PERLIN NOISE

11. April 2022

Perlin Noise ist eine spannende mathematische Funktion mit der viele Probleme der Informartik gelöst werden können. Z.B. ist es möglich Buchstaben etwas unregelmäßiger erscheinen zu lassen, sodass diese wie handgeschrieben aussehen. Des Weiteren kann man unzählige zufällige Texturen in 2D oder Meshes in 3D generieren. In der Spieleentwicklung eignet es sich hervorragend für die Terraingenerierung von prozedural generierten Leveln. Aus diesen Gründen möchte ich mich in diesem Beitrag ausführlich mit diesem Thema beschäftigen.

Einführung

Was ist Noise?
Noise wurde 1982 von Ken Perlin entwickelt, um natürliche Phänomene wie Wolken, Landschaftstopologien oder Wasser zu simulieren. Im gegensatz zu *random()* ist *noise()* eine geglätette Version einer zufälligen Verteilung. Das bedeutet zwei aufeinanderfolgende Punkte hängen unmittelbar miteinander zusammen. So gibt die Funktion bei gleichem Parameter von Noise immer den gleichen Wert zurück z.B. *noise(0.176) = 0.516*. Die Noise Funktion gibt, wie die random Funktion, Werte zwischen 0 und 1 zurück. Während die Werte bei random auf lange Sicht immer gleich verteielt sind, sind sie bei der Noise Funktion normalverteilt. D.h. sie bewegen sich eher um den Wert 0.5 als 0 oder 1. Das kann man in dieser Gegenüberstellung schön sehen:

Random Beispiel

i	random()
1	0.962
2	0.773
3	0.789
4	0.318
5	0.0502

Noise Beispiel

i	x	noise(x)
1	0.16	0.587
2	0.17	0.589
3	0.18	0.591
4	0.19	0.592
5	0.20	0.592

Basiswissen für Noise

Interpolation
Die Noise Funktion ist im Prinzip eine Funktion, die mit etwas Mathematik über die normale random Funktion gebaut ist. Als Erstes müssen also random Zahlen über ein bestimmtes Interval gepickt werden, die sogenannte Wellenlänge bzw. *Wavelength*. Nach dem generieren dieser Zahlen wird mithilfe einer Interpolationsfunktion zwischen diesen Weiteren interpoliert. Beispielhafte Interpolationsfunktion:

$$f(x) = x^3 \cdot (x \cdot (x \cdot 6 - 15) + 10)$$

```js
// Umsetzung der Funktion im Code:
function interpolate(xq, w0, w1) {
    let sx = xq**3 * (xq * (xq * 6.0 - 15.0) + 10.0);
    return (1 - sx) * w0 + sx * w1;
}
```

Oktaven

Nach dem Interpolieren wird die *Wavelength* halbiert. Tatsächlich ist das nicht der einzige Wert der halbiert werden muss: Die Höhe der Curve bzw. *Wave* wird Amplitude genannt. Diese muss auch halbiert werden. Warum das ganze? Die einzelnen *Waves* werden im nächsten Schritt addiert. Diese *Waves* nennt man dann auch Oktaven oder *Octaves*. Das erklärt auch warum immer halbiert werden muss, da so nicht der Wertebereich von 0-1 überschritten wird. Beweis:

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$, konvergiert (nach Konvergenzkriterien)

Wrap-up

Um die letzten Absätze nochmal zusammenzufassen, habe ich eine Grafik erstellt, die ungefähr den Ablauf bei der Berechnung zeigt:

Schritt 1: Zufällige Punkte auswählen
Schritt 2: Mit Interpolationsfunktion zwischen den Punkten Interpolieren
Schritt 3: Alle Oktaven miteinander addieren

Zufall? Oder doch nicht?

Warum sollte man eine eigene Random Funktion bauen?
Die Intuition würde ganz klar auf die schon vorhandene *random()* Funktion hindeuten, warum also extra Arbeit investieren? Die standard Random Funktion besitzt keine möglichkeit die erzeugten Werte nochmal in der gleichen Reihenfolge zu generieren, d.h. sie ist nicht deterministisch. Nun gibt es einen kleinen Trick: man führt das Konzept von sogennanten *Seeds* ein. Das bedeutet die Random Funktion wird am Anfang des Programms mit einem Seed erzeugt und verhält sich so immer gleich bei jedem Programmstart. Das kann z.B. bei der Levelgenerierung Sinnvoll sein, wenn ein Spieler die gleiche Welt nochmal spielen möchte.

Implementierung der Random Funktion
Eine einfache Möglichkeit ist es, einen *Pseudo-Random Number Generator* zu bauen:

```js
class PRNG {
    constructor(seed=0) {
        if (isNaN(seed)) seed = seed.split('').map(
            (_,i) => seed.charCodeAt(i)).reduce((a,b)=>a+b);
        this._seed = seed % 2147483647;
        if (this._seed <= 0) this._seed += 2147483646;
    }
    next() {
        let v = this._seed = this._seed * 16807 % 2147483647;
        return (v - 1) / 2147483646;
    }
}
```

```js
// Beispiel
let r = new PRNG(123456);
r.next()        // 0.9662122432759146
r.next()        // 0.1291730023260908
```

Implementierung von 1D Noise

Einführung von Permuationen

Damit die Punkte der Noise Funktion voneinander abhängig werden muss die PRNG Klasse noch erweitert werden. Dafür wird eine Permutationstabelle benutzt, da bei dieser bei gleichem Input das gleiche Ergebnis geliefert wird und die Ergebnisse bei naheliegenden Zahlen weit auseinander sind. Die Permutationstabelle wird mit der *next()* Funktion befüllt, die etwas weiter oben zu finden ist. So könnte eine Implementierung der Permutationstabelle aussehen:

```js
class PRNG {
    constructor(seed=0) {
        // ...
        let tmp = new Array(256).fill().map(
                    v => Math.floor(this.next() * 256));
        this.perm = new Array(512).fill().map((_,i) => tmp[i % 256])
    }
    // ...
}
```

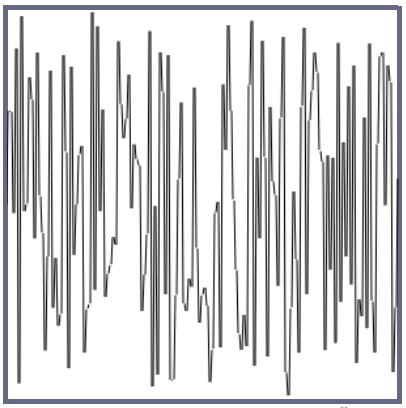
Die tatsächliche Implementierung

Nach viel Theorie und Vorbereitung kann nun wirklich die 1D Perlin Noise erstellt werden, dafür habe ich folgenden Code benutzt:

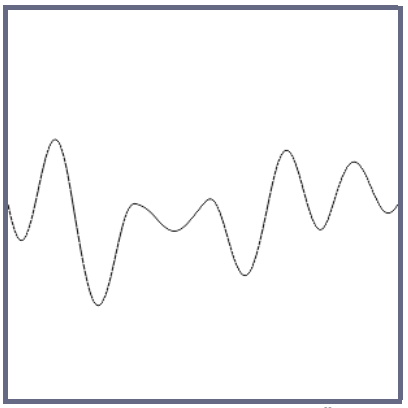
```js
class Perlin {
    constructor(seed=0) {
        this._random = new PRNG(seed);
        this._perm = this._random.perm;
    }
    interpolate(xq, w0, w1){
        let sx = xq**3 * (xq * (xq * 6.0 - 15.0) + 10.0);
        return (1 - sx) * w0 + sx * w1;
    }
    noise(x) {
        let _x = x >= 0 ? x % 256 : 256 - Math.abs(x) % 256;
        let p = Math.floor(_x);
        let g0 = this._perm[p] / 256 * 2 - 1;
        let g1 = this._perm[p + 1] / 256 * 2 - 1;
        let xq = _x - p;
        let w0 = g0 * xq;
        let w1 = g1 * (xq - 1);
        let w = this.interpolate(xq, w0, w1);
        return 0.5 + w;
    }
}
```

Einige Beispiele mit 1D Perlin Noise

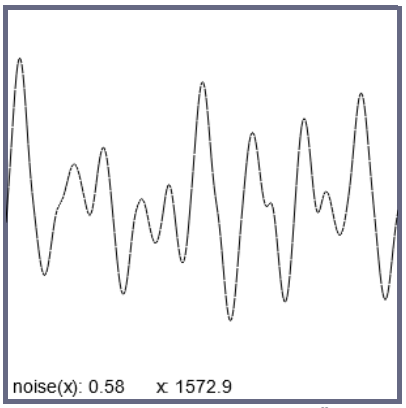
Im nachfolgenden Absatz habe ich ein Paar Zeichnungen mit 1D Perlin Noise erstellt und zum Vergleich eine Zeichnung mit der *random()* Funktion.



Verteilung mit random()



Verteilung mit noise()



Scrolling über noise()

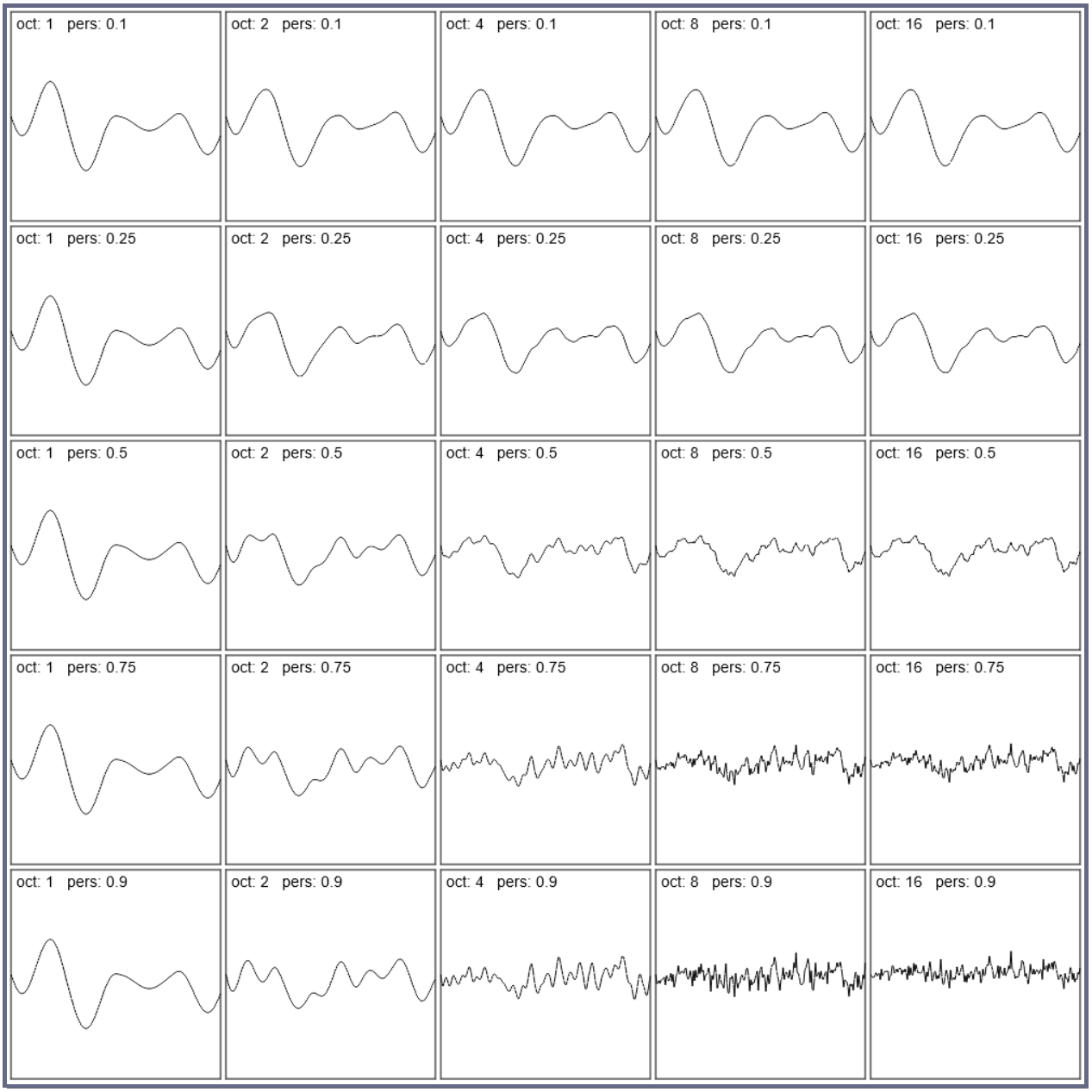
1D Perlin Noise mit Oktaven

Funktionsweise von Oktaven

Für die Noise Funktion mit Oktaven wird im Prinzip die schon vorhandene Noise Funktion benutzt und so oft aufaddiert, bis ein Zähler bei der Oktavenanzahl gelandet ist. Währenddessen wird die Noise Funktion mit der Frequenz und Amplitude gestreckt bzw. gestaucht. Dadurch gibt es später sehr viele Feinheiten bei dem Endergebnis. Die Anzahl, um die die Funktion gestreckt bzw. gestaucht wird, muss mitgezählt werden, sodass am Ende die Werte wieder normalisiert, d.h. im Bereich zwischen 1 und 0, sind. Außerdem bekommt die Funktion einen weiteren Parameter: *Persistence*, dieser gibt an wie fein die Noise dargestellt werden soll. Die Umsetzung im Code sieht dann wie folgt aus:

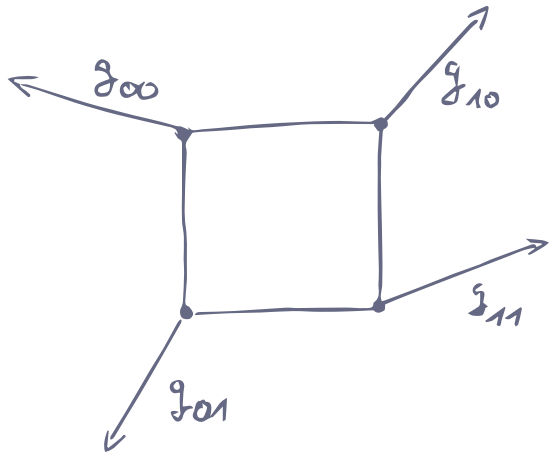
```js
class Perlin {
    // ...
    noiseOctave(x, oct=8, pers=0.5) {
        let sum = 0;
        let f = 1;
        let a = 1;
        let norm = 0;
        for (let i = 0; i < oct; i++) {
            sum += a * this.noise(f * x);
            norm += a;
            f *= 2;
            a *= pers;
        }
        return sum / norm;
    }
}
```

Damit noch klarer wird, was die Werte *Octaves* und *Persistance* tun, hier eine kleine Übersicht:

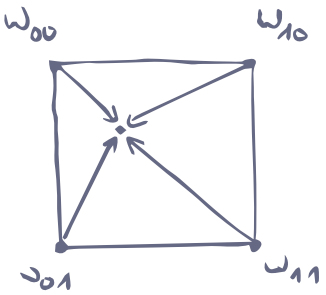


2D Perlin Noise

Was passiert wenn sich die Dimension erhöht?
Im zweidimensionalen Raum wird nicht zwischen 2 sondern zwischen 4 Punkten interpoliert. Die allgemeine Formel für die Anzahl an zufälligen Punkten wäre $n = 2^R$, d.h. für die dritte Dimension wären es 8 zufällige Punkte. Die Punkte werden als Gradientvektoren beschrieben, was man auch gut in der Abbildung unten sehen kann. Von diesen wird dann der Distanzvektor zum Zielpunkt gebildet. Zwischen den beiden Vektoren wird das Skalarprodukt gebildet und interpoliert.



Gradientvektoren



Distanzvektoren

Interpolation in 2D

Die Interpolation in 2D funktioniert fast analog zur 1D Interpolation. Es werden jeweils 2 Eckpuckte interploiert und die Ergebnisse werden nochmals interpoliert. Das bedeutet konkret zwischen w00 und w10 bzw. w01 und w11 wird interpoliert. Eine Implementierung könnte dann so aussehen:

```js
function interpolate2D(xq, yq, w00, w10, w01, w11){
    let s = (x) => x**3 * (x * (x * 6.0 - 15.0) + 10.0);
    let w0 = (1 - s(xq)) * w00 + s(xq) * w10;
    let w1 = (1 - s(xq)) * w01 + s(xq) * w11;
    return (1 - s(yq)) * w0 + s(yq) * w1;
}
```

Skalarprodukt von Gradientvektoren

Um das Skalarprodukt der Gradientvektoren und der Distanzvektoren zu berechnen, benötigt man eine Hilfsfunktion. Diese bestimmt die richtigen Vorzeichen der einzelnen Vektorwerte.

```js
function grad2D(g, x, y) {
    switch (g % 4) {
        case 0: return x + y;
        case 1: return -x + y;
        case 2: return x -y;
        case 3: return -x -y;
    }
}
```

Implementierung von Perlin Noise 2D

Die Inhalte der letzten Paragraphen ergeben folgenden Code für die 2D Perlin Noise Funktion:

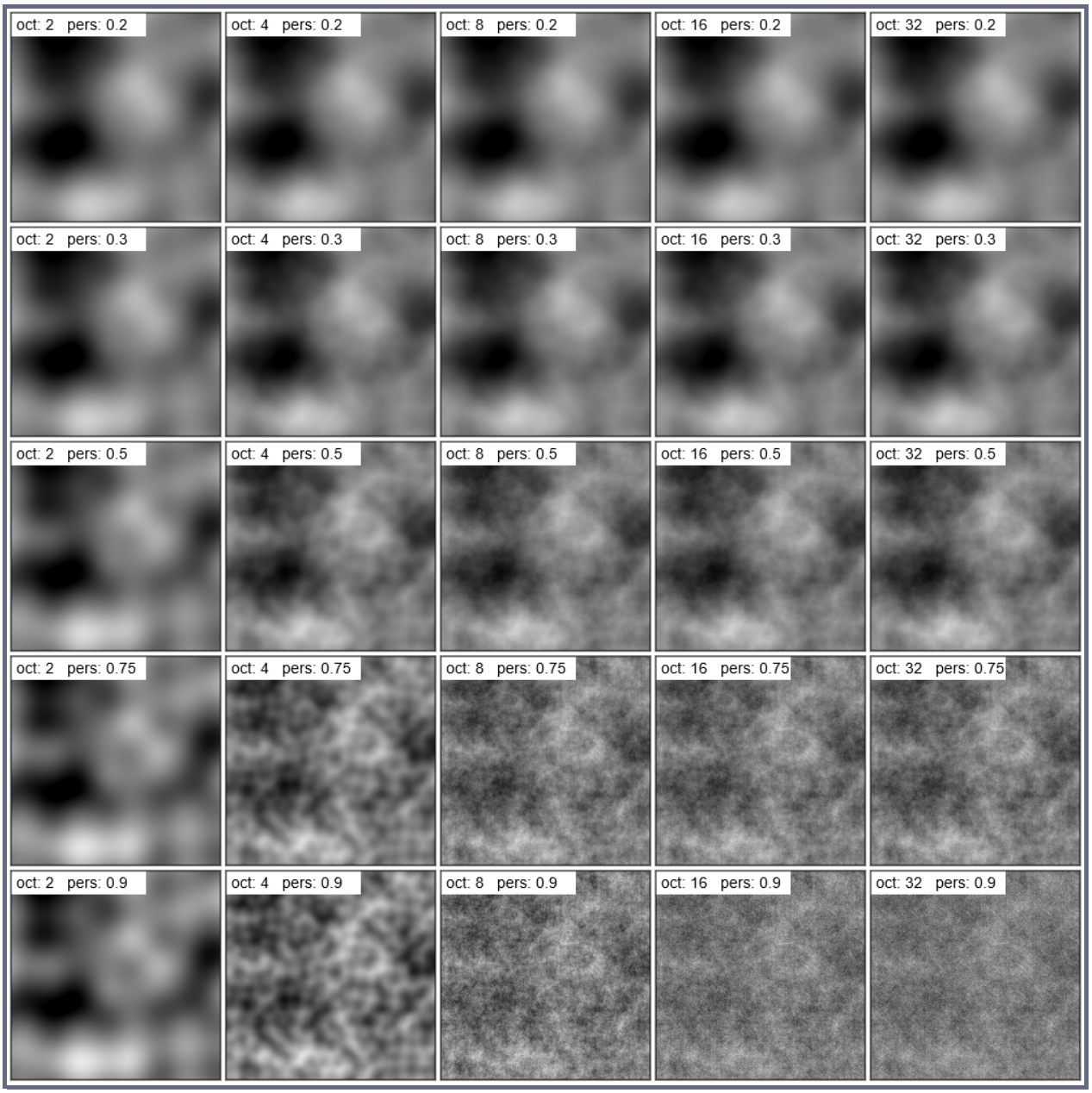
```js
class Perlin {
    // ...
    noise2D(x, y) {
        let _x = x >= 0 ? x % 256 : 256 - Math.abs(x) % 256;
        let _y = y >= 0 ? y % 256 : 256 - Math.abs(y) % 256;
        let px = Math.floor(_x);
        let py = Math.floor(_y);
        let g00 = this._perm[this._perm[px] + py];
        let g10 = this._perm[this._perm[px + 1] + py];
        let g01 = this._perm[this._perm[px] + py + 1];
        let g11 = this._perm[this._perm[px + 1] + py + 1];
        let xq = _x - px;
        let yq = _y - py;
        let w00 = this.grad2D(g00, xq, yq);
        let w10 = this.grad2D(g10, xq - 1, yq);
        let w01 = this.grad2D(g01, xq, yq - 1);
        let w11 = this.grad2D(g11, xq - 1, yq - 1);
        let w = this.interpolate2D(xq, yq, w00, w10, w01, w11);
        return 0.5 + w;
    }
}
```

2D Perlin Noise mit Oktaven

Oktaven Implementierung für 2D Perlin Noise
Die Funktion ist fast identisch zur 1D *noiseOctave()*, es wird im Prinzip nur die Noise Funktion ausgetauscht und y als Parameter hinzugefügt:

```js
class Perlin {
    // ...
    noise2DOctave(x, y, oct=8, pers=0.5) {
        let sum = 0;
        let f = 1;
        let a = 1;
        let norm = 0;
        for (let i = 0; i < oct; i++) {
            sum += a * this.noise2D(f * x, f * y);
            norm += a;
            f *= 2;
            a *= pers;
        }
        return sum / norm;
    }
}
```

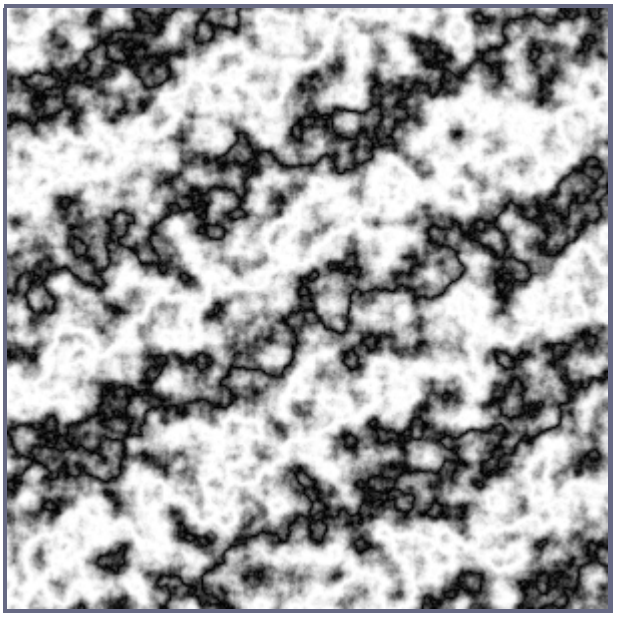
Verschiedene Oktaven für 2D Perlin Noise



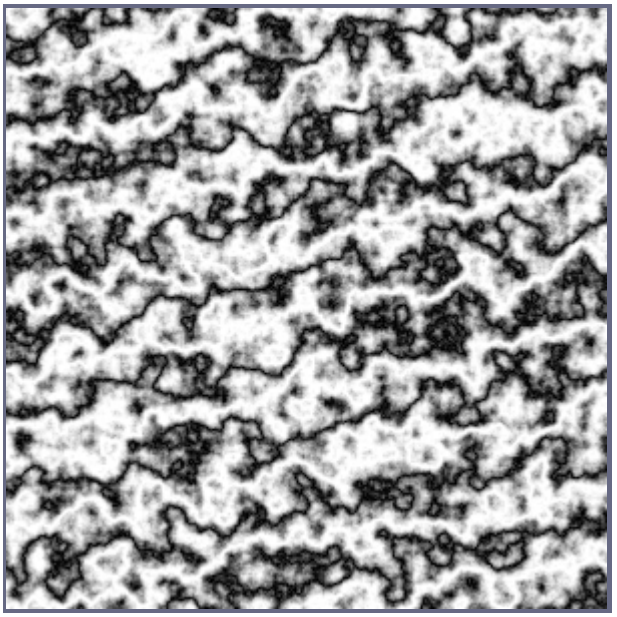
Der komplette Source Code für 1D und 2D Perlin Noise befindet sich auf GitHub

Weitere Experimente

Texturgenerierung



Marmor Textur Test



Marmor Textur Test 2

Jeder Pixel der Textur wird nach dieser Formel berechnet:

```js
let xPeriod = 5;
let yPeriod = 10;
let turbPower = 3;
let turbSize = 32;
let xyVal = x * xPeriod / WIDTH + y * yPeriod / HEIGHT +
            turbPower * pn.noise2DOctave(xoff, yoff, turbSize);
let b = Math.abs(Math.sin(xyVal + Math.PI)) * 256; // Wert auf 0-255
```

Perlin Noise Flow Field

Zum Abschluss gibt es noch eine etwas interessantere Sache: ein Flow Field. Ich werde nicht auf die genaue Implementierung eingehen, weil das den Rahmen für diesen Beitrag sprengen würde. Die Quelle zu dem Source Code ist aber in den Referenzen verlinkt.

Referenzen

Burger, Wilhelm (16.05.2015). Gradientenbasierte Rauschfunktionen und Perlin Noise
https://www.researchgate.net/profile/Wilhelm-Burger/publication/277014654_Gradientenbasierte_Rauschfunktionen_und_Perlin_Noise

Biagioli, Adrian (09.08.2014). Understanding Perlin Noise
https://adrianb.io/2014/08/09/perlinnoise.html

Elias, Hugo (10.02.1999). Perlin Noise
http://web.archive.org/web/20160418004148/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
alternativer Link

Xiang, Fanbo (03.11.2017). Implementing 2D Perlin Noise
https://www.fbxiang.com/blog/2017/11/03/implementing_perlin_noise.html

Miller, Park (24.09.2005). Pseudo-Random Number Generator
https://www.firstpr.com.au/dsp/rand31

Vandevenne, Lode (2004). Texture Generation using Random Noise
https://lodev.org/cgtutor/randomnoise.html

Shiffman, Daniel (27.06.2016). Perlin Noise Flow Field
https://thecodingtrain.com/CodingChallenges/024-perlinnoiseflowfield.html