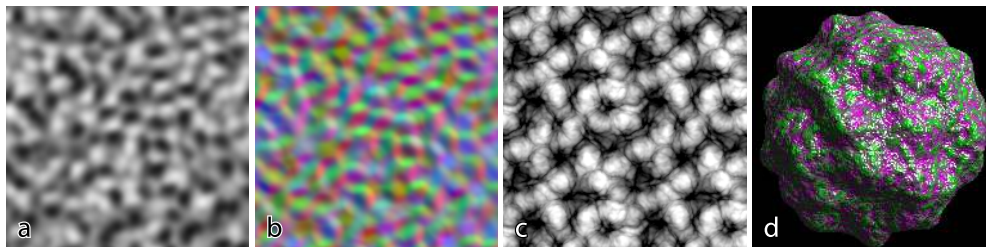# Tiling Simplex Noise and Flow Noise
in Two and Three Dimensions

Stefan Gustavson   Ian McEwan
Linköping University

Figure 1. (a) A 2D slice of 3D tiling simplex noise, (b) its analytical gradient visualized as an RGB image, (c) an animated tiling flow noise fractal sum using gradient advection, and (d) a 3D mapped sphere using analytic normals for animated flow noise displacements and bumps. The supplementary material includes many additional animated WebGL examples.

Abstract

Simplex noise is a highly useful procedural primitive, but it is defined in weirdly oriented grids that do not tile easily. We set that straight, literally, by publishing implementations of 2D and 3D tiling simplex noise in GLSL. As an added bonus, our tiling noise functions support the animation technique called *flow noise*, which we extend further to 3D. At a small additional cost, these functions can compute their exact analytical gradient, and second-order derivatives as well if desired, making them suitable for gradient-dependent applications like bump mapping, analytical antialiasing and particle animations, e.g. with *curl noise*.

1. Introduction

Simplex noise is a procedural texturing primitive, a lattice gradient noise, introduced by Perlin [2002] as an improved alternative to his classic *Perlin noise* [1985]. This article assumes that the reader is at least somewhat familiar with those algorithms. For a summary, we recommend the explanation by Gustavson [2005] rather than the original articles, which can be a challenge to follow.

Here, we present a 3D noise variant using a *3D simplex grid*, which tiles over any integer-sized period in the Cartesian coordinates (x, y, z), as well as a modification of 2D noise to a *2D simplex grid*, which tiles over any integer period in x and even-sized integer periods in y. The tiling property, which comes at a small computational cost, makes these noise variants eligible for use also as precomputed, stored textures for greater flexibility, scalability, and portability of noise-based procedural methods.

The concept of *flow noise*, invented by Perlin and Neyret [2001], uses a rotation of the noise function's generating gradients, along with a texture domain warp along the gradient of the noise field, to create a "swirling" and "billowing" look for animations. The original version is restricted to 2D, but we publish an extension to *3D flow noise*, useful for creating the appearance of turbulent flow for object-space texture mapping and volumetric effects. Additionally, we publish a re-implementation of *2D flow noise* based on simplex noise. Both are defined on our modified, tiling grids.

We publish efficient, self-contained *GLSL implementations* of these new periodic simplex noise functions with rotating gradients in 2D and 3D, using some implementation details from previous joint work [McEwan et al. 2012]. The functions require no setup, are compatible with WebGL 1.0, and come with a permissive MIT open source license. Their exact *analytic gradients* can be computed with far less computations than finite difference approximations. Optional additional code computes second-order derivatives as well.

This article is aimed at potential adopters and users of these functions. A more thorough exposition with implementation details that would be of interest primarily to fellow researchers and developers is in the supplementary material. This shorter article describes *what* we did, but not always exactly *how*.

2.   Algorithm Overview

Compared to other simplex noise algorithms, our algorithm for simplex noise has several additional steps. Even though we took care to format and comment the GLSL source code properly, source code is never an easy read, and without experience with similar noise implementations, it would be particularly challenging. Table 1 presents a verbal description of what operations are performed, and in what order. The source code listing in Section 12 refers to the numbered steps in this table.

The steps marked by an asterisk in the table are optional: Steps 5 to 6 are omitted if a specific tiling period is not required. Steps 9a or 9b are omitted if animated, rotating gradients in the style of flow noise are not needed. Step 14 is performed only if the analytic gradient is needed, and Step 15 only if second-order derivatives are desired. These optional steps are the main contributions of our work. Compared to previously published algorithms, we have also made improvements and speed-ups to Steps 2, 3, 7, 8, 9, and 10.

|		
1.	Input point in texture space (2D or 3D) is $\vec{x}$.	
2.	Transform to simplex space, determine in which simplex $\vec{x}$ is.	
3.	Transform each corner of that simplex back to texture space.	
4.	Compute vectors from each corner to $\vec{x}$.	
*5.	Wrap corners to the desired period along each axis.	
*6.	Transform the wrapped corners to simplex space again.	
7.	Compute a hash k for each corner from its simplex coordinates.	
8.	Generate pseudo-random gradients $\hat{g}_i$ from the hash k.	
*9a.	For 2D: Rotate the gradients in the plane by a varying angle α.	
*9b.	For 3D: Rotate gradients around a pseudo-random axis by α.	
10.	Compute radial falloffs w_i based on distances from $\vec{x}$ to corners.	
11.	Make a linear ramp along the gradient from each simplex corner.	
12.	Multiply the ramp values at **x** with their corresponding falloff.	
13.	The final noise value n is the sum of those multiplications.	
*14.	Compute ∇n, the exact partial derivatives of n.	
*15.	Compute second-order partial derivatives of n as well.	

Table 1. A step-by-step overview of our 2D and 3D noise algorithms.

3. Tiling Simplex Grids in 2D and 3D

The tiling properties of these functions stem from slight changes to the simplex grids. In 2D, the optimal tiling with equilateral triangles was rotated and stretched somewhat to make a staggered lattice, where every second row of vertices hits integer coordinates in the (x, y) grid; see Figure 2. This grid tiles over any integer period in x and any *even integer* period in y, and the slightly wider spacing of vertices along the y direction is not visually noticeable.

Regarding 3D tiling, there seems to be a common misconception that the optimal simplex grid in 3D has nonrational angles and is therefore unfit for axis-aligned tiling. This is most likely due to the unconventional transformation used by Perlin in his original implementation when transforming between (x, y, z) texture space and (u, v, w) simplex space, which maps to a rotated simplex grid. However, in its most straightforward orientation, the simplex grid can be conveniently mapped to a cubical grid that tiles with any integer period in (x, y, z).

The simplex grid in its axis-aligned orientation is shown in Figure 3. A transformation matrix and its inverse are given for reference, where M is the matrix for transforming a column vector from (x, y, z) coordinates to skewed (u, v, w) coordinates in the realigned simplex grid, and M^{-1} is its inverse. Note that the extraordinarily simple form of M enables an optimized implementation to use fewer operations than a full matrix multiplication, as $(u, v, w) = (y + z, x + z, x + y)$ requires only additions. The inverse has multiplications, but only with the binary-friendly factor $\frac{1}{2}$.

$$(u, v) = (x + \tfrac{1}{2}y, y)$$

$$(x, y) = (u - \tfrac{1}{2}v, v)$$
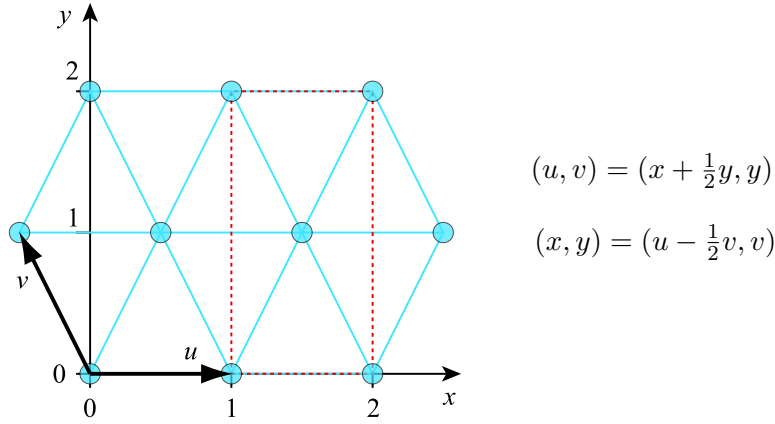
Figure 2. The modified 2D simplex grid and its corresponding coordinate transformations. The slightly stretched hexagonal grid (blue circles and lines) tiles over any $N \times 2M$ sized rectangle, where N and M are integers. The dotted red lines show the translational rectangle of this tiling.
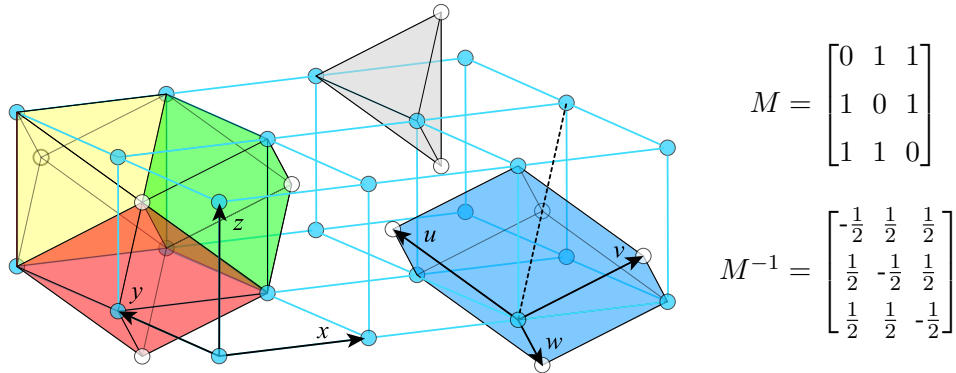
The transformation used by Perlin aims to save on multiplications, but it is not a perfect match for a GPU implementation, because it performs operations that are not as readily mapped to the accelerated multiply-and-add functions in a modern GPU. Furthermore, it has factors $\frac{1}{3}$ and $\frac{1}{6}$ that require full multiplications and cause numerical errors. Testing on an NVIDIA GTX 1660 GPU, Perlin's optimized transformations were somewhat faster than the equivalent 3×3 matrix multiplications, but slower than transforming to and from the axis-aligned simplex grid using the matrices M and M^{-1} in Figure 3.

More details on the 3D grid are in the supplementary material, but it is worth pointing out the nonobvious cluster of six tetrahedra (the parallelepiped to the right in Figure 3) that can be used to construct the tiling rather than the more obvious octahedral clusters of four tetrahedra. This construction makes it algorithmically simple to determine which simplex contains a certain point. Perlin's original formulation of simplex noise uses this method, although in its differently oriented grid.

It should be noted that Perlin's rotated 3D grid still has rational components for its principal vectors and *does* indeed tile, with an axis-aligned translational cube of size $3 \times 3 \times 3$. We provide an option in our code to use Perlin's orientation of the simplex grid instead of the axis-aligned version. The tiling periods along either dimension then become somewhat inconveniently restricted to multiples of 3.

4. Pseudo-random Hash Function

Borrowing a useful idea from our own previous work [McEwan et al. 2012], we use a permutation polynomial to generate pseudo-random hashes from the integer coor-

$$M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

Figure 3. Left: The simplex grid can be constructed from groups of four tetrahedra forming octahedra in three different orientations (yellow, green, and red), which together tile 3D space. Top: A single simplex tetrahedron (gray), with two corners vertex-centered on the (x, y, z) cubical lattice (blue circles and lines) and two body-centered (white circles). Right: Six tetrahedra sharing an edge along a grid diagonal (dotted line) form a parallelepiped (blue), which tiles 3D space with translations alone. Regarding the matrices, see the text.

dinates of the simplex grid points. There are many other options for hash functions, and permutation polynomials are generally an inferior choice in terms of statistical quality. However, the method is computationally simple and floating-point compatible, and by taking care to pick a suitable polynomial, we can achieve a satisfactory apparent randomness for this particular application.

Our permutation field has size $17^2 = 289$ for the same reason as in the previous work: it is reasonably large, while also keeping the integers in all intermediate results small enough to be implemented in single-precision floating point arithmetic without truncation errors. However, our choice of permutation polynomial is different from our previously published noise functions, because we have since found that the old choice $34x^2 + x$ caused visible structures in the noise pattern in the form of frequent diagonal streaks. Our new choice is $34x^2 + 10x$, which does not cause any such pattern defects. For more details on this, see the supplementary material.

5. Rotating Gradients in 2D and 3D

Flow noise [Perlin and Neyret 2001] performs an animation of the noise pattern by allowing its underlying generating gradients at each grid point to rotate dynamically. The original implementation was restricted to 2D, but here we extend it to 3D in a nontrivial manner.

In order to make 3D flow noise, we assign an individual, pseudo-random rotation axis to each gradient, orthogonal to it. All rotations are still performed with the same angle, as the function is supposed to be used in sums over different scales to

mimic turbulent flow, where swirls of similar size have similar velocity. The rotation is implemented in code by generating *two* orthonormal vectors for each vertex and summing them with sine and cosine weights to combine them into one gradient rotating in their common plane. We generate both of these vectors from the same hash in a nonobvious but efficient manner, making use of a Fibonacci spiral to generate equal-area distributions of unit vectors on a sphere. We designed and implemented two algorithms with slightly different properties, both of which are described in detail in the supplementary material.

In 2D space, the generating gradients are all rotated in the same direction in the plane, at the same angular speed, as originally suggested by Perlin and Neyret [2001]. Our implementation of the pseudo-random gradient selection for this 2D simplex noise enables that rotation to be performed by a single addition.

Our gradient generation for both 2D and 3D noise makes use of the highly accelerated $\sin$ and $\cos$ functions in modern GPUs. Previously published versions of noise, including our own work, have made a point to avoid trigonometric functions for performance reasons, but as GPUs have evolved, they are now the better option.

6. Analytic Derivatives

Many uses of procedural noise require computing its derivatives, among them bump and displacement mapping, analytical antialiasing, and particle animation using *curl noise* [Bridson et al. 2007]. Such derivatives can be computed with good accuracy, but at a high computational cost, by finite difference approximations. In our case, an analytical approach proves to be a lot more efficient.

Simplex noise is a sum of polynomials. Differentiation is distributive with respect to addition: the derivative of a sum is the sum of the derivatives of the individual terms. Thus, the equations for the derivatives of simplex noise have the same summation structure as the noise itself. Taking proper care to reuse intermediate results, the computation of analytic derivatives comes at only a small extra cost. This is a strong advantage of simplex noise compared to classic Perlin noise.

In the equations in Table 2, $\vec{x}_i$ are the vectors from each of the influencing simplex vertices to the point being evaluated, $\hat{g}_i$ are the generating gradients for each vertex, n is the noise value, and ∇n is the gradient of the noise field. For the 2D case the components of $\vec{x}_i$ are (x_i, y_i), and for the 3D case they are (x_i, y_i, z_i). Accordingly, the components of $\hat{g}_i$ are either $(g_{x,i}, g_{y,i})$ or $(g_{x,i}, g_{y,i}, g_{z,i})$. The sum over i has one term for each vertex of the simplex: three for 2D noise, and four for 3D.

Second-order derivatives can be computed reasonably cheaply as well. They are useful for noise-based particle animation, although they are of much less utility than the universally handy gradient. The second-order derivatives are covered in the supplementary material, as well as variants of the GLSL functions to compute them.

2D	3D
$w_i = \frac{4}{5} - \|\vec{x}_i\|^2 = \frac{4}{5} - \vec{x}_i \bullet \vec{x}_i$	$w_i = \frac{1}{2} - \vec{x}_i \bullet \vec{x}_i$
$n = \sum_i w_i^4(\vec{x}_i \bullet \hat{g}_i)$	$n = \sum_i w_i^3(\vec{x}_i \bullet \hat{g}_i)$
$\nabla n = \sum_i \left(w_i^4 \hat{g}_i - 8w_i^3(\vec{x}_i \bullet \hat{g}_i)\vec{x}_i \right)$	$\nabla n = \sum_i \left(w_i^3 \hat{g}_i - 6w_i^2(\vec{x}_i \bullet \hat{g}_i)\vec{x}_i \right)$

Table 2. Equations for noise and its derivatives in 2D and 3D.

7. Removing Discontinuities

Simplex noise aficionados might notice a discrepancy between the expression for the 3D noise value as presented in the previous section and the original reference implementation in Java by Perlin [Perlin 2002]. He uses a polynomial of $r^2 = x^2 + y^2 + z^2$ for the spherical decay of the wiggle, and his exact choice for the polynomial is $(0.6 - r^2)^4$, which would translate to $w_i = 0.6 - \|\vec{x}_i\|^2$ in the notation used here. Contrary to Perlin's claim, this creates a too large region of influence around each vertex, and the noise and its gradient have discontinuities at simplex boundaries. This has been pointed out previously by other authors [Sharpe 2012], and related changes have made their way into some implementations, but given the relative obscurity and informal nature of the references, it deserves mention here. Changing 0.6 to 0.5 reduces the region of influence to its appropriate size, and in order to keep the general visual character of the noise field, we reduce the order of the polynomial and use $(0.5 - r^2)^3$ to make it decay in almost the same fashion as Perlin's function; see Figure 4.
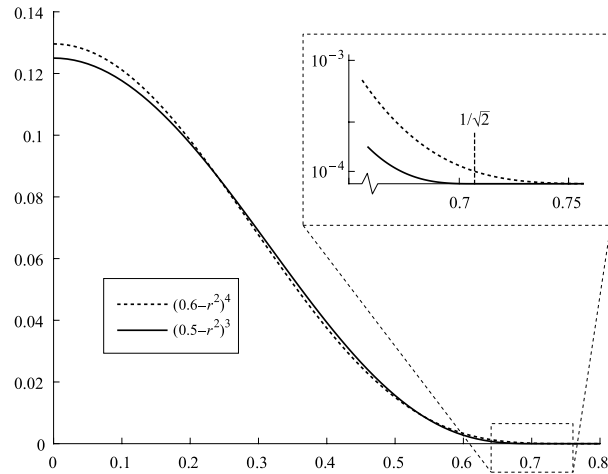
Figure 4. Perlin's original decay $(0.6 - r^2)^4$, having a slightly too large region of influence, as shown in the detail view at the top right, and the modified decay $(0.5 - r^2)^3$, giving a similar result but avoiding discontinuities at simplex boundaries.

8. Function Declarations

The source code for the GLSL functions listed here is provided in the supplementary material, and also in the online repository on https://github.com/stegu/psrdnoise/. The function declarations are as follows:

```
float psrdnoise(vec2 x, vec2 period, float alpha, out vec2 gradient)
float psrdnoise(vec3 x, vec3 period, float alpha, out vec3 gradient)
```

These two overloaded functions return 2D or 3D noise, depending on the type of the arguments, for the point `v`, repeating with periods as specified in `period` and rotating the generating gradients by angle `alpha` (specified in radians). The gradient of the noise value is returned in `gradient`. For the 2D and 3D versions alike, the range of values for the noise is scaled to cover the range $[-1, 1]$ well without clipping.

Functions that compute analytic second-order derivatives require extra output arguments. These are described in the supplementary material.

9. Performance and Speed-ups

Despite the considerable amount of computations in these functions, particularly for the 3D version, they perform well on modern hardware, as shown in Table 3. The execution times for the functions were measured as the difference between an almost zero effort shader that writes a constant color and a shader making several calls to the noise function, to tax even the high-performance GPUs. This was done to remove the influence of the overhead from the clearing, writing, and copying of render buffers. This is an indirect way to measure the execution time for a function, and the exact performance of a certain GPU depends heavily on many details like memory speed and clock frequency. For these reasons, the benchmarks in the table give a general indication of speed, but they are approximate measures.

It is not an easy task to assess the relative performance of computational noise versus traditional texture mapping, because many outside factors come into play. Apart from the make and model of the GPU, and even the exact version of the driver, one must consider how much the ALU is utilized by other parts of the current shader, and how much texture memory bandwidth is required for other purposes. As a synthetic but hopefully useful example, we can report that computing a single component of the 2D procedural noise that we publish here is comparable in speed to making a single 2D texture lookup, even slightly outperforming it on some higher-end GPU models. Considering that a modern GPU is heavily optimized to make efficient texture lookups, we think this indicates that noise-based procedural methods are generally useful in shader programming, now as well as in the near future.

Periodic tiling, rotating gradients, and analytic derivatives all come at an additional computational cost. If the derivative values are statically not used in computing

GPU	2D `psrdnoise`		3D `psrdnoise`	
	M values/s	ns/value	M values/s	ns/value
NVIDIA RTX 3080 (desktop)	57300	0.0174	21100	0.0474
NVIDIA GTX 1080 Ti (desktop)	34000	0.0294	13100	0.0763
NVIDIA GTX 1660 (desktop)	18100	0.0552	6680	0.1500
NVIDIA GF 940MX (old laptop)	3710	0.270	1230	0.813
AMD Vega 10 (laptop)	2720	0.368	1080	0.926
AMD Vega 8 (laptop)	2480	0.403	949	1.050
Intel UHD 650 (laptop)	1360	0.698	516	1.840
Intel HD 520 (old laptop)	943	1.010	334	2.840

Table 3. Execution speed for the new noise functions, measured as millions of evaluations of the function per second (frames/second × pixels/frame × number of calls to `psrdnoise` in the shader), and as nanoseconds per evaluation. The selection of GPUs is simply what we had available for testing. (Yes, that's *57 billion* evaluations of noise per second at the top.)

the shader output, the corresponding dead code is removed by modern shader compilers, and the function executes faster. If all periods are set to 0.0 or a negative value, the tiling wraparounds are skipped, and if the gradient rotation angle is set to exactly 0.0 for the 3D noise, a much simpler and faster method is used to compute the static gradients. Modern GPUs can detect if a conditional statement takes the same branch across all parallel cores in one execution unit, and then executes only that branch for all cores. As a consequence, these noise functions can be sped up both at compile time and at runtime by removing or bypassing code for computations that are not needed, according to Figure 5.
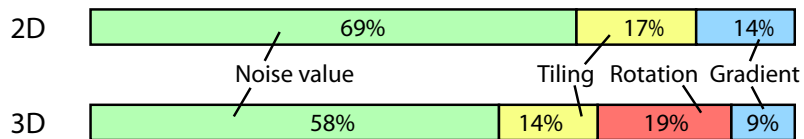
Figure 5. Not using the extra features of the functions speeds up their execution accordingly, e.g. not tiling to a period for 2D noise cuts 17% off its execution time. The scales are relative – 3D noise takes about three times longer to compute than 2D noise.

Notable in Figure 5 is that the animated gradient rotation comes for free in 2D due to how we implemented it, and rotations are reasonably cheap even in 3D due to our efficient algorithm. It also deserves special mention that the gradient is computed at a *remarkably* low cost compared to several repeated evaluations of the entire function for a finite difference approximation.

If neither tiling, gradient rotations, nor analytical gradients are used, these functions are similar to previously published simplex noise functions. However, the 3D function in particular is faster than many other implementations, including the likewise purely computational one we published previously [McEwan et al. 2012], and our 2D and 3D functions alike yield procedural noise with considerably fewer artifacts and better consistency across platforms than some of the noise functions in current circulation among experimenting shader programmers.

The second-order derivatives were not included in the profiling in Figure 5, because we imagine that they would be of less general interest. Computing second derivatives in addition to first derivatives increases the execution time by about 18% for 2D and 20% for 3D.

On a final note concerning performance, we implemented two algorithms for gradient rotation. Both are included in the supplementary material, and there is an option in the 3D noise function to use a faster gradient generation algorithm. This is controlled by a `#define` statement. If rotating gradients are desired, using the alternative algorithm speeds up the function as a whole by 10–15%, depending on GPU model, meaning that the faster rotations take only around 10% of the total execution time rather than 19% as indicated in Figure 5. However, in return for its efficiency, this algorithm has no dynamic shortcut when the rotation angle is exactly zero. Therefore, when animated gradient rotations are not needed, this faster algorithm is instead *slower* and should not be used. For that reason, we did not make it the default.

10. Flaws and Remedies

The 3D simplex grid used in the tiling noise is the same as in Perlin's original formulation, only with a different orientation. One of Perlin's stated advantages of simplex noise over classic noise is that it lacks visible artifacts from the underlying grid, but it does in fact have such artifacts in certain planes—they are just not aligned with the (x, y, z) coordinate grid. Because we deliberately orient the simplex grid to tile nicely, these artifacts now occur in axis-aligned planes with constant x, y, or z. In a manner similar to classic Perlin noise, the artifacts cause the noise to change in character as the cut plane moves from passing through grid vertices to passing between them, with a period of $\frac{1}{2}$ units, as can be seen in Figure 6. Unfortunately, the "Hello World!" use case of 3D noise, commonly used for quick testing and demonstration purposes, is to generate a noise pattern in the (x, y) plane with z held constant or varying slowly over time. The 3D noise function presented here looks its *absolute worst* in this exact scenario when the z coordinate hits multiples of 0.5.

We advise against using noise patterns from axis-aligned or nearly axis-aligned planar 2D slices of this 3D tiling simplex noise. This might sound quite restrictive, but it can be avoided. Even if the noise is precomputed and stored as a tiling 3D tex-
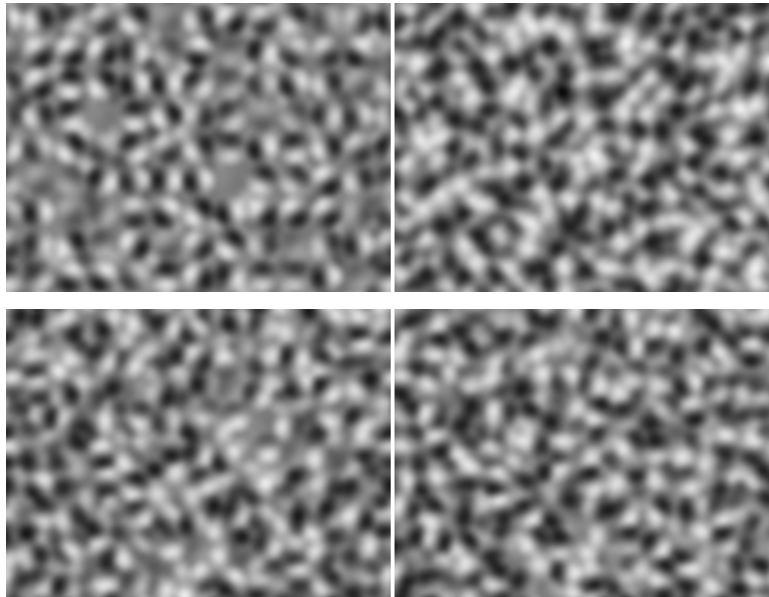
Figure 6. Using the new tiling simplex grid, 3D noise has grid artifacts in axis-aligned planes. Top: Noise with the new grid, in the (x, y) plane at $z = 0.0$ and $z = 0.25$, respectively. Bottom: Noise with Perlin's rotated grid, also at $z = 0.0$ and $z = 0.25$. This noise *has* the same ugly artifacts, but in off-axis planes. (The single plane of 2D noise does not have this problem.)

ture, it is a simple matter to add a skew or a rotation to the texture coordinates before lookup to avoid these artifacts on large planar surfaces. If this is still a problem, we provide an additional `#define` statement in the GLSL code to use Perlin's original grid instead of the "new" straightened grid. This has the side effect of requiring all periods to be multiples of 3, but tiling is still supported, and the grid artifacts appear in off-axis planes instead. The function becomes ever so slightly slower when the alternative grid is used, but not enough to make any real difference.

11. Slicing versus Gradient Rotations

Referring to the previous section, it can be argued that the kind of animated pattern that is commonly generated by making a 2D planar cut through 3D noise and moving it slowly in the third dimension constitutes an abuse of 3D noise. As pointed out by others [Cook and DeRose 2005], the Fourier projection-slice theorem tells us that taking a 2D slice of a band-limited 3D function introduces a substantial amount of low-frequency content in the pattern, which might cause problems. The common practice of *frequency clamping*, dropping higher-order terms from a fractal sum under the assumption that they contain only high-frequency content that would cause aliasing, is inherently wrong in such cases.

Creating an animated 2D noise field can be handled by rotating the gradients in our 2D function. Not only is this a lot cheaper than slicing a 3D function, it also preserves the band-pass character of the noise. For the same reason, a 3D "slice" of 4D noise, with the pattern being animated by moving the "slice" along the fourth dimension, introduces additional low-pass content in the pattern, whereas our 3D noise function can generate appropriately band-limited animated 3D noise by rotating the gradients, and it can do so at a lower computational cost. Neither our 2D nor our 3D noise changes its visual or statistical character with the rotations. Furthermore, animations made by rotating the gradients are periodic and therefore well suited for precomputed animation cycles.

12. Source Code

The following listings contain fully functional code, but some options have been removed for clarity, and the comments in the original code have been replaced with brief references to the steps listed in Table 1. The supplementary material lists a version with ample spacing and more verbose comments, and the stand-alone source files we provide as supplementary downloads are extensively commented.

2D Noise

```glsl
// psrdnoise2.glsl, version 2021-12-02
// Copyright (c) 2021 Stefan Gustavson and Ian McEwan
// (stefan.gustavson@liu.se, ijm567@gmail.com)
// Published under the MIT license, see:
// https://opensource.org/licenses/MIT

float psrdnoise(vec2 x, vec2 period, float alpha, out vec2 gradient)
{
// 2.  Transform input point to find simplex "base" i0.
  vec2 uv = vec2(x.x+x.y*0.5, x.y);
  vec2 i0 = floor(uv), f0 = fract(uv);
// 3.  Enumerate simplex corners and transform back.
  float cmp = step(f0.y, f0.x);
  vec2 o1 = vec2(cmp, 1.0-cmp);
  vec2 i1 = i0 + o1, i2 = i0 + 1.0;
  vec2 v0 = vec2(i0.x - i0.y*0.5, i0.y);
  vec2 v1 = vec2(v0.x + o1.x - o1.y*0.5, v0.y + o1.y);
  vec2 v2 = vec2(v0.x + 0.5, v0.y + 1.0);
// 4.  Compute distances to corners before we wrap them.
  vec2 x0 = x - v0, x1 = x - v1, x2 = x - v2;
  vec3 iu, iv, xw, yw;
// 5, 6.  wrap to period and adjust i0, i1, i2 accordingly.
  if(any(greaterThan(period, vec2(0.0)))) {
    xw = vec3(v0.x, v1.x, v2.x); yw = vec3(v0.y, v1.y, v2.y);
    if(period.x > 0.0)
```

```
        xw = mod(vec3(v0.x, v1.x, v2.x), period.x);
    if(period.y > 0.0)
        yw = mod(vec3(v0.y, v1.y, v2.y), period.y);
    iu = floor(xw + 0.5*yw + 0.5); iv = floor(yw + 0.5);
  } else {
    iu = vec3(i0.x, i1.x, i2.x); iv = vec3(i0.y, i1.y, i2.y);
  }
// 7.  Compute the hash for each of the simplex corners.
  vec3 hash = mod(iu, 289.0);
  hash = mod((hash*51.0 + 2.0)*hash + iv, 289.0);
  hash = mod((hash*34.0 + 10.0)*hash, 289.0);
// 8, 9a.  Generate the gradients with an optional rotation.
  vec3 psi = hash*0.07482 + alpha;
  vec3 gx = cos(psi); vec3 gy = sin(psi);
  vec2 g0 = vec2(gx.x, gy.x);
  vec2 g1 = vec2(gx.y, gy.y);
  vec2 g2 = vec2(gx.z, gy.z);
// 10.  Compute radial falloff.
  vec3 w = 0.8 - vec3(dot(x0, x0), dot(x1, x1), dot(x2, x2));
  w = max(w, 0.0); vec3 w2 = w*w; vec3 w4 = w2*w2;
// 11.  Linear ramp along gradient (by a scalar product)
  vec3 gdotx = vec3(dot(g0, x0), dot(g1, x1), dot(g2, x2));
// 12, 13.  Multiply and sum up noise terms.
  float n = dot(w4, gdotx);
// 14.  Compute first-order partial derivatives.
  vec3 w3 = w2*w; vec3 dw = -8.0*w3*gdotx;
  vec2 dn0 = w4.x*g0 + dw.x*x0;
  vec2 dn1 = w4.y*g1 + dw.y*x1;
  vec2 dn2 = w4.z*g2 + dw.z*x2;
  gradient = 10.9*(dn0 + dn1 + dn2);
// Scale the noise value to [-1,1] (empirical factor).
  return 10.9*n;
}
```

3D Noise

```glsl
// psrdnoise3.glsl, version 2021-12-02
// Copyright (c) 2021 Stefan Gustavson and Ian McEwan
// (stefan.gustavson@liu.se, ijm567@gmail.com)
// Published under the MIT license, see:
// https://opensource.org/licenses/MIT

vec4 permute(vec4 i) {
  vec4 im = mod(i, 289.0);
  return mod(((im*34.0)+10.0)*im, 289.0);
}

float psrdnoise(vec3 x, vec3 period, float alpha, out vec3 gradient)
{
  const mat3 M = mat3(0.0,1.0,1.0, 1.0,0.0,1.0, 1.0,1.0,0.0);
  const mat3 Mi = mat3(-0.5,0.5,0.5, 0.5,-0.5,0.5, 0.5,0.5,-0.5);
// 2.  Transform to simplex space and find simplex "base" i0.
  vec3 uvw = M*x;
  vec3 i0 = floor(uvw); vec3 f0 = fract(uvw);
// 3.  Enumerate simplex corners and transform back.
  vec3 g_ = step(f0.xyx, f0.yzz); vec3 l_ = 1.0 - g_;
  vec3 g = vec3(l_.z, g_.xy); vec3 l = vec3(l_.xy, g_.z);
  vec3 o1 = min(g, l); vec3 o2 = max(g, l);
  vec3 i1 = i0 + o1, i2 = i0 + o2, i3 = i0 + 1.0;
  vec3 v0 = Mi*i0, v1 = Mi*i1, v2 = Mi*i2, v3 = Mi*i3;
// 4.  Compute distances to corners before we wrap.
  vec3 x0 = x - v0, x1 = x - v1, x2 = x - v2, x3 = x - v3;
// 5, 6.  Wrap to periods and update i0, i1, i2, i3 accordingly.
  if(any(greaterThan(period, vec3(0.0)))) {
    vec4 vx = vec4(v0.x, v1.x, v2.x, v3.x);
    vec4 vy = vec4(v0.y, v1.y, v2.y, v3.y);
    vec4 vz = vec4(v0.z, v1.z, v2.z, v3.z);
    if(period.x > 0.0) vx = mod(vx, period.x);
    if(period.y > 0.0) vy = mod(vy, period.y);
    if(period.z > 0.0) vz = mod(vz, period.z);
    i0 = floor(M * vec3(vx.x, vy.x, vz.x) + 0.5);
    i1 = floor(M * vec3(vx.y, vy.y, vz.y) + 0.5);
    i2 = floor(M * vec3(vx.z, vy.z, vz.z) + 0.5);
    i3 = floor(M * vec3(vx.w, vy.w, vz.w) + 0.5);
  }
// 7.  Compute hash for each of the four corners.
  vec4 hash = permute( permute( permute(
              vec4(i0.z, i1.z, i2.z, i3.z ))
            + vec4(i0.y, i1.y, i2.y, i3.y ))
            + vec4(i0.x, i1.x, i2.x, i3.x ));
// 8.  Compute rotating gradients using the "well-behaved" method.
  vec4 theta = hash*3.883222077;
  vec4 sz = 0.996539792 - 0.006920415*hash;
  vec4 psi = hash*0.108705628;
```

```
    vec4 Ct = cos(theta); vec4 St = sin(theta);
    vec4 sz_prime = sqrt(1.0 - sz*sz);
    vec4 gx, gy, gz;
// 9b.  Rotate with alpha if desired, else take shortcut.
    if(alpha != 0.0) {
      vec4 px = Ct*sz_prime; vec4 py=St*sz_prime;
      vec4 pz = sz;
      vec4 Sp = sin(psi); vec4 Cp = cos(psi);
      vec4 Ctp = St*Sp - Ct*Cp;
      vec4 qx = mix( Ctp*St, Sp, sz);
      vec4 qy = mix(-Ctp*Ct, Cp, sz);
      vec4 qz = -(py*Cp + px*Sp);
      vec4 Sa = vec4(sin(alpha)); vec4 Ca = vec4(cos(alpha));
      gx = Ca*px + Sa*qx; gy = Ca*py + Sa*qy; gz = Ca*pz + Sa*qz;
    } else {
      gx = Ct * sz_prime; gy = St * sz_prime; gz = sz;
    }
    vec3 g0 = vec3(gx.x, gy.x, gz.x), g1 = vec3(gx.y, gy.y, gz.y);
    vec3 g2 = vec3(gx.z, gy.z, gz.z), g3 = vec3(gx.w, gy.w, gz.w);
// 10.  Compute radial falloff.
    vec4 w = 0.5-vec4(dot(x0,x0), dot(x1,x1), dot(x2,x2), dot(x3,x3));
    w = max(w, 0.0); vec4 w2 = w*w; vec4 w3 = w2*w;
// 11.  Linear ramps along gradients (by scalar product)
    vec4 gdotx = vec4(dot(g0,x0), dot(g1,x1), dot(g2,x2), dot(g3,x3));
// 12, 13.  Multiply and sum up the four noise terms.
    float n = dot(w3, gdotx);
// 14.  Compute first-order partial derivatives.
    vec4 dw = -6.0*w2*gdotx;
    vec3 dn0 = w3.x*g0 + dw.x*x0; vec3 dn1 = w3.y*g1 + dw.y*x1;
    vec3 dn2 = w3.z*g2 + dw.z*x2; vec3 dn3 = w3.w*g3 + dw.w*x3;
    gradient = 39.5 * (dn0 + dn1 + dn2 + dn3);
// Scale noise value to range [-1,1] (empirical factor).
    return 39.5*n;
}
```

Index of Supplementary Materials

The source code is in the GitHub repository https://github.com/stegu/psrdnoise/, as well, along with an expanded WebGL gallery with a tutorial on how to use the functions. Flow noise is a powerful creative tool for writing animated shaders, and it does not really show its full utility in still images. It is our intention to maintain the
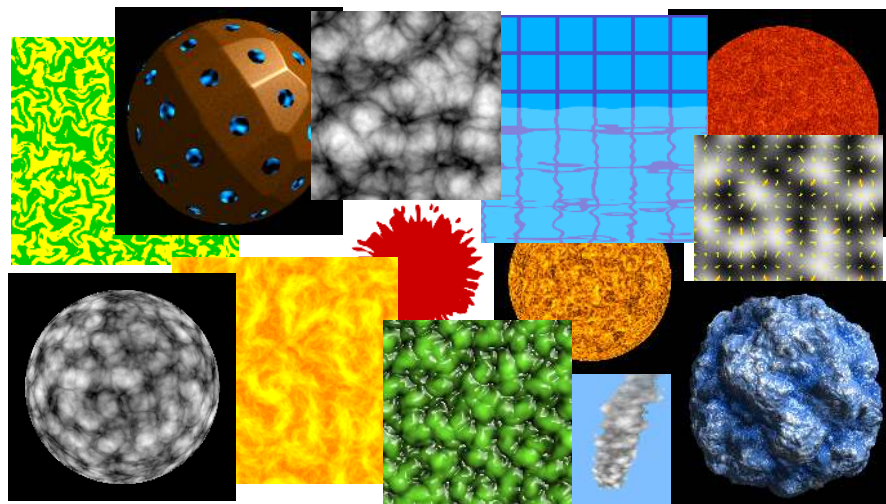
Figure 7. Some screenshots from the online gallery and tutorials.

code in this GitHub repository, and we will use it to post any updates, improvements, or bug fixes to the functions made after the publication of this article. Figure 7 shows a collage of screenshots from the online gallery at the time of publication.

References

BRIDSON, R., HOURIHAN, J., AND NORDENSTAM, M. 2007. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics 26*, 3, 46–es. URL: https://doi.org/10.1145/1276377.1276435, doi:10.1145/1276377.1276435. 22

COOK, R. L., AND DEROSE, T. 2005. Wavelet noise. *ACM Transactions on Graphics 24*, 3, 803–811. URL: https://doi.org/10.1145/1073204.1073264, doi:10.1145/1073204.1073264. 27

GUSTAVSON, S., 2005. Simplex noise demystified. Self-published tutorial. URL: https://www.researchgate.net/publication/216813608_Simplex_noise_demystified. 17

MCEWAN, I., SHEETS, D., RICHARDSON, M., AND GUSTAVSON, S. 2012. Efficient computational noise in GLSL. *Journal of Graphics Tools 16*, 2, 85–94. URL: https://doi.org/10.1080/2151237X.2012.649621, doi:10.1080/2151237X.2012.649621. 18, 20, 26

PERLIN, K., AND NEYRET, F. 2001. Flow noise. In *Siggraph 2001 Technical Sketches and Applications*, ACM, 187. URL: https://hal.inria.fr/inria-00537499/document. 18, 21, 22

PERLIN, K. 1985. An image synthesizer. *ACM SIGGRAPH Computer Graphics 19*, 3, 287–296. URL: https://doi.org/10.1145/325334.325247, doi:10.1145/325334.325247. 17

PERLIN, K. 2002. Noise hardware. In *Course Notes from Siggraph 2002, Course 36: Real-Time Shading Languages*, SIGGRAPH 2002, ACM. URL: `https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf`. 17, 23

SHARPE, B., 2012. Simplex noise. Personal blog post, January 13. URL: `https://briansharpe.wordpress.com/2012/01/13/simplex-noise/`. 23

Author Contact Information

Stefan Gustavson
Media and Information Technology, ITN
Linköping University,
Norrköping, Sweden SE-60174
stefan.gustavson@liu.se

Ian McEwan
ijm567@gmail.com