Übung 2 - Perlin-Noise

Marko Nikic

5. November 2024

1 Einführung

In dieser Übung werden Sie eine einfache Version eines Perlin-Noise-Algorithmus implementieren. Der Perlin-Noise-Algorithmus, entwickelt von Ken Perlin und erstmals in seiner Arbeit von 1985 beschrieben [1], erzeugt glatte, pseudozufällige Muster, die oft als Grundlage für prozedurale Texturen und natürliche Landschaften verwendet werden. Typische Anwendungsbeispiele umfassen die Erstellung von realistisch wirkenden Landschaften in Computerspielen und die Animation von Wasser- oder Wolkeneffekten in Filmen.

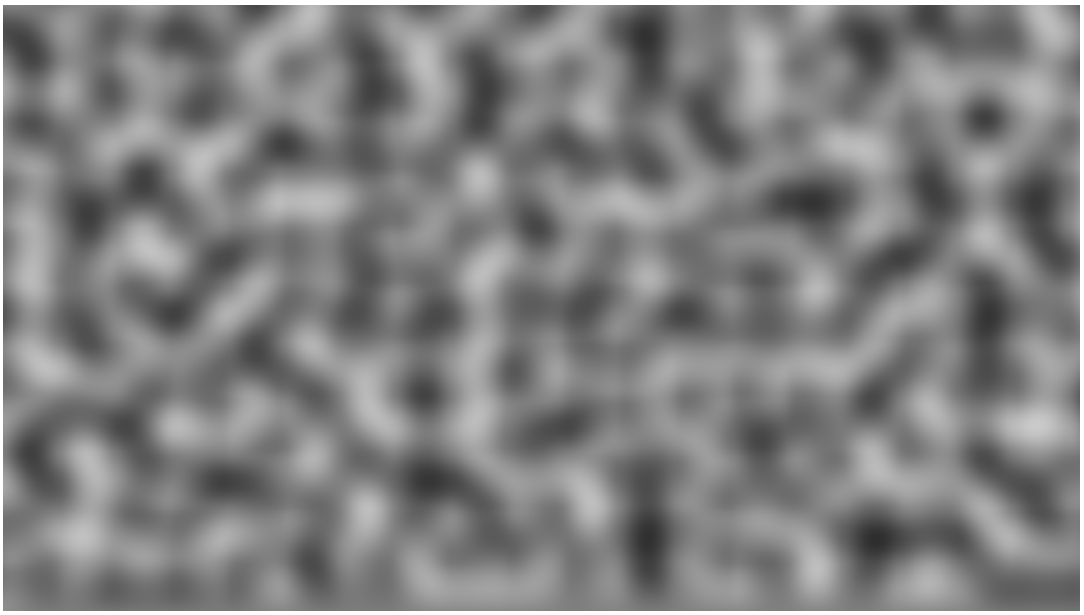


Abbildung 1: Eine Perlin Noise Textur mit 1920 x 1080 Pixeln

2 Mathematischer Hintergrund

Bevor Sie den Algorithmus implementieren können, müssen Sie verstehen, wie er funktioniert. Generell ist es möglich, den Algorithms in beliebig vielen Dimensionen zu implementieren aber Sie fokussieren sich auf eine 2D-Implementierung. Der Algorithmus läuft grob in drei Schritten ab, die in den nachfolgenden Abschnitten erklärt werden.

2.1 Ein Gitter aus zufälligen Gradientvektoren erstellen

.

Als erstes definieren Sie ein 2-dimensionales Gitter mit Gitterweite 1. In jedem Gitterpunkt wird ein normierter Vektor gespeichert, der in eine zufällige Richtung zeigt. Abbildung 2 zeigt ein Beispiel eines 10x5 Gitters.

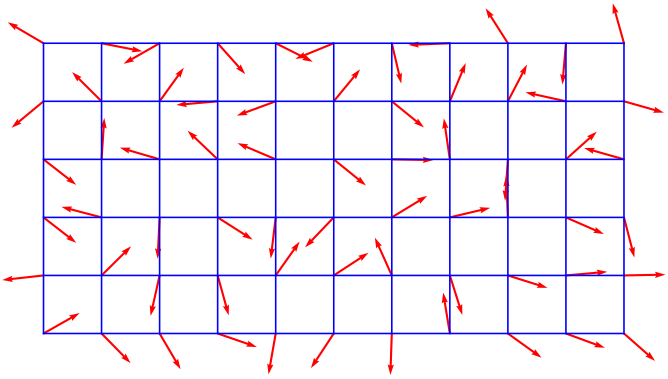
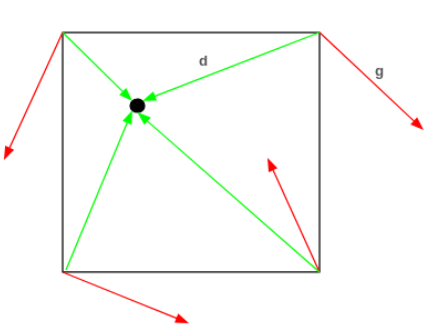


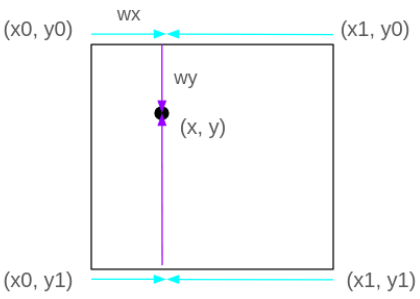
Abbildung 2: Ein 2D-Gitter mit Gradientvektoren [1]

2.2 Skalarprodukt von Gradientvektoren

Zur Berechnung des Perlin-Noise an einem beliebigen Punkt $(x, y) \in \mathbb{R}^2$ innerhalb unseres Gitters, werden im Allgemeinen die vier Abstandsvektoren zu den umliegenden vier Gitterpunkte benötigt. Diese Abstandsvektoren werden mit den entsprechenden Gradientvektoren skalarmultipliziert. Liegt der Punkt (x, y) exakt auf einem Gitterpunkt, so ist der Abstandsvektor 0 und auch das Skalarprodukt 0. Liegt der Punkt (x, y) exakt auf einer Gitterlinie, so sind nur die beiden benachbarten Gitterpunkte relevant. Folgen Sie der Notation in Abbildung 3a, so entspricht das Skalarprodukt der oberen rechten Ecke der Zelle $n = d \cdot g$.



(a) Skalarprodukt



(b) Interpolation

2.3 Bi-Interpolation der Skalarprodukte

Für eine Punkt (x, y), der im Innern einer Gitterzelle liegt, haben Sie nun vier Skalarprodukte, die bi-interpoliert werden, um den Perlin-Noise an der Stelle (x, y) zu ermitteln. Bei dieser Bi-

[1] Quelle: `https://en.wikipedia.org/wiki/Perlin_noise`

Interpolation werden die vier Skalarprodukte in Abhängigkeit der horizontalen und vertikalen Distanzen zu den entsprechenden Gitterpunkten gewichtet. Ist eine horizontale bzw. vertikale Distanz 0, so ist das zugehörige Gewicht 1. Ist eine horizontale bzw. vertikale Distanz 1, so ist das zugehörige Gewicht 0. Entspricht das Gewicht gerade 1 minus der horizontalen bzw. vertikalen Distanz, so spricht man von bilinearer Interpolation.

Um die Bi-Interpolation über die vier Skalarprodukte zu berechnen, gehen Sie folgendermassen vor:

1. Zuerst berechnen Sie den horizontalen Abstand wx zur linken vertikalen Gitterlinie und den vertikalen Abstand wy zur oberen horizontalen Gitterline der Gitterzelle (siehe Abbildung 3b).

2. Anschliessend interpolieren Sie die Skalarprodukte $n0$ und $n1$, die den Ecken $(x0, y0)$ und $(x1, y0)$ entsprechen, wobei $\text{smooth}(d)$ die Gewichtung der Interpolation beeinflusst: $\text{perlin}_{x1} = n0 + \text{smooth}(wx) \cdot (n1 - n0)$.

3. Als nächstes interpolieren Sie die Skalarprodukte $n2$ und $n3$, die den Ecken $(x0, y1)$ und $(x1, y1)$ entsprechen, auf analoge Weise: $\text{perlin}_{x2} = n2 + \text{smooth}(wx) \cdot (n3 - n2)$.

4. Um den abschliessenden Perlin-Noise-Wert an der Stelle (x, y) zu bekommen, interpolieren Sie die beiden zurvor bestimmten horizontalen Interpolationswerte in vertikaler Richtung: perlin_{x1} und perlin_{x2} mit dem Gewicht wy und erhalten $\text{perlin}_{xy} = \text{perlin}_{x1} + \text{smooth}(wy) \cdot (\text{perlin}_{x2} - \text{perlin}_{x1})$.

Bei der Funktion $\text{smooth}(d)$ handelt es sich um eine Interpolationsfunktion, welche einen Wert d zwischen 0 und 1 entgegennimmt. Wir empfehlen als Erstes folgende Funktion zu verwenden:

$$\text{smooth}(d) = 3d^2 - 2d^3 \tag{1}$$

Eine Eigenschaft, welche die Funktion in 1 zu einem guten Kandidatien für unseren Algorithmus macht, ist die Tatsache, dass die Ableitungen der Funktion an den Extremalstellen 0 und 1 jeweils 0 sind.

Sobald Sie eine erste funktionierende Implementation haben, ist es empfehlenswert auch folgende Funktionen für $\text{smooth}(d)$ auszuprobieren.

$$\text{smooth}(d) = d \tag{2}$$

Ken Perlin schlug damals eine noch glattere Funktion vor, bei welcher die ersten zwei Ableitungen an den Extremalstellen 0 sind:

$$\text{smooth}(d) = 6d^5 - 15d^4 + 10d^3 \tag{3}$$

3 Programmierung

Gehen Sie bei der Implementierung des Perlin-Noise-Algorithmus wie folgt vor:

1. Vervollständigen Sie die Klasse `Vector2d`, die sich in den beiden Dateien `Vector2d.h` und `Vector2d.cpp` befindet.

2. Vervollständigen Sie die Klasse `PerlinNoiseGenerator`, die sich in den beiden Dateien `PerlinNoiseGenerator.h` und `PerlinNoiseGenerator.cpp` befindet.

3.1 Vector2d

Für die Klasse `Vector2d` implementieren Sie folgende Methoden:

- `double x() const;` gibt die x-Koordinate zurück
- `double y() const;` gibt die y-Koordinate zurück
- Den multiplikations Operator, welcher das Skalarprodukt zweier Vektoren berechnet $\mathbf{v}_1 \cdot \mathbf{v}_2$ wobei $\mathbf{v}_1$ und $\mathbf{v}_2$ `Vector2d`'s sind.
- Die arithmetischen Operatoren $\mathbf{v}_1 + \mathbf{v}_2$, $c \cdot \mathbf{v}$, $\mathbf{v} \cdot c$ und $\mathbf{v}/c$, wobei $\mathbf{v}, \mathbf{v}_1, \mathbf{v}_2$ jeweils `Vector2d`'s sind und c eine Konstante vom Typ double ist.
- `void normalize();` normalisiert den Vektor.
- `Vector2d normalized() const;` gibt eine normalisierte Version des `this`-Objektes zurück, ohne das `this`-Objekt zu verändern.
- `double norm() const;` berechnet die Länge des Vektors.

Sie implementieren ebenfalls die Move Semantik der `Vector2d` Klasse (Move Konstruktor und Move Assignment Operator). Ihre Implementation sollte dabei das Vektor Objekt welches bewegt wird mit 0-Werten zurücklassen. Halten Sie bei der Implementierung der Move Operationen die Rule of 5 ein.

Schlussendlich implementieren Sie noch Versionen der arithmetischen Operatoren, bei welchen mindestens einer der beiden Operanden (sofern es sich um ein `Vector2d` Objekt handelt) eine R-Value ist.

3.2 PerlinNoiseGenerator

Die Klasse verfügt über folgende Member-Variablen:

- `m_rand` ist ein Zufallszahlengenerator der Klasse `Random`, die benutzt werden kann, um zufällige Gradientvektoren zu erzeugen. Mit dem Konstruktor `Random(seed, min, max)` kann der Generator initialisiert werden, so dass jeder Aufruf von `next()` eine zufällige Fliesskommazahl im Bereich $[\min, \max]$ zurückgibt.
- `m_gridWidth` enthält die Anzahl Zellen in x-Richtung
- `m_gridHeight` enthält die Anzahl Zellen in y-Richtung
- `m_gradients` speichert alle Gradientvektoren des Gitters.

Das Herzstück dieser Aufgabe ist die Implementierung der Funktion: `double eval(const Vector2d& p)`. Diese Funktion gibt uns für einen Punkt (x, y) den Perlin-Noise-Wert an dieser Stelle zurück.

3.3 Erstellen von Noise-Maps

Im Projekt steht Ihnen eine Klasse `Image` zur Verfügung, die ein Bild mit nxm Pixeln repräsentiert, wobei ein einzelner Pixel ein `uint8_t` ist und somit Werte von 0 bis 255 annehmen kann. Mit der Funktion `PGMWriter::write(outputFile, image);` können Sie Ihr erstelltes Bild im PGM-Format abspeichern und mittels dem mitgelieferten Picsi.jar oder einem anderen Bildprogramm visualisieren.

3.4 Einsatz des Programms

In der Funktion `main()` werden einige Kommandozeilenparameter für eingelesen, um das Erstellen von Bildern zu erleichtern. Sie können dabei folgende Parameter in der angegebenen Reihenfolge mitgeben:

- `imageWidth`
- `imageHeight`
- `cellSize`
- `seed`
- `outputFile`

Die ersten zwei Parameter definieren die Bildgrösse in Pixeln. Der nächste Parameter definiert die Zellgrösse in Pixeln. Wird ein Bild der Grösse 1000x1000 erstellt und sollen 50x50 Zellen vorhanden sein, dann muss die `cellSize` = 1000 / 50 = 20 sein. Mit dem Parameter `seed` können Sie deterministische und wiederholbare Bilder generieren, was das Debuggen erleichtert. Optional können Sie noch einen Dateipfad angeben, unter welchem das generierte Bild gespeichert wird. Wird kein Pfad angegeben, landet das Bild im Ordner `output` und trägt den Namen *PerlinNoise_{seed}.pgm*. Der folgende Aufruf des Programs im Terminal

$$\$ > ./\text{PerlinNoise.exe } 1000\ 1000\ 20\ 42$$

sollte ein Bild der Grösse 1000x1000 mit einer Zellengrösse von 20 erstellen und der seed ist 42. Sie werden das generierte Bild im Ordner *output* unter dem Namen *PerlinNoise_42.pgm* finden.

3.5 Unit-Tests

Das Projekt enhält Unit-Tests, welche alle Funktionalitäten der Klasse `Vector2d` testen. Sie können hier auch weitere Tests hinzufügen.

4 Abgabe

Geben Sie neben Ihrem Code bitte noch folgende Dokumente ab. Senden Sie alle Dokumente als E-Mail-Anhänge ungezippt an nikicm@ethz.ch und im cc an christoph.stamm@fhnw.ch.

- Mindestens ein Bild, welches Sie generiert haben, mit den zugehörigen Parametern.
- Ein kurzer Text (max. 1/2 Seite), der Ihre Beobachtungen beschreibt, wenn Sie die Gittergrösse variieren und verschiedene Interpolationsfunktionen verwenden.

5 Tipps

Der Perlin-Noise-Algorithmus ist um einiges einfacher zu implementieren, wenn Sie annehmen, dass alle Gitterzellen immer die Grösse 1 haben. Um trotzdem mit Zellgrössen von mehreren Pixeln arbeiten zu können, sollten Sie die ganzzahligen Bildkoordinaten in reelle Gitterkoordinaten umrechnen. Bei einer Bildgrösse von 1000x1000 und einer Zellgrösse von 20 (also ein 50x50 Gitter), sollten Sie mit reellen Koordinaten im Bereich von 0 bis 50 die Funktion `eval()` aufrufen.

Literatur

[1] K. Perlin, “An image synthesizer,” *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.