Gradientenbasierte Rauschfunktionen und Perlin Noise

von

Wilhelm Burger

wilhelm.burger@fh-hagenberg.at

Technischer Bericht HGBTR08-02

20. November 2008

Digitale Medien

Fakultät für Informatik, Kommunikation und Medien, FH-Oberösterreich,
Campus Hagenberg, Softwarepark 11, 4232 Hagenberg, Austria

www.fh-hagenberg.at/dm

Zusammenfassung

Dieses Dokument enthält eine relativ detaillierte algorithmische Darstellung der Erzeugung von ein- und mehrdimensionalem Gradientenrauschen unter besonderer Berücksichtigung der Methode von Perlin.

Schlüsselworte: Gradient noise, multi-dimensional noise function, Perlin noise, image synthesis, algorithm, integer hash function, Java, ImageJ

Quellenangabe

Wilhelm Burger: *Gradientenbasierte Rauschfunktionen und Perlin Noise*, Technischer Bericht HGBTR08-02, School of Informatics, Communications and Media, Upper Austria University of Applied Sciences, Hagenberg, Austria, November 2008.
http://staff.fh-hagenberg.at/burger/

```
@techreport{BurgerGradientNoise2008,
          author = {Burger, Wilhelm},
           title = {Gradientenbasierte Rauschfunktionen und Perlin Noise},
        language = {german},
     institution = {School of Informatics, Communications and Media, Upper
                    Austria University of Applied Sciences},
         address = {Hagenberg, Austria},
          number = {HGBTR08-02},
            year = {2008},
           month = nov,
             url = {http://staff.fh-hagenberg.at/burger/}
}
```

1 Einführung

Rauschfunktionen auf Basis pseudo-zufälliger Gradientenwerte finden zahlreiche Anwendungen, speziell in der Bildsynthese. Bekannt wurden diese Methoden nicht zuletzt durch Ken Perlin, der für seine eindrucksvollen Arbeiten 1997 einen „Oscar“ (Academy Award for Technical Achievement) erhielt. Gradientenrauschen ist natürlich nur eine mögliche Form der Erzeugung von Rauschfunktionen. Eine Übersicht zu alternativen Ansätzen findet sich beispielsweise in [4].

Obwohl die Details der Perlin Noise Methode vielfach publiziert wurden [1, 4, 5, 7, 9] und eine konkrete Referenzimplementierung auf Ken Perlins eigener Website[1] vorliegt, lässt die verfügbare Information überaschend viele Fragen offen. In der Tat gehört der Perlin-Algorithmus (aus persönlicher Sicht des Autors) zu den dürftigst dokumentierten Verfahren in diesem Bereich überhaupt. Eine Implementierung ist auf dieser Basis mühevoll, weil einige wichtige Details nur aus dem Quellkode der (äußerst „kompakten“) Referenzimplementierung abgeleitet werden können.

Diese Dokument ist eine Zusammenfassung der wesentlichen Grundlagen und Implementierungsschritte zur Erzeugung 1-, 2- und N-dimensionaler Rauschfunktionen mit der Perlin-Methode. Die gezeigten Algorithmen sollten leicht in einer konkreten Programmiersprache wie Java oder C umzusetzen sein. Eine prototypische Java-Implementierung dazu ist online verfügbar.

2 Eindimensionale Noise-Funktionen

Ziel ist die Erzeugung einer kontinuierlichen, eindimensionalen Zufallsfunktion

$$\mathsf{noise}(x) : \mathbb{R} \to \mathbb{R}, \tag{1}$$

basierend auf einer diskreten Folge von vorgegebenen (aber ebenfalls zufälligen) Werten

$$g_u \in \mathbb{R}, \tag{2}$$

für die diskreten Raster- oder Gitterpunkte $u \in \mathbb{Z}$. Die Werte g_u in Gl. 2 spezifizieren also die kontinuierliche Rauschfunktion $\mathsf{noise}(x)$ an den diskreten Positionen $x = u \in \mathbb{Z}$. Es können dies u. a. die vorgegebenen *Werte* (values) der Funktion sein oder auch der *Anstieg* (gradient) der Funktion an diskreten Stellen (siehe Abb. 1). Im ersten Fall spricht man von „Value Noise“, im zweiten Fall von „Gradient Noise“ [4, S. 70]. Der Wert von $\mathsf{noise}(x)$ an einer beliebigen kontinuierlichen Position $x \in \mathbb{R}$ ergibt sich durch lokale Interpolation in der Form

$$\mathsf{noise}(x) = F(x, g_p, g_{p+1}) = F(x, g_{\lfloor x \rfloor}, g_{\lfloor x \rfloor + 1}) \tag{3}$$

[1] http://mrl.nyu.edu/~perlin

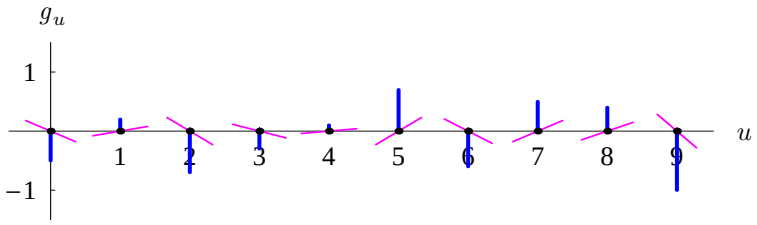


Abbildung 1: Die Werte der diskreten Zufallsfolge g_u (blau) geben die Steigung der Rauschfunktion $\mathsf{noise}(x)$ an den ganzzahligen Positionen $x = u \in \mathbb{Z}$ vor (die resultierenden Tangenten sind in magenta angezeigt). Diese Positionen sind gleichzeitig Nullstellen der Funktion.

zwischen den Werten an den x umgebenden Rasterpunkten $p = \lfloor x \rfloor$ und $p + 1$.

2.1 Gradienten-Noise

Perlin Noise ist eine Form von Gradient Noise, die Zufallswerte g_u definieren also den *Anstieg* (gradient) der Funktion $\mathsf{noise}(x)$ an den diskreten Punkten $x = u$. Die diskreten Gradientenwerte werden von einer Funktion $\mathsf{grad}(u)$ erzeugt, d. h.,

$$g_u = \mathsf{grad}(u) \in [-1, 1].$$

Dabei ist $\mathsf{grad}(u)$ eine Abbildung der ganzen Zahlen $\mathbb{Z}$ auf reellwertige Pseudo-Zufallszahlen im Intervall $[-1, 1]$, die (so wird angenommen) annähernd gleichverteilt sind. Diese Abbildung wird üblicherweise mithilfe einer Hash-Funktion realisiert, wie in Abschn. 2.4 genauer beschrieben wird.

2.1.1 Stückweise, lokale Interpolation

Die Eigenschaften der Rauschfunktion werden maßgeblich von der gewählten Interpolationsfunktion F bestimmt. Diese sollte nicht nur die oben genannten Bedingungen erfüllen, sondern auch zu einem visuell guten Ergebnis führen. In den folgenden Abschnitten geht es daher um die Definition der Interpolationsfunktion F.

An den diskreten Positionen $x = u$ sind nicht nur die Gradientenwerte g_u vorgegeben, sie sollen zudem auch Nullstellen der Rauschfunktion sein. Für die Rauschfunktion $\mathsf{noise}(x)$ soll also zunächst gelten

$$\mathsf{noise}(u) = 0 \qquad \text{und} \qquad \mathsf{noise}'(u) = g_u, \tag{4}$$

für alle $u \in \mathbb{Z}$. Für die lokale Interpolationsfunktion F (Gl. 3) muss daher gelten:

$$\begin{aligned} F(p, g_p, g_{p+1}) &= 0, & F'(p, g_p, g_{p+1}) &= g_p, \\ F(p+1, g_p, g_{p+1}) &= 0, & F'(p+1, g_p, g_{p+1}) &= g_{p+1}. \end{aligned} \tag{5}$$

$F'(x, g_p, g_{p+1})$ bezeichnet dabei die erste Ableitung der Funktion f in Bezug auf deren Parameter x, d. h.

$$F'(x, g_p, g_{p+1}) = \frac{\partial F(x, g_p, g_{p+1})}{\partial x}\,. \tag{6}$$

2.1.2 Interpolation im $[0, 1]$-Intervall

Da an jeder Stelle der Rauschfunktion das Interpolationsergebnis nur von den benachbarten Rasterpunkten abhängig ist, muss für gegebene Randwerte g_a, g_b zwischen jedem beliebigen Paar von benachbarten Rasterpunkten dasselbe Ergebnis entstehen, d. h.,

$$F(x, g_a, g_b) = F(x + i, g_a, g_b), \tag{7}$$

für beliebige $i \in \mathbb{Z}$. Insbesondere gilt natürlich auch

$$F(x, g_a, g_b) = F(x - \lfloor x \rfloor, g_a, g_b), \tag{8}$$

mit $(x - \lfloor x \rfloor) \in [0, 1]$. Wir können daher die Interpolation ohne Einschränkung der Allgemeinheit auf das Einheitsintervall $[0, 1]$ beschränken und ersetzen dazu

$$F(\dot{x}, g_0, g_1) = F(x - p, g_p, g_{p+1}), \tag{9}$$

mit $p = \lfloor x \rfloor$, $\dot{x} = x - p$, $g_0 = g_p$ und $g_1 = g_{p+1}$. Da $\dot{x} \in [0, 1]$, interessiert uns die Funktion f im Folgenden nur für Argumente im Intervall $[0, 1]$.[2]

2.1.3 Interpolation mit einer Polynomfunktion

Wenn wir – naheliegenderweise – als Interpolationsfunktion $F(\dot{x}, g_0, g_1)$ eine Polynomfunktion verwenden, so benötigen wir dafür mindestens ein Polynom dritten Grades, also eine Funktion der Form

$$F_3(\dot{x}, g_0, g_1) = a_3 \cdot \dot{x}^3 + a_2 \cdot \dot{x}^2 + a_1 \cdot \dot{x} + a_0 \tag{10}$$

mit den (noch zu bestimmenden) Koeffizienten $a_0, a_1, a_2, a_3 \in \mathbb{R}$. In diesem Fall ist die erste Ableitung (der Gradient) der Funktion

$$F_3'(\dot{x}, g_0, g_1) = 3a_3 \cdot \dot{x}^2 + 2a_2 \cdot \dot{x} + a_1 \tag{11}$$

und die zugehörige Lösung unter den Bedingungen in (5) ist

$$a_3 = g_0 + g_1, \quad a_2 = -2g_0 - g_1, \quad a_1 = g_0, \quad a_0 = 0, \tag{12}$$

[2] Zur Klarheit wird nachfolgend die Variable $\dot{x}$ anstelle von x verwendet, um anzuzeigen, dass $\dot{x}$ im Intervall $[0, 1]$ liegt.

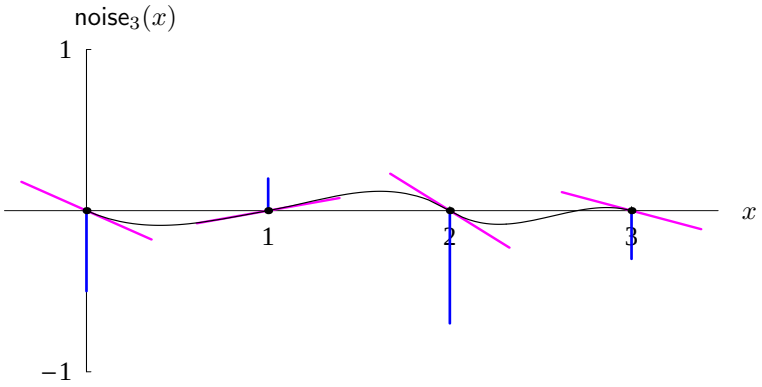


Abbildung 2: Interpolation mit dem kleinstmöglichen (kubischen) Polynom nach Gl. 13. Die blauen Balken zeigen die zufälligen Gradientenwerte g_i an den ganzzahligen Positionen, an denen gleichzeitig die Nullstellen der Funktion liegen. Die entsprechenden Tangenten (deren Steigung durch die Gradienten g_i vorgegeben ist) sind ebenfalls dargestellt. Die interpolierte Funktion ist C1-kontinuierlich, u. a. weil die erste Ableitung (der Anstieg) an den Übergängen zwischen benachbarten Segmenten jeweils identisch ist. Die Funktion ist nicht C2-kontinuierlich, da die zweite Ableitung (Krümmung) der Funktion an den ganzzahligen Positionen im Allgemeinen nicht stetig ist.

und somit

$$F_3(\dot{x}, g_0, g_1) = (g_0 + g_1) \cdot \dot{x}^3 - (2g_0 + g_1) \cdot \dot{x}^2 + g_0 \cdot \dot{x}. \tag{13}$$

In etwas anderer Form ausgedrückt ergibt das

$$F_3(\dot{x}, g_0, g_1) = g_0 \cdot (\dot{x} - 1)^2 \cdot \dot{x} + g_1 \cdot (\dot{x} - 1) \cdot \dot{x}^2. \tag{14}$$

Abb. 2 zeigt ein Beispiel für die stückweise Interpolation mit einem kubischen Polynom nach Gl. 13 bzw. 14.

2.1.4 Original Perlin-Interpolation

In der Originalversion [7] seines Verfahrens verwendet Perlin zur Interpolation statt des kubischen Polynoms (Gl. 10) allerdings ein Polynom *vierter* Ordnung der Form

$$F(\dot{x}, g_0, g_1) = a_4 \cdot \dot{x}^4 + a_3 \cdot \dot{x}^3 + a_2 \cdot \dot{x}^2 + a_1 \cdot \dot{x} + a_0, \tag{15}$$

das unter den Randbedingungen in Gl. 5 die Lösung

$$a_4 = a_2 + 2g_0 + g_1, \quad a_3 = -2a_2 - 3g_0 - g_1, \quad a_1 = g_0, \quad a_0 = 0,$$

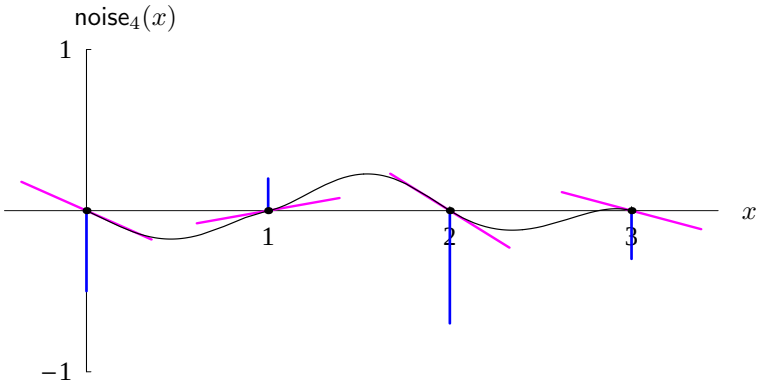


Abbildung 3: Anwendung der ursprünglichen Perlin-Interpolationsfunktion [7] nach Gl. 17 für die ersten vier Gradientenwerte aus Abb. 1. An den ganzzahligen Positionen ist der Funktionswert jeweils null und der Anstieg der Funktion enspricht dem Wert g_i der Zufallsfolge in Abb. 1 (a). Der Graph in (b) zeigt die selbe Funktion mit vergrößertem Maßstab in der y-Richtung.

aufweist, wobei a_2 noch frei wählbar ist. Für $a_2 = -3g_1$ ergibt sich die Funktion

$$F_4(\dot{x}, g_0, g_1) = 2(g_0 - g_1) \cdot \dot{x}^4 - (3g_0 - 5g_1) \cdot \dot{x}^3 - 3g_1 \cdot \dot{x}^2 + g_0 \cdot \dot{x} \qquad (16)$$

oder, in der von Perlin verwendeten Schreibweise,

$$F_4(\dot{x}, g_0, g_1) = g_0 \cdot \dot{x} + \big(3\dot{x}^2 - 2\dot{x}^3\big) \cdot \big(g_1 \cdot (\dot{x} - 1) - g_0 \cdot \dot{x}\big). \qquad (17)$$

Abb. 3 zeigt ein Beispiel für die in dieser Form interpolierte Funktion unter Verwendung der ersten vier Gradientenwerte aus Abb. 1. Die zweite Ableitung (Krümmung) von $F_4()$ bezüglich $\dot{x}$ beträgt an den Randstellen übrigens

$$F_4''(0, g_0, g_1) = -6g_1 \quad \text{und} \quad F_4''(1, g_0, g_1) = 6g_0, \qquad (18)$$

ist also im Allgemeinen ungleich null.[3] Dies ist nicht unproblematisch, da sich dadurch an den Übergängen zwischen benachbarten Segmenten die Krümmung der Funktion sprungartig ändern kann (s. Abschnitt 2.1.6).

2.1.5 Interpolation der Tangenten

Im eindimensionalen Fall spezifiziert jeder der vorgegebenen Gradientenwerte g_i eine *Tangente* im Punkt i, also eine Gerade mit der Steigung g_i, die die x-Achse an der Position i schneidet. Diese Tangentengerade wird durch die lineare Funktion

$$h_i(x) = g_i \cdot (x - i) \qquad (19)$$

[3] Interessanterweise ist also die Krümmung der Funktion an den Randpunkten des $[0, 1]$-Intervalls nur vom jeweils *gegenüber* liegenden Gradientenwert abhängig.

(für beliebige $x \in \mathbb{R}$, $i \in \mathbb{Z}$) beschrieben.

Die Funktion $F_4(\dot{x}, g_0, g_1)$ in Gl. 17 kann auch interpretiert werden als Spline-Interpolation der zugehörigen Tangenten (mit den Steigungen g_0 bzw. g_1) an den Endpunkten des aktuellen Intervalls.[4] Die Geradengleichungen der Tangenten für die Punkte $i = 0, 1$ ist nach Gl. 19

$$\begin{aligned} h_0(x) &= g_0 \cdot x, \qquad \text{bzw.} \\ h_1(x) &= g_1 \cdot (x - 1). \end{aligned} \tag{20}$$

Durch Ersetzung der Ausdrücke

$$g_0 \cdot \dot{x} \;\leftarrow\; h_0(\dot{x}) \qquad \text{und} \qquad g_1 \cdot (\dot{x} - 1) \;\leftarrow\; h_1(\dot{x}) \tag{21}$$

in Gl. 17 ergibt sich die Interpolationsfunktion in der Form

$$F_4(\dot{x}, g_0, g_1) = h_0(\dot{x}) + \big(3\dot{x}^2 - 2\dot{x}^3\big) \cdot \big(h_1(\dot{x}) - h_0(\dot{x})\big) \tag{22}$$

$$= h_0(\dot{x}) + s(\dot{x}) \cdot \big(h_1(\dot{x}) - h_0(\dot{x})\big), \tag{23}$$

mit der kubischen *Überblendungsfunktion*[5]

$$s(x) = 3x^2 - 2x^3 \tag{24}$$

(siehe Abb. 4). Durch geringfügige Umformung erhalten wir aus Gl. 23

$$F_4(\dot{x}, g_0, g_1) = h_0(\dot{x}) + s(\dot{x}) \cdot h_1(\dot{x}) - s(\dot{x}) \cdot h_0(\dot{x}) \tag{25}$$

$$= (1 - s(\dot{x})) \cdot h_0(\dot{x}) + s(\dot{x}) \cdot h_1(\dot{x}). \tag{26}$$

Dies ist analog zur klassischen Überblendung[6] zwischen zwei konstanten Werten c_0 und c_1 mit einer S-förmigem (sigmoiden) Gewichtungsfunktion $s(x)$ in der Form

$$c(x) = (1 - s(x)) \cdot c_0 + s(x) \cdot c_1,$$

für $x \in [0, 1]$.

2.1.6 Modifizierte Perlin-Interpolation

Wie in Gl. 18 gezeigt, ist die zweite Ableitung (Krümmung) der ursprünglichen Interpolationsfunktion von Perlin an den Randpunkten des Einheitsintervalls ungleich null. In der modifizierten Formulierung [9] wird daher

[4] Siehe http://www.mandelbrot-dazibao.com/Perlin/Perlin1.htm.

[5] Die eingangs beschriebene Interpolation mit dem minimalen kubischen Polynom (Gl. 13) – lässt sich übrigens *nicht* in dieser Form, d. h. als Überblendung der Tangentialgeraden darstellen.

[6] In [9] wird $s(x)$ auch als *fade*-Funktion bezeichnet.

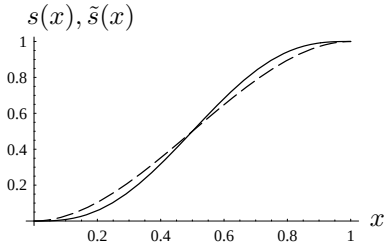


Abbildung 4: S-förmige Überblendungsfunktionen zur örtlichen Interpolation innerhalb des $[0, 1]$-Intervalls. Originalversion $s(x) = 3x^2 - 2x^3$ (durchgehende Kurve) [7]; modifizierte Uberblendungsfunktion $\tilde{s}(x) = 10x^3 - 15x^4 + 6x^5$ (unterbrochene Kurve) [9].

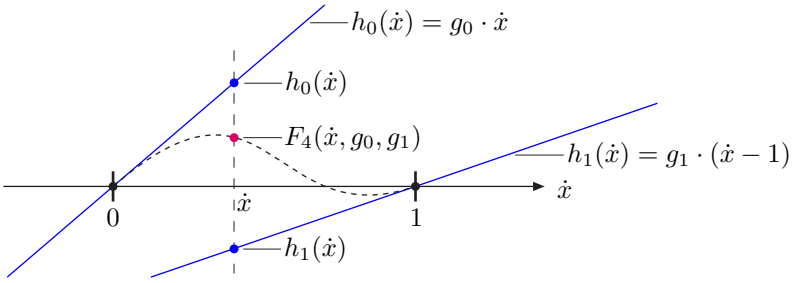


Abbildung 5: Interpolation der Tangentialgeraden im $[0, 1]$-Intervall. Die Tangentialgeraden h_0 und h_1 verlaufen durch die Punkte $x = 0$ bzw. $x = 1$ und ihre Steigungen sind bestimmt durch die zugehörigen Gradienten g_0 bzw. g_1. Für einen spezifischen Punkt $\dot{x}$ im Intervall $[0, 1]$ ergibt sich der interpolierte Funktionswert durch Gewichtung der Geradenwerte $h_0(\dot{x})$ und $h_1(\dot{x})$ mithilfe der S-förmigen Überblendungsfunktion $s(\dot{x})$ in der Form $F_4(\dot{x}, g_0, g_1) = (1 - s(\dot{x})) \cdot h_0(\dot{x}) + s(\dot{x}) \cdot h_1(\dot{x})$.

anstelle von Gl. 24 als Überblendungsfunktion ein Polynom *fünften* Grades eingesetzt, konkret

$$\tilde{s}(x) = 10x^3 - 15x^4 + 6x^5 \tag{27}$$

$$= x^3 \cdot (x \cdot (x \cdot 6 - 15) + 10). \tag{28}$$

(Der Ausdruck in Gl. 28 wird wegen des geringeren Rechenaufwands für die Implementierung verwendet.) Diese Überblendfunktion ist in Abb. 4 im Vergleich zur ursprünglichen Version gezeigt. Damit wird die Perlin'sche In-

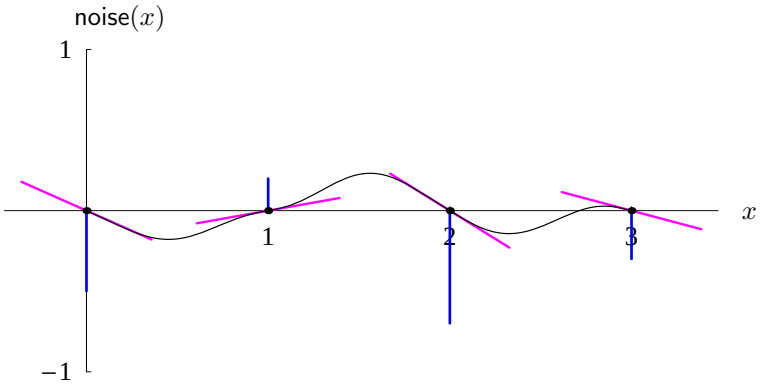


Abbildung 6: Anwendung der modifizierten Perlin-Interpolationsfunktion [9] nach Gl. 29 für die ersten vier Gradientenwerte aus Abb. 1.

terpolationsfunktion (Gl. 17) modifiziert zu

$$\begin{aligned} F_5(\dot{x}, g_0, g_1) &= h_0(\dot{x}) + \big(10\dot{x}^3 - 15\dot{x}^4 + 6\dot{x}^5\big) \cdot \big(h_1(\dot{x}) - h_0(\dot{x})\big) \\ &= g_0 \cdot \dot{x} + \big(10\dot{x}^3 - 15\dot{x}^4 + 6\dot{x}^5\big) \cdot \big(g_1 \cdot (\dot{x} - 1) - g_0 \cdot \dot{x}\big) \\ &= (-6g_0 + 6g_1) \cdot \dot{x}^6 + (15g_0 - 21g_1) \cdot \dot{x}^5 \\ &\qquad + (-10g_0 + 25g_1) \cdot \dot{x}^4 + (-10g_1)\dot{x}^3 + g_0 \cdot \dot{x} \end{aligned} \tag{29}$$

für $0 \leq \dot{x} \leq 1$, und somit ein Polynom *sechsten* Grades (in $\dot{x}$). Die Funktion in Gl. 29 erfüllt nicht nur die vier Bedingungen in Gl. 5, sondern hat zusätzlich an den Endstellen auch die Eigenschaft

$$F_5''(0, g_0, g_1) = F_5''(1, g_0, g_1) = 0, \tag{30}$$

für beliebige g_0, g_1.[7] Damit ist die Krümmung der Funktion an beiden Enden des $[0, 1]$-Intervalls null und es ergeben sich C2-kontinuierliche Übergänge zwischen benachbarten Segmenten (das Ergebnis der ursprüngliche Perlin-Interpolation in Gl. 17 ist hingegen nur C1-kontinuierlich).

2.1.7 Zusammenfassung

Gegeben ist eine diskrete Folge von zufällig gewählten aber fixen, im Intervall $[-1, 1]$ gleichverteilten Gradientenwerten g_i, mit $i \in \mathbb{Z}$. Die Werte der zugehörigen eindimensionalen Perlin-Funktion $\mathsf{noise}(x)$ an einer beliebigen Position $x \in \mathbb{R}$ erhält man durch stückweise Interpolation mit der lokalen Interpolationsfunktion $F(\dot{x}, g_0, g_1)$ in der Form

$$\mathsf{noise}(x) = F(x - p, g_p, g_{p+1}), \tag{31}$$

[7] $F_5''(\dot{x}, \ldots)$ bezeichnet die zweite Ableitung von F_5 an der Stelle $\dot{x}$.

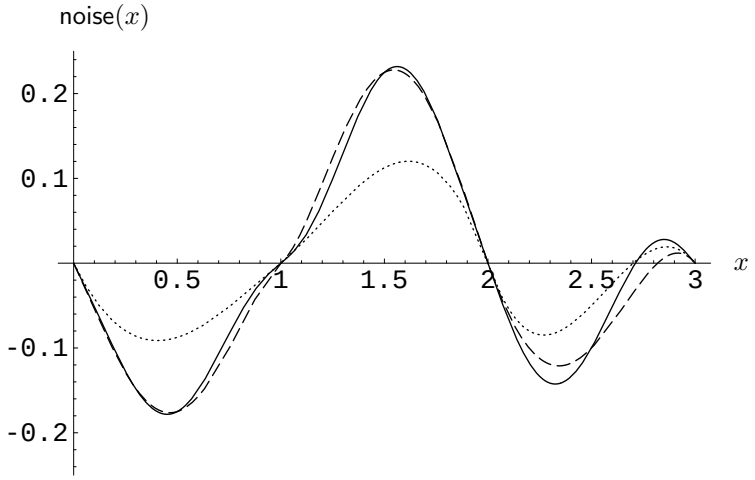


Abbildung 7: Vergleich zwischen unterschiedlichen Interpolationsfunktionen. Kubisches Polynom nach Gl. 13 (punktierte Kurve); ursprüngliche Perlin-Interpolation nach Gl. 23 (unterbrochene Kurve); modifizierte Perlin-Interpolation nach Gl. 29 (durchgehende Kurve).

mit $p = \lfloor x \rfloor$. Dieser Vorgang ist zur besseren Übersicht auch in Alg. 1 nochmals zusammengefasst. Darin sind die Tangenten*werte* an der Position $\dot{x}$ mit

$$w_0 = h_0(\dot{x}) = g_0 \cdot \dot{x} \qquad \text{und} \tag{32}$$

$$w_1 = h_1(\dot{x} - 1) = g_1 \cdot (\dot{x} - 1) \tag{33}$$

bezeichnet, und die eigentliche Interpolation übernimmt die Funktion

$$\mathsf{interpolate}(\dot{x}, w_0, w_1) \equiv F\big(\dot{x}, g_0 \cdot \dot{x}, g_1 \cdot (\dot{x} - 1)\big). \tag{34}$$

Für die Interpolation wird die modifizierte Überblendungsfunktion $\tilde{s}(x)$ aus Gl. 27–28 verwendet und somit die Perlin-Funktion $F_5()$ aus Gl. 29 implementiert.

2.2 Frequenz der Rauschfunktion

Der realistische Eindruck von Perlin Noise basiert zu einem guten Teil auf der geschickten Kombination von Rauschfunktionen unterschiedlicher Frequenzen. Die einfache Rauschfunktion $\mathsf{noise}(x)$ hat regelmäßige Nulldurchgänge im Abstand $\tau = 1$ und besitzt somit eine inherente Periodizität mit der Frequenz

$$f = \frac{1}{\tau} = 1 \tag{35}$$

Da das Intervall τ eine einheitslose Größe darstellt, können wir natürlich auch für f keine konkrete Einheit angeben (s. auch [2, 3, Abschn. 13.3.4]).

Algorithmus 1: Generierung einer eindimensionalen Perlin Noise-Funktion. Die deterministische Funktion $\mathsf{grad}(i)$ liefert für jeden ganzzahligen Punkt $i \in \mathbb{Z}$ einen zufälligen Gradienwert im Intervall $[-1, 1]$. $\tilde{s}(x)$ ist Überblendungsfunktion aus Gl. 24 oder die (hier verwendete) modifizierte Version aus Gl. 27.

1: $\mathsf{noise}(x)$ ▷ $x \in \mathbb{R}$
   Returns the value of the one-dimensional Perlin noise function for position $x \in \mathbb{R}$.
2: $p \leftarrow \lfloor x \rfloor$
3: $g_0 \leftarrow \mathsf{grad}(p)$ ▷ gradient at position p
4: $g_1 \leftarrow \mathsf{grad}(p+1)$ ▷ gradient at position $p+1$
5: $\dot{x} = x - p$ ▷ $\dot{x} \in [0, 1]$
6: $w_0 \leftarrow g_0 \cdot \dot{x}$ ▷ tangent value at position 0
7: $w_1 \leftarrow g_1 \cdot (\dot{x} - 1)$ ▷ tangent value at position 1
8: $w \leftarrow \mathsf{interpolate}(\dot{x}, w_0, w_1)$
9: **return** w
10: **end**

11: $\mathsf{interpolate}(\dot{x}, w_0, w_1)$ ▷ $\dot{x}, w_0, w_1 \in \mathbb{R}$
   Returns the locally interpolated noise function value for the interval $\dot{x} \in [0, 1]$, given the tangent values w_0, w_1.
12: $s_x \leftarrow \tilde{s}(\dot{x})$ ▷ blending function: $\tilde{s}(x) = 10x^3 - 15x^4 + 6x^5$
13: **return** $(1 - s_x) \cdot w_0 + s_x \cdot w_1$
14: **end**

Tatsächlich ist die resultierende Rauschfunktion $\mathsf{noise}(x)$ selbst *bandbegrenzt* mit einer Maximalfrequenz von ungefähr 1 [4, S. 68].[8]

Die gleiche Funktion, jedoch mit der *doppelten* Frequenz $f = 2$, erhält man durch

$$\mathsf{noise}^{(2)}(x) = \mathsf{noise}(2 \cdot x),$$

also durch „Stauchung“ der Funktion um den Faktor 2. Analog erhält man die Funktion mit der Frequenz $f = 3$ durch

$$\mathsf{noise}^{(3)}(x) = \mathsf{noise}(3 \cdot x),$$

und so weiter. Abbildung 8 zeigt die in den bisherigen Beispielen verwendete Funktion mit den Frequenzen $f = 1, 2, 3, 4$. Genauso kann man durch Dehnung der Funktion die Frequenz entsprechend reduzieren, beispielsweise auf die Hälfte durch

$$\mathsf{noise}^{(0.5)}(x) = \mathsf{noise}(0.5 \cdot x).$$

[8] Dieser Umstand ist u. a. für die Erzeugung der diskreten Rauschfunktion wichtig (siehe Abschn. 2.3.1).

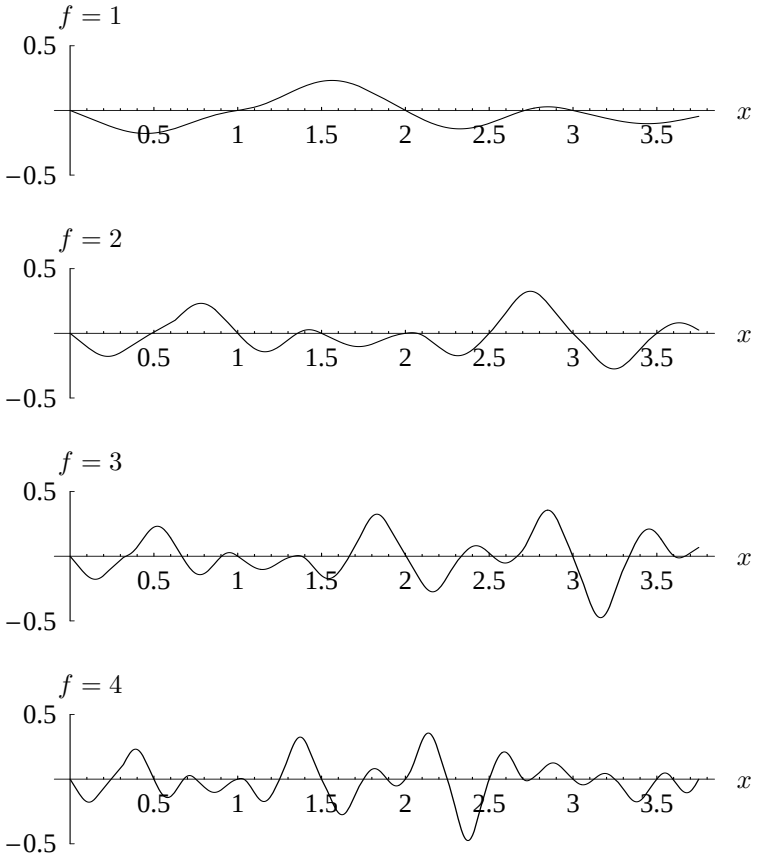


Abbildung 8: Beispiele für eindimensionale Perlin-Rauschfunktionen der Form $\mathsf{noise}^{(f)}(x) = \mathsf{noise}(f \cdot x)$ mit unterschiedlichen Frequenzen $f = 1, \ldots 4$.

Im Allgemeinen erhält man die Funktion $\mathsf{noise}(x)$ mit einer beliebigen Frequenz $f \in \mathbb{R}$ durch

$$\mathsf{noise}^{(f)}(x) = \mathsf{noise}(f \cdot x). \tag{36}$$

2.3 Kombination mehrerer Rauschfunktionen

Bei der Methode von Perlin werden mehrere skalierte Versionen *derselben* Rauschfunktion mit M unterschiedlichen **Frequenzen** f_i und **Amplituden** a_i kombiniert in der Form

$$\mathsf{Noise}(x) = \sum_{i=0}^{M-1} a_i \cdot \mathsf{noise}^{(f_i)}(x) = \sum_{i=0}^{M-1} a_i \cdot \mathsf{noise}(f_i \cdot x). \tag{37}$$

Dabei werden üblicherweise nicht *beliebige* Frequenzen verwendet, sondern – ausgehend von einer bestimmten Basisfrequenz $f_{\min}$ – eine Folge von jeweils

um den Faktor 2 erhöhter Frequenzen

$$f_0 = f_{\min}, \quad f_1 = 2 \cdot f_{\min}, \quad f_2 = 4 \cdot f_{\min}, \quad f_3 = 8 \cdot f_{\min}, \quad \ldots,$$

im Allgemeinen also

$$f_n = 2^n \cdot f_{\min}, \qquad \text{für } n = 0, 1, 2, 3, \ldots \tag{38}$$

Aufeinanderfolgende Frequenzen f_n und f_{n+1} stehen also im Verhältnis $1 : 2$ und bilden zueinander „Oktaven“.

2.3.1 Nutzbarer Frequenzbereich

Wie viele oder welche Frequenzen sind überhaupt praktikabel oder sinnvoll? Bei der Generierung *diskreter* Rauschfunktionen ist insbesondere das Abtasttheorem zu beachten, wonach Signalfrequenzen über der Hälfte der Abtastfrequenz nicht zulässig sind. Erfolgt die Abtastung der kontinuierlichen Rauschfunktion im Abstand von $\tau_s = 1$ (s steht für *sample*), so ist auch die zugehörige Abtastfrequenz $f_s = 1$. Die maximal zulässige Frequenz in der Rauschfunktion sollte demnach nicht über $\frac{1}{2}$ liegen, d. h.

$$f_{\max} \leq 0.5. \tag{39}$$

Frequenzen über 0.5 führen zu Aliasing-Effekten und sind daher nicht sinnvoll. Bei vorgesehenen K Einzelfrequenzen (Oktaven) ist demnach die höchste zulässige *Basisfrequenz*

$$f_{\min} \leq \frac{0.5}{2^{K-1}} = \frac{1}{2^{(K)}}\,. \tag{40}$$

Also ist beispielsweise bei $K = 4$ Oktaven die maximale sinnvolle Basisfrequenz

$$f_{\min} = f_0 = \frac{1}{2^4} = \frac{1}{16}, \tag{41}$$

wobei in der Folge $f_1 = 2 \cdot f_0 = \frac{1}{8}$, $f_2 = 2 \cdot f_1 = \frac{1}{4}$ und schließlich (nur um sicher zu gehen)

$$f_{\max} = f_3 = 2 \cdot f_2 = \frac{1}{2}\,. \tag{42}$$

2.3.2 Amplituden

Die Wahl der zugehörigen Amplitudenwerte a_n ist mitentscheidend für das Aussehen der kombinierten Rauschfunktion $\mathsf{Noise}(x)$. Grundsätzlich können beliebige Amplitudenwerte gewählt werden. In Perlin's ursprünglicher Formulierung [7] sind mit steigender Frequenz logarithmisch abnehmende Amplitudenwerte a_n vorgesehen, spezifiziert durch

$$a_n = \phi^n, \quad \text{mit } 0 < \phi < 1, \tag{43}$$

Algorithmus 2: Generierung einer eindimensionalen Perlin Noise-Funktion mit mehreren Frequenzen.

1: $\mathsf{Noise}(x, f_{\min}, f_{\max}, \phi)$ ▷ $x, f_{\min}, f_{\max}, p \in \mathbb{R}$
   Returns the value of the one-dimensional, multi-frequency Perlin noise function for position x. $f_{\min}, f_{\max}$ specify the frequency range, ϕ denotes the *persistence* value for attenuating the amplitudes $(0 < \phi < 1)$.
2: $sum \leftarrow 0$
3: $n \leftarrow 0$ ▷ $n = 0, 1, \ldots$
4: $f \leftarrow f_{\min}$ ▷ frequency $f_n \leftarrow f_{\min} = f_0$
5: $a \leftarrow 1$ ▷ amplitude $a_n \leftarrow a_0 = \phi^0$
6: **while** $f \leq f_{\max}$ **do**
7: $sum \leftarrow sum + a \cdot \mathsf{noise}(f \cdot x)$ ▷ noise() as defined in Alg. 1
8: $n \leftarrow n + 1$
9: $f \leftarrow 2 \cdot f$ ▷ $f_n = 2^n \cdot f_{\min}$
10: $a \leftarrow \phi \cdot a$ ▷ $a_n = \phi^n$
11: **end while**
12: **return** *sum*.
13: **end**

wobei der konstante Wert $\phi \in \mathbb{R}$ als *Persistence* bezeichnet wird. Typische Werte für ϕ sind $0.25 \ldots 0.5$. Die Generierung einer eindimensionalen Rauschfunktion ist in Alg. 2 nochmals übersichtlich zusammengestellt. Abbildung 9 zeigt ein Beispiel einer kombinierten Funktion bestehend aus drei Teilfunktionen mit $f_0 = 1$ und $\phi = 0.5$.

2.4 Erzeugung der „zufälligen“ Gradientenwerte

Bisher noch offen ist die Frage, wie die pseudo-zufälligen Gradientenwerte $g_i = \mathsf{grad}(i)$ für beliebige $i \in \mathbb{Z}$ erzeugt werden. Die Abbildung $\mathsf{grad}(i)$ ist für eine bestimmte Rauschfunktion *fix*, d. h. $\mathsf{grad}(i)$ muss für ein bestimmtes i auch bei wiederholter Anwendung immer den selben Wert liefern. Es ist also nicht zulässig, bei jedem Aufruf von $\mathsf{grad}(i)$ einen beliebigen neuen Zufallswert zu berechnen.

2.4.1 Verwendung einer Hash-Funktion

Die übliche Methode ist die „on-the-fly“ Berechnung der Gradientenwerte für einen gegebenen Index i mittels einer Hash-Funktion

$$\mathrm{hash}(i) : \mathbb{Z} \rightarrow [0, 1],$$

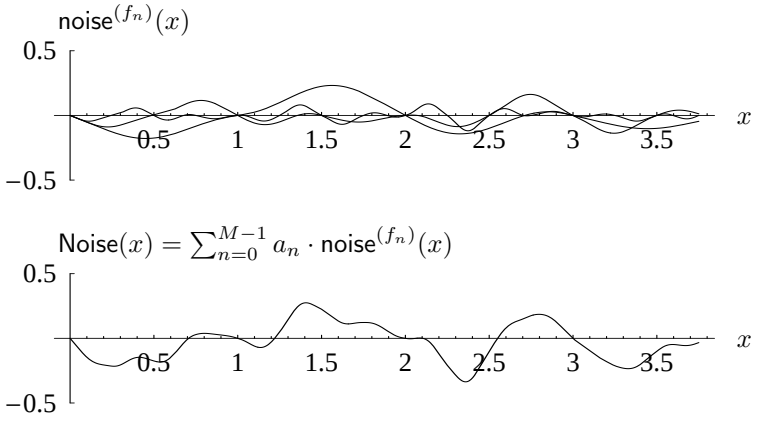


Abbildung 9: Kombinierte Perlin Noise-Funktion $\mathsf{Noise}(x)$, bestehend aus $M = 3$ gewichteten Teilfunktionen $a_n \cdot \mathsf{noise}^{(f_n)}(x)$, mit $n = 0, 1, 2$, den Frequenzen $f_0 = 1$, $f_1 = 2$, $f_2 = 4$ und den Amplituden $a_0 = 1$, $a_1 = \frac{1}{2}$, $a_2 = \frac{1}{4}$ (Persistenz $\phi = \frac{1}{2}$).

die für jedes beliebige $p \in \mathbb{Z}$ wiederholbar und eindeutig einen pseudo-zufälligen, reellen Wert im Intervall $[0, 1]$ liefert. Die durch

$$\mathsf{grad}(i) = 2 \cdot \mathrm{hash}(i) - 1, \tag{44}$$

erzeugten Gradientenwerte liegen somit gleichverteilt im Intervall $[-1, 1]$. Perlin verwendet dafür eine spezielle Hashmethode, die auf der wiederholten Anwendung einer kurzen, „zufälligen“ Permutationstabelle beruht. Details zu diesem Verfahren und alternativen Hashfunktionen finden sich nachfolgend in Abschnitt 5.

2.4.2 Verwendung eines vorberechneten Arrays

Eine Alternative zur Verwendung einer Hashfunktion ist, alle notwendigen Gradientenwerte *einmal* mithilfe eines herkömmlichen Zufallsgenerators zu berechnen und in einer entsprechenden Tabelle

$$G[i] \leftarrow \mathrm{random}(-1, 1)$$

abzulegen und anschließend durch einfachen Tabellenzugriff

$$\mathsf{grad}(i) := G[i]$$

beliebig oft und ohne Neuberechnung zu verwenden. Ein Vorteil ist u. a., dass wir zur Initialisierung des Arrays einen beliebigen Standard-Zufallsgenerator verwenden können und uns daher um die Periodenlänge keine Gedanken machen müssen.

Wie groß muss das Array G sein? Angenommen, wir möchten eine eindimensionale Rauschfunktion $\mathsf{noise}^{(f)}(x)$ mit der Frequenz f im Intervall $x \in [0, x_{\max}]$ erzeugen. Zur Berechnung des Funktionswerts

$$\mathsf{noise}^{(f)}(x_{\max}) = \mathsf{noise}(f \cdot x_{\max})$$

benötigen wir die Gradientenwerte g_p, g_{p+1} mit

$$p = \lfloor f \cdot x_{\max} \rfloor.$$

Wir benötigen daher ein Array mit den Gradientenwerten $(g_0, g_1, \ldots g_K)$ mit

$$K = \lfloor f \cdot x_{\max} \rfloor + 1,$$

also ein Array der Länge $\lfloor f \cdot x_{\max} \rfloor + 2$. Halten wir uns an die Konvention in Gl. 39, dass nämlich die Frequenz f nicht über 0.5 liegen soll, dann ist die Anzahl der erforderlichen Gradientenwerte etwa die halbe Länge der zu diskreten Rauschfunktion.

Bei mehrdimensionalen Rauschfunktionen steigt allerdings die Zahl der Gitterpunkte und damit die Größe der erforderlichen Tabellen rasch an, so dass diese Methode in der Praxis kaum attraktiv ist.

3 Zweidimensionale Rauschfunktionen

Zweidimensionale Noise-Funktionen eignen sich z. B. zur Generierung von zufälligen Texturbildern. Die Vorgangsweise ist analog zum eindimensionalen Fall. Es werden zweidimensionale Rauschfunktionen mit unterschiedlicher Grundfrequenz erzeugt und additiv kombiniert.

Zunächst betrachten wir wiederum eine (diesmal zweidimensionale) Noise-Funktion

$$\mathsf{noise}(x, y) : \mathbb{R}^2 \to \mathbb{R}, \tag{45}$$

mit fixer Grundfrequenz $f = 1$ in beiden Dimensionen. Wiederum sind die ganzzahligen Gitterpunkte $(u, v) \in \mathbb{Z}^2$ auch Nullstellen der Funktion mit den (zufällig) spezifizierten Gradienten

$$\mathsf{grad}(u, v) : \mathbb{Z}^2 \to \mathbb{R}^2,$$

beziehungsweise $\boldsymbol{g}_{i,j}$, analog zur bisher verwendeten Kurzschreibweise. Jeder örtliche Gradient $\boldsymbol{g}_{u,v}$ ist dabei ein zweidimensionaler *Vektor*

$$\boldsymbol{g}_{u,v} = \begin{pmatrix} \mathsf{grad}_x(u, v) \\ \mathsf{grad}_y(u, v) \end{pmatrix}, \tag{46}$$

dessen Elemente

$$\mathsf{grad}_x(u, v) = \frac{\partial\, \mathsf{noise}}{\partial x}(u, v) \qquad \text{bzw.} \qquad \mathsf{grad}_y(u, v) = \frac{\partial\, \mathsf{noise}}{\partial y}(u, v) \tag{47}$$

die partiellen Ableitungen der zweidimensionalen Noise-Funktion in x- bzw. y-Richtung sind. Diese definieren eine *Tangentialebene* mit der Steigung $\mathsf{grad}_x(u,v)$ in x-Richtung bzw. $\mathsf{grad}_y(u,v)$ in y-Richtung, die die x/y-Ebene im Punkt (u,v) schneidet, mit der Gleichung

$$h_{u,v}(x,y) = \boldsymbol{g}_{u,v}^T \cdot \left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} u \\ v \end{pmatrix} \right] = \boldsymbol{g}_{u,v}^T \cdot \begin{pmatrix} x-u \\ y-v \end{pmatrix} \tag{48}$$

$$= \mathsf{grad}_x(u,v) \cdot (x-u) + \mathsf{grad}_y(u,v) \cdot (y-v), \tag{49}$$

wobei mit $\cdot$ in Gl. 48 das Skalarprodukt (oder *innere* Produkt) der beiden Vektoren gemeint ist.[9]

Wie im eindimensionalen Fall ergibt sich die kontinuierliche, zweidimensionale Rauschfunktion $\mathsf{noise}(x,y)$ für reellwertige Koordinaten x, y durch Interpolation zwischen den Gradienten an den ganzzahligen Rasterpunkten (u,v). Es ist naheliegend, die lokale Interpolation wie im eindimensionalen Fall auf das Einheitsquadrat $[0,1]^{\mathbb{R}}$ zu beschränken, mit den normalisierten Koordinaten

$$\dot{x} = x - p_x \qquad \text{und} \quad \dot{y} = y - p_y, \tag{50}$$

wobei $p_x = \lfloor x \rfloor$ und $p_y = \lfloor y \rfloor$. Der Interpolationswert für eine gegebene Position (x,y) berücksichtigt die vier unmittelbar umgebenden Rastpunkte

$$\boldsymbol{g}_{00} = \mathsf{grad}(p_x, p_y), \qquad \boldsymbol{g}_{10} = \mathsf{grad}(p_x+1, p_y), \tag{51}$$

$$\boldsymbol{g}_{01} = \mathsf{grad}(p_x, p_y+1), \qquad \boldsymbol{g}_{11} = \mathsf{grad}(p_x+1, p_y+1), \tag{52}$$

Die Gleichungen $h_{00} \ldots h_{11}$ Tangentialebenen an den vier Gitterpunkten des Einheitsquadrats sind (nach Gl. 48)

$$h_{00}(\dot{x},\dot{y}) = \boldsymbol{g}_{00}^T \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right], \quad h_{10}(\dot{x},\dot{y}) = \boldsymbol{g}_{10}^T \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right], \tag{53}$$

$$h_{01}(\dot{x},\dot{y}) = \boldsymbol{g}_{01}^T \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right], \quad h_{11}(\dot{x},\dot{y}) = \boldsymbol{g}_{11}^T \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right], \tag{54}$$

und die zugehörigen Tangentenwerte sind

$$w_{00} = h_{00}(\dot{x},\dot{y}), \qquad w_{10} = h_{10}(\dot{x},\dot{y}), \tag{55}$$

$$w_{01} = h_{01}(\dot{x},\dot{y}), \qquad w_{11} = h_{11}(\dot{x},\dot{y}). \tag{56}$$

Die Interpolation erfolgt wiederum stückweise über die Tangentenwerte $w_{00} \ldots w_{11}$ und – ähnlich einer bilinearen Interpolation, jedoch unter Verwendung der

[9] Vergleiche dazu Gl. 19 zur Beschreibung der Tangentialgeraden im eindimensionalen Fall.

S-förmigen Überblendungsfunktion (analog zu Gl. 26) – in zwei Schritten: zunächst entlang der x-Richtung mit den Zwischenergebnissen

$$w_0 = (1 - s(\dot{x})) \cdot w_{00} + s(\dot{x}) \cdot w_{10}, \tag{57}$$
$$w_1 = (1 - s(\dot{x})) \cdot w_{01} + s(\dot{x}) \cdot w_{11}, \tag{58}$$

und anschließend entlang der y-Richtung mit dem Endergebnis

$$\mathsf{noise}(x, y) = w = (1 - s(\dot{y})) \cdot w_0 + s(\dot{y}) \cdot w_1. \tag{59}$$

Die eindimensionale Überblendungsfunktion $s(\cdot)$ ist dabei entweder die ursprüngliche (kubische) Perlin-Funktion aus Gl. 24 oder die modifizierte Version $\tilde{s}(\cdot)$ aus Gl. 27.

Der gesamte Interpolationsvorgang für den zweidimensionalen Fall ist in Alg. 3 nochmals übersichtlich zusammengefasst. Abbildung 10 zeigt ein Beispiel für die interpolierte Rauschfunktion $\mathsf{noise}(x, y)$ für vier zufällig gewählte Gradientenvektoren $\boldsymbol{g}_{00} \ldots \boldsymbol{g}_{11}$.

Die Erzeugung einer kombinierten Rauschfunktion über mehrere Frequenzen f_n erfolgt analog zum eindimensionalen Fall (Gl. 37) in der Form

$$\mathsf{Noise}(x, y) = \sum_{i=n}^{M} a_n \cdot \mathsf{noise}^{(f_n)}(x, y) \tag{60}$$
$$= \sum_{i=n}^{M} a_n \cdot \mathsf{noise}(f_n \cdot x, f_n \cdot y)\,, \tag{61}$$

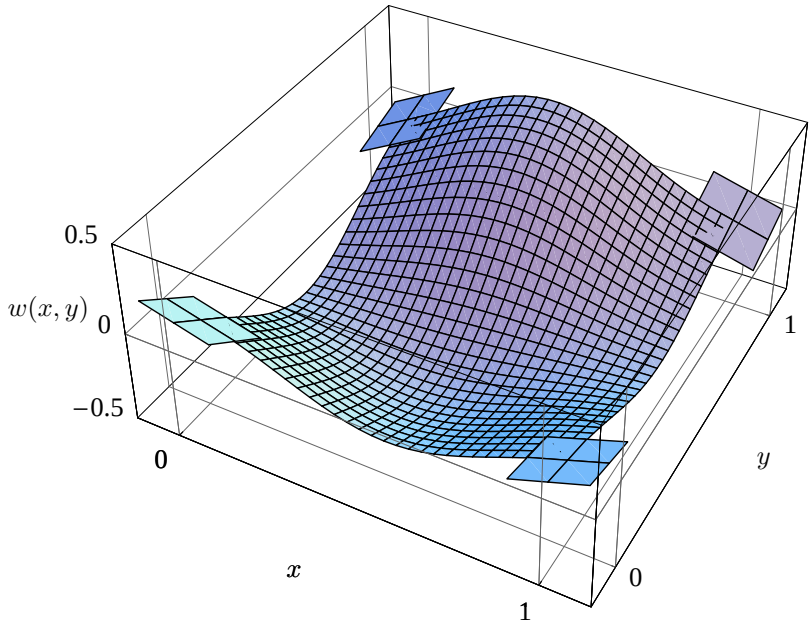
wiederum mit entsprechenden Amplitudenwerten a_n. Für beide Dimensionen wird jeweils die selbe Frequenz f_n verwendet.

3.0.3 Beispiele

Abbildung 11 zeigt Beispiele für zweidimensionalen Perlin-Noise mit unterschiedlicher Anzahl von Frequenzkomponenten. Abbildung 12 verdeutlicht den Einfluss des Persistence-Werts ϕ, also die relative Größe der Amplituden.

4 Erweiterung auf N Dimensionen

Die Erweiterung des Konzepts auf gradientenbasierte Rauschfunktionen im N-dimensionalen Raum ist relativ einfach, allerdings sind die Ergebnisse i. Allg. natürlich nur schwer zu visualisieren. Dreidimensionale Rauschfunktionen werden beispielsweise in der Computergrafik zur Texturierung oder Oberflächenstrukturierung (bump mapping) von Körpern verwendet, wobei die Texturierung nicht nur die sichtbaren Oberflächen, sondern das gesamte Innere des Körpers betrifft. Ein so texturiertes Objekt ist damit nur ein



$$\boldsymbol{g}_{00} = \begin{pmatrix} -0.5 \\ -1.2 \end{pmatrix}, \quad \boldsymbol{g}_{10} = \begin{pmatrix} 0.7 \\ -0.4 \end{pmatrix}, \quad \boldsymbol{g}_{01} = \begin{pmatrix} 1.0 \\ 0.5 \end{pmatrix}, \quad \boldsymbol{g}_{11} = \begin{pmatrix} -0.5 \\ 0.6 \end{pmatrix}.$$

Abbildung 10: 2D Interpolationsfunktion $\mathsf{noise}(x, y)$ (siehe Gl. 59) im Einheitsquadrat $x, y \in [0, 1]$. Die Eckpunkte des Einheitsquadrats sind Gitterpunkte und somit auch Nullstellen der Funktion. Die (zufällig gewählten) Gradienten $\boldsymbol{g}_{00} \ldots \boldsymbol{g}_{11}$ bestimmen den Anstieg der Rauschfunktion in x- und y-Richtung an den Eckpunkten des Einheitsquadrats, veranschaulicht durch die zugehörigen Tangentenebenen. Als Überblendungsfunktion wurde die modifizierte Perlinfunktion $\tilde{s}(\cdot)$ (siehe Gl. 27) verwendet.

Ausschnitt aus einem größeren dreidimensionalen Block, in dem die Rauschfunktion überall definiert ist. Ein typisches Beispiel ist die Anwendung von Perlin Noise für 3D-Texturen, die das Aussehen von Marmor annähern.[10]

Analog zur Situation in 2D ergibt sich die N-dimensionale Rauschfunktion in der Form

$$\mathsf{noise}(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}, \tag{62}$$

wobei

$$\boldsymbol{x} = \big(x_0, x_1, \ldots x_{N-1}\big)^T, \quad \text{mit } x_i \in \mathbb{R},$$

einen kontinuierlicher Koordinatenvektor im N-dimensionalen Raum bezeichnet. Die Funktion $\mathsf{noise}(\boldsymbol{x})$ liefert also für jede Position $\boldsymbol{x} \in \mathbb{R}^N$ einen skalaren Wert.

Wir nehmen weiters an, dass die Grundfrequenz in allen N Dimensionen mit $f = 1$ fixiert ist, also die Rasterung in allen Richtungen gleichförmig

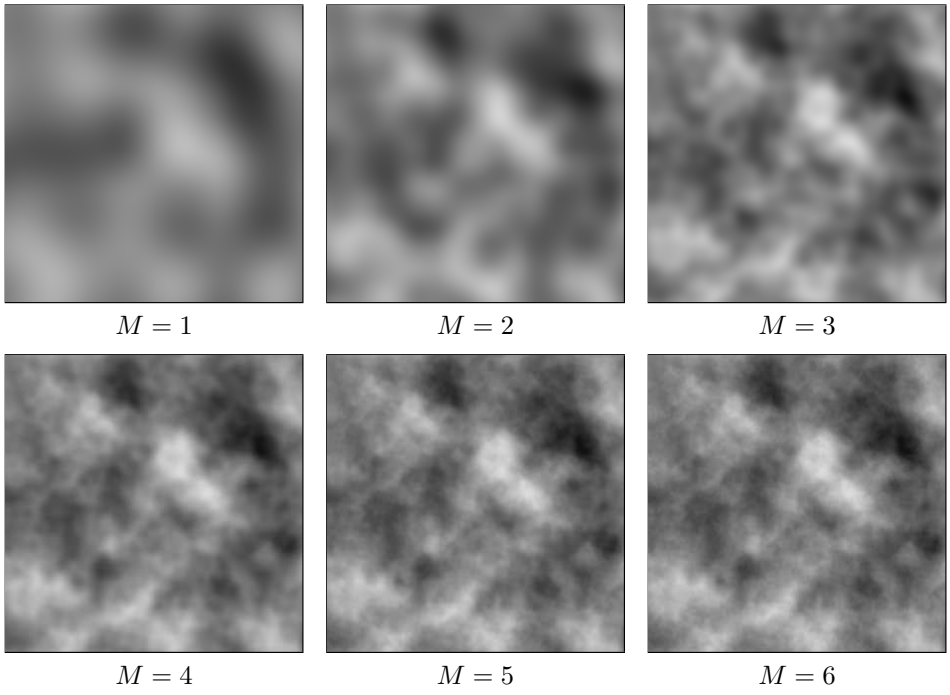
[10] Siehe beispielsweise http://mrl.nyu.edu/~perlin/doc/vase.html.

Algorithmus 3: Zweidimensionale Perlin Noise-Funktion. Die deterministische Funktion $\mathsf{grad}(u, v)$ liefert für jeden Gitterpunkt $(u, v) \in \mathbb{Z}^2$ einen zufälligen, zweidimensionalen Gradientenvektor $\boldsymbol{g} \in \mathbb{R}^2$. $s(x)$ ist die eindimendionale Überblendungsfunktion aus Gl. 24 oder die (hier verwendete) modifizierte Version aus Gl. 27.

1: $\mathsf{noise}(x, y)$ ▷ $x, y \in \mathbb{R}$
   Returns the value of the two-dimensional Perlin noise function for the continuous position (x, y).
2: $p_x \leftarrow \lfloor x \rfloor$
3: $p_y \leftarrow \lfloor y \rfloor$
4: $\boldsymbol{g}_{00} \leftarrow \mathsf{grad}(p_x, p_y)$ ▷ 2D gradient vectors
5: $\boldsymbol{g}_{10} \leftarrow \mathsf{grad}(p_x + 1, p_y)$
6: $\boldsymbol{g}_{01} \leftarrow \mathsf{grad}(p_x, p_y + 1)$
7: $\boldsymbol{g}_{11} \leftarrow \mathsf{grad}(p_x + 1, p_y + 1)$
8: $\dot{x} \leftarrow x - p_x$ ▷ $\dot{x}, \dot{y} \in [0, 1]$
9: $\dot{y} \leftarrow y - p_y$
10: $w_{00} \leftarrow \boldsymbol{g}_{00}^T \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$ ▷ tangent values (dot vector products)
11: $w_{10} \leftarrow \boldsymbol{g}_{10}^T \cdot \begin{pmatrix} \dot{x}-1 \\ \dot{y} \end{pmatrix}$
12: $w_{01} \leftarrow \boldsymbol{g}_{01}^T \cdot \begin{pmatrix} \dot{x} \\ \dot{y}-1 \end{pmatrix}$
13: $w_{11} \leftarrow \boldsymbol{g}_{11}^T \cdot \begin{pmatrix} \dot{x}-1 \\ \dot{y}-1 \end{pmatrix}$
14: $w \leftarrow \mathsf{interpolate}(\dot{x}, \dot{y}, w_{00}, w_{10}, w_{01}, w_{11})$
15: **return** w
16: **end**

17: $\mathsf{interpolate}(\dot{x}, \dot{y}, w_{00}, w_{10}, w_{01}, w_{11})$
   Returns the locally interpolated noise function value for $\dot{x}, \dot{y} \in [0, 1]$, given the tangent values $w_{00}, \ldots w_{11} \in \mathbb{R}$.
18: $s_x \leftarrow s(\dot{x})$ ▷ blending function: $s(x) = 10x^3 - 15x^4 + 6x^5$
19: $s_y \leftarrow s(\dot{y})$
20: $w_0 = (1 - s_x) \cdot w_{00} + s_x \cdot w_{10}$ ▷ interpolate in x-direction ($2\times$)
21: $w_1 = (1 - s_x) \cdot w_{01} + s_x \cdot w_{11}$
22: **return** $(1 - s_y) \cdot w_0 + s_y \cdot w_1$
23: **end**

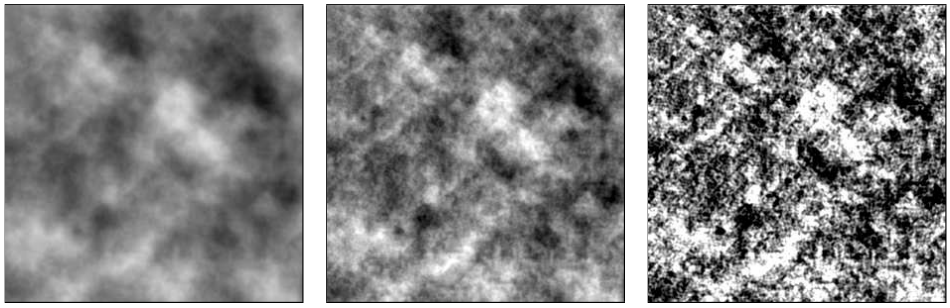
ist. Die ganzzahligen Gitterpunkte $\boldsymbol{p} = (p_0, p_1, \ldots p_{N-1})^T \in \mathbb{Z}^N$ sind wiederum auch Nullstellen der Funktion $\mathsf{noise}(\boldsymbol{x})$ mit den zugehörigen (zufällig



$M = 1$ $M = 2$ $M = 3$

$M = 4$ $M = 5$ $M = 6$

Abbildung 11: Beispiele für 2D Perlin Noise für $M = 1 \ldots 6$ Frequenzen und fixem Persistence-Wert ($f_i = 0.01, 0.02, 0.04, 0.08, 0.16, 0.32,\ \phi = 0.5$).



$p = 0.5$ $p = 0.7$ $p = 1.0$

Abbildung 12: Beispiele für 2D Perlin Noise für verschiedene Persistence-Werte ($\phi = 0.5, 0.7, 1.0$).

bestimmten) lokalen Gradientenvektors

$$\boldsymbol{g}_{\boldsymbol{p}} = \begin{pmatrix} \mathsf{grad}_0(\boldsymbol{p}) \\ \mathsf{grad}_1(\boldsymbol{p}) \\ \vdots \\ \mathsf{grad}_{N-1}(\boldsymbol{p}) \end{pmatrix}, \tag{63}$$

dessen Elemente $\mathsf{grad}_k(\boldsymbol{p})$ die partiellen Ableitungen der kontinuierlichen Rauschfunktion $\mathsf{noise}(\boldsymbol{p})$ in der Richtung x_k am N-dimensionalen Punkt $\boldsymbol{p}$ spezifizieren, d. h.,

$$\mathsf{grad}_k(\boldsymbol{p}) = \frac{\partial\, \mathsf{noise}(\boldsymbol{p})}{\partial x_k}\,. \tag{64}$$

Jeder Gradientenvektor $\boldsymbol{g_p}$ definiert wiederum eine lineare Tangentenfunktion $h_{\boldsymbol{p}}(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}$, mit

$$h_{\boldsymbol{p}}(\boldsymbol{x}) = \boldsymbol{g}_{\boldsymbol{p}}^T \cdot \big(\boldsymbol{x} - \boldsymbol{p}\big) = \sum_{k=0}^{N-1} \mathsf{grad}_k(\boldsymbol{p}) \cdot (x_k - p_k). \tag{65}$$

Die Tangentenfunktion $h_{\boldsymbol{p}}(\boldsymbol{x})$ liefert einen skalaren, reelen Wert für jeden Punkt $\boldsymbol{x} \in \mathbb{R}^N$. Sie bildet eine Tangential-Hyperebene im N+1-dimensionalen Raum.[11] Der Wert der Funktion ist für den zugehörigen Gitterpunkt $\boldsymbol{p}$ selbst null, d. h.,

$$h_{\boldsymbol{p}}(\boldsymbol{p}) = \boldsymbol{g}_{\boldsymbol{p}}^T \cdot \big(\boldsymbol{p} - \boldsymbol{p}\big) = \boldsymbol{g}_{\boldsymbol{p}}^T \cdot \boldsymbol{0} = 0,$$

zumal ja die Gitterpunkte als Nullstellen der kontinuierliche Rauschfunktion vorgegeben sind.

Wie im ein- und zweidimensionalen Fall ergibt sich die kontinuierliche Rauschfunktion $\mathsf{noise}(\boldsymbol{x})$ – für kontinuierliche Koordinaten $\boldsymbol{x} \in \mathbb{R}^N$ – auch im N-dimensionalen Fall durch Interpolation zwischen den zu den umliegenden Gitterpunkten $\boldsymbol{p}_j$ gehörigen Tangentialfunktionen $h_{\boldsymbol{p}_j}(\boldsymbol{x})$.

Wurden in 1D jeweils zwei und in 2D jeweils vier Nachbarpunkte (bzw. deren Gradienten) berücksichtigt, so sind im dreidimensionalen Fall die $2^3 = 8$ Eckpunkte des $\boldsymbol{x}$ umgebenden Würfels in die Interpolation einzubeziehen. Im allgemeinen, N-dimensionalen Fall sind jeweils 2^N umliegenden Gitterpunkte $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots \boldsymbol{p}_{2^N-1}$ zu berücksichtigen. Diese Gitterpunkte bilden einen N-dimensionalen Einheits-Hyperwürfel (*Hypercube*) um die kontinuierliche Koordinate $\boldsymbol{x}$. Ausgehend vom kanonischen Eckpunkt

$$\boldsymbol{p}_0 = \begin{pmatrix} \lfloor x_0 \rfloor \\ \lfloor x_1 \rfloor \\ \vdots \\ \lfloor x_{N-1} \rfloor \end{pmatrix}, \tag{66}$$

können die Koordinaten aller 2^N Eckpunkte $\boldsymbol{p}_j$ dieses Hyperwürfels in der Form

$$\boldsymbol{p}_j = \boldsymbol{p}_0 + \boldsymbol{q}_j \tag{67}$$

generiert werden. Die Koordinaten $\boldsymbol{q}_j$ stehen für die Eckpunkte des N-dimensionalen Einheitswürfels, der am Koordinatenursprung verankert ist.

[11] Im eindimensionalen Fall ist dies eine 2D-Gerade; im Fall einer zweidimensionalen Funktion ergibt sich eine Tangentialebene in 3D. Siehe auch Abschnitt 4.2.

Jeder Punkt $\boldsymbol{q}_j$ ist daher ein N-dimensionaler Vektor mit Elementen aus $\{0,1\}$ (von denen es genau 2^N verschiedene gibt), d. h.,

$$\begin{array}{rcll} \boldsymbol{p}_0 &=& \boldsymbol{p}_0 + \boldsymbol{q}_0 &= \boldsymbol{p}_0 + (0,0,0,\ldots,0)^T, \\ \boldsymbol{p}_1 &=& \boldsymbol{p}_0 + \boldsymbol{q}_1 &= \boldsymbol{p}_0 + (1,0,0,\ldots,0)^T, \\ \boldsymbol{p}_2 &=& \boldsymbol{p}_0 + \boldsymbol{q}_2 &= \boldsymbol{p}_0 + (0,1,0,\ldots,0)^T, \\ \boldsymbol{p}_3 &=& \boldsymbol{p}_0 + \boldsymbol{q}_3 &= \boldsymbol{p}_0 + (1,1,0,\ldots,0)^T, \\ \vdots && \vdots && \vdots \\ \boldsymbol{p}_{2^N-1} &=& \boldsymbol{p}_0 + \boldsymbol{q}_{2^N-1} &= \boldsymbol{p}_0 + (1,1,1,\ldots,1)^T. \end{array} \tag{68}$$

Die Elemente des Vektors $\boldsymbol{q}_j$ entsprechen offensichtlich dem binären *Bitmuster* der Zahl j, d. h.,

$$\boldsymbol{q}_j[k] = \mathrm{bit}_k(j) = \left(j \div 2^k\right) \bmod 2, \tag{69}$$

für $0 \le j < 2^N$, $0 \le k < N$.[12] Wir verwenden nachfolgend (in Alg. 4–5) die Funktion $\boldsymbol{q}_j = \mathsf{vertex}(j, N)$ zur Generierung der Eckkoordinaten des N-dimensionalen Einheitswürfels.

Die den 2^N Gitterpunkten $\boldsymbol{p}_j$ zugehörigen Tangentialebenen sind (siehe Gl. 65) definiert durch die linearen Funktionen

$$h_{\boldsymbol{p}_j}(\boldsymbol{x}) = \boldsymbol{g}_{\boldsymbol{p}_j}^T \cdot \left(\boldsymbol{x} - \boldsymbol{p}_j\right), \tag{70}$$

oder, in verkürzter Schreibweise,[13]

$$h_j(\boldsymbol{x}) = \boldsymbol{g}_j^T \cdot \left(\boldsymbol{x} - \boldsymbol{p}_j\right), \tag{71}$$

für $j = 0 \ldots 2^N - 1$.

4.1 Interpolation der Tangentenebenen

Den eigentlichen Wert der N-dimensionalen Rauschfunktion an einer bestimmten Position $\boldsymbol{x} = \left(x_0, x_1, \ldots x_{N-1}\right)^T$ erhalten wir durch Interpolation der Tangentenfunktionen $h_j(\boldsymbol{x})$ sämtlicher 2^N umliegender Rasterpunkte $\boldsymbol{p}_j$, das sind

$$\boldsymbol{w}(\boldsymbol{x}) = \left[w_0, w_1, \ldots, w_{2^N-1}\right], \quad \text{mit } w_j = h_j(\boldsymbol{x}). \tag{72}$$

Diese Interpolation wird jedoch nicht in *einem* Schritt durchgeführt, sondern (wie bereits im zweidimensionalen Fall demonstriert) nacheinander für jede einzelne der N Raumdimensionen.

[12] $a \div b$ steht für die ganzzahlige Division a durch b (Quotient von a, b).

[13] Die verkürzte Schreibweise $h_j(\boldsymbol{x})$ – anstelle von $h_{\boldsymbol{p}_j}(\boldsymbol{x})$ – für die Tangentialfunktion im Rasterpunkt $\boldsymbol{p}_j$ sowie $\boldsymbol{g}_j$ – anstelle von $\boldsymbol{g}_{\boldsymbol{p}_j}$ – für die Gradientenvektoren dient nur zur besseren Lesbarkeit.

Ein N-dimensionaler Hyperwürfel besitzt entlang jeder Dimension k genau 2^{N-1} Kanten, die jeweils zwei seiner Eckpunkte $\boldsymbol{p}_a, \boldsymbol{p}_b$ verbinden. Die Koordinatenvektoren dieser Endpunkte unterscheiden sich nur bezüglich der Dimension k, d. h.,

$$\boldsymbol{p}_a[k] \neq \boldsymbol{p}_b[k] \quad \text{und} \quad \boldsymbol{p}_a[l] = \boldsymbol{p}_b[l] \quad \text{für } l \neq k,\ 0 \leq k < N.$$

In jedem Interpolationsschritt wird zwischen den Werten der Tangentenfunktionen $w_a = h_a(\boldsymbol{x})$ und $w_b = h_b(\boldsymbol{x})$ von zwei in Dimension k benachbarten Eckpunkten $\boldsymbol{p}_a, \boldsymbol{p}_b$ des $\boldsymbol{x}$ umschließenden Hyperwürfels interpoliert, und zwar in der bereits bekannten Form

$$\bar{w}_{a,b} = (1 - s(x_k)) \cdot w_a + s(x_k) \cdot w_b\,, \tag{73}$$

wobei x_k die k-te Komponente von $\boldsymbol{x}$ ist und $s()$ die nichtlineare Überblendungsfunktion (Gl. 24 bzw. Gl. 27) bezeichnet. Bei jeder einzelnen Interpolation werden also die Werte von *zwei* Nachbarknoten auf *einen* Wert reduziert. Die so entstehenden Zwischenwerte können wiederum als Eckpunkte eines Hyperwürfels interpretiert werden, dessen Dimension gegenüber dem ursprünglichen Würfel um eins reduziert ist.

Wir benötigen also N Interpolationsschritte, um die ursprünglich 2^N Knotenwerte eines N-dimensionalen Hyperwürfels auf einen Skalarwert (entspricht Dimension null) zu reduzieren. Es sollte belanglos sein, in welcher Reihenfolge entlang der einzelnen Dimensionen interpoliert wird.[14] Beispielsweise könnte die schrittweise Interpolation in folgender Sequenz über die Dimensionen $k = 0, 1, \ldots N-1$ ausgeführt werden:

$$\begin{aligned}
\boldsymbol{w}^{(N)} &\leftarrow (w_0, \ldots w_{2^N-1}) = (h_j(\boldsymbol{x}), \ldots h_{2^N-1}(\boldsymbol{x})) \\
\boldsymbol{w}^{(N-1)} &\leftarrow \mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N)}, 0) \\
\boldsymbol{w}^{(N-2)} &\leftarrow \mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N-1)}, 1) \\
&\vdots \quad \vdots \\
\boldsymbol{w}^{(N-k-1)} &\leftarrow \mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N-k)}, k) \\
&\vdots \quad \vdots \\
w = \boldsymbol{w}^{(0)} &\leftarrow \mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(1)}, N-1)
\end{aligned} \tag{74}$$

Dabei ist $\boldsymbol{w}^{(n)}$ jeweils ein Vektor der Länge 2^n; die Länge des Vektors $\boldsymbol{w}$ wird also in jedem Schritt um die Hälfte verkleinert. Das finale Ergebnis der Interpolation an der Position $\boldsymbol{x}$, $w = \boldsymbol{w}^{(0)}$, ist ein skalarer Wert (bzw. ein null-dimensionaler Vektor).

[14] Es wäre zu überprüfen, inwieweit konkret bei der Perlin-Interpolation das Ergebnis von der Reihenfolge der Dimensionen abhängt.

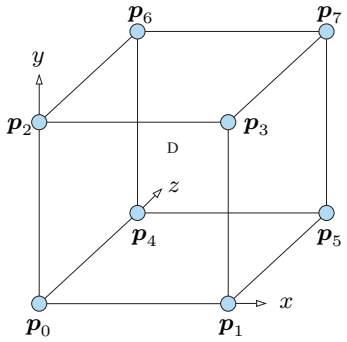


Abbildung 13: Lexikographische Ordnung der Eckpunkte des Einheitswürfels in 3D: $\boldsymbol{p}_0 = \boldsymbol{p}_{000}$, $\boldsymbol{p}_1 = \boldsymbol{p}_{100}$, $\boldsymbol{p}_2 = \boldsymbol{p}_{010}$, …, $\boldsymbol{p}_7 = \boldsymbol{p}_{111}$.

Die Funktion $\mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}, k)$ interpoliert die 2^{N-k} Werte in $\boldsymbol{w}$ für die Dimension k und liefert einen neuen Vektor mit der halben Länge von $\boldsymbol{w}$. Dabei wird angenommen, dass die Werte in $\boldsymbol{w}$ nach ansteigender Dimension „lexikographisch“ sortiert angeordnet sind, also

$$w^{(N-k)} = (w_0, w_1, \ldots w_i, w_{i+1}, \ldots w_{2^{N-k}-1})$$

(siehe Abb. 13). Beispielsweise ergeben sich im dreidimensionalen Fall ($N = 3$), mit den ursprünglichen $2^3 = 8$ Vektorelementen $w_j = w_{xyz}$ ($0 \le j < 8$), die nachstehende Interpolationsfolge und Zwischenergebnisse:

$$\begin{aligned}
\boldsymbol{w}^{(3)} &= (w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7) \\
&= (\underbrace{w_{000}, w_{100}}_{\downarrow\, w_{00}}, \underbrace{w_{010}, w_{110}}_{\downarrow\, w_{10}}, \underbrace{w_{001}, w_{101}}_{\downarrow\, w_{01}}, \underbrace{w_{011}, w_{111}}_{\downarrow\, w_{11}}) \\
\boldsymbol{w}^{(2)} &= (\underbrace{w_{00}, w_{10}}_{\downarrow\, w_0}, \underbrace{w_{01}, w_{11}}_{\downarrow\, w_1}) \\
\boldsymbol{w}^{(1)} &= (\underbrace{w_0, w_1}_{\downarrow\, w})
\end{aligned}$$

Im ersten Interpolationsschritt wird also nach der x-Koordinate (über 4 Wertepaare), im zweiten nach der y-Koordinate (über 2 Wertepaare) und abschließend nach der z-Koordinate (über 1 Wertepaar) interpoliert, mit dem Endergebnis w. Die lexiographische Anordnung gewährleistet, dass die zusammengehörenden Wertepaare innerhalb des Vektors immer nacheinander angeordnet sind. Für zwei aufeinanderfolgende Ergebnisvektoren $\boldsymbol{w}^{(N-k)}$,

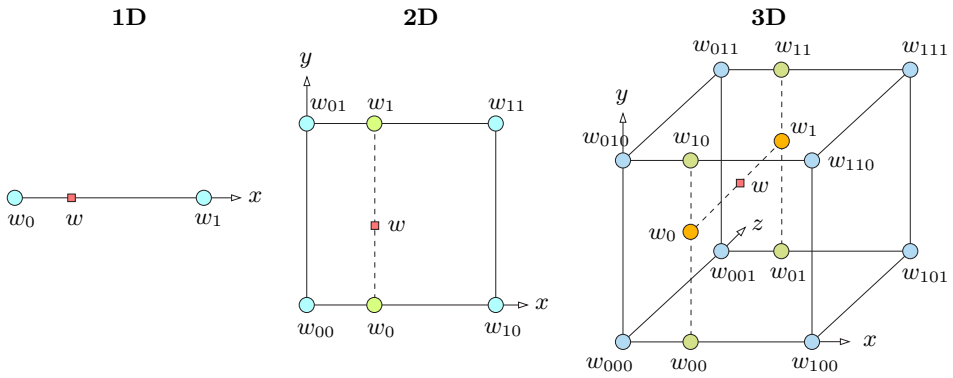


Abbildung 14: Schrittweise Interpolation im ein-, zwei- und dreidimensionalen Fall. Die blauen Knoten repräsentieren die ursprünglichen Tangentenwerte an den 2^N Ecken des N-dimensionalen Würfels. Es wird in jedem Schritt entlang einer Dimension interpoliert. Die grünen Knoten sind die Zwischenwerte nach dem ersten Interpolationsschritt, die gelben Knoten nach dem zweiten Interpolationsschritt. Das rote Quadrat markiert den Endwert für die zugehörige Position.

$\boldsymbol{w}^{(N-k-1)}$ gilt somit

$$w_i^{(N-k-1)} = (1 - s(x_k)) \cdot w_{2i}^{(N-k)} + s(x_k) \cdot w_{2i+1}^{(N-k)}, \tag{75}$$

für $0 \leq k < N$.

Dieser Interpolationsvorgang und die gesamt Rauscherzeugung für den N-dimensionalen Fall sind in Alg. 4 nochmals übersichtlich zusammengefasst. Die oben beschriebene Funktion $\mathsf{interpolate}(\boldsymbol{x}, \boldsymbol{w}, k)$ ist dabei rekursiv ausgeführt; sie lässt sich natürlich (wie in Gl. 74 gezeigt) sehr einfach auch iterativ realisieren.

4.2 Exkurs: Hyperbenen im N-dimensionalen Raum

Die Behauptung ist, dass eine lineare Funktion $f(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}$ der Form

$$y = f(x_0, x_1, \ldots x_{N-1}) = a + a_0 \cdot x_0 + a_1 \cdot x_1 + \ldots + a_{N-1} \cdot x_{N-1}, \tag{76}$$

(mit $a_k \in \mathbb{R}$) eine Hyperebene in $\mathbb{R}^{N+1}$ darstellt. Durch willkürliche Umbenennung von y in die zusätzliche freie Variable x_{N+1} erhalten wir aus Gl. 76

$$a + a_1 \cdot x_1 + \ldots + a_{N-1} \cdot x_{N-1} - \underbrace{x_N}_{f(x_1, \ldots x_{N-1})} = 0, \tag{77}$$

und dies entspricht der allgemeinen Gleichung einer Ebene in $\mathbb{R}^{N+1}$,

$$a + a_0 \cdot x_0 + \ldots + a_{N-1} \cdot x_{N-1} + a_N \cdot x_N = 0, \tag{78}$$

Algorithmus 4: N-dimensionale Perlin Noise-Funktion. Die deterministische Funktion $\mathsf{grad}(\boldsymbol{p})$ liefert für jeden Gitterpunkt $\boldsymbol{p}_j \in \mathbb{Z}^N$ einen zufälligen, N-dimensionalen Gradientenvektor $\boldsymbol{g}_j \in \mathbb{R}^N$. Die Funktion $\mathsf{vertex}()$ ist in Alg. 5 definiert. $s(x)$ ist die eindimensionale Überblendungsfunktion aus Gl. 24 oder (wie hier verwendet) Gl. 27.

1: $\mathsf{noise}(\boldsymbol{x})$ ▷ $\boldsymbol{x} \in \mathbb{R}^N, \mathsf{noise}(\boldsymbol{x}) \in \mathbb{R}$
Returns the (scalar) value of the Perlin noise function for the continuous, N-dimensional position $\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1})$.
2: $N \leftarrow \mathrm{Dim}(\boldsymbol{x})$ ▷ dimension of space
3: $K \leftarrow 2^N$ ▷ number of hypercube vertices
4: $\boldsymbol{p}_0 \leftarrow \lfloor \boldsymbol{x} \rfloor$ ▷ origin of hypercube around $\boldsymbol{x}$
5: $\boldsymbol{Q} \leftarrow (\boldsymbol{q}_0, .. \boldsymbol{q}_{K-1})$, $\boldsymbol{q}_j = \mathsf{vertex}(j, N)$ ▷ unit cube vertices $\in \{0,1\}^N$
6: $\boldsymbol{G} \leftarrow (\boldsymbol{g}_0, .. \boldsymbol{g}_{K-1})$, $\boldsymbol{g}_j = \mathsf{grad}(\boldsymbol{p}_0 + \boldsymbol{q}_j)$ ▷ gradient vectors $\in \mathbb{R}^N$
7: $\dot{\boldsymbol{x}} \leftarrow \boldsymbol{x} - \boldsymbol{p}_0$ ▷ $\dot{\boldsymbol{x}} \in [0,1]^N$
8: $\boldsymbol{w} \leftarrow (w_0, .. w_{K-1})$, $w_j = \boldsymbol{g}_j^T \cdot (\dot{\boldsymbol{x}} - \boldsymbol{q}_j)$ ▷ gradient values $\in \mathbb{R}$
9: **return** $\mathsf{interpolate}(\dot{\boldsymbol{x}}, \boldsymbol{w}, 0)$
10: **end**

11: $\mathsf{interpolate}(\dot{\boldsymbol{x}}, \boldsymbol{w}, d)$
Interpolate the 2^{N-d} values in $\boldsymbol{w}$ at point $\dot{\boldsymbol{x}}$ along dimension d ($\dot{\boldsymbol{x}} \in [0,1]^N$, $0 \le d \le N$).
12: **if** $\mathrm{Dim}(\boldsymbol{w}) = 1$ **then** ▷ done, end of recursion ($d = N$)
13: **return** w_0
14: **else** ▷ $\mathrm{Dim}(\boldsymbol{w}) > 1$
15: $\dot{x} \leftarrow \dot{\boldsymbol{x}}[d]$ ▷ select dimension d of $\dot{\boldsymbol{x}}$
16: **let** $s = 10\dot{x}^3 - 15\dot{x}^4 + 6\dot{x}^5$ ▷ blending function $s(x)$
17: $M \leftarrow \mathrm{Dim}(\boldsymbol{w}) \div 2$
18: **let** $\acute{\boldsymbol{w}}$ be a new vector of size M ▷ $\acute{\boldsymbol{w}}$ is half the size of $\boldsymbol{w}$
19: **for** $i \leftarrow 0 \ldots M-1$ **do** ▷ fill the new vector $\boldsymbol{w}'$
20: $w_a \leftarrow \boldsymbol{w}[2i]$
21: $w_b \leftarrow \boldsymbol{w}[2i+1]$
22: $\acute{\boldsymbol{w}}[i] \leftarrow (1-s) \cdot w_a + s \cdot w_b$ ▷ actual interpolation
23: **end for**
24: **return** $\mathsf{interpolate}(\dot{\boldsymbol{x}}, \acute{\boldsymbol{w}}, d+1)$ ▷ do next dimension
25: **end if**
26: **end**

mit dem Koeffizienten $a_N = -1$. Somit gilt, dass die Punkte

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ f(x_0, x_1, \ldots x_{N-1}) \end{pmatrix}$$

Algorithmus 5: Vertex function.

1: vertex(j, N)
   Returns the coordinate vector for vertex j of the N-dimensional unit-hypercube, which is of length N with elements in $\{0, 1\}$, equivalent to the binary bit-pattern of j. There are 2^N different vertices.
2: **let** $\boldsymbol{v} \leftarrow [v_0, v_1, \ldots, v_{N-1}]$, $v_k = \left(j \div 2^k\right) \bmod 2$ ▷ $v_k \in [0, 1]_{\mathbb{Z}}$
3: return $\boldsymbol{v}$.
4: **end**

eine Ebene in $\mathbb{R}^{N+1}$ bilden, wenn die Funktion $f(\boldsymbol{x})$ linear ist.[15]

5 Hashfunktionen

Hashfunktionen hash() werden im vorhergehenden Abschnitt verwendet, um für einen diskreten ein- oder mehrdimensionalen Gitterpunkt $\boldsymbol{p} \in \mathbb{Z}^N$ einen „zufälligen“ Gradientenvektor $\boldsymbol{g_p} = \mathsf{grad}(\boldsymbol{p})$ der Dimension N zu erzeugen, mit Elementen im Wertebereich $[-1, 1]$. Gleichzeitig sollen die Gradientenvektoren aber insofern deterministisch sein, dass jede mehrmalige Anwendung der Funktion $\mathsf{grad}(\boldsymbol{p})$ für einen bestimmten Gitterpunkt $\boldsymbol{p}$ immer dasselbe Ergebnis liefert.

Betrachten wir zunächst den *eindimensionalen* Fall. Wie bereits in Abschn. 2.4 beschrieben, setzen wir für die Berechnung der zufälligen Gradientenwerte in der Form $\mathsf{grad}(p) = 2 \cdot \mathrm{hash}(p) - 1$ (Gl. 44) eine Hashfunktion

$$\mathrm{hash}(p) : \mathbb{Z} \to [0, 1] \tag{79}$$

voraus, die für ein ganzzahliges Argument p wiederholbar ein bestimmtes reellwertiges Ergebnis im Intervall $[0, 1]$ (gleichverteilt) erzeugt.

Für eine N-dimensionalen Rauschfunktion benötigen wir analog dazu eine Hashfunktion, deren Ergebnis von den Koordinaten der diskreten Rasterpunkte $\boldsymbol{p} \in \mathbb{Z}^N$ abhängig ist und einen N-dimensionalen Zufallsvektor erzeugt, d. h.,

$$\mathrm{hash}(\boldsymbol{p}) : \mathbb{Z}^N \to [0, 1]^N. \tag{80}$$

Das Hash-Ergebnis sollte natürlich zwischen den einzelnen Dimensionen möglichst unkorreliert sein, d. h., wir benötigen eigentlich für jede der N Dimensionen eine eigene Hash-Funktion. Im zweidimensionalen Fall mit $\boldsymbol{p} = (u, v)$ wäre dies konkret

$$\mathrm{hash}(u, v) = \begin{pmatrix} \mathrm{hash}_x(u, v) \\ \mathrm{hash}_y(u, v) \end{pmatrix}, \tag{81}$$

also unter Verwendung von zwei unterschiedlichen Hash-Funktionen $\mathrm{hash}_x()$ und $\mathrm{hash}_y()$. Die resultierende Gradientenfunktion ist im zweidimensionallen

[15] Auf den zweiten Blick möglicherweise eine triviale Tatsache.

Fall dann

$$\mathsf{grad}(u,v) = \begin{pmatrix}\mathsf{grad}_x(u,v,w)\\ \mathsf{grad}_y(u,v,w)\end{pmatrix} = 2\cdot\begin{pmatrix}\mathrm{hash}_x(u,v)\\ \mathrm{hash}_y(u,v)\end{pmatrix} - \begin{pmatrix}1\\1\end{pmatrix}. \tag{82}$$

beziehungsweise im dreidimensionalen Fall

$$\mathsf{grad}(u,v,w) = \begin{pmatrix}\mathsf{grad}_x(u,v,w)\\ \mathsf{grad}_y(u,v,w)\\ \mathsf{grad}_z(u,v,w)\end{pmatrix} = 2\cdot\begin{pmatrix}\mathrm{hash}_x(u,v,w)\\ \mathrm{hash}_y(u,v,w)\\ \mathrm{hash}_z(u,v,w)\end{pmatrix} - \begin{pmatrix}1\\1\\1\end{pmatrix}. \tag{83}$$

Damit liegen die Komponenten des resultierenden Gadientenvektors jeweils wiederum im Intervall $[-1,1]$. Im allgemeinen, N-dimensionalen Fall weist der zu erzeugende Grandientenvektor N Elemente auf, d. h.,

$$\mathsf{grad}(\boldsymbol{p}) = \begin{pmatrix}\mathsf{grad}_0(\boldsymbol{p})\\ \mathsf{grad}_1(\boldsymbol{p})\\ \vdots\\ \mathsf{grad}_{N-1}(\boldsymbol{p})\end{pmatrix} = 2\cdot\begin{pmatrix}\mathrm{hash}_0(\boldsymbol{p})\\ \mathrm{hash}_1(\boldsymbol{p})\\ \vdots\\ \mathrm{hash}_{N-1}(\boldsymbol{p})\end{pmatrix} - \begin{pmatrix}1\\1\\\vdots\\1\end{pmatrix} = 2\cdot\mathrm{hash}(\boldsymbol{p}) - \mathbf{1}. \tag{84}$$

Die Verknüpfung der Koordinaten (u,v,w) sowie die Implementierung der drei Hash-Funktionen ist vom jeweiligen Hash-Verfahren abhängig, wie nachfolgend gezeigt.

5.1 Hashing mit Permutationstabellen

In Perlin's Methode [8] wird die Hash-Funktion aus Effizienzgründen[16] durch mehrfache Verwendung einer fixen, zufälligen Permutationstabelle P der Größe 256 realisiert. Beispielsweise wird in [8] eine Permutationstabelle

$$P = (151, 160, 137, 91, \ldots, 61, 156, 180)$$

verwendet, mit $0 \le i, P[i] < 256$ und $P(i) \neq P[j]$ für $i \neq j$ (siehe die Auflistung auf S. 31). Im eindimensionalen Fall ist dann einfach

$$\mathrm{hash}(u) = \frac{1}{255}\cdot P[u \bmod 256]. \tag{85}$$

Dabei wiederholt sich das Ergebnis zwar zyklisch mit der (relativ kurzen) Periode 256, allerdings fällt dies in der kombinierten Noise-Funktion aufgrund der Verwendung mehrerer Frequenzen nicht unmittelbar auf. Größere Periodenlängen können aber durch Kombination mehreren kurzen Permutationstabellen erzielt werden, ohne dabei auf den Vorteil der schnellen Berechenbarkeit zu verzichten [6].

[16] Ursprünglich gedacht zur Implementierung auf eingeschränkter Grafik-Hardware.

Die eigentliche Vorteil dieser Methode wird erst im mehrdimensionalen Fall deutlich. Hier wird einfach dieselbe Permutationstabelle hintereinander auf die ganzzahligen Koordinatenwerte angewandt, also im Prinzip in der Form

$$P[P[u]+v] \qquad \text{oder} \qquad P[P[P[u]+v]+w] \tag{86}$$

in 2D bzw. in 3D. Das funktioniert natürlich nur, wenn dabei der beschränkte Indexbereich der Permutationstabelle P berücksichtigt wird, etwa in der Form

$$P[(P[u \bmod 256]+v) \bmod 256], \qquad \text{bzw.} \tag{87}$$

$$P[(P[(P[u \bmod 256]+v) \bmod 256]+w) \bmod 256]. \tag{88}$$

Dies lässt sich natürlich auf beliebig viele Dimensionen erweitern.

Zur schnellen Berechnung des Ausdrucks in Gl. 87 bzw. 88 verwendet Perlin allerdings einen weiteren Trick. So ergibt etwa die Änderung von Gl. 87 nach

$$P\Big[\underbrace{P[u \bmod 256]}_{\in[0,255]} + \underbrace{(v \bmod 256)}_{\in[0,255]}\Big] \qquad \in[0,510] \tag{89}$$

zwar nicht dasselbe, jedoch ein ähnlich „zufälliges“ Ergebnis. Wie leicht ersichtlich, liegt in diesem Fall jeder mögliche Tabellenindex für P sicher im Intervall $[0, 510]$. Perlin nutzt diesen Umstand, indem er die Permutationstabelle durch Verkettung einfach verdoppelt, d. h.,

$$\begin{aligned} P' = P||P &= (\mathbf{151}, 160, \ldots, 156, 180) \,||\, (\mathbf{151}, 160, \ldots, 156, 180) && (90)\\ &= (\mathbf{151}, 160, \ldots, 156, 180, \mathbf{151}, 160, \ldots, 156, 180). && (91) \end{aligned}$$

P' hat damit die Länge 512, wobei

$$P'[i] = \begin{cases} P[i] & \text{für } 0 \le i < 256, \\ P[i-256] & \text{für } 256 \le i < 512. \end{cases} \tag{92}$$

Mit der verlängerten Permutationstabelle P' und $u' = u \bmod 256$, $v' = v \bmod 256$, $w' = w \bmod 256$, kann etwa der Ausdruck in Gl. 88 für drei Dimensionen sehr einfach in der Form

$$\mathrm{h}_8(u, v, w) = P'[P'[P'[u'] + v'] + w'] \tag{93}$$

berechnet werden, und analog im N-dimensionalen Fall für den Gitterpunkt $\boldsymbol{p} = (u_0, u_1, \ldots, u_{N-1}) \in \mathbb{Z}^N$ durch

$$\mathrm{h}_8(\boldsymbol{p}) = P'[\ldots P'[P'[u'_0] + u'_1] + u'_2] + \ldots + u'_{N-1}]. \tag{94}$$

Die konkrete Berechnung für 3D ist in der folgenden Java-Mehode gezeigt:

```
1  int h8 (int u, int v, int w) {
2    u = u & 0xFF;
3    v = v & 0xFF;
4    w = w & 0xFF;
5    return P[P[P[u]+v]+w];
6  }
```

Die mod-Operation ist dabei durch die bitweise Maskierung mit `& 0xFF` realisiert.[17]

Die zugehörige Permutationstabelle `P` $\equiv P'$ wird dabei (ähnlich wie in Perlin's eigener Referenzimplementierung[18]) durch das statische Java-Kodesegment in Prog. 1 definiert.

Die oben gezeigte Methode `h8()` liefert natürlich nur einen einzigen `int`-Wert im Intervall $[0, 255]$, der für *eindimensionale* Anwendungen leicht in einen entsprechenden `double`-Wert im Intervall $[0, 1]$ umgewandelt werden kann (siehe Gl. 85).

Für *mehrdimensionale* Noise-Funktionen könnte man zur Definition der Funktionen $\text{hash}_x()$, $\text{hash}_y()$ etc. (Gl. 81) im einfachsten Fall aus dem 8-Bit Rückgabewert von `h8()` für jede Dimension eine Gruppe von Bits selektieren,[19] also für den zweidimensionalen Fall beispielsweise in der Form

$$\text{hash}(u,v) = \begin{pmatrix}\text{hash}_x(u,v)\\ \text{hash}_y(u,v)\end{pmatrix} = \frac{1}{63}\cdot\begin{pmatrix}\text{h8}(u,v) \bmod 64\\ (\text{h8}(u,v) \div 4) \bmod 64\end{pmatrix}. \tag{95}$$

Durch $\text{hash}_x()$ werden dabei die untersten 6 Bits und von $\text{hash}_y()$ die obersten 6 Bits des 8-Bit Ergebnisses von h8() verwendet. Eine zugehörige Java-Implementierung könnte folgendermaßen aussehen:

```
1 double[] hash(int u, int v) {
2     final int M = 0x3F;
3     int h = h8(u, v);
4     double hx = h & M;           // use bits 0..5 for hx()
5     double hy = (h >> 2) & M;    // use bits 2..7 for hy()
6     return new double[] {hx/M, hy/M};
7   }
```

Es wäre hier allerdings völlig ausreichend (und naheliegend), zwei getrennte Bitblöcke von jeweils nur 4 Bit Länge pro Dimension zu verwenden. Das ist im folgenden Beispiel für den dreidimensionalen Fall gezeigt:

```
1 double[] hash(int u, int v, int w) {
2     final int M = 0x0F;
3     int h = h8(u, v, w);
```

[17] Die Maskierung entspricht eigentlich eine „Rest"-Operation, die nur für positive Werte dasselbe numerische Ergebnis wie die mod-Operation liefert (siehe auch Abschnitt B.1.2 auf S. 437 in [2]). In diesem Fall ist das aber egal, da es ohnehin nur um Pseudo-Zufallszahlen geht.

[18] http://mrl.nyu.edu/~perlin/noise/

[19] Diese Idee ist eine Erweiterung zu der in [8] vorgeschlagenen Implementierung.

```
1  static final int P[] = new int[512];
2
3  // static initializer block:
4
5  static {int[] perm = {
6        151, 160, 137, 91, 90, 15, 131, 13,
7        201, 95, 96, 53, 194, 233, 7, 225,
8        140, 36, 103, 30, 69, 142, 8, 99,
9        37, 240, 21, 10, 23, 190, 6, 148,
10       247, 120, 234, 75, 0, 26, 197, 62,
11       94, 252, 219, 203, 117, 35, 11, 32,
12       57, 177, 33, 88, 237, 149, 56, 87,
13       174, 20, 125, 136, 171, 168, 68, 175,
14       74, 165, 71, 134, 139, 48, 27, 166,
15       77, 146, 158, 231, 83, 111, 229, 122,
16       60, 211, 133, 230, 220, 105, 92, 41,
17       55, 46, 245, 40, 244, 102, 143, 54,
18       65, 25, 63, 161, 1, 216, 80, 73,
19       209, 76, 132, 187, 208, 89, 18, 169,
20       200, 196, 135, 130, 116, 188, 159, 86,
21       164, 100, 109, 198, 173, 186, 3, 64,
22       52, 217, 226, 250, 124, 123, 5, 202,
23       38, 147, 118, 126, 255, 82, 85, 212,
24       207, 206, 59, 227, 47, 16, 58, 17,
25       182, 189, 28, 42, 223, 183, 170, 213,
26       119, 248, 152, 2, 44, 154, 163, 70,
27       221, 153, 101, 155, 167, 43, 172, 9,
28       129, 22, 39, 253, 19, 98, 108, 110,
29       79, 113, 224, 232, 178, 185, 112, 104,
30       218, 246, 97, 228, 251, 34, 242, 193,
31       238, 210, 144, 12, 191, 179, 162, 241,
32       81, 51, 145, 235, 249, 14, 239, 107,
33       49, 192, 214, 31, 181, 199, 106, 157,
34       184, 84, 204, 176, 115, 121, 50, 45,
35       127, 4, 150, 254, 138, 236, 205, 93,
36       222, 114, 67, 29, 24, 72, 243, 141,
37       128, 195, 78, 66, 215, 61, 156, 180 };
38
39    // create final permutation table:
40    for (int i = 0; i < 256; i++)
41      P[256 + i] = P[i] = perm[i];
42
43 }
```

Programm 1: Initialisierung der Permutationstabelle.

```
4     double hx =  h & M;            // use bits 0..3 for hx()
5     double hy = ((h >> 2) & M);    // use bits 2..5 for hy()
6     double hz = ((h >> 4) & M);    // use bits 4..7 for hz()
7     return new double[] {hx/M, hy/M, hz/M};
8   }
```

Diese Methode liefert für eine gegebene Hash-Koordinate (u, v, w) einen `double`-Vektor mit 3 Elementen im Intervall $[0, 1]$. Dafür werden aus dem ursprünglichen 8-Bit Hash-Wert drei unterschiedliche Gruppen von jeweils 4 Bits für die Gradienten in x, y bzw. z-Richtung verwendet, die `double`-Ergebnisse sind also in 16 mögliche Werte gestuft. Das ist zwar im Vergleich zu den nachfolgenden Methoden recht primitiv, erfüllt aber dennoch seinen Zweck. Die Anforderungen an die Zufälligkeit der Gradientenwerte sind bei der Perlin-Methode augenscheinlich recht gering.[20] Abbildung 15 zeigt einige Beispiele, die mit der Permutationsmethode und variierender Anzahl von Bits je Gradientenrichtung berechnet wurden.

5.2 Integer-Hashing

Ein Nachteil der in Abschnitt 5.1 beschriebenen Permutationsmethode ist die kurze Periodenlänge der erzeugten Zufallsfolgen. Die gängige Alternative ist die Verwendung einer Integer-Hashfunktion mit mindestens 32 Bits. Derartige Hash-Funktionen werden u. a. zur Verschlüsselung in der Kryptographie eingesetzt, wobei die dort gestellten Anforderungen an die statistische Qualität der erzeugten Zufallsfolgen für unsere Anwendung nicht relevant sind. Interessante Details zu diesem Thema und akuelle Verfahren finden sich beispielsweise unter „Random Hashes and Random Bytes“ in [10, Abschn. 7.1.4]. Hochwertige Hashverfahren arbeiten heute in der Regel mit 64 Bits und kombinieren mehrere unterschiedliche Hashing-Verfahren.

Zur Generierung von Perlin Noise erscheinen aber einfachere Methoden auf der Basis von 32 Bit-Arithmetik vollkommen ausreichend. Algorithmus 6 zeigt drei unterschiedliche Realisierungen von Hashfunktionen des Typs

$$\text{hashInt}(k) : \mathbb{Z}_{32} \to \mathbb{Z}_{32}^{+},$$

für beliebige 32-Bit Schlüssel k. Die Funktion HashIntWard() wird in [12, S. 615] mit `frand(s)` bezeichnet. Die beiden übrigen Funktionen HashIntShift() und HashIntShiftMult() sind in [11] beschrieben. Die darin verwendeten Bitoperationen sind in Tabelle 1 zusammengestellt. Über die statistischen Eigenschaften dieser drei Hashfunktionen, insbesondere über das Verhalten in den bekannten *Diehard*-Tests[21] sind dem Autor keine Informationen bekannt.

[20] Tatsächlich wird bei Perlin [8] das 8-Bit Ergebnis der Hashfunktion modulo 12 verwendet und damit eine der 12 möglichen Kanten eines dreidimensionalen Würfels ausgewählt. Die Richtungen der resultierenden 3D-Gradienten werden demnach extrem grob gestuft, was sich allerdings auf das visuelle Ergebnis kaum auswirkt.

[21] http://i.cs.hku.hk/~diehard/

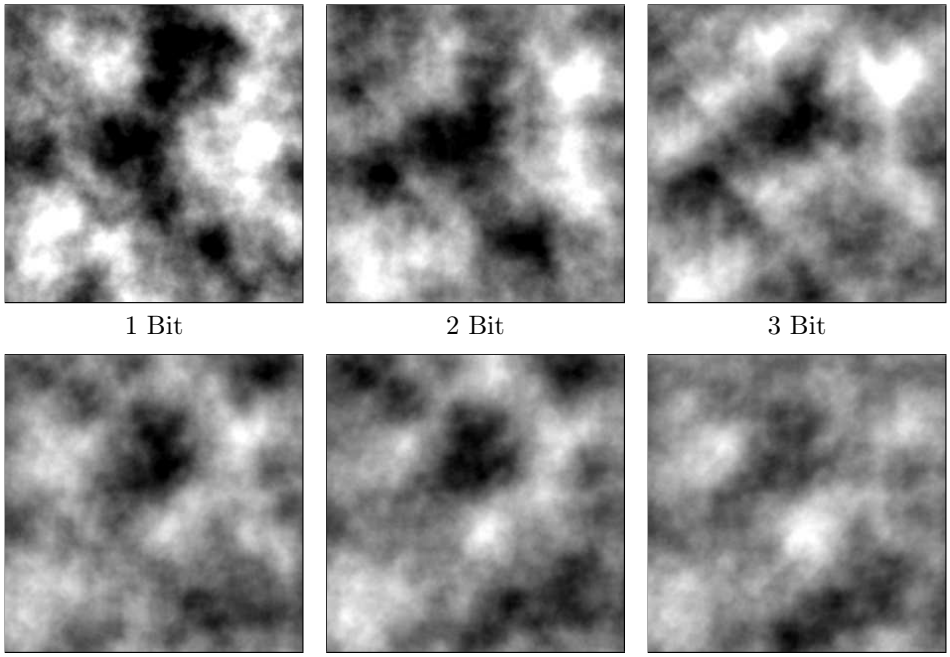


Abbildung 15: Beispiele für Permutations-Hashing mit variierender Anzahl von Bits je Gradientenrichtung. Bei 1 Bit gibt es nur 2 mögliche Gradientenwerte mit -1 oder $+1$; bei 6 Bits treten 64 linear verteilte Werte im Intervall $[-1, 1]$ auf.

Für die Realisierung der Funktion $\text{hash}(i)$ in Gl. 79 zur Abbildung auf das Intervall $[0, 1]$ ist lediglich das Ergebnis der obigen Integer-Hashfunktion entsprechend zu skalieren, d. h.,

$$\text{hash}(i) = \frac{1}{\texttt{maxInt}} \cdot \text{hashInt}(i), \tag{96}$$

mit $\texttt{maxInt} = 2^{31} - 1 = 2147483647$ und $\text{hash}(i) \in [0, 1]$.

5.2.1 Anwendung im mehrdimensionalen Fall

Zur Erzeugung des zugehörigen Hashwerts für einen n-mehrdimensionalen Gitterpunkt müssen alle n Koordinatenwerte dieses Punkts in der Hashfunktion berücksichtigt werden. Beim Permutation-Hashing wird dies (wie in Gl. 86, 93 gezeigt) durch aufeinanderfolgende Anwendung der Permutation auf die Koordinaten bewirkt. Das könnte man in ähnlicher Form natürlich auch durch aufeinanderfolgende Anwendung einer Integer-Hashfunktion erreichen.

Einfacher ist es, die Rasterkoordinaten zunächst in geschickter Weise zu *einem* Integer-Wert zu kombinieren („pre-hashing“) und dann die eigentliche Integer-Hashfunktion nur auf diesen Wert anzuwenden. Für den dreidi-

Algorithmus 6: Implementierungsbeispiele für die 32-Bit Integer-Hashfunktion hashInt(k). Angenommen wird die übliche Darstellung der Zahlenwerte im Zweierkomplement. Der Parameter k ist ein beliebiger Integerwert im Intervall $[-2^{-31}, 2^{31}-1]$; der Rückgabewert ist eine positive Integer-Zahl im Intervall $[0, 2^{31}-1]$. Siehe Tab. 1 zur Beschreibung der hier verwendeten Bit-Operatoren. `maxInt` bezeichnet die maximale (positive) 32 Bit-Zahl ($2^{31}-1 = 2147483647$).

```
1:  HashIntWard(k)                                          ▷ aus [12]
2:      k ← (k ≪ 13) xor k
3:      k ← k · (k² · 15731 + 789221) + 1376312589
4:      return (k and maxInt)
5:  end
```

```
6:  HashIntShift(k)                                         ▷ aus [11]
7:      M ← (2^31 − 1)
8:      k ← k̄ + (k ≪ 15)
9:      k ← k xor (k ≫ 12)
10:     k ← k + (k ≪ 2)
11:     k ← k xor (k ≫ 4)
12:     k ← k · 2057
13:     k ← k xor (k ≫ 16)
14:     return (k and maxInt).
15: end
```

```
16: HashIntShiftMult(k)                                     ▷ aus [11]
17:     C ← 668265261                       ▷ a prime or at least odd constant
18:     M ← (2^31 − 1)
19:     k ← (k xor 61) xor (k ≫ 16)
20:     k ← k + (k ≪ 3)
21:     k ← k xor (k ≫ 4)
22:     k ← k · C
23:     k ← k xor (k ≫ 15)
24:     return (k and maxInt).
25: end
```

mensionalen Fall mit Gitterkoordinaten (u, v, w) kann der Integer-Hashwert beispielsweise durch folgende Methode berechnet werden:

$$h = \text{hashInt}(p_u \cdot u + p_v \cdot v + p_w \cdot w), \tag{97}$$

wobei p_u, p_v, p_w unterschiedliche (typischerweise kleine) Primzahlen sind.

Damit können wir auf einfache Weise auch jene n Hashfunktionen definieren, die uns für jeden Gitterpunkt im n-dimensionalen Raum die n (unabhängigen) Elemente des zufälligen Gradientenvektors liefern. Diese hatten wir

Tabelle 1: Bit-Operationen. Mathematische Notation (linke Spalte) und die entsprechenden Java-Operatoren (rechte Spalte). a, b bezeichnen Bitmuster, typischerweise mit 32 Bit, wobei das *least significant bit* ganz rechts steht.

Notation	*Beschreibung*	*in Java*
$\bar{a}$	Bitweise Inversion von a	`~a`
a and b	Bitweise UND-Operation zwischen a und b	`a & b`
a xor b	Bitweise Exclusive-OR-Operation zwischen a und b	`a ^ b`
$a \ll n$	Bitweise Verschiebung von a um n Bitpositionen nach links, rechts wird mit Nullen aufgefüllt	`a << n`
$a \gg n$	Bitweise „unsigned“ Verschiebung von a um n Bitpositionen nach rechts, links wird mit Nullen aufgefüllt	`a >>> n`

etwa für den dreidimensionalen Fall in Gl. 83 mit $\text{hash}_x()$, $\text{hash}_y()$, $\text{hash}_z()$ bezeichnet.

Genauso wie beim Permutationsverfahren können wir die Komponenten der mehrdimensionalen Hashfunktion wieder durch Auswahl bestimmter Bitgruppen aus dem Ergebnis von `hashInt()` erzeugen. Da nun 31 Bits[22] (statt 8 Bits bei der Permutationsmethode) zur Verfügung stehen, ist das keinerlei Problem.[23] Eine Java-Implementierung dieser Variante könnte für den dreidimensionalen Fall etwa folgendermaßen aussehen:

```
1 double[] hash(int u, int v, int w) {
2   final int M = 0x000000FF;
3   int h = hashInt(59*u + 67*v + 71*w);
4   double hx =  h & M;            // extract bits 0..7
5   double hy = (h >> 8) & M;    // extract bits 8..15
6   double hz = (h >> 16) & M;   // extract bits 16..23
7   return new double[] {hx/M, hy/M, hz/M};
8 }
```

Dabei ergeben sich pro Hashwert immer noch 256 Abstufungen, was für die Erzeugung von Gradientenrauschen mehr als ausreichend ist.[24] Der Vorteil ist natürlich, dass die Integer-Hashfunktion nur einmal pro Gitterpunkt aufgerufen wird.

Als Alternative könnten die drei Komponenten durch drei unterschiedliche Integer-Hashfunktionen definiert werden [12], also beispielsweise in der

[22] Das Ergebnis von `hashInt()` ist immer positiv und damit das Vorzeichenbit (32. Bit) immer null.

[23] Bei guten Integer-Hashfunktionen sollten auch beliebig aus dem Ergebnis gewählte Bitblöcke völlig zufällig verteilt sein.

[24] Man kann für 3D natürlich bis zu 10 Bits und für 2D bis zu 15 Bits pro Dimension verwenden. Falls das wider Erwarten nicht ausreicht, gibt es ja auch noch die modernen 64-Bit Hashfunktionen.

Form

$$\text{hash}_x(u,v,w) = \frac{1}{\text{maxInt}} \cdot \text{hashInt}(59u + 67v + 71w), \tag{98}$$

$$\text{hash}_y(u,v,w) = \frac{1}{\text{maxInt}} \cdot \text{hashInt}(73u + 79v + 83w), \tag{99}$$

$$\text{hash}_z(u,v,w) = \frac{1}{\text{maxInt}} \cdot \text{hashInt}(89u + 97v + 101w), \tag{100}$$

unter Verwendung der beliebig gewählten kleinen Primzahlen $59, 67, \ldots, 101$. Anstelle von hashInt() kann hier natürlich jede der konkreten Funktionen in Alg. 6 oder ähnlich geeignete Integer-Hashfunktion verwendet werden. In Java implementiert könnte das dann etwa so aussehen:

```
1 double[] hash(int u, int v, int w) {
2   final int maxInt = 0x7FFFFFFF;  // = 2147483647
3   double hx = hashInt(59 * u + 67 * v +  71 * w);
4   double hy = hashInt(73 * u + 79 * v +  83 * w);
5   double hz = hashInt(89 * u + 97 * v + 101 * w);
6   return new double[] {hx/maxInt, hy/maxInt, hz/maxInt};
7 }
```

Für den N-dimensionalen Fall bietet sich natürlich letztere Variante an, wobei man die erforderlichen Primzahlen leicht aus einer entsprechenden Tabelle entnehmen kann.

5.3 Noch mehr Zufälligkeit durch „*seed*“

Alle in Abschn. 5 beschriebenen Hashfunktionen sind naturgemäß deterministisch, d. h., sie liefern für gegebene Gitterkoordinaten auch bei wiederholter Anwendung immer dieselben Werte. Mit einer bestimmten Hashfunktion generierte Rauschfunktionen sehen also immer völlig identisch aus. Um dem Abhilfe zu schaffen, könnte man beispielsweise für ein 2D-Rauschbild oder ein 3D-Volumen jeweils andere Startkoordinaten wählen. Im Folgenden wird die oben beschriebene Hashmethode – ähnlich wie bei gängigen Zufallsgeneratoren – durch einen ganzzahligen „Seed“-Wert S parametrisiert, mit dem unterschiedliche Rauschergebnisse erzeugt werden können.

Im naheliegensten Fall wird der Parameter S einfach als konstanter Offset in der Form

$$\text{hashInt}_S(k) = \text{hashInt}(k + S), \qquad \text{mit } S \in \mathbb{Z}, \tag{101}$$

verwendet. Abbildung 16 zeigt Beispiele für 2D-Rauschfunktionen, die mit drei unterschiedlichen Integer-Hashfunktionen (HashIntShift, HashIntShiftMult und HashIntWard aus Alg. 6) und Seed-Werten $S = 0, 1, 2$ berechnet wurden.

In ähnlicher Weise könnte man bei der Permutationsmethode (siehe Abschn. 5.1) den Seed-Wert S durch einen zusätzlichen Permutationsschritt

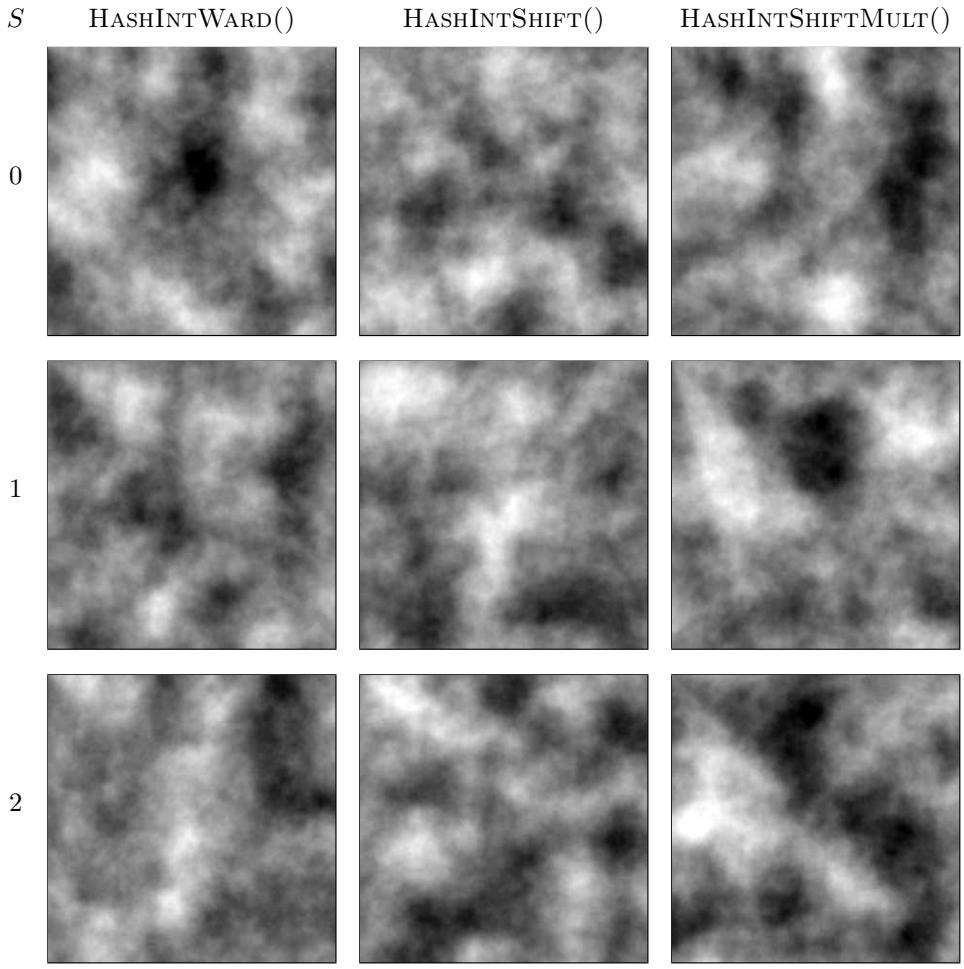


Abbildung 16: Beispiele für 2D Noisefunktionen, die jeweils mit der selben Hash-Funktion aber unterschiedlichen Seed-Werten $S = 0, 1, 2$ erzeugt wurden. Die Noise-Parameter sind für alle Beispiele identisch mit $p = 0.5$, $f_{\min} = 0.01$, $f_{\max} = 0.5$ (6 Oktaven mit Frequenzen $f_i = 0.01$, 0.02, 0.04, 0.08, 0.16, 0.32 und Amplituden $a_i = 1.0$, 0.5, 0.25, 0.125, 0.0625, 0.03125).

berücksichtigen oder – als Alternative – die Perputationstabelle P auf Basis von S jeweils neu berechnen. Natürlich kann der Wert von S auch mit einem üblichen Zufallsgenerator bestimmt werden, um völlige „Zufälligkeit“ zu erreichen.

6 Zusammenfassung

Dieser Bericht ist als Einführung und Tutorial gedacht, vorwiegend um das Verständnis und die Implementierung von Gradientenrauschen nach der Methode von Ken Perlin zu erleichtern. Die allermeisten der hier gezeigten Zusammenhänge sind entweder in den zitierten Quellen zu finden oder können relativ einfach selbst hergeleitet werden, sofern dazu ausreichend Zeit und Bereitschaft vorliegt.

Hier nicht berücksichtigt ist der Umstand, das Perlin [9] zur Verbesserung der Orientierungsunabhängigkeit beim 3D-Verfahren die Gradienten nicht an den 8 Eckpunkten des Würfels, sondern an den Mittelpunkten der 12 Kanten definiert. Die Interpolation der Tangentenwerte erfolgt in der selben Form wie hier beschrieben. Ausführlichere Hinweise und eine Implementierung dazu findet man in [5]. Ebenfalls nicht berücksichtigt ist „Simplex Noise“, die neuere und – in höherdimensionalen Räumen wesentlich effizientere – Variante von Perlin's Rauschverfahren. Auch dazu ist allerdings kaum publiziertes Originalmaterial auffindbar, man findet aber wiederum in [5] weiterführende Informationen und einen konkreten Implementierungsvorschlag. Weitere interessante Online-Quellen zu diesem Thema sind die Seiten von Hugo Elias,[25] Mandelbrot Dazibao[26] und Matt Zucker.[27]

Eine prototypische Java-Implementierung der hier beschriebenen Algorithmen (für ImageJ[28]) findet sich auf der Homepage des Autors (s. Titelseite). Bei dieser Implementierung wurde allerdings vorrangig Wert auf gute Lesbarkeit und Klarheit gelegt, um die Algorithmen durch Abbildung in eine konkreten Programmiersprache zu verdeutlichen. Entsprechend zahlreich sind daher auch die Möglichkeiten zur Erhöhung der Effizienz. Darüber hinaus sind natürlich Fragen oder Anregungen zu diesem Text immer willkommen.

Literatur

[1] Bourke, Paul: *Perlin noise and turbulence.* Technischer Bericht, University of Western Australia, January 2000. http://local.wasp.uwa.edu.au/~pbourke/texture_colour/perlin/.

[2] Burger, Wilhelm und Mark J. Burge: *Digitale Bildverarbeitung.* Springer, 2. Auflage, 2006. www.imagingbook.com.

[3] Burger, Wilhelm und Mark J. Burge: *Digital Image Processing—An Algorithmic Introduction using Java.* Springer, New York, 2008. www.imagingbook.com.

[25] http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
[26] http://www.mandelbrot-dazibao.com
[27] http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html
[28] http://rsb.info.nih.gov/ij/

[4] Ebert, David S., F. Kenton Musgrave, Darwin Peachey, Ken Perlin und Steven Worley: *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, San Francisco, 3. Auflage, 2002.

[5] Gustavson, Stefan: *Simplex noise demystified*. Technischer Bericht, Linköping University, 2005. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[6] Lagae, Ares und Philip Dutré: *Long period hash functions for procedural texturing*. In: Kobbelt, L., T. Kuhlen, T. Aach und R. Westermann (Herausgeber): *Vision, Modeling, and Visualization 2006*, Seiten 225–228, Aachen, Germany, 2006. IOS Press.

[7] Perlin, Ken: *An image synthesizer*. In: *SIGGRAPH'85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, Seiten 287–296, New York, NY, USA, 1985. ACM, ISBN 0-89791-166-0.

[8] Perlin, Ken: *An image synthesizer*. SIGGRAPH Computer Graphics, 19(3):287–296, 1985, ISSN 0097-8930.

[9] Perlin, Ken: *Improving noise*. In: *SIGGRAPH'02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, Seiten 681–682, New York, NY, USA, 2002. ACM, ISBN 1-58113-521-1.

[10] Press, William H., Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery: *Numerical Recipes The Art of Scientific Computing*. Cambridge University Press, 3. Auflage, 2007.

[11] Wang, Thomas: *Integer Hash Function*, March 2007. http://www.concentric.net/~Ttwang/tech/inthash.htm, Version 3.1.

[12] Ward, Greg: *A recursive implementation of the perlin noise function*. In: Arvo, James (Herausgeber): *Graphics Gems II*, Kapitel VIII.10, Seiten 396–401. Academic Press, 1991.

Änderungen

2009-11-24 Geringfügige Korrekturen bei der Formulierung der Noise-Funktion mit mehreren Frequenzen (Laufweiten der Indexvariablen) in Abschnitt 2.3.