Simplex noise demystified

Stefan Gustavson, Linköping University, Sweden (stegu@itn.liu.se), 2005-03-22

In 2001, Ken Perlin presented "simplex noise", a replacement for his classic noise algorithm. Classic "Perlin noise" won him an academy award and has become an ubiquitous procedural primitive for computer graphics over the years, but in hindsight it has quite a few limitations. Ken Perlin himself designed simplex noise specifically to overcome those limitations, and he spent a lot of good thinking on it. Therefore, it is a better idea than his original algorithm. A few of the more prominent advantages are:

- Simplex noise has a lower computational complexity and requires fewer multiplications.
- Simplex noise scales to higher dimensions (4D, 5D and up) with *much* less computational cost, the complexity is $O(N)$ for N dimensions instead of the $O(2^N)$ of classic Noise. [Should be O(N^2)]
- Simplex noise has no noticeable directional artifacts.
- Simplex noise has a well-defined and continuous gradient everywhere that can be computed quite cheaply.
- Simplex noise is easy to implement in hardware.

Sadly, even now in early 2005 very few people seem to understand simplex noise, and almost nobody uses it, which is why I wrote this. I will try to explain the algorithm a little more thoroughly than Ken Perlin had time to do in his course notes from Siggraph 2001 and 2002, and hopefully make it clear that it is not as difficult to grasp as it first seems.

From what I've learned, what confuses people the most is the impenetrable nature of Ken Perlin's reference implementation in Java. He presents very compact and uncommented code to demonstrate the principle, but that code is clearly not meant to be read as a tutorial. After a few attempts I gave up on the code and read his paper instead, which was a lot more clear. Not crystal clear, though, as he presents the algorithm mostly in words and code snippets. I would have appreciated some graphs and figures and a few helpful equations, and that's what I try to provide here, to make it easier for others to understand the greatness and beauty of simplex noise. I will also explain things in one and two dimensions first to make things easier to explain with graphs and images, and then move on to three and four dimensions.

Classic noise

In order to explain simplex noise, it is helpful to have a good understanding of classic Perlin noise. I have seen quite a few bad and misinformed explanations in this area, so to make sure that you have the necessary groundwork done, I will present classic Perlin noise first.

Perlin noise is a so-called gradient noise, which means that you set a pseudo-random gradient at regularly spaced points in space, and interpolate a smooth function between those points. To generate Perlin noise in one dimension, you associate a pseudo-random gradient (or slope) for the noise function with each integer coordinate, and set the function value at each integer coordinate to zero.


For a given point x somewhere between two integer points, the value is interpolated between two values, namely the values that would have been the result if the closest linear slopes from the left and from the right had been extrapolated to the point in question. This interpolation is

not linear with distance, because that would not satisfy the constraint that the derivative of the noise function should be continuous also at the integer points. Instead, a blending function is used that has zero derivative at its endpoints. Originally, Ken Perlin used the Hermite blending function $f(t) = 3t^2 - 2t^3$, but because it is highly desirable to have a continuous second derivative for the noise function, he later changed that to suggest a fifth degree polynomial $f(t) = 6t^5 - 15t^4 + 10t^3$. These two functions are very similar, but the fifth degree curve also has a zero *second* derivative at its endpoints, which makes the noise function have a continuous second derivative everywhere, and that makes the noise function better suited for the common computer graphics tasks of surface displacement and bump mapping.
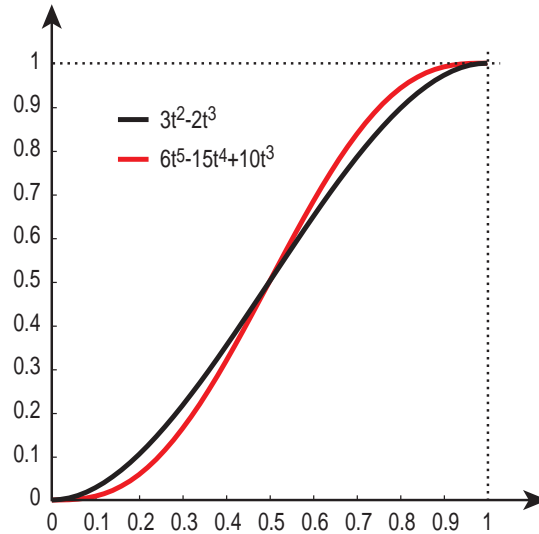
In two dimensions, the integer coordinate points form a regular square grid. At each grid point a pseudo-random 2D gradient is picked. For an arbitrary point P on the surface, the noise value is computed from the four closest grid points. As for the 1D case, the contribution from each of the four corners of the square grid cell is an extrapolation of a linear ramp with a constant gradient, with the value zero at its associated grid point.
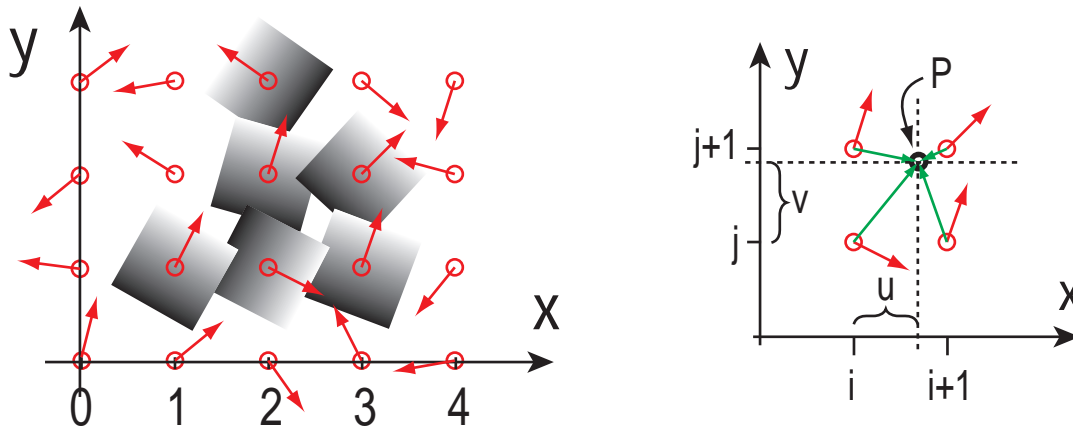
The value of each gradient ramp is computed by means of a scalar product (dot product) between the gradient vectors of each grid point and the vectors from the grid points $(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)$ to the point P being evaluated. In equations:

$P = (x, y), i = floor(x), j = floor(y)$

$g_{00} =$ gradient at (i, j), $g_{10} =$ gradient at $(i + 1, j)$

$g_{01} =$ gradient at $(i, j + 1)$, $g_{11} =$ gradient at $(i + 1, j + 1)$

$$u = x - i, \quad v = y - j$$

$$n_{00} = g_{00} \bullet \begin{bmatrix} u \\ v \end{bmatrix}, \quad n_{10} = g_{10} \bullet \begin{bmatrix} u-1 \\ v \end{bmatrix}, \quad n_{01} = g_{01} \bullet \begin{bmatrix} u \\ v-1 \end{bmatrix}, \quad n_{11} = g_{11} \bullet \begin{bmatrix} u-1 \\ v-1 \end{bmatrix}$$

The blending of the noise contribution from the four corners is performed in a manner similar to bilinear interpolation, using the fifth order blending curve to compute the interpolant:

$$f(t) = 6t^5 - 15t^4 + 10t^3$$
$$n_{x0} = n_{00}f(u) + n_{10}(1 - f(u)), \quad n_{x1} = n_{01}f(u) + n_{11}(1 - f(u))$$
$$n_{xy} = n_{x0}f(v) + n_{x1}(1 - f(v))$$

(1-f()) and f() should be swapped in these three equations. My bad.

The result n_{xy} is the final value of the noise function for the point (x, y). In 3D, the gradients are three-dimensional and the interpolation is performed along three axes, one at a time. I do not present details here, but you can find it in the code examples at the end of this document. Generalization can be made to an arbitrary number of dimensions.

Picking the gradients

For the noise function to be repeatable, i.e. always yield the same value for a given input point, gradients need to be pseudo-random, not truly random. They need to have enough variation to conceal the fact that the function is not truly random, but too much variation will cause unpredictable behaviour for the noise function. A good choice for 2D and higher is to pick gradients of unit length but different directions. For 2D, 8 or 16 gradients distributed around the unit circle is a good choice. For 3D, Ken Perlin's recommended set of gradients is the midpoints of each of the 12 edges of a cube centered on the origin.
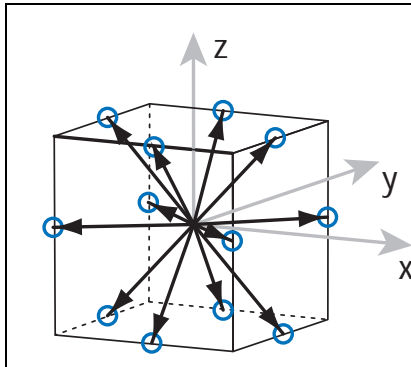
$$g_0 = (0, 1, 1), \quad g_1 = (0, 1, -1),$$
$$g_2 = (0, -1, 1), \quad g_3 = (0, -1, -1),$$
$$g_4 = (1, 0, 1), \quad g_5 = (1, 0, -1),$$
$$g_6 = (-1, 0, 1), \quad g_7 = (-1, 0, -1),$$
$$g_8 = (1, 1, 0), \quad g_9 = (1, -1, 0),$$
$$g_{10} = (-1, 1, 0), \quad g_{11} = (-1, -1, 0)$$

For 4D, a suitable set of gradients is the midpoints of each of the 32 edges of a 4D hypercube:

$$g_0 = (0, 1, 1, 1), \; g_1 = (0, 1, 1, -1), \; g_2 = (0, 1, -1, 1), \; g_3 = (0, 1, -1, -1),$$
$$g_4 = (0, -1, 1, 1), \; g_5 = (0, -1, 1, -1), \; g_6 = (0, -1, -1, 1), \; g_7 = (0, -1, -1, -1),$$
$$g_8 = (1, 0, 1, 1), \; g_9 = (1, 0, 1, -1), \; g_{10} = (1, 0, -1, 1), \; g_{11} = (1, 0, -1, -1),$$
$$g_{12} = (-1, 0, 1, 1), \; g_{13} = (-1, 0, 1, -1), \; g_{14} = (-1, 0, -1, 1), \; g_{15} = (-1, 0, -1, -1),$$
$$g_{16} = (1, 1, 0, 1), \; g_{17} = (1, 1, 0, -1), \; g_{18} = (1, -1, 0, 1), \; g_{19} = (1, -1, 0, -1),$$
$$g_{20} = (-1, 1, 0, 1), \; g_{21} = (-1, 1, 0, -1), \; g_{22} = (-1, -1, 0, 1), \; g_{23} = (-1, -1, 0, -1),$$
$$g_{24} = (1, 1, 1, 0), \; g_{25} = (1, 1, -1, 0), \; g_{26} = (1, -1, 1, 0), \; g_{27} = (1, -1, -1, 0),$$
$$g_{28} = (-1, 1, 1, 0), \; g_{29} = (-1, 1, -1, 0), \; g_{30} = (-1, -1, 1, 0), \; g_{31} = (-1, -1, -1, 0)$$

It really doesn't matter very much exactly which gradients are picked, as long as they are not too few and are reasonably evenly distributed over all directions. Gradients with values of only 0, 1 and -1 for each component are chosen because taking a dot product with such a vector does not require any multiplications, only additions and subtractions.

To associate each grid point with exactly one gradient, the integer coordinates of the point can be used to compute a hash value, which in turn can be used as the index into a look-up table of the gradients.

This concludes my presentation of classic Perlin noise.

Simplex grids

Simplex grids, or simplical tesselation of N-space, sounds fancy and is a bit hard to grasp, but what it boils down to is quite simple: for a space with N dimensions, pick the simplest and most compact shape that can be repeated to fill the entire space. For a one-dimensional space, the simplest (in fact, the only possible) space-filling shape is intervals of equal length placed one after another, head to tail. In two dimensions, the obvious choice for a space-filling shape is a square, but that shape has more corners than what is necessary. The simplest shape that tiles a 2D plane is a triangle, and the formal simplex shape in 2D is an equilateral triangle. Two of these make a rhombus, a shape that can be thought of as a square that has been squashed along its main diagonal.
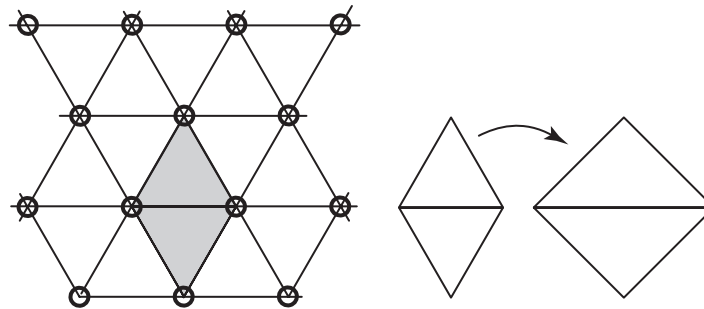
In three dimensions, the simplex shape is a slightly skewed tetrahedron, six of which make a cube that has been squashed along its main diagonal.
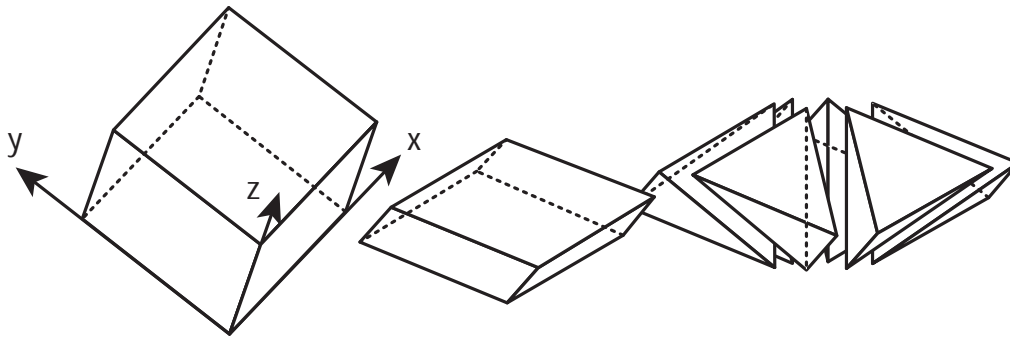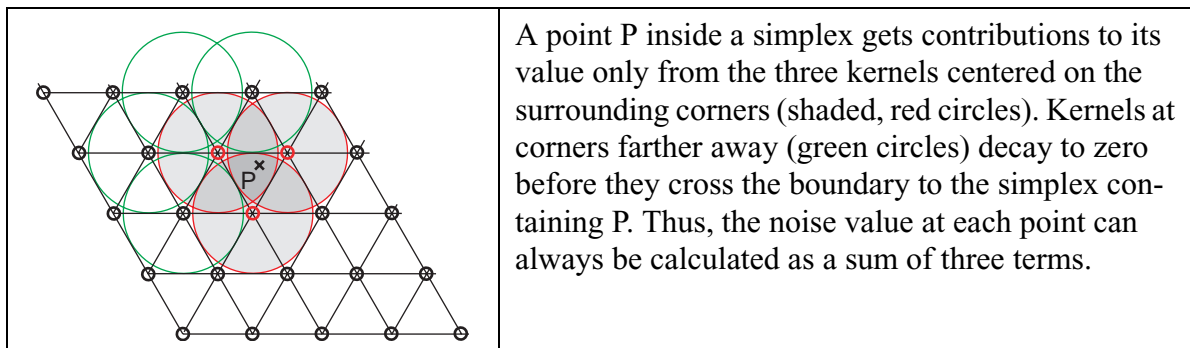

In four dimensions, the simplex shape is very hard to visualize, but it has five corners, and 24 of these shapes make a 4D hypercube that has been squashed along its main diagonal. Generally speaking, the simplex shape for N dimensions is a shape with $N+1$ corners, and $N!$ of these shapes can fill an N-dimensional hypercube that has been squashed along its main diagonal.

The definite advantage of a simplex shape is that it has as few corners as possible, much fewer corners than a hypercube. This makes it easier to interpolate values in the interior of the simplex based on the values at its corners. While a hypercube in N dimensions has 2^N corners, a sim-

plex in N dimensions has only $N+1$ corners. As we move to higher dimensions, the complexity of evaluating a function at each corner of a hypercube and interpolating along each principal axis is an $O(2^N)$ problem that quickly becomes intractable, while a similar evaluation for each corner of a simplex shape followed by interpolation presents a much less daunting computational task of complexity $O(N)$. ← N^2, actually.

Moving from interpolation to summation

A fundamental problem of classic noise is that it involves sequential interpolations along each dimension. Apart from the rapid increase in computational complexity as we move to higher dimensions, it becomes more and more of a problem to compute the analytic derivative of the interpolated function. Simplex noise instead uses a straight summation of contributions from each corner, where the contribution is a multiplication of the extrapolation of the gradient ramp and a radially symmetric attenuation function. In signal processing terms, this is a signal reconstruction kernel. The radial attenuation is carefully chosen so that the influence from each corner reaches zero before crossing the boundary to the next simplex. This means that points inside a simplex will only be influenced by the contributions from the corners of that particular simplex.


A point P inside a simplex gets contributions to its value only from the three kernels centered on the surrounding corners (shaded, red circles). Kernels at corners farther away (green circles) decay to zero before they cross the boundary to the simplex containing P. Thus, the noise value at each point can always be calculated as a sum of three terms.

For 2D, the influence from each corner can be visualized as a small wiggle function (Ken Perlin uses the term "surflet") around the corner point, with any point on the surface having at most three non-zero parts of a wiggle covering it.
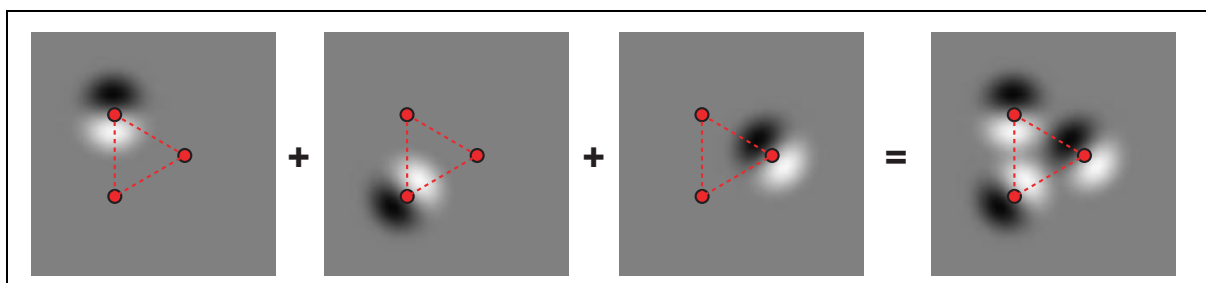

Selecting and traversing a simplex

The final trick to making a simplex noise algorithm work is to decide which simplex you are in for any point in space. This is most easily performed in two steps. First you skew the input coordinate space along the main diagonal so that each squashed hypercube of $N!$ simplices transforms to a regular, axis-aligned N-dimensional hypercube. Then you can easily decide which hypercube you are in by taking the integer part of the coordinates for each dimension, similarly to classic noise. Then, the further decision on which particular simplex the point is in can be made by comparing the magnitudes of the distances in each dimension from the hypercube origin to the point in question. Figures make this a lot more clear.
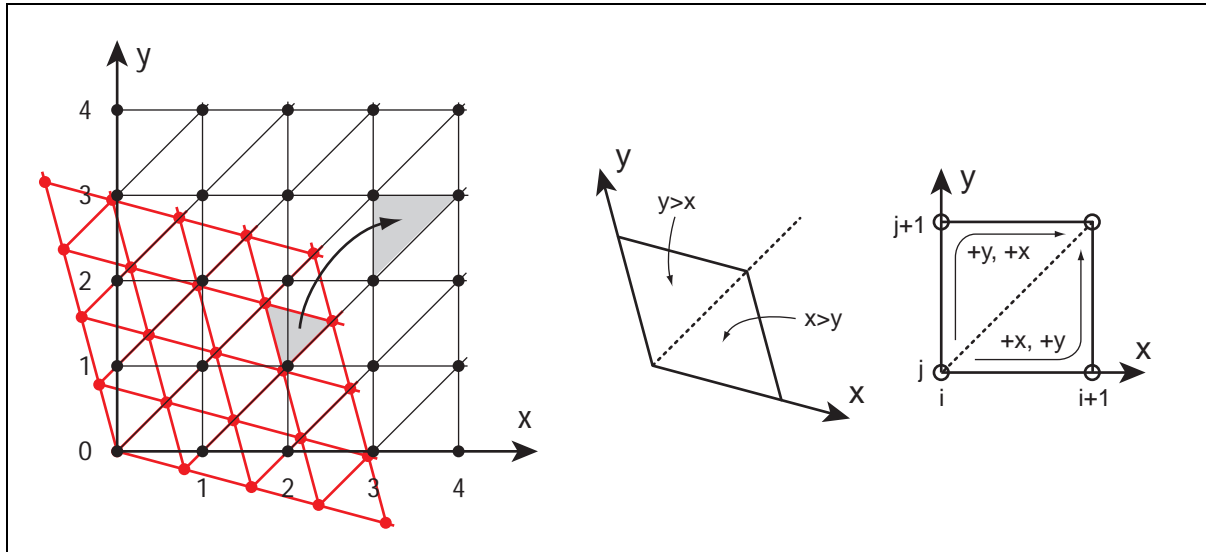
A 2D simplex grid of triangles can be skewed by a nonuniform scaling to a grid of right-angle isosceles triangles, two of which form a square with sides of length 1. By looking at the integer parts of the transformed coordinates (x, y) for the point we want to evaluate, we can quickly determine which cell of two simplices that contains the point. By also comparing the magnitudes of x and y, we can determine whether the point is in the upper or the lower simplex, and traverse the correct three corner points.

Just like a 2D simplex grid can be skewed to a regular square grid, a 3D simplex grid can be skewed to a regular cubical grid by a scaling along the main diagonal, and the integer parts of the coordinates for the transformed point can be used to determine which cell of 6 simplices the point is in. To further determine which of the 6 simplices we are in, we look at the magnitude of the coordinates relative to the cell origin. The figure below is a view of a cube cell along its main diagonal $x = y = z$. Points in the six simplices obey quite simple rules.
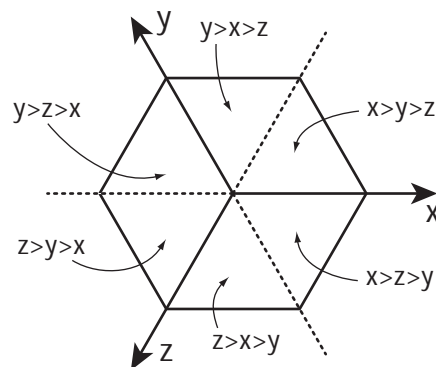
For the 4D case, things become incomprehensible visually, but the methods presented for 2D and 3D generalize nicely to higher dimensions. Sorting the (x,y,z,w) coordinates within the hypercube reveals $4! = 24$ possible outcomes for the ordering of the magnitude of the coordinates, and each particular ordering is unique to one of the 24 simplices.

In 2D, if $x > y$ the simplex corners are $(0, 0)$, $(1, 0)$ and $(1, 1)$, else the simplex corners are $(0, 0)$, $(0, 1)$ and $(1, 1)$. The simplex traversal always takes one unit step in x and one unit step in y, but in different order for each of the simplices.

The traversal scheme for 2D generalizes straight off to 3D and further to an arbitrary number of dimensions: to traverse each corner of a simplex in N dimensions, we should move from the origin of the hypercube $(0, 0, \ldots, 0)$ to the opposite corner $(1, 1, \ldots, 1)$, and move unit steps along each principal axis in turn, from the coordinate with the highest magnitude to the coordinate with the lowest magnitude.

The magnitude sorting of components to decide which order of unit steps along each dimension that traverses the simplex can be performed explicitly, by the usual pair-wise comparisons and swaps, but the results from the comparisons can also be used as an index into a lookup table. This is efficient and hardware friendly up to 5 dimensions, but becomes unwieldy for 6D and downright silly for higher dimensions. Explicit sorting will of course add to the complexity.

Example code

NOTE: Better public domain code from 2012 is now available at
http://www.itn.liu.se/~stegu/simplexnoise/SimplexNoise.java

All this can be put into code, and the code can be made a lot more clear than Ken Perlin's compact and uncommented Java reference implementation. I will use a hybrid approach for clarity, using the gradient hash method from classic noise but the simplex grid and straight summation of noise contributions of simplex noise. That is actually a faster method in software. My implementation gives the same visual appearance as Perlins simplex noise, but it does not give exactly the same pattern pixel for pixel, because the gradients are picked differently. If required, an exact match can be made by redesigning the gradient array and the permutation table.

To make the difference clear between classic noise and simplex noise, I first present code for classic Perlin noise in 3D, in an implementation and a coding style that is aimed at being similar to the simplex noise implementation to facilitate direct comparison. I then move on to 2D simplex noise, which Perlin does not present in his article, continue with my re-implementation of 3D simplex noise, and conclude with 4D simplex noise, which Perlin does not present either.

My code is unnecessarily long-winded, but readable. I tried to stay away from obfuscating shortcuts. I even avoided loops in order to show exactly what is being done, so there is quite a lot of stupid cut-and-paste and repetitive code that could be rewritten in a much more compact form by using short loops and a few small arrays instead of individual named variables.

Like Ken Perlin, I chose Java as the programming language for these examples. The Java code is very basic and should be easy to port to other languages even if you don't know Java. In fact, the code I present is a backport to Java from a GLSL project where I implemented simplex noise in a fragment shader. Some parts of the code are leftovers from the limitations of GLSL, and might not be particularly efficient for a pure software implementation. It is faster than Ken Perlins reference implementation, but his code is not particularly fast either, because it uses a long sequence of bit-wise operations on computing a hash value which I instead read directly from a small permutaton array. Perlin's code, although it is written in Java, is really aimed at a hardware implementation. So, if you are looking for a *fast* implementation of simplex noise, the code on the following pages might not be what you want. It is not dead slow, but it is first and foremost explanatory and meant to be read by humans, so it could be speeded up.

One disadvantage of my version is that it uses more memory than Perlin's simplex noise. I use a few look-up tables for some stuff that could be implemented in other ways if memory is a tight or nonexistent resource, e.g. if you are aiming at a slimmed bare-bones hardware implementation with logic gates and registers, or if you need to do this entirely in a small set of registers in a DSP architecture or something else with a high penalty for memory accesses. On modern programmable graphics hardware, there is an abundance of texture memory and texture access functions with good performance, so look-up tables of reasonable size can be implemented efficiently, and in software on a general CPU, a few hundred bytes of storage is not a problem.

```java
public class ClassicNoise { // Classic Perlin noise in 3D, for comparison

  private static int grad3[][] = {{1,1,0},{-1,1,0},{1,-1,0},{-1,-1,0},
                                  {1,0,1},{-1,0,1},{1,0,-1},{-1,0,-1},
                                  {0,1,1},{0,-1,1},{0,1,-1},{0,-1,-1}};

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

  // To remove the need for index wrapping, double the permutation table length
  private static int perm[] = new int[512];
  static { for(int i=0; i<512; i++) perm[i]=p[i & 255]; }

  // This method is a *lot* faster than using (int)Math.floor(x)
  private static int fastfloor(double x) {
    return x>0 ? (int)x : (int)x-1;
  }

  private static double dot(int g[], double x, double y, double z) {
    return g[0]*x + g[1]*y + g[2]*z;
  }

  private static double mix(double a, double b, double t) {
    return (1-t)*a + t*b;
  }

  private static double fade(double t) {
    return t*t*t*(t*(t*6-15)+10);
  }

  // Classic Perlin noise, 3D version
  public static double noise(double x, double y, double z) {

    // Find unit grid cell containing point
    int X = fastfloor(x);
    int Y = fastfloor(y);
    int Z = fastfloor(z);

    // Get relative xyz coordinates of point within that cell
    x = x - X;
    y = y - Y;
    z = z - Z;

    // Wrap the integer cells at 255 (smaller integer period can be introduced here)
    X = X & 255;
    Y = Y & 255;
    Z = Z & 255;
```

```
    // Calculate a set of eight hashed gradient indices
    int gi000 = perm[X+perm[Y+perm[Z]]] % 12;
    int gi001 = perm[X+perm[Y+perm[Z+1]]] % 12;
    int gi010 = perm[X+perm[Y+1+perm[Z]]] % 12;
    int gi011 = perm[X+perm[Y+1+perm[Z+1]]] % 12;
    int gi100 = perm[X+1+perm[Y+perm[Z]]] % 12;
    int gi101 = perm[X+1+perm[Y+perm[Z+1]]] % 12;
    int gi110 = perm[X+1+perm[Y+1+perm[Z]]] % 12;
    int gi111 = perm[X+1+perm[Y+1+perm[Z+1]]] % 12;

    // The gradients of each corner are now:
    // g000 = grad3[gi000];
    // g001 = grad3[gi001];
    // g010 = grad3[gi010];
    // g011 = grad3[gi011];
    // g100 = grad3[gi100];
    // g101 = grad3[gi101];
    // g110 = grad3[gi110];
    // g111 = grad3[gi111];

   // Calculate noise contributions from each of the eight corners
    double n000= dot(grad3[gi000], x, y, z);
    double n100= dot(grad3[gi100], x-1, y, z);
    double n010= dot(grad3[gi010], x, y-1, z);
    double n110= dot(grad3[gi110], x-1, y-1, z);
    double n001= dot(grad3[gi001], x, y, z-1);
    double n101= dot(grad3[gi101], x-1, y, z-1);
    double n011= dot(grad3[gi011], x, y-1, z-1);
    double n111= dot(grad3[gi111], x-1, y-1, z-1);

    // Compute the fade curve value for each of x, y, z
    double u = fade(x);
    double v = fade(y);
    double w = fade(z);

     // Interpolate along x the contributions from each of the corners
    double nx00 = mix(n000, n100, u);
    double nx01 = mix(n001, n101, u);
    double nx10 = mix(n010, n110, u);
    double nx11 = mix(n011, n111, u);

    // Interpolate the four results along y
    double nxy0 = mix(nx00, nx10, v);
    double nxy1 = mix(nx01, nx11, v);

    // Interpolate the two last results along z
    double nxyz = mix(nxy0, nxy1, w);

   return nxyz;
  }
}
```

```java
public class SimplexNoise {  // Simplex noise in 2D, 3D and 4D

  private static int grad3[][] = {{1,1,0},{-1,1,0},{1,-1,0},{-1,-1,0},
                     {1,0,1},{-1,0,1},{1,0,-1},{-1,0,-1},
                     {0,1,1},{0,-1,1},{0,1,-1},{0,-1,-1}};

  private static int grad4[][]= {{0,1,1,1}, {0,1,1,-1}, {0,1,-1,1}, {0,1,-1,-1},
            {0,-1,1,1}, {0,-1,1,-1}, {0,-1,-1,1}, {0,-1,-1,-1},
            {1,0,1,1}, {1,0,1,-1}, {1,0,-1,1}, {1,0,-1,-1},
            {-1,0,1,1}, {-1,0,1,-1}, {-1,0,-1,1}, {-1,0,-1,-1},
            {1,1,0,1}, {1,1,0,-1}, {1,-1,0,1}, {1,-1,0,-1},
            {-1,1,0,1}, {-1,1,0,-1}, {-1,-1,0,1}, {-1,-1,0,-1},
            {1,1,1,0}, {1,1,-1,0}, {1,-1,1,0}, {1,-1,-1,0},
            {-1,1,1,0}, {-1,1,-1,0}, {-1,-1,1,0}, {-1,-1,-1,0}};

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

  // To remove the need for index wrapping, double the permutation table length
  private static int perm[] = new int[512];
  static { for(int i=0; i<512; i++) perm[i]=p[i & 255]; }

  // A lookup table to traverse the simplex around a given point in 4D.
  // Details can be found where this table is used, in the 4D noise method.
  private static int simplex[][] = {
   {0,1,2,3},{0,1,3,2},{0,0,0,0},{0,2,3,1},{0,0,0,0},{0,0,0,0},{0,0,0,0},{1,2,3,0},
   {0,2,1,3},{0,0,0,0},{0,3,1,2},{0,3,2,1},{0,0,0,0},{0,0,0,0},{0,0,0,0},{1,3,2,0},
   {0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},
   {1,2,0,3},{0,0,0,0},{1,3,0,2},{0,0,0,0},{0,0,0,0},{0,0,0,0},{2,3,0,1},{2,3,1,0},
   {1,0,2,3},{1,0,3,2},{0,0,0,0},{0,0,0,0},{0,0,0,0},{2,0,3,1},{0,0,0,0},{2,1,3,0},
   {0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},
   {2,0,1,3},{0,0,0,0},{0,0,0,0},{0,0,0,0},{3,0,1,2},{3,0,2,1},{0,0,0,0},{3,1,2,0},
   {2,1,0,3},{0,0,0,0},{0,0,0,0},{0,0,0,0},{3,1,0,2},{0,0,0,0},{3,2,0,1},{3,2,1,0}};

  // This method is a *lot* faster than using (int)Math.floor(x)
  private static int fastfloor(double x) {
    return x>0 ? (int)x : (int)x-1;
  }

  private static double dot(int g[], double x, double y) {
    return g[0]*x + g[1]*y; }

  private static double dot(int g[], double x, double y, double z) {
    return g[0]*x + g[1]*y + g[2]*z; }

  private static double dot(int g[], double x, double y, double z, double w) {
    return g[0]*x + g[1]*y + g[2]*z + g[3]*w; }
```

```java
// 2D simplex noise
public static double noise(double xin, double yin) {

  double n0, n1, n2; // Noise contributions from the three corners

  // Skew the input space to determine which simplex cell we're in
  final double F2 = 0.5*(Math.sqrt(3.0)-1.0);
  double s = (xin+yin)*F2; // Hairy factor for 2D
  int i = fastfloor(xin+s);
  int j = fastfloor(yin+s);

  final double G2 = (3.0-Math.sqrt(3.0))/6.0;
  double t = (i+j)*G2;
  double X0 = i-t; // Unskew the cell origin back to (x,y) space
  double Y0 = j-t;
  double x0 = xin-X0; // The x,y distances from the cell origin
  double y0 = yin-Y0;

  // For the 2D case, the simplex shape is an equilateral triangle.
  // Determine which simplex we are in.
  int i1, j1; // Offsets for second (middle) corner of simplex in (i,j) coords
  if(x0>y0) {i1=1; j1=0;} // lower triangle, XY order: (0,0)->(1,0)->(1,1)
  else {i1=0; j1=1;}      // upper triangle, YX order: (0,0)->(0,1)->(1,1)

  // A step of (1,0) in (i,j) means a step of (1-c,-c) in (x,y), and
  // a step of (0,1) in (i,j) means a step of (-c,1-c) in (x,y), where
  // c = (3-sqrt(3))/6

  double x1 = x0 - i1 + G2; // Offsets for middle corner in (x,y) unskewed coords
  double y1 = y0 - j1 + G2;
  double x2 = x0 - 1.0 + 2.0 * G2; // Offsets for last corner in (x,y) unskewed coords
  double y2 = y0 - 1.0 + 2.0 * G2;

  // Work out the hashed gradient indices of the three simplex corners
  int ii = i & 255;
  int jj = j & 255;
  int gi0 = perm[ii+perm[jj]] % 12;
  int gi1 = perm[ii+i1+perm[jj+j1]] % 12;
  int gi2 = perm[ii+1+perm[jj+1]] % 12;

  // Calculate the contribution from the three corners
  double t0 = 0.5 - x0*x0-y0*y0;
  if(t0<0) n0 = 0.0;
  else {
    t0 *= t0;
    n0 = t0 * t0 * dot(grad3[gi0], x0, y0);  // (x,y) of grad3 used for 2D gradient
  }

  double t1 = 0.5 - x1*x1-y1*y1;
  if(t1<0) n1 = 0.0;
  else {
    t1 *= t1;
    n1 = t1 * t1 * dot(grad3[gi1], x1, y1);
  }
```

```java
  double t2 = 0.5 - x2*x2-y2*y2;
  if(t2<0) n2 = 0.0;
  else {
    t2 *= t2;
    n2 = t2 * t2 * dot(grad3[gi2], x2, y2);
  }

  // Add contributions from each corner to get the final noise value.
  // The result is scaled to return values in the interval [-1,1].
  return 70.0 * (n0 + n1 + n2);
}


// 3D simplex noise
public static double noise(double xin, double yin, double zin) {

  double n0, n1, n2, n3; // Noise contributions from the four corners

  // Skew the input space to determine which simplex cell we're in
  final double F3 = 1.0/3.0;
  double s = (xin+yin+zin)*F3; // Very nice and simple skew factor for 3D
  int i = fastfloor(xin+s);
  int j = fastfloor(yin+s);
  int k = fastfloor(zin+s);

  final double G3 = 1.0/6.0; // Very nice and simple unskew factor, too
  double t = (i+j+k)*G3;
  double X0 = i-t; // Unskew the cell origin back to (x,y,z) space
  double Y0 = j-t;
  double Z0 = k-t;
  double x0 = xin-X0; // The x,y,z distances from the cell origin
  double y0 = yin-Y0;
  double z0 = zin-Z0;

  // For the 3D case, the simplex shape is a slightly irregular tetrahedron.
  // Determine which simplex we are in.
  int i1, j1, k1; // Offsets for second corner of simplex in (i,j,k) coords
  int i2, j2, k2; // Offsets for third corner of simplex in (i,j,k) coords

  if(x0>=y0) {
    if(y0>=z0)
      { i1=1; j1=0; k1=0; i2=1; j2=1; k2=0; } // X Y Z order
      else if(x0>=z0) { i1=1; j1=0; k1=0; i2=1; j2=0; k2=1; } // X Z Y order
      else { i1=0; j1=0; k1=1; i2=1; j2=0; k2=1; } // Z X Y order
    }
  else { // x0<y0
    if(y0<z0) { i1=0; j1=0; k1=1; i2=0; j2=1; k2=1; } // Z Y X order
    else if(x0<z0) { i1=0; j1=1; k1=0; i2=0; j2=1; k2=1; } // Y Z X order
    else { i1=0; j1=1; k1=0; i2=1; j2=1; k2=0; } // Y X Z order
  }

  // A step of (1,0,0) in (i,j,k) means a step of (1-c,-c,-c) in (x,y,z),
  // a step of (0,1,0) in (i,j,k) means a step of (-c,1-c,-c) in (x,y,z), and
  // a step of (0,0,1) in (i,j,k) means a step of (-c,-c,1-c) in (x,y,z), where
  // c = 1/6.
```

```
    double x1 = x0 - i1 + G3; // Offsets for second corner in (x,y,z) coords
    double y1 = y0 - j1 + G3;
    double z1 = z0 - k1 + G3;
    double x2 = x0 - i2 + 2.0*G3; // Offsets for third corner in (x,y,z) coords
    double y2 = y0 - j2 + 2.0*G3;
    double z2 = z0 - k2 + 2.0*G3;
    double x3 = x0 - 1.0 + 3.0*G3; // Offsets for last corner in (x,y,z) coords
    double y3 = y0 - 1.0 + 3.0*G3;
    double z3 = z0 - 1.0 + 3.0*G3;

    // Work out the hashed gradient indices of the four simplex corners
    int ii = i & 255;
    int jj = j & 255;
    int kk = k & 255;
    int gi0 = perm[ii+perm[jj+perm[kk]]] % 12;
    int gi1 = perm[ii+i1+perm[jj+j1+perm[kk+k1]]] % 12;
    int gi2 = perm[ii+i2+perm[jj+j2+perm[kk+k2]]] % 12;
    int gi3 = perm[ii+1+perm[jj+1+perm[kk+1]]] % 12;

    // Calculate the contribution from the four corners
    double t0 = 0.6 - x0*x0 - y0*y0 - z0*z0;
    if(t0<0) n0 = 0.0;
    else {
      t0 *= t0;
      n0 = t0 * t0 * dot(grad3[gi0], x0, y0, z0);
    }

    double t1 = 0.6 - x1*x1 - y1*y1 - z1*z1;
    if(t1<0) n1 = 0.0;
    else {
      t1 *= t1;
      n1 = t1 * t1 * dot(grad3[gi1], x1, y1, z1);
    }

    double t2 = 0.6 - x2*x2 - y2*y2 - z2*z2;
    if(t2<0) n2 = 0.0;
    else {
      t2 *= t2;
      n2 = t2 * t2 * dot(grad3[gi2], x2, y2, z2);
    }

    double t3 = 0.6 - x3*x3 - y3*y3 - z3*z3;
    if(t3<0) n3 = 0.0;
    else {
      t3 *= t3;
      n3 = t3 * t3 * dot(grad3[gi3], x3, y3, z3);
    }

    // Add contributions from each corner to get the final noise value.
    // The result is scaled to stay just inside [-1,1]
    return 32.0*(n0 + n1 + n2 + n3);
}
```

Should be 0.5, not 0.6, else the noise is not continuous at simplex boundaries. Same for 4D case.

```
// 4D simplex noise
double noise(double x, double y, double z, double w) {

  // The skewing and unskewing factors are hairy again for the 4D case
  final double F4 = (Math.sqrt(5.0)-1.0)/4.0;
  final double G4 = (5.0-Math.sqrt(5.0))/20.0;
  double n0, n1, n2, n3, n4; // Noise contributions from the five corners

  // Skew the (x,y,z,w) space to determine which cell of 24 simplices we're in
  double s = (x + y + z + w) * F4; // Factor for 4D skewing
  int i = fastfloor(x + s);
  int j = fastfloor(y + s);
  int k = fastfloor(z + s);
  int l = fastfloor(w + s);

  double t = (i + j + k + l) * G4; // Factor for 4D unskewing
  double X0 = i - t; // Unskew the cell origin back to (x,y,z,w) space
  double Y0 = j - t;
  double Z0 = k - t;
  double W0 = l - t;

  double x0 = x - X0;  // The x,y,z,w distances from the cell origin
  double y0 = y - Y0;
  double z0 = z - Z0;
  double w0 = w - W0;

  // For the 4D case, the simplex is a 4D shape I won't even try to describe.
  // To find out which of the 24 possible simplices we're in, we need to
  // determine the magnitude ordering of x0, y0, z0 and w0.
  // The method below is a good way of finding the ordering of x,y,z,w and
  // then find the correct traversal order for the simplex we're in.
  // First, six pair-wise comparisons are performed between each possible pair
  // of the four coordinates, and the results are used to add up binary bits
  // for an integer index.
  int c1 = (x0 > y0) ? 32 : 0;
  int c2 = (x0 > z0) ? 16 : 0;
  int c3 = (y0 > z0) ? 8 : 0;
  int c4 = (x0 > w0) ? 4 : 0;
  int c5 = (y0 > w0) ? 2 : 0;
  int c6 = (z0 > w0) ? 1 : 0;
  int c = c1 + c2 + c3 + c4 + c5 + c6;

  int i1, j1, k1, l1; // The integer offsets for the second simplex corner
  int i2, j2, k2, l2; // The integer offsets for the third simplex corner
  int i3, j3, k3, l3; // The integer offsets for the fourth simplex corner

  // simplex[c] is a 4-vector with the numbers 0, 1, 2 and 3 in some order.
  // Many values of c will never occur, since e.g. x>y>z>w makes x<z, y<w and x<w
  // impossible. Only the 24 indices which have non-zero entries make any sense.
  // We use a thresholding to set the coordinates in turn from the largest magnitude.
  // The number 3 in the "simplex" array is at the position of the largest coordinate.
  i1 = simplex[c][0]>=3 ? 1 : 0;
  j1 = simplex[c][1]>=3 ? 1 : 0;
  k1 = simplex[c][2]>=3 ? 1 : 0;
  l1 = simplex[c][3]>=3 ? 1 : 0;
  // The number 2 in the "simplex" array is at the second largest coordinate.
  i2 = simplex[c][0]>=2 ? 1 : 0;
  j2 = simplex[c][1]>=2 ? 1 : 0;
```

```
k2 = simplex[c][2]>=2 ? 1 : 0;
l2 = simplex[c][3]>=2 ? 1 : 0;
// The number 1 in the "simplex" array is at the second smallest coordinate.
i3 = simplex[c][0]>=1 ? 1 : 0;
j3 = simplex[c][1]>=1 ? 1 : 0;
k3 = simplex[c][2]>=1 ? 1 : 0;
l3 = simplex[c][3]>=1 ? 1 : 0;
// The fifth corner has all coordinate offsets = 1, so no need to look that up.

double x1 = x0 - i1 + G4; // Offsets for second corner in (x,y,z,w) coords
double y1 = y0 - j1 + G4;
double z1 = z0 - k1 + G4;
double w1 = w0 - l1 + G4;
double x2 = x0 - i2 + 2.0*G4; // Offsets for third corner in (x,y,z,w) coords
double y2 = y0 - j2 + 2.0*G4;
double z2 = z0 - k2 + 2.0*G4;
double w2 = w0 - l2 + 2.0*G4;
double x3 = x0 - i3 + 3.0*G4; // Offsets for fourth corner in (x,y,z,w) coords
double y3 = y0 - j3 + 3.0*G4;
double z3 = z0 - k3 + 3.0*G4;
double w3 = w0 - l3 + 3.0*G4;
double x4 = x0 - 1.0 + 4.0*G4; // Offsets for last corner in (x,y,z,w) coords
double y4 = y0 - 1.0 + 4.0*G4;
double z4 = z0 - 1.0 + 4.0*G4;
double w4 = w0 - 1.0 + 4.0*G4;

// Work out the hashed gradient indices of the five simplex corners
int ii = i & 255;
int jj = j & 255;
int kk = k & 255;
int ll = l & 255;
int gi0 = perm[ii+perm[jj+perm[kk+perm[ll]]]] % 32;
int gi1 = perm[ii+i1+perm[jj+j1+perm[kk+k1+perm[ll+l1]]]] % 32;
int gi2 = perm[ii+i2+perm[jj+j2+perm[kk+k2+perm[ll+l2]]]] % 32;
int gi3 = perm[ii+i3+perm[jj+j3+perm[kk+k3+perm[ll+l3]]]] % 32;
int gi4 = perm[ii+1+perm[jj+1+perm[kk+1+perm[ll+1]]]] % 32;

// Calculate the contribution from the five corners
double t0 = 0.6 - x0*x0 - y0*y0 - z0*z0 - w0*w0;
if(t0<0) n0 = 0.0;
else {
  t0 *= t0;
  n0 = t0 * t0 * dot(grad4[gi0], x0, y0, z0, w0);
}

double t1 = 0.6 - x1*x1 - y1*y1 - z1*z1 - w1*w1;
if(t1<0) n1 = 0.0;
else {
  t1 *= t1;
  n1 = t1 * t1 * dot(grad4[gi1], x1, y1, z1, w1);
}

double t2 = 0.6 - x2*x2 - y2*y2 - z2*z2 - w2*w2;
if(t2<0) n2 = 0.0;
else {
  t2 *= t2;
  n2 = t2 * t2 * dot(grad4[gi2], x2, y2, z2, w2);
}
```

```
  double t3 = 0.6 - x3*x3 - y3*y3 - z3*z3 - w3*w3;
   if(t3<0) n3 = 0.0;
   else {
     t3 *= t3;
     n3 = t3 * t3 * dot(grad4[gi3], x3, y3, z3, w3);
   }

  double t4 = 0.6 - x4*x4 - y4*y4 - z4*z4 - w4*w4;
   if(t4<0) n4 = 0.0;
   else {
     t4 *= t4;
     n4 = t4 * t4 * dot(grad4[gi4], x4, y4, z4, w4);
   }

   // Sum up and scale the result to cover the range [-1,1]
   return 27.0 * (n0 + n1 + n2 + n3 + n4);
  }
}
```
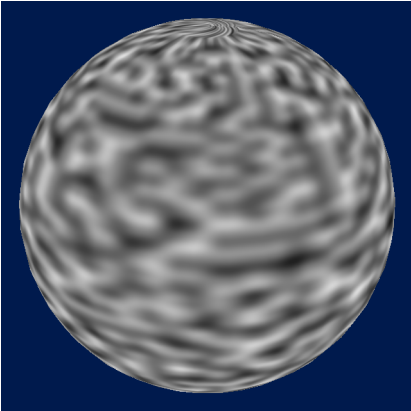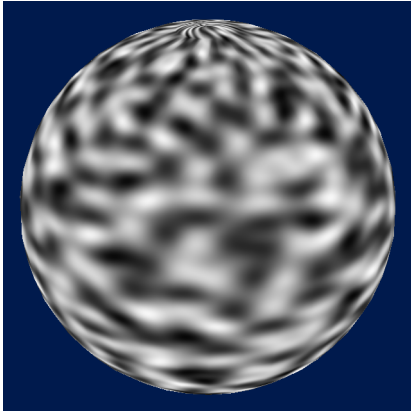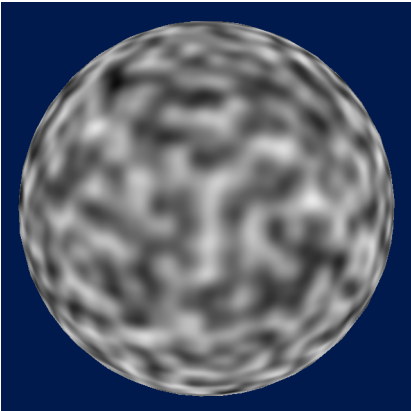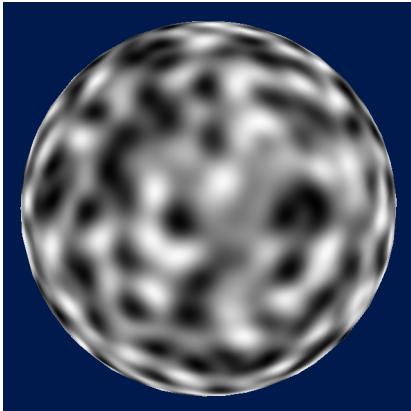
Visual comparison between classic and simplex noise

It is evident that simplex noise in most respects is a better idea than classic noise, and that it meets the criteria Ken Perlin set up to characterise a "good" Noise function. However, it has a slightly different visual character to it, so it's not always a direct plug-in replacement for classic noise. Applications that depend on the detailed characteristics of classic noise, like the precise feature size, the exact range of values or higher order statistics, might need some modification to look good when using simplex noise instead. Note in particular that a 3D section of 4D simplex noise is different from 3D simplex noise. This is the visual result of moving from the time-consuming interpolation one dimension at a time to a fast, direct summation.

	Classic Perlin noise	Simplex noise
2D		
3D		
4D		