It's a Noisy World Out There

Wyatt Sanders
UC Santa Cruz
Santa Cruz, CA 95060

ABSTRACT
In this paper, we explore the application of simplex noise to generate scalable and infinite universes, each with their own procedurally generated and deterministic galaxies, solar systems, and worlds.

Categories and Subject Descriptors
Algorithms, Design, Experimentation

Keywords
Procedural, PCG, noise, Simplex, universe, world, Perlin, density, voxel, tile, chunk, deltas, biomes

1. INTRODUCTION
Infinite worlds are a hot topic in today's gaming world. A lot of effort has been put into the creation of "limitless" amounts of procedurally generated worlds. Examples of these strides can be seen in games like Minecraft, Terraria, and Starbound.

In early versions of Minecraft, several 2D Perlin noise height maps were used to set the shape of the world. One was used for the maps overall elevation, one used for terrain roughness, and another used for local detail. The height of any given column of blocks was (elevation + (roughness * detail)) * 64 + 64. This method did not generate any overhangs so the method was switched to one that used 3D Perlin noise with the noise value treated as a density value rather than a height value [1].

Starbound uses deterministic terrain and planet generation systems. This means that if a world was generated at a given coordinate in space and it was generated at that same location every time, the final pass of that generation would look the same every time it is generated. One of the benefits for this approach is a smaller save file. All you would need to record are the coordinates, the seed, and the deltas created by the player(s) [2]. Everything else would be deterministic: the loot, the foliage, the biomes, the enemies, etc.

Not so much was found regarding how Terraria's terrain is generated but it looks like it uses Perlin noise to generate height maps for its worlds. It seems to be more like Minecraft than Starbound, in that regard.

For this program, an approach similar to Starbound's terrain generation was used.

2. NOISE
2.1 Types of Noise
There are several types of noise all with their own applications. Games like Minecraft, Terraria, and Starbound use variations of Perlin noise.

2.1.1 Perlin
Perlin noise is a type of gradient noise developed by Ken Perlin in 1983. Perlin noise is most commonly implemented as a multi-dimensional function and involves a definition of the grid space of values, the computation of the dot product between the distance-gradient vectors and the interpolation between these values [3].

For the n-dimensional grid, each coordinate is assigned a gradient vector in *n* dimensions. After the grid is setup, a distance vector between is computed for each gird coordinate. The dot product between the gradient vector and the distance vector is then calculated. This is done in order to determine in which grid cell a point lies. These dot products are then interpolated between each node using a linear function [3].

2.1.2 Simplex
.Simplex noise is a modification of Perlin noise and is also designed by Ken Perlin. It is characterized with a lower computational complexity, specifically in larger *n* dimensions [4]. This makes Simplex noise an ideal choice for procedural generation in video games and is the type of noise used for It's A *Noisy World Out There*.

2.2 Why Simplex Noise?
Simplex noise has a few advantages over Perlin noise. These include [4]:

A lower computational complexity

Scales to higher dimensions. Order is $O(n^2)$ instead of the $O(2^n)$ complexity that you get with classic Perlin noise.

No noticeable directional artifacts.

Continuous gradient.

Easy implementation.

[4]

Simplex noise creates a grid that, for an *n*-dimensional space, uses the simplest "space-filling" shapes that can be repeated to fill that entire space [5]. For 2-dimensional space, this is an equilateral triangle. For 3-dimensinoal space, this is a tetrahedron. When thinking of a grid for a 2-dimensional space, the obvious shape to populate the space would be a square [5]. This does not fit the simplex model, however because equilateral triangles do not fit

into squares. So, something must be done to the shape of the equilateral triangles to fit the shapes into the square shaped grid. Thus, the equilateral triangles are skewed to nicely fit the 2-dimensional space [4]. This also reduces the complexities of calculating which grid space a specific point lays which leads to easier computation of interpolated and weighted grid space values. The most complicated calculation to perform at this point is a line check from grid point A to grid point B. If a point lies above this line, the point belongs to the grid space of the line. If a point lies below this line, it belongs to the grid space below the line.

2.3 Simplex Noise Code

2.3.1 Raw 2D Simplex Noise [5]

```
double SimplexNoise::rawNoise(double x, double y) const
{
    // Noise contributions from the three corners
    double n0, n1, n2;

    // Skew the input space to determine which simplex cell we're in
    double F2 = 0.5 * (sqrtf(3.0) - 1.0);
    // Hairy factor for 2D
    double s = (x + y) * F2;
    int i = MathUtils::fastFloor(x + s);
    int j = MathUtils::fastFloor(y + s);

    double G2 = (3.0 - sqrtf(3.0)) / 6.0;
    double t = (i + j) * G2;
    // Unskew the cell origin back to (x,y) space
    double X0 = i - t;
    double Y0 = j - t;
    // The x,y distances from the cell origin
    double x0 = x - X0;
    double y0 = y - 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 * MathUtils::dot(gradient[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 * MathUtils::dot(gradient[gi1], x1, y1);
    }

    double t2 = 0.5 - x2*x2 - y2*y2;
    if (t2<0) n2 = 0.0;
    else {
        t2 *= t2;
        n2 = t2 * t2 * MathUtils::dot(gradient[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);
}
```

2.3.2 Octave Noise [5]

```
double SimplexNoise::octaveNoise(const float octaves, const float persistence, const float scale, const float x, const float y) {
    double total = 0;
    double frequency = scale;
    double amplitude = 1;

    // We have to keep track of the largest possible amplitude,
    // because each octave adds more, and we need a value in [-1, 1].
    double maxAmplitude = 0;

    for (int i = 0; i < octaves; i++) {
        total += rawNoise(x * frequency, y * frequency) * amplitude;

        frequency *= 2;
        maxAmplitude += amplitude;
        amplitude *= persistence;
    }

    return total / maxAmplitude;
}
```

2.3.3 Scaled Noise [5]

```
double SimplexNoise::scaledRawNoise(const float loBound, const float hiBound, const float x, const float y) {
    return rawNoise(x, y) * (hiBound - loBound) / 2 + (hiBound + loBound) / 2;
}

double SimplexNoise::scaledOctaveNoise(const float octaves, const float persistence, const float scale, const float loBound, const float hiBound, const float x, const float y) {
    return octaveNoise(octaves, persistence, scale, x, y) * (hiBound - loBound) + (hiBound + loBound) / 2;
}
```

3. THE GENERATION PROCESS

The generation process begins with the creation of the universe, then the galaxies, then the planetary systems, then the worlds.

3.1 The Big Bang

The first step is the generation of the seed value for the universe. This value is used to determine the number of galaxies that exist in the universe, the size of these galaxies, along with their locations within the generated universe.

3.2 The Universe

Given the seed value for the universe, the number of galaxies is determined pseudo-randomly. For each of these locations, a noise value used as a density value to determine the galaxies total "mass." This mass determines the size of the galaxy.

3.3 The Galaxy

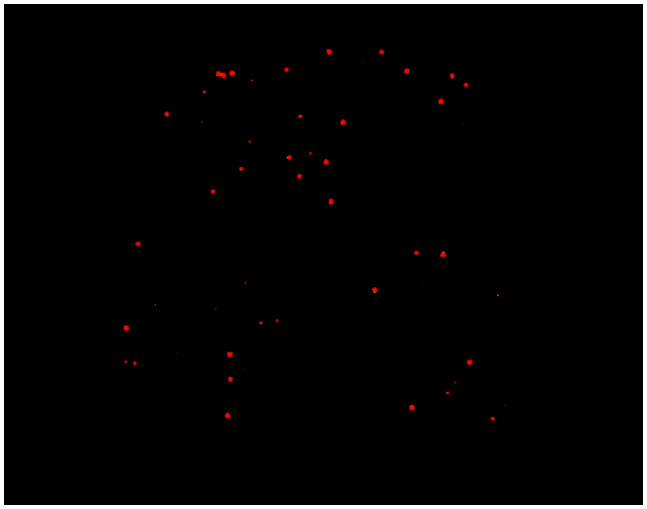
Once a galaxy is created, its mass value is used to determine its size. Based on this size and location in the universe as a seeding value, a number is generated to determine the number of planetary systems the galaxy contains. Then the galaxy undergoes the same process as the universe in order to determine the size of each of the planetary systems. Then each planetary system is generated.

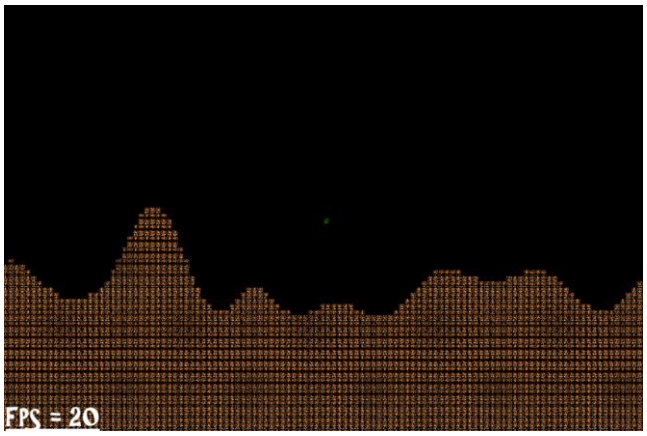
3.4 The Planetary Systems

Given the size of the planetary system, the number of worlds in the system is determined. This is done the same way the universe and galaxies were created except that the density of each world determines its chances for having satellite worlds.

3.5 The Worlds/Satellites

Once a world is generated, its size and location are used as a seed to generate the terrain. This seed value determines the persistence, frequency, and octave value for the noise field. First, tile chunks are generated based on the size. No noise has been applied yet to these chunks. At this point the world is a bunch of tiles and the surface of the world is completely flat. After the chunks have been generated, noise is applied to vary the surface height and create caves. For each surface location, an octave noise value is calculated and scaled to a max offset height determined by the size of the world and then the chunks of the world are trimmed away based on each of these height values. Then the world undergoes a cave-creation pass where each tile position is used to calculate a density value. This density value is scaled based on its depth to create caves that get smaller and trail more the closer they are to the surface. This gives results that look nice but they also more closely resemble a noise field than natural caves.





4. CONCLUSION

Although Simplex noise is a fast and easy way of implementing procedurally generated terrain, it produces terrain that doesn't look extremely natural. It in no way, with the current implementation, creates overhangs and interpolates between biomes. One way to do this would be to use 3D Simplex noise to create more complex surface geometry such as overhangs. To create different biomes, low frequency simplex noise could be used to create temperature maps for each world in order to determine the type of biome for that area in the world. Simplex noise is fairly fast, however, and definitely would be a good choice for generating terrain procedurally. There is very little overhead compared to classical Perlin noise and, with some additional modifications and terrain passes, could create some fairly interesting terrain.

5. ACKNOWLEDGMENTS

Special thanks go out to Jim Whitehead for teaching us all this cool stuff!

6. REFERENCES

[1] Persson, Markus. 2011. Terrain Generation, Part 1. *The Word of Notch.* 15, 5 (Mar. 2011), 795-825. DOI= http://notch.tumblr.com/post/3746989361/terrain-generation-part-1.

[2] Deadstone. 17 October, 2013. *A Note About Terrain Generation.* http://www.reddit.com/r/starbound/comments/1on4yz/a_note_about_terrain_generation/

[3] *Perlin Noise.* 18 March, 2015. http://en.wikipedia.org/wiki/Perlin_noise.

[4] *Simplex* Noise. 18 March, 2015. http://en.wikipedia.org/wiki/Simplex_noise.

[5] Gustavson, Stefan. *Simplex Noise Demystified.* Linköping University, Sweden. 22 March, 2005. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[6] Tippets, Joshua. 2011. *3D Cube World Level Generation.* http://accidentalnoise.sourceforge.net/minecraftworlds.html