Synthetic Gradient Noise

Research · May 2015

DOI: 10.13140/RG.2.1.3427.7284

CITATIONS
0

READS
2,999

2 authors:

Wilhelm Burger
Fachhochschule Oberösterreich
184 PUBLICATIONS **1,498** CITATIONS
SEE PROFILE

Mark Burge
idemia
185 PUBLICATIONS **1,943** CITATIONS
SEE PROFILE

Wilhelm Burger · Mark J. Burge

Principles of Digital Image Processing

Advanced Methods

Supplementary Material

With 16 figures and 4 algorithms

Springer-Verlag

Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Preface to the Print Edition

This is the 3rd volume of the authors' textbook series on ***Principles of Digital Image Processing*** that is predominantly aimed at undergraduate study and teaching:

Volume 1: ***Fundamental Techniques***,
Volume 2: ***Core Algorithms***,
Volume 3: ***Advanced Methods*** (this volume).

While it builds on the previous two volumes and relies on the their proven format, it contains all new material published by the authors for the first time. The topics covered in this volume are slightly more advanced and should thus be well suited for a follow-up undergraduate or Master-level course and as a solid reference for experienced practitioners in the field.

The topics of this volume range over a variety of image processing applications, with a general focus on "classic" techniques that are in wide use but are at the same time challenging to explore with the existing scientific literature. In choosing these topics, we have also considered input received from students, lecturers and practitioners over several years, for which we are very grateful. While it is almost unfeasible to cover all recent developments in the field, we focused on popular "workhorse" techniques that are available in many image processing systems but are often used without a thorough understanding of their inner workings. This particularly applies to the contents of the first five chapters on automatic thresholding, filters and edge detectors for color images, and edge-preserving smoothing. Also, an extensive part of the book is devoted to David Lowe's popular *SIFT* method for invariant local feature detection, which has found its way into so many applications and has become a standard tool in the industry, despite (as the text probably shows) its inherent sophistication and complexity. An additional "bonus chapter" on Synthetic

Gradient Noise, which could not be included in the print version, is available for download from the book's website.

As in the previous volumes, our main goal has been to provide *accurate*, *understandable*, and *complete* algorithmic descriptions that take the reader all the way from the initial idea through the formal description to a working implementation. This may make the text appear bloated or too mathematical in some places, but we expect that interested readers will appreciate the high level of detail and the decision not to omit the (sometimes essential) intermediate steps. Wherever reasonable, general prerequisites and more specific details are summarized in the Appendix, which should also serve as a quick reference that is supported by a carefully compiled Index. While space constraints did not permit the full source code to be included in print, complete (Java) implementations for each chapter are freely available on the book's website (see below). Again we have tried to make this code maximally congruent with the notation used in the text, such that readers should be able to easily follow, execute, and extend the described steps.

Software

The implementations in this book series are all based on Java and ImageJ, a widely used programmer-extensible imaging system developed, maintained, and distributed by Wayne Rasband of the National Institutes of Health (NIH).[1] ImageJ is implemented completely in Java and therefore runs on all major platforms. It is widely used because its "plugin"-based architecture enables it to be easily extended. Although all examples run in ImageJ, they have been specifically designed to be easily ported to other environments and programming languages. We chose Java as an implementation language because it is elegant, portable, familiar to many computing students, and more efficient than commonly thought. Note, however, that we incorporate Java purely as an instructional vehicle because precise language semantics are needed eventually to achieve ultimate clarity. Since we stress the simplicity and readability of our programs, this should not be considered production-level but "instructional" software that naturally leaves vast room for improvement and performance optimization. Consequently, this book is not primarily on Java programming nor is it intended to serve as a reference manual for ImageJ.

Online resources

In support of this book series, the authors maintain a dedicated website that provides supplementary materials, including the complete Java source code,

[1] http://rsb.info.nih.gov/ij/

the test images used in the examples, and corrections. Readers are invited to visit this site at

www.imagingbook.com

It also makes available additional materials for educators, including a complete set of figures, tables, and mathematical elements shown in the text, in a format suitable for easy inclusion in presentations and course notes. Also, as a free add-on to this volume, readers may download a supplementary "bonus chapter" on synthetic noise generation. Any comments, questions, and corrections are welcome and should be addressed to

imagingbook@gmail.com

Acknowledgments

As with its predecessors, this volume would not have been possible without the understanding and steady support of our families. Thanks go to Wayne Rasband (NIH) for continuously improving ImageJ and for his outstanding service to the imaging community. We appreciate the contributions from the many careful readers who have contacted us to suggest new topics, recommend alternative solutions, or to suggest corrections. A special debt of graditude is owed to Stefan Stavrev for his detailed, technical editing of this volume. Finally, we are grateful to Wayne Wheeler for initiating this book series and Simon Rees and his colleagues at Springer's UK and New York offices for their professional support, for the high quality (full-color) print production and the enduring patience with the authors.

Hagenberg, Austria / Washington DC, USA
January 2013

This document is made available as supplementary online material (Ch. 8) to

W. Burger and M. J. Burge. "Principles of Digital Image Processing – Advanced Methods (Vol. 3)". Undergraduate Topics in Computer Science. Springer-Verlag, London (2013).

BibTeX citation entry:

```
@book{BurgerBurgeUtics3,
    title = {Principles of Digital Image Processing --
             Advanced Methods (Vol. 3)},
    author = {Burger, Wilhelm and Burge, Mark J.},
    series = {Undergraduate Topics in Computer Science},
    publisher = {Springer-Verlag},
    address = {London},
    year = {2013},
    note = {supplementary Ch. 8},
    url = {www.imagingbook.com}
}
```

Contents

8
Synthetic Gradient Noise

The topic of this chapter may appear "exotic" in the sense that is does not deal with processing images or extracting useful information from images, but with *creating* new imagery. Since the techniques described here were originally developed in computer graphics and for texture synthesis in particular, they are typically not taught in image processing courses, although they seem to fit well into this context. Moreover, this is one of several interesting topics, where the computer graphics and image processing communities share common interests and methods.

Noise functions based on pseudo-random gradient values have various applications, particularly in image synthesis. These methods became widely known and popular due to Ken Perlin, who received an "Oscar" (Academy Award for Technical Achievement) in 1997 for his technique to create realistic synthetic scenes for the film industry. Noise synthesis is therefore often associated with his name, although gradient noise is only one of several methods for generating "interesting" noise functions. A good overview of this general topic can be found in [5]. Although details of Perlin's noise generation method have been published in many places [1, 5, 6, 9, 10] and a (extremely compact) reference implementation has been made available by the author,[1] it is certainly not a trivial task to understand and implement these methods from the available information. The purpose of this chapter is to expose the underlying concepts of Perlin noise in a concise and accessible way and to show the necessary steps towards the implementation of 1-, 2- and N-dimensional noise functions. The

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

algorithms described here should be easy to implement in any suitable programming language and a prototypical implementation in Java is available on the book's supporting web site.

8.1 One-dimensional noise functions

Our first task is to create a continuous, one-dimensional random function

$$\text{noise}(x) : \mathbb{R} \to \mathbb{R}, \tag{8.1}$$

based on a given sequence of discrete (but also randomly selected) values

$$g_u \in \mathbb{R}, \tag{8.2}$$

for discrete raster (lattice) points $u \in \mathbb{Z}$. The values g_u in Eqn. (8.2) thus specify the continuous noise function $\text{noise}(x)$ at regularly-spaced integer positions $x = u \in \mathbb{Z}$. These could be the actual function *values* $\text{noise}(u)$ but also, for example, the *slope* or first derivative of the function at discrete positions, as illustrated in Fig. 8.1. In the first case we talk about "value noise" and "gradient noise" in the second case [5, p. 70]. In both cases, the values of $\text{noise}(x)$ at a continuous position $x \in \mathbb{R}$ are calculated by local interpolation in the form

$$\text{noise}(x) = F(x, g_{\lfloor x \rfloor}, g_{\lfloor x \rfloor + 1}) = F(x, g_u, g_{u+1}), \tag{8.3}$$

between the discrete sample values g_u, g_{u+1} next to the position x, where F is a suitable interpolating function, as described below.

8.1.1 Gradient noise

Perlin noise is a kind of gradient noise, that is, the given random values g_u define the slope (*gradient*) of the function $\text{noise}(x)$ at discrete positions $x = u$. These random gradients are specified by a mapping

$$\mathsf{grad} : \mathbb{Z} \to [-1, 1] \tag{8.4}$$

and the values $g_u = \mathsf{grad}(u)$ are assumed to be uniformly distributed in $[-1, 1]$. The mapping grad is typically implemented by a hash function, as described in Sec. 8.1.4.

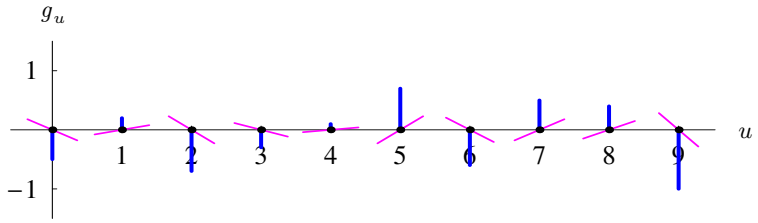


Figure 8.1 Discrete sequence of random gradient values. The values of the discrete random sequence $g_u = \mathsf{grad}(u)$ (blue bars) specify the slope of the noise function noise(x) at integer positions $u \in \mathbb{Z}$ (tangents shown in magenta). These are also defined to be zero positions of the continuous noise function.

Piecewise, local interpolation

The properties of the noise function depend crucially on the chosen interpolating function F, which must not only satisfy a few formal properties but should also give results that are visually appealing. The following sections deal with the definition of the interpolating function F.

The discrete positions u do not only hold the random gradient values g_u but are also defined to be zero positions of the noise function, that is,

$$\text{noise}(u) = 0 \qquad \text{and} \qquad \text{noise}'(u) = g_u, \tag{8.5}$$

for all $u \in \mathbb{Z}$. Thus, for any discrete position u, the interpolating function F (Eqn. (8.3)) must satisfy the following requirements:

$$\begin{aligned} F(u, g_u, g_{u+1}) &= 0, & F'(u, g_u, g_{u+1}) &= g_u, \\ F(u{+}1, g_u, g_{u+1}) &= 0, & F'(u{+}1, g_u, g_{u+1}) &= g_{u+1}. \end{aligned} \tag{8.6}$$

Here F' denotes the first derivative of the function F with respect to the position parameter x, that is,

$$F' = \frac{\partial F}{\partial x}. \tag{8.7}$$

Since the value of the interpolating function only depends on two adjacent gradient values g_u, g_{u+1}, we can limit the interpolation to the $[0, 1]$ unit interval and reformulate the interpolating function as

$$F(\dot{x}, g_0, g_1) \equiv F(x{-}u, g_u, g_{u+1}), \qquad \text{for } \dot{x} \in [0, 1], \tag{8.8}$$

with $u = \lfloor x \rfloor$ and $\dot{x} = x{-}u$.[2] The gradient samples positioned to the left and right of x are denoted $g_0 = g_u$, and $g_1 = g_{u+1}$, respectively.

[2] The symbol $\dot{x}$ is used in the following for a continuous position in the $[0, 1]$ interval. Analogously, $\dot{\boldsymbol{x}}$ is denotes a vector with components in $[0, 1]$.

Interpolating with a polynomial function

If we use a polynomial (which is quite an obvious choice) to specify the interpolating function F, it needs to be at least of third order, that is,

$$F_3(\dot{x}, g_0, g_1) = a_3 \cdot \dot{x}^3 + a_2 \cdot \dot{x}^2 + a_1 \cdot \dot{x} + a_0, \tag{8.9}$$

with the (yet to be determined) coefficients $a_0, a_1, a_2, a_3 \in \mathbb{R}$. The first derivative of this function is

$$F_3'(\dot{x}, g_0, g_1) = 3a_3 \cdot \dot{x}^2 + 2a_2 \cdot \dot{x} + a_1 \tag{8.10}$$

and therefore, given the constraints in Eqn. (8.6), the solution for the coefficients is

$$a_3 = g_0 + g_1, \qquad a_2 = -2g_0 - g_1, \qquad a_1 = g_0, \qquad a_0 = 0. \tag{8.11}$$

The interpolating function in Eqn. (8.9) can thus be written as

$$F_3(\dot{x}, g_0, g_1) = (g_0{+}g_1) \cdot \dot{x}^3 - (2g_0{+}g_1) \cdot \dot{x}^2 + g_0 \cdot \dot{x} \tag{8.12}$$

or, after a little rearrangement, as

$$F_3(\dot{x}, g_0, g_1) = g_0 \cdot (\dot{x}{-}1)^2 \cdot \dot{x} + g_1 \cdot (\dot{x}{-}1) \cdot \dot{x}^2. \tag{8.13}$$

Figure 8.2 shows an example of piecewise interpolation with the third-order (cubic) polynomial defined in Eqns. (8.12–8.13). The resulting curve is C1-continuous because the first derivative (slope) of the function has no discontinuities at the transitions between successive segments. The result is *not* C2-continuous, however, since its second derivative (curvature) is generally not continuous at integer positions.

Original Perlin interpolation

In his original publication [9], Perlin did not use a minimal, cubic polynomial as an interpolating function but a *fourth-order* polynomial of the form

$$F(\dot{x}, g_0, g_1) = a_4 \cdot \dot{x}^4 + a_3 \cdot \dot{x}^3 + a_2 \cdot \dot{x}^2 + a_1 \cdot \dot{x} + a_0, \tag{8.14}$$

which, under the constraints in Eqn. (8.6), has the coefficients

$$a_4 = a_2 + 2g_0 + g_1, \quad a_3 = -2a_2 - 3g_0 - g_1, \quad a_1 = g_0, \quad a_0 = 0, \tag{8.15}$$

where a_2 can be chosen freely. Setting $a_2 = -3g_1$, the result is

$$F_4(\dot{x}, g_0, g_1) = 2(g_0{-}g_1) \cdot \dot{x}^4 - (3g_0{-}5g_1) \cdot \dot{x}^3 - 3g_1 \cdot \dot{x}^2 + g_0 \cdot \dot{x} \tag{8.16}$$

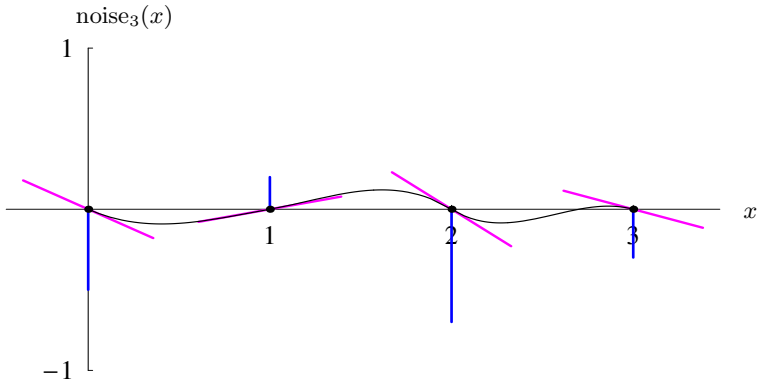


Figure 8.2 Interpolation with the minimal (cubic) polynomial, as defined in Eqn. (8.12). The blue bars represent the predefined, randomly chosen gradient values g_u at integer positions $u \in \mathbb{Z}$, which are also zero positions of the continuous noise function. The corresponding tangents (drawn in magenta) show the slope of the interpolated function at positions u.

or equivalently (in Perlin's notation),

$$F_4(\dot{x}, g_0, g_1) = g_0 \cdot \dot{x} + \big(3\dot{x}^2 - 2\dot{x}^3\big) \cdot \big(g_1 \cdot (\dot{x}-1) - g_0 \cdot \dot{x}\big). \tag{8.17}$$

Figure 8.3 shows an example for interpolating with this fourth-order polynomial, using the same discrete gradient samples as in Fig. 8.2. The second derivative (curvature) of F_4 at the integer positions $u = 0, 1$ is

$$F_4''(0, g_0, g_1) = -6g_1 \qquad \text{and} \qquad F_4''(1, g_0, g_1) = 6g_0, \tag{8.18}$$

and is thus non-zero in general.[3] This should be kept in mind, because it means that the function's curvature may change spontaneously at segment boundaries and thereby cause unwanted visible artifacts (also see Sec. 8.1.1).

Interpolating tangents

In the one-dimensional case, each given gradient sample $g_u = \mathsf{grad}(u)$ specifies the corresponding tangent at position u, i. e., a straight line with slope g_u that intersects the x-axis at position u. This line can be described by the linear function

$$h_u(x) = g_u \cdot (x - u), \tag{8.19}$$

for arbitrary values $x \in \mathbb{R}$ and $i \in \mathbb{Z}$. Now the function $F_4(\dot{x}, g_0, g_1)$ in Eqn. (8.16) can be viewed as a spline interpolation between the two tangents at the

[3] This reveals the interesting fact that the function's curvature at any end point of the $[0, 1]$ interval only depends on the gradient value at the opposite end of the segment.

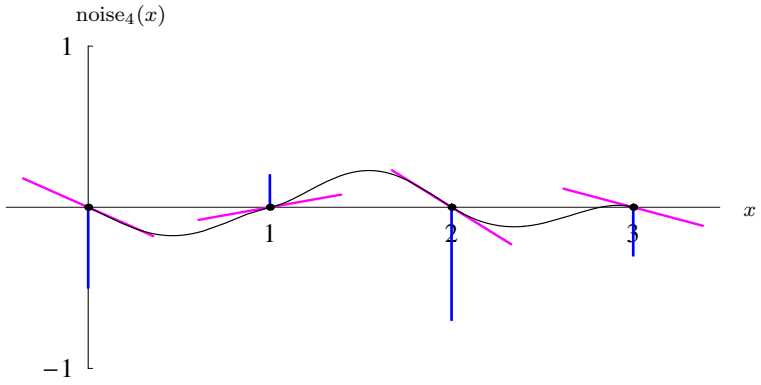


Figure 8.3 Applying Perlin's original (fourth-order polynomial) interpolating function, as defined in Eqn. (8.16), for the same gradient samples as used in Fig. 8.2.

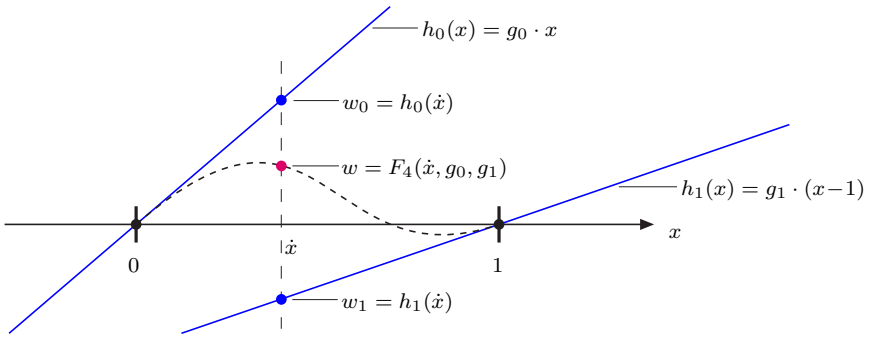


Figure 8.4 Interpolating tangent lines in the unit interval using a sigmoid blending function. The tangent lines h_0 and h_1 pass through the points $x = 0$ and $x = 1$, respectively. Their slope is specified by the discrete gradient samples g_0 and g_1. For a specific point $\dot{x} \in [0, 1]$, the interpolated function value w is obtained as a weighted average of the tangent values $w_0 = h_0(\dot{x})$ and $w_1 = h_1(\dot{x})$, with the weights specified by the S-shaped blending function $s(\dot{x})$ (see Eqn. (8.23)).

end points of the affected unit segment (with slopes g_0 and g_1, respectively).[4] as illustrated in Fig. 8.4. Using Eqn. (8.19), the line equations for the tangents at sample points $u = 0, 1$ are

$$h_0(x) = g_0 \cdot x \qquad \text{and} \qquad h_1(x) = g_1 \cdot (x - 1), \tag{8.20}$$

[4] See http://shiny3d.de/mandelbrot-dazibao/Perlin/Perlin1.htm.

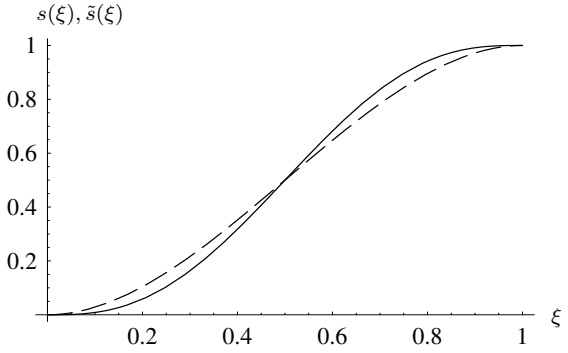


Figure 8.5 S-shaped blending function for the local interpolation of tangent slopes over the $[0, 1]$ interval. Original version $s(\xi) = 3\xi^2 - 2\xi^3$ (continuous curve, see Eqn. (8.22)); modified blending function $\tilde{s}(\xi) = 10\xi^3 - 15\xi^4 + 6\xi^5$ (dashed curve, see Eqn. (8.24)).

respectively. Thus the interpolating function in Eqn. (8.17) can be reformulated as

$$F_4(\dot{x}, g_0, g_1) = h_0(\dot{x}) + \underbrace{(3\dot{x}^2 - 2\dot{x}^3)}_{s(\dot{x})} \cdot \big(h_1(\dot{x}) - h_0(\dot{x})\big), \tag{8.21}$$

where $s()$ is the cubic *blending function*

$$s(\xi) = 3\xi^2 - 2\xi^3, \tag{8.22}$$

that is shown in Fig. 8.5. By minor rearrangement of Eqn. (8.21) we get

$$\begin{aligned} F_4(\dot{x}, g_0, g_1) &= h_0(\dot{x}) + s(\dot{x}) \cdot h_1(\dot{x}) - s(\dot{x}) \cdot h_0(\dot{x}) \\ &= (1 - s(\dot{x})) \cdot \underbrace{h_0(\dot{x})}_{w_0} + s(\dot{x}) \cdot \underbrace{h_1(\dot{x})}_{w_1}, \end{aligned} \tag{8.23}$$

which is just the weighted average of the values $w_0 = h_0(\dot{x})$ and $w_1 = h_1(\dot{x})$, with the relative weights determined by $s(\dot{x})$.[5]

Modified Perlin interpolation

As shown in Eqn. (8.18), the second derivative of Perlin's original interpolating function is generally non-zero at the boundary points of the unit interval. In a modified formulation, Perlin [10] proposed to replace the cubic blending function in Eqn. (8.22) by a fifth-order polynomial,

$$\tilde{s}(\xi) = 10 \cdot \xi^3 - 15 \cdot \xi^4 + 6 \cdot \xi^5 \;=\; \xi^3 \cdot (\xi \cdot (\xi \cdot 6 - 15) + 10). \tag{8.24}$$

[5] $s(\xi)$ is called the *fade* function in [10]. Note that the interpolation with a minimal (cubic) polynomial used in our first attempt (Eqn. (8.12)) cannot be described as a smooth blending of tangent slopes.

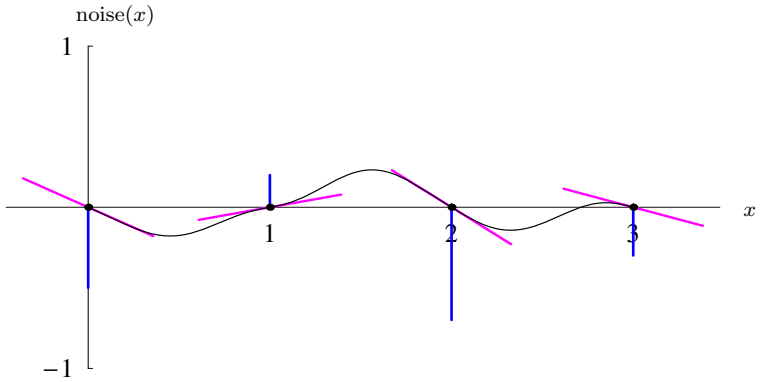


Figure 8.6 Result of interpolation with the modified Perlin function in Eqn. (8.25).

Both blending functions $s(\xi)$ and $\tilde{s}(\xi)$ is shown in Fig. 8.5.[6] The quartic interpolating function in Eqn. (8.17) now turns into the *sixth*-order polynomial

$$\begin{aligned} F_5(\dot{x}, g_0, g_1) &= h_0(\dot{x}) + \big(10\dot{x}^3 - 15\dot{x}^4 + 6\dot{x}^5\big) \cdot \big(h_1(\dot{x}) - h_0(\dot{x})\big) \\ &= g_0 \cdot \dot{x} + \big(10\dot{x}^3 - 15\dot{x}^4 + 6\dot{x}^5\big) \cdot \big(g_1 \cdot (\dot{x} - 1) - g_0 \cdot \dot{x}\big) \\ &= (-6g_0 + 6g_1) \cdot \dot{x}^6 + (15g_0 - 21g_1) \cdot \dot{x}^5 \\ &\qquad + (-10g_0 + 25g_1) \cdot \dot{x}^4 + (-10g_1)\dot{x}^3 + g_0 \cdot \dot{x}\ , \end{aligned} \tag{8.25}$$

for $0 \leq \dot{x} \leq 1$. The function F_5 not only satisfies the constraints in Eqn. (8.6) but has the additional property

$$F_5''(0, g_0, g_1) = F_5''(1, g_0, g_1) = 0, \tag{8.26}$$

for arbitrary gradient values g_0, g_1. Thus the curvature of the interpolated function is zero at both ends of the unit segment and therefore there is no change of curvature at the segment boundaries. In other words, the function F_5 (unlike Perlin's original interpolating function in Eqn. (8.16)) is C2-continuous. An example is shown in Fig. 8.6 and the results obtained with all three interpolation functions F_3, F_4, F_5 (Eqns. (8.12, 8.21, 8.25)) are compared in Fig. 8.7.

Preliminary summary

The raw material for the 1D noise function is a discrete sequence of random gradient values g_u ($u \in \mathbb{Z}$) that are uniformly distributed in the $[-1, 1]$ interval. The values of the continuous noise function at arbitrary positions $x \in \mathbb{R}$ are

[6] The right expression in Eqn. (8.24) is typically preferred because it is less costly to calculate.

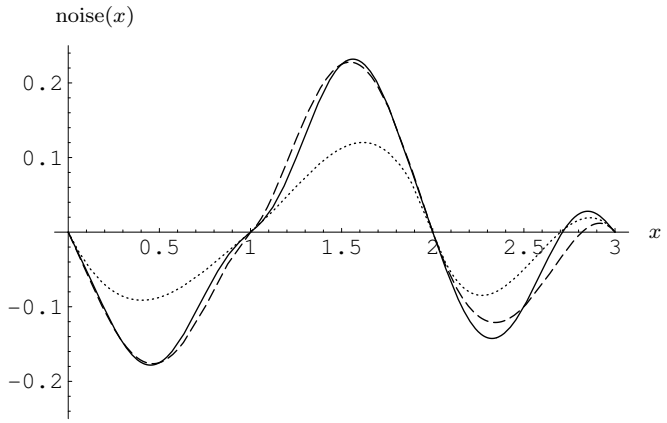


Figure 8.7 Continuous 2D gradient noise obtained with different interpolation functions. Interpolation with the cubic polynomial F_3 (Eqn. (8.12), dotted curve); original Perlin interpolation F_4 (Eqn. (8.21), dashed curve); modified Perlin interpolation F_5 (Eqn. (8.25), continuous curve).

obtained by piecewise interpolation using a local interpolating function F in the form

$$\text{noise}(x) = F(x-u, g_u, g_{u+1}), \tag{8.27}$$

with $u = \lfloor x \rfloor$. This 1D function is implemented in Alg. 8.1 by NOISE(x, grad), which takes a continuous position x and a discrete map of gradient values grad. The actual interpolation is performed by the function

$$\text{INTERPOLATE}(\xi, w_0, w_1) = (1 - \tilde{s}(\xi)) \cdot w_0 + \tilde{s}(\xi) \cdot w_1, \tag{8.28}$$

analogous to Eqn. (8.23). It uses the modified blending function $\tilde{s}(\xi)$ defined in Eqn. (8.24) and thus implements the interpolating function F_5 in Eqn. (8.25).

8.1.2 Frequency of the noise function

Perlin noise owes its realistic appearance to a great extent to the proper combination of multiple noise functions with different frequencies. The individual noise function noise(x) has regularly spaced zero transitions at distance $\tau = 1$ and thus has an inherent periodicity with frequency

$$f = \frac{1}{\tau} = 1. \tag{8.29}$$

Cite THIS series!

Since the interval τ is a unit-less quantity so far, the frequency f has no concrete unit either (see also [2, Sec. 13.3.4]). The continuous signal generated by the function noise(x) is band-limited with a maximum frequency of approximately 1

Algorithm 8.1 Generating single-frequency, one-dimensional Perlin noise. The map grad specifies a particular gradient value in $[-1, 1]$ for any integer coordinates $u \in \mathbb{Z}$.

1: NOISE(x, grad)
Input: continuous position $x \in \mathbb{R}$; gradient map $\mathsf{grad} : \mathbb{Z} \to [-1, 1]$.
Returns a scalar noise value for the continuous position x.
2: $u \leftarrow \lfloor x \rfloor$
3: $g_0 \leftarrow \mathsf{grad}(u)$ ▷ gradient at position u
4: $g_1 \leftarrow \mathsf{grad}(u + 1)$ ▷ gradient at position $u + 1$
5: $\dot{x} = x - u$ ▷ $\dot{x} \in [0, 1]$
6: $w_0 \leftarrow g_0 \cdot \dot{x}$ ▷ value of tangent h_0 at position $\dot{x}$
7: $w_1 \leftarrow g_1 \cdot (\dot{x} - 1)$ ▷ value of tangent h_1 at position $\dot{x}$
8: $w \leftarrow$ INTERPOLATE$(\dot{x}, w_0, w_1)$
9: **return** w

10: INTERPOLATE$(\dot{x}, w_0, w_1)$ ▷ $\dot{x} \in [0, 1], w_0, w_1 \in \mathbb{R}$
11: $s \leftarrow 10 \cdot \dot{x}^3 - 15 \cdot \dot{x}^4 + 6 \cdot \dot{x}^5$ ▷ blending fun. $\tilde{s}(\xi)$, cf. Eqn. (8.24)
12: **return** $(1 - s) \cdot w_0 + s \cdot w_1$.

[5, p. 68].[7] The same noise function, but with *twice* the frequency ($f = 2$), is obtained by

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

i. e., by scaling the function coordinates by the factor 2. Analogously we can reduce the signal frequency by stretching the function, for example, to half the original frequency by

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

In general, the function noise(x) with arbitrary frequency $f \in \mathbb{R}$ is obtained by

$$\text{noise}^{(f)}(x) = \text{noise}(f \cdot x). \tag{8.30}$$

Figure 8.8 shows the noise function from the previous examples with frequencies $f = 1, 2, 3, 4$.

8.1.3 Combining multiple noise functions

Perlin's method involves the combination of multiple, scaled versions of the same noise function with K different frequencies and amplitudes a_i in the form

$$\text{Noise}(x) = \sum_{i=0}^{K-1} a_i \cdot \text{noise}^{(f_i)}(x) = \sum_{i=0}^{K-1} a_i \cdot \text{noise}(f_i \cdot x). \tag{8.31}$$

[7] This fact is important for sampling the continuous function to generate discrete noise signals (see Sec. 8.1.3).

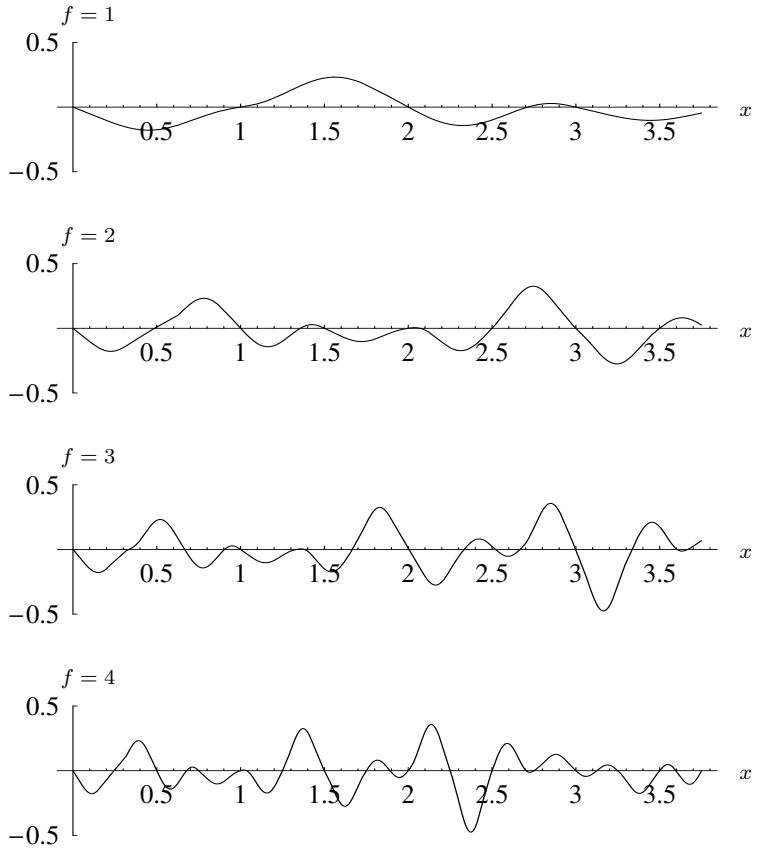


Figure 8.8 A one-dimensional Perlin noise function with different frequencies $f = 1, \ldots 4$.

The frequencies used are not arbitrary but, starting from a given base frequency $f_{\min}$, are successively doubled, i. e., $f_0 = f_{\min}$, $f_1 = 2f_{\min}$, $f_2 = 4f_{\min}$, $f_3 = 8f_{\min}$, etc., and in general

$$f_i = 2^i \cdot f_{\min}, \qquad \text{for } i = 0, 1, \ldots, K-1. \tag{8.32}$$

Successive frequencies f_i and f_{i+1} thus have a $1 : 2$ ratio, which is equivalent to an "octave" step.

Usable frequency range

So how many different frequencies are practical and useful? When generating discrete noise functions we have to consider the sampling theorem which says that signal frequencies above half of the sampling frequency must be avoided. If a continuous function is sampled at regular distances $\tau_s = 1$ (pixel units),

the corresponding sampling frequency is $f_s = 1/\tau_s = 1$ as well. The maximum frequency in the noise function should therefore not be higher than $0.5 f_s$, that is,

$$f_{\max} \leq 0.5. \tag{8.33}$$

Frequencies above 0.5 would lead to aliasing effects and are therefore not useful. For a given number of frequencies or octaves K, the maximum permissible base frequency is thus

$$f_{\min} \leq \frac{f_{\max}}{2^{K-1}} = \frac{0.5}{2^{K-1}} = \frac{1}{2^{(K)}}\,. \tag{8.34}$$

With $f_s = 1$ and $K = 4$ octaves, for example, the maximum base frequency is

$$f_{\min} = f_0 = \frac{1}{2^4} = \frac{1}{16}, \tag{8.35}$$

and consequently $f_1 = 2f_0 = \frac{1}{8}$, $f_2 = 2f_1 = \frac{1}{4}$, and $f_3 = f_{\max} = 2f_2 = \frac{1}{2}$.

Amplitude coefficients

The individual amplitude coefficients a_i can be set arbitrarily but it turns out that their proper choice is crucial to the appearance of the combined noise function Noise(x). In his original formulation, Perlin proposed amplitude values that decrease logarithmically with the frequency index i,

$$a_i = \phi^i, \tag{8.36}$$

where $\phi \in [0, 1]$ is a fixed "persistence" value, typically in the range $0.25 \ldots 0.5$. An example of a combined noise function consisting of three component functions with $f_0 = 1$ and $\phi = 0.5$ is shown in Fig. 8.9. The generation of a one-dimensional, composite noise function is summarized in Alg. 8.2.

8.1.4 Generating "random" gradient samples

A question still being unanswered is how to generate the pseudo-random sequence of gradient samples $g_i = \mathsf{grad}(i)$ for arbitrary positions $i \in \mathbb{Z}$. The mapping $\mathsf{grad}(i)$ should be fixed for a given noise function, i. e., for a any given position i, $\mathsf{grad}(i)$ should always and repeatedly return the same value. So we cannot simply calculate a new random value in each invocation of $\mathsf{grad}(i)$ but need to look for other solutions.

The usual approach is to calculate the gradient samples for a given position index i "on the fly" by a hash function

$$\text{hash}(i) : \mathbb{Z} \to [0, 1] \tag{8.37}$$

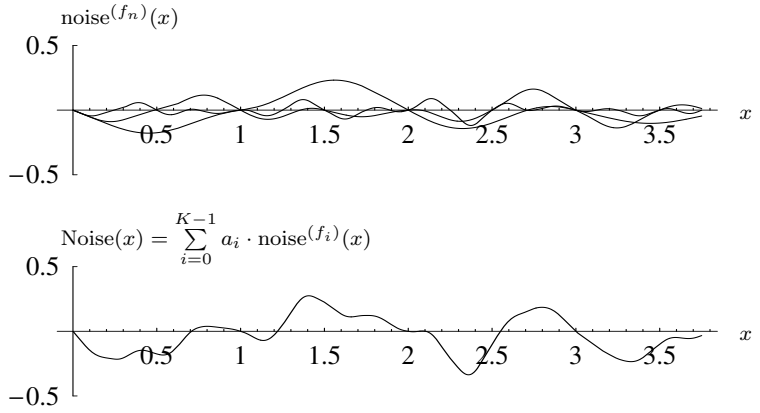


Figure 8.9 Multi-frequency Perlin noise function. Noise(x) is composed of $K = 3$ weighted noise functions $a_i \cdot \text{noise}^{(f_i)}(x)$, with $i = 0, 1, 2$, frequencies $f_0 = 1$, $f_1 = 2$, $f_2 = 4$, and amplitude coefficients $a_0 = 1$, $a_1 = \frac{1}{2}$, $a_2 = \frac{1}{4}$ (persistence $\phi = 0.5$).

Algorithm 8.2 Generating multi-frequency, one-dimensional Perlin noise. The map grad specifies a particular gradient value in $[-1, 1]$ for any integer coordinates $u \in \mathbb{Z}$. $f_{\min}, f_{\max}$ specify the frequency range, the persistence value ϕ controls the component amplitudes.

1: NOISE$(x, \mathsf{grad}, f_{\min}, f_{\max}, \phi)$
   Input: continuous position $x \in \mathbb{R}$; gradient map $\mathsf{grad} : \mathbb{Z} \to [-1, 1]$; min./max. frequency $f_{\min}, f_{\max}$; persistence ϕ.
   Returns a scalar noise value for the continuous position x.
2: $f \leftarrow f_{\min}$ ▷ base frequency $f_0 = f_{\min}$
3: $a \leftarrow 1$ ▷ amplitude $a_0 = \phi^0$
4: $\Sigma \leftarrow 0$
5: **while** $f \le f_{\max}$ **do**
6: $\Sigma \leftarrow \Sigma + a \cdot \text{NOISE}(f \cdot x, \mathsf{grad})$ ▷ NOISE() as defined in Alg. 8.1
7: $f \leftarrow 2 \cdot f$ ▷ $f_i = 2^i \cdot f_{\min}$
8: $a \leftarrow \phi \cdot a$ ▷ $a_i = \phi^i$
9: **return** Σ.

which returns a particular pseudo-random value that only depends on i. If the values produced by the has function are uniformly distributed in $[0, 1]$ then the gradient samples, calculated as

$$\mathsf{grad}(i) = 2 \cdot \text{hash}(i) - 1, \tag{8.38}$$

are also uniformly distributed in the $[-1, 1]$ interval. For the implementation of hash(i), Perlin proposed an efficient hashing method that is based on successive lookup operations on a short permutation table. Details and alternative hash

functions are described in Sec. 8.4.

8.2 Two-dimensional noise functions

Two-dimensional noise functions can be used to generate random images, such as textures, among other things. The process in 2D is analogous to the 1D case: s individual 2D noise functions with different frequencies are generated and then combined by weighted point-wise summation. Let

$$\text{noise}(x, y) : \mathbb{R}^2 \to \mathbb{R}, \tag{8.39}$$

be a single, continuous 2D noise function with a fixed base frequency $f = 1$ in both dimensions. The discrete 2D lattice points $(u, v) \in \mathbb{Z}^2$ are again zero positions of the noise function, with the (random) gradient samples

$$\mathrm{g}_{u,v} = \begin{pmatrix} \mathsf{grad}_x(u, v) \\ \mathsf{grad}_y(u, v) \end{pmatrix} = \mathsf{grad}(u, v) \tag{8.40}$$

analogous to the 1D notation. Each sample $\mathrm{g}_{u,v}$ is a two-dimensional gradient vector, whose elements

$$\mathsf{grad}_x(u, v) = \frac{\partial\, \text{noise}}{\partial x}(u, v) \qquad \text{and} \qquad \mathsf{grad}_y(u, v) = \frac{\partial\, \text{noise}}{\partial y}(u, v) \tag{8.41}$$

specify the partial derivatives of the 2D noise function with respect to x- and y-axis, respectively. Each gradient vector defines a *tangential plane* that intersects the x/y-plane at the lattice point (u, v) with slopes $\mathsf{grad}_x(u, v)$ and $\mathsf{grad}_y(u, v)$, defined by the linear equation[8]

$$h_{u,v}(x, y) = \mathrm{g}_{u,v}^{\mathsf{T}} \cdot \left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} u \\ v \end{pmatrix} \right] = \mathrm{g}_{u,v}^{\mathsf{T}} \cdot \begin{pmatrix} x-u \\ y-v \end{pmatrix} \tag{8.42}$$

$$= \mathsf{grad}_x(u, v) \cdot (x-u) + \mathsf{grad}_y(u, v) \cdot (y-v), \tag{8.43}$$

where $\cdot$ in Eqn. (8.42) denotes the inner (dot) product of the involved vectors.

As in the one-dimensional case, the continuous 2D noise function $\text{noise}(x, y)$ is calculated by interpolating the gradients given at the discrete lattice points (u, v). Here the local interpolation is limited to the unit square $[0, 1] \times [0, 1]$ using the normalized coordinates

$$\dot{x} = x - u \qquad \text{and} \qquad \dot{y} = y - v, \tag{8.44}$$

[8] Compare this to the definition of the 1D tangent in Eqn. (8.19).

with $u = \lfloor x \rfloor$ and $v = \lfloor y \rfloor$. The interpolation at a given position $(x, y) \in \mathbb{R}^2$ considers exactly four surrounding lattice points:

$$\begin{aligned} \mathrm{g}_{00} &= \mathsf{grad}(u, v), & \mathrm{g}_{10} &= \mathsf{grad}(u+1, v), \\ \mathrm{g}_{01} &= \mathsf{grad}(u, v+1), & \mathrm{g}_{11} &= \mathsf{grad}(u+1, v+1). \end{aligned} \tag{8.45}$$

Following Eqn. (8.42), the values of the four tangent planes $h_{00} \ldots h_{11}$ at the target position $(\dot{x}, \dot{y})$ are

$$\begin{aligned} w_{00} = h_{00}(\dot{x}, \dot{y}) &= \boldsymbol{g}_{00}^{\mathsf{T}} \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right], \quad w_{10} = h_{10}(\dot{x}, \dot{y}) = \boldsymbol{g}_{10}^{\mathsf{T}} \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right], \\ w_{01} = h_{01}(\dot{x}, \dot{y}) &= \boldsymbol{g}_{01}^{\mathsf{T}} \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right], \quad w_{11} = h_{11}(\dot{x}, \dot{y}) = \boldsymbol{g}_{11}^{\mathsf{T}} \cdot \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right]. \end{aligned}$$

Interpolation is now performed over the four tangent values $w_{00} \ldots w_{11}$ in two successive steps, using a sigmoidal blending function $s(\cdot)$ as in the 1D case (see Eqn. (8.23)). In the first step, interpolation is performed in the x-direction with the intermediate results

$$\begin{aligned} w_0 &= (1 - s(\dot{x})) \cdot w_{00} + s(\dot{x}) \cdot w_{10}, \\ w_1 &= (1 - s(\dot{x})) \cdot w_{01} + s(\dot{x}) \cdot w_{11}, \end{aligned} \tag{8.46}$$

followed by interpolation in the y direction with the final result

$$\mathrm{noise}(x, y) = w = (1 - s(\dot{y})) \cdot w_0 + s(\dot{y}) \cdot w_1. \tag{8.47}$$

The one-dimensional blending function $s(\cdot)$ is again either the original (cubic) Perlin function in Eqn. (8.22) or the modified function $\tilde{s}(\cdot)$ in Eqn. (8.24).

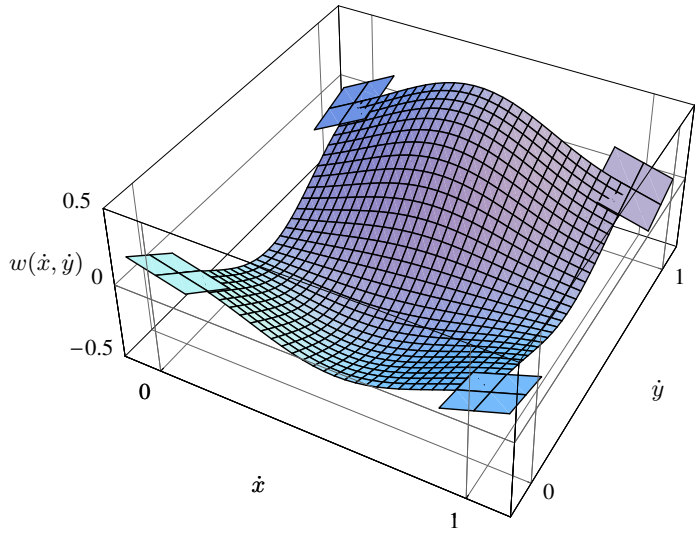
Figure 8.10 illustrates the continuous interpolation in 2D from four randomly chosen gradient vectors $\boldsymbol{g}_{00} \ldots \boldsymbol{g}_{11}$ placed at the corners of the unit square. The above process of single-frequency 2D noise synthesis is summarized in Alg. 8.3. Multi-frequency 2D noise functions are composed analogous to the 1D case, as defined in Eqn. (8.31),

$$\mathrm{Noise}(x, y) = \sum_{i=0}^{K-1} a_i \cdot \mathrm{noise}^{(f_i)}(x, y) = \sum_{i=0}^{K-1} a_i \cdot \mathrm{noise}(f_i \cdot x, f_i \cdot y), \tag{8.48}$$

again using suitable amplitude coefficients a_i (see Eqn. (8.36)). Note that the same frequency f_i is used in both dimensions.

2D examples

Figure 8.11 shows examples of two-dimensional Perlin noise with different numbers of frequency components. The influence of the persistence value ϕ, which controls the relative magnitude of the amplitude coefficients, is demonstrated in Fig. 8.12



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

Figure 8.10 Continuous noise function in 2D. Four randomly chosen gradient vectors $\boldsymbol{g}_{00}$, ..., $\boldsymbol{g}_{11}$ are placed at the corners of the unit square, which define the four tangent planes. The continuous surface is obtained by interpolating the values of the four tangent planes using the blending function $\tilde{s}(\cdot)$ in Eqn. (8.24).

8.3 N-dimensional noise

Extending the concept of gradient-based noise functions from 2D space to N-dimensional space is straightforward, although the results are difficult to visualize, of course. Three-dimensional noise functions, for example, are used in computer graphics for texturing 3D bodies, such that the resulting texture is a real 3D property that affects not only the body surface but the complete body itself. A typical example is the use of Perlin noise for generating the 3D texture of marble and other natural materials.[9]

Analogous to the situation in 2D, a single-frequency, N-dimensional noise function can be written as

$$\text{noise}(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}, \tag{8.49}$$

where $\boldsymbol{x} = \big(x_0, x_1, \ldots x_{N-1}\big)^\mathsf{T}$ denotes a continuous coordinate point in N-dimensional space. The function noise($\boldsymbol{x}$) thus returns a scalar value for any position $\boldsymbol{x} \in \mathbb{R}^N$. We also assume that the base frequency is set to $f = 1$ for any of the N dimensions, i. e., the sampling rate is the same in all directions. The

[9] See http://mrl.nyu.edu/~perlin/doc/vase.html for a striking example.

Algorithm 8.3 Generating single-frequency, two-dimensional Perlin noise. The map $\mathsf{grad}(u, v)$ specifies a particular (randomly distributed) 2D gradient vector $\mathrm{g} \in \mathbb{R}^2$ for any 2D lattice coordinate $(u, v) \in \mathbb{Z}^2$.

NOISE(x, y, grad)
Input: continuous 2D position $x, y \in \mathbb{R}^2$; 2D gradient map $\mathsf{grad} : \mathbb{Z}^2 \to [-1, 1]^2$.
Returns a scalar noise value for position (x, y).

$u \leftarrow \lfloor x \rfloor, \quad v \leftarrow \lfloor y \rfloor$
$\mathrm{g}_{00} \leftarrow \mathsf{grad}(u, v)$ ▷ 2D gradient vectors
$\mathrm{g}_{10} \leftarrow \mathsf{grad}(u+1, v)$
$\mathrm{g}_{01} \leftarrow \mathsf{grad}(u, v+1)$
$\mathrm{g}_{11} \leftarrow \mathsf{grad}(u+1, v+1)$
$\dot{x} \leftarrow x - u, \quad \dot{y} \leftarrow y - v$ ▷ $\dot{x}, \dot{y} \in [0, 1]$
$w_{00} \leftarrow \boldsymbol{g}_{00}^{\mathsf{T}} \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$ ▷ tangent values (dot vector products)
$w_{10} \leftarrow \boldsymbol{g}_{10}^{\mathsf{T}} \cdot \begin{pmatrix} \dot{x}-1 \\ \dot{y} \end{pmatrix}$
$w_{01} \leftarrow \boldsymbol{g}_{01}^{\mathsf{T}} \cdot \begin{pmatrix} \dot{x} \\ \dot{y}-1 \end{pmatrix}$
$w_{11} \leftarrow \boldsymbol{g}_{11}^{\mathsf{T}} \cdot \begin{pmatrix} \dot{x}-1 \\ \dot{y}-1 \end{pmatrix}$
$w \leftarrow$ INTERPOLATE$(\dot{x}, \dot{y}, w_{00}, w_{10}, w_{01}, w_{11})$
return w.

INTERPOLATE$(\dot{x}, \dot{y}, w_{00}, w_{10}, w_{01}, w_{11})$
$s_x \leftarrow 10 \cdot \dot{x}^3 - 15 \cdot \dot{x}^4 + 6 \cdot \dot{x}^5$ ▷ blending fun. $\tilde{s}(\xi)$, see Eqn. (8.24)
$s_y \leftarrow 10 \cdot \dot{y}^3 - 15 \cdot \dot{y}^4 + 6 \cdot \dot{y}^5$
$w_0 = (1-s_x) \cdot w_{00} + s_x \cdot w_{10}$ ▷ interpolate in x-direction ($2\times$)
$w_1 = (1-s_x) \cdot w_{01} + s_x \cdot w_{11}$
return $(1 - s_y) \cdot w_0 + s_y \cdot w_1$. ▷ interpolate in y-direction ($1\times$)

discrete lattice points $\boldsymbol{p} = (p_0, p_1, \ldots p_{N-1})^{\mathsf{T}} \in \mathbb{Z}^N$ are again zero positions of the resulting noise function, with the attached (randomly distributed) gradient vectors

$$\mathrm{g}_{\boldsymbol{p}} = \big(\mathsf{grad}_0(\boldsymbol{p}), \mathsf{grad}_1(\boldsymbol{p}), \ldots, \mathsf{grad}_{N-1}(\boldsymbol{p})\big)^{\mathsf{T}}, \tag{8.50}$$

whose elements $\mathsf{grad}_k(\boldsymbol{p})$ are equivalent to the partial derivatives of the continuous noise function in the direction x_k at the lattice point $\boldsymbol{p}$, that is,

$$\mathsf{grad}_k(\boldsymbol{p}) = \frac{\partial\,\mathrm{noise}}{\partial x_k}(\boldsymbol{p})\,. \tag{8.51}$$

Each gradient vector $\mathrm{g}_{\boldsymbol{p}}$ specifies a tangent hyper-plane[10] in $\mathbb{R}^{N+1}$ that is es-

[10] A tangent to a scalar-valued 1D function is a line in 2D and the tangent to a

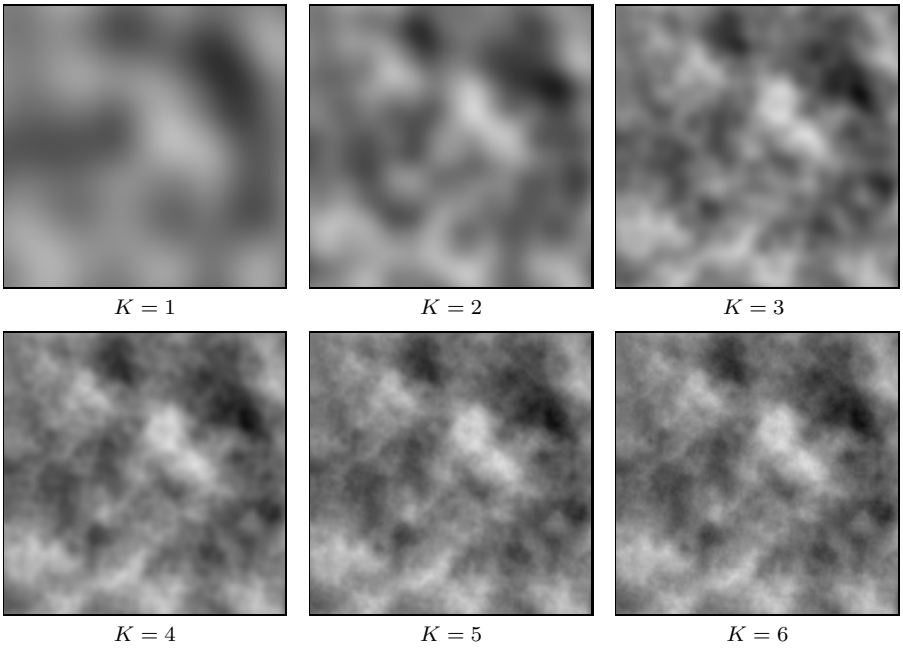


Figure 8.11 Examples of multi-frequency 2D Perlin noise with $K = 1, \dots, 6$ frequency components $f_i = 0.01, 0.02, 0.04, 0.08, 0.16, 0.32$ (cycles per pixel unit). Persistence $\phi = 0.5$, images size is 300×300 pixels.

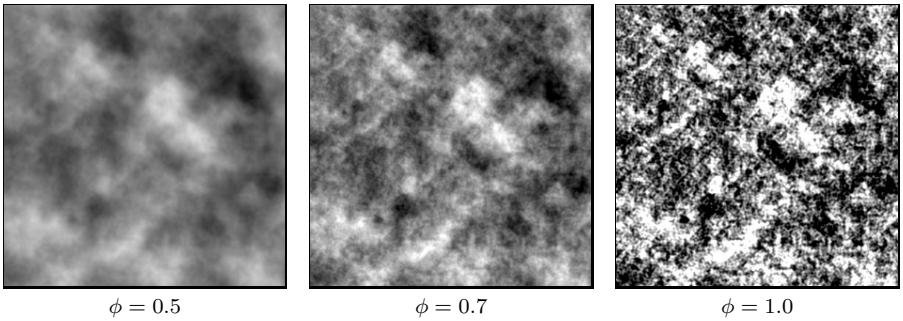


Figure 8.12 Examples of 2D Perlin noise for different persistence values ϕ.

tablished by the linear function $h_{\boldsymbol{p}}(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}$,

$$h_{\boldsymbol{p}}(\boldsymbol{x}) = \mathrm{g}_{\boldsymbol{p}}^{\mathsf{T}} \cdot (\boldsymbol{x} - \boldsymbol{p}) = \sum_{k=0}^{N-1} \mathsf{grad}_k(\boldsymbol{p}) \cdot (x_k - p_k). \tag{8.52}$$

The tangent function $h_{\boldsymbol{p}}(\boldsymbol{x})$ returns a scalar value for each position $\boldsymbol{x} \in \mathbb{R}^N$; it

2D function is a plane in 3D. In general, the tangent to a scalar function in N dimensions forms a hyperplane in a space with $N+1$ dimensions.

is zero at corresponding lattice point $\boldsymbol{p}$,

$$h_{\boldsymbol{p}}(\boldsymbol{p}) = \mathrm{g}_{\boldsymbol{p}}^{\mathsf{T}} \cdot \big(\boldsymbol{p} - \boldsymbol{p}\big) = \mathrm{g}_{\boldsymbol{p}}^{\mathsf{T}} \cdot \mathbf{0} = 0, \tag{8.53}$$

since the hyperplane must pass through $\boldsymbol{p}$, which is also required to be a zero position of the noise function.

Analogous to the one- and two-dimensional case, the continuous noise function noise($\boldsymbol{x}$), at some point $\boldsymbol{x} \in \mathbb{R}^N$, is again obtained by interpolation over the values of the tangent functions $h_{\boldsymbol{p}_j}(\boldsymbol{x})$ attached to the surrounding lattice points $\boldsymbol{p}_j$. While the gradients of *two* and *four* lattice points are considered in the 1D and 2D cases, respectively, the interpolation in 3D takes the $2^3 = 8$ corner points of the cube surrounding $\boldsymbol{x}$. In general, 2^N surrounding lattice points $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots \boldsymbol{p}_{2^N-1}$ are used for the interpolation in N-dimensional space. These lattice points form an N-dimensional unit-hypercube around the continuous coordinate $\boldsymbol{x} = (x_0, \ldots, x_{N-1})^{\mathsf{T}}$. Starting from the canonical corner point of this hypercube,

$$\boldsymbol{p}_0 = \lfloor \boldsymbol{x} \rfloor = \big(\lfloor x_0 \rfloor, \lfloor x_1 \rfloor, \ldots, \lfloor x_{N-1} \rfloor\big)^{\mathsf{T}}, \tag{8.54}$$

the coordinates of all 2^N corner points (vertices) $\boldsymbol{p}_j$ can be calculated in the form $\boldsymbol{p}_j = \boldsymbol{p}_0 + \boldsymbol{q}_j$, where each $\boldsymbol{q}_j$ is an N-dimensional vector consisting of 0/1 elements, that is,

$$\begin{aligned}
\boldsymbol{p}_0 &= \boldsymbol{p}_0 + \boldsymbol{q}_0 &&= \boldsymbol{p}_0 + (0,0,0,\ldots,0)^{\mathsf{T}},\\
\boldsymbol{p}_1 &= \boldsymbol{p}_0 + \boldsymbol{q}_1 &&= \boldsymbol{p}_0 + (1,0,0,\ldots,0)^{\mathsf{T}},\\
\boldsymbol{p}_2 &= \boldsymbol{p}_0 + \boldsymbol{q}_2 &&= \boldsymbol{p}_0 + (0,1,0,\ldots,0)^{\mathsf{T}},\\
\boldsymbol{p}_3 &= \boldsymbol{p}_0 + \boldsymbol{q}_3 &&= \boldsymbol{p}_0 + (1,1,0,\ldots,0)^{\mathsf{T}},\\
&\vdots \qquad\quad \vdots && \qquad\quad \vdots\\
\boldsymbol{p}_{2^N-1} &= \boldsymbol{p}_0 + \boldsymbol{q}_{2^N-1} &&= \boldsymbol{p}_0 + (1,1,1,\ldots,1)^{\mathsf{T}}.
\end{aligned} \tag{8.55}$$

The elements of any vector $\boldsymbol{q}_j$ are obviously identical to the bit sequence ($\mathtt{b}_0$, $\mathtt{b}_1, \ldots, \mathtt{b}_{N-1}$) obtained by expressing integer j as a binary number,

$$j = \mathtt{j}_0 + \mathtt{j}_1 \cdot 2^1 + \ldots + \mathtt{j}_k \cdot 2^k + \ldots \mathtt{j}_{N-1} \cdot 2^{N-1}. \tag{8.56}$$

Thus, the k^{th} component of $\boldsymbol{q}_j$ is identical to bit $\mathtt{j}_k$ in j's binary representation, that is,

$$\boldsymbol{q}_j(k) = (\mathtt{j}_0, \mathtt{j}_1, \ldots, \mathtt{j}_{N-1}), \qquad \text{with } \mathtt{j}_k = \big(j \div 2^k\big) \bmod 2, \tag{8.57}$$

for $0 \le j < 2^N$ and $0 \le k < N$.[11] For the following algorithms, we define the function VERTEX(j, N) to return the coordinate $\boldsymbol{q}_j$ of the j^{th} corner point of

[11] In Eqn. (8.57), $a \div b$ denotes the integer division of a and b (quotient of a, b). $\mathtt{j}_0$ is the least-significant bit.

the N-dimensional unit cube (see Alg. 8.4). Also, we use the short notation g_j for the discrete gradient vectors $\mathrm{g}_{\boldsymbol{p}_j}$ and

$$h_j(\boldsymbol{x}) = \mathrm{g}_j^\mathsf{T} \cdot \big(\boldsymbol{x} - \boldsymbol{p}_j\big)\,, \tag{8.58}$$

for the corresponding gradient planes $h_{\boldsymbol{p}_j}(\boldsymbol{x})$, defined in Eqn. (8.52).

8.3.1 Blending N-dimensional tangent hyperplanes

The final (scalar) value of the N-dimensional noise function at the given position $\boldsymbol{x} = \big(x_0, x_1, \ldots x_{N-1}\big)^\mathsf{T}$ is obtained by blending the tangent functions $h_j(\boldsymbol{x})$ of all surrounding hypercube vertices $\boldsymbol{p}_j$. The interpolation is not done in a single step, but (analogous to the two-dimensional case) step by step for each of the N dimensions.

Along any dimension k, the N-dimensional hypercube has exactly 2^{N-1} edges, each connecting two vertices $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$, whose coordinate vectors are the same except for dimension k, that is,

$$\boldsymbol{p}_a(k) \neq \boldsymbol{p}_b(k) \quad \text{and} \quad \boldsymbol{p}_a(l) = \boldsymbol{p}_b(l), \quad \text{for } l \neq k,\ 0 \leq k < N. \tag{8.59}$$

In each step, the interpolation involves the tangent values $w_a = h_a(\boldsymbol{x})$ and $w_b = h_b(\boldsymbol{x})$ of two hypercube vertices $\boldsymbol{p}_a, \boldsymbol{p}_b$ that are adjacent along dimension k in the (familiar) form

$$\bar{w}_{a,b} = (1 - s(\dot{x}_k)) \cdot w_a + s(\dot{x}_k) \cdot w_b\,, \tag{8.60}$$

where $\dot{x}_k$ is the k^{th} component of the normalized position $\dot{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{p}_0$, and $s(\xi)$ is the non-linear blending function defined in Eqns. (8.22, 8.24). Thus, in each interpolation step, the values from *two* neighboring vertices are reduced to a single value. The resulting values can be associated to the vertices of another hypercube of one dimension less than the previous hypercube and it therefore takes N interpolation steps to reduce the original 2^N vertex values of the N-dimensional hypecube to a scalar value (of dimension zero). Starting with the initial sequence of all 2^N vertex values

$$\boldsymbol{w}^{(N)} = (w_0, \ldots w_{2^N-1}) = (h_0(\boldsymbol{x}), \ldots h_{2^N-1}(\boldsymbol{x})), \tag{8.61}$$

the N-dimensional interpolation could be accomplished by the following steps:

$$\begin{aligned}
\boldsymbol{w}^{(N-1)} &\leftarrow \textsc{Interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N)}, 0)\\
\boldsymbol{w}^{(N-2)} &\leftarrow \textsc{Interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N-1)}, 1)\\
&\vdots \qquad\qquad \vdots\\
\boldsymbol{w}^{(N-k)} &\leftarrow \textsc{Interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(N-k+1)}, k+1)\\
&\vdots \qquad\qquad \vdots\\
w = \boldsymbol{w}^{(0)} &\leftarrow \textsc{Interpolate}(\boldsymbol{x}, \boldsymbol{w}^{(1)}, N-1)
\end{aligned} \tag{8.62}$$

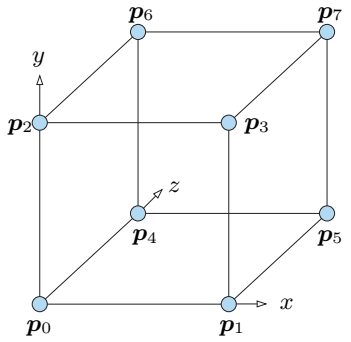


Figure 8.13 Vertices of the 3D unit cube arranged in lexicographic order, $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots \boldsymbol{p}_7$. $\boldsymbol{p}_j \equiv \boldsymbol{p}_{\text{xyz}}$, where xyz denotes the binary representation of j: $\boldsymbol{p}_0 \equiv \boldsymbol{p}_{000}$, $\boldsymbol{p}_1 \equiv \boldsymbol{p}_{100}$, $\boldsymbol{p}_2 \equiv \boldsymbol{p}_{010}$, $\ldots$, $\boldsymbol{p}_7 \equiv \boldsymbol{p}_{111}$. Note that successive pairs of vertices $\boldsymbol{p}_j, \boldsymbol{p}_{j+1}$ are connected by an edge along the first dimension (x).

In each step, the length of $\boldsymbol{w}$ is thus cut in half and the final result is a scalar value $w = \boldsymbol{w}^{(0)}$. The function INTERPOLATE($\boldsymbol{x}, \boldsymbol{w}, k$) (see Alg. 8.4) averages the 2^{N-k} values in $\boldsymbol{w}$ and returns a new vector with half the length of the input vector $\boldsymbol{w}$. It is assumed that the elements of the vector $\boldsymbol{w}^{(n)}$,

$$\boldsymbol{w}^{(N)} = (w_0, w_1, \ldots w_i, \ldots w_{2^N-1}), \tag{8.63}$$

are arranged in "lexicographic" order with respect to the coordinates of the hypercube, as illustrated in Fig. 8.13 for a 3D cube ($N = 3$). If the vector $\boldsymbol{w}$ is lexicographically ordered, pairs of adjacent elements correspond to vertices of the hypercube connected by an edge along the first dimension. The elements of an interpolated vector $\boldsymbol{w}^{(n-1)}$ are calculated from the elements of the previous vector $\boldsymbol{w}^{(n)}$ as

$$\boldsymbol{w}^{(n-1)}(i) = (1-s(x_k)) \cdot \boldsymbol{w}^{(n)}(2i) + s(x_k) \cdot \boldsymbol{w}^{(n)}(2i+1)\,, \tag{8.64}$$

for $i = 0, \ldots, 2^{(n-1)}$ and $k = N-n$. x_k denotes the k^{th} component of the N-dimensional position vector $\boldsymbol{x}$ and $s()$ is the sigmoidal blending function (see Eqn. (8.22)).

In this case, the hypercube has $2^3 = 8$ elements $w_j = w_{\text{xyz}}$, where xyz is the binary representation of j (x = $\mathtt{j}_0$, y = $\mathtt{j}_1$, z = $\mathtt{j}_2$). The interpolation is

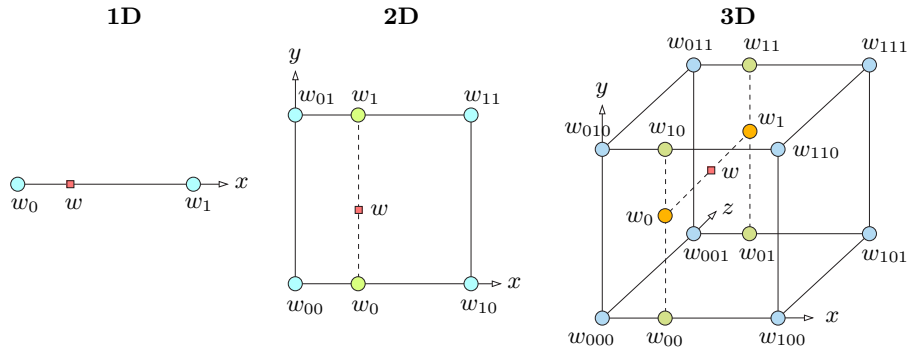


Figure 8.14 Step-wise interpolation in 1D, 2D, and 3D. The blue dots represent values w_{xyz} attached to the vertices of the N-dimensional hypercube. In each step, interpolation is performed along one dimension. The green dots mark the intermediate values after the first interpolation step, the yellow dots after the second step. The red box marks the final value w for the corresponding spatial position.

now calculated in three steps:

$$\begin{aligned} \boldsymbol{w}^{(3)} &= (w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7) \\ \text{Step 1:} \quad &= (\underbrace{w_{000}, w_{100}}_{\downarrow w_{00}}, \underbrace{w_{010}, w_{110}}_{\downarrow w_{10}}, \underbrace{w_{001}, w_{101}}_{\downarrow w_{01}}, \underbrace{w_{011}, w_{111}}_{\downarrow w_{11}}) \qquad (8.65) \\ \text{Step 2:} \quad \boldsymbol{w}^{(2)} &= (\underbrace{w_{00}, w_{10}}_{\downarrow w_0}, \underbrace{w_{01}, w_{11}}_{\downarrow w_1}) \qquad (8.66) \\ \text{Step 3:} \quad \boldsymbol{w}^{(1)} &= (\underbrace{w_0, w_1}_{\downarrow w}) \qquad (8.67) \end{aligned}$$

In Step 1, interpolation is performed along the x-axis (dimension $k = 0$) over 4 pairs of values (Eqn. (8.65)), then along the y-axis (dimension $k = 1$) over 2 pairs of values (Eqn. (8.66)), and finally along the z-axis (dimension $k = 2$) over 1 pair of values (Eqn. (8.67)), with the scalar value w as the final result.

This interpolation process and the generation of N-dimensional, single-frequency Perlin noise are summarized in Alg. 8.4. Note that the function INTERPOLATE() is defined recursively but can easily be converted to an iterative procedure, analogous to Eqn. (8.62). *Multi-frequency* N-dimensional noise can be generated as in the 1D and 2D case (see Eqns. (8.31, 8.68) as

$$\text{Noise}(\boldsymbol{x}) = \sum_{i=0}^{K-1} a_i \cdot \text{noise}(f_i \cdot \boldsymbol{x}) \,, \qquad (8.68)$$

Algorithm 8.4 Generating single-frequency, N-dimensional Perlin noise. The map $\mathsf{grad}(\boldsymbol{p})$ specifies a particular (randomly distributed) N-dimensional gradient vector $\mathrm{g} \in \mathbb{R}^N$ for any lattice coordinate $\boldsymbol{p} \in \mathbb{Z}^N$.

NOISE($\boldsymbol{x}$, grad)
Input: continuous position $\boldsymbol{x} \in \mathbb{R}^N$; N-dimensional gradient map $\mathsf{grad} : \mathbb{Z}^N \to [-1,1]^N$. Returns a scalar noise value for position $\boldsymbol{x}$.
$N \leftarrow |\boldsymbol{x}|$ ▷ dimension of space
$\boldsymbol{p}_0 \leftarrow \lfloor \boldsymbol{x} \rfloor$ ▷ origin of hypercube around $\boldsymbol{x}$
$\boldsymbol{Q} \leftarrow (\boldsymbol{q}_0, .. \boldsymbol{q}_{2^N-1})$, with $\boldsymbol{q}_j = \text{VERTEX}(j, N)$ ▷ vertices $\boldsymbol{q}_j \in \{0,1\}^N$
$\boldsymbol{G} \leftarrow (\boldsymbol{g}_0, .. \boldsymbol{g}_{2^N-1})$, with $\boldsymbol{g}_j = \mathsf{grad}(\boldsymbol{p}_0 + \boldsymbol{q}_j)$ ▷ gradient vectors $\in \mathbb{R}^N$
$\dot{\boldsymbol{x}} \leftarrow \boldsymbol{x} - \boldsymbol{p}_0$ ▷ $\dot{\boldsymbol{x}} \in [0,1]^N$
$\boldsymbol{w} \leftarrow (w_0, .. w_{2^N-1})$, with $w_j = \boldsymbol{g}_j^\mathsf{T} \cdot (\dot{\boldsymbol{x}} - \boldsymbol{q}_j)$ ▷ gradient values $\in \mathbb{R}$
$w \leftarrow \text{INTERPOLATE}(\dot{\boldsymbol{x}}, \boldsymbol{w}, 0)$
return w.

INTERPOLATE($\dot{\boldsymbol{x}}, \boldsymbol{w}, k$)
Interpolate the 2^{N-k} values in $\boldsymbol{w}$ at point $\dot{\boldsymbol{x}}$ along dimension k ($\dot{\boldsymbol{x}} \in [0,1]^N$, $0 \le k < N$).
if $|\boldsymbol{w}| = 1$ **then** ▷ done, end of recursion ($k = N-1$)
  return $\boldsymbol{w}(0)$.
else ▷ $|\boldsymbol{w}| > 1$
  $\dot{x}_k \leftarrow \dot{\boldsymbol{x}}(k)$ ▷ select dimension k of $\dot{\boldsymbol{x}}$
  $s \leftarrow 10 \cdot \dot{x}_k^3 - 15 \cdot \dot{x}_k^4 + 6 \cdot \dot{x}_k^5$ ▷ blending function $\tilde{s}(x)$
  $M \leftarrow |\boldsymbol{w}| \div 2$ ▷ $\boldsymbol{w}'$ is half the size of $\boldsymbol{w}$
  $\boldsymbol{w}' \leftarrow$ new vector of size M, $w'_i \in \mathbb{R}$
  for $i \leftarrow 0, \ldots, M-1$ **do** ▷ fill the new vector $\boldsymbol{w}'$
    $w_a \leftarrow \boldsymbol{w}(2i)$
    $w_b \leftarrow \boldsymbol{w}(2i+1)$
    $\boldsymbol{w}'(i) \leftarrow (1-s) \cdot w_a + s \cdot w_b$ ▷ actual interpolation
  return $\text{INTERPOLATE}(\dot{\boldsymbol{x}}, \boldsymbol{w}', k+1)$. ▷ do next dimension

VERTEX(j, N)
Returns the coordinate vector $\boldsymbol{p} \in \{0,1\}^N$ for vertex j of the N-dimensional unit-hypercube ($0 \le j < 2^N$).
$\boldsymbol{p} \leftarrow (p_0, v_1, \ldots, p_{N-1})$, with $p_k = \left(j \div 2^k\right) \bmod 2$ ▷ see Eqn. (8.57)
return $\boldsymbol{p}$.

with suitable amplitude coefficients a_i (see Eqn. (8.36) and Alg. 8.2).

8.4 Integer hash functions

In the previous section, we discussed the use of hash functions for implementing the map $\mathsf{grad}(\boldsymbol{p})$, which associates a fixed (but randomly chosen) gradient vector $\boldsymbol{g_p}$ to the discrete lattice point $\boldsymbol{p} \in \mathbb{Z}^N$. The map $\mathsf{grad}(\boldsymbol{p})$ must be deterministic in the sense that repeated lookup operations must always return the *same* result for a given lattice point $\boldsymbol{p}$. Moreover, the components of the gradient vectors should be uniformly distributed in the range $[-1, 1]$.

Let us first consider the *one-dimensional* case. As described in Sec. 8.1.4, we want to use a map of the form $\mathsf{grad}(p) = 2 \cdot \mathrm{hash}(p) - 1$ (see Eqn. (8.38)) for calculating the pseudo-random gradient values, based on a hash function

$$\mathrm{hash}(p) : \mathbb{Z} \to [0, 1], \tag{8.69}$$

which returns a particular real value in the range $[0, 1]$ for a given integer argument p. For varying arguments $p \in \mathbb{Z}$, the results should be "random" and uniformly distributed in $[0, 1]$.

In general, for noise functions over N dimensions, a hash function is required that accepts discrete lattice coordinates $\boldsymbol{p} \in \mathbb{Z}^N$ and returns N-dimensional random vectors, that is,

$$\mathrm{hash}(\boldsymbol{p}) : \mathbb{Z}^N \to [0, 1]^N. \tag{8.70}$$

The results should be uncorrelated between the different dimensions, which can be accomplished by using a different scalar hash function for every dimension. For example, in the two-dimensional case with $\boldsymbol{p} = (u, v)$, this could be done in the form

$$\mathrm{hash}(u, v) = \begin{pmatrix} \mathrm{hash}_x(u, v) \\ \mathrm{hash}_y(u, v) \end{pmatrix}, \tag{8.71}$$

using the two hash functions $\mathrm{hash}_x()$ and $\mathrm{hash}_y()$. The corresponding 2D gradient map is

$$\mathsf{grad}(u, v) = 2 \cdot \begin{pmatrix} \mathrm{hash}_x(u, v) \\ \mathrm{hash}_y(u, v) \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{8.72}$$

such that the components of the resulting vector are uniformly distributed in the range $[-1, 1]$. In the N-dimensional case, this is

$$\mathsf{grad}(\boldsymbol{p}) = 2 \cdot \begin{pmatrix} \mathrm{hash}_0(\boldsymbol{p}) \\ \mathrm{hash}_1(\boldsymbol{p}) \\ \vdots \\ \mathrm{hash}_{N-1}(\boldsymbol{p}) \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, \tag{8.73}$$

for $\boldsymbol{p} \in \mathbb{Z}^N$. How the individual (scalar-valued) hash functions are defined is discussed in the following.

8.4.1 Hashing with permutation tables

For efficiency reasons,[12] the hash operation in Perlin's approach [9] is implemented by multiple lookups into a fixed, pseudo-random permutation map $\mathsf{P} : \mathbb{Z} \to \mathbb{Z}$ of size 256, with $\mathsf{P}(i) \in [0, 255]$ and $\mathsf{P}(i) \neq \mathsf{P}(j)$ for any $i \neq j$. For example, the table

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

is used in [9] (see the complete listing in Prog. 8.1). In the one-dimensional case, the permutation-based hash function is simply

$$\text{hash}(u) = \frac{1}{255} \cdot \mathsf{P}(u \bmod 256), \tag{8.74}$$

which is a value in $[0, 1]$. Although the result repeats with the (relatively short) period 256, this is not immediately visible in the final noise signal when multiple frequencies are combined. Longer cycles can be be obtained by cascading multiple short permutation maps, without sacrificing the fast computability [7]. The main advantage of this method becomes apparent in the multi-dimensional case. Here the same permutation table P is applied repeatedly to the individual components of the lattice coordinates, for example,

$$\text{hash}(u, v) = \tfrac{1}{256}\,(\mathsf{P}(\mathsf{P}(u \bmod 256) + v) \bmod 256), \tag{8.75}$$

$$\text{hash}(u, v, w) = \tfrac{1}{256}\,(\mathsf{P}(\mathsf{P}(\mathsf{P}(u \bmod 256) + v) \bmod 256) + w) \bmod 256), \tag{8.76}$$

for the 2D and 3D case, respectively. Of course this can be extended to arbitrary dimensions. Perlin proposed an additional trick for calculating multi-dimensional hash functions by slightly modifying the expression in Eqn. (8.75) to

$$\mathsf{P}(\underbrace{\mathsf{P}(\underbrace{u \bmod 256}_{\in[0,255]}) + \underbrace{(v \bmod 256)}_{\in[0,255]}}_{\in[0,510]}), \tag{8.77}$$

which does not produce the *same* but similarly "random" results. Obviously, the maximum possible table index in the above calculation is 510. Perlin made use of this fact by concatenating P with itself and thereby doubling the size of the permutation map to

$$\begin{aligned} \mathsf{P}' = \mathsf{P} \frown \mathsf{P} &= (\mathbf{151}, 160, \ldots, 156, 180) \frown (\mathbf{151}, 160, \ldots, 156, 180) \\ &= (\mathbf{151}, 160, \ldots, 156, 180, \mathbf{151}, 160, \ldots, 156, 180). \end{aligned} \tag{8.78}$$

[12] The method was supposed to be executed on limited graphics hardware.

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

Program 8.1 Setting up the extended permutation table P' of length 512 [10].

With the extended permutation table P' (of size 512) the expression in Eqn. (8.76) (for three dimensions) can simply be calculated in the form

$$\mathrm{h}_8(u, v, w) = \mathsf{P}'(\mathsf{P}'(\mathsf{P}'(u') + v') + w'), \tag{8.79}$$

where $u' = (u \bmod 256)$, $v' = (v \bmod 256)$ and $w' = (w \bmod 256)$. Analogously, for an N-dimensional lattice point $\boldsymbol{p} = (p_0, p_1, \ldots, p_{N-1}) \in \mathbb{Z}^N$, this is

$$\mathrm{h}_8(\boldsymbol{p}) = \mathsf{P}'(\ldots(\mathsf{P}'(\mathsf{P}'(p'_0) + p'_1) + p'_2) + \ldots + p'_{N-1}), \tag{8.80}$$

with $p'_i = (p_i \bmod 256)$. The following Java method shows an implementation of the function $\mathrm{h}_8(u, v, w)$:

```
int h8 (int u, int v, int w) {
  u = u & 0xFF;
  v = v & 0xFF;
  w = w & 0xFF;
  return P[P[P[w] + v] + u];
}
```

The extended permutation table $\mathtt{P} \equiv \mathsf{P}'$ is defined (similar to Perlin's reference implementation)[13] by the static code segment listed in Prog. 8.1. The method `h8()` in the above code segment returns a single `int` value in the range $[0, 255]$, which can be easily converted to a normalized floating-point value in $[0, 1]$ (see Eqn. (8.74))

For multi-dimensional noise functions, the required component functions $\mathrm{hash}_x()$, $\mathrm{hash}_y()$ etc. (see Eqn. (8.71)) should be mutually uncorrelated. For example, a simple approach is to use the hash function $\mathrm{h}_8(u, v)$ above and assign a particular group of bits of its 8-bit result to each dimension, for example, in the 2D case,

$$\mathrm{hash}(u, v) = \begin{pmatrix} \mathrm{hash}_x(u, v) \\ \mathrm{hash}_y(u, v) \end{pmatrix} = \frac{1}{64} \cdot \begin{pmatrix} \mathrm{h8}(u, v) \bmod 64 \\ (\mathrm{h8}(u, v) \div 4) \bmod 64 \end{pmatrix}. \tag{8.81}$$

In this case, the hash value for the x- and y-direction is based on the lower and upper 6 bits, respectively, of the 8-bit integer value returned by $\mathrm{h8}(u, v)$. The corresponding Java implementation could look like this:

```
double[] hash(int u, int v) {
  final int M = 0x3F;
  int h = h8(u, v);           // h ∈ [0,255] has 8 bits
  double hx = h & M;          // use bits 0...5 for hash_x(u,v)
  double hy = (h >> 2) & M;   // use bits 2...7 for hash_y(u,v)
  return new double[] {hx/M, hy/M};
}
```

This would produce 64 distinct and uniformly distributed gradient values for each dimension. An obvious alternative is to work with two non-overlapping blocks of 4 bits (corresponding to 16 distinct gradient values) for each dimension, which would still be sufficient. The following example shows a 3D hash function using one of three overlapping blocks of 4 bits for each dimension:

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

```
double[] hash(int u, int v, int w) {
  final int M = 0x0F;
  int h = h8(u, v, w);          // h ∈ [0,255] has 8 bits
  double hx = h & M;            // use bits 0...3 for hash_x(u,v,w)
  double hy = ((h >> 2) & M);   // use bits 2...5 for hash_y(u,v,w)
  double hz = ((h >> 4) & M);   // use bits 4...7 for hash_z(u,v,w)
  return new double[] {hx/M, hy/M, hz/M};
}
```

This method returns a `double` array with three elements in $[0, 1]$ for each discrete hash coordinate $(u, v, w) \in \mathbb{Z}^3$. Each component of the returned vector originates from a 4-bit "random" integer and can only take one of 16 distinct values. Although this appears quite primitive compared to the following methods, this is sufficient to produce visually appealing results. In fact, the required degree of randomness of the gradient values appears to quite low in Perlin's method.[14] Figure 8.15 shows 2D images created with Perlin's permutation method using different numbers of bits assigned to the components of the gradient vectors.

8.4.2 Integer hashing

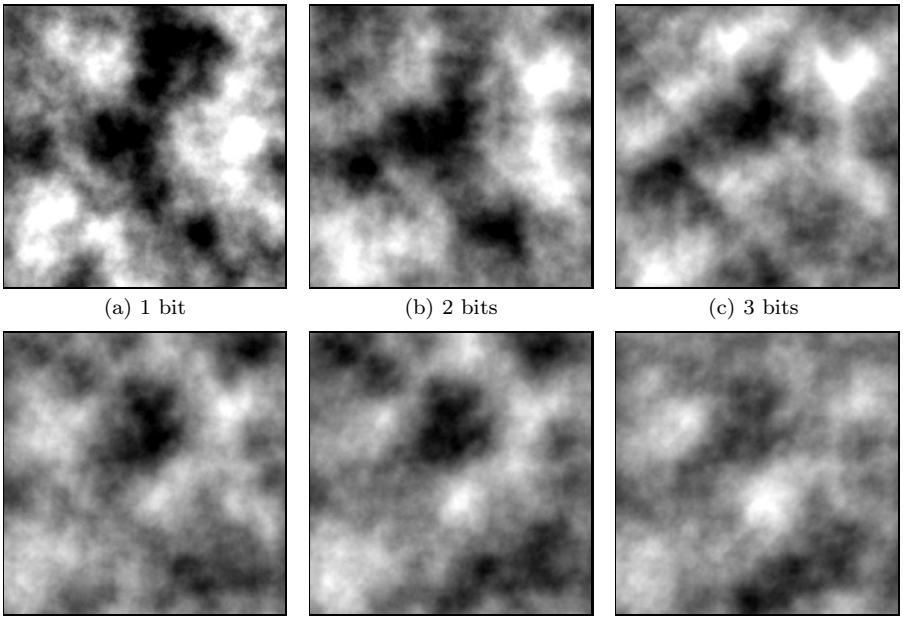
The disadvantage of the 8-bit permutation method described in Sec. 8.4.1 is the short period of the generated random sequences, i. e., images generated in this way will show repetitive patterns that may destroy the illusion of randomness. A common solution is to apply hash functions based on integer numbers with at least 32 bits, as used in cryptography, where the stochastic qualities of random sequences are much more important. A good overview on this topic and a description of current techniques can be found in [11, Sec. 7.1.4], for example. Most modern high-quality hash methods are based on 64 bit integer numbers and typically combine different hash functions.

For generating Perlin noise, however, simpler methods based on 32 bit integers seem to be more than adequate. An integer hash function

$$h = \text{hashInt}(key) \tag{8.82}$$

returns a 32-bit integer value for any integer $key \in \mathbb{Z}_{32}$. Three such functions are listed in Prog. 8.2. Since they rely strongly on low-level bit operations, we only show their Java implementations without going through an algorithmic description. All three methods return a signed 32-bit integer value for any integer argument `key`.

[14] Perlin [9] actually takes the 8-bit result of the hash function modulo 12 and thus selects one of the 12 edges of the three-dimensional cube for specifying the orientation of the gradient plane. The visual effects of this coarse quantization are surprisingly moderate.



(a) 1 bit (b) 2 bits (c) 3 bits

(d) 4 bits (e) 5 bits (f) 6 bits

Figure 8.15 Random noise images crated with the permutation lookup method using different numbers of bits for the gradient components. With 1 bit per component (a), gradient component values are limited to only -1 or $+1$. With 6 bits per component (f), 64 different random values are generated that are uniformly distributed in $[-1, 1]$. All other parameters are identical.

Multi-dimensional applications

To generate a useful hash value for an N-dimensional lattice point, all its coordinates must be considered. With permutation hashing, this was accomplished by repeated application of the permutation table to the individual coordinates (see Eqn. (8.79)). This could be done with integer hashing in a similar fashion. An alternative is to first combine ("prehash") the individual lattice coordinates into a single integer value and then apply the hash function only once. For example, for three-dimensional coordinates (u, v, w) the prehash value could be calculated in the form

$$key = (n_u \cdot u + n_v \cdot v + n_w \cdot w), \tag{8.83}$$

where n_u, n_v, n_w are selected (typically small) prime numbers. Of course this can be easily scaled to N-dimensional coordinates. To obtain individual and independent random values for the gradient components, we can again base the functions $\text{hash}_x()$, $\text{hash}_y()$, etc. (see Eqn. (8.71)) on predefined bit groups of the inter hash value. Since integer hash values have 32 bits (instead of 8 in

```
 1 int hashIntWard(int key) {
 2   key = (key << 13) ^ key;
 3   key = (key * (key * key * 15731 + 789221) + 1376312589);
 4   return key;
 5 }
 6
 7 int hashIntShift(int key) {
 8   key = ~key + (key << 15);
 9   key = key ^ (key >>> 12);
10   key = key + (key << 2);
11   key = key ^ (key >>> 4);
12   key = key * 2057;
13   key = key ^ (key >>> 16);
14   return key;
15 }
16
17 int hashIntShiftMult(int key) {
18   int c2 = 668265261; // a prime or an odd constant
19   key = (key ^ 61) ^ (key >>> 16);
20   key = key + (key << 3);
21   key = key ^ (key >>> 4);
22   key = key * c2;
23   key = key ^ (key >>> 15);
24   return key;
25 }
```

Program 8.2 Java implementations of various integer hash functions: `hashIntWard()` from [13], `hashIntShift()` and `hashIntShiftMult()` from [12]. They make use of the following Java bit operators: `^` (bit-wise exclusive-OR), `~` (bit-wise inverse), `<<` n (shift left by n bit positions and inserting zeros on the right), `>>>` n (shift right by n bit positions and inserting zeros on the right, ignoring the sign bit),

the permutation method), this should not pose a problem at all.[15] In the 3D case, for example, a suitable Java method could look like this:

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

Eight bits are used for each component value, i. e., there are 256 distinct gradient values, which is usually sufficient.[16] In this case, the integer hash function

[15] It is a feature of a good integer hash function that any subset of bits in its results is again randomly distributed.

[16] Of course one could extract up to 10 bits in 3D for each dimension or up to 15 bits in 2D. In the unlikely case that this is not sufficient, a 64-bit hash function could be used.

is only invoked once for every lattice point (u, v, w).[17]

Alternatively, one could calculate each gradient component by a separate call to the hash function, as in the following example [13]:

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

Instead of `hashInt()`, any of the specific methods listed in Prog. 8.2 or any other integer hash function can be used. Of course, his approach can be easily extended to N dimensions.

8.4.3 "Seed" for additional randomness

All hash functions described Sec. 8.4 are naturally deterministic in the sense that the always return the same hash value for a given lattice point; after all, this is an important requirement for a hash function. Noise functions generated with a particular hash function thus always look the same. To generate differently looking 2D images or 3D volume data with the same hash function, one could choose different starting coordinates, for example. Alternatively, one could parameterize the hash functions using different seed values S, as is common practice in random number generation. For example, the seed parameter S could simply be used as an offset to the integer hash argument in the form

$$h = \text{hashInt}(key + S), \tag{8.84}$$

for some $S \in \mathbb{Z}$. Figure 8.16 shows 2D noise images calculated with the three different integer hash functions listed in Prog. 8.2 and seed values $S = 0, 1, 2$. Of course, the seed value itself could be generated with a standard random number generator or from clock data to achieve "ultimate" randomness.

8.5 Implementation and further reading

This chapter is intended as an introduction to gradient noise with a particular focus on Perlin's classical method, which is not easily accessible from the original material, despite its high popularity. The chapter should also serve as a tutorial to help understand the purpose of the various steps involved and their geometric interpretation. Most of the described facts and procedures can

[17] Note that the `hashIntWard()` function (Prog. 8.2) does not perform well in this situation, since the lower 16 bits of its result are strongly correlated and may thus produce repetitive patterns in the lower channels.

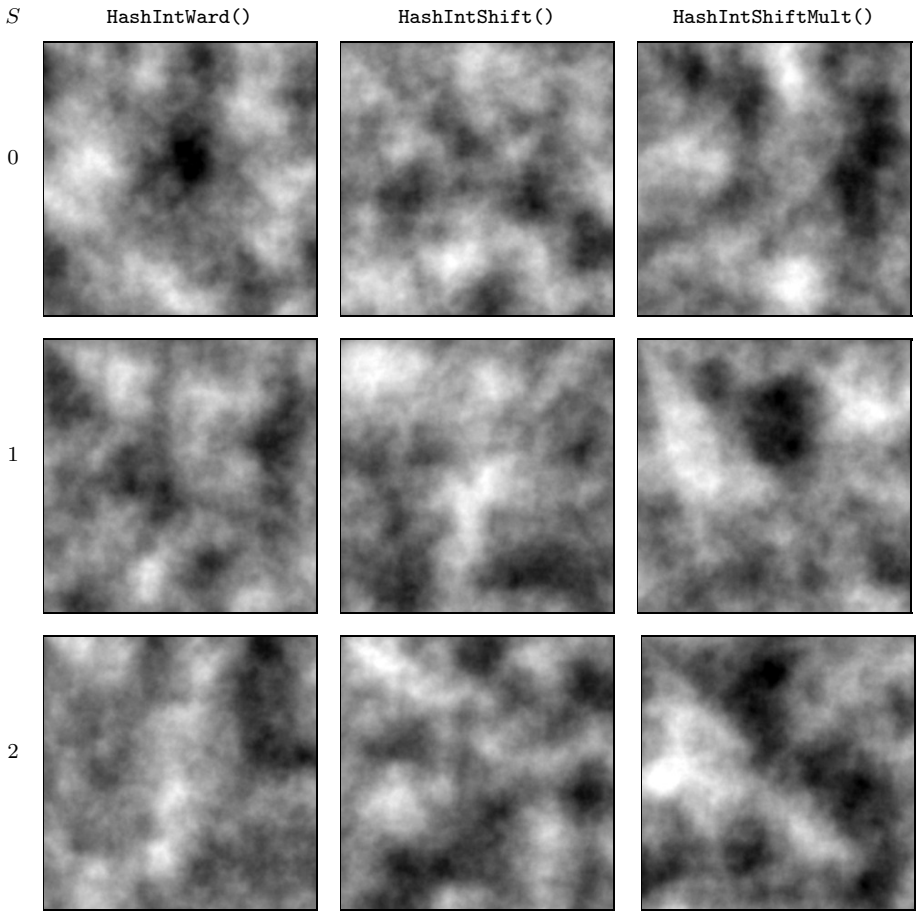


Figure 8.16 Gradient noise images generated with different integer hash functions listed in Prog. 8.2, for varying seed values $S = 0, 1, 2$. All other noise parameters are fixed: $p = 0.5$, $f_{\min} = 0.01$, $f_{\max} = 0.5$ (6 octaves with frequencies $f_i = 0.01$, 0.02, 0.04, 0.08, 0.16, 0.32 and amplitudes $a_i = 1.0$, 0.5, 0.25, 0.125, 0.0625, 0.03125).

either be found in the referenced literature or can be derived independently with little effort. All the Java code listed in this chapter and additional examples are available in the source code section of the book's supporting web site, including the ImageJ plugin listed in Prog. 8.3 which shows an example for the use of the associated classes[18] and methods.

Today, Perlin-type gradient noise is likely the most popular method for 2D and 3D texture synthesis and sometimes referred to as a "workhorse" for the graphics industry. Nevertheless, many other methods have been developed for

[18] Contained in the packages `imagingbook.noise` and `imagingbook.hashing`.

```
1  import noise.PerlinNoiseGen2d;
2  import hashing.*;
3  import ij.ImagePlus;
4  import ij.plugin.PlugIn;
5  import ij.process.FloatProcessor;
6  import ij.process.ImageProcessor;
7
8  public class Demo_Perlin_2d implements PlugIn {
9    int w = 300;       // image width
10   int h = 300;       // image height
11   int K = 3;         // number of noise frequencies
12   double f_min = 0.01;    // min. frequency (in cycles per pixel unit)
13   double f_max = f_min * Math.pow(2, K);
14   double persistence = 0.5; // persistence
15   int seed = 2;      // hash seed
16   HashFun hf = new HashPermute(seed); // chose a hash function
17
18   public void run(String arg0) {
19
20     // create the noise generator:
21     PerlinNoiseGen2d ng =
22          new PerlinNoiseGen2d(f_min, f_max, persistence, hf);
23
24     // create a new image and fill with noise:
25     ImageProcessor fp = new FloatProcessor(w, h);
26     for (int v = 0; v < h; v++) {
27       for (int u = 0; u < w; u++) {
28         double val = ng.NOISE(u, v);
29         fp.putPixelValue(u, v, val);
30       }
31     }
32
33     // display the new image:
34     (new ImagePlus("Perlin Noise Image", fp)).show();
35   }
36 }
```

Program 8.3 Example ImageJ plugin for creating multi-frequency Perlin noise images. The number of frequency components (K) and the base frequency ($f_{\min}$) are specified in lines 11 and 12, respectively. The persistence value (ϕ) is set in line 14. The type of hash function is selected in line 16 (other options are `Hash32Ward`, `Hash32Shift`, and `Hash32ShiftMult`) with an arbitrary seed value.

similar purposes or other applications, such as wavelet noise, convolution noise or anisotropic noise. Interested readers will find an excellent summary and many relevant references in [8].

Bibliography

[1] P. Bourke. Perlin noise and turbulence. Technical Report, University of Western Australia (January 2000).

[2] W. Burger and M. J. Burge. "Digital Image Processing—An Algorithmic Introduction using Java". Texts in Computer Science. Springer, New York (2008).

[3] W. Burger and M. J. Burge. "Principles of Digital Image Processing – Fundamental Techniques (Vol. 1)". Undergraduate Topics in Computer Science. Springer, New York (2009).

[4] W. Burger and M. J. Burge. "Principles of Digital Image Processing – Core Algorithms (Vol. 2)". Undergraduate Topics in Computer Science. Springer, London (2009).

[5] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. "Texturing & Modeling: A Procedural Approach". Morgan Kaufmann Publishers, San Francisco, 3 ed. (2002).

[6] S. Gustavson. Simplex noise demystified. Technical Report, Linköping University (2005).

[7] A. Lagae and P. Dutré. Long period hash functions for procedural texturing. In L. Kobbelt, T. Kuhlen, T. Aach, and R. Westermann, editors, "Vision, Modeling, and Visualization 2006", pp. 225–228, Aachen, Germany (2006). IOS Press.

[8] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. Ebert, J. Lewis, K. Perlin, and M. Zwicker. A survey of proce-

dural noise functions. *Computer Graphics Forum* **29**(8), 2579–2600 (December 2010).

[9] K. Perlin. An image synthesizer. *SIGGRAPH Computer Graphics* **19**(3), 287–296 (1985).

[10] K. Perlin. Improving noise. In "SIGGRAPH'02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques", pp. 681–682, San Antonio, Texas (2002).

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. "Numerical Recipes". Cambridge University Press, third ed. (2007).

[12] T. Wang. "Integer Hash Function" (March 2007). Version 3.1, http://www.concentric.net/~Ttwang/tech/inthash.htm.

[13] G. Ward. A recursive implementation of the perlin noise function. In J. Arvo, editor, "Graphics Gems II", ch. VIII.10, pp. 396–401. Academic Press (1991).