Tiling simplex noise and flow noise in two and three dimensions: The nitty-gritty details

Stefan Gustavson and Ian McEwan

December 16, 2021

Introduction

This is supplementary material for the article "Tiling simplex flow noise in two and three dimensions". It repeats most of what is in the article, while digging deeper into the background, theory and implementation details. Furthermore, you will not find formal citations or a list of references here. Those are in the article proper.

1 Algorithm overview

The algorithm for our simplex noise has several steps, and the GLSL source code can be challenging to read for someone not already familiar with similar implementations. The verbal description in Table 1 is a general guide to what operations are performed, and in which order, and it is our hope that it will put the implementation details into some context, as well as make the GLSL code easier to understand.

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. The 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.

2 Tiling simplex grids

The formal definition of an N-dimensional *simplex grid* is a space-filling tiling (a *honeycomb*) of congruent polytopes (polygons in 2-D, polyhedra

1)	Input point in texture space (2-D or 3-D) is $\vec{x}$
2)	Transform to simplex space, determine which simplex $\vec{x}$ is in
3)	Transform each corner of that simplex back to texture space
4)	Compute vectors from each simplex corner to *vecx*
*5)	Wrap corners to the desired period along each axis
*6)	Transform the wrapped corners to simplex space again
7)	Compute a pseudo-random hash k for each corner
8)	Generate pseudo-random gradients $\hat{g}_i$ from the hash k
*9a)	For 2-D: Rotate the gradients in the plane by a varying angle α
*9b)	For 3-D: Rotate gradients around a pseudo-random axis by α
10)	Compute radial falloffs w_i based on distances from $\vec{v}$ to corners
11)	Make a linear ramp along the gradient from each simplex corner
12)	Multiply the ramp values at $\mathbf{v}$ 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

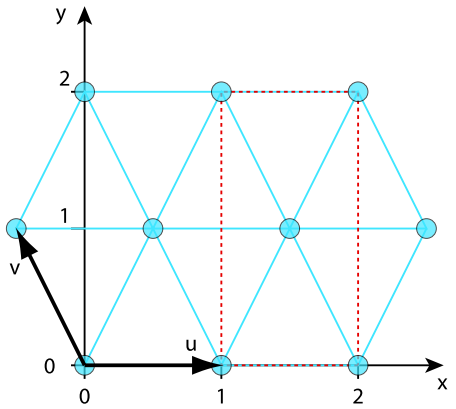
Table 1: *A step-by-step overview of our 2-D and 3-D noise algorithms.*

in 3-D) of the most simple kind (*simplex*) for the particular dimension N. The *optimal simplex grid* has the most uniform edge lengths for the tiling polytopes. For 2-D space, this is a tiling of equilateral triangles, and for 3-D space, it is the *tetragonal disphenoid honeycomb*, a tiling of congruent tetrahedra whose faces are isosceles triangles with one long edge and two short edges, the edges differing in length by a factor of $\sqrt{3}/2$. (Regular tetrahedra with equilateral faces do not tile 3-D space.)

The tetragonal disphenoid honeycomb can be extended to grids of a similar structure in higher dimensions, although the tiling simplex polytopes become increasingly elongated with increasing dimensionality N, and the distances between vertices increasingly non-uniform. This in turn causes a high-dimensional noise field generated by Perlin's method of summation of oriented wavelets of limited N-spherical extent ("wiggles") to lack detail in some regions – the function will be zero over large portions of N-space. 4-dimensional simplex noise using this kind of grid is still reasonably rich in detail and has proven useful, but considering how a 4-D function would require a lot of storage space to precompute, we have not pursued creating a tiling version of it. The most common use of 4-D Perlin noise is to animate the fourth coordinate to make 3-D patterns that change over time. A similar effect can be achieved at a considerably lower cost, and in a strictly periodic fashion better suited for precomputed animation loops, by gradient rotation in our 3-D noise.

2.1 2-D tiling

In 2-D, the vertices of tiled equilateral triangles form a regular hexagonal lattice which can be reinterpreted as a staggered rectangular lattice. Given the non-rational proportions between the base and the height of equilateral triangles, there is no rotation or uniform scaling of the optimal grid that will make the vertices coincide with integers along both dimensions, but we can adjust the proportions of the triangles to make the grid slightly suboptimal but tile over integer periods. The non-uniform scaling could be custom made to match the desired tile size, causing only a very slight distortion for larger tiling periods. However, considering the inherently irregular and broad-band nature of the noise pattern as such, we opt for simplicity and scale the grid to make a staggered square lattice where every second vertex hits integer coordinates, making the grid tile over any integer period in the x direction and any *even integer* period in the y direction, see Figure 1. The transformations are simply $(u, v) = (x + \frac{1}{2}y, y)$ and $(x, y) = (u - \frac{1}{2}v, v)$. The resulting slight stretching of the grid spacing along one direction is generally not noticeable in practice. The individual noise wiggles are still isotropic in (x, y), only spaced somewhat further apart along the y direction, and therefore the noise pattern as such appears more isotropic than what would have been the case if a simple texture domain scaling had been used.



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

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

Figure 1: *The modified 2-D simplex grid with base vectors* $\vec{u} = (1, 0)$, $\vec{v} = (-0.5, 1)$. *The slightly stretched hexagonal grid (blue lines and circles) 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.*

2.2 3-D tiling

There seems to be a misconception that the optimal simplex grid in 3-D has weird, non-rational angles and is therefore a bad fit for axis-aligned tiling. This is most likely due to the unconventional transformation used by Ken

Perlin when transforming between (x, y, z) texture space and (u, v, w) simplex space. However, the vertices of the tetragonal disphenoid honeycomb form a highly regular body-centered cubic lattice which can be conveniently mapped to an integer grid, see Figure 2. Perlin's transformation rotates the (u, v, w) grid 180 degrees around its main diagonal $u = v = w$, which obscures its regular nature and makes tiling over axis-aligned integer periods in (x, y, z) cumbersome. The angles are still rational, and in fact the rotated grid repeats periodically with a translational cube of size $3 \times 3 \times 3$ units, but a grid that tiles only with periods that are multiples of 3 is somewhat too restrictive for practical use, and inherent numerical errors in Perlin's transformations add to the hassle of wrapping the vertices properly to the period. Performing the transformations between (x, y, z) and (u, v, w) space with ordinary matrix multiplications readily produces the grid orientation shown in Figure 1, which makes it tile with *any* integer period in (x, y, z).

The transformation used by Perlin is $(u, v, w) = (x+a, y+a, z+a)$ where $a = \frac{1}{3}(x+y+z)$, involving one multiplication and five additions, which is indeed considerably less work than a general matrix multiplication. However, our forward transformation matrix has three zeroes and six coefficients equal to 1, meaning that it actually requires no multiplications and only three additions in an optimized implementation: $(u, v, w) = (y + z, x + z, x + y)$. Perlin's inverse is $(x, y, z) = (u-b, v-b, q-b)$ with $b = \frac{1}{6}(u+v+w)$, which requires one multiplication and five additions/subtractions, whereas our inverse requires three multiplications with the very binary-friendly factor $\frac{1}{2}$ and six additions/subtractions: $(x, y, z) = \frac{1}{2}(-u+v+w, u-v+w, u+v-w)$. A multiplication with $\frac{1}{2}$ can be implemented as a shift operation in fixed-point arithmetic or a simple exponent decrement in floating point, whereas Perlin's multiplications have considerably less convenient factors, $\frac{1}{3}$ and $\frac{1}{6}$, each requiring a full multiplication and introducing more prominent numerical errors. Moreover, we have found that it doesn't even execute faster than a 3×3 matrix multiplication in actual implementations on modern GPUs. The matrix transformations and their inverses are given below for reference, where M is the matrix for transforming a column vector from (x, y, z) coordinates to (u, v, w) coordinates in our grid, P is the matrix for performing that transformation to Perlin's grid, and M^{-1} and P^{-1}, respectively, are their inverses. The M and P matrices can be transformed into each other by the matrix R, which performs a 180 degree rotation around the main diagonal. R is its own inverse – an *involutory matrix*.

$$M=\begin{bmatrix}0 & 1 & 1\\ 1 & 0 & 1\\ 1 & 1 & 0\end{bmatrix}\ M^{-1}=\begin{bmatrix}-\frac{1}{2} & \frac{1}{2} & \frac{1}{2}\\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2}\\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2}\end{bmatrix}\ P=\begin{bmatrix}\frac{4}{3} & \frac{1}{3} & \frac{1}{3}\\ \frac{1}{3} & \frac{4}{3} & \frac{1}{3}\\ \frac{1}{3} & \frac{1}{3} & \frac{4}{3}\end{bmatrix}\ P^{-1}=\begin{bmatrix}\frac{5}{6} & -\frac{1}{6} & -\frac{1}{6}\\ -\frac{1}{6} & \frac{5}{6} & -\frac{1}{6}\\ -\frac{1}{6} & -\frac{1}{6} & \frac{5}{6}\end{bmatrix}\ R=\begin{bmatrix}-\frac{1}{3} & \frac{2}{3} & \frac{2}{3}\\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3}\\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3}\end{bmatrix}$$

A single tetrahedral simplex of the optimal simplex grid, in the orientation represented by the transformation matrix M above, is shown to the far right of Figure 2. Translated and 90 degree rotated instances of this shape can fill 3-D space. The most straightforward construction of the space-filling arrangement is to form octahedra by 4-clusters of tetrahedra touching along one of their long edges (blue shapes in Figure 2). An infinite repeat of such octahedra body-centered in a cubical grid, as indicated in the figure, fill one third of space, and two more repeats (not shown) are required, both being edge-centered on the cubical grid in the plane where the top and bottom cusps of the first set touch, and each oriented at a right angle with the others.

An alternative and less obvious construction is to cluster *six* tetrahedral simplices touching along one of their short edges to form a parallelepiped (center of Figure 2, pink shape) with principal edge vectors $\vec{u} = (-\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$, $\vec{v} = (\frac{1}{2}, -\frac{1}{2}, \frac{1}{2})$, $\vec{w} = (\frac{1}{2}, \frac{1}{2}, -\frac{1}{2})$. This shape tiles with translations alone, and two such "tilted and squashed cubes" (formally, the shape is a *trigonal trapezohedron*) arranged face to face by a unit translation along either of the axes u, v, w form a convex shape that tiles 3-D space when replicated at integer steps along x, y, z. The motivation for using this 6-cluster instead of the 4-cluster is that it makes it algorithmically simpler to determine exactly which simplex contains a certain point. Identifying the origin of the 6-cluster that contains a point is only a matter of finding the nearest smaller integer coordinates in (u, v, w), and identifying which of the 6 tetrahedra the point is in is a matter of ranking the magnitude of the components of the local (u, v, w) vectors from that local origin to the point. Perlin's original formulation of simplex noise uses this method, although in its differently oriented grid.

3 Pseudo-random hash function

Historically, most implementations of noise have used a lookup table to provide a controlled and repeatable pseudo-random gradient for each grid point. In 2001, Perlin recognized the need for a hardware-friendly, less memory-intensive hash, and used a very small lookup table together with some bit-twiddling for his simplex noise, with good visual results, but it was specific to his use of only 12 unique gradients to save on multiplications, and for our 3-D flow noise we need more than 12 or 16 unique pseudo-random values to generate both gradients and rotation axes with enough variation.

Hash functions from cryptography are usually purely computational, but they use many operations to achieve good spectral properties for their output. This is not a good fit for a noise function that needs to be fast on a GPU, and where we only need enough randomness for the noise to look okay to a human observer, not for the hash to be cryptographically strong. Some

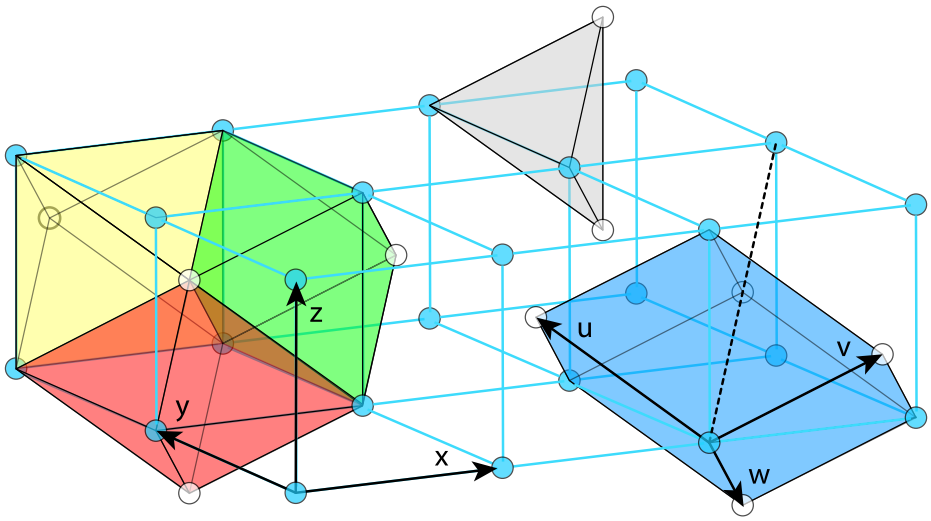


Figure 2: ***Left:*** *The simplex grid can be constructed from groups of four tetrahedra forming octahedra in three different orientations (yellow, green and red), which together tile 3-D space.* ***Top:*** *A single simplex tetrahedron (gray), with two corners vertex-centered on the* (x, y, z) *cubical lattice (blue circles) and two body-centered (white circles).* ***Right:*** *Six tetrahedra sharing an edge along a grid diagonal (dotted line) form a parallelepiped with edges parallel to the* (u, v, w) *basis vectors (blue) which tiles 3-D space with translations alone.*

simple "quick and dirty" cryptographic-like hashes have been suggested in literature, and others of the same general structure could be imagined, but they use integer arithmetic, shifts and bit-wise logic operations, which is still unsuitable for GPU implementations when aiming for maximum performance and platform compatibility.

Borrowing a useful idea from our own previous work, we use permutation polynomials to generate pseudo-random hashes from the integer coordinates of simplex grid points. It is our opinion that this method, despite its inferior statistical properties, combines computational simplicity with a good *apparent* randomness of the noise pattern. Our permutation field has size $17^2 = 289$ for the same reason as in the previous work: it's reasonably large but keeps the integers in all intermediate results small enough to be implemented in single-precision floating point arithmetic without losing precision. However, our choice of permutation polynomials are different from our previously published noise functions, because we have since found that the old choice $34x^2 + x$ caused some visible structures in the noise pattern in the form of frequent diagonal streaks. Our new choice is $34x^2 + 10x$ for the 3-D case, which is only slightly different but does not cause any of the pattern defects we encountered with the old polynomial, and a second polynomial

$34x^2 + 10x$	$68x^2 + 6x$	$85x^2 + 8x$	$102x^2 + 6x$
$51x^2 + 2x$	$68x^2 + 10x$	$102x^2 + 2x$	$102x^2 + 12x$

Table 2: *Permutation polynomials that were found to generate noise of the best visual quality with our particular method for selecting gradients.*

$51x^2 + 2x$ for the 2-D case. In the 3-D noise, the gradient selection from a Fibonacci sphere decorrelates the vectors, but for 2-D noise we used a different polynomial for each dimension to get rid of objectionable defects in the visual noise pattern. Changing even a seemingly minor detail in how the 2-D gradients are selected can reintroduce strong patterns in the noise field. Deciding on the empirically determined "magic factor" 0.07482 in the code required quite a bit of experimentation, and there may still be better choices.

In our visual assessment of polynomials, we tested all float-compatible permutation polynomials with ring size 289 of the form $P(x) = 17Ax^2 + Bx$ and found that only very few of them, as listed in Table 2, performed well in generating 3-D noise patterns with "good apparent randomness", meaning a fine-grained irregular-looking pattern without recurring patterns or streaks on a larger scale anywhere in its 289^3 domain. The test was performed as a subjective visual evaluation with only a small panel of volunteers (one of the authors, his wife and one colleague), because it was a rather time consuming and boring process for the test subjects to actually watch the entire noise domain pass before their eyes. However, the subjective evaluations were very clear and consistent between observers. The eight polynomials in Table 2 showed equally good performance, meaning that no objectionable structure or regularity was spotted in the visual evaluation, and we arbitrarily chose the one with the smallest A coefficient. Most polynomials performed very badly, despite being formal permutations like the others. Specifically, our previously published polynomial $34x^2 + x$ did not fare well at all in this test. In retrospect, we consider it barely passable as a hash function for 3-D noise, and downright bad for 2-D noise. At the time of writing, we have no good way other than visual evaluation to determine which permutation polynomials and related gradient selection strategies are good for this purpose, and we decided not to investigate that further within the scope of this article.

4 Rotating gradients

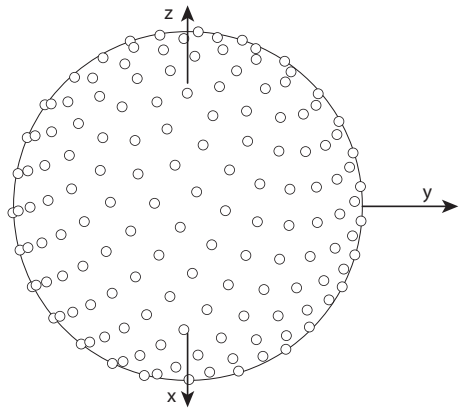
Flow noise, as decribed by Perlin and Neyret, performs an animation of the noise pattern by allowing its underlying generating gradients at each grid point to rotate dynamically. Their implementation was restricted to 2-D, but we extend it to 3-D.

4.1 2-D gradients

In 2-D space, the gradients are all rotated in the same direction in the plane, at the same angular speed, as suggested by Perlin and Neyret. The pseudo-random gradients for our 2-D simplex noise are uniformly distributed on the unit circle and generated by sine and cosine functions of a pseudo-random angle, which means that an extra rotation can be performed by one single addition, which comes at a negligible cost. Instead of calling sin and cos with our chosen pseudo-random hashed argument $2\pi\frac{k}{41}$, $k \in \{0, \ldots, 288\}$, we call them with the argument $2\pi\frac{k}{41} + \alpha$, where α is the dynamic rotation.

4.2 3-D gradients

Our approach for selecting a single gradient $\hat{g}_i$ without the rotation uses a Fibonacci spiral mapping from a 1-D interval to evenly distributed points on a sphere, see Figure 3. To select a pseudo-random integer k in the interval $\{0, \ldots, 288\}$, we use the permutation polynomial mentioned above, $P(x) = (34x^2 + 10x) \bmod 289$, and perform the permutation on the (u, v, w) integer coordinates for a simplex vertex one at a time in sequence as $k = P(P(P(u) + v) + w)$.



$$
\begin{aligned}
k &\in \{0, \ldots, 288\} \\
\theta &= 4\pi k/(1+\sqrt{5}) \\
z &= \cos\phi = 1 - 2(k+\tfrac{1}{2})/289 \\
w &= \sin\phi = \sqrt{1-z^2} \\
x &= w\cos\theta \\
y &= w\sin\theta
\end{aligned}
$$

Figure 3: *289 points in an equal-area distribution along a Fibonacci spiral on a sphere (only the front facing half are shown), and their coordinates.*

An extension to 3-D of the rotating gradients would look too regular if all rotations were to be performed around the same axis as for the 2-D case. Therefore, we abandon the design constraint of 2-D flow noise that the rotations should not alter the inter-correlation between gradients, and pick an individual, pseudo-random rotation axis for each gradient. 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. Our choice of algorithm for these rotations is to compute not one, but two pseudo-random orthonormal generating

gradients $\hat{p}_i, \hat{q}_i$ for each influencing vertex i, and sum them with sine and cosine weights to combine into one gradient rotating in their common plane: $\hat{g}_i = \hat{p}_i \cos\alpha + \hat{q}_i \sin\alpha$, see Figure 4.

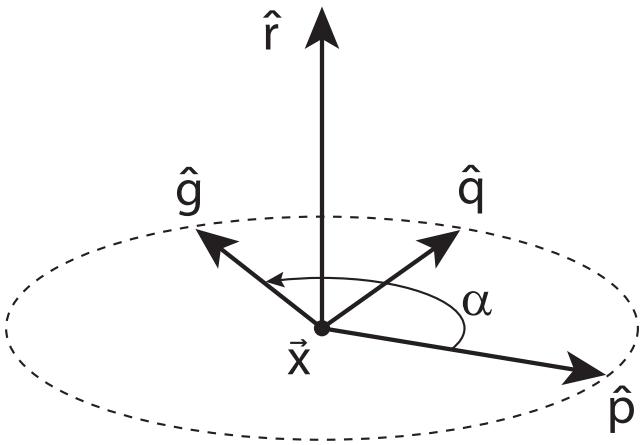


Figure 4: *Rotating gradient $\hat{g}$ created from two orthonormal vectors $\hat{p}$ and $\hat{q}$*

A straightforward way to generate $\hat{q}$ would be to construct a vector at a pseudo-random angle in the plane orthogonal to $\hat{p}$, for example by using a second hash function to generate a second vector, not coincident with $\hat{p}$, and making it orthonormal to the first vector by Gram-Schmidt orthogonalization. Using a Fibonacci sphere distribution for the two vectors, this construction would involve to two hash lookups, one pair of sin and cos operations for each vector, one normalization operation and one extra sin/cos pair for the rotation with α, along with various multiplications and additions. This would be a reasonably efficient method, but we can do better.

Our approach is to generate both vectors $\hat{p}$ and $\hat{q}$ from a single hash value, in a manner that does not require subsequent orthonormalization, but instead guarantees orthonormality by the construction method. This turns out to be substantially more efficient. Somewhat counterintuitively, we begin by selecting the rotation axis $\hat{r}$ from a Fibonacci spiral in the same manner as we selected $\hat{g}$ without the rotation. Then we compute the cross product between $\hat{r}$ and the positive z axis $(0, 0, 1)$, which amounts to just one component swap and a sign flip: $(x, y, z) \times (0, 0, 1) = (y, -x, 0)$, and normalize the two-component result. (In fact, while generating $\hat{r}$, the normalized (x, y) components of this vector are available as an intermediate result, so the normalization is implicit.) This results in a vector $\hat{q}$ in the plane $z = 0$ which is orthonormal to $\hat{r}$. Note that this requires that none of the pseudo-random choices for $\hat{r}$ are parallel to the z axis. Our particular mapping satisfies this condition.

$\hat{r}$ from Fibonacci sphere according to Figure 3

$$\hat{q} = \frac{\hat{r} \times \hat{z}}{\|\hat{r} \times \hat{z}\|}$$

$$\hat{p} = \hat{q} \times \hat{r}$$

$$\psi = \frac{10\pi}{289} k \text{ (re-use } k \text{ from selecting } \hat{r})$$

$$\hat{g} = \hat{p} \cos(\psi + \alpha) + \hat{q} \sin(\psi + \alpha)$$

Table 3: *Our fast algorithm for computing* $\hat{g}(\alpha)$

Next, we compute $\hat{p}$ as $\hat{q} \times \hat{r}$. We now have two orthonormal vectors, $\hat{p}$ and $\hat{q}$, which can be used to generate a rotating vector by summing them with sine and cosine weights. However, $\hat{q}$ is in the plane $z = 0$, which would make the gradients rather strongly correlated: every rotating gradient would hit the plane $z = 0$ at the same rotation angles $\alpha = \pi/2$ and $\alpha = 3\pi/2$. To decorrelate our vectors, we nudge the rotations out of sync by offsetting with a pseudo-random angle ψ that is computed from the same hashed index value k that was used for the selection of $\hat{p}$. Then, just like in the 2-D case, the dynamic rotation with an angle α is performed by a simple addition: the angle argument for the sine and cosine factors for $\hat{p}$ and $\hat{q}$ is $\psi + \alpha$.

This algorithm for generating rotating 3-D gradients, summarized in Table 3, performs the hash-based rotation by ψ and the animated rotation by α around the same axis, just like in the 2-D case, thereby saving on trigonometric functions as well as on multiplications.

One disadvantage of this algorithm is that the vectors for $\alpha = 0$ are different from the unrotated vectors picked directly from the Fibonacci sphere. Ideally, we would want the case $\alpha = 0$ to correspond to $\hat{g} = \hat{p}$, so that the rotation-enabled noise pattern with a zero rotation matches the noise pattern from when rotations are disabled. This could be achieved by resorting to the two-vector method presented above, but there is a better optimized way that requires less computations. It takes almost twice as long to generate a rotated gradient than the algorithm of Table 3, but it's still faster than the two-vector method. A strong advantage of this method is that it allows a dynamic run-time shortcut if $\alpha = 0$ across all cores in a SIMD execution kernel, which gives a dramatic speedup if rotations are not required. We present this more well-behaved algorithm in Table 4 without comments. It pains us to do this, but the full derivation requires a more formal explanation of the vector math than what we did above, and we pushed that to a separate document, a "supplement to the supplement", instead of making this already long text even longer. That extra document gets quite math

$\hat{p}$ from Fibonacci sphere according to Figure 3

($\sin\theta$, $\cos\theta$, $\sin\phi$ and $\cos\phi$ are re-used below)

$$\psi = \frac{10\pi}{289}k \text{ (re-use } k \text{ from selecting } \hat{p})$$

$$q_x = \sin\psi\cos\phi + \sin\theta(\sin\psi\sin\theta - \cos\psi\cos\theta)(1 - \cos\phi)$$

$$q_y = \cos\psi\cos\phi - \cos\theta(\sin\psi\sin\theta - \cos\psi\cos\theta)(1 - \cos\phi)$$

$$q_z = -(p_y\cos\psi + p_x\sin\psi)$$

$$\hat{q} = (q_x, q_y, q_z)$$

$$\hat{g} = \hat{p}\cos\alpha + \hat{q}\sin\alpha$$

Table 4: *Our well-behaved algorithm for computing $\hat{g}_i(\alpha)$. For the rather lengthy derivation of this, see the separate supplementary document. Note that if $\alpha = 0$, $\hat{g} = \hat{p}$ and everything but the first step can be skipped.*

intense indeed, but it presents *all* the details concerning our generation of the rotating vectors.

5 Analytic derivatives

Simplex noise is a sum of polynomials, and its derivative is a sum of polynomials one degree lower. Hence, the analytic derivative is easy to compute. Taking proper care to save intermediate results, the computation comes at very little extra cost. This is a strong advantage of simplex noise. The final value of classic Perlin noise is computed by repeated interpolation rather than summation, creating a polynomial of polynomials in three nested iterations for 3-D noise. This results in a comparably high order polynomial for the final noise value: 18^{th}-degree using the now-recommended 5^{th}-degree interpolant. The classic noise value as such is computed by sequential evaluation in a simple enough fashion, but its closed expression is convoluted, its derivative is quite cumbersome to compute, and the expressions for second order mixed partial derivatives become downright nasty in comparison.

The expressions for the noise value and its first order partial derivatives of our version of simplex noise are presented in Table 5. In the equations, $\vec{x}_i$ are the vectors from 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 2-D case, the components of $\vec{x}_i$ are (x_i, y_i), and for the 3-D case they are (x_i, y_i, z_i). Accordingly, components of ∇n are either $(\frac{\partial n}{\partial x}, \frac{\partial n}{\partial y})$ or $(\frac{\partial n}{\partial x}, \frac{\partial n}{\partial y}, \frac{\partial n}{\partial z})$, and $\hat{g}_i$ is either $(g_{x,i}, g_{y,i})$ or $(g_{x,i}, g_{y,i}, g_{z,i})$. The expressions in

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

Table 5: *Equations for simplex noise n and its gradient ∇n*

2-D

$$\frac{\partial^2 n}{\partial x^2} = \sum_i (48 w_i^2 (\hat{g}_i \bullet \vec{x}_i) {x_i}^2 - 8 w_i^3 ((\hat{g}_i \bullet \vec{x}_i) + 2 g_{x,i} x_i))$$

$$\frac{\partial^2 n}{\partial y^2} = \sum_i (48 w_i^2 (\hat{g}_i \bullet \vec{x}_i) {y_i}^2 - 8 w_i^3 ((\hat{g}_i \bullet \vec{x}_i) + 2 g_{y,i} y_i))$$

$$\frac{\partial^2 n}{\partial x \partial y} = \sum_i (48 w_i^2 (\hat{g}_i \bullet \vec{x}_i) x_i y_i - 8 w_i^3 (g_{x,i} y_i + g_{y,i} x_i))$$

Table 6: *Second order derivatives for 2-D simplex noise*

Tables 6 and 7 for second order derivatives use the individual components in places. The computations described by each expression do not amount to significantly more work than those for the gradient, although there are three unique second-order partial derivatives compared to two first-order derivatives for the 2-D case, and six instead of three for the 3-D case.

Second order derivatives are not nearly as often useful as the gradient, but the source code files in the supplementary material and the Github repository include GLSL functions named `psrddnoise()` (note the extra "d" in the name) that compute them in addition to the gradient.

Should you ever need differential properties of noise, please remember that for simplex noise, it takes a whole lot less work to compute them analytically than by finite difference approximations.

6 Removing discontinuitites

In the summation of oriented wavelet "wiggles" that constitute the noise field, each such wiggle centred at a vertex of the simplex grid should have a region of influence that does not extend past the immediately surrounding simplices. A fundamental assumption of our vertex traversal is that the noise value at a certain point is influenced only by the four corners of the simplex

3-D

$$\frac{\partial^2 n}{\partial x^2} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) x_i^2 - 6 w_i^2 ((\hat{g}_i \bullet \vec{x}_i) + 2 g_{x,i} x_i))$$

$$\frac{\partial^2 n}{\partial y^2} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) y_i^2 - 6 w_i^2 ((\hat{g}_i \bullet \vec{x}_i) + 2 g_{y,i} y_i))$$

$$\frac{\partial^2 n}{\partial z^2} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) z_i^2 - 6 w_i^2 ((\hat{g}_i \bullet \vec{x}_i) + 2 g_{z,i} z_i))$$

$$\frac{\partial^2 n}{\partial x \partial y} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) x_i y_i - 6 w_i^2 (g_{x,i} y_i + g_{y,i} x_i))$$

$$\frac{\partial^2 n}{\partial y \partial z} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) y_i z_i - 6 w_i^2 (g_{y,i} z_i + g_{z,i} y_i))$$

$$\frac{\partial^2 n}{\partial x \partial z} = \sum_i (24 w_i (\hat{g}_i \bullet \vec{x}_i) z_i x_i - 6 w_i^2 (g_{z,i} x_i + g_{x,i} z_i))$$

Table 7: *Second order derivatives for 3-D simplex noise*

which contains the point. In the 3-D simplex grid, the largest allowable spherical region around a vertex has the radius $1/\sqrt{2}$. Ken Perlin's original reference Java implementation of 3-D simplex noise actually violates this by making the region of influence slightly larger, $\sqrt{3/5}$. Contrary to his claim, this causes slight discontinuities in both the noise and its gradient at simplex boundaries. Our version rectifies this by reducing the extent to its allowed maximum, making the noise continuous with continuous first and second order derivatives.

Perlin's reference implementation 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 $(0.6 - r^2)^4$. Our correction of the radius of influence means we use 0.5 instead of 0.6 in this expression. This makes the region of influence smaller, 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 more or less the same fashion as Perlin's function, see Figure 5.

The discontinuities are small in amplitude but would cause intermittent problems with a finite difference approximation to the gradient, and they cause straight edge artifacts when thresholding noise at zero. Moreover, when the analytical gradient is used, these discontinuities mess up analytic antialiasing at simplex borders, and they can cause visible creases when using noise for bump mapping.

Perlin provided a reference implementation only of 3-D simplex noise, but his general design can be applied to lower as well as higher dimensions. For the 2-D grid in Figure 1, the maximum allowable radius is $\frac{2}{\sqrt{5}}$, which

squared is $\frac{4}{5} = 0.8$, so we start with the polynomial $(0.8 - r^2)$ and use w^4 for the circular decay function. This is by no means the only possible choice, and you might want to experiment with other decays. Just keep in mind that the decay function must have a radial extent of no more than $\frac{2}{\sqrt{5}}$, that it should be continous and smooth everywhere, and that the function, its derivative and its second derivative should all be zero at the edge of its extent. Remember also to adjust the gradient computations accordingly if you change the decay function.

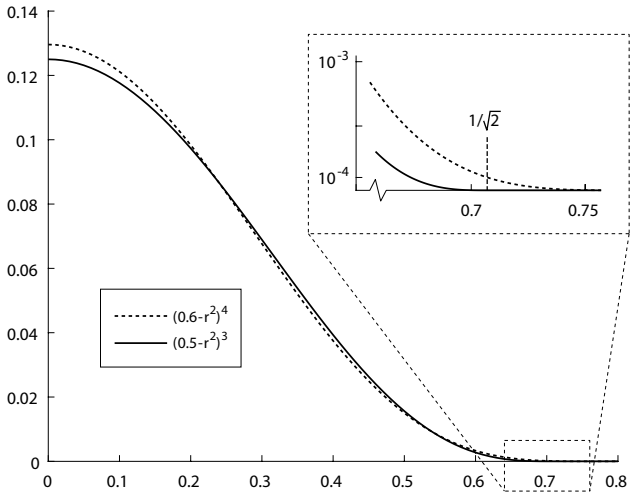


Figure 5: *Perlin's original decay* $(0.6 - r^2)^4$ *having a slighty too large region of influence, and our modified decay* $(0.5 - r^2)^3$*, yielding a very similar result but avoiding discontinuities at simplex boundaries.*

7 Function declarations

The source code for the functions presented below is provided among the supplementary material, and also in the online repository on https://github.com/stegu/psrdnoise/, where we will be posting any updates, improvements or bug fixes made after the publication of this article. It is our intention to maintain the source code in that repository for as long as there is reasonably active interest in it.

```
float psrdnoise(vec2 v, vec2 period, float angle, out vec2 gradient)
float psrdnoise(vec3 v, vec3 period, float angle, out vec3 gradient)
```

These two overloaded functions return 2-D and 3-D noise, respectively, for point `v`, repeating with periods as specified in `period`, rotating the generating gradients with angle `angle` (specified in radians). The gradient of the

noise value is returned in `gradient`. For the 2-D and 3-D versions alike, the range of values for the noise is scaled to cover the range [-1,1] reasonably well without clipping.

```
float psrddnoise(vec2 v, vec2 p, float a, out vec2 g, out vec3 dg)
float psrddnoise(vec3 v, vec3 p, float a,
  out vec3 g, out vec3 dg, out vec3 dg2)
```

These differently named functions additionally compute the partial derivatives of the gradient components, i.e. the second derivatives of the noise function, and return them in `vec3 dg` as $(\frac{\partial^2 n}{\partial x^2}, \frac{\partial^2 n}{\partial y^2}, \frac{\partial^2 n}{\partial xy})$ for the 2-D case and `vec3 dg, vec3 dg2` for the 3-D case, grouped together as $(\frac{\partial^2 n}{\partial x^2}, \frac{\partial^2 n}{\partial y^2}, \frac{\partial^2 n}{\partial z^2})$ and $(\frac{\partial^2 n}{\partial xy}, \frac{\partial^2 n}{\partial yz}, \frac{\partial^2 n}{\partial xz})$, respectively.

The functions have different argument lists than their first derivative counterparts, so we could have picked the same name for them. However, seeing how second order derivatives are a more specialised requirement, we used a different name to avoid confusion. For practical use, you might want to edit the 3-D function to group the return values differently, exclude the computation of values you don't need, or return a compound property such as the Laplacian directly.

8 Performance and speed-ups

Despite the considerable amount of computations in these functions, particularly for the 3-D version, they show good performance on modern hardware, as shown in Table 8. The execution times for the functions were measured as the difference between an almost zero effort shader that writes a constant color and noise shaders operating on a full-screen quad that made ten (for 3-D) or fifteen (for 2-D) calls to the noise function for each fragment, to be at least somewhat taxing even to the high performance GPUs. The constant color shader was clocked to remove the influence of the overhead from clearing, writing and copying render buffers. Of course, the exact performance of a certain GPU depends heavily on the window system, display mode, driver version, memory speed and clock frequency. For high performance laptops, a strongly variable clock frequency combined with an often inadequate cooling made those measurements particularly variable, and therefore all the high performance GPUs we list in the table are desktop systems that did not haphazardly thermal throttle during our short benchmark runs. Even so, the benchmarks were run only on one system of each kind. The figures in Table 8 give a general indication of speed, but they are approximate measures.

It's 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 GPU, and even the exact ver-

GPU	2-D psrdnoise		3-D 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	0.150
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	1.05
Intel UHD 650 (laptop)	1360	0.698	516	1.84
Intel HD 520 (old laptop)	943	1.01	334	2.84

Table 8: *Execution speed for the 2-D and 3-D noise functions, measured as millions of evaluations per second: (frames/second) × (pixels/frame) × (number of calls to* `psrdnoise()` *in the fragment shader), and as nanoseconds per evaluation. The selection of GPUs is simply what we had available for testing.*

sion 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 it requires. As a synthetic but hopefully useful example, we made comparisons between shaders using one component of noise and a shader using a single texture lookup. The 2-D noise performed on par with a texture lookup, even being somewhat faster on high end GPUs, while the 3-D noise generally took two to three times as long to execute than a single 2-D texture lookup.

In these new noise functions, periodic tiling, rotating gradients and analytic derivatives all come at an additional computational cost. We make use of modern optimizing shader compiler and the now seemingly omnipresent GPU feature that conditional statements that take the same branch across all cores in a SIMD fragment batch execute only that branch instead of both. Analytic derivatives that are statically unused by the shader code causes the compiler to remove the related "dead code" for those computations. Not asking for any periodic wrappings by setting the periods to zero in a call makes the function skip quite a few modulo operations and transformations back and forth between texture space and simplex space, and setting the rotation angle α to exactly zero causes the function to generate $\hat{g}$ a lot faster. The total execution time decreases for each feature that is disabled in this manner, according to Figure 6.

Notable in Figure 6 is that the gradient rotation in 2-D costs close to nothing, that its extension to 3-D is still reasonably cheap, and that the gradient is computed at a *remarkably* low cost compared to several repeated evaluations of the entire function for a finite difference approximation. The cost for tiling is higher in the 3-D function (in absolute terms, though not in relative terms), because the 3-D function has three indices rather than

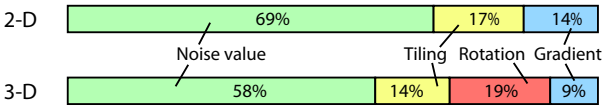


Figure 6: *Not using the extra features of the functions speeds up their execution accordingly, e.g. not tiling to a particular period for 2-D noise cuts 17% off its execution time. The scales are relative – 3-D noise with all features takes about three times longer to compute than 2-D noise.*

two that need to be wrapped to the period, for each of four simplex corners instead of three. Our choice of 2-D grid has one coordinate in common between the (u,v) and (x,y) spaces, hence only one of the two coordinates needs transformations before and after the wrapping in the 2-D case, and the coordinate transformations as such require considerably more computations in the 3-D case.

Switching to the fast rotation algorithm speeds up the 3-D noise function with around 10-15%, depending on the GPU. However, as noted in the section on rotating gradients, if $\alpha = 0$, the "fast" rotation is 10% slower than the default, because it has no early-out shortcut for that special case.

If the tiling, rotation and analytical gradients are *all* disabled, these functions are similar to previously published simplex noise functions. However, they use the same simplex grid and the same noise generation algorithm as the tiling versions, for a qualitative visual match and a less fragmented code base if several variants of noise are desired in the same application. Furthermore, the 3-D function in particular is faster than many other implementations, including the one we published previously and cited in the main article, and both yield a noise with less artifacts and more consistency across platforms than some of the noise functions in current circulation among experimenting shader programmers.

The second order derivatives of the noise field require slightly more computations, even though they also make heavy use of intermediate results from the computation of the noise value. Adding all unique second order derivatives to the functions `psrddnoise()` made them take about 25% longer to execute for 2-D noise and 30% longer for 3-D noise. While this does not come as cheap as the gradient, it is still a low cost compared to the three extra evaluations for 2-D (in addition to the two extra for the gradient) and the six extra for 3-D (in addition to the three extra for the gradient) that would have been required to make finite difference approximations to second order derivatives.

Finally, as a fun benchmark figure that is easy to remember but hard to fathom, we want to mention that top performing GPUs of today (specifically, we used an Nvidia RTX 3080) can evaluate *100 billion noise values per*

second using our 2-D simplex noise without any extra frills, and over 40 billion plain 3-D noise values. We can only begin to imagine what creative shader programmers could do with that kind of procedural power at their fingertips.

9 Discussion

The 3-D simplex grid used in our tiling version is essentially the same as in Perlin's original formulation – it only has a different orientation. Although one of Perlin's stated advantages of simplex noise over classic noise is that it lacks visible artifacts from the underlying grid, it does in fact have such artifacts in certain planes - they are just not aligned with the (x, y, z) coordinate grid, making the problematic orientations unlikely to occur across large planar surfaces in practical use. Because we deliberately align our (u, v, w) simplex grid to the (x, y, z) grid to tile nicely, these grid artifacts are now visible in axis-aligned planes with constant x, y or z. In a manner similar to classic noise, the artifacts 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 7. Unfortunately, the "Hello World!" use case of 3-D 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. Our function looks its absolute worst in this exact scenario.

We advise against using planar 2-D slices of our 3-D tiling simplex noise in axis-aligned or very nearly axis-aligned planes. Even if the noise is precomputed and stored as a tiling 3-D texture, 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. Rotating all texture coordinates 120 degrees around the $u = v = w$ diagonal by means of the matrix R in Section 2.2 creates the same appearance as Perlin's simplex noise, which is more or less free from visible grid artifacts in the (x, y, z) planes. There is an additional `#define` statement in our code to use Perlin's original grid instead. This has the side effect of requiring all periods to be multiples of 3, but tiling is still supported. Requesting a period other than a multiple of 3 makes the actual tiling period three times longer.

It is an unfortunate side effect of the tiling property that implementers need to take special precautions to get good looking planar slices of 3-D noise. Classic Perlin noise has the same problem, but simplex noise was an attempt at solving it, so this represents a throwback of sorts. Perlin's grid orientation leaves a better first impression in random experiments, and shader programmers tend to use that method quite a lot when trying out and evaluating new tools. In some implementations, it might be advisable to set Perlin's grid as the default for these functions and require tilings to

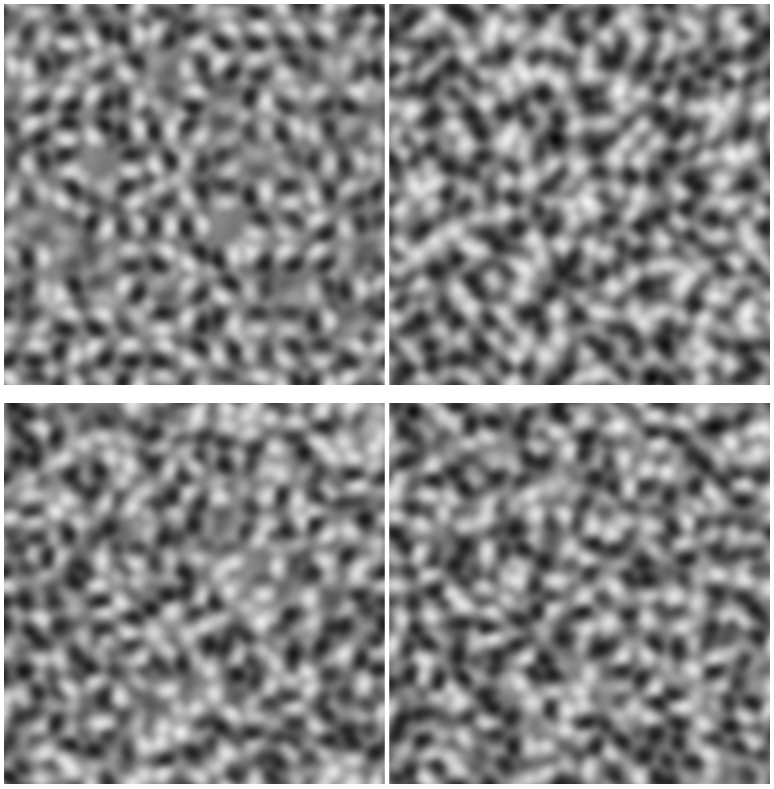


Figure 7: *Using the new tiling simplex grid, 3-D noise has grid artifacts in some axis-aligned planes.* ***Top:*** *Noise with the new grid, in the (x,y) plane at z=0.0 (left) and z=0.25 (right). In the top left image, the individual noise wiggles which are cut right through the middle by the plane dominate the image and are visibly arranged in a regular square grid.* ***Bottom:*** *Noise with Perlin's rotated grid, in the same planes. Noise on Perlin's grid does have the same artifacts, but in planes with off-axis alignments.*

be with periods evenly divisible by 3, reserving the new grid for situations where less restricted tiling periods are required.

On a related note, the kind of animation that is generated by making a 2-D planar cut through 3-D noise and moving it slowly in the third dimension can be much cheaper handled by rotating the gradients in the 2-D function. Additionally, this creates a periodic loop which would be better suited for precomputed animation cycles. The 2-D noise is better band limited than a 2-D slice through 3-D noise (because of the Fourier projection-slice theorem) and, contrary to domain translations, the gradient rotations do not change the general character of the noise pattern.

As mentioned above, our 2-D simplex grid is somewhat stretched to fit an integer tiling scheme. Because of the fairly broad-band, stochastic nature of the noise pattern, we have not found any detectable visual difference

between 2-D simplex noise defined in an optimal, regular hexagonal grid and our modified, slightly anisotropic tiling grid. Nevertheless, should the anisotropy be a problem, a better proportioned tiling grid could be created by mapping an $N \times 2M$ rectangle to a square in texture space, where the ratio $N/2M$ approximates the irrational number $\sqrt{3}/2$. This would create a more isotropic tiling 2-D simplex noise, at the cost of placing stronger constraints on the possible tiling periods.

For the particular functions presented here, the underlying permutation that selects the gradients has a period of $17^2 = 289$, which means that the maximum tiling period that can be requested along any dimension is 289. Specifying larger tiling periods will result in errors in the noise field. If in some application this proves to be a problem, a conditional can be added to not honour requests for larger periods and instead resort to the maximum period of 289 (288 for Perlin's rotated grid). For this kind of performance sensitive functions, we are not fond of precautions that slow down execution when they are not needed, so we decided to leave that potential vulnerability in there. However, the slowdown would be minimal if such conditionals were added, e.g. as one call to the `clamp()` function, so this particular safety measure could be a good idea in applications that allow experimentation. Because WebGL still lacks integer types, the periods are floating-point values, but only integers will work as expected. For GLSL versions where integer types are supported, it would seem like a good idea to change the type of that argument. However, using integer arithmetic for the computations is currently not an advantage. Even high end GPUs of the current generation are better at handling floats than integers. This makes many traditional CPU code optimization tricks useless, often even counter-productive.

The rotation axes for 3-D flow noise should ideally be picked in some actively de-correlated fashion that guarantees no two adjacent gradients being nearly collinear and having similar rotation axes, because two adjacent gradients rotating in sync can stand out as a noticeable rhythmic regularity in our 3-D flow noise when displayed directly as a pattern. Our current implementation exhibits this problem only rarely, and the typical use of animated flow noise is not to display one single instance by itself. A fractal sum across several scales will hide such regularities in the emergent complexity of the pattern.

The modified decay kernel we used for 3-D noise with its reduced extent is a polynomial of a lower degree than for the 2-D noise, and a lower degree than Perlin's original implementation. As a result of this, our 3-D noise is slightly less smooth at simplex boundaries when compared to the 2-D noise. It doesn't show in the noise pattern as such or its gradient, but it does show in the second derivatives. They are still continuous, but not as smooth. If this is a problem, a kernel of higher order but similar shape and extent could be designed, but the simplicity of the current polynomial is appealing, as it

requires few multiplications and has computationally simple derivatives.

An even more uniform 3-D tiling is the *tetrahedral-octahedral honeycomb*, which maps to the face-centered cubical lattice instead of the body-centered cubical lattice. This grid corresponds to optimal sphere packing in 3-D, the equivalent of hexagonal packing in 2-D, and all the edge lengths of all tiling polyhedra are of the same length. However, the tiling is composed of alternating regular tetrahedra and regular octahedra, and its mappings from texture space to grid space and back is not as simple to compute as for the simplex grid. This alternative tiling is used by *OpenSimplex Noise*, but its design was not focused on execution speed.

10 Source code

10.1 2-D noise

This version of the source code has a shortened comment header and contains commented-out code for the second order derivatives. In all other respects, the code is equivalent to the source code in the GLSL files of the supplementary material and in the Github repository.

```
//
// psrdnoise2.glsl
//
// Version: 2021-12-02
// Authors: Stefan Gustavson (stefan.gustavson@liu.se)
//          and Ian McEwan (ijm567@gmail.com)
//
// Published under the MIT license, see:
// https://opensource.org/licenses/MIT
// (In short: free for any use, but give credit to the authors,
// and pass the license on to any derivative works you create.)
//
// Copyright (c) 2021 Stefan Gustavson and Ian McEwan.
//

float psrdnoise(vec2 x, vec2 period, float alpha, out vec2 gradient)
{
  // Transform to simplex space (axis-aligned hexagonal grid)
  vec2 uv = vec2(x.x + x.y*0.5, x.y);

  // Determine which simplex we're in, with i0 being the "base"
  vec2 i0 = floor(uv);
  vec2 f0 = fract(uv);
  // o1 is the offset in simplex space to the second corner
  float cmp = step(f0.y, f0.x);
  vec2 o1 = vec2(cmp, 1.0-cmp);
  // Enumerate the remaining simplex corners
  vec2 i1 = i0 + o1;
  vec2 i2 = i0 + vec2(1.0, 1.0);

  // Transform corners back to texture space
  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);

  // Compute vectors from x to each of the simplex corners
  vec2 x0 = x - v0;
  vec2 x1 = x - v1;
  vec2 x2 = x - v2;

  vec3 iu, iv;
  vec3 xw, yw;
  // Wrap to periods, if desired
  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);
  }

  // Compute one pseudo-random hash value for each corner
  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);

  // Pick a pseudo-random angle and add the desired rotation
  vec3 psi = hash * 0.07482 + alpha;
  vec3 gx = cos(psi);
  vec3 gy = sin(psi);

  // Reorganize for dot products below
  vec2 g0 = vec2(gx.x,gy.x);
  vec2 g1 = vec2(gx.y,gy.y);
  vec2 g2 = vec2(gx.z,gy.z);

  // Radial decay with distance from each simplex corner
  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;

  // The value of the linear ramp from each of the corners
  vec3 gdotx = vec3(dot(g0, x0), dot(g1, x1), dot(g2, x2));

  // Multiply by the radial decay and sum up the noise value
  float n = dot(w4, gdotx);

  // Compute the 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);

//  Second order derivatives can be computed like this, with an
//  argument "out vec3 dg" returning (d2n/dx2, d2n/dy2, d2n/dxy)
//
//  vec3 dg0, dg1, dg2;
//  vec3 dw2 = 48.0 * w2 * gdotx;
//  // d2n/dx2 and d2n/dy2
//  dg0.xy = dw2.x * x0*x0 - 8.0 * w3.x * (2.0*g0*x0 + gdotx.x);
//  dg1.xy = dw2.y * x1*x1 - 8.0 * w3.y * (2.0*g1*x1 + gdotx.y);
//  dg2.xy = dw2.z * x2*x2 - 8.0 * w3.z * (2.0*g2*x2 + gdotx.z);
//  // d2n/dxy
//  dg0.z = dw2.x * x0.x*x0.y - 8.0 * w3.x * dot(g0, x0.yx);
//  dg1.z = dw2.y * x1.x*x1.y - 8.0 * w3.y * dot(g1, x1.yx);
//  dg2.z = dw2.z * x2.x*x2.y - 8.0 * w3.z * dot(g2, x2.yx);
//  dg = 10.9 * (dg0 + dg1 + dg2);

  // Scale the return value to fit nicely into the range [-1,1]
  return 10.9 * n;
}
```

10.2 3-D noise

For brevity and clarity, this version of the source code has a shortened header and omits the options to use the rotated simplex grid and to use the faster rotation algorithm. It also includes commented-out code for the second order derivatives. In all other respects, the code is equivalent to the source code in the GLSL files of the supplementary material and in the Github repository.

```
//
// psrdnoise3.glsl
//
// Version: 2021-12-02
// Authors: Stefan Gustavson (stefan.gustavson@liu.se)
//          and Ian McEwan (ijm567@gmail.com)
//
// Published under the MIT license, see:
// https://opensource.org/licenses/MIT
// (In short: free for any use, but give credit to the authors,
// and pass the license on to any derivative works you create.)
//
// Copyright (c) 2021 Stefan Gustavson and Ian McEwan.
//

// Permutation polynomial for the hash value
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)
{
  // Transformation matrices for the axis-aligned simplex grid
  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);
  vec3 uvw;
  // Transform to simplex space (tetrahedral grid)
  uvw = M * x;

  // Determine which simplex we're in, i0 is the "base corner"
  vec3 i0 = floor(uvw);
  vec3 f0 = fract(uvw); // coords within "skewed cube"

  // To determine which simplex corners are closest, rank order the
  // magnitudes of u,v,w, resolving ties in priority order u,v,w,
  // and traverse the corners from largest to smallest magnitude.
  // o1, o2 are offsets in simplex space to the 2nd and 3rd corners.
  vec3 g_ = step(f0.xyx, f0.yzz); // Makes comparison "less-than"
  vec3 l_ = 1.0 - g_;             // complement: "greater-or-equal"
  vec3 g = vec3(l_.z, g_.xy);
  vec3 l = vec3(l_.xy, g_.z);
  vec3 o1 = min( g, l );
  vec3 o2 = max( g, l );
```

```
// Enumerate the remaining simplex corners
vec3 i1 = i0 + o1;
vec3 i2 = i0 + o2;
vec3 i3 = i0 + vec3(1.0);

vec3 v0, v1, v2, v3;
// Transform the corners back to texture space
v0 = Mi * i0;
v1 = Mi * i1;
v2 = Mi * i2;
v3 = Mi * i3;
// Compute vectors to each of the simplex corners
vec3 x0 = x - v0;
vec3 x1 = x - v1;
vec3 x2 = x - v2;
vec3 x3 = x - v3;

// Wrap to periods and transform back to simplex space
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);
// Wrap to periods where specified
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);
  // Transform back and fix rounding errors
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);
}

// Compute one pseudo-random hash value for each corner
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 ));

// Compute gradients from a Fibonacci spiral on the unit sphere
vec4 theta = hash * 3.883222077;  // 2*pi/golden ratio
vec4 sz = hash * -0.006920415 + 0.996539792; // 1-2*(hash+0.5)/289
vec4 psi = hash * 0.108705628 ; // 10*pi/289, avoids correlation

vec4 Ct = cos(theta);
vec4 St = sin(theta);
vec4 sz_prime = sqrt( 1.0 - sz*sz ); // s is on a fib-sphere

vec4 gx, gy, gz;
// Rotate g_i by alpha around a pseudo-random orthogonal axis.
// This is not the fastest algorithm, but it has g=s for alpha=0
// and allows a strong dynamic speedup when alpha==0.0.
if(alpha != 0.0) {
  vec4 px = Ct * sz_prime;      // px = sx
  vec4 py = St * sz_prime;      // py = sy
  vec4 pz = sz;

  vec4 Sp = sin(psi);           // q' from psi on equator
  vec4 Cp = cos(psi);
  vec4 Ctp = St*Sp - Ct*Cp; // q=(rotate(cross(s,n),dot(s,n))(q')
  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)); // psi, alpha in different planes
    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; // when alpha=0, use s(=p) for g
    gy = St * sz_prime;
    gz = sz;
  }

  // Reorganize for dot products below
  vec3 g0 = vec3(gx.x, gy.x, gz.x);
  vec3 g1 = vec3(gx.y, gy.y, gz.y);
  vec3 g2 = vec3(gx.z, gy.z, gz.z);
  vec3 g3 = vec3(gx.w, gy.w, gz.w);

  // Radial decay with distance from each simplex corner
  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;

  // The value of the linear ramp from each of the corners
  vec4 gdotx = vec4(dot(g0,x0), dot(g1,x1), dot(g2,x2), dot(g3,x3));

  // Multiply by the radial decay and sum up the noise value
  float n = dot(w3, gdotx);

  // Compute the 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);

//  Second order derivatives can be computed like this, with
//  "out vec3 dg" returning (d2n/dx2, d2n/dy2, d2n/dz2) and
//  "out vec3 dg2" returning (d2n/dxy, d2n/dyz, d2n/dxz)
//
//  vec4 dw2 = 24.0 * w * gdotx;
//  vec3 dga0 = dw2.x*x0*x0 - 6.0* w2.x*(gdotx.x + 2.0 * g0 * x0);
//  vec3 dga1 = dw2.y*x1*x1 - 6.0* w2.y*(gdotx.y + 2.0 * g1 * x1);
//  vec3 dga2 = dw2.z*x2*x2 - 6.0* w2.z*(gdotx.z + 2.0 * g2 * x2);
//  vec3 dga3 = dw2.w*x3*x3 - 6.0* w2.w*(gdotx.w + 2.0 * g3 * x3);
//  dg = 35.0*(dga0+dga1+dga2+dga3); // (d2n/dx2, d2n/dy2, d2n/dz2)
//  vec3 dgb0 = dw2.x*x0*x0.yzx - 6.0*w2.x*(g0*x0.yzx + g0.yzx*x0);
//  vec3 dgb1 = dw2.y*x1*x1.yzx - 6.0*w2.y*(g1*x1.yzx + g1.yzx*x1);
//  vec3 dgb2 = dw2.z*x2*x2.yzx - 6.0*w2.z*(g2*x2.yzx + g2.yzx*x2);
//  vec3 dgb3 = dw2.w*x3*x3.yzx - 6.0*w2.w*(g3*x3.yzx + g3.yzx*x3);
//  dg2 = 39.5*(dgb0+dgb1+dgb2+dgb3); // (d2n/dxy, d2n/dyz, d2n/dxz)

  // Scale the return value to fit nicely into the range [-1,1]
  return 39.5 * n;
}
```