Direct computational noise in GLSL
Supplementary material

Ian McEwan[1], David Sheets[1], Stefan Gustavson[2] and Mark Richardson[1]

[1]Ashima Research, 600 S. Lake Ave., Suite 104, Pasadena CA 91106, USA
[2]Media and Information Technology, Linköping University, Sweden

1 Permutation polynomials

A permutation polynomial is an automorphism, a bijective map from a set onto itself, that shuffles the elements of a finite ring and has the form:

$$f\colon X \to X \quad x \mapsto \sum_{i=0}^{L} a_i x^i$$

Quadratic permutation polynomials (i.e. $L = 2$) can be easily constructed for many rings: Assume $X = \mathbf{Z}/N\mathbf{Z}$ (the finite ring of size N) and $N = \prod_i p_i^{k_i}$ where p_i are prime numbers (the prime factorization of N) then the quadratic $f(x) = a_2x^2 + a_1x + a_0$ will be a permutation polynomial of X if

$$\begin{array}{rl} \text{for some } i & \left\{ \begin{array}{l} a_2 \bmod p_i = 0 \\ a_2 \bmod p_i^{k_i} \neq 0 \end{array} \right. \Rightarrow k_i > 1 \\ \text{for all } j & a_1 \bmod p_j \neq 0 \end{array}$$

For example for $\mathbf{Z}/9\mathbf{Z}$ (where $N = 3^2$) and $\mathbf{Z}/12\mathbf{Z}$ (where $N = 2^2 \times 3$) both admit the polynomial $6x^2 + x$ giving the respective permutations

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 7 & 8 & 3 & 1 & 2 & 6 & 4 & 5 \end{pmatrix} \text{and} \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 0 & 7 & 2 & 9 & 4 & 11 & 6 & 1 & 8 & 3 & 10 & 5 \end{pmatrix}$$

The second of these permutations is obviously not a very good choice for a pseudo-random shuffling, because every second number stays in place. Far from all polynomial permutations have good shuffling poperties, so care needs to be taken to pick a suitable combination of a ring and a polynomial.

Another potential problem with this approach is that the number of possible permutation tables, and therefore the number of possible distinct noise fields, is vastly reduced from any possible shuffling (which is $N!$ permutations for each $\mathbf{Z}/N\mathbf{Z}$) to just those tables that can be described by one of these polynomials.

However, there are still plenty of options for our purposes. Looking at a reasonable range for N, there are 1138 rings with $N < 2896$ that have a $k_i > 1$, and each of these will admit many polynomials.

We do not claim that our choice of the polynomial $(34x^2 + x) \bmod 289$ is the final word. It seems to be good enough for 2D and 3D noise, but for 4D noise there may be others that are better. Some visual artifacts seem to be present in our current implementation of 4D simplex noise. These artifacts are possibly due to an insufficiently randomized shuffling.

2 3-cube to 4-cross polytope mapping

As presented in the main article, the mapping from a 3-D cube to a 4-D surface to create evenly distributed normals in 4-D space is:

$$\begin{aligned}
\textbf{4-D:}\quad & x_0, y_0, z_0 \in [-1, 1], \quad w = 1.5 - |x_0| - |y_0| - |z_0| \\
& \textbf{if } w > 0 \textbf{ then } x = x_0,\ y = y_0,\ z = z_0 \\
& \textbf{else } x = x_0 - \operatorname{sign}(x_0),\ y = y_0 - \operatorname{sign}(y_0),\ z = z_0 - \operatorname{sign}(z_0)
\end{aligned}$$

A graphical illustration of one octant of the 3D part of the mapping is presented in Figure 1. The blue region in the octant $x > 0, y > 0, z > 0$ where $|x| + |y| + |z| < 1.5$ maps to the hyper-face $x, y, z, w > 0$, while the orange region where $|x| + |y| + |z| > 1.5$ maps to the opposite hyper-face $x, y, z, w < 0$. All eight octants are bisected in the same manner, and the outer region of each octant is shifted into the diagonally opposing octant in (x, y, z) and negative w. All the 4-D points thus created satisfy $|x| + |y| + |z| + |w| = 1.5$, which means they are on the surface of the 4-D cross polytope of radius 1.5, but the mapping is slightly incomplete. Each of the 16 facets of a complete 4-D cross polytope is a tetrahedron and not the slightly truncated tetrahedron created by bisecting a 3-D cube. The corners $1 < |x| \leq 1.5$, $1 < |y| \leq 1.5$ and $1 < |z| \leq 1.5$ are cut off, while the corners $1 < |w| \leq 1.5$ are not. Some of the 4-D directions will be missing from the set of gradients. However, this mapping defect is acceptable for our purposes and seems to have no visible influence on the noise field.

3 Rank ordering

Rank ordering involves pair-wise comparisons between each element and every other element, to find out how many of the other elements are larger. To properly handle the case where elements can be equal, comparisons should be performed with consideration to the existing order, and preference should be given in a consequent manner to either the first or the second element in case of a tie. A concrete implementation can solve this and reduce the number of comparisons by half by using the complement of the comparison $x_i < x_j$ to generate the corresponding comparison $x_j \leq x_i$. A rank ordering of a vector

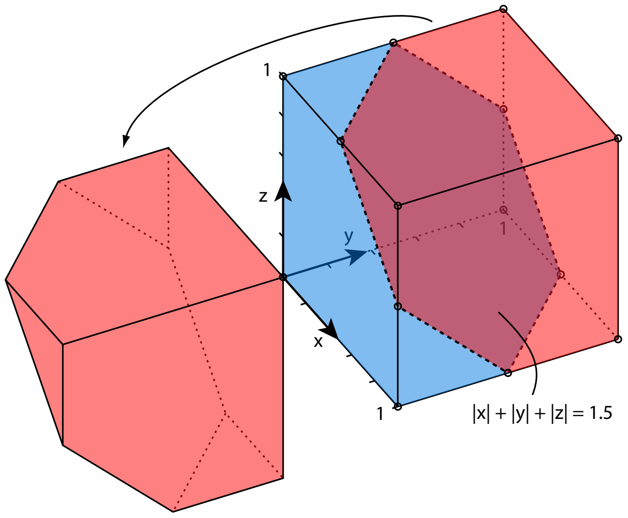

Figure 1: Mapping from a 3D cube to the (partial) boundary of a 4-D cross polytope.

$\mathbf{x} = \{x_1, x_2, \cdots, x_N\}$ is performed by the following algorithm:

```
$\mathbf{r} = \{r_1, r_2, \cdots, r_N\} := \{0, 0, \ldots, 0\}$
**for** $i$ **in** $[1, N-1]$
    **for** $j$ **in** $[i+1, N]$
        **if** $x_i > x_j$ **then** $r_i := r_i + 1$ **else** $r_j := r_j + 1$
    **end for** $j$
**end for** $i$
```

After execution, the vector $\mathbf{r}$ contains the rank index of each corresponding element in $\mathbf{x}$, such that $\{x_{r_1}, x_{r_2}, \cdots, x_{r_N}\}$ would be a sorted vector. This algorithm is $\mathcal{O}(N^2)$, which is not particularly good for a general purpose sorting algorithm, but we are only sorting a few elements, four in the case of 4-D simplex noise, so asymptotic complexity is of no importance. What matters more is that the algorithm is simple and fast for short vectors, and that the comparisons and the rank vector increments can be performed in any order, even in parallel, which maps nicely to GLSL vector operations.

4 Arbitrary constants

The final noise values are scaled to cover a reasonable portion of the range $[-1, 1]$. Scaling constants were chosen in an empirical fashion, by simply looking at the min and max values across a very large number of evaluations. Some additional tricks in the code also deserve mentioning. The mapping from $(N-1)$-D to N-D requires generation of a regular distribution of points in an $(N-1)$-D cube from a single integer in the range 0 to 288. Our choice for 2-D was to wrap that range several times to a smaller range of 0 to 40, creating 41 unique gradients. 41 gradients are more than enough for a good visual appearance, 41 has no common prime factor with 289, which creates some additional decorrelation, and it is reasonably close to an even divisor of 289, thereby avoiding strong anisotropy in the gradients. For 3-D, we chose to generate a square of $7 \times 7 = 49$ points, which again has no common prime factors with 289 and is reasonably close to an even divisor of 289. The obvious choice of 17×17 points was tried but abandoned, as it created visible correlation artifacts in the noise field. In the 4-D case, a cube of $7 \times 7 \times 6$ points turned out to be adequate. These choices are all somewhat arbitrary, and we do not claim that they are optimal, but they seem good enough.

5 Source code

The recommended way of obtaining the source code is through the Git repository `git@github.com:ashima/webgl-noise`, reachable also by a web interface at `http://github.com/ashima/webgl-noise/`. The code in that repository will be maintained and updated. The listings here are frozen snapshots of the code at the time of publication, provided mainly as a backup for posterity.

5.1 2D simplex noise

```
// Description : Array and textureless GLSL 2D simplex noise function.
//      Author : Ian McEwan, Ashima Arts.
//     Lastmod : 20110403 (stegu)
//     License : Copyright (C) 2011 Ashima Arts. All rights reserved.
//               Distributed under the MIT License. See LICENSE file.

vec3 permute(vec3 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec3 taylorInvSqrt(vec3 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

float snoise(vec2 v) {
  const vec2 C = vec2(0.211324865405187134, // (3.0-sqrt(3.0))/6.0;
                      0.366025403784438597); // 0.5*(sqrt(3.0)-1.0);
// First corner
  vec2 i  = floor(v + dot(v, C.yy) );
  vec2 x0 = v -   i + dot(i, C.xx);
// Other corners
  vec2 i1;
  i1.x = step( x0.y, x0.x ); // 1.0 if x0.x > x0.y, else 0.0
  i1.y = 1.0 - i1.x;
  // x0 = x0 - 0.0 + 0.0 * C.xx ;
  // x1 = x0 - i1 + 1.0 * C.xx ;
  // x2 = x0 - 1.0 + 2.0 * C.xx ;
  vec4 x12 = x0.xyxy + vec4( C.xx, C.xx * 2.0 - 1.0);
  x12.xy -= i1;
// Permutations
  i = mod(i, 289.0); // Avoid truncation in polynomial evaluation
  vec3 p = permute( permute( i.y + vec3(0.0, i1.y, 1.0 ))
    + i.x + vec3(0.0, i1.x, 1.0 ));
// Circularly symmetric blending kernel
  vec3 m = max(0.5 - vec3(dot(x0,x0), dot(x12.xy,x12.xy), dot(x12.zw,x12
      .zw)), 0.0);
  m = m*m ;
  m = m*m ;
// Gradients from 41 points on a line, mapped onto a diamond
  vec3 x = fract(p * (1.0 / 41.0)) * 2.0 - 1.0 ;
  vec3 gy = abs(x) - 0.5 ;
  vec3 ox = floor(x + 0.5); // round(x) is a GLSL 1.30 feature
  vec3 gx = x - ox;
// Normalise gradients implicitly by scaling m
  m *= taylorInvSqrt( gx*gx + gy*gy );
// Compute final noise value at P
  vec3 g;
  g.x  = gx.x  * x0.x  + gy.x  * x0.y;
  g.yz = gx.yz * x12.xz + gy.yz * x12.yw;
  return 130.0 * dot(m, g);
}
```

5.2 3D simplex noise

```
// Description : Array and textureless GLSL 3D simplex noise.
//      Author : Ian McEwan, Ashima Arts.
//     License : Copyright (C) 2011 Ashima Arts. All rights reserved.
//               Distributed under the MIT License. See LICENSE file.

vec4 permute(vec4 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

float snoise(vec3 v)
{
  const vec2  C = vec2(1.0/6.0, 1.0/3.0) ;
  const vec4  D = vec4(0.0, 0.5, 1.0, 2.0);

// First corner
  vec3 i  = floor(v + dot(v, C.yyy) );
  vec3 x0 =   v - i + dot(i, C.xxx) ;

// Other corners
  vec3 g = step(x0.yzx, x0.xyz);
  vec3 l = 1.0 - g;
  vec3 i1 = min( g.xyz, l.zxy );
  vec3 i2 = max( g.xyz, l.zxy );

  //  x0 = x0 - 0. + 0.0 * C
  vec3 x1 = x0 - i1 + 1.0 * C.xxx;
  vec3 x2 = x0 - i2 + 2.0 * C.xxx;
  vec3 x3 = x0 - 1. + 3.0 * C.xxx;

// Permutations
  i = mod(i, 289.0 );
  vec4 p = permute( permute( permute(
             i.z + vec4(0.0, i1.z, i2.z, 1.0 ))
           + i.y + vec4(0.0, i1.y, i2.y, 1.0 ))
           + i.x + vec4(0.0, i1.x, i2.x, 1.0 ));

// Gradients from 7x7 points over a square, mapped onto an octahedron
  float n_ = 1.0/7.0;
  vec3  ns = n_ * D.wyz - D.xzx;
  vec4 j = p - 49.0 * floor(p * ns.z *ns.z);  //  mod(p,7*7)

  vec4 x_ = floor(j * ns.z);
  vec4 y_ = floor(j - 7.0 * x_ );    // mod(j,7)

  vec4 x = x_ *ns.x + ns.yyyy;
  vec4 y = y_ *ns.x + ns.yyyy;
  vec4 h = 1.0 - abs(x) - abs(y);

  vec4 b0 = vec4( x.xy, y.xy );
  vec4 b1 = vec4( x.zw, y.zw );

  //vec4 s0 = vec4(lessThan(b0,0.0))*2.0 - 1.0;
  //vec4 s1 = vec4(lessThan(b1,0.0))*2.0 - 1.0;
  vec4 s0 = floor(b0)*2.0 + 1.0;
  vec4 s1 = floor(b1)*2.0 + 1.0;
  vec4 sh = -step(h, vec4(0.0));

  vec4 a0 = b0.xzyw + s0.xzyw*sh.xxyy ;
  vec4 a1 = b1.xzyw + s1.xzyw*sh.zzww ;
  vec3 p0 = vec3(a0.xy,h.x);
  vec3 p1 = vec3(a0.zw,h.y);
  vec3 p2 = vec3(a1.xy,h.z);
  vec3 p3 = vec3(a1.zw,h.w);

//Normalise gradients
  vec4 norm = taylorInvSqrt(vec4(dot(p0,p0), dot(p1,p1), dot(p2, p2),
      dot(p3,p3)));
  p0 *= norm.x;
  p1 *= norm.y;
  p2 *= norm.z;
  p3 *= norm.w;

// Mix final noise value
  vec4 m = max(0.6 - vec4(dot(x0,x0), dot(x1,x1), dot(x2,x2), dot(x3,x3)
      ), 0.0);
  m = m * m;
  return 42.0 * dot( m*m, vec4( dot(p0,x0), dot(p1,x1),
                                dot(p2,x2), dot(p3,x3) ) );
}
```

5.3 4D simplex noise

```
// Description : Array and textureless GLSL 4D simplex noise.
//      Author : Ian McEwan, Ashima Arts.
//     License : Copyright (C) 2011 Ashima Arts. All rights reserved.
//               Distributed under the MIT License. See LICENSE file.

vec4 permute(vec4 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

float permute(float x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

float taylorInvSqrt(float r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

vec4 grad4(float j, vec3 ip) {
  vec4 p,s;
  p.xyz = floor( fract (vec3(j) * ip) * 7.0) * ip.z - 1.0;
  p.w = 1.5 - dot(abs(p.xyz), vec3(1.0));
  s = vec4(lessThan(p, vec4(0.0)));
  p.xyz = p.xyz + (s.xyz * 2.0 - 1.0) * s.www;
  return p;
}

float snoise(vec4 v)
{
  const vec2  C = vec2( 0.138196601125010504,  // (5 - sqrt(5))/20  G4
                        0.309016994374947451); // (sqrt(5) - 1)/4   F4
// First corner
  vec4 i  = floor(v + dot(v, C.yyyy) );
  vec4 x0 = v -   i + dot(i, C.xxxx);

// Other corners
// Rank sorting by Bill Licea-Kane, AMD (formerly ATI)
  vec4 i0;

  vec3 isX = step( x0.yzw, x0.xxx );
  vec3 isYZ = step( x0.zww, x0.yyz );
//  i0.x = dot( isX, vec3( 1.0 ) );
  i0.x = isX.x + isX.y + isX.z;
  i0.yzw = 1.0 - isX;

  //  i0.y += dot( isYZ.xy, vec2( 1.0 ) );
  i0.y += isYZ.x + isYZ.y;
  i0.zw += 1.0 - isYZ.xy;

  i0.z += isYZ.z;
  i0.w += 1.0 - isYZ.z;

  // i0 now contains the unique values 0,1,2,3 in each channel
  vec4 i3 = clamp( i0, 0.0, 1.0 );
  vec4 i2 = clamp( i0-1.0, 0.0, 1.0 );
  vec4 i1 = clamp( i0-2.0, 0.0, 1.0 );

  //  x0 = x0 - 0.0 + 0.0 * C
  vec4 x1 = x0 - i1 + 1.0 * C.xxxx;
  vec4 x2 = x0 - i2 + 2.0 * C.xxxx;
  vec4 x3 = x0 - i3 + 3.0 * C.xxxx;
  vec4 x4 = x0 - 1.0 + 4.0 * C.xxxx;

// Permutations
  i = mod(i, 289.0);
  float j0 = permute( permute( permute( permute(i.w)
  + i.z) + i.y) + i.x);
  vec4 j1 = permute( permute( permute( permute (
             i.w + vec4(i1.w, i2.w, i3.w, 1.0 ))
           + i.z + vec4(i1.z, i2.z, i3.z, 1.0 ))
           + i.y + vec4(i1.y, i2.y, i3.y, 1.0 ))
           + i.x + vec4(i1.x, i2.x, i3.x, 1.0 ));

     // Gradients by 7*7*6 points in a cube mapped to a hyper-
         octahedron
// 7*7*6 = 294, which is close to the ring size 17*17 = 289.
  vec3 ip = vec4(1.0/294.0, 1.0/49.0, 1.0/7.0);
  vec4 p0 = grad4(j0,   ip);
  vec4 p1 = grad4(j1.x, ip);
  vec4 p2 = grad4(j1.y, ip);
  vec4 p3 = grad4(j1.z, ip);
  vec4 p4 = grad4(j1.w, ip);

// Normalise gradients
  vec4 norm = taylorInvSqrt(vec4(dot(p0,p0), dot(p1,p1), dot(p2, p2),
      dot(p3,p3)));
  p0 *= norm.x;
  p1 *= norm.y;
  p2 *= norm.z;
  p3 *= norm.w;
  p4 *= taylorInvSqrt(dot(p4,p4));

// Mix contributions from the five corners
  vec3 m0 = max(0.6 - vec3(dot(x0,x0), dot(x1,x1), dot(x2,x2)), 0.0);
  vec2 m1 = max(0.6 - vec2(dot(x3,x3), dot(x4,x4)            ), 0.0);
  m0 = m0 * m0;
  m1 = m1 * m1;
  return 49.0 * ( dot(m0*m0, vec3( dot( p0, x0 ), dot( p1, x1 ), dot( p2
      , x2 )))
              + dot(m1*m1, vec2( dot( p3, x3 ), dot( p4, x4 ) ) ) ) ;

}
```

5.4 2D classic Perlin noise

```
// GLSL textureless classic 3D noise "cnoise",
// with an RSL-style periodic variant "pnoise".
// Author:  Stefan Gustavson (stefan.gustavson@liu.se)
// Copyright (c) 2011 Stefan Gustavson. All rights reserved.
// This code is licensed under the terms of the MIT license.
// See the file LICENSING for details.

vec4 permute(vec4 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

vec2 fade(vec2 t) {
  return t*t*t*(t*(t*6.0-15.0)+10.0);
}

// Classic Perlin noise
float cnoise(vec2 P)
{
  // Integer and fractional coords for all four corners
  vec4 Pi = floor(P.xyxy) + vec4(0.0, 0.0, 1.0, 1.0);
  vec4 Pf = fract(P.xyxy) - vec4(0.0, 0.0, 1.0, 1.0);
  Pi = mod(Pi, 289.0); // Avoid truncation effects in permutation
  vec4 ix = Pi.xzxz;
  vec4 iy = Pi.yyww;
  vec4 fx = Pf.xzxz;
  vec4 fy = Pf.yyww;

  vec4 i = permute(permute(ix) + iy);

  // Gradients from 41 points on a line mapped to a diamond
  vec4 gx = 2.0 * fract(i * (1.0 / 41.0)) - 1.0 ;
  vec4 gy = abs(gx) - 0.5 ;
  vec4 tx = floor(gx + 0.5);
  gx = gx - tx;

  vec2 g00 = vec2(gx.x,gy.x);
  vec2 g10 = vec2(gx.y,gy.y);
  vec2 g01 = vec2(gx.z,gy.z);
  vec2 g11 = vec2(gx.w,gy.w);

  // Normalise gradients
  vec4 norm = taylorInvSqrt(vec4(dot(g00, g00), dot(g01, g01), dot(g10,
      g10), dot(g11, g11)));
  g00 *= norm.x;
  g01 *= norm.y;
  g10 *= norm.z;
  g11 *= norm.w;

  // Extrapolation from the four corners
  float n00 = dot(g00, vec2(fx.x, fy.x));
  float n10 = dot(g10, vec2(fx.y, fy.y));
  float n01 = dot(g01, vec2(fx.z, fy.z));
  float n11 = dot(g11, vec2(fx.w, fy.w));

  // Interpolation to compute final noise value
  vec2 fade_xy = fade(Pf.xy);
  vec2 n_x = mix(vec2(n00, n01), vec2(n10, n11), fade_xy.x);
  float n_xy = mix(n_x.x, n_x.y, fade_xy.y);
  return 2.3 * n_xy;
}
```

```
// Classic Perlin noise, periodic variant
float pnoise(vec2 P, vec2 rep)
{
  vec4 Pi = floor(P.xyxy) + vec4(0.0, 0.0, 1.0, 1.0);
  vec4 Pf = fract(P.xyxy) - vec4(0.0, 0.0, 1.0, 1.0);
  Pi = mod(Pi, rep.xyxy); // Creates noise with explicit period
  Pi = mod(Pi, 289.0); // The rest is exactly the same as "cnoise"
  vec4 ix = Pi.xzxz;
  vec4 iy = Pi.yyww;
  vec4 fx = Pf.xzxz;
  vec4 fy = Pf.yyww;

  vec4 i = permute(permute(ix) + iy);

  vec4 gx = 2.0 * fract(i / 41.0) - 1.0 ;
  vec4 gy = abs(gx) - 0.5 ;
  vec4 tx = floor(gx + 0.5);
  gx = gx - tx;

  vec2 g00 = vec2(gx.x,gy.x);
  vec2 g10 = vec2(gx.y,gy.y);
  vec2 g01 = vec2(gx.z,gy.z);
  vec2 g11 = vec2(gx.w,gy.w);

  vec4 norm = taylorInvSqrt(vec4(dot(g00, g00), dot(g01, g01), dot(g10,
      g10), dot(g11, g11)));
  g00 *= norm.x;
  g01 *= norm.y;
  g10 *= norm.z;
  g11 *= norm.w;

  float n00 = dot(g00, vec2(fx.x, fy.x));
  float n10 = dot(g10, vec2(fx.y, fy.y));
  float n01 = dot(g01, vec2(fx.z, fy.z));
  float n11 = dot(g11, vec2(fx.w, fy.w));

  vec2 fade_xy = fade(Pf.xy);
  vec2 n_x = mix(vec2(n00, n01), vec2(n10, n11), fade_xy.x);
  float n_xy = mix(n_x.x, n_x.y, fade_xy.y);
  return 2.3 * n_xy;
}
```

5.5 3D classic Perlin noise

```
// GLSL textureless classic 3D noise "cnoise",
// with an RSL-style periodic variant "pnoise".
// Author:  Stefan Gustavson (stefan.gustavson@liu.se)
// Copyright (c) 2011 Stefan Gustavson. All rights reserved.
// This code is licensed under the terms of the MIT license.
// See the file LICENSING for details.

vec4 permute(vec4 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

vec3 fade(vec3 t) {
  return t*t*t*(t*(t*6.0-15.0)+10.0);
}
```

```
// Classic Perlin noise
float cnoise(vec3 P)
{
  vec3 Pi0 = floor(P); // Integer part for indexing
  vec3 Pi1 = Pi0 + vec3(1.0); // Integer part + 1
  Pi0 = mod(Pi0, 289.0); // Avoid truncation effects
  Pi1 = mod(Pi1, 289.0);
  vec3 Pf0 = fract(P); // Fractional part for interpolation
  vec3 Pf1 = Pf0 - vec3(1.0); // Fractional part - 1.0
  vec4 ix = vec4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
  vec4 iy = vec4(Pi0.yy, Pi1.yy);
  vec4 iz0 = Pi0.zzzz;
  vec4 iz1 = Pi1.zzzz;

  vec4 ixy = permute(permute(ix) + iy);
  vec4 ixy0 = permute(ixy + iz0);
  vec4 ixy1 = permute(ixy + iz1);

  vec4 gx0 = ixy0 * (1.0 / 7.0);
  vec4 gy0 = fract(floor(gx0) * (1.0 / 7.0)) - 0.5;
  gx0 = fract(gx0);
  vec4 gz0 = vec4(0.5) - abs(gx0) - abs(gy0);
  vec4 sz0 = step(gz0, vec4(0.0));
  gx0 -= sz0 * (step(0.0, gx0) - 0.5);
  gy0 -= sz0 * (step(0.0, gy0) - 0.5);

  // Gradients: 7x7 points over a square mapped to an octahedron
  vec4 gx1 = ixy1 * (1.0 / 7.0);
  vec4 gy1 = fract(floor(gx1) * (1.0/ 7.0)) - 0.5;
  gx1 = fract(gx1);
  vec4 gz1 = vec4(0.5) - abs(gx1) - abs(gy1);
  vec4 sz1 = step(gz1, vec4(0.0));
  gx1 -= sz1 * (step(0.0, gx1) - 0.5);
  gy1 -= sz1 * (step(0.0, gy1) - 0.5);

  vec3 g000 = vec3(gx0.x,gy0.x,gz0.x);
  vec3 g100 = vec3(gx0.y,gy0.y,gz0.y);
  vec3 g010 = vec3(gx0.z,gy0.z,gz0.z);
  vec3 g110 = vec3(gx0.w,gy0.w,gz0.w);
  vec3 g001 = vec3(gx1.x,gy1.x,gz1.x);
  vec3 g101 = vec3(gx1.y,gy1.y,gz1.y);
  vec3 g011 = vec3(gx1.z,gy1.z,gz1.z);
  vec3 g111 = vec3(gx1.w,gy1.w,gz1.w);

  // Normalise gradients
  vec4 norm0 = taylorInvSqrt(vec4(dot(g000, g000), dot(g010, g010), dot(
      g100, g100), dot(g110, g110)));
  g000 *= norm0.x;
  g010 *= norm0.y;
  g100 *= norm0.z;
  g110 *= norm0.w;
  vec4 norm1 = taylorInvSqrt(vec4(dot(g001, g001), dot(g011, g011), dot(
      g101, g101), dot(g111, g111)));
  g001 *= norm1.x;
  g011 *= norm1.y;
  g101 *= norm1.z;
  g111 *= norm1.w;

  // Extrapolate ramps from all eight corners
  float n000 = dot(g000, Pf0);
  float n100 = dot(g100, vec3(Pf1.x, Pf0.yz));
  float n010 = dot(g010, vec3(Pf0.x, Pf1.y, Pf0.z));
  float n110 = dot(g110, vec3(Pf1.xy, Pf0.z));
  float n001 = dot(g001, vec3(Pf0.xy, Pf1.z));
  float n101 = dot(g101, vec3(Pf1.x, Pf0.y, Pf1.z));
  float n011 = dot(g011, vec3(Pf0.x, Pf1.yz));
  float n111 = dot(g111, Pf1);
  // Interpolate to compute final noise value
  vec3 fade_xyz = fade(Pf0);
  vec4 n_z = mix(vec4(n000, n100, n010, n110), vec4(n001, n101, n011,
      n111), fade_xyz.z);
  vec2 n_yz = mix(n_z.xy, n_z.zw, fade_xyz.y);
  float n_xyz = mix(n_yz.x, n_yz.y, fade_xyz.x);
  return 2.2 * n_xyz;
}

// Classic Perlin noise, periodic variant
float pnoise(vec3 P, vec3 rep)
{
  vec3 Pi0 = mod(floor(P), rep); // Integer part, modulo period
  vec3 Pi1 = mod(Pi0 + vec3(1.0), rep); // Integer part + 1, mod period
  Pi0 = mod(Pi0, 289.0); // Avoid truncation effects
  Pi1 = mod(Pi1, 289.0);
  vec3 Pf0 = fract(P); // Fractional part for interpolation
  vec3 Pf1 = Pf0 - vec3(1.0); // Fractional part - 1.0
  vec4 ix = vec4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
  vec4 iy = vec4(Pi0.yy, Pi1.yy);
  vec4 iz0 = Pi0.zzzz;
  vec4 iz1 = Pi1.zzzz;

  vec4 ixy = permute(permute(ix) + iy);
  vec4 ixy0 = permute(ixy + iz0);
  vec4 ixy1 = permute(ixy + iz1);

  vec4 gx0 = ixy0 * (1.0 / 7.0);
  vec4 gy0 = fract(floor(gx0) * (1.0 / 7.0)) - 0.5;
  gx0 = fract(gx0);
  vec4 gz0 = vec4(0.5) - abs(gx0) - abs(gy0);
  vec4 sz0 = step(gz0, vec4(0.0));
  gx0 -= sz0 * (step(0.0, gx0) - 0.5);
  gy0 -= sz0 * (step(0.0, gy0) - 0.5);

  vec4 gx1 = ixy1 * (1.0 / 7.0);
  vec4 gy1 = fract(floor(gx1) * (1.0 / 7.0)) - 0.5;
  gx1 = fract(gx1);
  vec4 gz1 = vec4(0.5) - abs(gx1) - abs(gy1);
  vec4 sz1 = step(gz1, vec4(0.0));
  gx1 -= sz1 * (step(0.0, gx1) - 0.5);
  gy1 -= sz1 * (step(0.0, gy1) - 0.5);

  vec3 g000 = vec3(gx0.x,gy0.x,gz0.x);
  vec3 g100 = vec3(gx0.y,gy0.y,gz0.y);
  vec3 g010 = vec3(gx0.z,gy0.z,gz0.z);
  vec3 g110 = vec3(gx0.w,gy0.w,gz0.w);
  vec3 g001 = vec3(gx1.x,gy1.x,gz1.x);
  vec3 g101 = vec3(gx1.y,gy1.y,gz1.y);
  vec3 g011 = vec3(gx1.z,gy1.z,gz1.z);
  vec3 g111 = vec3(gx1.w,gy1.w,gz1.w);

  vec4 norm0 = taylorInvSqrt(vec4(dot(g000, g000), dot(g010, g010), dot(
      g100, g100), dot(g110, g110)));
  g000 *= norm0.x;
  g010 *= norm0.y;
  g100 *= norm0.z;
  g110 *= norm0.w;
  vec4 norm1 = taylorInvSqrt(vec4(dot(g001, g001), dot(g011, g011), dot(
      g101, g101), dot(g111, g111)));
  g001 *= norm1.x;
  g011 *= norm1.y;
  g101 *= norm1.z;
  g111 *= norm1.w;

  float n000 = dot(g000, Pf0);
  float n100 = dot(g100, vec3(Pf1.x, Pf0.yz));
  float n010 = dot(g010, vec3(Pf0.x, Pf1.y, Pf0.z));
  float n110 = dot(g110, vec3(Pf1.xy, Pf0.z));
  float n001 = dot(g001, vec3(Pf0.xy, Pf1.z));
  float n101 = dot(g101, vec3(Pf1.x, Pf0.y, Pf1.z));
  float n011 = dot(g011, vec3(Pf0.x, Pf1.yz));
  float n111 = dot(g111, Pf1);

  vec3 fade_xyz = fade(Pf0);
  vec4 n_z = mix(vec4(n000, n100, n010, n110), vec4(n001, n101, n011,
      n111), fade_xyz.z);
  vec2 n_yz = mix(n_z.xy, n_z.zw, fade_xyz.y);
  float n_xyz = mix(n_yz.x, n_yz.y, fade_xyz.x);
  return 2.2 * n_xyz;
}
```

5.6 4D classic Perlin noise

```
// GLSL textureless classic 4D noise "cnoise",
// with an RSL-style periodic variant "pnoise".
// Author:  Stefan Gustavson (stefan.gustavson@liu.se)
// Copyright (c) 2011 Stefan Gustavson. ALl rights reserved.
// This code is licensed under the terms of the MIT license.
// See the file LICENSING for details.

vec4 permute(vec4 x) {
  return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r) {
  return 1.79284291400159 - 0.85373472095314 * r;
}

vec4 fade(vec4 t) {
  return t*t*t*(t*(t*6.0-15.0)+10.0);
}

// Classic Perlin noise
float cnoise(vec4 P)
{
  vec4 Pi0 = floor(P); // Integer part for indexing
  vec4 Pi1 = Pi0 + 1.0; // Integer part + 1
  Pi0 = mod(Pi0, 289.0); // Avoid truncation effects
  Pi1 = mod(Pi1, 289.0);
  vec4 Pf0 = fract(P); // Fractional part for interpolation
  vec4 Pf1 = Pf0 - 1.0; // Fractional part - 1.0
  vec4 ix = vec4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
  vec4 iy = vec4(Pi0.yy, Pi1.yy);
  vec4 iz0 = vec4(Pi0.zzzz);
  vec4 iz1 = vec4(Pi1.zzzz);
  vec4 iw0 = vec4(Pi0.wwww);
  vec4 iw1 = vec4(Pi1.wwww);

  vec4 ixy = permute(permute(ix) + iy);
  vec4 ixy0 = permute(ixy + iz0);
  vec4 ixy1 = permute(ixy + iz1);
  vec4 ixy00 = permute(ixy0 + iw0);
  vec4 ixy01 = permute(ixy0 + iw1);
  vec4 ixy10 = permute(ixy1 + iw0);
  vec4 ixy11 = permute(ixy1 + iw1);

  // Gradients from 7x7x6 points in a 3-cube mapped to a 4-CP
  vec4 gx00 = ixy00 * (1.0 / 7.0);
  vec4 gy00 = floor(gx00) * (1.0 / 7.0);
```

```
vec4 gz00 = floor(gy00) * (1.0 / 6.0);
gx00 = fract(gx00) - 0.5;
gy00 = fract(gy00) - 0.5;
gz00 = fract(gz00) - 0.5;
vec4 gw00 = vec4(0.75) - abs(gx00) - abs(gy00) - abs(gz00);
vec4 sw00 = step(gw00, vec4(0.0));
gx00 -= sw00 * (step(0.0, gx00) - 0.5);
gy00 -= sw00 * (step(0.0, gy00) - 0.5);

// More gradients needed
vec4 gx01 = ixy01 * (1.0 / 7.0);
vec4 gy01 = floor(gx01) * (1.0 / 7.0);
vec4 gz01 = floor(gy01) * (1.0 / 6.0);
gx01 = fract(gx01) - 0.5;
gy01 = fract(gy01) - 0.5;
gz01 = fract(gz01) - 0.5;
vec4 gw01 = vec4(0.75) - abs(gx01) - abs(gy01) - abs(gz01);
vec4 sw01 = step(gw01, vec4(0.0));
gx01 -= sw01 * (step(0.0, gx01) - 0.5);
gy01 -= sw01 * (step(0.0, gy01) - 0.5);

// And still more gradients
vec4 gx10 = ixy10 * (1.0 / 7.0);
vec4 gy10 = floor(gx10) * (1.0 / 7.0);
vec4 gz10 = floor(gy10) * (1.0 / 6.0;
gx10 = fract(gx10) - 0.5;
gy10 = fract(gy10) - 0.5;
gz10 = fract(gz10) - 0.5;
vec4 gw10 = vec4(0.75) - abs(gx10) - abs(gy10) - abs(gz10);
vec4 sw10 = step(gw10, vec4(0.0));
gx10 -= sw10 * (step(0.0, gx10) - 0.5);
gy10 -= sw10 * (step(0.0, gy10) - 0.5);

// And still more gradients to make 16
vec4 gx11 = ixy11 * (1.0 / 7.0);
vec4 gy11 = floor(gx11) * (1.0 / 7.0);
vec4 gz11 = floor(gy11) * (1.0 / 6.0);
gx11 = fract(gx11) - 0.5;
gy11 = fract(gy11) - 0.5;
gz11 = fract(gz11) - 0.5;
vec4 gw11 = vec4(0.75) - abs(gx11) - abs(gy11) - abs(gz11);
vec4 sw11 = step(gw11, vec4(0.0));
gx11 -= sw11 * (step(0.0, gx11) - 0.5);
gy11 -= sw11 * (step(0.0, gy11) - 0.5);

vec4 g0000 = vec4(gx00.x,gy00.x,gz00.x,gw00.x);
vec4 g1000 = vec4(gx00.y,gy00.y,gz00.y,gw00.y);
vec4 g0100 = vec4(gx00.z,gy00.z,gz00.z,gw00.z);
vec4 g1100 = vec4(gx00.w,gy00.w,gz00.w,gw00.w);
vec4 g0010 = vec4(gx10.x,gy10.x,gz10.x,gw10.x);
vec4 g1010 = vec4(gx10.y,gy10.y,gz10.y,gw10.y);
vec4 g0110 = vec4(gx10.z,gy10.z,gz10.z,gw10.z);
vec4 g1110 = vec4(gx10.w,gy10.w,gz10.w,gw10.w);
vec4 g0001 = vec4(gx01.x,gy01.x,gz01.x,gw01.x);
vec4 g1001 = vec4(gx01.y,gy01.y,gz01.y,gw01.y);
vec4 g0101 = vec4(gx01.z,gy01.z,gz01.z,gw01.z);
vec4 g1101 = vec4(gx01.w,gy01.w,gz01.w,gw01.w);
vec4 g0011 = vec4(gx11.x,gy11.x,gz11.x,gw11.x);
vec4 g1011 = vec4(gx11.y,gy11.y,gz11.y,gw11.y);
vec4 g0111 = vec4(gx11.z,gy11.z,gz11.z,gw11.z);
vec4 g1111 = vec4(gx11.w,gy11.w,gz11.w,gw11.w);

// Normalise gradients
vec4 norm00 = taylorInvSqrt(vec4(dot(g0000, g0000), dot(g0100, g0100),
     dot(g1000, g1000), dot(g1100, g1100)));
g0000 *= norm00.x;
g0100 *= norm00.y;
g1000 *= norm00.z;
```

```
  g1100 *= norm00.w;

  vec4 norm01 = taylorInvSqrt(vec4(dot(g0001, g0001), dot(g0101, g0101),
       dot(g1001, g1001), dot(g1101, g1101)));
  g0001 *= norm01.x;
  g0101 *= norm01.y;
  g1001 *= norm01.z;
  g1101 *= norm01.w;

  vec4 norm10 = taylorInvSqrt(vec4(dot(g0010, g0010), dot(g0110, g0110),
       dot(g1010, g1010), dot(g1110, g1110)));
  g0010 *= norm10.x;
  g0110 *= norm10.y;
  g1010 *= norm10.z;
  g1110 *= norm10.w;

  vec4 norm11 = taylorInvSqrt(vec4(dot(g0011, g0011), dot(g0111, g0111),
       dot(g1011, g1011), dot(g1111, g1111)));
  g0011 *= norm11.x;
  g0111 *= norm11.y;
  g1011 *= norm11.z;
  g1111 *= norm11.w;

  // Extrapolate gradient ramp from all 16 corners
  float n0000 = dot(g0000, Pf0);
  float n1000 = dot(g1000, vec4(Pf1.x, Pf0.yzw));
  float n0100 = dot(g0100, vec4(Pf0.x, Pf1.y, Pf0.zw));
  float n1100 = dot(g1100, vec4(Pf1.xy, Pf0.zw));
  float n0010 = dot(g0010, vec4(Pf0.xy, Pf1.z, Pf0.w));
  float n1010 = dot(g1010, vec4(Pf1.x, Pf0.y, Pf1.z, Pf0.w));
  float n0110 = dot(g0110, vec4(Pf0.x, Pf1.yz, Pf0.w));
  float n1110 = dot(g1110, vec4(Pf1.xyz, Pf0.w));
  float n0001 = dot(g0001, vec4(Pf0.xyz, Pf1.w));
  float n1001 = dot(g1001, vec4(Pf1.x, Pf0.yz, Pf1.w));
  float n0101 = dot(g0101, vec4(Pf0.x, Pf1.y, Pf0.z, Pf1.w));
  float n1101 = dot(g1101, vec4(Pf1.xy, Pf0.z, Pf1.w));
  float n0011 = dot(g0011, vec4(Pf0.xy, Pf1.zw));
  float n1011 = dot(g1011, vec4(Pf1.x, Pf0.y, Pf1.zw));
  float n0111 = dot(g0111, vec4(Pf0.x, Pf1.yzw));
  float n1111 = dot(g1111, Pf1);

  // Interpolate to compute final noise value
  vec4 fade_xyzw = fade(Pf0);
  vec4 n_0w = mix(vec4(n0000, n1000, n0100, n1100), vec4(n0001, n1001,
      n0101, n1101), fade_xyzw.w);
  vec4 n_1w = mix(vec4(n0010, n1010, n0110, n1110), vec4(n0011, n1011,
      n0111, n1111), fade_xyzw.w);
  vec4 n_zw = mix(n_0w, n_1w, fade_xyzw.z);
  vec2 n_yzw = mix(n_zw.xy, n_zw.zw, fade_xyzw.y);
  float n_xyzw = mix(n_yzw.x, n_yzw.y, fade_xyzw.x);
  return 2.2 * n_xyzw;
}

// Classic Perlin noise, periodic version
float cnoise(vec4 P, vec4 rep)
{
  vec4 Pi0 = mod(floor(P), rep); // Integer part modulo rep
  vec4 Pi1 = mod(Pi0 + 1.0, rep); // Integer part + 1 mod rep
  Pi0 = mod(Pi0, 289.0); // Avoid truncation effects
  Pi1 = mod(Pi1, 289.0);
  vec4 Pf0 = fract(P); // Fractional part for interpolation
  vec4 Pf1 = Pf0 - 1.0; // Fractional part - 1.0
  vec4 ix = vec4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
  vec4 iy = vec4(Pi0.yy, Pi1.yy);
  vec4 iz0 = vec4(Pi0.zzzz);
  vec4 iz1 = vec4(Pi1.zzzz);
  vec4 iw0 = vec4(Pi0.wwww);
  vec4 iw1 = vec4(Pi1.wwww);
```

```
vec4 ixy = permute(permute(ix) + iy);
vec4 ixy0 = permute(ixy + iz0);
vec4 ixy1 = permute(ixy + iz1);
vec4 ixy00 = permute(ixy0 + iw0);
vec4 ixy01 = permute(ixy0 + iw1);
vec4 ixy10 = permute(ixy1 + iw0);
vec4 ixy11 = permute(ixy1 + iw1);

vec4 gx00 = ixy00 * (1.0 / 7.0);
vec4 gy00 = floor(gx00) * (1.0 / 7.0);
vec4 gz00 = floor(gy00) * (1.0 / 6.0);
gx00 = fract(gx00) - 0.5;
gy00 = fract(gy00) - 0.5;
gz00 = fract(gz00) - 0.5;
vec4 gw00 = vec4(0.75) - abs(gx00) - abs(gy00) - abs(gz00);
vec4 sw00 = step(gw00, vec4(0.0));
gx00 -= sw00 * (step(0.0, gx00) - 0.5);
gy00 -= sw00 * (step(0.0, gy00) - 0.5);

vec4 gx01 = ixy01 * (1.0 / 7.0);
vec4 gy01 = floor(gx01) * (1.0 / 7.0);
vec4 gz01 = floor(gy01) * (1.0 / 6.0);
gx01 = fract(gx01) - 0.5;
gy01 = fract(gy01) - 0.5;
gz01 = fract(gz01) - 0.5;
vec4 gw01 = vec4(0.75) - abs(gx01) - abs(gy01) - abs(gz01);
vec4 sw01 = step(gw01, vec4(0.0));
gx01 -= sw01 * (step(0.0, gx01) - 0.5);
gy01 -= sw01 * (step(0.0, gy01) - 0.5);

vec4 gx10 = ixy10 * (1.0 / 7.0);
vec4 gy10 = floor(gx10) * (1.0 / 7.0);
vec4 gz10 = floor(gy10) * (1.0 / 6.0);
gx10 = fract(gx10) - 0.5;
gy10 = fract(gy10) - 0.5;
gz10 = fract(gz10) - 0.5;
vec4 gw10 = vec4(0.75) - abs(gx10) - abs(gy10) - abs(gz10);
vec4 sw10 = step(gw10, vec4(0.0));
gx10 -= sw10 * (step(0.0, gx10) - 0.5);
gy10 -= sw10 * (step(0.0, gy10) - 0.5);

vec4 gx11 = ixy11 * (1.0 / 7.0);
vec4 gy11 = floor(gx11) * (1.0 / 7.0);
vec4 gz11 = floor(gy11) * (1.0 / 6.0);
gx11 = fract(gx11) - 0.5;
gy11 = fract(gy11) - 0.5;
gz11 = fract(gz11) - 0.5;
vec4 gw11 = vec4(0.75) - abs(gx11) - abs(gy11) - abs(gz11);
vec4 sw11 = step(gw11, vec4(0.0));
gx11 -= sw11 * (step(0.0, gx11) - 0.5);
gy11 -= sw11 * (step(0.0, gy11) - 0.5);

vec4 g0000 = vec4(gx00.x,gy00.x,gz00.x,gw00.x);
vec4 g1000 = vec4(gx00.y,gy00.y,gz00.y,gw00.y);
vec4 g0100 = vec4(gx00.z,gy00.z,gz00.z,gw00.z);
vec4 g1100 = vec4(gx00.w,gy00.w,gz00.w,gw00.w);
vec4 g0010 = vec4(gx10.x,gy10.x,gz10.x,gw10.x);
vec4 g1010 = vec4(gx10.y,gy10.y,gz10.y,gw10.y);
vec4 g0110 = vec4(gx10.z,gy10.z,gz10.z,gw10.z);
vec4 g1110 = vec4(gx10.w,gy10.w,gz10.w,gw10.w);
vec4 g0001 = vec4(gx01.x,gy01.x,gz01.x,gw01.x);
vec4 g1001 = vec4(gx01.y,gy01.y,gz01.y,gw01.y);
vec4 g0101 = vec4(gx01.z,gy01.z,gz01.z,gw01.z);
vec4 g1101 = vec4(gx01.w,gy01.w,gz01.w,gw01.w);
vec4 g0011 = vec4(gx11.x,gy11.x,gz11.x,gw11.x);
vec4 g1011 = vec4(gx11.y,gy11.y,gz11.y,gw11.y);
vec4 g0111 = vec4(gx11.z,gy11.z,gz11.z,gw11.z);
```

```
  vec4 g1111 = vec4(gx11.w,gy11.w,gz11.w,gw11.w);

  vec4 norm00 = taylorInvSqrt(vec4(dot(g0000, g0000), dot(g0100, g0100),
       dot(g1000, g1000), dot(g1100, g1100)));
  g0000 *= norm00.x;
  g0100 *= norm00.y;
  g1000 *= norm00.z;
  g1100 *= norm00.w;

  vec4 norm01 = taylorInvSqrt(vec4(dot(g0001, g0001), dot(g0101, g0101),
       dot(g1001, g1001), dot(g1101, g1101)));
  g0001 *= norm01.x;
  g0101 *= norm01.y;
  g1001 *= norm01.z;
  g1101 *= norm01.w;

  vec4 norm10 = taylorInvSqrt(vec4(dot(g0010, g0010), dot(g0110, g0110),
       dot(g1010, g1010), dot(g1110, g1110)));
  g0010 *= norm10.x;
  g0110 *= norm10.y;
  g1010 *= norm10.z;
  g1110 *= norm10.w;

  vec4 norm11 = taylorInvSqrt(vec4(dot(g0011, g0011), dot(g0111, g0111),
       dot(g1011, g1011), dot(g1111, g1111)));
  g0011 *= norm11.x;
  g0111 *= norm11.y;
  g1011 *= norm11.z;
  g1111 *= norm11.w;

  float n0000 = dot(g0000, Pf0);
  float n1000 = dot(g1000, vec4(Pf1.x, Pf0.yzw));
  float n0100 = dot(g0100, vec4(Pf0.x, Pf1.y, Pf0.zw));
  float n1100 = dot(g1100, vec4(Pf1.xy, Pf0.zw));
  float n0010 = dot(g0010, vec4(Pf0.xy, Pf1.z, Pf0.w));
  float n1010 = dot(g1010, vec4(Pf1.x, Pf0.y, Pf1.z, Pf0.w));
  float n0110 = dot(g0110, vec4(Pf0.x, Pf1.yz, Pf0.w));
  float n1110 = dot(g1110, vec4(Pf1.xyz, Pf0.w));
  float n0001 = dot(g0001, vec4(Pf0.xyz, Pf1.w));
  float n1001 = dot(g1001, vec4(Pf1.x, Pf0.yz, Pf1.w));
  float n0101 = dot(g0101, vec4(Pf0.x, Pf1.y, Pf0.z, Pf1.w));
  float n1101 = dot(g1101, vec4(Pf1.xy, Pf0.z, Pf1.w));
  float n0011 = dot(g0011, vec4(Pf0.xy, Pf1.zw));
  float n1011 = dot(g1011, vec4(Pf1.x, Pf0.y, Pf1.zw));
  float n0111 = dot(g0111, vec4(Pf0.x, Pf1.yzw));
  float n1111 = dot(g1111, Pf1);

  vec4 fade_xyzw = fade(Pf0);
  vec4 n_0w = mix(vec4(n0000, n1000, n0100, n1100), vec4(n0001, n1001,
      n0101, n1101), fade_xyzw.w);
  vec4 n_1w = mix(vec4(n0010, n1010, n0110, n1110), vec4(n0011, n1011,
      n0111, n1111), fade_xyzw.w);
  vec4 n_zw = mix(n_0w, n_1w, fade_xyzw.z);
  vec2 n_yzw = mix(n_zw.xy, n_zw.zw, fade_xyzw.y);
  float n_xyzw = mix(n_yzw.x, n_yzw.y, fade_xyzw.x);
  return 2.2 * n_xyzw;
}
```

5.7 The MIT license