Terrain synthesis using noise

Tuomo Hyttinen

University of Tampere
Faculty of Natural Sciences
Computer Science
M.Sc. Thesis
Supervisor: Timo Poranen
May 2017

University of Tampere
Faculty of Natural Sciences
Computer Science
Tuomo Hyttinen: Terrain synthesis using noise
M.Sc. thesis, 51 pages
May 2017

Noise functions are versatile base functions used in many procedural generation methods. They can produce natural-like patterns usable in procedural textures, models and animations. They have been extensively adapted in procedural terrain implementations in games and other applications. Noise-based procedural terrains offer many advantages over static terrain models but designing such terrains is largely unintuitive by nature. Whereas traditional terrain models can be designed, e.g., in spatial editors, procedural terrains are implemented in algorithms.

The purpose of this thesis is firstly to evaluate noise functions in the context of procedural terrain generation and especially example based procedural terrain synthesis. Secondly, a novel example based procedural terrain synthesis method is presented. A prototype application implementing the method was also constructed and evaluated. The prototype is a practical solution for example based procedural terrain design, aiming to bridge the gap between intuitive virtual terrain design and the advantages of procedural terrain functions.

Key words and terms: noise, procedural terrains, example-based terrain synthesis

Contents

1. Introduction

Noise is a function that takes an n-dimensional point, typically n=1, n=2 or n=3 and returns a real number [Perlin, 2001]. It's pseudo-random by nature, meaning that the result appears to be random, but given the same input coordinates the function always returns the same value. Noise is continuous and non-periodic – it covers the entirety of n-space without seams or repetition [Lagae et al., 2010a]. Depending on the noise implementation and parameters, various patterns can be seen in the result. For example, by using a two-dimensional noise one could produce a grayscale image that has seemingly random lighter and darker areas, with smooth transitions in between. Procedural noise can be used for creating patterns automatically and on the fly, rather than, for example, drawing a texture by hand [Smelik et al., 2009]. They often require very little memory to be executed and are randomly accessible, meaning that they can be evaluated independently for each input point. The random access feature, among its other benefits, makes extensive parallelism possible [Lagae et al., 2010a].

Perlin noise is a widely used and referenced noise function that was originally designed by Ken Perlin related to his work with the special effects in Disney's science fiction film Tron in 1982. Later Perlin was awarded an Oscar for his work with computer generated imagery in movies. Perlin noise was originally designed for creating textures [Perlin, 1985], but noise functions also have usages in many other areas of computer graphics. Procedural generation, often utilizing Perlin noise and other noise functions, is widely used today in software. Noise functions have been used for creating natural-like content such as textures, models and animations in 3D modeling software, animation films and other computer graphics applications [Lagae et al., 2010a]. In games it has been used for decades for creating large game worlds [Smelik et al., 2009].

Noise functions are often used in games for creating large virtual worlds, including their terrain topology, texturing and cave systems etc. [Santamaría-Ibirika et al., 2014]. Minecraft, released in 2011, is perhaps today's most well known example of using noise functions and procedural terrain generation in games. It's the best selling video game of all time, having sold over hundred million copies world wide (including mobile, pc and console versions sales) [Hill, 2016]. It features a near-infinite procedurally generated world. As seen in Figure 1, the game world features a variety of visually distinct, procedurally generated landscapes. The game world is created by using several 2 and 3-dimensional Perlin noise functions [Persson, 2011]. Because of the procedural noise based approach the terrain can be generated in chunks as the player advances in the game world and the resulting terrain is the same regardless of, say, from which direction the player approaches a certain area. There are also many other voxel-based games with procedurally generated worlds – a popular gaming platform Steam even lists "Procedural generation" as its own genre of games. Especially indie-developers seem to have adopted procedural generation. It makes creating large, interesting game worlds possible without the massive work of designing and modeling them manually.

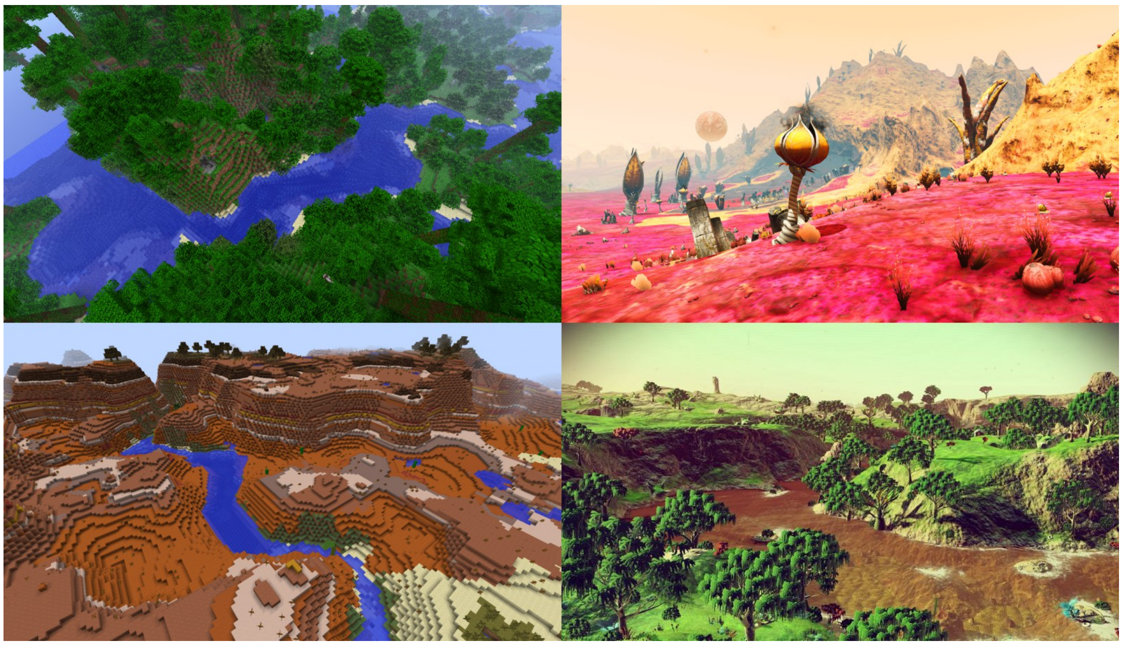


Figure 1: Procedural terrain examples from Minecraft (left) and No Man's Sky (right) [http://minecraft.gamepedia.com, http://nomanssky.gamepedia.com].

No Man's Sky (2016), also an indie game, is another example of extensive use of procedural generation. It was four years in production and highly anticipated before release. The game world of No Man's sky is an entire universe, containing 18 quintillion planets that can be visited by player [Parkin, 2014]. Planets in turn are procedurally generated, each containing a planet sized world to explore. Examples of the procedurally generated terrain are shown in Figure 1. The game can be considered a commercial success [Yin-Poole, 2016], but it received crushingly negative reviews shortly after release mostly due perceived lack of promised gameplay features. Despite its shortcomings it can still be considered a prime example of the possibilities of procedural generation in games. The enormously large multi-player game world could not have been designed by hand or stored in memory – the randomly accessible, pseudo-random nature of noise functions and other procedural generation techniques make this kind of game world possible.

Procedural noise has been extensively used for creating procedural worlds and their terrain. Noise is the powerful base element of the procedural techniques used to generate complex terrains. Procedural terrain allows infinite variations of terrain features to be created from an extremely compact function representation. However, designing noise summations and modeling to achieve desired results usually requires extensive manual work. Noise can only be controlled in a global scale, which makes the design process inherently unintuitive. Therefore it would be useful if combining and parametrizing noise could be done automatically by using an example height map from which noise parameters are extracted.

There are various design tools for creating virtual landscapes for games, animation etc. However, while these tools make use of procedural methods to create the terrain, the resulting terrain is usually static and finite – the end product is not a procedural terrain. In this thesis the aim is to construct a method for synthesizing features of an example terrain topology using noise functions. This is achieved by analyzing statistics of the reference terrain to automatically parametrize and combine noise to generate procedural terrain with similar features. The end result is a noise based procedural terrain function – not, e.g., a finite terrain model. Synthesis methods are implemented in a prototype which aims to make the noise based procedural terrain design process more intuitive. Instead of the use of unintuitive parameters, the design process relies on example height maps. The prototype is practical solution to the problem of designing noise based procedural terrains. The resulting procedural terrain function should retain following properties of a true procedural noise:

- *Random accessibility, not based on discretely sampled data*: Height value at any given point can be calculated without access to any previously calculated values. In a sense, this amounts to a pure function with no side effects.
- *Non-periodicity*: Capable of creating near-infinite procedural terrains with desired features, without seams or repetition.
- *Low storage*: Partly contained in the previous requirement, the function's storage requirement should be very low compared to e.g. sampling non-procedural height maps.

Resulting procedural terrain function should also be fast enough to allow usage in applications with real-time rendering. Additionally, the terrain design process should intuitive and user friendly, allowing procedural terrains to be generated using example height maps, hiding the noise parametrization process from end user. In other words, ideal solution would allow user to design a traditional, finite size height map and use it to generate similar but infinite procedural terrain.

First, definitions of noise and theoretical background are given. Then various implementations of noise and their features and usefulness in terrain synthesis are covered. Related work on procedural noise based terrain generation and example based terrain synthesis are then assessed. Final part of the thesis focuses on constructing a prototype that aids the procedural terrain design process by extracting terrain features from a sample terrain height map and uses procedural noise functions to semi-automatically reproduce them in a procedural terrain. The constructed synthesis method and the thesis in general mainly focus on noise based procedural terrains that utilize two-dimensional noise.

2. Noise function fundamentals

2.1. Defining noise

Noise is repeatable pseudorandom function of its inputs [Ebert et al., 2002]. Outputs of a pseudo-random function are apparently random, but always return the same value given the same input. An ideal procedural noise is non-periodic [Lage et al., 2010a]. In practice, noise implementations are always periodic, but the period can be very large – thus periodicity is not apparent in applications [Ebert et al., 2002]. The actual range of an ideal noise function implementation is limited only by the storage space reserved for numeric data types used. Noise is also randomly accessible. It can be evaluated for each input point in constant time without access to any previously calculated values [Lagae et al., 2010a]. Thus, noise functions can generate extensively large, seamless patterns that can be generated on-demand in batches. An ideal noise function is stationary and isotropic. This means that the noise's statistical characteristics are not affected by rotation or translation of its coordinate system. [Ebert et al., 2002]. However, it should be noted that in context of synthesizing terrain by example some anisotropic properties may be desirable, if they can be accurately controlled, because the reference terrain could have anisotropic features.

Methods and terminology from Fourier analysis and the field of signal processing in general are useful when describing noise functions [Lagae et al., 2010a]. Frequency domain is used to represent a function or signal by specifying the amplitude and phase of each of its frequencies, instead of plotting its value at each location. In frequency domain representation, amplitude refers to the strength of each frequency that together form the signal. Signal is represented as sum of a number of sinusoids. [Smith, 1997] In other words, any signal can be represented as a sum of sinusoidal waves. Phase represents the time offset that must be applied to the sinusoids forming the signal in order to reproduce the original signal. The processes of transforming a signal into the frequency domain and back to spatial domain are known as Fourier transform and inverse Fourier transform, respectively. For a fixed set of data points, the method is called discrete Fourier transform. Noise function represents a random pattern that is better described in the frequency domain as opposed to spatial domain. However, noise is unstructured and therefore the phase information is not relevant in describing it. Thus, noise is completely characterized by its power spectrum, which describes only the magnitude of each of its frequencies. Discrete Fourier transform produces complex numbers. These numbers can be represented using two sets of real numbers, one for magnitude and the other for phase. In case of noise function, all relevant information is contained in the first set. Tasks involving noise can be described as manipulating the power spectrum, e.g., modeling noise can be seen as shaping the power spectrum whereas filtering corresponds to damping of certain frequencies [Lagae et al., 2010a].

Characteristics of a noise function, and also value maps, like terrain height maps, can be described using nth-order statistics. First order statistics refer to the probability distributions of individual values [Field, 1989]. In case of a noise function, probability density function is a first order statistic that describes the amplitude distribution of the noise [Lagae et al., 2010a]. First order statistics are restricted to individual values, and thus they can not represent “unstructured” aspects of the data [Pouli et al., 2011]. For example, randomly permuting pixels of an image produces an image with identical first order statistics. Despite this, the resulting image naturally looks nothing like the original one. Second and higher order statistics can be used to more accurately describe characteristics of noise. Second order statistics refers to the relations between pairs of values [Pouli et al., 2011]. Autocorrelation is a measure of the correlations between values as a function of the distance between them and, finally, the power spectrum is the Fourier transform of the autocorrelation function [Field, 1989; Pouli et al., 2011].

Reproducing terrain features using a height map as a source effectively means also reproducing the source height map's statistics. Julesz [1962] also states that textures with similar first and second order statistics are difficult to discriminate by humans. Therefore, reproducing these statistics may work as a key for automatically reproducing source terrain height map features.

2.2. Simple value noise function

The most simple example of a noise function implementation is probably value noise. It models band limited white noise. In white noise, all frequencies contribute the equal amount. Band limiting refers to reducing the power of certain frequencies in the noises power spectrum to zero [Lagae et al., 2010a]. Band-limited white noise serves as a basis for more sophisticated noise functions [Perlin and Hoffert, 1989]. Lagae [2010a] notes that the term “band limited” is commonly used in previous research in place of more appropriate term “band pass”. Band pass correctly refers to limiting signal to zero outside a certain frequency interval whereas band limiting actually means eliminating frequencies above a certain threshold.

A simple, one dimensional value noise implementation could take a single floating-point value – a coordinate, and also return a float value. Depending on the purpose, a simple hash function could be used to generate sufficient amount of randomness to simulate white noise [Ebert et al., 2002]. For demonstrating a simple noise function this is sufficient. The function's value could be calculated by first taking the integer part of the input coordinate and applying an integer hash function to the result. These integer points at which the white noise is sampled will be referred as lattice points. After calculating the surrounding lattice point values for given input value, some sort of interpolation is required for smooth transitions between lattice points. For example, for input value 1.5, the values at the lattice points 1 and 2 contribute the equal amount.

The same approach can be generalized to higher dimensions. For the purposes of this thesis, 2-dimensional noise functions are the most interesting, because they can straightforwardly be used to generate terrain height maps. Two dimensional version takes two values – the x and y coordinates. The function in pseudo-code is given in Code fragment 1.

```
float noise(float x, float y) {
  // find starting lattice point coordinates and corresponding deltas
  int xi = floor(x)
  int yi = floor(y)
  float tx = x - xi
  float ty = y - yi

  // get hash function value at the 4 surrounding lattice points
  float sw = hash(xi,   yi)
  float se = hash(xi+1, yi)
  float nw = hash(xi,   yi+1)
  float ne = hash(xi+1, yi+1)

  // smoothstep - remap the deltas
  float sx = smoothstep(tx)
  float sy = smoothstep(ty)

  // linear interpolation along both axis
  float s = lerp(sw, se, sx)
  float n = lerp(nw, ne, sx)
  return lerp(s, n, sy)
}
```

Code fragment 1: Simple value noise function.

First, the closest integer points and corresponding deltas are calculated for the input coordinates. Then the function calculates the pseudo-random hash function value at the four integral lattice points, or “corners” surrounding the input point. The hash function simply maps the two integer values to a pseudo-random float value in a predefined output range. The smoothstep part changes the resulting noise pattern slightly. Smoothstep function is widely used in computer graphics and also in Perlin's original noise implementation [Perlin, 2002]. Perlin noise will be covered later in the thesis. Smoothstep function is defined as $f(x) = 3x^2-2x^3$. The function produces a smooth curve in range $(0,1)$. The plot of the function from this range is given in Figure 3.

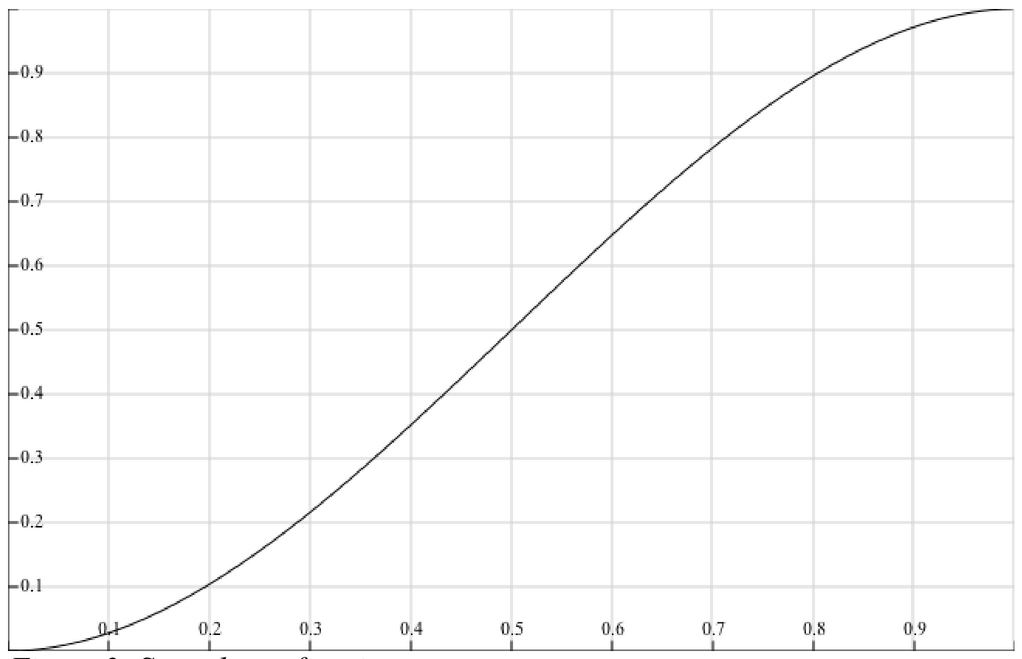


Figure 2: Smoothstep function.

Because the delta parts *tx* and *ty* calculated in Code fragment 1 are in the range (0,1), the smoothstep function can be used to remap these delta parts of the input coordinates to produce more visually pleasing result. In other words, it remaps the fractional parts or deltas between lattice points, making transitions between the lattice point values less linear. Lastly, linear interpolation is used along both axis to calculate the final value using the remapped axis-specific deltas and values from surrounding lattice points. Figures 3 to 5 display the result of the function for input coordinate ranges $[0,20]$. Figure 3 displays the result of the raw hash function for the integer parts of the input coordinates – both the smoothstep and linear interpolation are removed. Figures 4 and 5 display the result with linear interpolation and with smoothstep remapping of the input coordinates respectively.

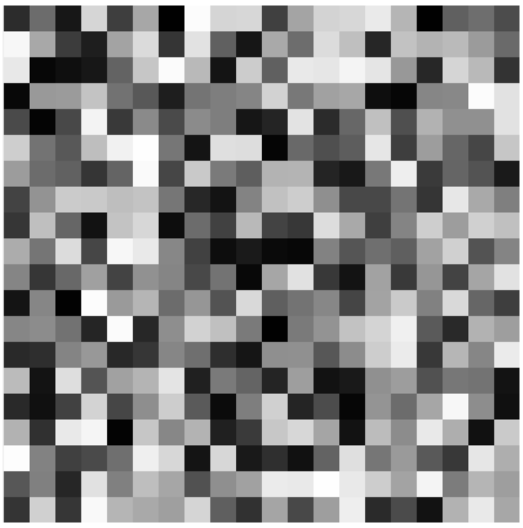


Figure 3: Raw hash values of the lattice.

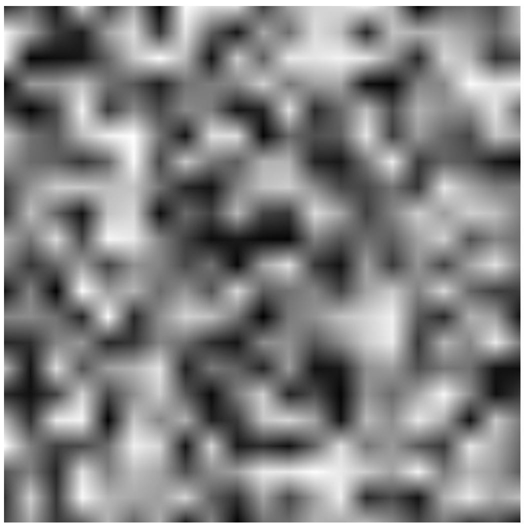


Figure 4: Linear interpolation.

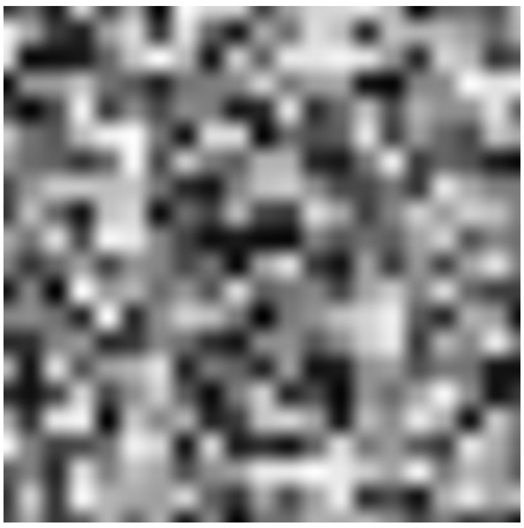


Figure 5: Linear interpolation and smoothstep.

The implementation is an example of procedural noise. It's randomly accessible – value at each point can be calculated individually with no relations to results at other points. The individual pixel values could be calculated in batches of any size or concurrently. Also, the input coordinate range is not bounded and the periodicity is determined only by the hash function implementation. However, as seen in Figures 4 and 5, this simple approach does not provide satisfactory results in terms of generating natural-like patterns. Directionality of the underlying axis aligned grid is clearly visible regardless of the interpolation method used, even though smoothstep function makes it a little less obvious. Clearly, the noise is not isotropic and the resulting pattern does not appear natural. Better implementations of noise functions will be discussed in the following chapters. However, this implementation of noise is not useless – it could be applied on top of itself multiple times using different scalings and magnitudes for each “octave” in order to produce fractal noise and more natural looking patterns. An octave of noise usually refers to a scaled version of the noise, that can be achieved by e.g. scaling the input coordinates. Fractal noise is also described in more detail later in the thesis.

2.3. Comparison to other methods

In terms of generating pseudorandom height maps, subdivision based methods are among the oldest [Smelik et al., 2009]. They are fundamentally different from noise functions, but can also be used to create pseudorandom height maps. Mid-point displacement is the simplest subdivision based method. The algorithm starts by assigning a random value to each corner of a fixed sized height map grid. Then it progresses by assigning the average height calculated from the corner points of each edge to the center point and adding a controlled random value. The center point is then calculated in similar fashion, using the previously calculated four edge values. Then the algorithm continues recursively until the whole map is filled. The Diamond-square method is a slightly more advanced subdivision method. It samples surrounding points in diamond and square shapes consequently. Both algorithms produce directional artifacts and improved algorithms have been presented addressing this problem [Miller, 1986]. Subdivision based methods are obviously very simple and fast compared to noise functions. But they have the disadvantage of requiring a known fixed size domain and intermediate values have to be held in memory during computation. They are not well suited for producing height map in batches or in parallel. For example, generating a currently rendered part of the height map first and then calculating a neighboring neighboring area later would almost unavoidably produce a visible seam between the maps.

Subdivision methods have been successfully used for generating terrains procedurally. Kamal and Uddin [2007] describe a subdivision based method for parametrically controlled terrain generation. The aim is to automatically produce terrain models based on input criteria and synthesize real world-like terrain. Parameters such as mountain height and mountain range spread direct the underlying subdivision to produce desired result. The result is a static terrain model of predefined size and level of detail. It offers control over the subdivision but still produces finite terrains that can not be calculated in a random accessible manner.

Fourier synthesis methods can also be used to generate pseudo-random patterns, including height maps [Ebert et al., 2002]. Like the previously described techniques, they represent an explicit way of creating pseudorandom patterns. A pseudorandom discrete frequency spectrum with desired frequency distributions is first created. Then discrete inverse Fourier transform is used to get a spatial domain representation of the data. Another approach would be to use an approximation of white noise as a basis and use a discrete Fourier transform to translate the data points into a frequency domain representation. The frequency domain representation of the data is then altered by damping some higher frequencies to produce a pattern representing band limited white noise. Finally, inverse Fourier transform is used to bring the band-limited version of the data back to spatial domain. The resulting pattern still pseudorandom, but as lower frequencies are more dominant, visible larger patterns start to emerge.

Advantage of Fourier synthesis methods over noise is the ease of spectral control, thus the resulting statistics are easier to control. But the required transforms between spatial and frequency domain make the methods much slower than procedural noise function based approaches [Ebert et al., 2002].

2.4. Noise classifications

Noise functions can be classified in a number of ways, including classification as value noise, gradient noise and lattice noise [Lagae et al., 2010a]. Lattice noises are the most popular noise implementations. Lattice based approach is simple and efficient, used also by the famous Perlin noise. Lattice noise works by retrieving pseudo-random values at lattice points and interpolating between them. Typically the values are uniformly distributed in an integer lattice [Ebert et al., 2002]. The lattice point values can be calculated either by using a hash function on the fly for the input coordinates or calculating a pre-defined hash table from which the lattice point values are retrieved. The first presented noise implementation in Code fragment 1 is an example of lattice noise – the pseudo-random values are accessed in an integer lattice and interpolated.

Lattice value noise is the simplest way to generate a noise function. Lattice point values are simply floating point values. The first noise implementation presented in the thesis is also an example of value noise, the randomness comes from pseudo-random values assigned to lattice points. Whereas value noise is based on individual pseudorandom values, gradient noise is based on gradient vectors. Gradient approach introduces less visible artifacts in the result pattern.

Lagae et al. [2010a] also make a distinction between procedural and explicit noise. Explicit noise functions lack some characteristics of procedural noise, e.g., they do not have the random access feature – instead the values are computed explicitly for a fixed size data set. Because intermediate values of other points need to be accessed during computation, explicit noise has potentially very large memory footprint. They may also not have the procedural noise's support for parallel computation. This thesis will mostly focus on finding methods for terrain synthesis using true procedural noise functions due to their obvious advantages in generating seamless near-infinite maps that can be calculated in chunks. This kind of procedural terrain is useful in applications where very large, detailed and innovative terrains are needed and holding the whole terrain model in memory or calculating it beforehand in full would be inconvenient or impossible.

3. Noise functions

Various noise functions have been created addressing issues such as performance, aliasing, computational complexity, statistical and spectral characteristics. The resulting base patterns are also distinct for each noise implementation. In context of terrain height map generation noise functions work as the workhorse of the methods – they can be combined and parametrized to obtain an infinite amount of distinct results. In this chapter a few well-known, good quality noise functions are assessed.

3.1. Perlin noise

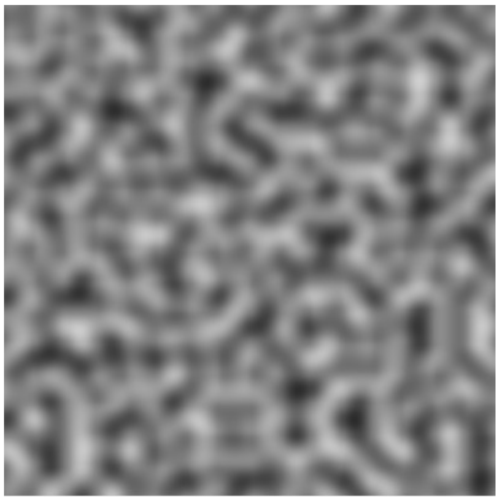


Figure 6: Perlin noise.

Perlin noise is a famous lattice gradient noise function introduced by Ken Perlin in 1985. Its smooth random pattern can be seen in Figure 6. It was the first implementation of gradient noise [Ebert et al., 2002]. The function is fast, simple and still extensively used in software today [Lagae et al., 2010a]. It can be used to generate various natural-looking patterns, models and textures, as described by Perlin [1985, 1981]. These include soft materials like cloth, marble, fire etc. 3-dimensional version can be used to generate smoothly animated textures, like fire, by using coordinates as the first two parameters and elapsed time as the third parameter. This way the resulting texture changes smoothly by changing the "slice" of the 3-dimensional noise as a function of time. The same approach can even be used for animated 3-dimensional volume rendering, using 4-dimensional noise with time as the fourth parameter.

Like the simple value noise implementation presented earlier, Perlin noise uses an integer lattice. But instead of using raw pseudorandom values for lattice points, it uses a gradient vector assigned to each of them [Perlin, 1985]. Gradient is simply an unit length vector in n dimensions, or a value -1 or 1 for the 1-dimensional case. A pseudorandom gradient is assigned for each lattice point by using the lattice point hash value. Originally pre-calculated lookup tables were used for the gradients. In any case, gradients to be used should be uniformly distributed around unit circle for 2-dimensional version or unit sphere for 3-dimensions to avoid directional artifacts [Perlin and Hoffert, 1989]. Original Perlin noise used the smoothstep function described earlier to remap the input deltas. However, the function resulted in visible artifacts when a surface was created using the noise result for displacement and then shaded. These artifacts were caused by the resulting surface normals having discontinuous derivatives. This was caused by the smoothstep function's derivative, which has a linear term, resulting in discontinuations at the lattice points [Perlin, 2001]. Perlin suggested improvements for the original implementation, including replacing the smoothstep function with $f(x) = 6x^5 - 15x^4 + 10x^3$. This function has continuous first and second derivatives at both x=0 and x=1, unlike the original smoothstep function, and it does not produce the shading artifacts. However, it's somewhat more computationally expensive to evaluate. Perlin [2002] also notes that the gradients themselves don't actually need to be random at all and a fixed set of gradient vectors is enough as long as they are randomly chosen for each lattice point.

The four neighbor values of an input point are calculated as dot product between the pseudorandom gradients and distance vectors from input point to the corner points. [Perlin, 1985]. Because the distance vector components are in range $[0,1]$ and the pseudorandom gradient is normalized, the result is in range $[-1,1]$. Interestingly, because the dot product with distance vector of zero length is zero, the noise function value at the lattice points evaluates to zero (which maps to a grey value in the image, exactly between black and white). This characteristic is not apparent looking at the resulting noise pattern in Figure 6. Musgrave [1993] notes that the function can be modified to have an arbitrary, non-zero value at the lattice points. Doing so increases the variance and adds low frequency energy to the signal. In the last step, linear interpolation is used along both axis in the same way as in the simple noise example. The resulting pattern (for 2-dimensional version) seen in Figure 6 is smooth and does not contain obvious directional artifacts. The function's complexity is $O(2^n)$ for n-dimensions.

3.2. Simplex noise

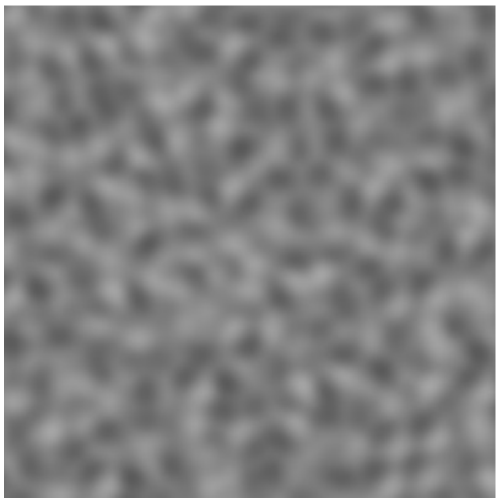


Figure 7: Simplex noise.

Simplex noise produces roughly the same looking patterns as the original Perlin noise implementation, as seen in Figure 7. It was designed by Perlin to address some shortcomings of the original noise. Firstly, Perlin noise is not visually isotropic. [Perlin, 2001] In other words, the rotation of the coordination system can be easily perceived in the resulting pattern. This is due to the use of axis aligned lattice. Also, the computational cost of implementing higher dimensional versions becomes very large, because the algorithm relies to an n-dimensional hypercube lattice where 2^n data points need to be calculated and interpolated for each input coordinate. This effectively means two dimensional lattice in the 2D case and cubic lattice in the 3D version. The cost is not a significant issue in two dimensional version commonly used for height map generation, but the $O(2^n)$ complexity obviously becomes an issue in higher dimensions.

Simplex noise addresses the hypercube lattice issue by introducing a simplex grid instead. Simplex represents the simplest possible shape that can be tiled to fill an n-space and the grid consist of these shapes. For two dimensional version this means creating a skewed grid consisting of tiled triangles. For three dimensions, the shape is a skewed tetrahedron and only four lattice point values need to be computed for each input point instead of the eight corners of the cube in original Perlin noise. The function's complexity is $O(n^2)$ for n dimensions, so the functions is significantly better suited for higher dimensions [Perlin, 2001]. Simplex based approach also eliminates visual directionality in the resulting pattern. It also addresses the derivative discontinuation problem of the original Perlin noise. Derivative functions of a 2D slice through 3D noise at z=0 of Perlin noise and Simplex noise are visualized in Figure 8. Discontinuities in the lattice grid boundaries are clearly visible in the old Perlin noise derivative pattern. Perlin noise is labeled as “old” and Simplex noise as “new” in the figure.

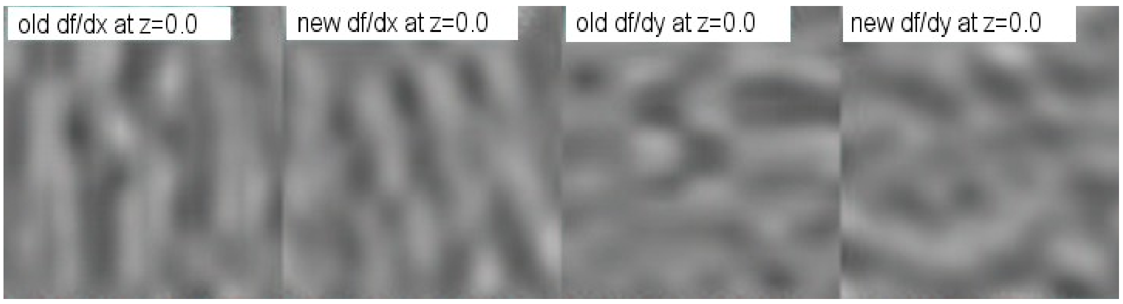


Figure 8: Perlin noise and Simplex noise derivative comparison [Perlin, 2001].

Simplex noise is a gradient noise and it still uses gradient vectors as in Perlin noise, but the gradients are accessed via the simplex grid instead of a cubic lattice. Input coordinates are “skewed” in order to access the corresponding simplex grid points. Interpolation step of the Perlin noise function is replaced with summation of contributions from surrounding simplex grid gradient values. This is faster for higher dimensions, as the n-linear interpolation for n-dimensions is not needed. The method also does not produce directional artifacts that arise from the use of bilinear interpolation in the axis aligned lattice (in the 2D case). As seen in Code fragment 2, Perlin's reference implementation in Java is very compact and highly optimized, consisting of only 33 lines of code. It's meant to be a reference for hardware implementations and does not use lookup tables for gradient values like Perlin noise. It utilizes bit manipulation to determine the pseudo-random gradients to be used on the fly. Gradient vector is defined using only bit operations and two additions, with no multiplications.

```
public final class Noise3 {
      static int i,j,k, A[] = {0,0,0};
      static double u,v,w;
      static double noise(double x, double y, double z) {
            double s = (x+y+z)/3;
            i=(int)Math.floor(x+s); j=(int)Math.floor(y+s); k=(int)Math.floor(z+s);
            s = (i+j+k)/6.; u = x-i+s; v = y-j+s; w = z-k+s;
            A[0]=A[1]=A[2]=0;
            int hi = u>=w ? u>=v ? 0 : 1 : v>=w ? 1 : 2;
            int lo = u< w ? u< v ? 0 : 1 : v< w ? 1 : 2;
            return K(hi) + K(3-hi-lo) + K(lo) + K(0);
      }
      static double K(int a) {
            double s = (A[0]+A[1]+A[2])/6.;
            double x = u-A[0]+s, y = v-A[1]+s, z = w-A[2]+s, t = .6-x*x-y*y-z*z;
            int h = shuffle(i+A[0],j+A[1],k+A[2]);
            A[a]++;
            if (t < 0)
            return 0;
            int b5 = h>>5 & 1, b4 = h>>4 & 1, b3 = h>>3 & 1, b2= h>>2 & 1, b = h & 3;
            double p = b==1?x:b==2?y:z, q = b==1?y:b==2?z:x, r = b==1?z:b==2?x:y;
            p = (b5==b3 ? -p : p); q = (b5==b4 ? -q : q); r = (b5!=(b4^b3) ? -r : r);
            t *= t;
            return 8 * t * t * (p + (b==0 ? q+r : b2==0 ? q : r));
      }
      static int shuffle(int i, int j, int k) {
            return b(i,j,k,0) + b(j,k,i,1) + b(k,i,j,2) + b(i,j,k,3) +
                  b(j,k,i,4) + b(k,i,j,5) + b(i,j,k,6) + b(j,k,i,7) ;
            }
      static int b(int i, int j, int k, int B) {
            return T[b(i,B)<<2 | b(j,B)<<1 | b(k,B)];
      }
      static int b(int N, int B) { return N>>B & 1; }
      static int T[] = {0x15,0x38,0x32,0x2c,0x0d,0x13,0x07,0x2a};
}
```

Code fragment 2: Simplex noise reference implementation [Perlin, 2001].

Despite its advantages, Simplex noise is not quite as widely used as the original Perlin noise. Perlin noise is considered good enough for many applications and it's easy to implement. Simplex noise is also covered by US 6867776 B2 patent, which may have reduced its popularity among game developers.

3.3. Wavelet noise

Wavelet noise, introduced by Cook and DeRose [2005], addresses limitations when Perlin noise is added into itself to form fractal noise. These limitations include a tradeoff between loss of detail and aliasing, which is a problem when fractal Perlin noise is used to generate textures. Adding a high frequency band adds detail, but it includes aliasing artifacts. In order to fight aliasing in Perlin noise case, high frequency bands need to be attenuated and a considerable amount of detail in surfaces farther in the distance is missing as result, making them look foggy.

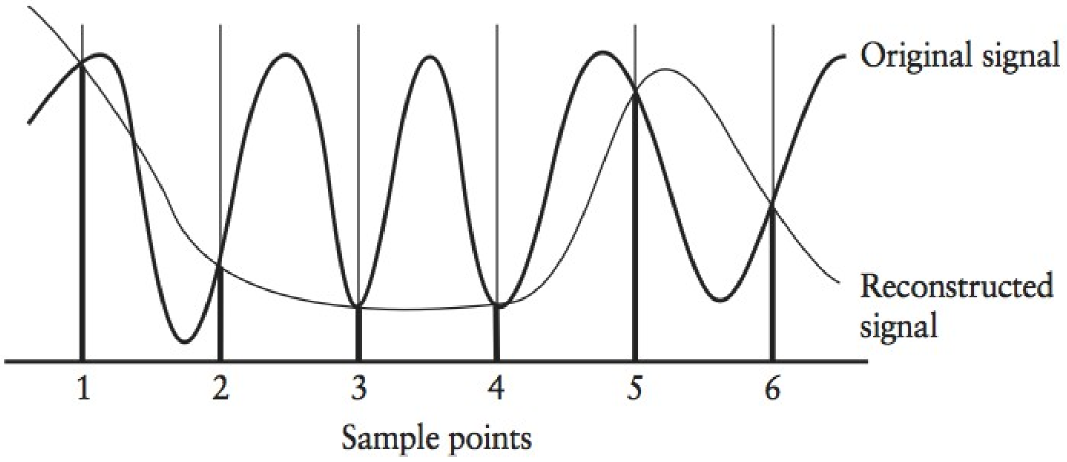


Figure 9: Aliasing [Ebert et al., 2002].

Aliasing occurs, when the maximum frequency of the original signal exceeds one-half of the sampling rate used. This is known as the Nyquist frequency [Ebert et al., 2002]. In the case seen in Figure 9, the sampling rate used fails to accurately capture the original signal and the reconstructed function (based on samples) contains erroneous low frequency energy – an alias of the higher frequency energy of the original signal. Reconstructed signal differs considerably from the original signal.

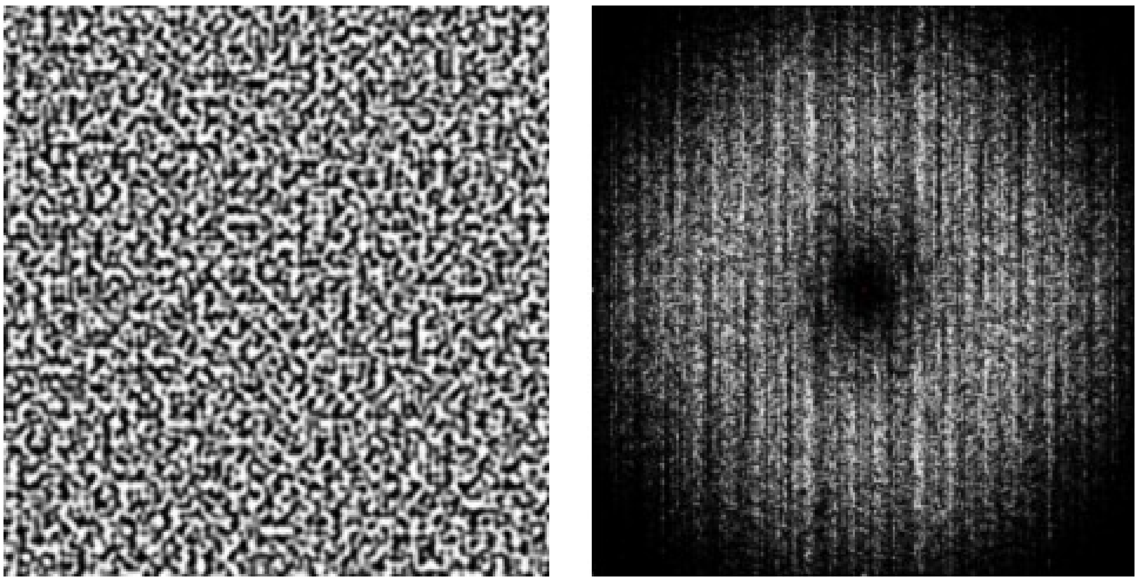


Figure 10: Perlin noise periodogram [Cook and DeRose, 2005].

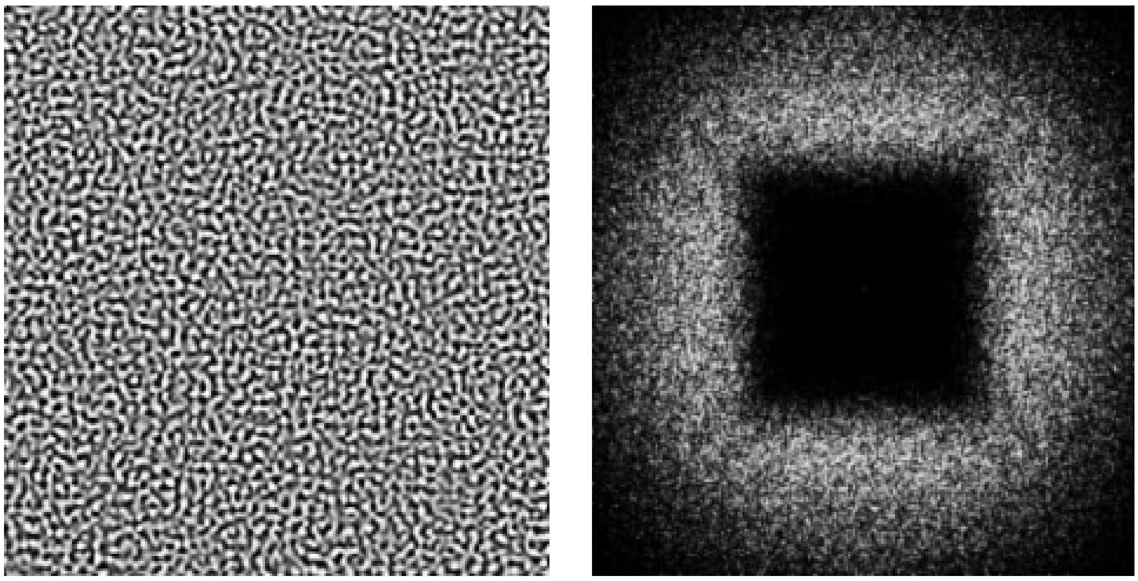


Figure 11: Wavelet noise periodogram [Cook and DeRose, 2005].

Single octave of Perlin noise consist of a band of frequencies, and when summing noises of different octaves, only the magnitude of each band as whole can be controlled for each octave. Thus accurate spectral control of the resulting noise is not possible. Cook and Derose [2005] note that Perlin noise is only weakly band limited, which causes the aliasing problems when summing the noise. Figures 10 and 11 show periodograms for 2D Perlin noise and Wavelet noise. Periodogram is defined as the magnitude squared of the Fourier transform [Lagae et al., 2010a]. Lighter colored pixels represent frequencies with larger magnitude. The center of the Fourier transform image represents the contribution of the lower frequencies, center pixel being the DC term – zero frequency. Wavelet noise cuts the limited frequencies very accurately, wheres a simple band of 2D Perlin noise contains a wide range of frequencies. The result of 2D slide through 3D Perlin noise is even less band-limited, closely resembling white noise. Because of the well limited bands, summed Wavelet noise's total power spectrum can be more easily controlled.

Wavelet noise requires a preprocessing step, where quadric B-spline coefficients are calculated to be used later in the noise evaluation. In practice, an image of white noise is first downsampled to create a representation of the original image with half the size. Then it's upsampled to the original size and subtracted from the original image. [Lagae et al., 2010a] The result is the part that is not representable at half the size – the band limited part, with the frequencies $|x| < 0.5$ and $|y| < 0.5$ cut [Cook and DeRose, 2005]. In the actual noise evaluation, a quadric B-spline function is evaluated at each input point using the previously calculated coefficients.

Whereas Perlin noise and Simplex noise are types of procedural noise, wavelet noise is explicit by nature. It requires a preprocessing step, in which a fixed size volume containing the coefficients that need to be calculated. The noise result will be tiled according to that volume [Lagae et al., 2010a]. The lack of true random accessibility introduces memory consumption problems typical to explicit noise when very large random patterns need to be created. While Wavelet noise is superior in generating textures without aliasing artifacts and the spectral control of the noise, it has the disadvantage of being limited to more or less repeating patterns, or at least solving the problem of seams between randomly generated tiles. It's perhaps better suited for what it was designed, that is, for texture generation. Avoiding aliasing and precise spectral control are more important and the size of the domain is known beforehand. But creating infinite terrains with certain characteristics is much less convenient without the true random access feature.

3.4. Worley noise

Worley noise is very different from the functions described earlier. It was presented by Worley in 1996 as a texture basis function and can produce various kinds of Voronoi-like patterns, useful for creating, e.g., rock and brick patterns or craters. [Lagae et al., 2010a] Instead of using points assigned to lattice points, it used randomly scattered points and the noise at a given location is derived from the distance to the nth closest feature point [Worley, 1996]. The function could also be implemented in a way that returns the result for n closest points, respectively. These results could then be used later to make different kind of patterns [Ebert et al., 2002]. Worley noise can be implemented as randomly accessible [Worley, 1996], which makes it interesting for terrain generation as well. In this case the the points contributing to an input point are calculated on the fly using similar kind of techniques described in Perlin noise gradient selection. Cellural patterns of Worley noise could be useful in terrain synthesis as well. However, it's a texture basis function by nature and does not aim to offer precise spectral control as some other types of procedural noise.

3.5. Gabor noise

Gabor noise is a type of sparse convolution noise [Lagae et al., 2009]. Sparse convolution noises generate noise by summing randomly positioned and weighted kernels [Lagae et al., 2010b]. Gabor noise uses a Gabor kernel introduced in 1946 by Gabor to study communication theory [Lagae et al., 2009]. Kernels are commonly used in image filtering, where they can be viewed as matrices that map each pixel value to another value using neighboring pixel values. Gabor kernel consists of a Gaussian envelope multiplied by a harmonic, two dimensional cosine in this case. Figure 12 visualizes the Gaussian envelope, the harmonic and resulting Gabor kernel.

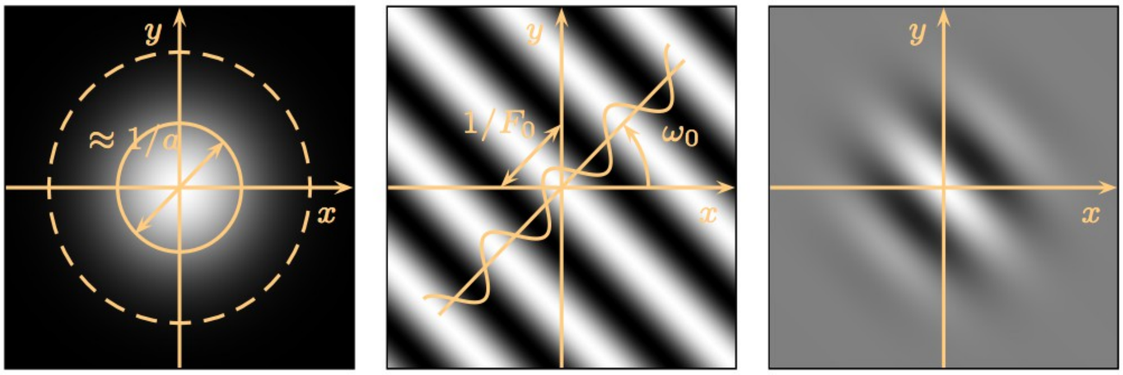


Figure 12: Gaussian envelope, cosine harmonic and resulting Gabor kernel [Lagae et al., 2009].

The power spectrum properties of Gabor kernel can be parametrized by setting parameters of the kernel for the principal frequency and width of the frequency band around the principal frequency to be used. The actual noise result is the convolution of the kernel and a impulses from a Poisson impulse process. [Lagae et al., 2009] Poisson impulse process refers to impulses of uncorrelated intensity at uncorrelated locations – it's the source of pseudo randomness in sparse convolution noises. Poisson impulse process is also referred to as sparse white noise because of its properties similar to white noise but as sparsely distributed impulses. Each input point may receive contribution from multiple impulses. [Lagae et al., 2010b] The resulting noise has the power spectrum of the kernel used, and thus it can be accurately controlled by parametrizing the kernel. [Lagae et al., 2009] Gabor noise utilizes a virtual grid, cell size of which equals the radius of the Gabor kernels used. Given an input point, only cell containing the input point and direct neighboring cells are needed for the evaluation. The properties of the Gabor kernels to be used for each point are generated on the fly using pseudo-random number generator, quite similarly as in selecting gradients or Worley noise's contributing points. The contributing impulses for each input point are also generated on the fly. As a result, Gabor noise is randomly accessible. However, it requires more computation per point compared to, say, lattice noises, since nine contributing kernels need to be used for each input point for the 2D case and the actual performance also relies on number of impulses that have been assigned per cell.

3.6. Comparison of the noise functions

Accurate band limiting of noise is important if the noise is to be summed together to reproduce a certain power spectrum. If an octave contains undesired frequencies, random phase of the noise makes it impossible to negate these frequencies later in the process, making accurate power spectrum reconstruction impossible [Lagae et al., 2010a]. Original Perlin noise or even value noise are easy to implement and work well for lower dimensions and when accurate spectral control is not needed. Simplex noise offers the best performance especially in higher dimensions and it also addresses some shortcomings of Perlin noise. But in order to synthesize terrain using power spectrum reconstruction based approach, noise functions with more accurate spectral control are needed. Wavelet noise offers more precise spectral control and quick per point evaluation, but in the expense of high storage requirements and a preprocessing step. Gabor noise has the most precise spectral control of the noise functions covered in this thesis, but it comes in the expense of some additional computation. Nevertheless, it retains many properties of an ideal procedural noise.

4. Related work on procedural generation

4.1. Noise-based procedural terrains

Proceduralism refers to a discipline of computer graphics that abstract the underlying form to be represented into a compact, elegant procedure or a function representation that can be lazily evaluated to produce complex models [Musgrave, 1993]. The function representation stores very little descriptive information – the focus is shifted into how to generate the model using a procedure instead of how to store a specific description of a model. In other words, overhead is shifted from storage space requirement to computational expense.

Fractional Brownian motion or FBm is a stochastic fractal function characterized by it's statistical behavior. It can be described with a function that consists of an infinite sum of sine waves with varying frequency and random phase shifts with amplitude decaying as a function of frequency. Weierstrass-Mandelbrot function for fractional Brownian motion is given as

$$V(t)=\sum_{f=-\infty}^{\infty} A_f r^{fH} \sin(2\pi r^{-f} t+\theta_f),$$

where A is a Gaussian random variable, r is the lacunarity – gap between summed fractals, θ is a uniform random variable for generating the random phase and H stands for Hölder exponent determining fractal dimension. Fractals are a more general concept that relate to fractal Brownian motion. They are defined with recurrance relations, e.g.,

$$x_{(i+2)} = f(x_{(i+1)}) = f(f(x_i)).$$

This maps to recursive ways of writing graphics procedures for computers. In computer graphics it's usually a good idea to terminate the process before reaching infinite level of detail – sufficient level of detail is determined by the sampling rate used. The recurrence adds to the realism and detail of a synthesized model. For example, when rendering a terrain model it does not have to be explicitly defined to the smallest detail – instead the detail is stored in the fractal terrain function and can be accessed when smaller details actually need to be rendered. Fractal procedural terrain's level of detail can be nearly infinite, while traditionally stored terrain model or, e.g., subdivision based procedural terrain has some fixed level of detail.

Fractal noise based terrains are created by summing band limited noise octaves each having a randomly varying amplitude. This can simply be viewed as rescaling and adding base noise functions to produce complex terrain functions. Musgrave [1993] notes that the term "octave" implies frequency doubling, and is actually only valid when the frequency is doubled for each octave. Musgrave [1993] presents a method for terrain synthesis using Perlin noise as the base function. Large part of Musgrave's research deals with procedural erosion models and finding procedural methods for capturing the features of real world terrain. Musgrave [1993] notes that there is a change in statistics when assessing features of real world mountain ranges at different altitudes. Foothills are rounded whereas the high mountain peaks are more jagged. To produce such terrains, an algorithm that scales the roughness of the noise sum based on altitude is presented.

Santamaría-Ibirika et al. [2014] introduce a method for volumetric terrain generation that utilizes Perlin noise for the terrain layer generation. It focuses on generating realistic terrain layers consisting of set of materials. Distribution of the materials is done by creating pseudo-random distribution of layers, veins and caverns. Layers are the base parts of the resulting volumetric terrain, obtained by sampling Perlin noise do determine the layer thickness and shape. Cave systems and material veins are then generated inside layers using a different algorithm. The topmost layer and it's terrain topology are not in the main focus of method. For assessing the quality of the terrain generation method the aspects presented in Table 1 are used. The aspects are generally useful for assessing procedural terrain generation methods.

Innovation	Ability to generate unique, unexpected results (within defined bounds).
Structure	Ability to generate structured results, distinguishable patterns that differ from random noise.
Interest	Ability to generate interesting, explorable terrains for the user to interact with.
Speed	Speed of the generation method. Whether the terrain can be generated on the fly in applications or it must be generated beforehand.
Usability	How well the generation method hides it's technical aspects for the designer.
Control	Amount of control the procedural terrain designer has over the result.
Scalability	Ability to produce terrains with various scales and levels of detail.
Realism	Ability to produce realistic, plausible terrains.

Table 1: Aspects for assessing procedural terrain generator quality.

Measurable metrics are given for each aspect for assessing how well the procedural method fulfills the requisite. The metrics provide a base method for assessing the quality of the procedural method and the resulting terrains. However, some of the metrics may be difficult to assess objectively. For example, measuring the interestingness of the resulting terrain is highly application specific. Noise based methods are generally strong at least on speed, scalability and innovation as noise functions are quite fast and the base frequency and sampling rate can be easily controlled. Non-periodic noise functions are strong on innovation, as they can generate infinite amount of distinct terrains, especially when combined. They may lack usability and control. As noted before, noise functions can only be controlled in global scale using quite unintuitive parameters.

4.2. Procedural noise by example

Using noise to automatically reproduce reference terrain features is an extremely difficult problem and no comprehensive solution exists [Lagae et al., 2010a]. Controlling noise function parameters and combining them is viewed as unintuitive or tedious [Smelik et al., 2010; Brosz et al., 2007]. Noise characteristics can only be controlled in global scale and the desired result is obtained by manually finding the correct combinations of noise and their parameters. Finding the correct parameters from example data have been presented to address this problem [Lagae et al., 2010a]. Another large portion of previous research has focused on using noise to enhance features of a base terrain model. Noise is used to add more detail and enhance the base terrain model rather than generating it from scratch. Commonly existing solutions require some degree of user interaction for, e.g., altering the noise parameters manually in order to produce satisfactory results [Lagae et al., 2010a]. Related work originally intended for textures is also assessed in the following, because height maps can also be viewed as a type of image textures or procedural textures that just contain scalar height values instead of color channel information.

Texture synthesis methods differ fundamentally from procedural noise based approaches. They are used for generating larger, continuous patterns from small example images – they produce image textures rather than procedural textures. [Lagae et al., 2010b]. Thus the methods lack the important properties of procedural noise based approaches, even though they can relatively simply produce continuous patterns from a small example image. Significant portion of the methods are based on Markov random fields. Markov random field methods model a texture as a local and stationary random process. Each pixel of an image is characterized by a set of its neighboring pixels. Different window sized can be used to obtain different results – smaller window sizes give increasingly randomized results. The output image should then be formulated in a way that each output pixel has the properties required by at least one input window. This base method has been enhanced in previous research [Wei et al., 2009].

Smelik et al. [2010] present a method for generating terrain for a military training game, where a rough user made sketch is used as a base for the terrain elevation model and various noise is applied on top of the sketch to generate finer details. Brosz et al. [2007] described a somewhat similar method. The method introduces a concept of base terrain and target terrain. Base terrain is an example terrain model that represents the high level characteristics of the terrain to be synthesized. Target terrain represents the finer grained characteristics that the synthesized terrain is supposed to have. The motivation is to synthesized the resulting terrain using these two examples, making the manual modeling less cumbersome. De Carpentier and Bidarra [2009] introduced the concept of procedural brushes. They are used in combination with more traditional 3D-drawing brushes to mark the areas where a type of noise should be applied on top of the base elevation model. These methods do not rely purely on noise – thus, they can only be used to create a fixed size terrain model and manual modeling is required to create the base terrain. They lack the features of pure procedural noise based approaches, like true random accessibility and ability to generate infinitely large terrains with desired characteristics.

Several researchers have addressed the issue of composing a noise as a summation of sine waves, derived from a reference image's Fourier transform. Lagae et al. [2010a] note that these spectral synthesis based approaches work best when the reference images contain strong periodicities that are visible in the power spectrum. Lagae et al. [2010b] introduce a method for generating procedural textures by extracting features from an example image's power spectrum. Wavelet noise was chosen as the base noise implementation for the method over Perlin noise because it offers more precise band-limiting and thus reproducing the original image's power spectrum is more straightforward when various bands of noise are being summed. In other words, shaping the power spectrum is easier, when noise bands are orthogonal and the desired frequency bands can accurately be controlled individually. The frequency domain representation of the example image is first analyzed. The frequencies that contribute to the image are divided into bands and the total weight of each band is calculated. Bands of wavelet noise are then parametrized using the acquired weights and summed, resulting in a procedural texture with a power spectrum similar to the example image. The resulting procedural texture always has a Gaussian histogram. This is due to the Gaussian amplitude distribution of the noise function used. For source images that don't have Gaussian intensity distribution, histogram matching can be used to produce textures more similar to the original. The method is aimed for generating procedural textures by example, but the same principle could be used in procedural 2-dimensional height map synthesis.

Galerne et al. [2012] introduced a method for constructing Gabor noise by example. The method parametrizes Gabor noise to produce a power spectrum similar to an example image. The method works on grayscale images with single color channel as well as colored textures and is completely automatic – the statistics for constructing the required Gabor kernels are extracted from the example images. Parameter estimation is done by using the periodogram of an example image as an approximation of its power spectrum and constructing Gabor noise with a one. The method relies purely on power spectrum reconstruction, it is capable of accurately creating procedural textures from Gaussian image textures – a type of textures that is completely characterized by their power spectrum. Despite this limitation, it can generate highly detailed procedural textures from a wide range of example textures and the result function has the useful characteristics of a pure procedural noise function. The resulting Gabor noise uses a summation of kernels to reproduce the reference image's power spectrum. Therefore, the resulting function's performance varies based on the example texture used. Authors report ~30FPS performance for an evaluation of the noise for 128x128 texture using NVIDIA Quadro 5000 GPU. In any case, the resulting noise function is significantly slower than simpler noise functions with fixed power spectrum, e.g., Perlin or Simplex noise.

5. Procedural terrain by example

Gabor noise is capable of producing an arbitrary power spectrum and is very well suited for automatically producing procedural textures that accurately represents the example. However, the resulting noise takes significantly longer to evaluate compared to, say, lattice noise functions presented earlier. The parameter estimation phase of the method presented by Galarne et al. [2012] takes approximately two minutes for a 128x128 reference grayscale image. For an interactive procedural terrain design by example, this is not convenient. There's a clear tradeoff between accurately reproducing the example data features in procedural noise and the evaluation speed of the resulting function. For designing useful procedural terrains to be used in applications, performance is an important factor – resulting function is not useful, unless it can generate terrain fast enough. Required performance varies greatly between applications, most demanding perhaps being games that need to generate new terrain on the fly as the player advances in the game world.

Lattice gradient noises have been extensively used for terrain generation and they are fast an robust. They lack accurate spectral control, so in order to produce patterns by example, spectral synthesis based methods are not an option. In this chapter, methods for creating procedural terrain by example using Simplex noise are presented. Simplex noise was selected because of its good quality and best performance among studied functions. However, the presented methods don't rely on the Simplex noise implementation – they work as well for any procedural base noise function that returns values with approximately known value distribution. Various noise implementations having approximately same properties and execution speed as Simplex noise exist and could be used instead. First methods for forming more complex patterns using base noise functions are presented. The last part of the thesis focuses on reproducing selected statistics of example data in procedural noise and automatically layering and masking noise to generate complex terrain topology. Discussed methods were implemented in a software prototype with a graphical user interface for procedural terrain design.

5.1. Fractal sum and turbulence

An octave of noise refers to scaled version of a base noise function. The whole frequency band making up the noise can be scaled to obtain patterns with faster or slower transitions between perceivable peak values. In case of lattice noise, this can be seen as scaling the integer lattice up or down – the distance between lattice points changes. Interchangeably this scaling can be viewed as scaling of the input coordinates.

Noise functions can be used to generate more complex patterns by using fractal sums [Ebert et al., 2002]. This is somewhat similar concept to Fourier synthesis, where a function is built up from sinusoidal components. But in case of fractal noise, each noise octave consist a number of frequencies that are added or blended together. This kind of method can be viewed as "poor man's Fourier series" [Musgrave, 1993]. It lacks precision (depending on the noise function used) but is very fast in comparison.

```
float fractalsum(point Q) {
  float value = 0;
  for (f = MINFREQ; f < MAXFREQ; f *= 2) {
    value += noise(Q * f)/f;
  }
  return value;
}
```

Code fragment 3: Fractal sum.

In Code fragment 3, fractal sum in a given point is calculated by first defining the minimum and maximum frequency at which the noise function is to be evaluated. The input point is scaled to reach the desired frequency band and the noise value at each octave is evaluated and added to the sum. Power spectrum of the result function is derived from the noise octaves used, but the contributions of the higher frequencies are diminished by the 1/f amplitude scaling.

```
float turbulence(point Q) {
  float value = 0;
  for (f = MINFREQ; f < MAXFREQ; f *= 2) {
    value += abs(noise(Q * f))/f;
  }
  return value;
}
```

Code fragment 4: Turbulence.

Turbulence implementation in Code fragment 4 is otherwise similar, but absolute value of each noise band evaluation (in range [-1,1]) is used instead of the raw value. As a result, the number of peaks – lighter values in the usual pattern image – are doubled and the pattern has visibly sharper edges. Summing noise bands is by no means limited to this kind of simple summing with 1/f amplitude damping – even though it's useful for keeping the result values within original noise function range. The contribution of each band can be individually controlled, e.g., by using a low frequency noise octave as a base with high amplitude contribution and using a higher frequency band with lower amplitude contribution to add minor details to the base noise.

5.2. Noise redistribution

Besides frequency scaling, a single noise layer can be controlled by applying a function to the noise function result. Raising noise result (which is transformed to range [0,1]) to power has the result of diminishing low and middle values. This simple method could be used to transform the smooth base noise into height map representing, e.g., high mountain peaks with low valleys in between. Negative exponents have the opposite effect, bringing the lower values up. Applying a power function, scaling the noise amplitude and shifting the output range are simple but very limited methods. More complex redistribution functions are needed to produce complex procedural terrains and match statistics of example data. Noise functions typically have Gaussian-like value distribution. The distribution can be altered by applying a piecewise defined redistribution function to the noise function result. In this way, arbitrary distributions can be obtained.

The redistribution function can be arbitrarily designed, but for matching a reference height map distribution it should include a well defined control over the result. One approach is to define the function using a series of data points that map the corresponding noise function values to different ones. For a relatively fine grained control, the redistribution function can be defined using *N* sample points (x,y), values of x spanning the amplitude of the original noise function. Ebert et al. [2002] presented a color spline function, which is a similar concept. The function involves storing, e.g., 11 equally spaced samples and interpolating between them to obtain a color value given a noise function value. In other words, color spline is an arbitrary mapping of an input noise value to an output color. Noise redistribution function presented here works similarly, but with scalar values as an output. The sequence of control points must span the whole noise amplitude range from A_{min} to A_{max}:

$$\langle (x_i, y_i) \rangle \mid x_0 = A_{min},\ x_n = A_{max},\ x_i < x_{i+1}\ .$$

The function is constructed using a finite number of sample points so some sort of interpolation in between them is required. Linear interpolation is the simplest and fastest form of interpolation. It simply joins the sample points with straight line segments and this results in a function with discontinuations at the data points. The result is not very usable as smooth transitions between sample points are required. Cosine interpolation is another interpolation method that uses cosine function to produce a smooth curve between the points. Cubic interpolation method samples four points and makes the result much smoother. It is slightly more computationally expensive than cosine interpolation. Performance is an important aspect as the function needs to be evaluated for each point of the height map to be created. Generally higher quality interpolation methods are also more computationally expensive.

5.3. Reference height maps

In order to use methods for calculating reference statistics and reproducing them in procedural noise, the reference height maps must first be loaded in the prototype. Input height maps are normal RGBA images with 8-bit precision per channel. This enables designing reference height maps and layer masking in any external image editor. Only one channel is used to determine height so external image editor should use e.g. color values where R=G=B, resulting greyscale images. R channel value 0 is interpreted as the lowest possible value, effectively offsetting height down the maximal amount. Middle value 0.5 maps to no offset in elevation and 1.0 the maximum elevation. Alpha channel values are interpreted as distance from center value, 0.5. For example, an RGBA color value, represented as four floating point values, (0.5, 0, 0, 1) is interpreted identically as value (1, 1, 1, 0.5). This makes layer design in external image editors more straightforward. Layers can be designed using the alpha channel and overlaying layers for preview using default alpha blending mode.

5.4. Extracting and reproducing terrain features

This section explains the methods and algorithms used in automatically parametrizing noise by example. The terrain design process consists of designing example terrain layers as images in an external editor and loading them in the procedural terrain design tool. Each reference layer is used to parametrize a single noise function redistribution and base frequency. In the final step the noise layers are summed to form the final procedural terrain. Graphical user interface of the prototype and the example based terrain design flow is explained in Section 5.6.

5.4.1. Generating noise layer

The method utilizes a redistribution function to approximately match the value distributions of procedural noise and the reference image for each layer. Control points or spline knots in the redistribution function need to be defined. The easiest approach would be to define the control points to evenly spaced in the expected noise function and reference image value range, [0,1] for example. However, this approach would make the control points affect uneven value ranges. In case of most noise functions the function's value distribution closely resembles a Gaussian distribution. As a result, for a noise function with value range [0,1], a control point at x=0.1 would actually affect a much smaller percentage of noise function values compared to e.g. x=0.5; the mean value of the noise function. In order to make each control point contribute the equal amount, the redistribution function needs to be equalized based on expected probability distribution of the data. Ebert et al. [2002] present a simple method for equalization. It's based on sampling the noise function for a limited, large number of points, and is thus an approximation. The algorithm starts by computing N subsequent noise function values. N should be sufficiently large to capture the noise's value distribution accurately. Then the values are sorted in an ascending array. The result is used as lookup table, from which the sample points are extracted, using K indexes that are evenly spaced in range [0,N].

An example of redistribution function with 16 control points calculated from a reference data in Figure 20 is shown in Table 2 and the corresponding redistribution function plot in Figure 13. *X* refers to the control point location – the noise function value to be mapped and *Y* the value to which it will be mapped to, interpolating between points. As seen in Figure 13, the control points are more dense near the noise function mean value, 0.5, than elsewhere in the shown range.

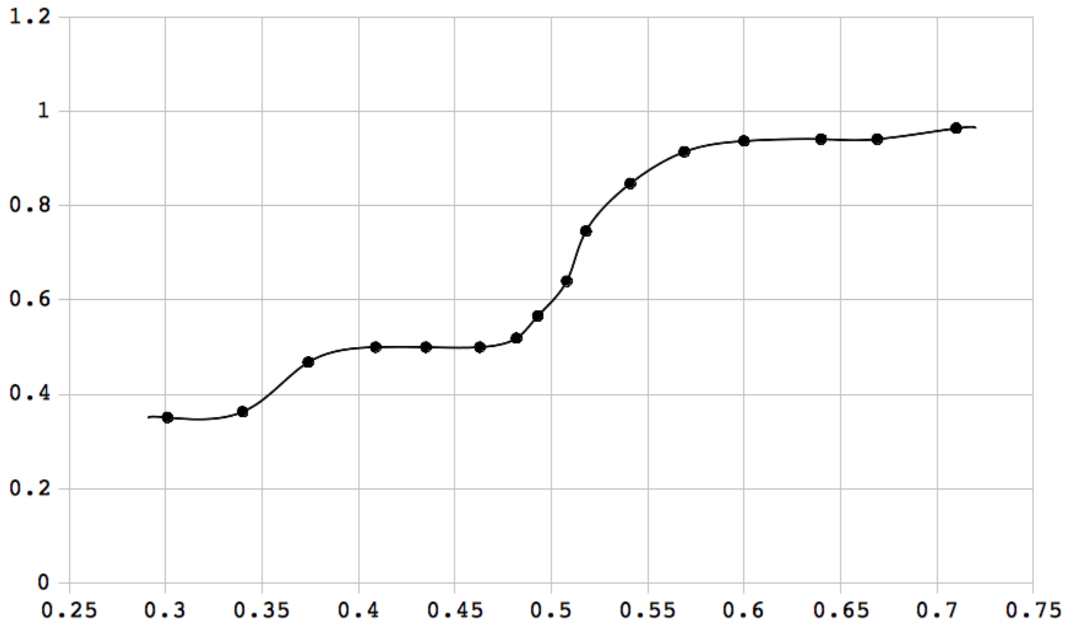


Figure 13: Redistribution function plot.

X	0.30	0.34	0.37	0.41	0.44	0.46	0.48	0.49	0.51	0.52	0.54	0.57	0.60	0.64	0.67	0.71
Y	0.35	0.36	0.47	0.50	0.50	0.50	0.52	0.57	0.64	0.75	0.85	0.91	0.93	0.94	0.94	0.96

Table 2: Redistribution function point values.

Once the equalized redistribution function control point X-values are calculated for both reference data and noise, the reference redistribution function Y-values are simply used in the noise's redistribution function. This effectively produces approximately similar probability distributions, making the noise and reference image to have similar histograms. The pattern of the original noise function is still more or less visible depending on the redistribution parameters used because the underlying noise function is still the same – redistribution function only maps its raw output values to different ones. This somewhat restricts the synthesis method depending on the noise function implementation used, but it can still produce a variety of good quality terrains by example as demonstrated later. Figures 14 to 19 display a reference height map and a window from the redistributed noise function obtained by using this technique with different number of control points. Chi-Square distance of the reference and result noise histograms were used to measure similarity. Chi-Square distance is defined as

$$\frac{1}{2}\sum_{i=1}^{n}\frac{(x_i-y_i)^2}{(x_i+y_i)}$$

where n is the number of histograms bins and x_i and y_i the values of the compared histogram bins. [Yang et al., 2015] Histogram distances (n=32) for each number of control points are given in Table 3. The first row of the table refers to the histogram distance of the reference image compared to unprocessed Simplex noise.

Number of control points	Chi-Square distance
-	0.43
4	0.32
8	0.20
16	0.05
32	0.04
64	0.03

Table 3: Histogram distances for different number of control points.

Figure 14: Unprocessed noise.

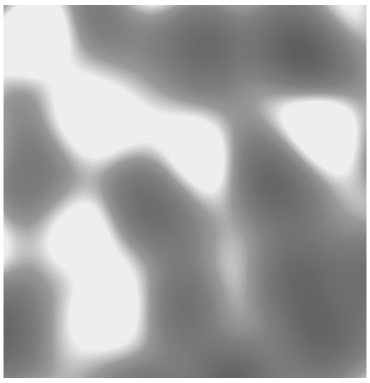


Figure 15: 4 control points.

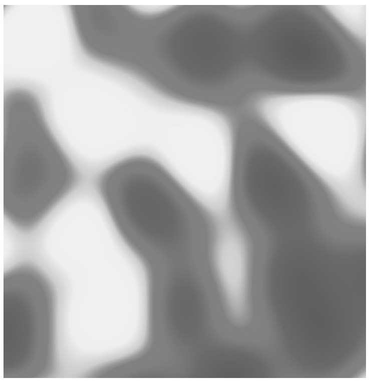


Figure 16: 8 control points.

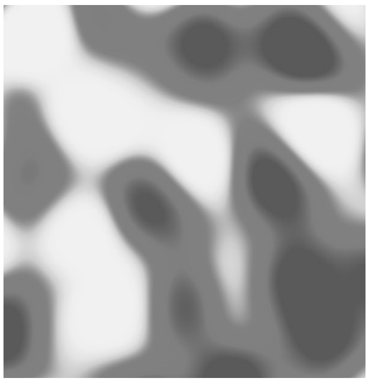


Figure 17: 16 control points.

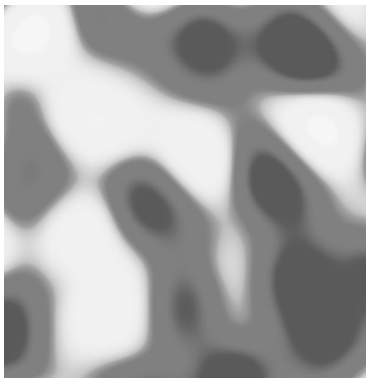


Figure 18: 32 control points.

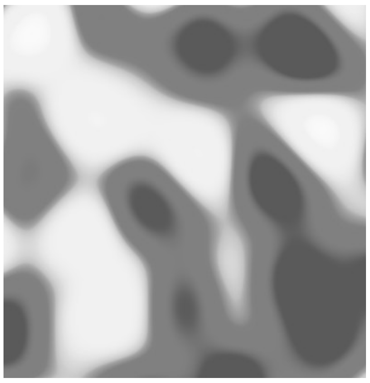


Figure 19: 64 control points.

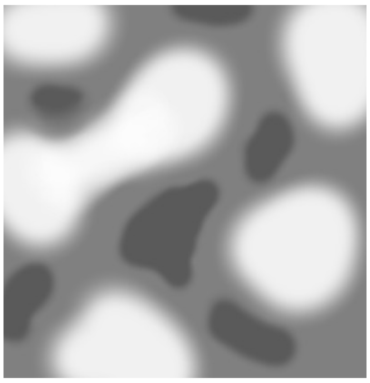


Figure 20: Reference height map.

Not surprisingly and as seen in Table 3, the histogram similarity increases with the number of control points used. This effect can also be perceived in resulting height maps in Figures 14-19. For a higher number of control points the resulting noise function captures the reference data features more accurately. Higher number of control points produces some computational overhead from finding the start and end control point values between which to interpolate, but this overhead is quite minimal in real world usage if the search is implemented efficiently. Binary search was used in the prototype for finding the correct redistribution point with the standard *O(log n)* performance.

Redistributed noise function manages to visually match the reference image closely only in limited cases. A single noise layer represents a single base frequency band that cannot be altered by the redistribution function. For example, a reference image with very heterogeneous sized features can not be reproduced using a single processed noise layer. Also, the value range of the reference image is mapped in a linear manner so that lowest values of the reference image are represented using the lowest values of the original noise function. As a result, the automatic mapping can not produce all results obtainable by manually adjusting the redistribution function.

Base noise frequency is estimated using observed average peak values per row. Peak values from reference data are calculated using a stateful search algorithm. The reference height map array is traveled horizontally and vertically, counting the points where the value trend changes from increasing to decreasing – these are considered the peak values. For masked reference layers, only visible values are considered. Finally a single scalar, row average, is calculated. Then the noise frequency is iteratively adjusted using a heuristic search until it matches the reference data. Adjustment is done using the fact that number of peak values in noise increases with frequency. The implementation is quite naive approximation and very sensitive to unwanted noise in the reference data. It works sufficiently for height maps that only contain the intended patterns representable by a single noise layer and no unwanted (high frequency) noise.

5.4.2. Automatic layer masking

Redistribution function works quite well for producing noise patterns by example but it fails to reproduce certain structured aspects of the reference data. For a reference height map representing terrain with very flat plains and jagged mountains with more detail in comparison, a single layer of redistributed noise is not enough. Single noise layer has only a single base frequency that can not be changed for different parts of the terrain. In order to add detail to said mountains, another noise layer with higher frequency is needed. The noise also needs to be masked so that it only alters height in areas determined by the underlying noise.

The final procedural terrain function consists of *N* layers of processed noise. It's somewhat similar to fractal noise presented earlier, only with more control over each "octave". To determine layer masking, reference height map layers are evaluated in order from top to bottom, and for each layer the underlying reference layers are checked against the current layer. Underlying layer is considered to mask a layer on top of it if the top layer does not span the whole value range of the underlying layer. Masked range is assumed to be continuous, determined by the minimum and maximum value where top reference layer overlaps the layer under it. Figures 21 and 22 display a reference image on the left and the corresponding automatically determined noise layer on the right. Figure 21 displays a bottom layer that is used as a mask for the layer in Figure 22. Figure 23 shows the combined reference images and noise layers, normalized to range [0,1].

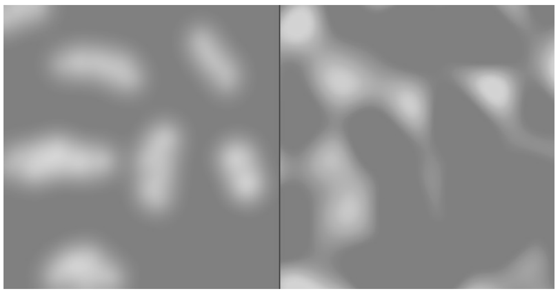


Figure 21: Bottom layer.

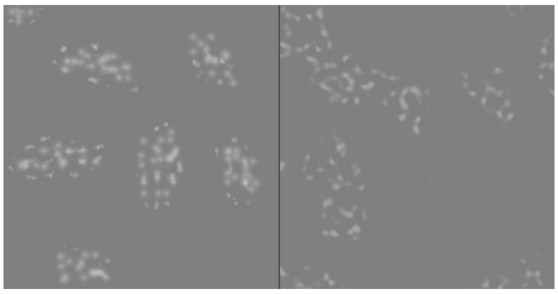


Figure 22: Masked layer.

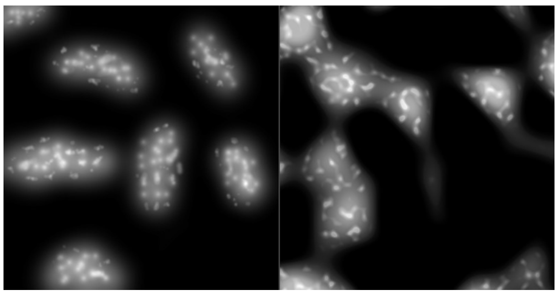


Figure 23: Combined layers.

When the mask range has been determined, it's saved in the processed noise function of that layer to be used in final noise evaluation. This effectively limits the output values of the noise function based on the mask range calculated earlier. Outside this range, the elevation change from the layer's processed noise function evaluates to zero. Smoothstepping was used instead of simple clamping near the mask range edges to make the seams less visible where the mask is applied. It could be beneficial to implement the mask as partial mask instead of binary – the mask would alter the masked layer values using value distributions learned from the example data. Determining such mask from the example data would not be a trivial and probably computationally expensive.

5.4.3. Procedural terrain storage format

For the designed procedural terrain to be useful in applications, it needs to be exported and saved after design process is complete. In this subsection, a JSON-representation of the procedural terrain is presented. JSON representation contains the minimal set of parameters to reproduce the designed terrain in applications. JSON schema is given in Code fragment 5 and descriptions of the schema objects in Table 4.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "noiseLayers": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "baseNoise": {
            "type": "object",
            "properties": {
              "frequency": {"type": "number"}
            }
          },
          "rd": {
            "type": "object",
            "properties": {
              "points": {
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "x": {type": "number"},
                    "y": {"type": "number"}
                  }
                }
              }
            }
          },
          "preAmpShift": { "type": "number"},
          "ampScale": {"type": "number"},
          "maskLayerIndex": {"type": "integer"},
          "maskMinValue": {"type": "number"},
          "maskMaxValue": {"type": "number"}
        }
      }
    },
    "maxValue": {"type": "number"},
    "minValue": {"type": "number"}
  }
}
```

Code fragment 5: JSON schema for procedural terrain.

Root object property	**Description**
noiseLayers	Array of processed noise layers to be summed when evaluating final procedural terrain function.
maxValue	Expected maximum value of the final terrain function.
minValue	Expected minimum value of the final terrain function.

Noise layer property	**Description**
baseNoise	Frequency scaling of the base noise used for the layer. Scaling should be applied directly to the base noise, e.g. SimplexNoise, before any processing.
rd	Redistribution function properties.
preAmpShift	Used together with *ampScale* to translate and scale the output value range of the noise function. For example, $X_{i,j}=(noise_{i,j}+preAmpShift)*ampScale$
ampScale	Value the translated noise value should be multiplied by.
maskLayerIndex	Index of another noise layer, value of which will be used for masking.
maskMinValue	Minimum value of the masked range, current layer's values outside the mask layer range will be discarded.
maskMaxValue	Maximum value of the masked range.

Redistribution function property	**Description**
points	Equalized points used to construct the redistribution function.

Table 4: Description of JSON properties.

In order to obtain correct results in applications rendering the procedural terrain, the terrain must be designed using the same base noise implementation as is used in final application. For the prototype, Perlin's reference implementation for Simplex noise was used but any other randomly accessible procedural noise function could be used as a plug-in replacement. For evaluating the noise in applications, the described redistribution function and noise layer summation must also be implemented.

5.5. From height map to terrain model

The aim of the synthesis method and the prototype is to design procedural terrains to be used in 3D graphics applications. This section discusses methods for turning the calculated procedural terrain representation into a rendered 3D terrain mesh. 3D rendering can be used to aid the procedural terrain design process and the methods are also generally useful in applications that make use of the generated procedural terrains. For a 3D rendering, terrain mesh with attributes including at least vertex coordinates, normals and optionally texture coordinates need to be calculated from the procedural terrain function result. The design process is built around height map representations of the terrain. Even if the method closely reproduces the example height map, it's useful to see the procedural terrain in different parts of the design process, especially when designing masked layers. For the prototype, a 3D preview of the designed procedural terrain is presented alongside the synthesis UI for assessing preliminary results of the terrain design process.

Random access feature makes it possible to calculate the terrain in chunks for efficient real time rendering. Visible terrain at a given time consists of a number of chunks and the chunks will be calculated and disposed based on which area is currently rendered. Each chunk produces a terrain mesh that seamlessly fits to the neighboring ones. To calculate the terrain mesh for a $N \times N$ sized chunk, first an (two dimensional) array of size $(N+2) \times (N+2)$ is calculated by sampling the noise function. The border values are only needed for calculating the partial terrain mesh normals correctly using height values of adjacent points. Actual vertex positions are calculated using the inner values. Calculating the chunk height map is not strictly necessary, as the noise function could be sampled directly in the shader for each vertex, but calculating the map beforehand in the central processing unit frees resources of the graphics processing unit for other things. This tradeoff should be addressed depending on the application. Height values calculated earlier are in the range determined by the number of layers and redistribution function used. Final height values are calculated by multiplying calculated heights by a global constant, scaling the whole terrain height range.

Usually in rendering at least vertex coordinates and vertex normals are needed for the rendered terrain mesh when using common lighting models. Vertex coordinates (*x, y, z*) for the terrain mesh are simply derived from the terrain array coordinates *i* and *j* for *x* and *y*, and height value stored at *(i, j)* for *z*. Since the terrain height array represents evenly spaced samples of the terrain function, the normal vector for a point (i, j) can be calculated by using the heights from neighboring points to form the normal vector and then normalizing it:

$$N_{(i,j)} = \left\| \left(h_{(i-1,\, j)} - h_{(i+1,\, j)}\,,\, h_{(i,\, j-1)} - h_{(i,\, j+1)},\, 2 \right) \right\|.$$

Creating a terrain mesh from height map effectively works by deforming a flat plane by using noise function values. The method is quite fast an efficient. The downside is that it works effectively by offsetting the z-coordinates of vertices of a flat plane. Because of this, textures appear stretched in areas where altitudes of subsequent vertices have large difference. To prevent this, triplanar texturing was used in the prototype's 3D preview. Triplanar texturing refers to a type texturing where the final color value of each fragment is produced by sampling texture along three axis aligned planes and blending these results based on the previously calculated vertex normal. Textures still appear blurred in parts where no clearly dominant blending weight exists, i.e., at points where normals are pointing in between the axis aligned planes, but it makes easily noticeable improvement over traditional texturing in terrains with steep curves.

Using a single texture for the whole terrain is perhaps not very common in applications. For terrains it often makes sense to vary the texture based on height. This is a simple method for, e.g., texturing lower areas with grass and mountains with a rocky texture using a single mesh and in a single render pass. Vertex normal can also be used in similar kind of texturing, e.g, by applying a cliff texture in steep mountain slopes, where the dot product of normal vector and up vector is lower than a certain value (say, 0.5). Figure 24 illustrates the basic texture mapping strategy for three textures.

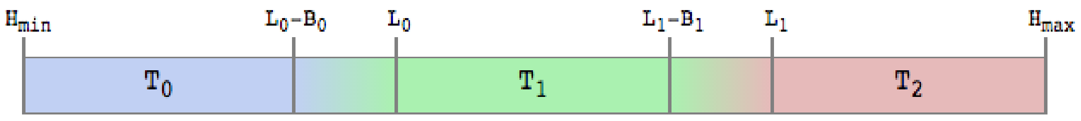


Figure 24: Height based texture mapping.

Textures T_i are mapped in the expected height range H_{min} to H_{max}, using a height limit L_n and a blend out length B_n for each texture. Within the blending range, two textures are blended to achieve smooth transition. In order to make texturing more interesting and natural looking, the vertex height used to determine the applied texture can be offset by a value from another, relatively high frequency noise function. This way the texture “seams” are randomly offset by amount determined pseudo-randomly, making the height-based texturing less obvious. Figures 25 and 26 display a final terrain preview constructed with three layers of redistributed noise. Texturing uses four textures that are sampled based on the fragment height. Figure 26 includes the texture height offset from noise. It makes the transition points less visible and as result the terrain looks less like it's made of layers stacked on top of each other.

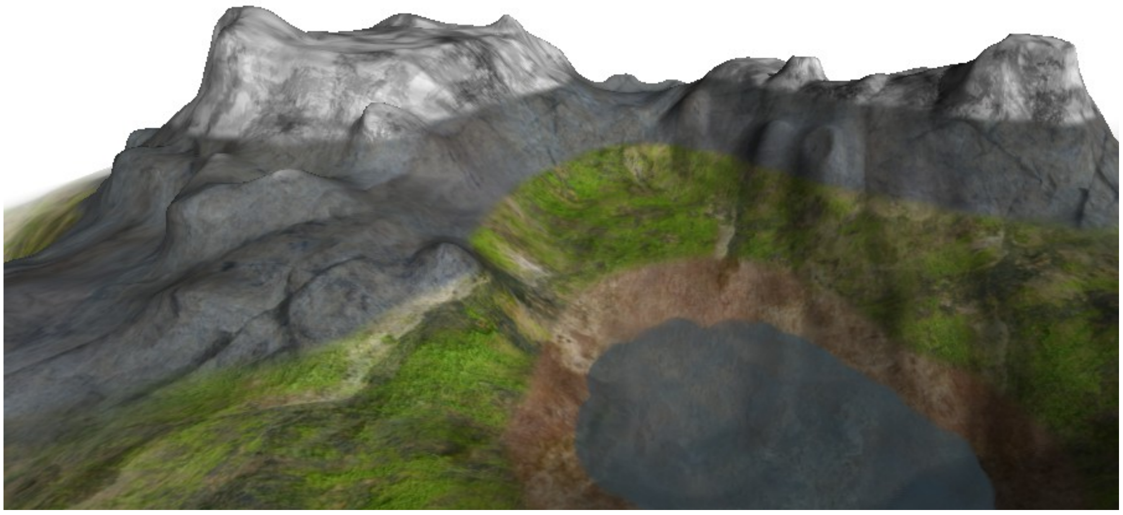


Figure 25: Textured terrain without texture height offsetting.

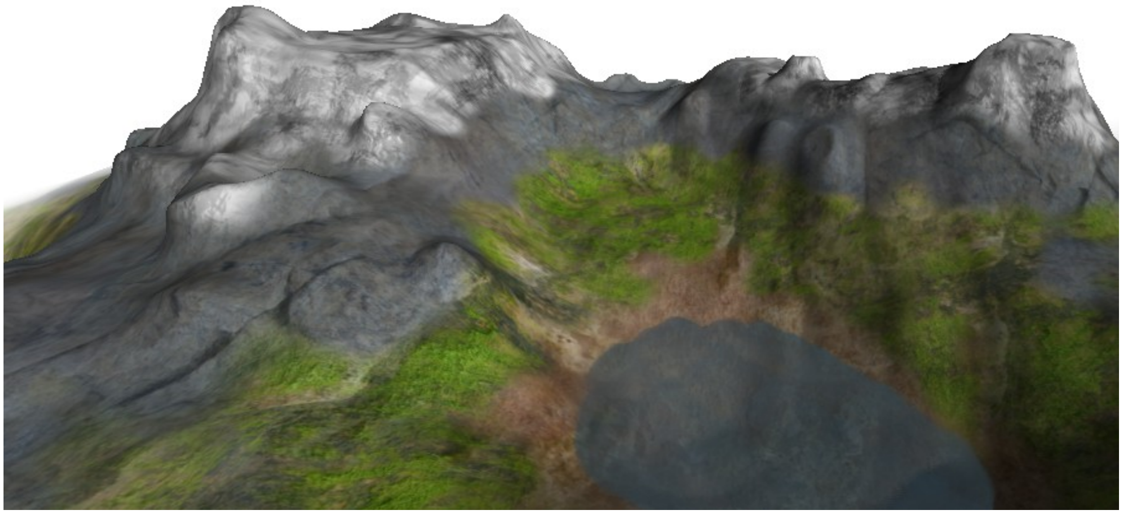
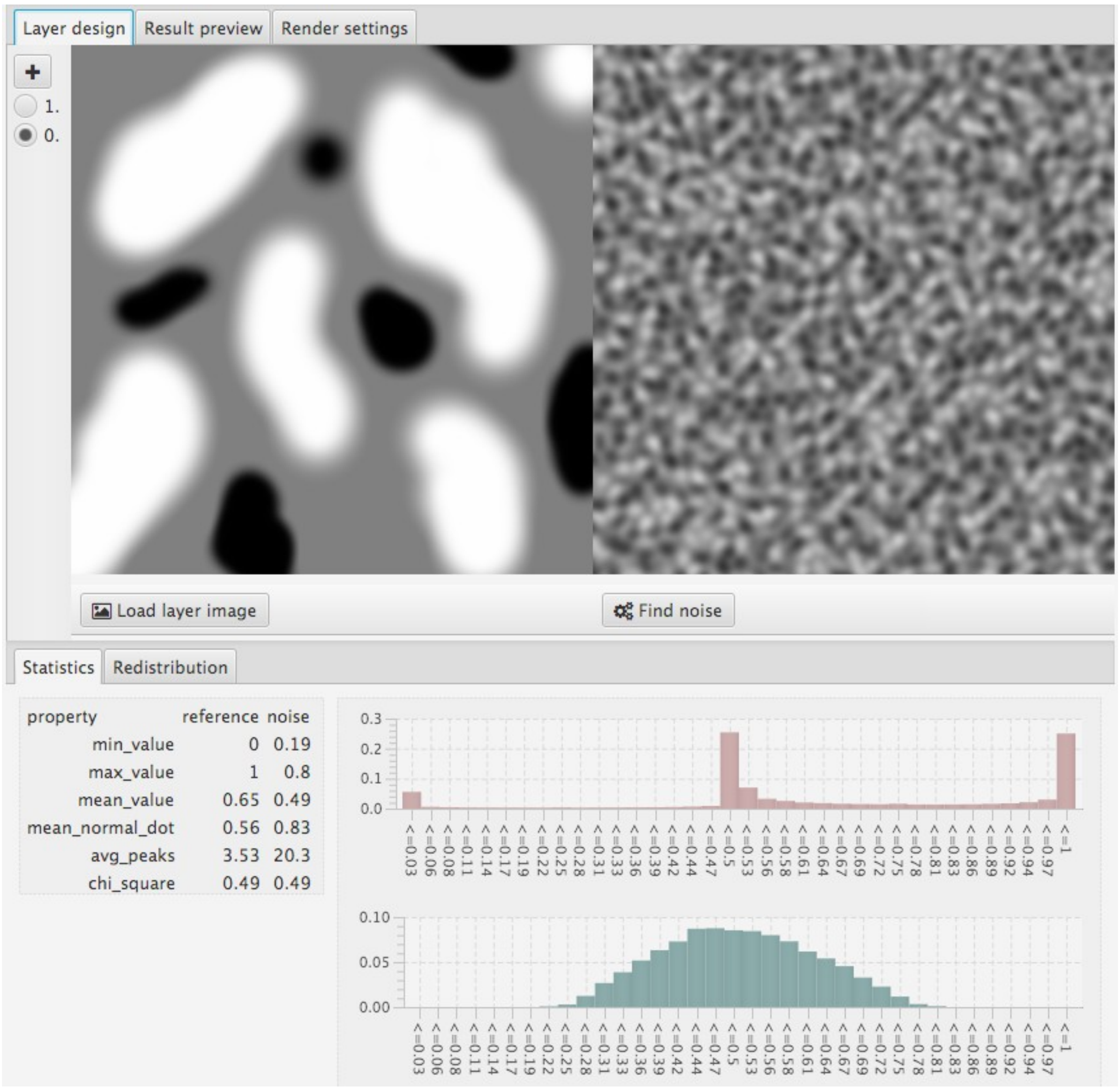


Figure 26: Textured terrain with texture height offset from noise.

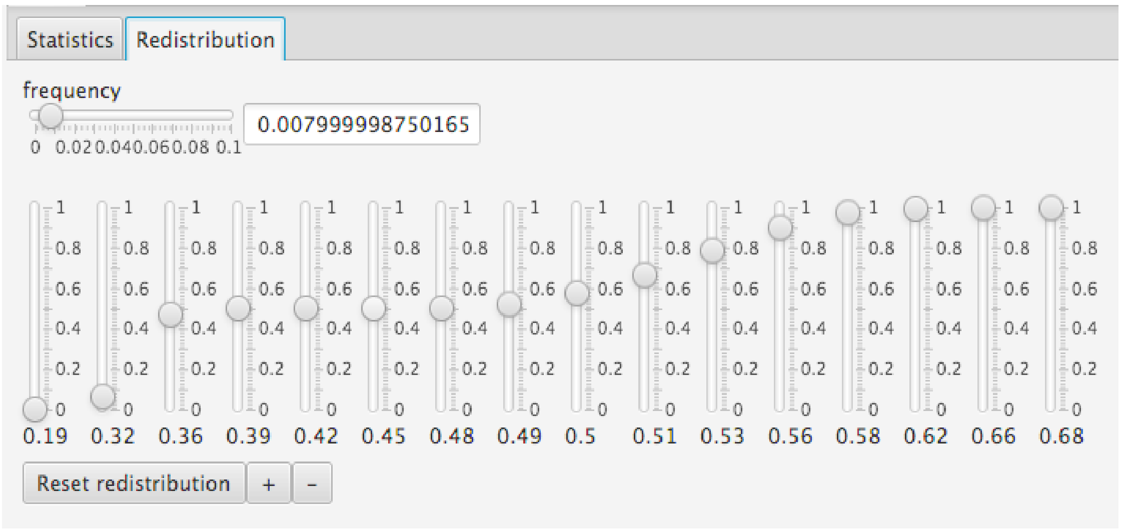
5.6. User interface for procedural terrain design

Graphical user interface was implemented for the prototype in order to further simplify the procedural terrain design process. It encapsulates the previously introduced noise-based terrain synthesis methods in a simple user interface. The user interface consists of three main views; Layer design, Result preview and Render settings.



property	reference	noise
min_value	0	0.19
max_value	1	0.8
mean_value	0.65	0.49
mean_normal_dot	0.56	0.83
avg_peaks	3.53	20.3
chi_square	0.49	0.49

Figure 27: Layer design view.



frequency 0.007999998750165

Figure 28: Manual redistribution function adjustment.

Layer design view, in Figure 27, is used for loading the reference height maps and previewing the parametrized noise layer for each reference height map. For each layer, a reference height map is first loaded into the editor. Then the Find noise feature can be used to automatically calculate the redistribution function and the noise frequency to be used using the previously discussed methods. Parametrized noise is then viewed beside the original reference height map with value histograms and general statistics for for comparison. The result can be fine tuned using Redistribution tab of the Layer design-view. Redistribution tab (see Figure 28) contains a slider representation of the redistribution function used and a slider for the noise frequency. The base noise frequency and Y-values of the redistribution function can be manually adjusted while viewing the result live in the displayed noise window.

The design process is intended to begin with the bottom layer with label 0 and end with the topmost layer *N*. This amounts to designing the terrain layer by layer starting from the bottom and the masking parameters are automatically determined. Typically, the bottom layers contain base noise with lower frequency but this depends on the desired masking. Layers can be added and removed in the design tool. The automatically calculated masking is also displayed in the noise preview of the layer for which the map is applied. It's important to notice that the displayed noise mask is not derived directly from the reference height map but from the parametrized noise which is based on the height map.

Result preview view simply displays the combined reference layer height maps and noise layers side by side similarly as in Figure 23. The view can be used for assessing the quality of the final synthesis result in any step of the design process. The view is also used for launching the 3D preview of the resulting terrain function. The 3D preview is displayed in a different window and is updated live when the result preview view is updated. Finally the view has a feature for saving the final terrain function in previously explained JSON-format.

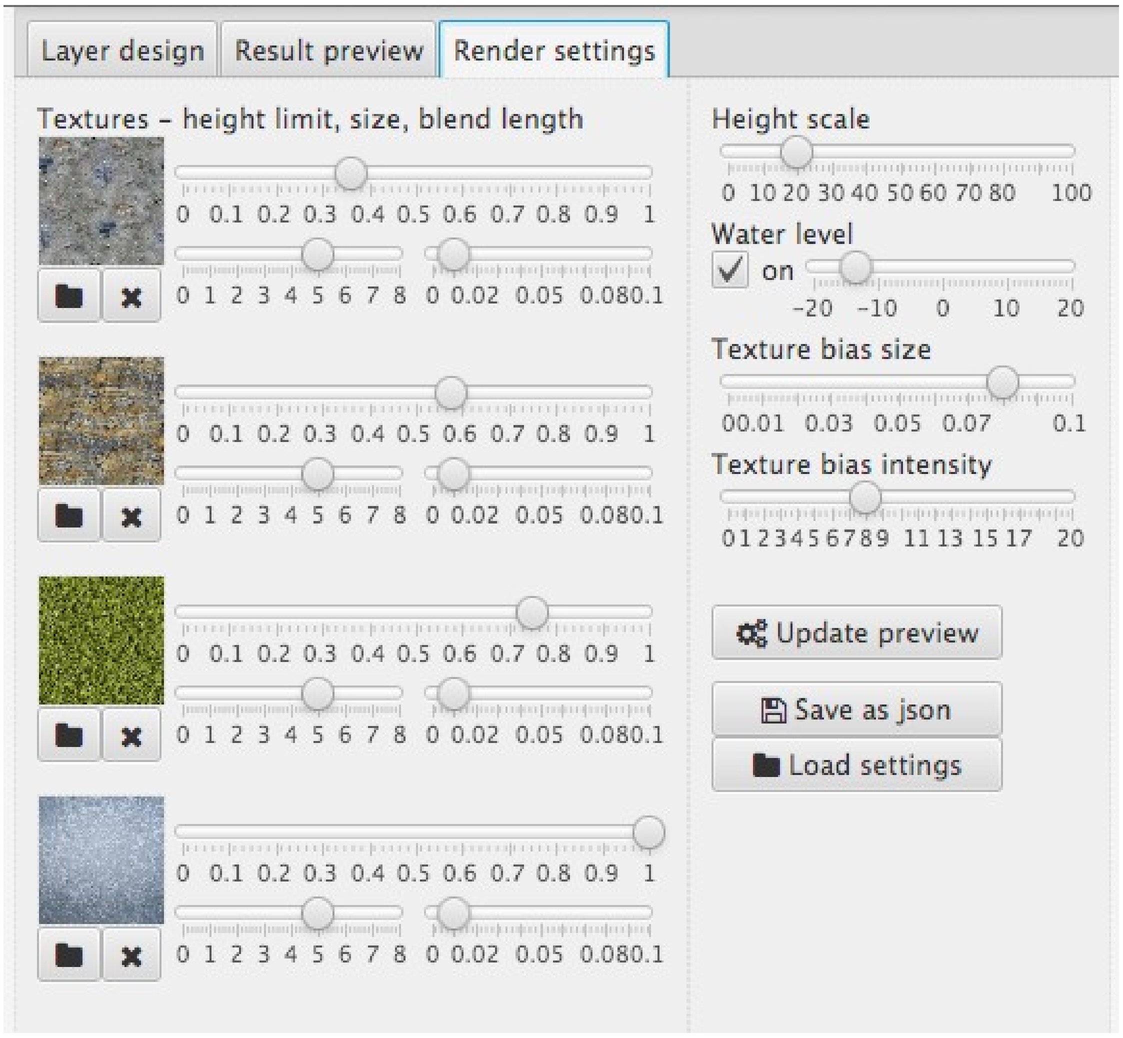


Figure 29: Terrain preview rendering settings

Render settings view (Figure 29) is only used for the 3D preview. It contains texture selection tool and controls for the textures to be used in the height-based terrain texturing. The controls include texture height limits, texture scaling used in the global texture mapping, texture blending lengths and the texture height offset noise parameters. Texturing settings can also be saved and loaded in a separate JSON-file. The settings only affect the preview rendering and are intended for getting a better understanding of the final look of the exported terrain and it's suitability for a specific application. The texture settings are not included in the actual exported procedural terrain function.

5.7. Result assessment

The main contribution of this thesis comes from application of the color spline function presented by Ebert et al. [2002] in a type of fractal noise to form complex terrains. A graphical user interface for a novel design process is also presented. The presented synthesis methods accomplish the goals set for the ideal procedural terrain function;

random accessibility, non-periodicity and low storage. The implementation does not sacrifice the useful point evaluation property of the underlying Simplex noise or affect its periodicity. The resulting terrain function is a fractal noise function by nature with a controllable redistribution function applied to each noise octave. Extra storage requirement comes almost solely from the redistribution point configuration. Thus, the procedural terrain remains extremely compact, comparable to simple fractal noise.

The synthesis method works by reproducing the value distribution of the reference image in a noise layer and approximately matching its base frequency. The synthesis is not perfect and as a result the user requires some understanding of the underlying noise function and how the reference images are interpreted. Reference images containing high frequency noise will distort the result of the naive frequency determination. However, the frequency can be easily adjusted manually to acquire desired results. In any case, determining the base noise frequency leaves most room for improvement and further research. The redistribution function matches the value distribution of the reference image accurately. Still, this does not always mean that the patterns formed with the parametrized noise visually resemble the reference image. For example, heterogenous sized patterns in reference images can not be appropriately reproduced using a single noise layer. Automatic layer masking aids in designing more complex terrains, but using them efficiently requires further understanding of the underlying methods.

The prototype does not include a drawing tool of any kind – the reference images are intended to be designed in an external editor. Designing the reference terrain height map directly in the graphical user interface would eliminate the extra step of loading the reference maps, streamlining the design process. Perhaps the reference terrain height map view could also be substituted with a 3D-editor, in order to get a more WYSIWYG-type of design experience. This would ease determining the actual contribution of each layer, as it may be challenging using solely the height map view of each layer and the combined view – the actual contribution is more clearly visualized in the 3D preview.

The quality of the implemented procedural terrain design tool will be discussed using the previously introduced aspects:

- Innovation
- Structure
- Interest
- Speed
- Usability
- Control
- Scalability
- Realism.

Structure and interest aspects are well covered in the prototype. The base noise produces structured patterns which can be completely remapped to match the reference image. Base noise implementation provides the original signal which is then altered to fit the example data distribution. The resulting procedural terrain provides infinite amount of variation, constrained only by the base noise implementation.

Final terrain function evaluation speed is an important aspect for the procedural terrain to be usable. For a 60FPS rendering there is theoretically 1000/60 = 16.666 milliseconds to draw each frame and execute required logic. This time must include terrain function evaluation at least in single threaded environments. The performance of the procedural terrain function was evaluated with various layer and redistribution point configurations and number of calculated values. The results are displayed in Table 5. The benchmark was run multiple times for each test case and results are averages from 128 runs. The result rows include comparison to raw Simplex noise fractal sum evaluation. Fractal sum was implemented similarly as in Code fragment 3 and using Simplex noise. This comparison was included to provide reference value from a well known, similar method. For each test case the number of fractal noise octaves is equal to the number of terrain function's noise layers.

Execution time using three different redistribution point configurations (16, 32 and 64 points) were tested. Bolded rows of the result table contain performance records for a 50x50 value chunk evaluation, which is the largest amount of terrain values the prototype preview calculates in a single render pass. As noted earlier, the terrain calculation is divided to fixed size chunks which evens out the terrain calculation load.

Number of values calculated	Number of octaves / layers	Fractal sum execution time (ms)	Terrain function 16P execution time (ms)	Terrain function 32P execution time (ms)	Terrain function 64P execution time (ms)
2500	**2**	**1**	**3**	**3**	**3**
10000	2	5	10	10	10
20000	2	8	12	13	15
30000	2	13	20	20	23
40000	2	16	26	28	31
50000	2	20	33	35	38
2500	**4**	**1**	**3**	**3**	**3**
10000	4	7	13	13	15
20000	4	14	26	27	33
30000	4	21	42	43	47
40000	4	29	55	55	65
50000	4	35	67	75	80
2500	**6**	**2**	**5**	**5**	**6**
10000	6	10	21	21	23
20000	6	19	38	41	55
30000	6	29	60	63	74
40000	6	40	77	83	100
50000	6	49	99	102	121

Table 5: Final terrain function performance test results.

Terrain function execution time is not significantly affected by the number of control points used due the implemented binary search. The values for the use case in the prototype application's terrain preview are well below the time limit for a 60FPS rendering which was also perceived in the prototype. No parallelism or concurrency was implemented in the prototype, but the terrain function offers the same parallelism support as raw Simplex noise. As expected, the execution time scales linearly with the input value count for both the implemented procedural terrain function and simple fractal sum.

Usability refers to how well the terrain generator hides its underlying technical aspects while staying usable. Actual usability or user experience of the prototype user interface was no assessed here. The general use case for terrain generation does not require knowledge about any of the noise parameters as the synthesis method is completely example based. However, some understanding of the underlying methods is still useful in order to get best results. It's important to note that each reference layer will be used to parametrize a single layer of procedural noise and this introduces limitations on what kind of patterns should be included in the reference terrain layer – a layer of noise can only represent a single base frequency. Also, noise – here referring to an unwanted high frequency signal – in the reference image causes higher than expected results in the automatic frequency calculation. In other words, the method will not fail to reproduce the selected statistics of the example image in any given case but the measured statistics are actually valid only in limited cases.

Example based approach gives the highest amount of control over the result in theory – the idea is to accurately reproduce the example in procedural noise. The result can also be iteratively corrected by altering the example layers that together form the final procedural terrain. Finally manual control of the noise parameters is also provided. The method is somewhat limited by the original patterns provided by the base noise implementation because all the executed actions are just noise post-processing by nature. The redistribution changes noise value distributions globally but does not alter the original pattern production.

Presented methods offer all the scalability benefits of procedural noise and the obtainable level of detail is not constrained. The prototype has a limited reference image size of 400x400 values. This size was chosen somewhat arbitrarily for the prototype and it's not at all enforced by the methods used. Explained techniques can work just as well with higher resolutions, even though the automatic parameter search time requirement will scale up with the resolution. The limited window size makes designing terrains with very high base noise frequency span difficult, as very low frequency base noise patterns can not be accurately viewed in the fixed window while designing the layers with higher frequency. Prototype could be enhanced by not constraining the reference image size and introducing a zoom feature for both the reference image and the noise layer.

Realism of the produced terrains depends largely on the reference images used. It is possible to acquire plausible terrains, however the complexity of some real world terrains makes it difficult to acquire such terrains even with high amount of layers. For example, natural-looking erosion patterns [Musgrave, 1993] are difficult to obtain using only layer masking and redistribution because of the limited expressive power of the base noise function. The prototype does not have a concept of “biomes”, meaning regions with completely different characteristics. Applications using the designed procedural terrain could however use other techniques and multiple terrain functions to acquire such feature as the result function is still procedural noise by nature.

6. Conclusions

Noise functions have been used in a wide range of procedural content creation methods, including terrain generation. In this thesis various noise functions, their features and suitability for example based procedural terrain implementations were introduced and assessed. Various noise functions have been designed for different purposes, often related to computer graphics and procedural textures. Perlin noise, originally designed for natural-like procedural texture generation, is perhaps the most well known and most referenced noise implementation and still widely used today. Simplex noise can be considered as a successor of Perlin noise since it is functionally similar and produces similar patterns as its predecessor. Many other noise implementations have also been designed addressing some limitations of original Perlin noise, including more accurate spectral control and dealing with aliasing artifacts in procedural textures.

Existing research on noise based procedural terrain generation and example based synthesis were also covered. Spectral synthesis based noise techniques are able to accurately reproduce a limited subset of texture examples and these methods might also be of use in creating example based procedural terrains. The methods are, however, somewhat lacking in evaluation performance or don't have all the advantages of procedural noise, e.g., long period or non-periodicity, random accessibility or the absence of preprocessing. In this thesis, the truly procedural, random accessible noise based terrains were on the main focus. While noise functions and procedural generation in general are quite well adapted in games and applications, existing research on the specific subject of example based procedural terrain design is quite scarce and leaves plenty of room for further research.

A novel example-based noise processing method was introduced based on the color spline function presented by Ebert et al. [2002]. The method was based on value distribution matching and base frequency approximation. Automatic noise layer masking process was designed to allow the creation of more complex procedural terrains. A prototype implementing the explained methods was implemented and presented, including a storage format for the final procedural terrain. Prototype includes a 3D preview of the designed terrain, providing a view of the preliminary result of the terrain design process. It also servers as an example for utilizing the designed procedural terrain function in applications. Simplex noise was selected as the base for an example based procedural terrain synthesis method for it's true random-access quality, quality of pattern production and evaluation speed. It also has a reference implementation available, making it possible to be implemented in different environments and programming languages with exactly equivalent results. The prototype provides an intuitive graphical user interface for the terrain design process. It can be used solely by providing example terrain height maps – all noise parametrization is done automatically.

The presented methods and the prototype bridge the gap between high variety, infinite procedural terrains and intuitive design process of traditional terrain models. The resulting procedural terrain function is just summed noise by nature and it retains all the useful qualities of fractal procedural noise. It also remains fast enough for on-demand evaluation in applications. While not being a comprehensive solution to the procedural terrain by example problem it aids the creation of a wide variety of truly procedural, high-performance terrains by example.

References

[Brosz et al., 2007] John Brosz, Faramarz F. Samavati and Mario Costa Sousa, Terrain Synthesis By-Example, *Advances in Computer Graphics and Computer Vision. Communications in Computer and Information Science* **4**, 2007, 58-77.

[Carpentier and Bidarra, 2009] Giliam de Carpentier and Rafael Bidarra, Interactive GPU-based procedural heightfield brushes, *Proceedings of the 4th International Conference on Foundations of Digital Games, FDG*, 2009, 26-30.

[Cook and DeRose, 2005] Robert L. Cook and Tony DeRose, Wavelet noise, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005* **24**(3), 2005, 803-811.

[Ebert et al., 2002] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R. Mark, John C. Hart, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin and Steven Worley, *Texturing and Modeling: A Procedural Approach - Third Edition*, Morgan Kauffman Publishers, 2002.

[Field, 1989] David J. Field, What the statistics of natural images tell us about visual coding, *Proceedings of SPIE 1077, Human Vision, Visual Processing, and Digital Display* **1077**, 1989, 269-277.

[Galerne et al., 2012] Bruno Galerne, Ares Lagae, Sylvain Lefebvre and George Drettakis, Gabor noise by example, *Proceedings of ACM SIGGRAPH 2012* **31**(4), 2012, Article No. 73.

[Hill, 2016] Owen Hill, WE'VE SOLD MINECRAFT MANY, MANY TIMES! LOOK!, 2016, Retrieved from https://mojang.com/2016/06/weve-sold-minecraft-many-many-times-look/ .

[Julesz, 1962] Béla Julesz, Visual pattern discrimination, *IRE Transactions of Information Theory*, 1962, IT-8, 84-92.

[Kamal and Uddin, 2007] K. Raiyan Kamal and Yusuf Sarwar Uddin, Parametrically controlled terrain generation, *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, 2007, 17-23.

[Lagae et al., 2009] Ares Lagae, Sylvain Lefebvre, George Drettakis and Philip Dutré, Procedural Noise using Sparse Gabor Convolution, *Proceedings of ACM SIGGRAPH 2009* **28**(3), 2009, 54-64.

[Lagae et al., 2010a] Ares Lagae, Sylvain Lefebvre, R. Cook, T. DeRose, George Drettakis, David S. Ebert, J.P. Lewis, Ken Perlin and Matthias Zwicker, A Survey of Procedural Noise Functions, *Computer Graphics Forum* **29**(8), 2010, 2579-2600.

[Lagae et al., 2010b] Ares Lagae, Peter Vangorp, Toon Lenaerts and Philip Dutré, Procedural Isotropic Stochastic Textures by Example, *Computers & Graphics* **34**(4), 2010, 312-321.

[Musgrave, 1993] F. Kenton Musgrave, *Methods for realistic landscape imaging*, Doctoral thesis, Yale University, 1993.

[Parkin, 2014] Simon Parkin, No Man's Sky: A Vast Game Crafted by Algorithms, *MIT Technology Review, 2014*. Retrieved from https://www.technologyreview.com/s/529136/no-mans-sky-a-vast-game-crafted-by-algorithms/ .

[Perlin, 1985] Ken Perlin, An image synthesizer, *ACM SIGGRAPH Computer Graphics* **19**(3), 1985, 287-296.

[Perlin, 2001] Ken Perlin, Noise Hardware, *ACM SIGGRAPH 2002 Course 36 Notes*. Retrieved from http://www.csee.umbc.edu/~olano/s2002c36/ .

[Perlin, 2002] Ken Perlin, Improving Noise, *ACM Transactions on Graphics* **21**(3)**,** 2002, 681-682.

[Perlin and Hoffert, 1989] Ken Perlin and Eric M. Hoffert, Hypertexture, *ACM SIGGRAPH Computer Graphics - Special issue: Proceedings of the 1989 ACM SIGGRAPH Conference* **23**(3), 1989, 253-262.

[Persson, 2011] Markus Persson, Terrain generation, Part 1, 2011. Retrieved from http://notch.tumblr.com/post/3746989361/terrain-generation-part-1 .

[Pouli et al., 2011] Tania Pouli, Douglas W. Cunningham and Erik Reinhard, A Survey of Image Statistics Relevant to Computer Graphics, *Computer Graphics Forum* **30**, 2011, 1761-1788.

[Santamaría-Ibirika et al., 2014] Aitor Santamaría-Ibirika, Xabier Cantero, Mikel Salazar, Jaime Devesa, Igor Santos, Sergio Huerta and Pablo G. Bringas, Procedural approach to volumetric terrain generation, *The Visual Computer* **30**(9), 2014, 997-1007.

[Smelik et al., 2009] Ruben M. Smelik, Klaas Jan De Kraker, Saskia A. Groenewegen, Tim Tutenel and Rafael Bidarra, A Survey of Procedural Methods for Terrain Modelling, *In Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS),* 2009, 25-34.

[Smelik et al., 2010] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker and Rafael Bidarra, Declarative terrain modeling for military training games, *International Journal of Computer Games Technology* **2010**, Article num. 2.

[Smith, 1997] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing,* California Technical Pub, 1997. Retrieved from http://www.dspguide.com/ .

[Worley, 1996] Steven Worley, A Cellular Texture Basis Function, *Proceedings of ACM SIGGRAPH 1996 on Computer Graphics and Interactive Techniques,* 1996, 291-294.

[Yang et al., 2015] Wei Yang, Luhui Xu, Xiaopan Chen, Fengbin Zheng and Yang Liu, Chi-Squared Distance Metric Learning for Histogram Data*, Mathematical Problems in Engineering, Volume 2015 (2015), Article ID 352849.*

[Yin-Poole, 2016] Wesley Yin-Poole, No Man's Sky Sony's 2nd biggest ever PS4 launch in UK, *Eurogamer*, 2016. Retrieved from http://www.eurogamer.net/articles/2016-08-15-no-mans-sky-sonys-2nd-biggest-ever-ps4-launch-in-uk .