inselbreite = rand\left(\frac{Kartenbreite}{20}, \frac{Kartenbreite}{10}\right)$$
$$inselhöhe = rand(inselbreite * 0.9, inselbreite * 1.1)$$

Formel 18 Werte der Inselbreite und Inselhöhe

wobei $rand(a, b)$ eine zufällige Zahl zwischen a und b (jeweils inklusive) generiert.

Die Koordinaten x/y seien hierbei der Punkt oben links jedes Rechtecks und der Ursprung sei oben links im Gesamtbild. Die x-Achse verläuft nach rechts, die y-Achse nach unten, so wie in vielen Programmiersprachen üblich.

Damit ergibt sich, dass die x- und y-Koordinaten nicht komplett frei gewählt werden dürfen, da sonst einige Inseln die Spielkarte verlassen würden. x und y müssen, als Formel ausgedrückt, folgenderweise berechnet werden:

$$x = rand(0, Kartenbreite - inselbreite)$$
$$y = rand(0, Kartenhöhe - inselhöhe)$$

Formel 19 Bestimmung der Inselposition

4.2 Insellandschaft mit Value-Noise

Die ersten Einstellungen für den Value-Noise zu finden, war ein längerer „Trial-and-Error-Task“ – es gibt hier keine falschen oder richtigen Ergebnisse. Ich habe stattdessen solange die Zahlen verändert, bis mir das Ergebnis gefallen hat. Die Insel in Abbildung 46 hatte folgende Werte:

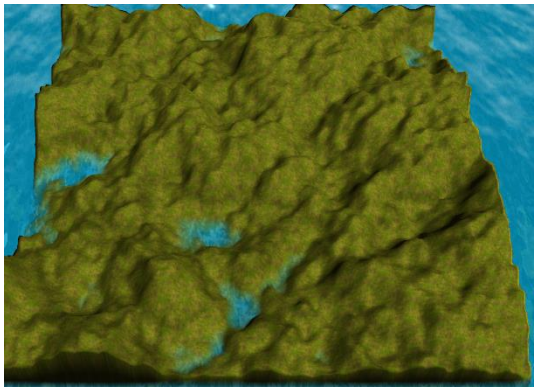

Abbildung 46 Die erste Inselversion in NHP

$$inselbreite = rand(102, 204)$$
$$inselhöhe = rand(inselbreite * 0.9, inselbreite * 1.1)$$
$$kx = \lfloor inselbreite * 0.07 \rfloor$$
$$ky = \lfloor inselhöhe * 0.07 \rfloor$$
$$Oktaven = rand(4, 9)$$
$$\beta = 2$$
$$\mu = \mathrm{rand}(1.8, 3.3)$$
$$Smoothing: nein$$
$$Standardisierung: ja$$
$$Interpolationsfunktion: kubisch$$

Die Wahl der Inselbreite und Inselhöhe wurde bereits in Kap. *4.1 Erzeugung der Inselplätze* erklärt. Das bedeutet, dass die Kartenbreite und die Kartenhöhe je den Wert 2040 haben (genau genommen 2048, dies wurde aber gerundet).

Aus der Berechnung ergibt sich, dass *kx* und *ky* jeweils zwischen 7 und 14 liegen – dies hat sich als guter Wert herausgestellt. Dadurch, dass *kx*, *ky* und die Oktavenanzahl zufällig gewählt wurden, ergibt sich eine unterschiedliche Inselcharakteristik. Manche Inseln sind dadurch rauer, andere weicher. Die Werte für β und μ sind Werte, die sich durch Ausprobieren ergaben. Auf Smoothing wurde verzichtet, da dies die rauen Inseln zu glattgemacht hätte. Da aber unterschiedliche Inselcharakteristiken gewünscht sind, hätte dies das Ergebnis verschlechtert.

Die Standardisierung wurde aktiviert. Dadurch war es möglich, einen weiteren zufälligen Wert p zwischen 1 und 5 zu generieren und diesen Wert mit dem Noise zu multiplizieren. Dies hat weiter die unterschiedlichen Inselcharakteristiken gebildet. Manche Inseln sind nun eher flach (nahe bei p=1), andere haben viele Berge (nahe bei p=5).

Die kubische Interpolationsfunktion aus *2.1. Interpolation* lieferte die besten Ergebnisse. Dennoch werden nun auch die anderen Interpolationen vorgestellt.

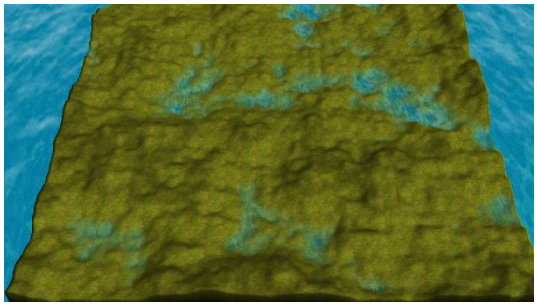

Abbildung 47 Negativbeispiel für die trigonometrische Interpolation

Die trigonometrische Interpolation lieferte im Allgemeinen vergleichbare Ergebnisse zur kubischen Interpolation. Bei niedrigen Werten von p, also bei flacheren Inseln ist aber deutlich eine Wellenform zu sehen, welche nicht der Realität entspricht (Beispiel in Abbildung 47 zu sehen).

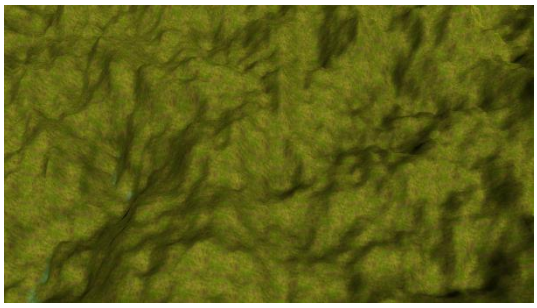

Abbildung 48 Beispiel für eine Landschaft mit linearer Interpolation und Value-Noise

Die lineare Interpolation liefert, wenn man die Inseln von der Distanz beobachtet, gute Ergebnisse. Allerdings fallen deutliche Kanten auf, sobald man die Kamera näher an die Inseln heranbewegt, so in Abbildung 48 zu sehen.

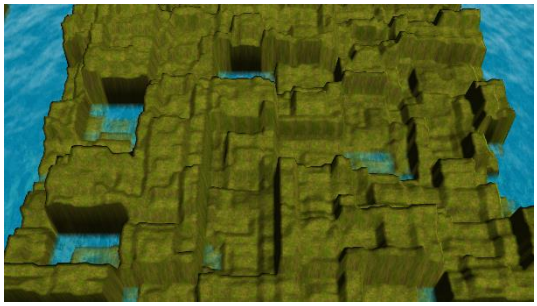

Abbildung 49 Eine "Insel" mit Value-Noise und Sprungfunktion

Deutlich extremere Werte liefert die Sprungfunktion, so in Abbildung 49 zu sehen. Damit die Landschaft nicht zu viele unterschiedliche Werte hatte, musste ich für diese Abbildung µ auf 3,5 erhöhen. Diese Interpolation orientiert sich überhaupt nicht mehr an der Realität, sondern generiert eckige, blockartige Inseln. Dies kann für ein Computerspiel je nach Genre wünschenswert sein. Einige Spiele bauen solche „merkwürdigen“ Level auch als Easteregg (Distelmeyer, 2007, S. 399) ein, wie zum Beispiel Diablo III in dessen Eastereggleve (IncGamers Ltd, 2014). Diese Interpolationsfunktion kann aber auch für sinnvolle Zwecke benutzt werden, wie zum Beispiel für die prozedurale Oberflächengenerierung eines Raumschiffes.

4.2.1 Strand

Ein großes Problem der Glaubwürdigkeit der bisherigen Inseln war, dass sie keinen Übergang ins Meer besitzen. Sie hören einfach auf, sind daher immer noch viereckig und überhaupt nicht realistisch. Die Behebung dieses Problems erkläre ich in diesem Abschnitt. Als „Strand“ sei hier einfach nur der Übergang ins Meer definiert. Die wirkliche Strandtextur wird erst in *4.2.2 Texturieren* hinzugefügt.

In NHP wird der 2D-Value-Noise mit einer Maske multipliziert. Eine Maske ist ein anderes 2-dimensionales Floatarray. Eine Multiplikation bedeutet dabei, dass elementweise mit der Maske multipliziert wird. Diese Maske soll 3 Eigenschaften besitzen:

1. In einem größeren Bereich in der Mitte soll sie den Wert 1 haben, also den Value-Noise nicht verändern.
2. Außen soll sie den Wert 0 haben. Nach der Multiplikation ist der Value-Noise also auch dort = 0.
3. Zwischen der Mitte und außen soll ein sauberer Übergang sein, so dass die Insel langsam im Meer verschwinden kann.

Diese 3 Eigenschaften können über folgende Formel erreicht werden:

$$maske(x,y) = clamp\left(2 * \left(1 - \frac{\left|\frac{b}{2} - x\right|}{\frac{b}{2}}\right)\right) * clamp\left(2 * \left(1 - \frac{\left|\frac{h}{2} - y\right|}{\frac{h}{2}}\right)\right)$$

Formel 20 Maske zur Stranderzeugung

mit:

- b = Breite der Maske in Pixel
- h = Höhe der Maske in Pixel

und:

$$clamp(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 1 \\ x, & sonst \end{cases}$$

Nach der Multiplikation sind die gewünschten Werte erreicht und die Inseln verschwinden am Strand im Meer.

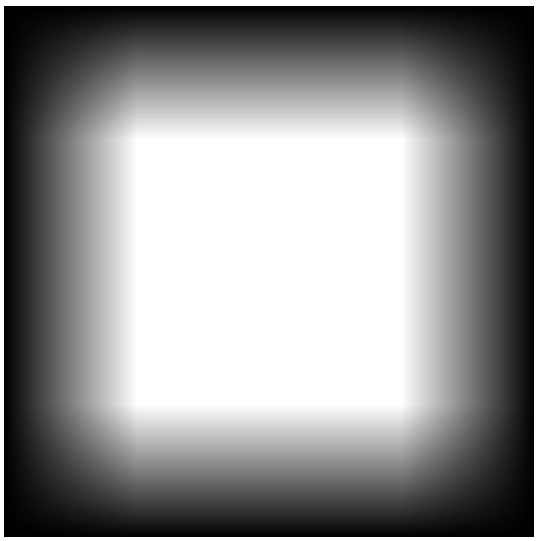

Abbildung 50 Die beschriebene Maske als Bild. Weiße Pixel sind 1, schwarze 0. Graustufen sind die entsprechenden Werte dazwischen.

Der tatsächliche Heightmapwert ist, als Formel ausgedrückt:

$$heightmap(x,y) = noise(x,y) * maske(x,y)$$

In Abbildung 51 wurde eine neue Insel mit den Werten von *4.2 Insellandschaft mit Value-Noise* generiert und die Maske wurde multipliziert. Wie zu sehen ist, gibt es nun einen sauberen Übergang ins Meer.

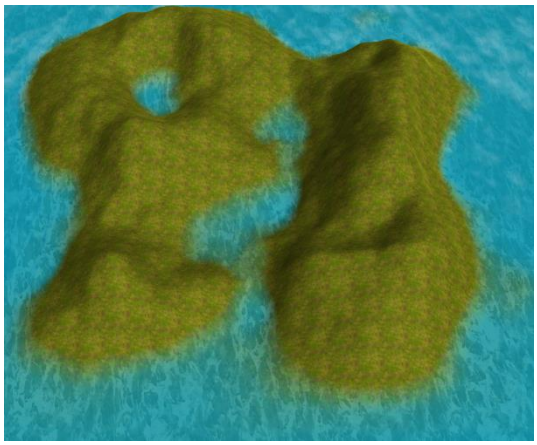

Abbildung 51 Beispiel für den Übergang ins Meer

4.2.2 Texturieren

Die Insel hat bis jetzt nur eine einzige Grastextur. Dies ist optisch nicht attraktiv und die Behebung wird im folgenden Abschnitt erläutert.

In NHP soll es 4 unterschiedliche Texturen geben: Sand, Gras, Steine und Gebirge. Bei meinen Versuchen hat sich herausgestellt, dass es für gute Ergebnisse reicht, nur die Höhe der Insel an einem jeweiligen Punkt zu betrachten und mit Hilfe dieses Höhenwertes die jeweilige Textur an dieser Stelle zu bestimmen.

In Unity hat jedes Terrain eine 3-dimensionale Texturmatrix, deren Spaltensumme jeweils 1 ergeben muss. Einfacher: Auf dem gesamten Terrain liegt ein Grid. In jeder Zelle dieses Grids gibt es einen Wert für jede der 4 Texturen, welcher den Prozentwert der Deckkraft der jeweiligen Textur angibt.

Ich habe mich für die Gaußsche Normalverteilung (Arens u. a., 2012, S. 1347) entschieden. Diese entspricht folgender Formel:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Formel 21 Gaußsche Normalverteilung

wobei die Werte σ und μ frei gewählt werden können und e die Eulersche Zahl ist.

Bei Tests hat sich herausgestellt, dass die maximale Höhe der Berge bei etwa 2,5 liegt (kann bei extremen Zufallswerten abweichen). Dies ist ein zufälliger Wert, der sich aus meinen für die Terraingenerierung genutzten Konstanten ergeben hat. Er hat einen anderen Wert bei der Wahl anderer Konstanten. Durch einige Tests haben sich für die 4 Texturen folgende Werte ergeben:

Sand: $\sigma = 0{,}15$ $\mu = 0$

Gras: $\sigma = 0{,}15$ $\mu = 1{,}1$

Stein: $\sigma = 0{,}15$ $\mu = 1{,}5$

Gebirge: $\sigma = 0{,}15$ $\mu = 2$

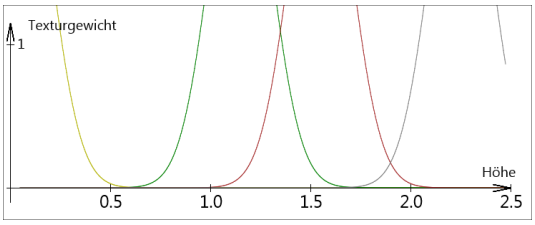

Abbildung 52 Die Kurven der Werte von links. Gelb = Sand, Grün = Gras, Braun = Stein, Grau = Gebirge

Durch diese Werte ergibt sich allerdings, wie in Abbildung 52 zu sehen, dass die Summe der einzelnen Funktionen den gewünschten Wert 1 verlässt (die Summe aller Prozentwerte muss 1 sein).

Um dieses Problem zu beheben, habe ich die Summe der Funktionen errechnet und anschließend den Wert jeder Funktion durch diese Summe geteilt. Dadurch ist die Summe = 1, die Verhältnisse bleiben aber erhalten. Das Ergebnis dieses Rechenschritts ist in Abbildung 54 zu sehen. In Abbildung 53 sind die gleichen Funktionen nochmals zu sehen, allerdings als Stacked Chart. Das bedeutet, dass die Funktionen in diesem Fall übereinander gelagert sind. Dadurch ist deutlich zu sehen, dass die Summe nun überall = 1 ist und das Verhältnis der unterschiedlichen Texturstärken ist besser erkennbar.

In Abbildung 55 ist zu sehen, wie eine Insel aussieht, nachdem die Texturfunktionen angewandt wurden. Wie man sieht, wirkt die Insel nun deutlich realistischer.

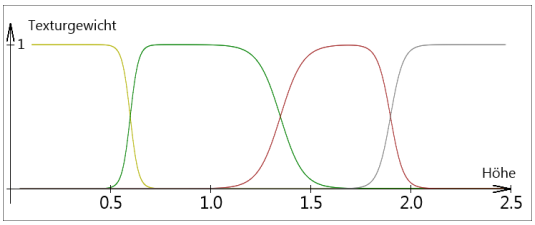

Abbildung 54 Jede Funktion wurde durch die ursprüngliche Summe geteilt. Dadurch ist die Summe der Funktionen an jeder Stelle = 1

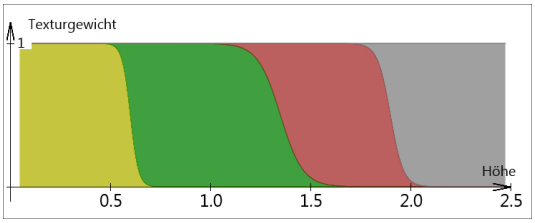

Abbildung 53 Die gleichen Werte, jedoch als Stacked Chart dargestellt.

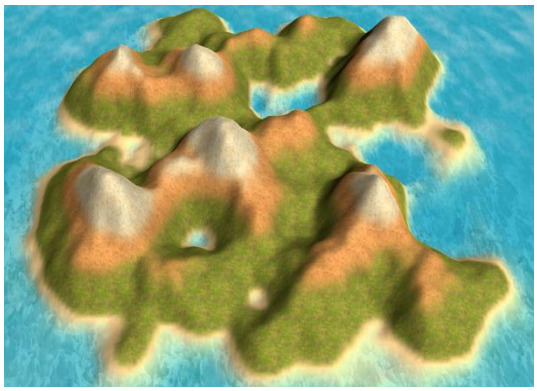

Abbildung 55 Die Texturen wurden auf die Insel gelegt.

4.2.3 Vulkane

Eine weitere Möglichkeit, unterschiedliche Inselcharakteristiken zu simulieren, ist es, auf manchen Inseln Vulkane zu erzeugen (z.B. auf 25% aller Inseln). Ich habe dies realisiert, indem Höhenwerte über einem Wert y invertiert werden, also die Höhe wieder sinkt. Als Formel ausgedrückt:

$$vulkan(x,y) = \begin{cases} x, & x < y \\ 2y - x, & sonst \end{cases}$$

Formel 22 Vulkanerzeugung

Dadurch sind deutlich stille Vulkane auf Abbildung 56 zu sehen. In dieser Abbildung ist aber auch ein Problem dieses Ansatzes zu sehen. 3 Vulkane sind klar sichtbar, während in der Mitte 2 weitere, nur angedeutete, kleine Vulkane zu sehen sind. Unklar ist, wo hier die Grenze gezogen werden sollte. Welche Fläche muss ein Vulkankopf ausfüllen, um als Vulkan zu gelten? Da diese Vulkane für das Gameplay in NHP keine Rolle spielen, sondern lediglich aus ästhetischen Gründen eingefügt wurden, ist dies für dieses Projekt nicht weiter von Bedeutung. Sollte dieser Algorithmus allerdings in anderen Spielen zum Einsatz kommen, müssen hier weitere Untersuchungen stattfinden.

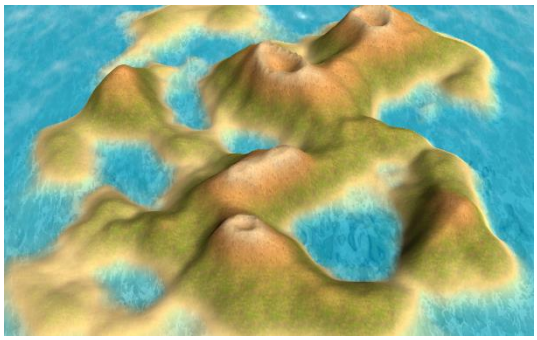

Abbildung 56 Eine Insel mit 3 (bzw. 5) Vulkanen

Ursprünglich wollte ich Vulkane nur auf Inseln mit einer gewissen Mindesthöhe erzeugen. Im Laufe der Entwicklung hat sich allerdings herausgestellt, dass auch schöne Ergebnisse bei Inseln erzeugt werden, die diese Mindesthöhe nicht erfüllen, wie in Abbildung 57 zu sehen. Dies wirkt dennoch glaubwürdig, da in der Realität die meisten kleinen Inseln ursprünglich an Bereichen mit hoher Vulkanaktivität (sog. „Hotspots“) entstanden sind.

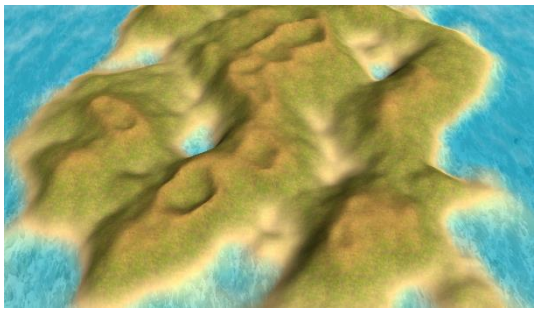

Abbildung 57 Eine flache Insel mit "Vulkanen"

4.2.4 Vegetation

Ein weiterer wichtiger Punkt der Glaubwürdigkeit ist die Vegetation. In NHP habe ich es hinsichtlich der Vegetation bei der Erstellung von Wäldern belassen (sowie der bereits beschriebenen Grastextur). Sonstige Details, wie Sträucher etc., wären bei der Skalierung nur schwer sichtbar gewesen und hätten die Glaubwürdigkeit der Inseln nicht wesentlich erhöht.

Getestet wurden zwei Ansätze. Beim ersten Ansatz werden zufällige Positionen auf der Karte generiert. Dann wird im Terrain, nachgeschaut wie hoch dieser Punkt ist und falls er über einem Minimum (im Meer und am Strand gibt es keine Bäume) und unter einem Maximum (auf hohen Gebirgen gibt es ebenfalls keine Bäume) liegt, so wird an dieser zufälligen Position ein Baum positioniert. Wie in Abbildung 58 zu sehen, erzielt dieser triviale Ansatz bereits gute Ergebnisse. Mir hat es allerdings noch nicht gefallen, dass die Dichteverteilung der Bäume auf der Landschaft an jedem Punkt ungefähr gleich ist.

Abbildung 58 Der erste Vegetationsansatz: Hier wurden die Positionen vollständig zufällig bestimmt.

Dieses Problem wurde im zweiten Ansatz behoben. Der zweite Ansatz baut direkt auf dem ersten Ansatz auf, jeder bisherige Schritt wird also übernommen. Hinzu kommt jedoch noch ein zusätzlicher 2D-Value-Noise, welcher die Größe der gesamten Karte hat. Der Wert des Value-Noise an der zufällig generierten Stelle gibt an, wie hoch die Wahrscheinlichkeit ist, dass dieser zufällig generierte Punkt angenommen wird. Nachdem also nach Ansatz eins ein zufälliger Punkt ermittelt wurde, wird zusätzlich eine weitere Zufallszahl generiert. Liegt diese über dem Wert im Value-Noise an dieser Stelle, so wird der Baum erstellt, sonst wird die Position verworfen und eine weitere Position wird generiert.

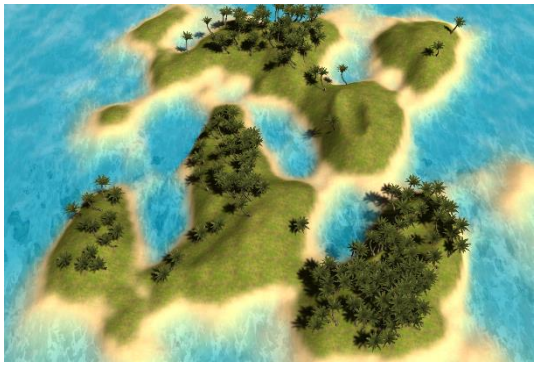

Abbildung 59 Der zweite Vegetationsansatz: Ein zusätzlicher Value-Noise wird generiert.

Beide Ansätze wurden so oft wiederholt bis eine beliebige Anzahl an Bäumen auf der ganzen Karte verteilt wurde (z.B. 1000 Stück).

4.2.5 Meeresgrund

Die natürliche Struktur der Inseln ist damit abgeschlossen. Bisher wurde die Struktur des Meeres komplett ignoriert, hier kann man das prozedural generierte Level erneut glaubwürdiger machen.

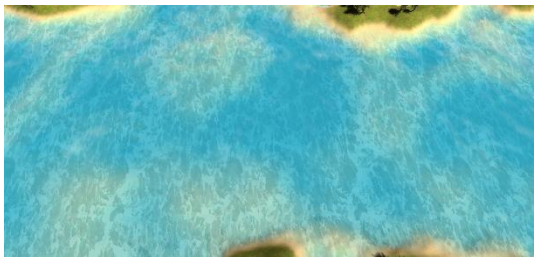

Abbildung 60 Ein weiterer Value-Noise wurde generiert, um die Struktur des Meeresbodens zu erzeugen.

Bis zu diesem Schritt ist der Meeresboden an keiner Position sichtbar, da er zu tief ist und überall flach. Dies wurde behoben, indem zusätzlich zum Value-Noise der einzelnen Inseln noch ein weiterer Value-Noise über die ganze Karte generiert wurde, welcher anschließend zu den einzelnen Höhenwerten hinzuaddiert wurde. Wie in Abbildung 60 zu sehen, gibt es durch diesen Schritt Sandbänke, die deutlich unter dem Meeresspiegel zu sehen sind. Für das Gameplay hat dieser Schritt keine Auswirkung – Schiffe bleiben also nicht stecken etc. Für diese Lösung habe ich mich entschieden, um zu verhindern, dass manche Inseln durch reinen Zufall vom Rest der Karte abgeschnitten werden.

4.2.6 Hafenbauplätze

In NHP kann der Spieler an vordefinierten Positionen Häfen errichten. An diesen Häfen gibt es zusätzliche Bauplätze, an denen der Spieler weitere Gebäude errichten kann, um auf diese Weise z.B. die Schiffe zu reparieren (Schiffswerft) oder mehr Gold zu erzeugen (Lagerhaus). Diese vordefinierten Hafenbauplätze mussten von mir (bzw. dem von mir geschriebenen Level-Generierungs-Algorithmus) noch gesetzt werden.

Zunächst mussten die Positionen der Hafenbauplätze erzeugt werden. Um dieses Problem zu lösen, habe ich mich zuerst für einen trivialen Ansatz entschieden. Es wird eine zufällige Position generiert und dann überprüft, ob die Höhe an dieser Position in einem gewünschten Wertebereich liegt. Dieser Ansatz hat sich allerdings als fehleranfällig herausgestellt. Alle Hafenbauplätze müssen vom Meer aus erreichbar sein, damit der Spieler ein Schiff zum jeweiligen Hafenbauplatz navigieren kann. Dieser triviale Ansatz erfüllte diese Anforderung nicht. Wie in Abbildung 61 zu sehen, können Hafenbauplätze mit diesem Ansatz auch in der Mitte der Insel erzeugt werden, da das Höhenlevel an dieser Stelle entsprechend niedrig sein kann.

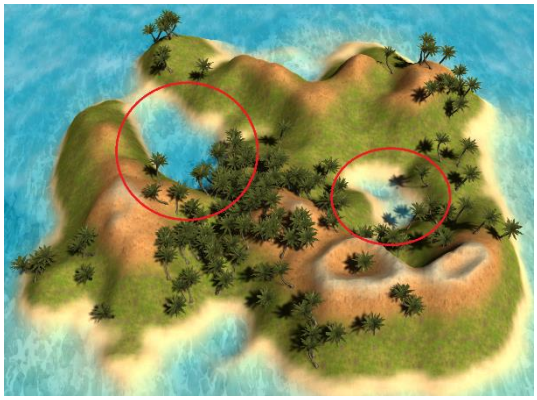

Abbildung 61 Der erste fehleranfällige Ansatz hat das Problem, dass Hafenbauplätze in der Mitte der Inseln erzeugt werden konnten (z.B. an den rot eingekreisten Positionen).

Es galt also herauszufinden, an welchen Stellen sich wirklich der Meeresstrand befindet, also wo die Insel ins Meer verläuft. Dies habe ich mit dem folgenden Algorithmus realisiert:

Zunächst wird eine Matrix über das gesamte Spielfeld gelegt. Diese Matrix kann in den einzelnen Indizes die folgenden 3 Werte haben: Insel, Meer oder Strand. Die komplette Matrix wird an jedem Index mit dem Wert Insel initialisiert – es wird also zunächst davon ausgegangen, dass die gesamte Karte aus einer einzigen massiven Insel besteht. Da wie in *4.1 Erzeugung der Inselplätze* bereits erklärt, jede Insel immer einen Sicherheitsabstand hat, wissen wir mit Sicherheit, dass die äußeren Indizes der Matrix mit dem Wert „Meer“ belegt werden müssen. Es ist aufgrund dieses Sicherheitsabstandes nicht möglich, dass sich an diesen Positionen eine Insel befindet.

Anschließend wird über die Randindizes iteriert und für jeden Index ein Floodfill (Asundi u. a., 1998) durchgeführt. Beim Floodfill werden rekursiv die Nachbarn jedes Index ebenfalls mit dem gewünschten Wert „Meer“ belegt unter der Voraussetzung, dass die Landschaft an dieser Stelle tief genug ist (unter dem Meeresspiegel). Nach diesen Floodfills ist also bekannt, an welchen Stellen sich Meer und an welchen Stellen sich Inseln befinden. Nun wird ein letztes

Mal über die gesamte Matrix iteriert. Handelt es sich beim iterierten Index um einen „Insel“-Index, so werden alle benachbarten „Meer“-Indizes auf „Strand“ gestellt. Dadurch sind alle potentiellen Hafenbauplätze bekannt.

Diese Matrix kann übrigens auch für das Pathfinding über A* (Brackeen u. a., 2004) verwendet werden (in dieser Thesis nicht weiter vorgestellt).

Diese potentiellen Hafenbauplätze werden nun in einer Liste gespeichert. Diese Liste wird so lange mit folgendem Algorithmus durchlaufen, bis sie leer ist oder die maximale Anzahl an Hafenbauplätzen gesetzt wurde:

1. Wähle einen zufälligen Index der Liste.
2. Platziere an dieser Position einen Hafenbauplatz.
3. Entferne alle Indizes aus der Liste, die unter einem gewissen Radius um den gewählten Index platziert sind (so wird verhindert, dass Hafenbauplätze direkt nebeneinander platziert werden).
4. Überprüfe Abbruchkriterien und gehe zu 1.

Wurde ein Hafenbauplatz platziert, so wird das Terrain anschließend noch etwas abgeflacht, d.h. auf Strandhöhe angepasst. Dadurch sind die Hafenbauplätze für den Spieler sichtbarer und die Wahrscheinlichkeit, dass sie durch einen hohen Berg verdeckt werden, wird verringert.

Abbildung 62 Die Hafenbauplätze wurden platziert.

Zusätzlich werden noch weitere Gebäudebauplätze um den Hafen herum platziert. Dabei werden zufällige Positionen in einem Kreis um den Hafenbauplatz generiert und wenn die Höhe des Terrains an diesen Positionen über dem Meeresspiegel liegt, platziert.

Der Vegetations-Algorithmus wurde noch etwas angepasst, so dass keine Bäume neben Häfen und Hafenbauplätzen platziert werden. Das bedeutet, dass der in diesem Abschnitt vorgestellte Algorithmus vor dem Vegetations-Algorithmus ausgeführt werden muss.

4.2.7 Ressourcenplatzierung

In NHP wurde auf jeder Insel eine zufällige Ressource platziert. Es wird zufällig aus den Ressourcen „Kaffee“, „Gewürze“, „Zucker“, „Silber“, „Gold“ und „Diamanten“ gewählt. Diese Ressource bestimmt, wie viel Gold die einzelnen Lagerhäuser, welche der Spieler bauen kann, produzieren. Die Ressourcen werden als ein 3D-Model dargestellt, welches zufällig auf der Insel platziert wird. Der Zufallsplatz wird folgendermaßen bestimmt:

Abbildung 63 Eine fertige Insel mit Zucker als Ressource

1. Generiere eine zufällige Koordinate.
2. Wenn die Höhe an dieser Koordinate höher ist als der Meeresspiegel, dann platziere die Ressource an dieser Koordinate und terminiere Algorithmus.
3. Sonst gehe wieder zu 1.

4.3 Landschaft mit Perlin-Noise

Die bereits erklärten Schritte wurden alle mit Value-Noise realisiert. Dies ändert sich in diesem Abschnitt. Auf diese Weise kann ich die Ergebnisse dann in *5.1. Qualität: Value-Noise- vs. Perlin-Noise-Landschaft* vergleichen und Schlüsse ziehen, welche der beiden Noise-Arten sich besser für NHP eignen. Daher werden alle Value-Noises durch Perlin-Noises ersetzt. Diese sind:

1. Höhenberechnung der einzelnen Inseln
2. Höhenberechnung des Meeresgrunds
3. Vegetationsnoise

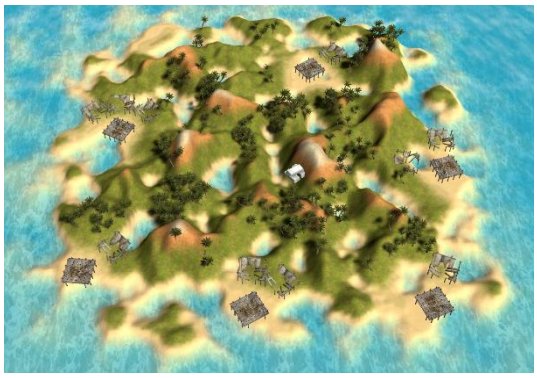

Abbildung 64 Einfache Ersetzung. Alle Value-Noises wurden zu Perlin-Noises ohne Änderung der Parameter.

Werden die einzelnen Noises ersetzt, die Parameter aber beibehalten (k, μ, α etc.), so fällt auf, dass die generierten Inseln deutlich mehr Berge und Täler haben, so wie in Abbildung 64 zu sehen. Dies liegt an der möglichen S-Form der Interpolation des Perlin-Noise. Stellt man sich den 1-dimensionalen Fall vor, so ist klar, dass die

Interpolation zwischen 2 positiven Steigungen die X-Achse durchlaufen muss. Daher entsteht eine S-Form. Die jeweilige Oktave durchläuft aber auch zwingend die jeweiligen Stützstellen. Im Schnitt hat der Perlin-Noise also 1,5-mal mehr Nullstellen als der Value-Noise.

Beheben kann man dies, indem man zum Beispiel den Parameter *k* mit $\frac{2}{3}$ multipliziert, dann kommen als Ergebnis ähnlich hügelige Landschaften heraus, wie in Abbildung 65 zu sehen. Diesen Wert bezeichne ich in den folgenden Kapiteln als Value-Faktor, da er die Ergebnisse von Perlin-Noise dem Value-Noise annähert.

Abbildung 65 Eine Perlin-Noise-Insel mit angepasstem k-Parameter

5. Evaluation

5.1. Qualität: Value-Noise- vs. Perlin-Noise-Landschaft

Bei meinen Tests hat sich herausgestellt, dass es zwischen dem Value- und dem Perlin-Noise bei der hier vorgestellten Inselerzeugung, unter der Voraussetzung, dass man für den Value-Noise die kubische Interpolation verwendet, keine sichtbaren Unterschiede gibt.

Dennoch würde ich für neue Projekte eher zum Value-Noise raten. Dies hat 3 Gründe:

1. Leichter zu implementieren
2. Keine Blockartefakte (in *2.4.4 Blockartefakte* beschrieben)
3. Mit kubischer Interpolation ist der Value-Noise schneller als der Perlin-Noise (siehe *5.2. Performance: Value-Noise vs. Perlin-Noise*).

5.2. Performance: Value-Noise vs. Perlin-Noise

Ein Vorteil des Value-Noise ist, dass er im Vergleich zum Perlin-Noise, unter Verwendung der kubischen Interpolation, nur ca. 75% der Berechnungszeit benötigt. Je nach Interpolationsfunktion kann der Value-Noise aber auch langsamer als der Perlin-Noise sein. Die trigonometrische Interpolation benötigt zum Beispiel im Vergleich zum Perlin-Noise ca. 114% der Zeit.

Die Performancetests wurden alle auf dem gleichen Laptop durchgeführt, mit einer i7-4720HQ CPU, ohne Übertaktung mit 2,60 GHz. Es wurde darauf geachtet, dass neben den Tests möglichst wenige andere Prozesse aktiv waren. Diejenigen, die aktiv waren, waren dies auch bei allen Tests. Jede Messung wurde 1024-mal wiederholt, um Ungenauigkeiten durch den Scheduler und andere Effekte zu reduzieren. Anschließend wurde der Durchschnitt der Messungen als finaler Messwert benutzt. Gemessen wurde in Unity mit C#. Alle Algorithmen wurden ohne die Benutzung von Threads implementiert. Hier soll die Performance der Algorithmen verglichen werden, der absolute Wert ist daher weniger interessant. Optimierungen mit Threads wären bei allen Algorithmen möglich und ähnlich effizient, daher wird auf solche Optimierungen bei diesem Performancecheck verzichtet.

Für diesen Performancecheck wurden Bilder mit größer werdenden Kantenlängen generiert. Verglichen wurden vier Value-Noise-Ansätze mit den Interpolationsfunktionen linear, trigonometrisch, kubisch und Sprung sowie zwei Perlin-Noise-Ansätze, einmal mit der Optimierung aus *4.3 Landschaft mit Perlin-Noise* (multiplizieren mit $\frac{2}{3}$) und einmal ohne. Die Parameter waren jeweils identisch (abgesehen von der Value-Faktor-Optimierung).

Folgende Parameter wurden verwendet:

$$bildbreite = wachsend\ in\ 10er\ Schritten\ von\ 10\ bis\ 200$$
$$bildhöhe = bildbreite$$
$$kx = \lfloor bildbreite * 0.07 \rfloor$$
$$ky = \lfloor bildhöhe * 0.07 \rfloor$$
$$Oktaven = 4$$
$$\beta = 2$$
$$\mu = 3.3$$
$$Smoothing: nein$$
$$Standardisierung: ja$$

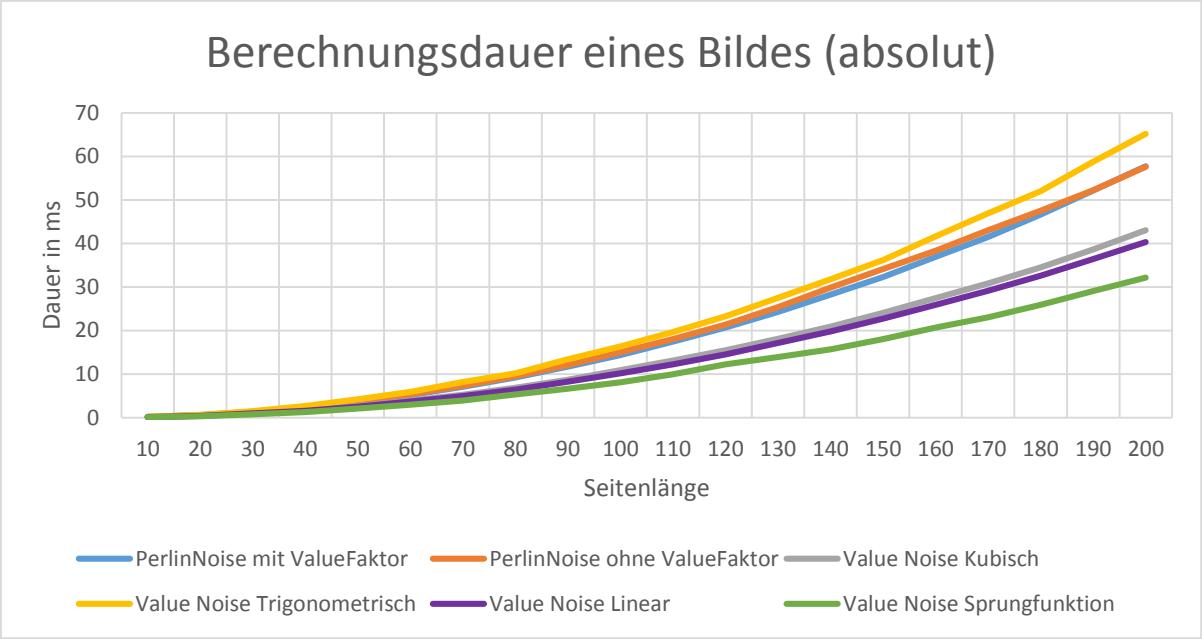

Abbildung 66 Die Ergebnisse meiner Messungen (absolut dargestellt)

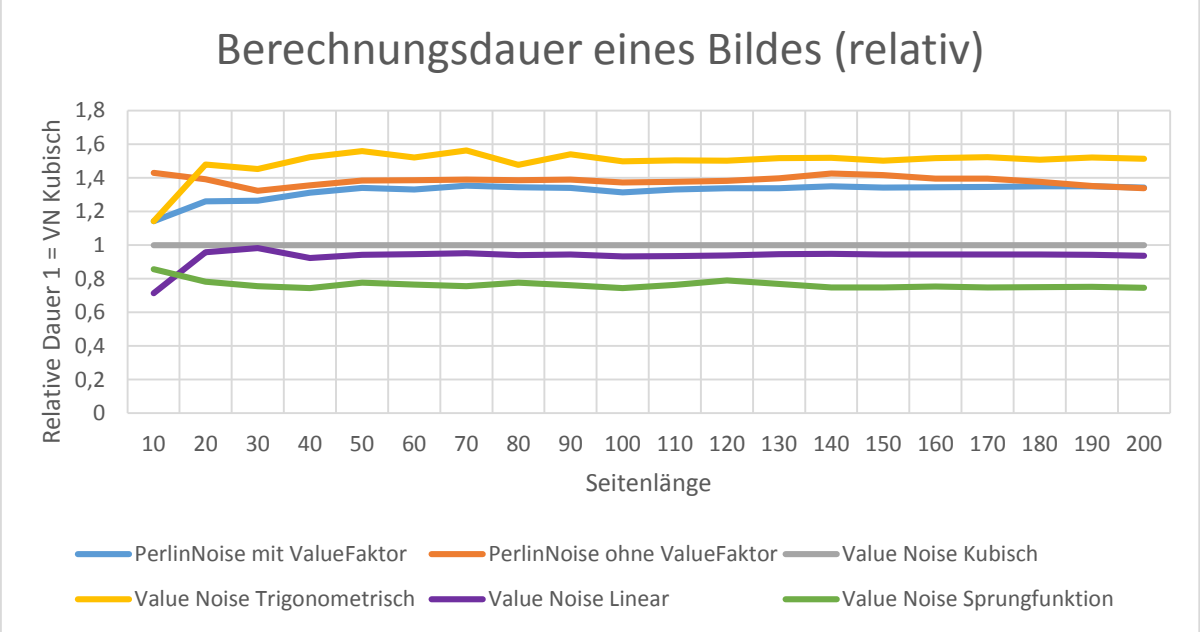

Abbildung 67 Die Ergebnisse meiner Messungen (relativ dargestellt, jede Kurve wurde durch die Ergebnisse von Value-Noise kubisch geteilt)

Die Ergebnisse dieser Tests sind in Abbildung 66 und Abbildung 67 zu sehen. Wie zu erwarten, steigen alle Kurven quadratisch an. Dies liegt daran, dass die Seitenlänge der Bilder linear steigt, die Pixel der Bilder dadurch aber quadratisch.

Da die Sprungfunktion (in Grün eingezeichnet) lediglich aus einer „if-Abfrage“ besteht, sind ihre Performancekosten auch dementsprechend gering und der Value-Noise ist mit dieser Interpolationsfunktion am schnellsten.

Obwohl in der kubischen Interpolation 9 Additionen und 5 Multiplikationen verwendet werden und in der linearen Interpolation jeweils nur 2, fällt dieser Unterschied nicht stark ins Gewicht und die Messwerte sind sehr ähnlich, wobei die lineare Interpolation etwas schneller ist.

Es folgen die beiden Perlin-Noise-Ansätze. Interessant ist, dass es zwischen ihnen keinen großen Unterschied gibt, obwohl der Ansatz mit der Optimierung (Value-Faktor) deutlich weniger Stützstellen generieren muss.

Am langsamsten war in meinen Tests die trigonometrische Interpolation.

Wie in Abbildung 67 zu sehen, verhalten sich die Kurven relativ gesehen immer ähnlich und verlaufen, abgesehen von kleinen Seitenlängen, linear (Scheduler, GC und andere Effekte fallen bei kleinen Messdauern stärker ins Gewicht).

5.3. Problematisierung: Inselform wenig kontrollierbar

Prozeduraler Content zeichnet sich dadurch aus, dass er unter gleichen Parametern (abgesehen des Seeds) ähnliche, jedoch jedes Mal unterschiedliche Ergebnisse produziert. Das bedeutet, dass zum Beispiel vor der Generierung einer Insel nicht klar ist, welche Inselform generiert wird. Daher kann die Inselform auch aus reinem Zufall die Formen von problematischer Symbolik (z.B. Hakenkreuz) annehmen oder an die menschliche Anatomie erinnern (Intimbereich).

Bei meinen Tests ist dies nur einmal bei einer fehlerhaften Perlin-Noise-Implementierung aufgetreten: Die Stützstellen hatten ausschließlich positive Gradienten. Bei korrekter Implementierung ist es nie vorgekommen (bei mehreren Tausend betrachteter Inseln). Dennoch sind solche Formen theoretisch immer möglich.

6. Fazit

Diese Thesis gibt einen kurzen Einblick und Einstieg in die prozedurale Content-Generierung und zeigt anhand eines Beispiels, wie diese zur Terrain-Generierung eingesetzt werden kann.

Kapitel 1 hat damit begonnen, zu zeigen, dass Computerspiele generell wirtschaftlich wichtig sind und in Zukunft noch weiter an Bedeutung gewinnen werden. Es wurde gezeigt, dass die Kosten zur Erstellung eines Computerspiels mit prozeduralem Content gesenkt werden können, wodurch sich der Gewinn steigert. Weiter wurden einige Algorithmen kurz genannt, welche ebenfalls zur prozeduralen Content-Erzeugung benutzt werden können, in dieser Thesis aber nicht behandelt werden. Dadurch kann der/die Leser/in bei Interesse selbst weiter recherchieren und hat nun einige Stichwörter, nach denen er/sie suchen kann.

Kapitel 2, der Stand der Forschung, konzentriert sich auf etablierte Algorithmen zur prozeduralen Content-Erstellung. Dieses Kapitel beginnt mit einer Erklärung der Interpolation zwischen diskreten Werten. Es ist wichtig, diese zu verstehen, da sie die Basis der später vorgestellten Value- und Perlin-Noises darstellt. Anschließend wurde der Unterschied zwischen Value- und Gradient-Noises erklärt. Value-Noises generieren diskrete Werte, Gradient-Noises diskrete Steigungen. Als Beispiel für Gradient-Noises wurde der Perlin-Noise erklärt. Sowohl Value- als auch Gradient-Noises wurden in mehreren Dimensionen beschrieben. Der 1D- und 2D-Fall wurde dabei ausführlich dargestellt, höherdimensionale Fälle wurden nur kurz skizziert, da diese im weiteren Verlauf dieser Thesis nicht wichtig sind, der Vollständigkeit halber aber auch nicht fehlen sollten. Am Ende von Kapitel 2 wurden zwei gängige Methoden gezeigt, um die Noises nach der eigentlichen Generierung noch zu manipulieren.

In Kapitel 3 folgen eigene (vor allem mathematische) Überlegungen über die Noises. Es wurde bewiesen, welchen maximalen und minimalen Wert beide Noises annehmen können, vorausgesetzt, man kennt die Parameter (abgesehen des Seeds). Weiter wurde die Sprungfunktion als eine Interpolationsalternative vorgestellt sowie die Ergebnisse, die durch diese entstehen können. Das dritte Kapitel schließt mit der Vorstellung eines eigenen Algorithmus ab, welcher Bilder erzeugt, die aussehen, als wären sie durch Pinselstriche gezeichnet worden.

Kapitel 4 implementiert konkret die gezeigten Algorithmen aus Kapitel 2 und benutzt diese anhand eines Spielebeispiels, um Insellandschaften zu erzeugen. Die Implementation ist in Unity vorgenommen worden. Dabei wurde auch die Vielseitigkeit der Noises gezeigt. Die Noises bestimmen nicht nur die Höhe des Terrains, sondern auch der Vegetation. Außerdem wurden die Noises nachträglich noch so manipuliert, dass Vulkane und vulkanähnliche Gebilde erzeugt werden konnten.

In Kapitel 5 folgt die Evaluation. Diese zeigt, dass es keinen sichtbaren Unterschied zwischen Value- und Perlin-Noise-Inseln gibt. Weiter wurde die Performance der unterschiedlichen Value- und Perlin-Noises untersucht. Abgeschlossen wurde die Evaluation damit, das Problem zu erläutern, dass die Ergebnisse von Value- und Gradient-Noises unvorhersehbar sind und daher die unterschiedlichsten Formen, auch ungewünschte, erzeugt werden können.

6.1 Weitere Forschungsmöglichkeiten

Beim Schreiben dieser Thesis sowie der Programme haben sich Ansätze für die weitere Forschung ergeben, welche in dieser Thesis allerdings nicht ausführlich besprochen werden, da sie zu weit vom eigentlichen Thema entfernt sind. Dennoch möchte ich einige dieser Ideen hier erwähnen.

Wie in *5.3. Problematisierung: Inselform wenig kontrollierbar* beschrieben, können zum Beispiel auch unerwünschte Inselformen entstehen. Eine Möglichkeit, dies zu verhindern, wäre die Mustererkennung mithilfe eines neuronalen Netzes (Rey u. a., 2011). Wurde eine Insel generiert, so könnte ein neuronales Netz diese untersuchen und entweder die Insel als unbedenklich oder problematisch kennzeichnen. Die Insel könnte, falls ein ungewünschtes Zeichen entdeckt wurde, verändert oder verworfen werden.

In *4.2.1 Strand* wurde eine Maske generiert, um die Inselränder langsam ins Meer verschwinden zu lassen. Zwar funktioniert der gezeigte Ansatz, jedoch ist die grundlegende Form viereckig. Hier sind weitere Tests notwendig, um dies zu verändern.

Die Vegetation wurde nach *4.2.4 Vegetation* generiert. Die dort vorgestellten Ansätze ignorieren die zugrundeliegende Steigung des Terrains. Das führt dazu, dass auch an steilen Stellen Wälder platziert werden können. Dies kann zwar auch in der Realität vorkommen, jedoch mit geringerer Wahrscheinlichkeit. Daher wäre eine Verbesserung möglich, indem die Steigung den Zufallswert auf irgendeine, noch zu untersuchende, Weise mit beeinflusst.

In *5.2. Performance: Value-Noise vs. Perlin-Noise* wurde gezeigt, dass die trigonometrische Interpolation im Vergleich zu anderen Interpolationsfunktionen sehr langsam ist. Da die Ergebnisse für die in dieser Thesis vorgestellte Inselerzeugung sowieso nicht zufriedenstellend waren (zu sehen in Abbildung 47) habe ich dies nicht weiter untersucht. Dennoch wäre eine schnellere Variante über eine Sinus-Approximation möglich, also eine Funktion, die den Sinus nicht exakt, dafür aber schneller ausrechnet. Eine Möglichkeit dafür wäre, nur einen Teil der Taylor-Folge (Arens u. a., 2012) zu verwenden, zum Beispiel:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

Formel 23 Teil der Taylor-Folge für die Berechnung des Sinus

Eine andere Möglichkeit ist es, die Sinuswerte in einer Tabelle im Voraus zu berechnen (z.B. 10.000 unterschiedliche Werte). Dafür benötigt man allerdings auch mehr Speicher, in diesem Fall 40.000 Byte (unter der Annahme, dass ein Wert 4 Byte belegt, was eine typische Größe für Fließkommazahlen ist) und eine längere Startzeit.

7. Verzeichnisse

7.1 Literaturverzeichnis

Arens, T., Hettlich, F., Karpfinger, Ch., Kockelkorn, U., Lichtenegger K., & Stachel, H. (2012). *Mathematik* (2. Auflage). Heidelberg: Spektrum Akademischer Verlag. ISBN-13: 978-3827423474

Asundi, A., & Wensen, Z. (1998). Fast phase-unwrapping algorithm based on a gray-scale mask and flood fill. *Applied Optics, 37*(23), 5416-5420. http://dx.doi.org/10.1364/AO.37.005416

Barnard, R., & Dr. Ural, S. (2005). Rendering translucency with Perlin noise. In S. N. Spencer (Ed.), *GRAPHITE 2005: Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, Dunedin, New Zealand, November 29-December 2, 2005* (pp. 131-134). New York: ACM. http://dx.doi.org/10.1145/1101389.1101414

Bergen, G. v. d. (1997). Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools, 2*(4), 1-13. doi:10.1080/10867651.1997.10487480

Beschorner, M., Giermann, O., Gralak, J., Kopp, S., & Staiger, U. (2016). *Traumfabrik Photoshop: Faszinierende Artworks, außergewöhnliche Composings*. Bonn: Rheinwerk. ISBN-13: 978-3836238564

BIU, Bundesverband Interaktive Unterhaltungssoftware (2016). *Deutscher Gesamtmarkt für digitale Spiele im ersten Halbjahr 2015* [Grafik]. Abgerufen am 02.02.2016 unter http://www.biu-online.de/de/fakten/marktzahlen-1-halbjahr-2015/kauf-umsatz-digitale-spiele.html

Bourke, P. (1999). *Interpolation methods.* Abgerufen am 02.02.2016 unter http://paulbourke.net/miscellaneous/interpolation/

Brackeen, D., Barker, B., & Vanhelsuwé, L. (2004). *Developing Games in Java*. Berkeley, CA: New Riders. ISBN-13: 978-1592730056

Burger, W. (2008). *Gradientenbasierte Rauschfunktionen und Perlin Noise*. Technischer Bericht HGBTR08-0220, FH-Oberösterreich, Campus Hagenberg. http://dx.doi.org/10.13140/RG.2.1.2641.2967

Distelmeyer, J. (2007). Spielräume, Videospiel, Kino und die intermediale Architektur der Film-DVD. In R. Leschke & J. Venus (Hrsg.), *Spielformen im Spielfilm: zur*

Medienmorphologie des Kinos nach der Postmoderne (S. 389 - 416). Bielefeld: transcript. ISBN-13: 978-3899426670

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (2003). *Texturing & Modeling: A Procedural Approach* (3rd ed.). San Francisco: Morgan Kaufmann. ISBN-13: 978-1558608481

Golem (2014). *46 Prozent der Deutschen spielen Computer- oder Videogames*. Abgerufen am 07.02.2016 unter http://www.golem.de/news/biu-46-prozent-aller-deutschen-greifen-zu-games-1403-105025.html

Hello Games (2015). *No Man's Sky.* Abgerufen am 07.02.2016 unter http://www.no-mans-sky.com/about/

Hendrikx, M., Meijer, S., Velden, J. v. d., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 9*(1), Article No. 1. http://doi.acm.org/10.1145/2422956.2422957

Huijser, R., Dobbe, J., Bronsvoort, W. F., & Bidarra, R. (2010). Procedural Natural Systems for Game Level Design. In IEEE Computer Society, & Institute of Electrical and Electronics Engineers, Inc. (Eds.), *2010 Brazilian Symposium on Games and Digital Entertainment: SBGames 2010, 8-10 November 2010, Florianópolis, Santa Catarina, Brazil,* (pp. 189-198). Los Alamitos, CA: IEEE Computer Society Conference Publishing Services (CPS). http://dx.doi.org/10.1109/SBGAMES.2010.31

IncGamers Ltd (2014). *Whimsyshire.* Abgerufen am 07.02.2016 unter http://www.diablowiki.net/Whimsyshire

Irish, D. (2005). *The Game Producer's Handbook*. Boston, MA: Thomson Course Technology. ISBN-13: 978-1592006175

Kuan, D. T., Sawchuk, A. A., Strand, T. C., & Chavel, P. (2009). Adaptive Noise Smoothing Filter for Images with Signal-Dependent Noise. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-7*(2), 165-177. http://dx.doi.org/10.1109/TPAMI.1985.4767641

Lagea, A., Lefebvre, S., Drettakis, G., & Dutré, P. (2009). Procedural Noise using Sparse Gabor Convolution. In H. Hoppe (Ed.), *ACM SIGGRAPH 2009 papers,* (Article No. 54). New York: ACM. http://dx.doi.org/10.1145/1576246.1531360

Lewis, J. P. (1989). Algorithms for Solid Noise Synthesis. In R. J. Beach (Ed.), *SIGGRAPH '89 Proceedings of the 16th annual conference on Computer graphics and interactive techniques,* (pp. 263-270). New York: ACM. http://dx.doi.org/10.1145/74334.74360

Linden, R. v. d., Lopes, R., & Bidarra R. (2014). Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games, 6*(1), 78-89. http://dx.doi.org/10.1109/TCIAIG.2013.2290371

Mandelbrot, B. B. (1991). *Die fraktale Geometrie der Natur*. Basel: Birkhäuser. ISBN-13: 978-3034850285

Michelon de Carli, D., Pozzer, C. T., Bevilacqua, F., & Schetinger, V. (2014). Procedural generation of 3D canyons. In IEEE Computer Society, & Institute of Electrical and Electronics Engineers, Inc. (Eds.), *2014 27th SIBGRAPI Conference on Graphics, Patterns and Images: SIBGRAPI 2014, Rio de Janeiro, Brazil 27-30 August 2014* (pp. 103-110). Los Alamitos, CA: IEEE Computer Society Conference Publishing Services (CPS). http://dx.doi.org/10.1109/SIBGRAPI.2014.41

Mojang Synergies AB (2016). *Minecraft*. Abgerufen am 02.02.2016 unter https://minecraft.net

Perlin, K. (1999). *Making Noise*. Abgerufen am 02.02.2016 unter http://www.noisemachine.com/talk1/

Perlin, K. (2001). *Noise Hardware*, Abgerufen am 02.02.2016 unter http://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf

Rey, G. D., & Wender, K. F. (2011). Neuronale *Netze: Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung* (2. Auflage). Bern: Verlag Hans Huber. ISBN-13: 978-3456848815

Roberts, J., & Ke Chen (2015). Learning-Based Procedural Content Generation. *IEEE Transactions on Computational Intelligence and AI in Games, 7*(1), 88-101. http://dx.doi.org/10.1109/TCIAIG.2014.2335273

Salomon, D. (2006). *Curves and Surfaces for Computer Graphics.* New York: Springer Science+Business Media, Inc. ISBN-13: 978-0387241968

Tatarchuk, N. (n.d.). *The Importance of Being Noisy: Fast, High Quality Noise*. Abgerufen am 02.02.2016 unter http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Tatarchuk-Noise%28GDC07-D3D_Day%29.pdf

Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). What is Procedural Content Generation?: Mario on the borderline. In *PCGames '11: Proceedings of the*

2nd International Workshop on Procedural Content Generation in Games, (Article No. 3). New York: ACM. http://dx.doi.org/10.1145/2000919.2000922

Unity Technologies (2016). *Unity.* Abgerufen am 02.02.2016 unter http://unity3d.com/

Wagner, R. (n.d.). *Multi-Linear Interpolation*. Abgerufen am 02.02.2016 unter http://rjwagner49.com/Mathematics/Interpolation.pdf

Yannakakis, G. N., & Togelius, J. (2011). Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing, 2*(3), 147-161. http://dx.doi.org/10.1109/T-AFFC.2011.6

7.2 Abbildungsverzeichnis

7.3 Formelverzeichnis