Technische-Hochschule Nürnberg

Fakultät Informatik

Masterstudiengang Medieninformatik (M. Sc.)

Studienarbeit für das Fach

GC2 – Programmieren von Grafik-Shadern

Mathias K r ä m e r

Matr.-Nr. 3000781

Noise

WiSe 2020/2021

1 Inhalt

2 Motivation

Bis zum neunten Kapitel von David Wolffs Buch: „OpenGL 4 Shading Language Cookbook“, welche in dieser Studienarbeit thematisiert werden, wurden bereits einige Grundlagen gelehrt und veranschaulicht um realistische Szenen in OpenGL zu erstellen: Dreidimensionale Objekte können, in eigens kreierten Welten, mit natürlich wirkenden Lichtern und Schatten dargestellt werden. Wie die Oberflächen dieser Objekte gestaltet sind bestimmen Texturen, die aus Fotografien extrahiert sind.

Ein Fallbeispiel zeigt den Nutzen und die Notwendigkeit eines neuen Werkzeugs, welches im Weiteren behandelt wird – Noise.

Angenommen es soll eine Szene erstellt werden, die über bloße Beispiele mit geometrischen Grundformen hinausgeht, beispielsweiße ein Wald. Unter Beachtung des jetzigen Wissenstandes ergeben sich mehrere Schwierigkeiten:

Für eine realistische Wiedergabe wäre es von Vorteil, wenn die verschiedenen Bäume in der Waldszene sich, wie in der Natur, im Aussehen unterscheiden würden. Mehrere Variationen einer Stammtextur zu haben, würden jedoch auch heißen, dass viele hochauflösende Texturen gespeichert und übertragen werden müssen. Außerdem wäre es sehr aufwändig, jeden einzelnen dieser Stämme per Hand in der Szene zu platzieren. Weitere Gedanken beziehen sich auf den Typ der Landschaft des Waldes, der durchaus hügelig sein könnte. Schließlich soll es auch möglich sein durch die Äste einen Blick auf den Himmel zu werfen, der, mit einer Skybox versehen, zwar einen Himmel realistisch nachbilden kann, der jedoch so natürliche Phänomene wie bewegliche Wolken missen muss.

Eine Lösung dieser Problematiken, wäre es den Zufall bei der Entstehung solcher Szenen mitwirken lassen zu können, um z.B. die Verteilungen der Objekte oder das Aussehen von Oberflächen natürlicher gestalten zu können. Noise, oder auf Deutsch Rauschen, ist ein solches Mittel und bildet ein Basiswerkzeug in der Computergrafik.

3 Zufälliges Rauschen

Wie im vorherigen Beispiel gesehen wird eine spezielle Art von Zufallsgenerator benötigt, der dann im Entstehungsprozess der Szene eingegliedert wird. Komplett zufälliges Noise, optisch vergleichbar mit dem statischen Rauschen eines analogen Fernsehers, findet zwar auch seine Anwendung in der Computergrafik, jedoch nicht für dieses Anwendungsgebiet. Es müssen wiederholbare Daten erzeugt werden, die mit den gleichen Eingangsparametern immer dieselben Ergebnisse produzieren, damit Objekte in jedem Frame gleich gerendert werden können, was besonders wichtig für Animationen ist.[1]

Abbildung 1: Texturbeispiele aus der Natur: Rost, Holz, Marmor und Wolken
(Quelle: keine Attribution erforderlich / pexels.com)

Ein Blick auf natürliche Oberflächen, wie in Abbildung 1 zu sehen, lässt vermuten, dass trotz der Zufälligkeit dennoch auch eine gewisse Ordnung notwendig ist, um sie korrekt nachzubilden. So finden sich z.B. sowohl längere als auch kürzere Farbübergange zwischen verschiedenen Holzphasen, von der Basis zum höchsten Punkt einer Wolke oder bei der Marmorisierung von Gesteinen, die uns zufällig verteilt aber dennoch harmonisch erscheinen.

4 Prozedural kohärentes Rauschen

4.1 Ken Perlin

Ähnlicher Beobachtungen könnte auch Kenneth H. Perlin, Informatikprofessor an der New York University, gehabt haben. 1982 suchte er für sein Filmprojekt mit Disney „Tron“ eine neue Lösung für computer-generierten Grafiken in der Filmindustrie, die bis dato einen sehr maschinellen und unnatürlicher Charakter hatten. Mit dem von ihm erfundenen Perlin Noise, gelang es ihm mathematisch ein Muster zu definieren, mit dem pseudo-zufällige Punkte mit natürlich wirkenden Übergängen erzeugt werden können. Für diesen wesentliche Beitrag für die Computergrafik und Filmindustrie wurde Ken Perlin 1997 mit dem Academy Award für technische Leistungen ausgezeichnet.[2]

4.2 Perlin Noise Überblick

Perlin Noise (siehe Abbildung 2) erzeugt ein anscheinend zufälliges Muster, jedoch mit weichen Übergängen, denn es ist eine kontinuierliche Funktion, die überall differenzierbar ist, d.h. alle Werte stehen mit vorherigen und folgenden Werten im Zusammenhang.[1] Perlin Noise kann in n-Dimensionen erzeugt werden, wobei die 2D und 3D Varianten am geläufigsten sind.

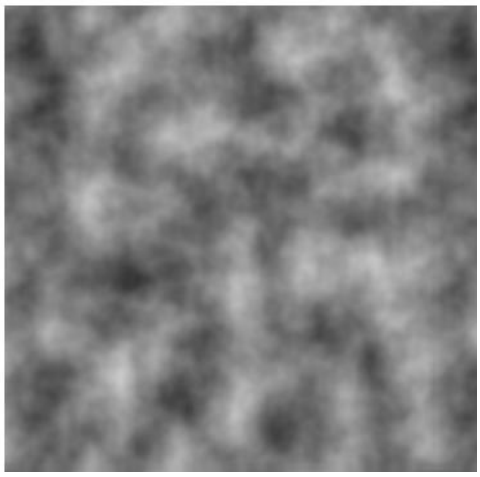

Abbildung 2: Perlin Noise
(Quelle: eigene Anfertigung auf Basis des Codes von David Wolff)

In der Computergrafik findet Perlin Noise vor allem Anwendung beim Erstellen von prozeduralen Texturen. Prozedurale Texturen benötigen kein Bild als Quelle, was zugunsten von Speicherplatz und Übertragungsgeschwindigkeit geht. Zusätzlich ist es nicht unbedingt notwendig solche Texturen zu mappen, da sie auch direkt auf 3D-Objekte appliziert werden können.[2]

4.3 Perlin Noise Algorithmus

Perlin Noise ist ein sogenanntes Gradientenrauschen (Englisch: gradient noise), denn es werden natürlich wirkende Übergänge mithilfe von pseudo-zufälligen Gradienten erzeugt.[3]

Da im Beispiel von David Wolff eine Texture Map in GLSL erzeugt wird, die 2D Perlin Noise beinhält, wird im Folgenden der Algorithmus für zweidimensionales Perlin Noise schematisch erklärt:

Im Wesentlichen besteht der Vorgang aus diesen drei Schritten: (1) Gitter-Definition und Erzeugung von pseudo-zufälligen Gradienten, (2) Berechnung der Skalarprodukte zwischen Gradienten und deren Versatz, (3) Interpolation der Werte mit Überblendungskurve.[4] Folgende Erklärung basiert nicht auf Perlins originaler Noise Funktion, sondern auf einer von ihm bereits verbesserten Variante aus dem Jahr 2002.[5]

4.3.1 Gitter-Definition und Erzeugung von pseudo-zufälligen Gradienten

Zu Beginn wird ein zweidimensionales Gitter erzeugt. Gewöhnliches Perlin Noise besteht normalerweise aus einem Gitter mit Quadraten und ganzzahligen Gitterpunkten. Höhe und Breite dieses Gitters sind Parameter, die unter anderem das Aussehen des erzeugten Musters bestimmen. Es ist auch möglich ein unterschiedlich proportioniertes Gitter zu verwenden welches aus rechteckigen Elementen besteht.

An jedem Gitterpunkt wird ein 2D-Gradientenvektor erzeugt. Da diese Gradienten nicht zufällig erstellt sind, sondern Permutationen aus einem vorher angelegten Nummern-array sind, sprechen wir von pseudo-zufälligen Gradienten. Dies ist wichtig, damit der Algorithmus wiederholbar ist und mit demselben Input immer derselbe Output generiert werden kann. Laut

Gustavson ist es ratsam für 2D Perlin Noise 8 oder 16 im Einheitskreis liegende, gleichmäßig verteilte und normalisierte Gradienten zu wählen. Da die Wahl der Gradienten ebenfalls einen optischen Einfluss auf das Ergebnis hat, ist es auch möglich andere Gradientenvektoren zu benutzen, jedoch sollte darauf geachtet werden, dass diese eine gleichmäßige Verteilung haben.[4] Abbildung 3 zeigt diesen ersten Schritt des Perlin Noise Algorithmus.

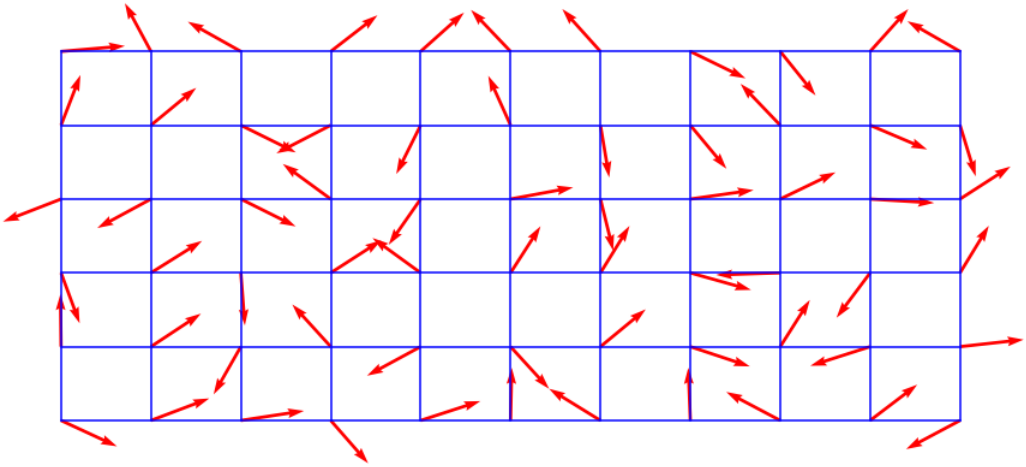

Abbildung 3: Gitter mit pseudo-zufälligen Gradienten visualisiert
(Quelle: Matthewslf, CC BY-SA 4.0 / https://commons.wikimedia.org/w/index.php?curid=77692522)

4.3.2 Berechnung der Skalarprodukte zwischen Gradienten und Versatzvektoren

Um den Rauschwert für einen Punkt P im Gitter zu errechnen, müssen zuerst die vier nächstgelegene Gitterpunkte und deren Gradientenvektoren identifiziert werden.

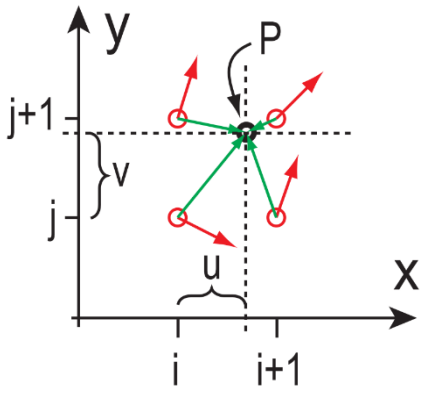

Abbildung 4: Punkt P im Gitter mit seinen entsprechenden Gradienten und Vektoren
(Quelle: Stefan Gustavson, Simplex noise demystified / http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf)

Mit jedem dieser vier in Abbildung 4 rot eingezeichneten Gradientenvektoren und einem zusätzlichen, hier grünen, Versatzvektor muss das Skalarprodukt berechnet werden. Die Versatzvektoren bestehen aus den *x*- und *y*-Abständen der Eckpunkte zum Punkt *P*, hier als *u* und *v* bezeichnet. Dieses Skalarprodukt gibt Auskunft über den Winkel zwischen den beiden Vektoren. Es ist Null, wenn Gradient und Offsetvektor orthogonal sind, wie z.B. auch wenn der Punkt *P* genau in der Gitterecke liegt.

4.3.3 Interpolation der Werte mit Überblendungskurve

Um einen gleichmäßigen optischen Verlauf der unterschiedlichen Gradienten zu gewährleisten müssen im letzten Schritt die Eckpunkte, die nun als Skalare vorliegen, genutzt werden um eine Interpolation zum Punkt *P* zu berechnen, wobei die Überlagerungskurve fünfter Ordnung zur Berechnung verwendet wird:

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

Es wird zuerst zwischen zwei Paaren interpoliert (siehe blaue Pfeile in Abbildung 5), um mit den beiden Ergebnissen auch die andere Richtung berechnen zu können (gelbe Pfeile).

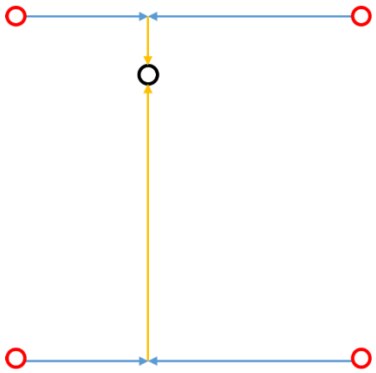

Abbildung 5: Veranschaulichung der Interpolation der Eckpunkte
(Quelle: eigene Anfertigung)

Dieser Wert kann nun genutzt werden, um den Farbwert des Pixels festzulegen, welches im Punkt *P* liegt. Für jeden Pixel oder Texel muss das Perlin Noise einzeln ausgerechnet werden, also das Skalarprodukt der Positions- und Gradientenvektoren an jedem Knoten der enthaltenen Gitterzelle ausgewertet werden. Perlin-Rauschen skaliert daher mit der Komplexität *O(2^n)* für n-Dimensionen.[4]

4.4 Fractal Noise

In der Praxis kommt Perlin Noise nicht direkt so zum Einsatz, wie es im letzten Kapitel gezeigt wurde. Um mehr „hochfrequente Variationen“, also feinere Details zu erzeugen, wie sie bei der Verwendung in Texturen oft benötigt werden, werden mehrere Perlin Noise Ergebnisse mit „zunehmende Frequenzen und abnehmende Amplituden“ addiert.[1] Das Ergebnis wird dann als Fractal Noise bezeichnet.

Wenn die Perlin Noise Funktion als Formel geschrieben *P(x, y)* lautet, dann kann man verschiedene Frequenzen mit folgender Formel beschreiben:

$$n_i(x, y) = P(2^i x, 2^i y)$$

Eine Verdopplung der Frequenz *i* wird auch als eine Oktave bezeichnet. Perlin Noise mit einer, zwei, drei und vier Oktaven ist in Abbildung 6 zu sehen.

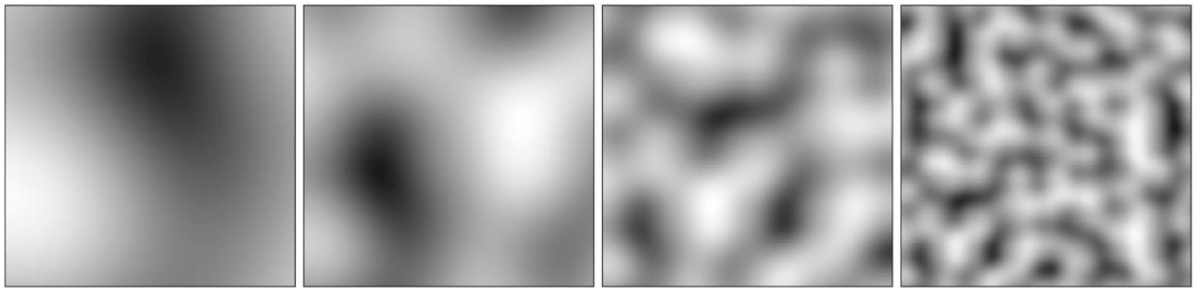

Abbildung 6: Perlin Noise mit 1, 2, 3 und 4 Oktaven, von links nach rechts
(Quelle: David Wolff, OpenGL 4 Shading Language Cookbook / https://static.packt-cdn.com/downloads/9781789342253_ColorImages.pdf)

Für Fractal Noise werden mehrere Oktaven von Perlin Noise übereinandergelegt, wie im Folgenden zu sehen:

$$n(x,y) = \sum_{i=0}^{N-1} \frac{P(2^i ax, 2^i ay)}{b^i}$$

Inwiefern sich die Parameter *i, a* und *b* optisch auf das Resultat auswirken, soll im beiliegenden Codebeispiel gezeigt werden.

5 Beispiele von David Wolff zum Thema „Noise“

Um Perlin Noise in einem Shader in OpenGL zu verwenden gibt es drei Möglichkeiten: Entweder man benutzt die in GLSL (oder genauer in GLM) integrierten Noise Funktionen wie:

```
T glm::perlin(vecType< T > const & p)
T glm::perlin(vecType< T > const & p, vecType< T > const & rep)
```

Natürlich kann man auch eine eigens in GLSL erstellte Noise Funktion nutzen. Letztendlich gibt es auch die Möglichkeit eine Texture Map zu benutzen, in der Werte enthalten sind, die vorher einmalig berechnet wurden. Dieses Vorgehen nutzt auch David Wolff in seinem Beispiel in Kapitel 9, indem ein Fractal Noise für eine Textur erzeugt wird, welches aus vier Oktaven besteht. Die verschiedenen Oktaven werden in einem RGBA Bild gespeichert, je Oktave in einem Farbkanal und dem Alphakanal, also Noise mit einer Oktave im roten Kanal, mit zweien im Grünen, mit drei im Blauen und mit vier im Alphakanal.[1]

5.1 Code: Scenenoise.cpp

Anhand der Beispielszene Scenenoise.cpp von David Wolff und einer eigenen Modifikation, soll näher darauf eingegangen werden, wie Perlin Noise in GLSL erzeugt und benutzt werden kann.

Die Hauptszene Scenenoise enthält ein simples Quad, welches mit einer Textur versehen ist und angezeigt wird. Der Aufruf, die Textur zu erzeugen und sie in die OpenGL Textur zu laden, wurde in die Update Methode verschoben, um bei Parameteränderungen direkt angezeigt zu werden. Mit der kostenlos zugänglichen Bibliothek „Dear ImGui“ wurde eine GUI für die Einstellung verschiedene Parameter zur Verfügung gestellt, die Aufschluss über deren Wirkung für das erzeugte Perlin Noise geben.[6] Dies wird über die Methode „renderGUI“ implementiert. Man kann die Frequenz, Persistenz, Breite und Höhe des Gitters (in einem brauchbaren Bereich) frei wählbar einstellen.

Die folgende Zeile ruft die Noisetex.cpp auf, in der die Textur mithilfe des Perlin Noise Algorithmus erstellt wird.

```
GLuint noiseTex = NoiseTex::generatePeriodic2DTex(uiParams.UiBaseFreq, uiParams.UiPersist, uiParams.UiWidth, uiParams.UiHeight);
```

Ohne den Zusatz „Periodic“ zur generate2DTex entsteht ein Perlin Noise, welches nicht periodisch, also nicht kachelbar ist. Hier ist zu beachten, dass die Frequenz aufgrund der Art wie sie berechnet wird, nur ganzzahlig gewählt werden kann.

5.2 Code: Noisetex.cpp

Statt, wie in früheren Kapiteln eine Textur von einem Bild in ein Texturobjekt zu überführen, wird in der Noisetex.cpp selbst eine Textur erstellt. Dazu wird zuerst Speicher für die Noise Werte alloziiert.

Über die Aufteilung der Breite und der Höhe wird in einer geschachtelten for-Schleife für jedes Texel einzeln pro Oktave (hier vier) die Perlin Noise Funktion aufgerufen, die einen Float zwischen -1 und 1 zurückgibt:

```
val = glm::perlin(p, glm::vec2(freq)) * persist;
```

Die Berechnung des Perlin Noise findet in der GLM Funktion „noise.inl“ statt und beinhält sämtliche Berechnung wie anfangs schematisch gezeigt. Der Eingabeparameter p besteht aus einem Vector2 der jeweils die x- und y-Werte mit der Frequenz multipliziert enthält:

```
glm::vec2 p(x * freq, y * freq);
```

Durch das Überladen der Perlin Funktion mit einem zweiten Vektor, der abermals die Frequenz beinhält, wird ein periodisches Perlin Noise erzeugt. Hier wird auch der Parameter „Persist“ eingeführt, dessen Erhöhung zu einer Wirkung ähnlich einer Kontrasterhöhung führt - helle Bereiche werden heller und dunkle Bereiche dunkler, während Grauwerte zurücktreten.

Dann wird die Summe „sum“ der vorherigen Gleichung erstellt, in deren ersten Komponent die erste Gleichung, im zweiten Komponent die ersten zwei, etc. gespeichert werden. Durch die Multiplikation mit 255 wird ein Grauwert erzeugt, welcher in ein Byte gewandelt wird:

```
data[((row * width + col) * 4) + oct] = (GLubyte) ( result * 255.0f );
```

Die Oktaven-Schleife wird beendet, indem Frequenz für die nächste Iteration verdoppelt wird. Durch die storeTex-Methode, wird die Textur gespeichert, wie bereits im vorangehenden Texturen-Kapitel bereits gesehen:

```
int texID = NoiseTex::storeTex(data, width, height);
```

Die Scenenoise.cpp und alle weiteren benötigten Code Dateien liegen in einer vollständig kommentierten Version im entsprechenden Git Repository vor.

5.3 Weitere Beispiele von David Wolff

In weiteren Beispielen von David Wolff ist zu sehen, wie die Perlin Noise Funktion als Grundlage für z.B. Wolkenhimmel-, Holz-, Erosions- und Rosttextur genutzt werden kann. Diese können über die Strings „sky“, „wood“, „decay“, „rust“ aufgerufen werden.

6 Einsatz verschiedener Arten von Noise in der Praxis

6.1 Limitationen von Perlin Noise

Im Jahr 2002 hat Perlin seinen Algorithmus verbessert. Das neue „Improved Perlin Noise” enthielt unter anderem bessere Lösungen für die Intepolationsfunktion und entfernte Artefakte bei Integerwerten.[5] Die Richtungsartefakten, wie Sie im Codebeispiel vor allem in höheren Oktaven deutlich zu sehen sind, sind davon nicht betroffen. Ein Lösungsansatz diese Richtungsartefakte zu entfernen bietet z.B. User samgak, der/die/das vorschlägt, jede Oktave des Fractal Noises um eine zufällige Gradzahl zu drehen, bevor diese addiert werden.[7]

Eine weitere Verbesserung von Perlin, das sogenannte Simplex Noise löst diese Artefakte zwar, jedoch steht die Benutzung unter Lizenz. Deswegen kann die oben erwähnte Verbesserung von Perlin Noise dennoch nützlich sein. Unabhängige Entwickler, die dennoch nicht auf Simplex Noise verzichten möchten, können etwaige Patentprobleme umgehen indem sie z.B das von KdotJPG entwickelte Open Simplex Noise benutzen, welches sehr ähnliche Ergebnisse liefert.[8] Weitere frei zugängliche Bibliotheken wie z.B. Libnoise können auch genutzt werden, in denen oft, die mittlerweile entdeckten Verbesserungen bereits integriert sind.[9]

6.2 Simplex Noise

Simplex Noise ist eine verbesserte Version von Perlin Noise durch Perlin selbst, welches aus equilateralen Dreiecken besteht. Es zeichnet sich vor allem durch eine sehr viel schnellere Berechnung aus, die vor allem in höheren Dimensionen spürbar ist, da die Komplexität statt der *O(2^n)* von Perlin Noise *O(n)* beträgt. Auch die Lösung des Problems von Richtungsartefakte ist in Simplex Noise bereits implementiert. Simplex Noise ist ebenfalls in der GLM Bibliothek enthalten, jedoch nicht in einer periodischen Variante.[10] Das bereits vorher erwähnte Open Simplex Noise enthält jedoch eine kachelbare Version.[11]

6.3 Weitere Arten von Noise

Trotz seines großen Beitrags zur Computergrafik gehen nicht alle Arten von Noise auf Erfindungen Perlins zurück. Es gibt weiterhin andere Arten von Noise, die für andere Einsatzgebiete benutzt werden können. Ein Beispiel wäre das sog. „Worley Noise“, welches häufig für zelluläre Texturen oder für Tierschuppen Verwendung findet.[12] Fertige Implementierung von Rauscharten werden oft im Internet zur freien Verfügung bereitgestellt, z.B. auf Github [13], und können ähnlich wie das genannte Perlin Noise genutzt werden. Hierbei sollten jedoch Lizenz- oder Patentrechte beachtet und selbst recherchiert werden, bevor sie für, vor allem für kommerzielle, Projekte genutzt werden.

6.4 Prozedurale Terrain Generation

Noise und vor allem Perlin Noise stellen ein Grundwerkzeug in der Computergrafik dar. Neben der Erstellung von prozeduralen Texturen kann mit Rauschen auch Terrain für Computerspielwelten und Szenen prozedural erstellt werden.

Das erste und zweite Bild in Abbildung 7 zeigt die Erstellung zweier 2D Simplex Noise Heightmaps. Die Grauwerte der einzelnen Pixel aus dem Ersteren werden als Höhenangaben einer Landmasse interpretiert und bilden die hier sog. Island Heightmap. Kombiniert mit dem zweiten Simplex Noise, der Moisture Heightmap, kann nun errechnet werden ob entsprechende Pixel der Landschaft See/Ozean, Strand, Gras-, Bergland, oder schneebedeckte Gipfel darstellen sollen.

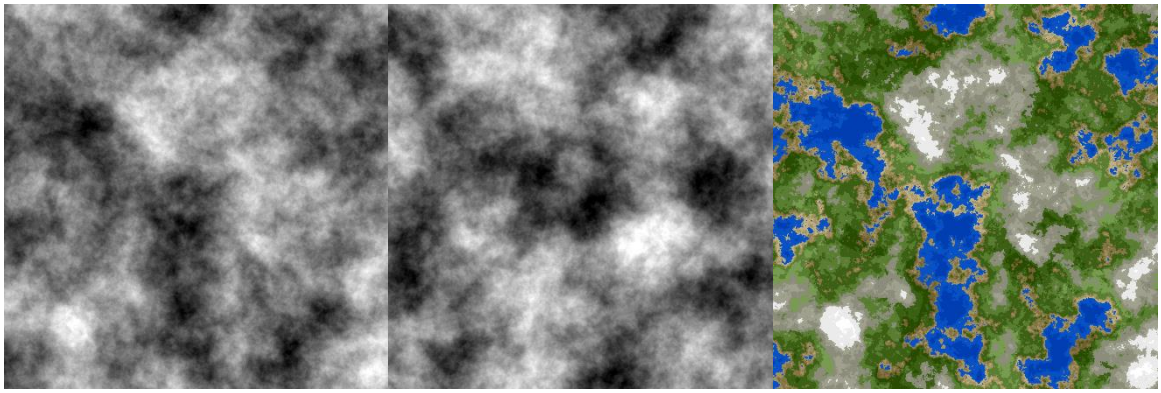

Abbildung 7: Island Heightmap, Moisture Heightmap und das Ergebnis, eine natürliche Landschaft in Form einer Reliefkarte (Quelle: Travall, Procedural 2D Island Generation — Noise Functions / https://medium.com/@travall/procedural-2d-island-generation-noise-functions-13976bddeaf9)

So oder nach sehr ähnlichem Vorgang funktioniert auch die prozedurale Generierung von Landschaften für moderne Computerspiele, wie z.B. Minecraft.

Literatur

[1] D. Wolff, OpenGL 4 Shading Language Cookbook: Build High-Quality, Real-time 3D Graphics with OpenGL 4. 6, GLSL 4. 6 and C++17, 3rd Edition, 3. Aufl. Birmingham: Packt Publishing Ltd, 2018. [Online]. Verfügbar unter: https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=5532271

[2] K. Perlin, *Making Noise: Archived from the original.* [Online]. Verfügbar unter: https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/ (Zugriff am: 2. Dezember 2020).

[3] D. S. Ebert, *Texturing & modeling: A procedural approach*, 3. Aufl. San Francisco, Calif: Morgan Kaufmann, 2003. [Online]. Verfügbar unter: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=81844

[4] S. Gustavson, *Simplex noise demystified.* [Online]. Verfügbar unter: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (Zugriff am: 6. Dezember 2020).

[5] K. Perlin, „Improving noise“, *ACM Trans. Graph.*, Jg. 21, Nr. 3, S. 681–682, 2002, doi: 10.1145/566654.566636.

[6] GitHub, *ocornut/imgui.* [Online]. Verfügbar unter: https://github.com/ocornut/imgui (Zugriff am: 27. Dezember 2020).

[7] karl88, *Fix directional artifacts generated by Perlin noise with another algorithm.* [Online]. Verfügbar unter: https://stackoverflow.com/questions/38536361/fix-directional-artifacts-generated-by-perlin-noise-with-another-algorithm (Zugriff am: 27. Dezember 2020).

[8] Gist, Visually isotropic coherent noise algorithm based on the simplectic honeycomb (Zugriff am: 25. Dezember 2020).

[9] *libnoise: Tutorial 1: Including the libnoise library.* [Online]. Verfügbar unter: http://libnoise.sourceforge.net/tutorials/tutorial1.html (Zugriff am: 25. Dezember 2020).

[10] *GLM_GTC_noise.* [Online]. Verfügbar unter: https://glm.g-truc.net/0.9.4/api/a00152.html (Zugriff am: 27. Dezember 2020).

[11] Gist, *Tileable 3D OpenSimplex Noise* (Zugriff am: 25. Dezember 2020).

[12] P. Cozzi und C. Riccio, Hg., *OpenGL insights: OpenGL, OpenGL ES, and WebGL community experiences*. Boca Raton, Fla.: CRC Press, 2013. [Online]. Verfügbar unter: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10574371

[13] Gist, *GLSL Noise Algorithms.* [Online]. Verfügbar unter: https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83 (Zugriff am: 3. Januar 2021).