Tampere University

Niko Sainio

TERRAIN GENERATION ALGORITHMS

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
April 2023

ABSTRACT

Niko Sainio: Terrain generation algorithms
M. Sc. Thesis
Tampere University
Master's Degree Programme in Computer Sciences
April 2023

Procedural terrain generation has become common in games as a whole and in indie games in particular. With procedural terrain generation developers can relatively easily create static or dynamically expanding game areas. Also it is more cost effective since large part of manual work can be automated which traditional game areas would require.

Goal of this thesis is to introduce and evaluate different algorithms that are used or have potential use cases in terrain generation. Such algorithms as various noise functions, which are widely used in the realm of terrain generation, a number of dungeon algorithms, which use variety of methods to generate the dungeon, fractal algorithm, and volumetric terrain generation algorithm which uses a combination of noise and fractal algorithms. Algorithms and techniques will be searched from various scientific articles and literary sources. Metrics used for terrain generation algorithm evaluation will also be introduced, and algorithms in this thesis will be evaluated using these metrics.

During evaluation it was noticed that the evaluated noise functions are generally capable of runtime terrain generation, but are lacking in customization and control since parameters are usually related to the algorithm rather than the resulting terrain. Albeit these shortcomings both Perlin and Simplex noise stand out for their ability to generate good quality terrains. On the other hand most of the evaluated dungeon generation algorithms are incapable of generating terrain during runtime with few exceptions. Also guaranteeing connectivity of rooms or areas in dungeon can be challenge in some algorithms. The introduced fractal algorithm is metrics wise similar to Perlin and Simplex noise even though it uses completely different method to generate the terrain. The volumetric terrain generation algorithm is the only algorithm capable of generating volumetric terrain and its high level of parametrization and customization is its strongest quality.

Keywords: procedural terrain generation, noise, dungeon, algorithms

1 Introduction

Procedural generation of terrain has become more common in game development. This enables a developer to create game worlds cost effectively, thus avoiding large amount of manual labor, saving both time and money [Smed and Hakonen, 2017]. Manual approaches are also inherently unscalable which procedural terrain generation solves. Especially in indie games procedural terrain generation has become common most likely due to its cost effectiveness. Currently there is a huge amount of games that use different methods and algorithms to generate terrain automatically. Some evidence of popularity can be seen on the Steam platform as it even has a keyword and its own category for games that use procedural terrain generation.

Algorithms can be used to create finite or near infinite two- or three dimensional game areas using, e.g., a heightmap to determine terrain shape and surface. Procedural stands for automatic creation of assets, terrain or game area when need arises. Perhaps the most well-known example of three dimensional procedurally generated terrain is Minecraft. Example of terrain can be seen on the left in Figure 1. Minecraft is currently the best-selling game ever, having sold over 200 million copies according to Statista. In Minecraft more terrain is generated automatically when player moves in the game world. Terrain consists of voxels, which are three dimensional pixels/datapoints and are further explained in Chapter 4.1. These form the shape of the terrain such as mountains, canyons, lakes, underground caves and mineral deposits.

Terraria on the other hand is a two-dimensional example of terrain generation where game world is seen from the side, as shown on the right in Figure 1. Terrain is pre-generated in the beginning of the game and consists of squares, which form mountains, underground caves and mineral deposits similarly to Minecraft.

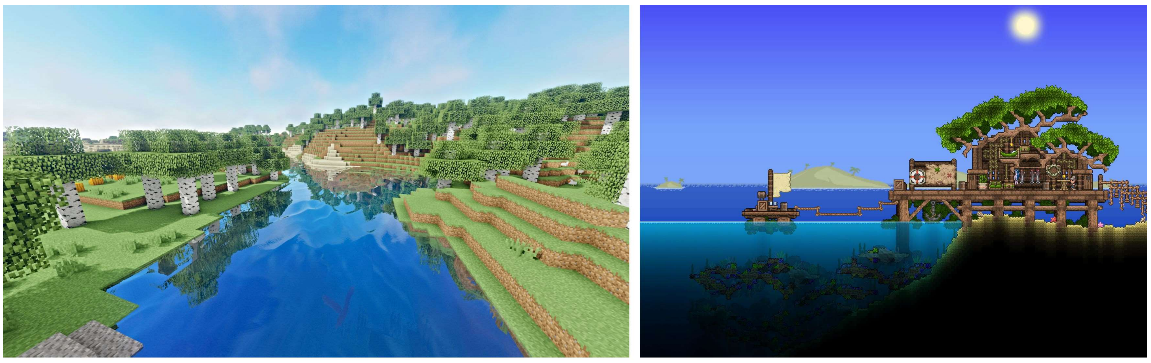


Figure 1. Minecraft (left) [Bayne, 2021], Terraria (right) [Conditt, 2021].

The simplest way to determine and generate a terrain surface is to use heightmaps. Figure 2 shows examples of a heightmap created using various algorithms. Heightmap is a two-dimensional matrix which contains numeric values. These values represent height. Heightmap can be visualized with a grayscale image where whiter areas have higher height value and vice versa blacker areas have lower height value.

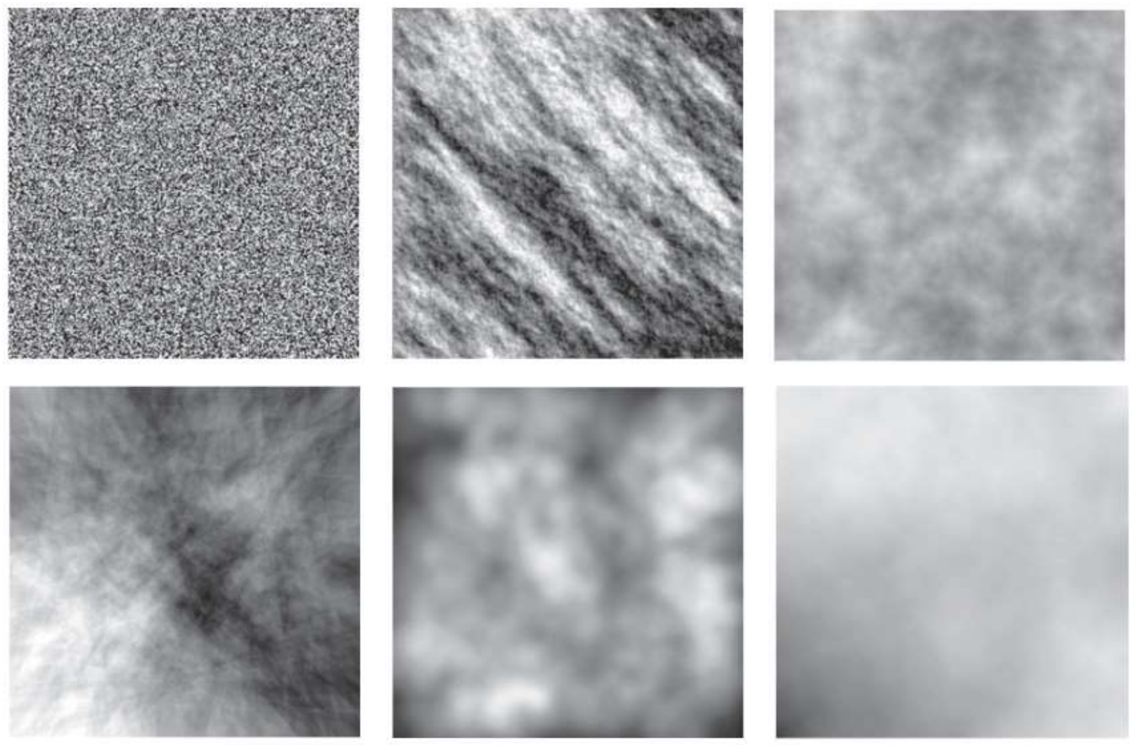


Figure 2. Examples of a heightmap [Smed and Hakonen, 2017].

To make the generated terrain more interesting it is possible to augment terrain generation by using procedural texture generation and other techniques. In procedural texture generation different threshold values for textures can be set so that terrain with specific height values are mapped with specific textures. This way for example mountains could be mapped with a rocky texture and their peaks with a snow texture as shown on the left in Figure 3. In addition a threshold for water level could be set so certain areas below such threshold would be covered in water. Similarly for board game like game areas, terrain tiles (e.g., sea, plains, hills, mountains) could be selected based on height values on a heightmap as shown on the right in Figure 3.

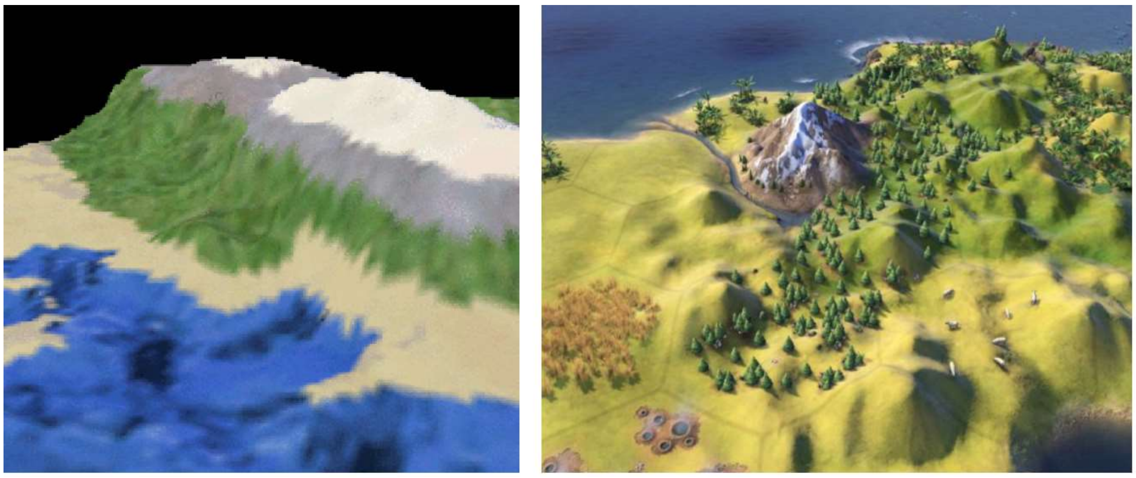


Figure 3. Simple procedural texture mapping (left) [Allain, 2019], a Civilization 6 tile map (right) [Robinson, 2021].

Goal of this thesis is to introduce and evaluate various algorithms that could be usable or have an interesting application in the realm of terrain generation. How they work will be also explained to some extent and their usability in terrain generation will be considered. This includes a number of noise functions and dungeon algorithms, fractal algorithm, and volumetric terrain generation algorithm that uses a combination of both noise and fractal algorithm. Metrics for evaluation of terrain generation algorithms will be introduced and described. Lastly the algorithms introduced in this thesis will be evaluated using these metrics.

In Chapter 2, the definition and theory of noise are given, then a number of most well-known and used implementations are covered. In Chapter 3, the theory of fractal algorithms and implementation of such is introduced. In Chapter 4, an algorithm that incorporates previous algorithms (noise, fractal) is presented. In Chapter 5, a number of dungeon algorithms and their theories are covered. In Chapter 6, metrics of evaluation of the terrain generation algorithms are described and algorithms introduced in this thesis will be evaluated. Final chapter consists of a conclusion.

2 Noise

Next, a number of noise functions will be introduced and roughly explained. First, a theory of noise function will be introduced. Then random value noise will be examined in Chapter 2.1 since it is the most basic form of noise and gives better understanding of noise. After that, in Chapter 2.2 and Chapter 2.3, Perlin noise and Simplex noise, which is also known as improved Perlin noise, will be introduced because they are widely known and used noise functions and well suited for generating both two- and three-dimensional terrain. In Chapter 2.4 the wavelet noise will be introduced since it is an intriguing application of wavelet function. In Chapter 2.5 the Worley noise will be examined for its interesting application of the Voronoi diagram. Lastly in Chapter 2.6 a number of noise functions not covered in this thesis will be briefly addressed.

Noise functions can be used to generate fascinating and realistically shaped terrains. First procedural noise function invented is the Perlin noise by Ken Perlin, which will be explained in greater detail later. An ideal procedural noise is non-periodic [Lage et al., 2010a]. Noise is pseudo-random, meaning that it appears random, but in actuality it returns the same value if given the same parameters. Noise is a mapping from R^n to R - you input an n-dimensional point with real coordinates, and it returns a real value [Perlin, 2001]. One- two- and three-dimensional noises are the most common use cases [Perlin, 2001]. Noise always returns a value between 0...1.

Perlin also states that noise is band-limited. This means that when looked as a signal the frequency range is limited causing most of its energy to be concentrated in a small part of the frequency spectrum. Thus low frequencies's (large shapes) and high frequencies's (visually small details) energy contribution is low. Its appearance is similar to what you would get if you took a big block of random values and blurred it (i.e. convolved with a Gaussian kernel). [Perlin, 2001]

Term *procedural* is used in computer science to distinguish entities that are described by program code rather than by data structures [Lagae et al., 2010].

Lagae et al. [2010] lists advantages of procedural noise:

- It requires very little space. Only a few kilobytes compared to megabytes for noise images, volumes.
- It can create noise at any resolution due to being inherently continuous, multi-resolution, and not based on discretely sampled data.
- It is non-periodic, filling the entirety of two-, three- to n-dimensional space.
- It is parametrized, meaning it can generate different kinds of noise patterns instead of just one fixed noise pattern. The parameters control the power spectrum of the noise, which characterizes the noise pattern.

- It is randomly accessible, meaning it can be evaluated in a constant time, regardless of the location of the point of evaluation, and regardless of previous evaluations.

The formal definition of noise is described next. For this, definitions for different random processes (see, e.g. [Papoulis & Pillai, 2002]) and Fourier analysis (see, e.g. [Bracewell, 1999]) will be examined. Following equations and explanations are based on the article presented by Lagae et al. [2010]. For a discrete-valued random process $y = N(x)$, the *nth order probability density function* (we shall call it *pdf*). [Lagae et al., 2010]

$$\int_N (y_1, y_2, \ldots, y_n; x_1, x_2, \ldots, x_n) = P(N(x_1) = y_1, N(x_2) = y_2, \ldots, N(x_n) = y_n) \tag{1}$$

is the simultaneous probability that the noise takes on particular values y_k at n specified locations x_k. The first-order pdf is commonly referred to as the amplitude distribution or the signal histogram. The *nth-order moments* are weighted averages of the corresponding nth-order pdfs. The first-order moment is the mean.

$$E[N(x)] = \int (y f_N(y)\, dy) \tag{2}$$

The second-order moment is the expected product of the noise at two locations and is termed the autocorrelation or autocovariance:

$$E[N(x_1) N(x_2)] = \iint y_1 y_2 f_N(y_{1.} y_2; x_1, x_2) dy_1 dy_2 \tag{3}$$

A stationary random function is one whose statistics are invariant to a shift in the origin of the coordinate system, and a random function is isotropic if its statistics are also invariant to rotation of the coordinate system. For a stationary and isotropic random function, the autocorrelation reduces to a function of a single variable.

$$E[N(x_1) N(x_2)] = R(|x_1 - x_2|) \tag{4}$$

The autocorrelation evaluated at zero is simply the standard definition of variance, $R(0) = E[(N(\cdot) - E[N(\cdot)])^2]$. Importantly, the power spectrum of the noise is the Fourier transform of the autocorrelation function of the noise. Most noise functions model or approximately model only the first- and second-order moments.

2.1 Value noise

Value noise is the simplest example of a noise function. It generates random values independent of each other. This is usually referred to as white noise. White noise contains equal amounts of all frequencies with a random phase, so it can be used as the raw material to generate unstructured signals with any combination of frequencies. A band-limited power spectrum is non-zero only within a specific range of frequencies. [Lagae et al., 2010]

Value noise itself generates values that are too random and as such seem unnatural, since in nature changes are more gradual. Therefore it is not suitable for, e.g., terrain generation alone. To make the value noise look more natural, a linear interpolation can be applied, or a more advanced noise function can be used instead to make changes more gradual and smooth, such as Perlin noise shown on the right in Figure 4.

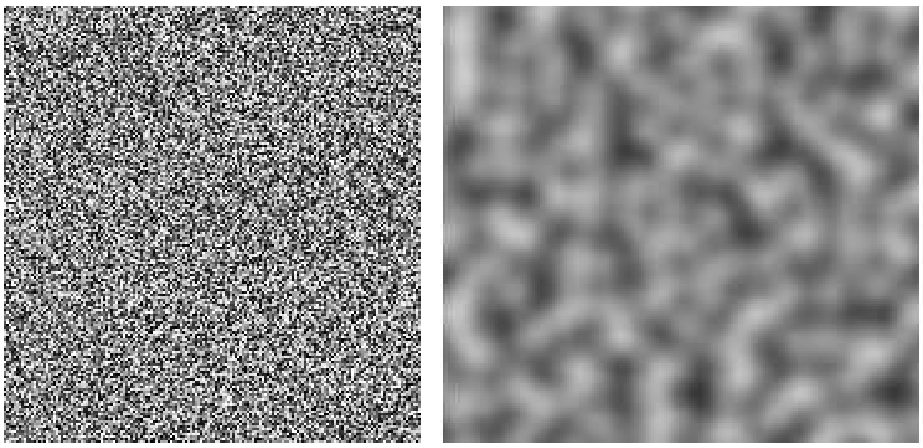


Figure 4. Value noise (left) [Stolfi, 2013] and Perlin noise (right).

2.2 Perlin noise

Perlin noise is a well-known lattice gradient noise function which was presented by Perlin [1985] and it was the first application of gradient noise [Ebert et al., 2002]. It was first used in the movie Tron made in 1982. An example of Perlin noise is shown in Figure 4 (right). It is often used in computer generated imagery (usually shortened CGI) and movie special effects industry since it can be used to create interesting textures and effects like rock, marble, fire, clouds etc [Perlin, 2001]. Due to this ability it is also used in terrain generation to create compelling terrain shapes and surfaces. Perlin introduced improved version of this noise [Perlin, 2001].

Perlin noise implements Gaussian distribution which means values are not uniform but centered at a midpoint between 0 and 1. Smoothness of the Perlin noise comes from the fact that any point in space is always related to its neighboring points.

For example the three-dimensional Perlin noise is gained for a point in space by calculating pseudo-random gradient (by hashing the lattice points) at each of the eight closest vertices on the three-dimensional integer lattice, and finally using a splined interpolation. The lattice points are hashed by breaking the correlation of the indices by successively applying a pseudo-random permutation to the coordinates. This generates an array of pseudo-random unit length gradient vectors. The set of 12 gradient vectors are determined by the directions from the center of a cube to its edges. To ensure continuous noise derivative, a quintic polynomial interpolation is used. 2D Perlin noise is calculated in a similar fashion except calculations happen in a two-dimensional space, meaning the pseudo-random gradients are calculated for four closest vertices instead of eight. Final noise is typically calculated by adding up multiple noise waveforms of different frequencies. Each waveform having half the frequency of the previous waveform, these are called octaves. [Lagae et al., 2010]

Perlin noise's wide use in terrain generation doesn't come as a surprise. It can generate realistic and varied terrains of different scales. Online nature of Perlin noise also has a couple of advantages: It can generate terrain theoretically infinitely, and since the terrain data is not pre-generated its requirements for memory are initially low.

2.3 Simplex noise

Perlin presented an improved version of Perlin noise, which uses a simplex grid. It conforms better to the ideal noise specifications represented in [Perlin, 1985] and addresses some of the shortcomings of the original noise. Perlin [2001] lists improvements of Simplex noise over the original noise:

- Provides a single uniform standard result on any platform.
- Is visually isotropic.
- Does not require significant table space to compute good pseudo-random gradients.
- Can have an arbitrarily large extent for its repeating virtual tile, at very low cost.
- Does not require multiplies to evaluate gradient at surrounding grid vertices.
- Does not produce visible grid artifacts.
- Does not produce visible artifacts in the derivative.
- Is cheaper to compute.
- Allows for a direct analytic computation of derivative at reasonable cost.
- Can be generalized to higher dimensions at relatively small computational expense.

Both noises look very similar, but the major visual difference between the two is that the Simplex noise implementation is visually isotropic. This means that it is impossible to see the produced image's original orientation. Visualization of this difference can be seen in Figure 5. [Perlin, 2001]

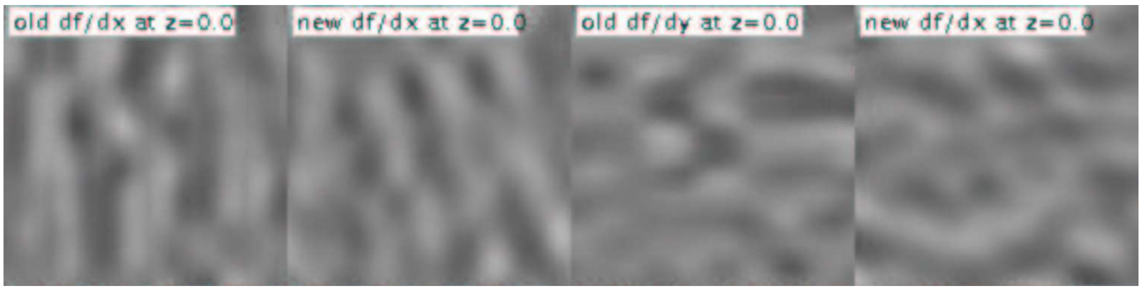


Figure 5. Comparison of the old and the new noise [Perlin, 2001].

To satisfy requirements mentioned previously, the Simplex noise calculates the index of a pseudo-random gradient at each surrounding vertex using a bit manipulation scheme, instead of a table lookup scheme. It also uses a simplicial grid instead of a cubic interpolation grid, which has two advantages during re-construction. First, only four component evaluations need to be done per noise evaluation, instead of the eight evaluations required in a cubic lattice. Second, the axis aligned visual impression of a grid structure is replaced by a far less visually noticeable simplicial packing structure. While the original Perlin noise uses a tricubic interpolation function, this interpolation scheme uses a spherically symmetric kernel, multiplied by a linear gradient, at each component surrounding vertex. This approach gives rise to three advantages. First, no directional artifacts due to the interpolation function. Second, no directional or discontinuity artifacts in gradient. Third, it is practicable to calculate the derivative function directly. [Perlin, 2001]

Simplex noise has clear improvements over Perlin noise both visually and computationally, and thus it is even better suited for terrain generation than the Perlin noise. The most interesting improvements being the lack of directional or discontinuity artifacts. And lower computational complexity even in higher dimensions, since complexity is only $O(n^2)$ compared to the original noise complexity $O(n2^n)$ [Perlin, 2001].

2.4 Wavelet noise

In 2005, Cook and DeRose noticed that when texturing two-dimensional surfaces by sampling a three-dimensional noise function, even if the three-dimensional noise function is perfectly band-limited, the resulting two-dimensional texture in general will not be band-limited. This means that just constructing a band-limited three-dimensional function does not solve the problem of a trade-off between loss of detail and aliasing. As a solution for this problem they introduced wavelet noise. Wavelet noise is almost perfectly band-limited, meaning it provides a good detail with minimal aliasing. This is demonstrated in Figure 6. [Cook & Derose, 2005]

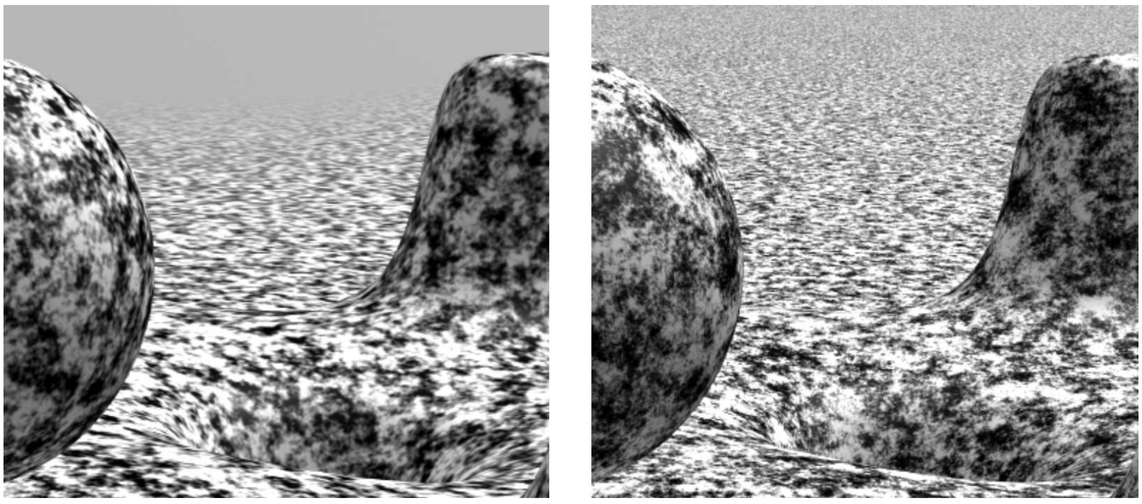


Figure 6. Comparison between Perlin noise (left) and Wavelet noise (right) [Cook & Derose, 2005].

For optimization purposes the algorithm uses noise tiles. In a pre-process, a tile of noise coefficients N is created. These coefficients represent the noise N(x) as a quadratic B-spline surface. This is done by creating an image R filled with random noise (shown in Figure 7a), downsampling R to create the half-size image R↓ (shown in Figure 7b), up-sampling R↓ to a full size image R↓↑ (shown in Figure 7c), and subtracting R↓↑ from the original R to create N (shown in Figure 7d). The tile of noise coefficients N is thus created by taking R and removing the part that is representable at half-size. The band-limited part is left as it is not representable at half-size. Wavelet analysis is used to ob-tain the filters used in the downsampling and upsampling steps which correspond to the analysis and refinement coefficients of the uniform quadratic B-spline basis function. The extension to further dimensions is straightforward. After the coefficients n_i have been determined, a value of N(x) for a given x can be computed during runtime using any evaluation method for quadratic B-splines. A small precomputed volume of noise coefficients is used and space is tiled with that volume. [Lagae et al., 2010]

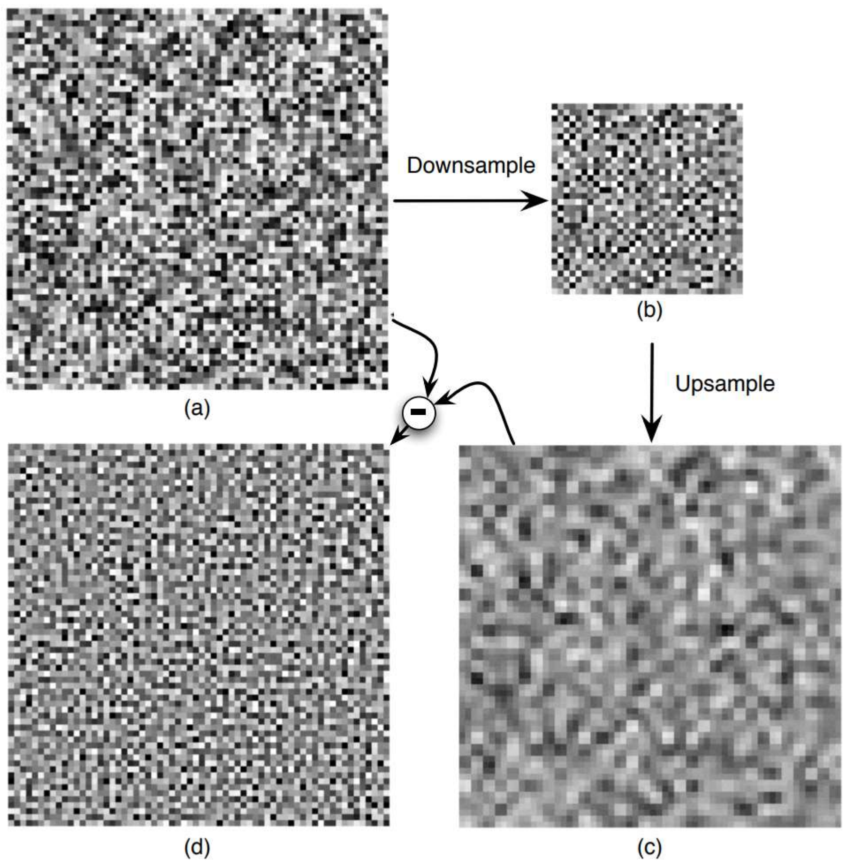


Figure 7. Wavelet noise process [Cook & Derose, 2005].

Hyttinen [2017] and Hyttinen et al. [2017] states, that wavelet noise is explicit by nature. This means that the noise is pre-processed and stored, unlike in, e.g., Perlin noise or Simplex noise. In the pre-processing step a fixed size volume of the coefficients is calculated. Then the volume is used to tile the resulting noise. The lack of true random accessibility causes memory consumption problems which are typical to explicit noises when very large random patterns need to be created. [Hyttinen, 2017]

Wavelet noise itself seems quite uniform and repeating as seen in Figure 7d, since it uses random noise as a basis. Thus it would not generate very interesting terrain and is better suited for texturing. Perhaps using Perlin noise as basis could alleviate such a problem. Fixed and pre-calculated values also make using wavelet noise for infinite and procedural terrain generation difficult.

2.5 Worley noise

Worley noise is a *texture basis function*, meaning it is used to create patterns that can be used as a basis for texture generation [Lagae et al., 2010]. It was introduced by Worley [1996] and is designed to complement Perlin noise. It is based on a partitioning of space into a random array of cells and is randomly accessible [Worley, 1996]. It can produce Voronoi like patterns and its implementation is very similar to that of sparse convolution noise [Lagae et al., 2010]. In original intended use it can be used to create textures such as craters, rock, ice, crumpled paper, organic crusty skin, mountain ranges and flagstone-like tiled areas [Lagae et al., 2010]. The noise uses randomly scattered points instead of points assigned to lattice points and the noise at a given location is derived from the distance to the *n*th closest feature point [Hyttinen, 2017]. Example of Worley noise can be seen in Figure 8. Even though Worley noise is mainly used for texturing it could have uses in terrain generation as well; such as creating moon or extraterrestrial planet surfaces, rocky areas or other terrains with similar surfaces.

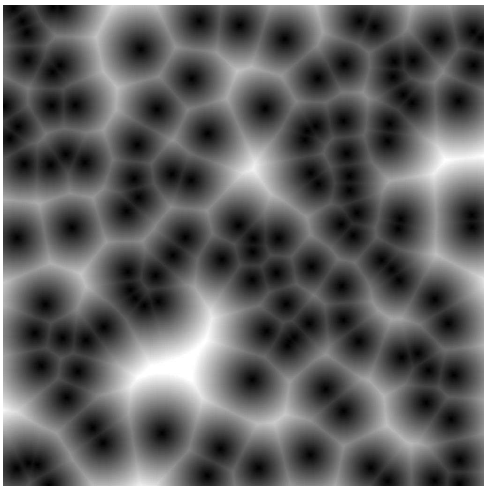


Figure 8. Worley noise [Rocchini, 2012].

2.6 Other noise functions

There are a plethora of other noises with possibly interesting applications for terrain generation, which are not covered in this thesis. Some uncovered lattice gradient noises (such as Perlin noise covered in Chapter 2.2) include noises like *lattice convolution noise* presented by Wyvill and Novins [1999], which is inspired by sphere packing and based on a more densely and evenly packed grid. *Better gradient noise* by Kensler et al. [2008], which adds three mutually orthogonal improvements to Perlin noise. *Flow noise* by Perlin and Neyret [2001], that is a Perlin-like noise function for generating time-varying flow textures with swirling and advection. And *curl noise* by Bridson et al. [2007], which is a Perlin-like noise function for generating time-varying incompressible turbulent velocity fields. [Lagae et al., 2010]

Some uncovered explicit noises (such as Wavelet noise introduced in Chapter 2.4) include noises like A*nisotropic noise* by Golderg et al. [2008], which supports high-quality anisotropic filtering. *Fourier spectral synthesis* introduces by Anjyo [1988], Saupe [1988] and Voss [1988], which generates a noise with a specific power spectrum by filtering white noise in the frequency domain. [Lagae et al., 2010]

Sparse convolution noises not examined in this thesis include noises such as the *sparse convolution noise* introduced first time by Lewis in [1984], which generates noise as the sum of randomly positioned and weighted kernels. *Gabor noise* introduced first time by Lagae et al. [2009] which produces noise by using the Gabor kernel. *Spot noise* by van Wijk [1991], which is used to generate stochastic textures for the visualization of scalar and vector fields over surfaces. [Lagae et al., 2010]

3 Fractal algorithm

Next, a fractal algorithm will be introduced for its intriguing application of fractals and because it is relatively easy to understand. First in Chapter 3.1 the theory of fractals is roughly explained. Then in Chapter 3.2, the basic general idea of fractal algorithm is explained and a simple three-dimensional application of the fractal algorithm will be introduced and explained.

3.1 Theory

The main feature of fractals is self similarity. This means that an object is said to be self-similar when magnified subsets of the object look like or are identical to the whole and to each other [Voss, 1988]. For example human circulatory system, where branching pattern of arteries is similar on different levels of magnification, or the surface of a rock where the shape of the surface looks similar regardless of the magnification. Fractal is an iterative process and an example of this process is shown in Figure 9. Fractal is called multifractal if self similarity is actualized in multiple different scales.

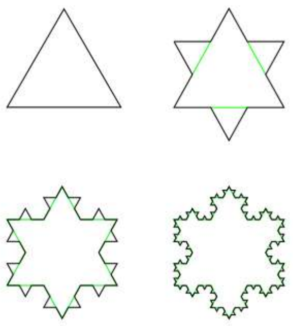


Figure 9. Four first iterations of Koch's snowflake [Phreneticc & Wxs, 2022].

3.2 Diamond-Square algorithm

One application of fractal algorithms is the Diamond-Square algorithm. It is also known as a random midpoint displacement fractal, a cloud fractal or a plasma fractal. The algorithm was first introduced by Fournier, Fussell and others at SIGGRAPH [1982] and it can be used to create terrain nearly infinitely. Basic principle is subdivision of primitive in a more stochastic, rather than deterministic manner by applying factorial Brownian motion (see [Fourier et al., 1982]). As an example this can be done by adding data points between data points to form equal sub-sections as shown in Figure 10 and giving them pseudo-random values. This value is usually calculated by taking the mean value of neighboring data points and randomizing it by adding, retracting or multiplying with random value. This is perhaps the easiest and most applicable algorithm for generating terrain.

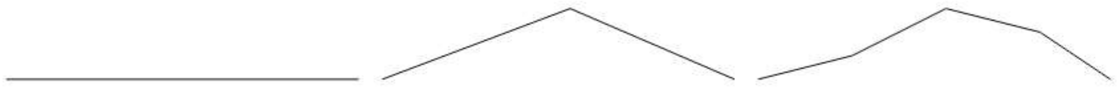


Figure 10. Basic principle of the fractal algorithm.

In the algorithm a square with uneven amount of datapoints is formed, the value of which can be calculated with formula (5), where l is the number of datapoints per side and n is a parameter in the exponent. For example $2^2 + 1$ would result in 5×5 datapoint grid. Datapoints are added forming smaller and smaller squares inside. The algorithm consists of three steps: starting step, where randomized values are set in corner data-points of the square. After this alternating diamond and square phases are done until the desired amount of sub-squares has been formed.

$$l=2^n+1 \tag{5}$$

In the diamond step, set the midpoint for each square in the grid. In square step, set the midpoint for each diamond in the grid. A simple example of an algorithm is to create a two-dimensional array with same uneven width and height and set randomized values to each corner cell. In the first step, which is the diamond step, calculate average of the corner values and set it to a midpoint of the square that has formed. In the square step calculate averages of the corner values and set it to a midpoint of the diamonds or a midpoint of hypotenuse of the triangles that has formed. Repeat last two steps in the same order until every cell has been assigned a value. These steps are shown in Figure 11. An algorithm has multiple variations how values can be randomized to increase variation and ”roughness”. Simplest ways to do this are to add or subtract a random value, or multiply the calculated mean value with roughness constant r. The roughness constant can be calculated with formula (6), where h is a parameter based on Gaussian distribution, meaning the value should be set between 0...1. Lower values of h result in rougher terrain and higher values in smoother terrain.

$$r=2^{-h} \tag{6}$$

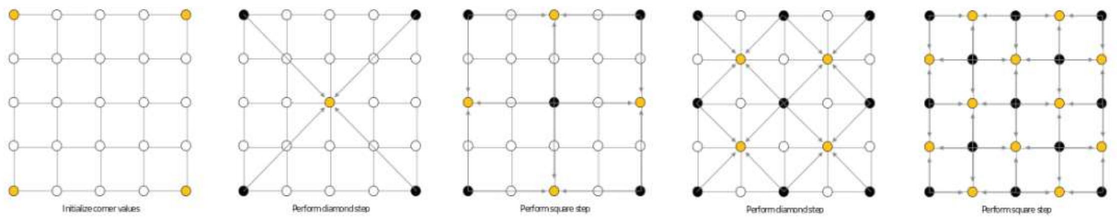


Figure 11. Diamond-square algorithm steps [Ewin, 2015].

Midpoint displacement algorithm’s strongest advantage is that it is easy to understand, and its simplicity, but it is not without problems. Miller [1986] states that fractal subdivision methods are slow and generate defects due to what is known as the 'creasing problem', which is the occurrence of creases or slope discontinuities along boundaries. In his paper he also presents an improved algorithm, which is not covered in this thesis.

4 Volumetric terrain generation algorithm

Next, a volumetric terrain generation will be introduced because of its interesting use of both a noise function and a fractal algorithm, and the ability to generate volumetric terrain like seen in the Minecraft video game. In Chapter 4.1 basic idea behind the algorithm and its major features will be explained roughly. Then in Chapter 4.2, terrain generation process and the mathematical theory of the algorithm will be explained.

4.1 Introduction

A volumetric terrain generation algorithm was presented in Vis Comput by Santamaría-Ibirika and others [2014]. The aim of the algorithm is to generate volumetric terrains with layered materials and other features such as mixtures of materials, mineral veins, underground caves, and underground material flow. An example of generated terrain is shown in Figure 12. The terrain generation process is also highly customizable as parameters can be used to influence characteristics of materials, terrain and the end result. [Santamaría-Ibirika et al., 2014]

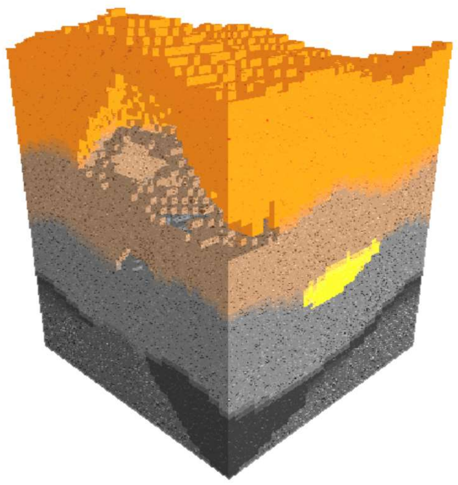


Figure 12. An example of the resulting terrain [Santamaría-Ibirika et al., 2014].

Traditionally procedurally generated terrains are based on fractals and noise functions. A shortcoming of these methods, while they generate realistic terrains, is that they are not very customizable. Commonly voxel-based representation has been used for volumetric data, which stores the volumetric data in three dimensional grid as shown in Figure 13. Also it is not very optimal considering the speed of accessing the data or the memory usage because three dimensional terrain ($N_t \times M_t \times L_t$) requires equal number of voxels to store the data. This method circumvents such issues by utilizing a representation system which divides the volumetric data into three levels: chunks, columns and limits. [Santamaría-Ibirika et al., 2014]

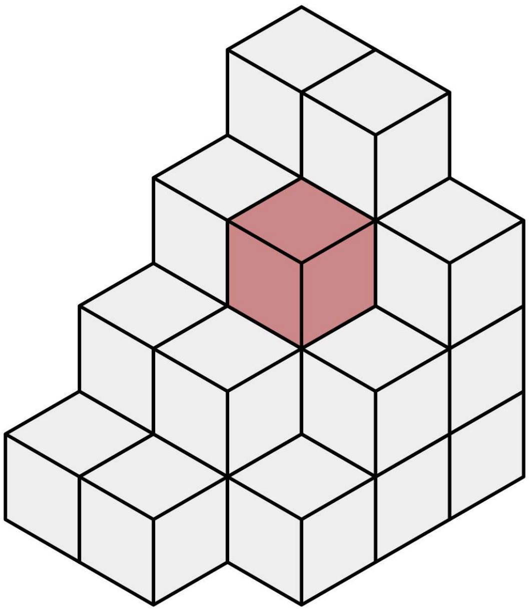


Figure 13. Stacked voxels with single shaded voxel [Santamaría-Ibirika et al., 2014].

The Chunk division is the highest level of data representation and shown as the green area in Figure 14. It is represented by a three-dimensional section of terrain with a size of N × M × L voxels. The Column division is the second highest level of data representation and shown as blue area in Figure 14. Each chunk is formed by a two-dimensional N × L regular grid of columns. Each column is represented by the stacked material at one point of the chunk which stores the data of M voxels. The Limit division is the lowest level of data representation and is shown as red blocks in Figure 14. The columns are formed by an arranged collection of elements (called limits), which represent the volumetric data in one point of the column. Each column has bottom and top limit and can have maximum of M limits meaning each voxel in a column can be a limit. [Santamaría-Ibirika et al., 2014]

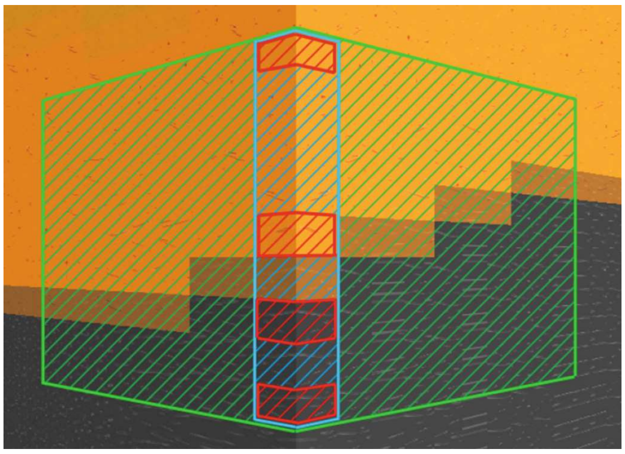


Figure 14. Chunk (green), column (blue) and limits (red) [Santamaría-Ibirika et al., 2014].

4.2 Terrain generation process

Terrain generation takes place in three steps. In the first step, input data is read and materials indexed by their parameters. In the second step, the algorithm creates veins and layers of the terrain, and in the last step they are converted to volume representation system introduced in the previous chapter. First step uses two sets of parameters: general parameters and material data. Material data has three additional sets of parameters. These sets of parameters are listed and described by Santamaría-Ibirika et al. [2014]:

First, material descriptors:

- Name
- Id
- Color, RGB value
- Special parameters,
  - *Void*, defines empty voxels in terrain.
  - *Base*, material for the bottom layer.
  - *Alone*, cannot be mixed with other materials.
  - *Vein*, structures material into vein instead of layer.
  - *Cave*, material for the empty voxels of the caves.
- Textures

Second, layer, vein and cave related parameters. These are defined for each material (all values are normalized to 0–1 range):

- *Area*, used with vein and cave materials. Specifies vein or cave area.
- *Depth*, determines the minimum, maximum and optimum depth of the elements formed with this material.
- *Thickness*, determines the minimum, maximum and optimum thickness of the layers, veins or caves formed with this material.
- *Roughness*, determines the minimum, maximum and optimum roughness of the elements formed with this material. For cave and vein materials, it represents the constants for the fractal algorithm, affecting the shape roughness. For regular materials, it represents the roughness of the surface.
- *Similarity*, specifies the similarity of the shape between two consecutive layers.
- *Blend,* specifies the suitability of the material to be mixed with others.

Third, affinity of materials. Each pair of materials is given two affinity values: the affinity rate of the materials in consecutive layers and the affinity rate of the materials in the same layer. Both values are expressed in a 0–1 range if the materials are compatible, and with a -1 if they are not. 0 means the materials have very low probability of being blended, but might be mixed for other reasons.

General parameters are values which affect whole terrain:

- *Terrain depth*, depth of the terrain from the deepest layer to the surface.
- *Layer thickness*, converts thickness of each layer into an absolute value.
- *Cave and vein area,* affects the size of the area of each vein and cave.
- *Cave and vein density,* establishes the density of veins and caves in the terrain.
- *Roughness,* converts roughness parameter of each material into an absolute value.
- *Layer similarity,* affects the similarity parameter of each material.
- *Scale*, Scale of the terrain.
- *Cave and vein thickness*, converts thickness of each cave and vein into an absolute value.

This process starts by generating layers, veins and caves of the terrain. The bottom layer is generated first. After this more layers, veins and caves are generated and stacked using given parameters. The process ends once desired depth has been achieved. Layers are generated by choosing the materials. For the bottom layer, material with a base parameter defined is chosen. Otherwise affinity parameters of the previous layer are used for compatible materials and a global affinity is calculated [Santamaría-Ibirika et al., 2014]:

$$A_i = \sum_{j=1}^{N} (a_{ij} \times c_j) \tag{7}$$

where
A_i is the global affinity of ith possible material,
N is number of materials in previous layer, and
a_{ij} is the affinity between ith possible material and jth material in the last layer.

After this the probability of each material to appear in the new layer is calculated:

$$P_i = A_i / (\sum_{j=1}^{N} A_j) \tag{8}$$

where
P_i is the probability of the ith possible material calculated in equation (7),
A_i is the global affinity ith possible material calculated in equation (7),

N is the number of possible materials, and
A_j is the global affinity of the *j*th possible material.

The random value and probabilities calculated in equation (8) are used to select the main material for the new layer. The layer has a 50% probability of containing second material and halving probability for consecutive materials using the already calculated affinities for previous materials so that the addition of concentrations equals 1. After this, the concentration for selected materials is calculated. If only one material is selected the concentration is 1, otherwise main material is given a concentration between 0.5...0.9 and the concentration for remaining materials is randomized. Then properties (relative thickness, similarity, roughness etc.) of the layer will be calculated [Santamaría-Ibirika et al., 2014]:

$$V=\sum_{i=1}^{N}(v_i \times c_i) \tag{9}$$

where
V is the property value,
N is the number of materials in the layer,
v_i is the property value of the each material, and
c_i is the concentration of the each material.

Then, layer heights are calculated by multiplying each layer's roughness by Perlin noise value which uses the scale of the terrain as frequency. Then a similarity modifier is applied [Santamaría-Ibirika et al., 2014]:

$$h=h_{-1}\times s+n\times(1-s) \tag{10}$$

where
h is the modified height,
h_{-1} is the height of the last layer at the same point,
s is the previously calculated similarity, and
n is the Perlin noise value.

Equation (10) modifies the height of the new layer to be comparable to the previous layer's height at the same point. This will cause the new layer to be relatively similar in shape to the previous layer. Lastly the underground material flow is generated and its tangent is set to the layer surface, and then rotated on Y-axis using Perlin noise. [Santamaría-Ibirika et al., 2014]

Vein generation happens in 6 steps. In the first step the algorithm selects a random location for the vein, generating a base bounding square on it. The length of the square is randomized between 0.5...1 times *VeinArea* where *VeinArea* is the general parameter for the vein area. In the second step, the algorithm chooses materials for the vein similarly to the layer generation process, but only selects the main material with a *vein parameter* and only gets the affinities of the materials located under the base bounding square. In the third step, properties for the vein (relative thickness, relative area, roughness etc.) are calculated similarly to an equation (9). Then the final bounding square is obtained by multiplying base bounding square by the relative area. In the fourth step, the relative vein shape is generated using a midpoint displacement fractal algorithm, which was introduced in the Chapter 3.1. The relative shape is generated by locating a square and displacing its vertices along the square diagonals (Figure. 15a). Then it takes a random point (P) from a segment, displacing it along the line formed by the new point (P) and the square center (Figure 15b). Equation (11) is used to calculate the maximum that is then used to randomize the point displacement value. This process is recursively performed in all segments. Due to time performance restraints, the algorithm's recursive depth is limited to 5. The minimum point displacement is $-d_i$. [Santamaría-Ibirika et al., 2014]

$$d_i = (d_{i-1} / SmoothConstant) / 2 \tag{11}$$

where
i is the recursive depth of the algorithm,
d_i is the maximum displacement at the ith recursive depth,
d_{i-1} is the maximum displacement at the i-1th recursive depth, and
SmoothConstant is a constant value (roughness parameter of the materials) which indicates roughness loss in each iteration.

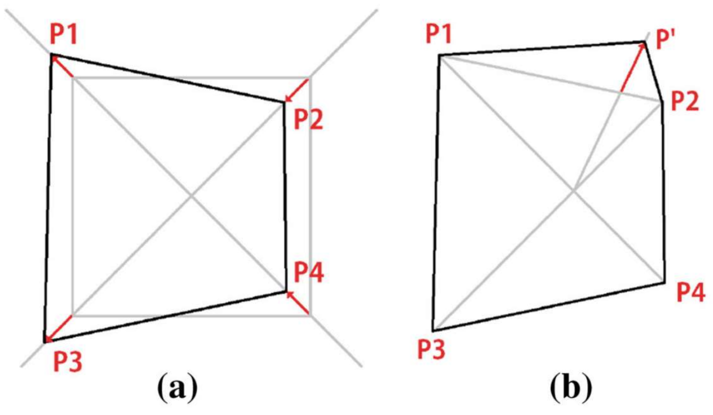


Figure 15. Vein generation with fractal algorithm (midpoint displacement) [Santamaría-Ibirika et al., 2014].

In the fifth step, the algorithm establishes the vein's relative height by using the minimum distance from each point to the border of the shape. Equation (12), which is based on the circumference located at point (1, 0) and a radius of 1 unit length, is used to calculate the height [Santamaría-Ibirika et al., 2014]:

$$h_{xy}=|\sqrt{1-(d_{xy}-1)^2}| \tag{12}$$

where,
h_{xy} is the height at the point (x,y), and
d_{xy} is the distance from the point (x,y) to the border.

In the final step, the algorithm multiplies the global thickness, thickness and relative height to calculate the final heights of the vein. Then it adds the heights of the last layer under the final vein shape. Lastly, the same steps used in the vein flow process are used to generate the underground material flow. [Santamaría-Ibirika et al., 2014]

Cave generation goes through the same steps as vein generation with a few differences. In the first step, the algorithm establishes a base bounding square and the location, but instead of the *VeinArea* parameter, the *CaveArea* parameter is used. In the second step, a material is selected similarly as in the vein material selection with couple differences: an algorithm selects only one material from materials with a *cave* parameter set. Then, the properties of the material are extracted and the bounding square of the cave modified as in the vein generation process. [Santamaría-Ibirika et al., 2014]

Previous sections explain how the algorithm generates plot of terrain from scratch. Extending an already generated terrain is similar to generating a terrain from scratch, except that the composition and properties of the layers are calculated beforehand. Firstly, the algorithm calculates the numbers of caves and veins in the new plot of terrain. This is done by taking a random value between 0.1...1.5 and multiplying the number of caves and veins in the original plot. After this, the algorithm randomly determines the positions of caves and veins between layers resulting in stacks of caves, veins and layers. Next, the algorithm generates the elements of the stack beginning from the bottom. It extends the layers horizontally along the X and Z coordinates obtaining new heights with the Perlin noise algorithm similarly to generating a layer from scratch, except that the composition and properties of the layer are pre-calculated. Caves and veins are generated in the same fashion as previously. [Santamaría-Ibirika et al., 2014]

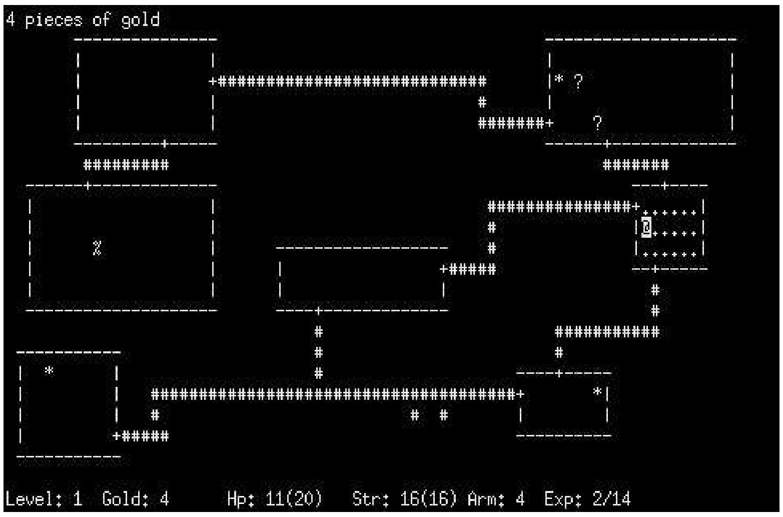
The algorithm is an interesting application of voxels in terrain generation which allows gameplay similar to the Minecraft. Instead of just exploring the surface of the terrain generated with, e.g., Perlin noise, the player can also dig deeper into the ground and explore the terrain in truly three-dimensional fashion. High level of parametrization is also one of its best features as it allows the terrain designer to have greater control over the resulting terrain.

5 Dungeon algorithms

Next, general idea and the most basic approaches to dungeon generation will be explained. Then a number of various dungeon generation algorithms will be introduced. In Chapter 5.1, the theory of cellular automaton algorithm and application of it in the realm of dungeon generation will be introduced. Then in Chapter 5.2, the theory of genetic algorithms and a number of its applications in dungeon generation will be described because of the intriguing nature of the genetic algorithm itself. In Chapter 5.3 a constraint-based dungeon generation method will be introduced because of its usage of an undirected graph, and since it can generate three-dimensional dungeons unlike other dungeon generation algorithms introduced in this chapter. Finally in Chapter 5.4, a method used in the TinyKeep video game and its process will be explained, because of its interesting use of multiple different techniques and methods.

Dungeon generation algorithms are one of the earliest forms of procedural terrain generation in computer games. First examples are considered to be Beneath Apple Manor (published in 1978), and Rogue (published in 1980) which gave the whole genre its name, the r*oguelike.* Procedurally generated dungeons are the main characteristic of the *roguelike* genre for many people [Garcia, 2020]. Possibly because of the infinite replayability and the dungeons being different each time. Maze algorithms could also be considered dungeon algorithms as maze like maps have been used in multiple Dungeon & Dragons dungeon crawling video games. In these games the player usually explores a two-dimensional maze with one or more adventurers collecting treasures and fighting monsters. The Bards Tale (published in 1986) and Dungeon Master (published in 1988) are earliest examples of this genre.

There are multiple ways to generate dungeons, but the most basic methods are as follows: In the first method, the rooms are placed first and then connected by laying tunnels between them. In the second method, the tunnels are placed first using some sort of maze generation algorithm and then the segments of the maze are connected by placing the rooms. In the third method, the rooms are connected similarly to The Binding of Isaac or The Legend of Zelda where rectangular rooms fill the whole space, and the player moves to another room by moving to one side of the screen. The third method is probably the easiest to understand since it does not require any tunneling algorithm and rooms can be added next to each other in a grid form which can be easily stored in an two-dimensional array, but a graph could be also used. The first method is probably the most common (shown in Figure 16), but the shortcoming of this method is that the connections between all parts of the dungeon are not guaranteed, so it may be required to re-generate the dungeon layout. In very complex dungeon layouts the algorithm used for tunneling can become quite complicated and may fail to connect the rooms, which consequently can result in disconnected segments of the dungeon. It is possible to validate connectivity between all rooms by using a spanning tree algorithm. This is done by adding rooms at the end of the corridor to the spanning tree. Additionally there exists a wide variety of algorithms of various complexities, which can be used to generate the dungeon or rooms to make the dungeon more interesting and/or natural. In most implementations the levels are pre-generated due to the finite nature of dungeons and mazes (there are distinct start and end points), but the run time generation of infinite dungeon is also possible.



```
4 pieces of gold
Level: 1  Gold: 4      Hp: 11(20)   Str: 16(16) Arm: 4  Exp: 2/14
```

Figure 16. An example dungeon from Rogue [Thedarkb, 2021].

5.1 Cellular automata

In addition to the procedural terrain generation, cellular automata have many uses in games. Such as modeling environmental systems like fire, heat, rain, fluid flow, pressure and explosions. They have also been used in procedural terrain generation to model both thermal and hydraulic erosion. [Johnson et al., 2010]

Cellular automaton is a self-organizing structure which consists of an N dimensional grid of cells. It is an iterative process, where each cell has an initial state at time zero (t = 0) and a reference to its neighbours. Cell’s and its neighbours’ state in next generation (t +1) will be calculated by applying a defined set of rules. It may then form patterns after some number of generations, that are influenced by the used cell states and rules. A grid of cells and cell states is the representational model of a cellular automaton. [Linden et al., 2014]

One good example is an algorithm proposed by Johnson et al. [2010] where they use a cellular automata to generate cave-like levels. The neighbourhood is defined with Moore neighbourhood, which means that all of the cell’s eight surrounding cells are considered its neighbours. Cell has three possible states which are wall, rock or floor.

The two dimensional grid is initialized by setting all cells' state to floor and then randomly changing a number of cells to rock. Then in multiple generations, following rule set is iteratively applied [Johnson et al., 2010]:

1. A cell is a rock if the neighborhood value is greater than or equal to T and floor otherwise.
2. A rock cell that has a neighboring floor cell is a wall cell.

The algorithm also uses these four parameters to control the generation process:

- Percentage of rock cells.
- Number of cellular automata generations.
- Neighborhood threshold value that defines a rock.
- Number of neighborhood cells.

A fascinating part of the method created by Johnson et al. [2010] is that it can generate infinite dungeons and generate new terrain even during gameplay. It is also quite straightforward. On the other hand there are some shortcomings, such as a lack of direct control, and that it can only be used to generate 2D terrains. Also, connection between two rooms cannot be guaranteed (as seen in Figure 17) and as such, connections between rooms would have to be tested and added if needed.

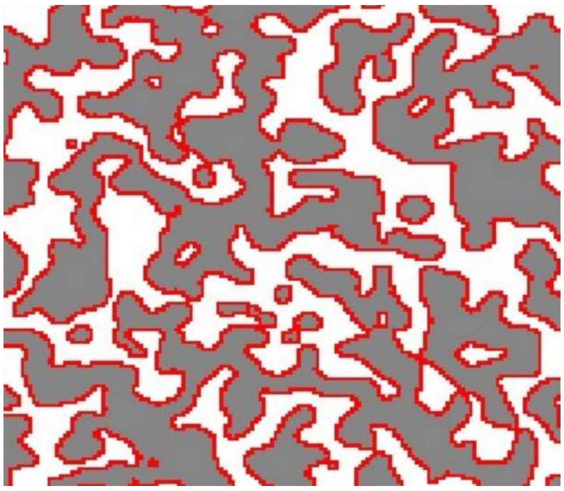


Figure 17. A cave like terrain generated by the cellular automata, where gray represents floor, red represents wall and white represents rock [Johnson et al., 2010].

5.2 Genetic algorithms

A genetic algorithm emulates the process of evolution, so they are also referred to as an evolutionary algorithm. The algorithm tries to find the best possible solution to an optimization problem. For it to function, the genetic algorithm requires a genetic representation and a fitness function. The genetic representation defines how a solution is encoded into a string. This encoded string is called a chromosome or a gene. The fitness function measures the quality of the solution and is used to select solutions (parents) to generate next the generation of solutions (children). A genetic algorithm is an iterative process. In each iteration it calculates fitness for the solutions and then selects and combines solutions into new solutions. Usually the probability for a solution to be selected is proportional to its fitness value. The process of combination is called a crossover. Also, in a mutation process a single character in a string may be randomly changed with some predefined probability. Mutations guarantee that in an infinite number of iterations, the best possible solution will be found. Obviously using infinite iterations is not feasible, so usually genetic algorithm is set to run a desired amount of iterations, and/or finish after a solution with a desired fitness value has been generated. [Linden et al., 2014]

5.2.1 **Search-based generation**

Ashlock et al. [2011] presents several methods to generate dungeons using genetic algorithm:

- Direct binary: Open or blocked squares within grid specified by binary gene.
- Direct colored: Squares are assigned colors specified by gene with letters.
- Indirect positive: A chromosome specifies structures that are placed on the empty grid.
- Indirect negative: A chromosomes specifies structures that are removed from a filled grid.

The most intriguing method is the direct binary method, since it manages to generate interesting and natural looking dungeons as seen in Figure 18. Direct colored method will be also examined for its ability to generate maze-like dungeons. Indirect positive and indirect negative methods will be addressed only briefly.

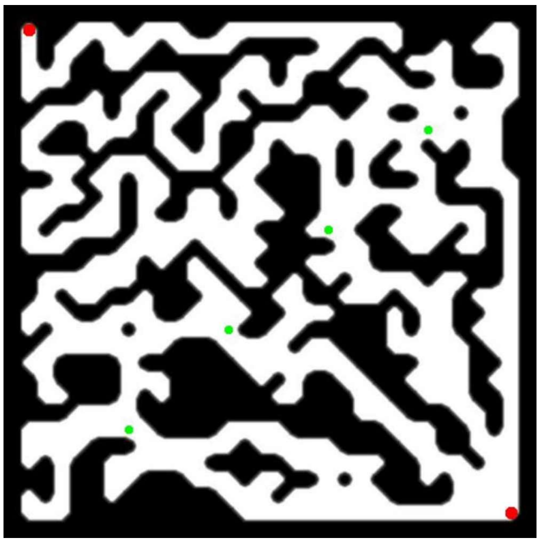


Figure 18. A dungeon generated using the direct binary method. Larger red dots represent the entrance and the exit. Smaller green dots represent checkpoints used by the fitness function [Ashlock et al., 2011].

The direct binary method uses two-dimensional XY board and its representation model is a string of XY bits. For the crossover the algorithm uses a uniform crossover with probability p_c, which combines two chromosomes by exchanging bits in each index end to end. This exchange happens independently at each index based on probability p_c. Mutation uses uniform mutation with probability p_m, which flips the value of a bit in each index of the string independently based on the probability p_m. A *fill* parameter: $0 < fill < 1$ was added to the algorithm. It defines the probability that a cell will be a wall. This parameter was added to address a problem where probability that the dungeon can be traversed is nearly zero if wall and void squares are equally likely. The *fill* parameter is set to 0.05, which causes the initial traversable mazes to have a relatively low fitness value. This will allow the crossover and mutation operators fill in added obstructions and increase fitness of the solution in a selection-guided manner. This technique will be called *sparse initialization*. [Ashlock et al., 2011]

The direct colored method uses two-dimensional *XY* board similarly to the direct binary method. The difference being that instead of using a string of *XY* bits, the algorithm uses a string of *XY* values in the range of $0 \leq x \leq 5$, which are mapped to values $\{R, O, Y, G, B, V\}$ (red, orange, yellow, green, blue, violet) as a representation model. Similarly to the direct binary method, the direct colored method uses a uniform crossover with probability p_c, which combines two chromosomes by exchanging color values at each index. This exchange happens independently at each index based on probability p_c. Mutation uses a uniform mutation with probability p_m, which changes the color value at each index independently based on probability p_m. Like the direct binary method, the direct colored method also uses a variation of *sparse initialization* to initialize the grid. The *sparse initialization* fills the grid with yellow and green squares with equal probability. This will cause the initial grid to have approximately the same amount of yellow and green squares. A maze initialized in this way initially allows movement between all pairs of squares and makes it likely that the mutation process creates barriers between the squares. Example result of a direct colored algorithm is shown in Figure 19. [Ashlock et al., 2011]

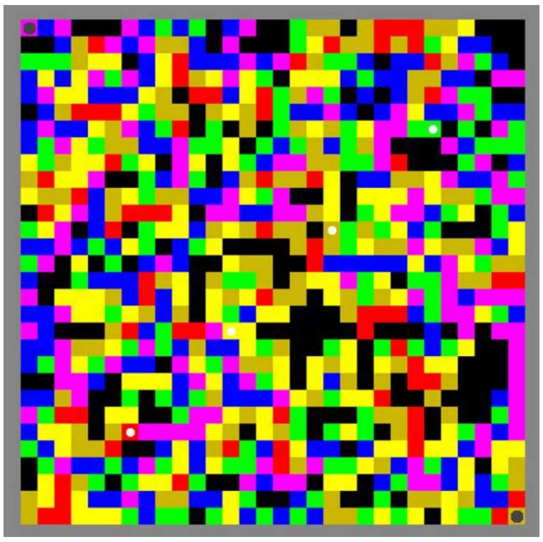


Figure 19. An example of the direct colored algorithm. The black circles mark the entrance and the exit, the white dots mark checkpoints [Ashlock et al., 2011].

The Positive indirect method uses a one-dimensional array of 80 cells to store integers in the range of 0...9999. The algorithm uses cellular representation similar to the one used in [Ashlock et al., 2006]. Instead of evolving the direct description of a solution, in the cellular representation the directions for how to construct the solution is evolved [Ashlock et al., 2006]. Barriers in an empty two-dimensional grid *XY* are defined by each pair of the integers in the array. Length of these barriers is calculated by bit-slicing the first integer into a one-bit number, a three-bit number, and a remainder R, which is taken modulo the larger of *X* or *Y*. The penetration of the barrier is determined by the one-bit number. The three-bit number tells which point of the compass (N, NE, E, SE, S, SW, W, NW) the barrier will run from its origin. The second integer *i* is split as $x = i \bmod X$ and $y = ((i \operatorname{div} X) \bmod Y)$ to obtain the starting point (x, y) of the barrier on the two-dimensional grid. The operator *div* represents integer division without the remainder. A nonpenetrating barrier runs its whole length unless it crosses any other barrier, and a penetrating barrier will run its whole length unless it encounters a wall on the edges of the two-dimensional grid. Example result of the positive indirect algorithm is shown on the left in Figure 20. [Ashlock et al., 2011]

The negative indirect method uses a one-dimensional array of 120 cells to store integers in the range of 0...9999 and uses pairs of these integers to define the dungeon in similar fashion to the positive direct method. The two-dimensional grid XY is filled in the initialization and 4 × 4 rooms are carved for the entrance and the exit. Then corridors and rectangular rooms are carved in it based on the pairs of integers in the array. Example result of the negative indirect algorithm is shown on the right in Figure 20. [Ashlock et al., 2011]

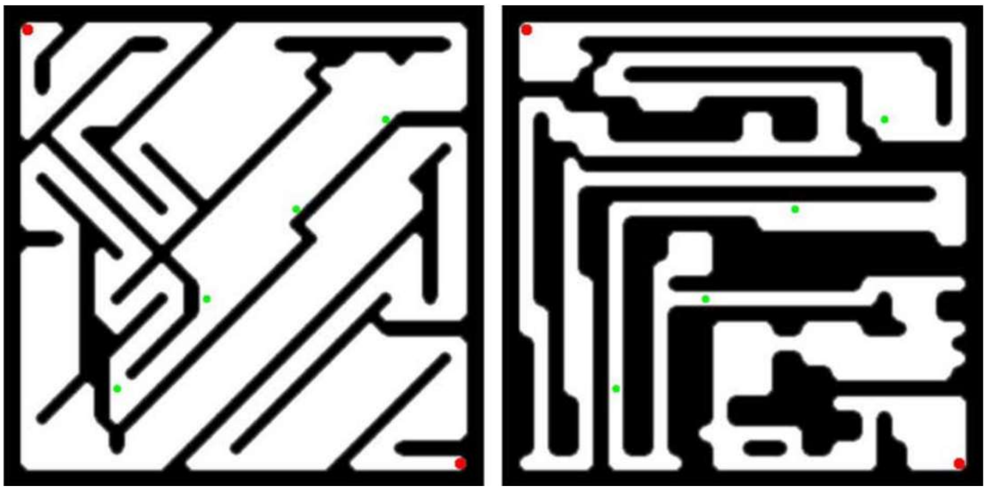


Fig 20. An example of the positive indirect (left) and the negative indirect (right) [Ashlock et al., 2011].

Five different fitness functions are also presented and explained by Linden et al. [2014]:

- F1 maximizess path length from entrance to exit.
- F2 maximizess accessibility to all checkpoints.
- F3 encourages paths to branch and run over checkpoints and meet up again later.
- F4 maximizess amount of dead ends.
- F5 maximizess length of dead ends.

The first binary representation is clearly superior at generating interesting and natural like dungeons. The second direct representation (chromatic) on the other hand is very different compared to the other three. While the other representations determine which squares are obstructed, the chromatic representation does not directly obstruct any square. The chromatic representation instead sets rules, which determine movement between squares, meaning a player can move from one square to the next if they are the same color or are adjanced in the set presented earlier. This means that the chromatic representation defines the maze by implicitly creating walls between squares, instead of defining it by obstructing squares. Minor shortcoming of these algorithms (at least direct 1, positive indirect, and negative indirect) is that the entrance and the exit are always in opposite corners. This makes the dungeon somewhat predictable as the player would know which direction to gravitate to. [Ashlock et al., 2011]

5.2.2 Search-based generation 2

Connor et al. [2018] presents another evolutionary search-based method. A generic level is an $m \times n$ grid of cells and similarly to the method presented by Ashlock et al. [2011]. The start and the exit are set in opposite corners of the grid, bottom left cell being the start and the exit being top right cell. Cells are either walls or void that merge into rooms and corridors. The algorithm uses "falling rectangles" to initialize the grids, instead of a cell based representation, which initializes the cells in the grid independently as a wall or a traversable space. It means to place a number of rectangular shapes into the grid randomly. This was implemented by assigning control parameters. The algorithm tries to fill 30% of the grid, meaning approximately 70% should be traversable space, which should result in a good and playable levels. This goal is defined as the desired space ratio, *R*. The following equation calculates the number of rectangles used (n_r) [Connor et al., 2018]:

$$n_r = \left(\frac{n_c \times R}{a_p}\right) \times (1 + O) \tag{13}$$

where

n_c is the total number of cells in the level determined by the width and height,

O is the overlap factor, which is a simple adjustment that takes into account potentially overlapping tiles, and

a_p is the projected average area of rectangles measured in cells that is determined from the upper and lower bounds on the sizes of the generated rectangles.

Each cell's state is defined during generation of the level to be either a traversable space or a wall and in each future operation the cell's state is manipulated. This process is repeated to generate initial individuals to create the initial population. In this method, chromosome encoding is generated by changing the two-dimensional grid into a linear array of bits by reading the grid from the bottom row to the top row left to right. Each bit on this linear array corresponds to one cell in the two-dimensional grid. In this representation a wall is signified with a bit value 1 and traversable space with a bit value of 0. This operation is shown in Figure 21. [Connor et al., 2018]

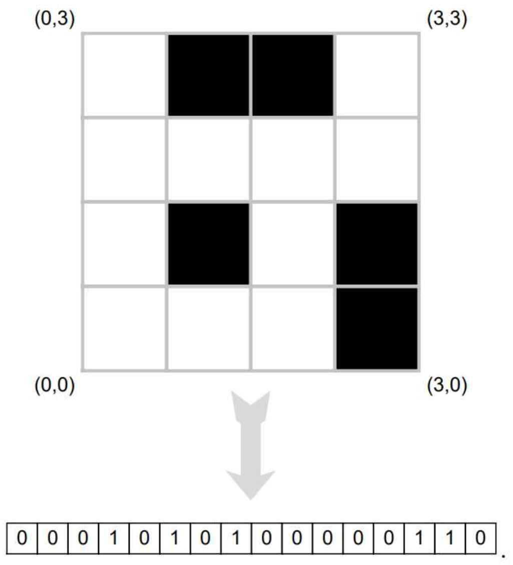


Figure 21. Chromosome encoding for 4×4 grid [Connor et al., 2018].

Furthermore, the selection is done by a roulette wheel selection method. In the roulette wheel selection method the solutions are ranked by their calculated fitness value. In addition, a cumulative probability of selection is added based on how much the individual solution contributes to the gross fitness value of the generation. Two solutions are selected for a crossover by comparing a randomly generated number to the selection probability. A simple single point crossover is used for the crossover. A point of the crossover is selected randomly by comparing a probability against a fixed value for each individual cell. The crossover point determines at which index parents are split into two new linear arrays of bits. In a crossover two parents are split and then these splits are recombined across to generate two new children. For example the bit arrays 111 and 000 would result in two children 110 and 001 when the crossover point is 1. Figure 22 demonstrates this process further. [Connor et al., 2018]

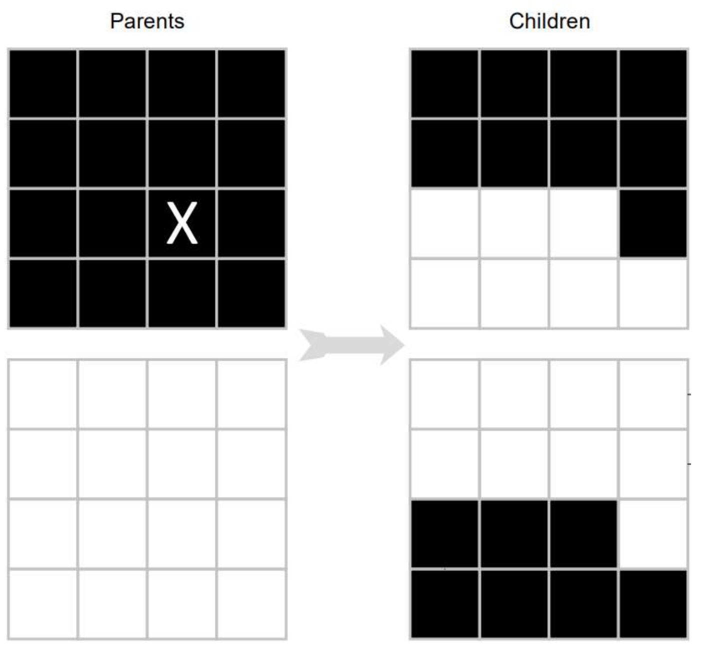


Figure 22. Single point crossover [Connor et al., 2018].

The mutation process uses two types of mutation: a standard mutation, which flips bit values of cells randomly, and a contiguous mutation. Mutation types are used in that order as a mutation strategy. The contiguous mutation selects a random traversable space in the solution and mutates the boundary of that space. This makes traversable spaces to either become larger or smaller in a mutation operation. This will result in a possibility that two separate rooms become connected or that one room to become split in to two new separate rooms, or dig corridors possibly connecting two far away rooms over multiple mutations. First, using a very low probability of a mutation each cell is tested for a standard mutation. In that case that standard mutation does not take place, a contiguous mutation is used. The contiguous mutation tries to find a traversable space by selecting a random cell in the solution. If the randomly selected cell is a wall, it selects a new cell. This process is repeated until it finds a cell that is a void. The contiguous mutation operation can perform two different operations on a traversable space. These operations either grow or shrink the chosen traversable space. The mutation operation is chosen completely randomly. The shrink operation tests each void cell on the edge of the chosen traversable space for the mutation. If a mutation occurs, that cell becomes a wall. This operation is shown in top part of the Figure 23 where the initial mutation site is shown with an X, and the set of cells available for mutation are shown with diagonal lines. In this case, cell Y is selected for mutation, hence becoming a wall. The grow operation is shown in lower part of Figure 23 and works the opposite way of the shrink operation. The grow operation tests each wall cell on the edge of the chosen traversable space for mutation. If a mutation occurs, that cell becomes a wall. These mutation operations can occur multiple times as each of the cells in the mutation set is tested for a mutation. Since random cell mutations occur less likely, the existing walls and traversable spaces can grow or shrink relatively fast. [Connor et al., 2018]

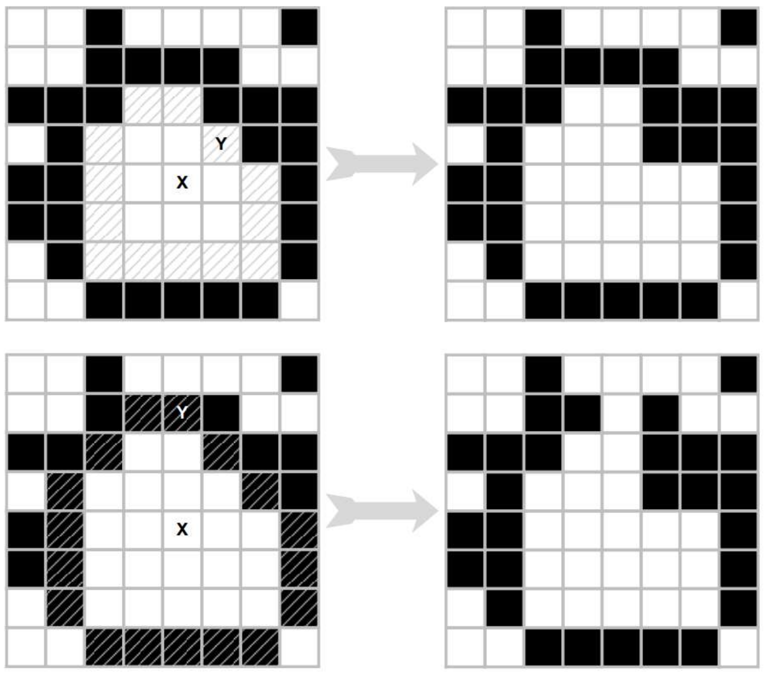


Figure 23. The shrink mutation operation (top) and the grow mutation operation (bottom) [Connor et al., 2018].

The fitness function evaluates the ratio of traversable space to size of the level. Also for playability reasons, there should exist at least a single path which connects the entrance and the exit cells. The space ratio is calculated by dividing the number of cells in the largest continuous traversable space that is accessible from the entrance cell with the total number of cells in the level. In addition, the space ratio can also be compared to the desired ratio of traversable space. This gives an error value which shows how far the solution is from the desired target solution. The actual space ratio (r_a) of the level and the space ratio error (R_e) is given by [Connor et al., 2018]:

$$r_a = \frac{n_{rc}}{n_c} \tag{14}$$

where

n_c is the total number of cells in the level, and

n_{rc} is the number of reachable cells in the largest contiguous space.

$$R_e = |R - r_a|. \tag{15}$$

Fitness function calculation also includes two other contributing factors. These factors are used to determine how close the entrance and the exit cells are from connecting by calculating the ratio between distance of the entrance and the exit cell and the proximity of the entrance or the exit cell to the largest traversable space. For example for the entrance cell, it calculates the ratio between the distance to the closest void cell in the largest traversable space and the distance to the exit cell. For the exit cell the comparison is inversed as it calculates the ratio between the distance to largest traversable space and the distance to the entrance cell. The proximity of the exit and the entrance are determined by the following distance ratios (example shown in Figure 24) [Connor et al., 2018]:

$$P_{exit} = \frac{CD}{AD} \tag{16}$$

$$P_{entry} = \frac{AB}{AD} \tag{17}$$

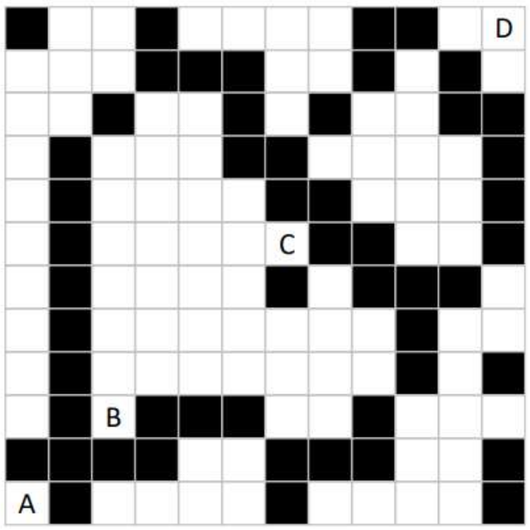


Figure 24. Proximity of largest traversable space to the entrance and the exit [Connor et al., 2018].

In the case that the entrance or the exit cell is connected to the largest continuous traversable space, the corresponding terms drop to zero. A fitness function is created by combining these two terms with the space ratio [Connor et al., 2018]:

$$Fitness = (1 + P_{exit})^2 \times (1 + P_{entry})^2 \times (1 + R_e)^2. \tag{18}$$

Theoretical minimum for the fitness function is 1 in the case that both entrance and exit cells are connected by the same traversable space which takes 70% of the whole level space. Example of the final result is shown in Figure 25. [Connor et al., 2018]

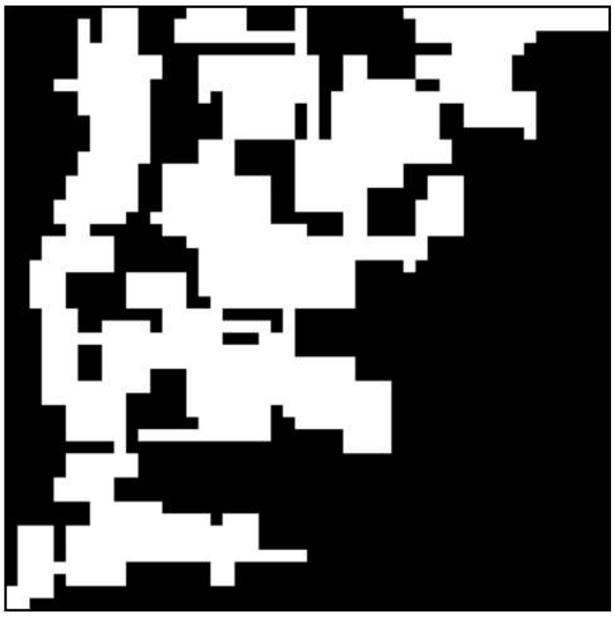


Figure 25. An example of the final result [Connor et al., 2018].

5.3 Constraint-based method

Roden and Parberry [2004] present a constraint-based method that stores the map data in three-dimensional undirected graph. Positions of the rooms are determined by the nodes, and the graph defines connections between the rooms. This first step in the level generation can be customized with parameters, which includes a set of various constraints for the nodes and the topology (such as a tree, a ring and a star). Additionally, hybrid topologies can be formed by connecting multiple sub-graphs with different topologies. Constraints used can be connected to one or more fixed nodes. These constraints include parametrized rules which can for example be used to determine what pre-fabricated geometries (later called prefabs in this chapter) are used to render the node, fixed connections to adjacent nodes and the minimum/maximum distance from the terminal nodes (entrance/exits). In addition, constraints can be used to force the player to visit a set of nodes in a specific order. The second step in the generation process forms basic geometry for the nodes. First, a number of prefabs are build by using a modeling program. These prefabs can be interconnected to create a larger level. Final geometry is build by selecting and combining the suitable set of prefabs. These geometries are then stored in a list which is used to select geometries for the nodes. This allows nodes to share geometries. Also, at this point, pre-set constraints that force nodes to use specific geometries from pre-selected set of prefabs are applied. This for example allows a designer to insert special rooms to the level giving the designer a greater control over the generation process. [Roden & Parberry, 2004]

This method's strongest quality is its ability to generate dungeon mazes in three-dimensional space (as shown in Figure 26) which adds to the complexity of the dungeon, and thus makes it more interesting to explore. Level of customization is also high as it allows the designer to affect shape of the dungeon, shape and content of the nodes and tunnels, and even add special rooms to the dungeon. The method could also be improved by generating the geometry of nodes completely procedurally, instead of selecting from pre-made prefabs, or by generating new nodes during gameplay. This could be done for example by generating a new node when the player is set number of nodes away from the terminal node. Exit nodes could be generated online after the player has explored certain number of nodes. These numbers could be parametrized to give the designer even greater control over the dungeon generation. Dungeon generation could also be modified to generate maps in 2D space by using two-dimensional undirected graphs.

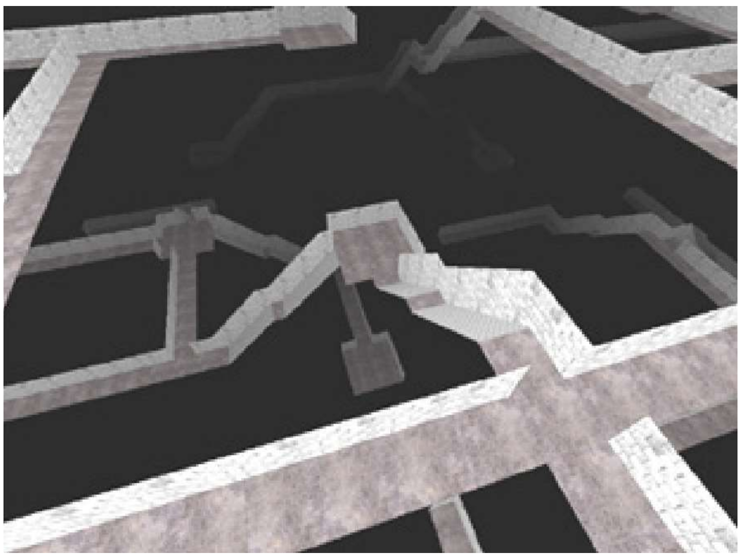


Figure 26. Dungeon generated with 3D undirected graph [Roden & Parberry, 2004].

5.4 Tiny Keep Method

Tiny Keep by Phigames is an intriguing example, since it combines various techniques to generate the dungeon. The procedure is explained by its developer in a reddit post [TinyKeep, 2013] and further explained in [A Adonaac, 2015] with some example code. The procedure consists of these main steps which are also visualized in Figure 27:

1. Generate N rectangles, where N is determined as a parameter. Rectangles are given random widths and heights using *Park-Miller Normal Distribution* (see, e.g., [Park & Miller, 1988]). This will cause the majority of the rooms to be small to medium size with some larger rooms. In addition, the ratio between

widths and heights is limited to a certain range so that the rooms will not be too skinny. Then rooms are randomly distributed within a random radius from the center point.

2. After this, a lot of rooms will be overlapping. To spread out rooms, *separation steering behaviour* [Reynolds, 1987] algorithm is used and any space between rooms is filled with 1 × 1 sized cells.
3. Actual rooms are determined by selecting rectangles above certain size threshold.
4. Rooms are linked by constructing a graph of all rooms center points using *Delaunay Triangulation* (see, e.g., [Aurenhammer, 1991]).
5. Use the graph to construct *minimal spanning tree*, some edges are re-incorporated to generate some loops in the dungeon, the amount of these edges can be set as parameter, e.g., 15%. This will ensure that all rooms are connected and also prevent the dungeon from being too linear and hence it will be more complex and interesting to explore.
6. To convert the graph to corridors, for each edge a series of straight or L shaped lines going from each room of the graph to its neighbours is constructed. Any cells which intersect with the lines become corridor tiles.

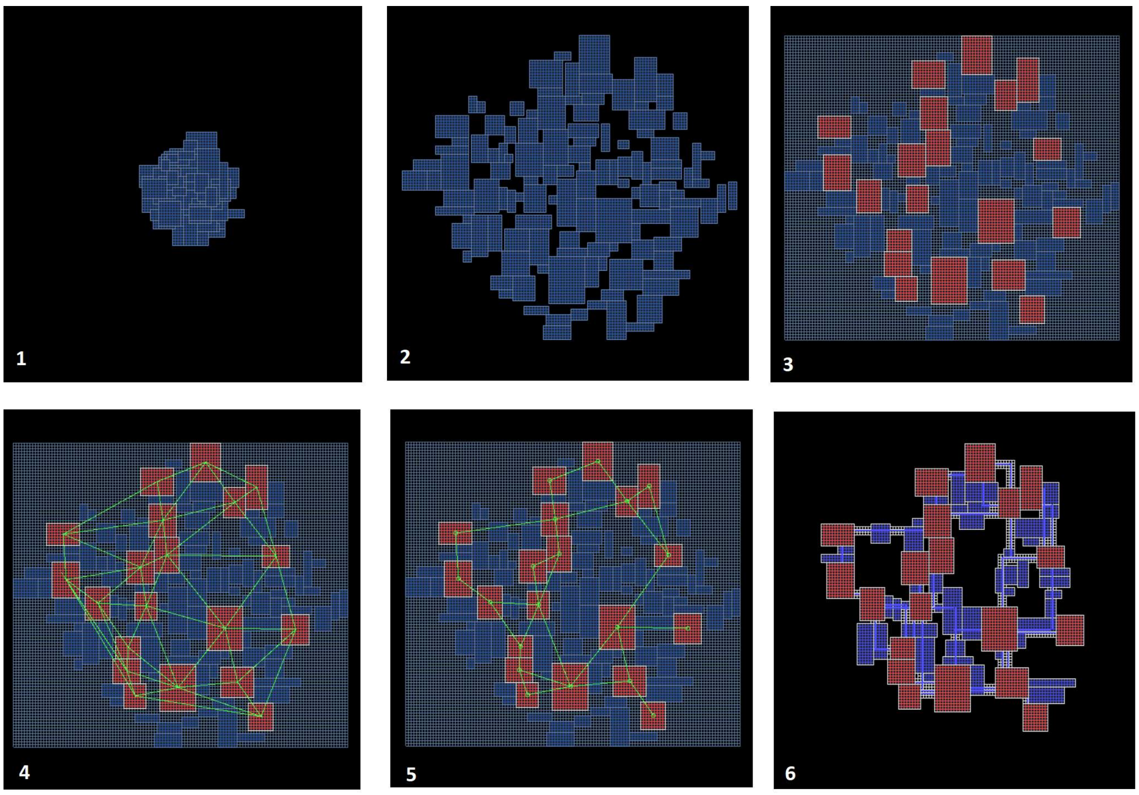


Figure 27. Visualization of the Tiny Keep dungeon generation steps [A Adonaac, 2015].

6 Evaluation metrics

When creating new algorithms or using existing ones to generate terrain, it is sensible to evaluate the algorithm and the results. Santamaría-Ibirika et al. [2014] present and describe an extensive set of metrics for evaluation:

- *Innovation*, the ability of the algorithm to generate unexpected results.
  - 0, there is no innovation, meaning the results are predictable.
  - 1, the algorithm generates same elements in all results, but with different properties.
  - 2, the results have unexpected elements.
- *Structure*, the ability of the algorithm to generate recognizable elements.
  - 0, the results are similar to the random noise.
  - 1, the results have independent recognizable elements.
  - 2, the results have recognizable coupled elements.
- *Interest*, the ability of the algorithm to generate interesting results to interact with.
  - 0, the results are not explorable.
  - 1, the terrains are explorable.
  - 2, the terrains motivates to explore.
- *Speed* of the algorithm to generate results.
  - 0, the results are generated offline (pre-generated).
  - 1, the algorithm is a hybrid where for example, the algorithm can be executed during loading screen.
  - 2, the algorithm can be executed online during gameplay.
- *Usability*, the ability of the algorithm to hide technical aspects from the designer.
  - 0, there are no input parameters.
  - 1, the parameters are related to the technique.
  - 2, the parameters are related to the end results.
- *Control*, the ability of the algorithm to allow the designer to define results' features.
  - 0, the designer cannot define any aspects of the result.
  - 1, the designer can define some aspects of the result.
  - 2, the designer can define all features.
  - 3, in addition to 2, the designer can correct results iteratively.
- *Ampliability*, the ability of the algorithm to generate different types of results.
  - 0, it is impossible to generate other types of results.

  - 1, the algorithm requires internal changes to generate different types of results.
  - 2, it is possible to generate different types of results by changing input values.
- *Scalability*, the ability of the algorithm to generate results of different levels of detail and scales.
  - 0, the results are always the same scale.
  - 1, the algorithm can generate results in different levels of detail and scales.
- *Realism*, the ability of the algorithm to generate realistic results.
  - 0, the results are completely chaotic and impossible in any reality (e.g. random noise).
  - 1, the results are plausible in some realities (e.g., ring worlds, floating isles).
  - 2, the results are based on our reality.
  - 3, the results are simulation of our reality.

All in all, these metrics cover all important qualitative aspects of terrain generation. Some metrics could benefit of better or less subjective definitions, or having smaller or larger range of evaluation. For example, *Interest* could just be a binary value that expresses if the resulting terrain is explorable or not. *Usability* could also measure the level of abstraction of the parameters (e.g., some parameters are abstracted, all parameters are abstracted). In addition, other potential metrics to evaluate could be the computational (time) complexity of the algorithm and its memory requirements. Memory requirement metric could be a separate metric or combined with the *speed* metric, since they are connected: Offline (pre-generated) terrain requires larger memory overhead than online generated. In the case of dungeon algorithms, *playability* could be an additional metric to evaluate, which would express if a dungeon is actually playable or not. *Playability* could be a binary metric to tell if there is a guaranteed path from the entrance to the exit. 0 meaning there is no path, and 1 meaning there is a path.

Next, terrain generation algorithms introduced in this thesis are evaluated using the metrics introduced earlier in this chapter. The results of evaluation can be seen in Table 1. Noise functions generally are capable of online terrain generation as can be seen from the *speed* metrics of the noise functions. Wavelet noise being exception to this, since it uses pre-generated base noise. This also means it is comparable to value noise since the base noise used is random noise. Noticeable is that Perlin noise and Simplex noise are well suited for terrain generation as they have high values in almost all metrics that asses generated terrain's quality. Worley noise has low metrics throughout meaning it is not very well suited for terrain generation, except for some specific use cases such as rocky or volcanic terrains. A diamond-square fractal algorithm in general is similar to

Perlin noise and Simplex noise metrics wise. Even though it is technically suited for online terrain generation, its *speed* value is 1 due to the fact, that whole data grids have to be generated at a time and additional logic is required to connect these grids to the existing terrain. Volumetric terrain generation algorithms' *usability* and *control* metrics stand up due to the high level of parametrization and abstraction of the terrain generation process, since the designer can define multitude of various parameters for materials, caves, veins and the whole terrain. For dungeon generators, interesting is the *speed* metric of the cellular automaton, since it is capable of generating a dungeon online, unlike other dungeon generation algorithms. Noteworthy is also the cellular automaton's inability to guarantee connection between rooms, which is not the case with the search-based direct binary, search-based 2, constraint-based and Tinykeep algorithms. Constraint-based algorithm is the most customizable of the dungeon algorithms as it allows a designer to affect the shape of the dungeon, shape and content of the nodes and tunnels, and even add special rooms to the dungeon.

	Innovation	Structure	Interest	Speed	Usability	Control	Ampliability	Scalability	Realism
Value noise	0	0	1	2	0	0	0	0	0
Perlin noise	2	2	2	2	1	1	2	1	3
Simplex noise	2	2	2	2	1	1	2	1	3
Wavelet noise	0	0	1	0	0	0	0	1	0
Worley noise	1	1	1	1	1	0	0	1	2
Diamond-Square	2	2	2	1	1	1	2	1	3
Volumetric	2	2	2	2	2	2	2	1	2
Cellular automaton	2	1	1	2	1	1	2	0	3
Search-based 1 (direct binary)	2	2	2	0	1	1	2	0	3
Search-based 2	2	2	2	0	1	1	2	0	2
Constraint-based	2	1	2	0	2	2	2	0	2
TinyKeep	1	2	2	0	1	1	2	0	2

Table 1. Evaluation of algorithms introduced in this thesis.

7 Conclusion

There is a multitude of different algorithmic approaches to generate terrain or game areas for video games. The first ones being over 30 years old; such as Beneath Apple Manor (published in 1978) and Rogue (published in 1980). Since then terrain generation algorithms have improved a long way, being able to generate detailed and whole universes like in No Man's Sky. It seems that there are almost infinite possibilities in terrain generation, since different ideas, methods and approaches can be combined to create new algorithms.

Noise functions have been used in many games to generate terrain even though originally they were designed for computer graphics and texturing. Perlin noise and it's successor Simplex noise which fixes and improves various aspects of the original noise, are perhaps the most well-known implementations. In this thesis various noise functions were introduced and their usability in terrain generation was evaluated from terrain generation standpoint. Noise functions can be used to generate variety of realistic looking terrains and surfaces. In addition to two- and three-dimensional terrain generation noise functions could possibly have applications in generating dungeon or board game like game areas.

Different kinds of dungeon algorithms were also introduced and evaluated in this thesis. Dungeon generation algorithms are the oldest form of terrain generation in computer games, which make them interesting also in the sense of history of video games, computation and terrain generation algorithms. Dungeon algorithms also show that many different techniques and methods can be applied to terrain generation from cellular automatas to genetic algorithms. Noise implementations in dungeon generation were not introduced in this thesis, but as speculated above there could be possible uses for noise functions as well.

The algorithms introduced and evaluated cover multiple methods and techniques used in terrain generation, but rather than being a comprehensive list, they are merely a glimpse into all the possibilities. Scope of this thesis also forced a number of algorithms to be excluded, which have or could have intriguing applications in the realm of terrain generation. Evaluation metrics introduced are comprehensive and give a good basis for further development and evaluation of terrain generation algorithms.

8 References

[A Adonaac, 2015] A Adonaac. 2015. Article describing TinyKeep terrain generation process with code examples. *https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm*.

[Allain, 2019] Allain, Alex. 2019. An image of procedural terrain texturing. Accessed 24.2.2023. *https://www.cprogramming.com/discussionarticles/texture_generation.html.*

[Anjyo, 1988] Anjyo, Ken-Ichi. 1988. *A simple spectral approach to stochastic modeling for natural objects*. In Eurographics. Proceedings of the European Graphics Conference and Exhibition (Nice, France), p. 285– 296.

[Ashlock et al., 2006] Ashlock, Daniel; Manikas, T; Ashenayi, K. 2006. *Evolving a diverse collection of robot path planning problems*. IEEE Congress on Evolutionary Computation, pp. 1837-1844.

[Ashlock et al., 2011] Ashlock, Daniel; Lee, Colin; McGuinness, Cameron. 2011. *Search-based procedural generation of maze-like levels*. IEEE Transactions on Computational Intelligence and AI in games, vol. 3, no. 3, p. 260-273.

[Aurenhammer, 1991] Aurenhammer, Franz. 1991. *Voronoi diagrams—a survey of a fundamental geometric data structure*. ACM Computing Surveys, vol. 23, no. 3, p. 345–405.

[Bayne, 2021] Bayne, Aaron. 2021. An image of Minecraft video game. Accessed 24.2.2023. *https://www.ggrecon.com/guides/best-minecraft-shaders/.*

[Bracewell, 1999] Bracewell, Ronald N. 1999. *The Fourier Transform and its Applications* (3rd edition). McGraw-Hill.

[Bridson et al., 2007] Bridson, Robert; Houriham, Jim; Nordenstam, Marcus. 2007. *Curl noise for procedural fluid flow*. ACM Transactions on Graphics, vol. 26, no. 3, art. 46.

[Cook & Derose, 2005] Cook, Robert; DeRose, Tony. 2005. *Wavelet noise*. ACM Transactions on Graphics,, vol. 24, no. 3, p. 803–811.

[Connor et al., 2018] Connor, A.M; Greig, T.J; Kruse, J. 2018 . *Evolutionary generation of game levels*. EAI Endorsed Transactions on Creative Technologies.

[Conditt, 2021] Conditt, Jessica. 2021. An image of Terraria video game. Accessed 24.2.2023. *https://www.engadget.com/terraria-google-stadia-indie-what-happened-190001711.html.*

[Ewin, 2015] Ewin, Christopher. 2015. An image of the diamond-square algorithm steps. Accessed 24.2.2023. *https://upload.wikimedia.org/wikipedia/commons/b/bf/Diamond_Square.svg*. License https://creativecommons.org/licenses/by-sa/4.0/deed.en.

[Fourier et al., 1982] Fournier, Alain; Fussell, Don; Carpenter, Loren. 1982. *Computer rendering of stochastic models*. Communications of the ACM, vol. 25, no. 6, p. 371–384.

[Garcia, 2020] Andre A. Garzia. 2020. *Roguelike Development with JavaScript*. Apress.

[Hyttinen, 2017] Hyttinen, Tuomo. 2017. *Terrain synthesis using noise*. M.Sc. Thesis, Faculty of Natural Sciences, University of Tampere.

[Hyttinen et al., 2017] Hyttinen, Tuomo; Mäkinen, Erkki; Poranen, Timo. 2017. *Terrain synthesis using noise*. Proceedings of the 21st International Academic Mindtrek Conference, p.17-25.

[Johnson et al., 2010] Johnson, Lawrence; Yannakakis, Georgios; Togelius, Julian. 2010. *Cellular automata for real-time generation of infinite cave levels*. In Proceedings of the Workshop Procedural Content Generation in Games, p. 1-4.

[Kensler et al., 2008] Kensker, Andrew; Knoll, Aaron; Shirley, Peter. 2008. *Better Gradient Noise*. Tech. Rep. UUSCI-2008-001, SCI Institute, University of Utah.

[Lagae et al., 2009] Lagae, Ares; Lefebvre, Sylvain; Drettakis, George; Dutré, Philip. 2009. *Procedural Noise using Sparse Gabor Convolution*. Proceedings of ACM SIGGRAPH, vol. 28, no. 3, p. 54-64.

[Lagae et al., 2010] Lagae, A; Lefebvre, S; Cook, R; DeRose, T; Drettakis, G; Ebert, D.S; Lewis, J.P; Perlin, K; Zwicker, M. 2010. *A Survey of Procedural Noise Functions*. Computer Graphics Forum, vol. 29, no. 8, p. 2579- 2600.

[Lewis, 1984] Lewis, John Peter. 1984. *Texture synthesis for digital painting*. In Computer Graphics, vol. 18, no. 3, pp. 245–252.

[Linden et al., 2014] van der Linden, Roland; Lopes, Ricardo; Bidarra, Rafael. 2014. *Procedural Generation of Dungeons*. IEEE transactions on computational intelligence and AI in games, vol. 6, no. 1, p.78-89.

[Miller, 1986] Miller, Gavin S. P. 1986. *The definition and rendering of terrain maps*. Computer Graphics, vol. 20, no. 4, p. 39–48.

[Papoulis & Pillai 2002] Papoulis, Athanasios; Pillai, S. Unnikrishna. 2002. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill.

[Park & Miller, 1988] Park, Stephen K; Miller, Keith W. 1988. *Random Number Generators: Good Ones Are Hard To Find*. Communications of the ACM, vol. 31, no. 10, p. 1192–1201.

[Perlin, 1985] Perlin, Ken. 1985. *An image synthesizer*. Proceedings of ACM SIGGRAPH, vol. 19, no. 3, pp. 287–296.

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

[Perlin & Neyret, 2001] Perlin, Ken; Neyret, Fabrice. 2001. *Flow noise*. 28th International Conference on Computer Graphics and Interactive Techniques, p. 187.

[Phreneticc & Wxs, 2022] Phreneticc; Wxs. 2022. An image of Koch's snowflake. Accessed 24.2.2023. *https://upload.wikimedia.org/wikipedia/commons/d/d9/KochFlake.svg*. License https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[Reynolds, 1987] Reynolds, Craig W. 1987. *Flocks, herds and schools: A distributed behavioral model*. Computer Graphics, vol. 21, no. 4, p. 25–34.

[Robinson, 2021] Robinson, Joe. 2021. An image of Civilization 6 video game. Accessed 24.2.2023. *https://www.pcgamesn.com/civilization-vi/mod-hills*.

[Rocchini, 2012] Rocchini. 2012. An image of Worley noise. Accessed 24.2.2023. *https://upload.wikimedia.org/wikipedia/commons/0/00/Worley.jpg*. License https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[Roden & Parberry, 2004] Roden, Timothy; Parberry, Ian. 2004. *From artistry to automation: A structured methodology for procedural content creation.* Entertainment Computing, pp. 151–156.

[Santamaría-Ibirika et al., 2014] Santamaría-Ibirika, Aitor; Cantero, Xabier; Salazar, Mikel; Devesa, Jaime; Santos, Igor; Huerta, Sergio; Bringas, Pablo G. 2014. *Procedural approach to volumetric terrain generation.* Visual Computer, vol. 30, no. 9, p. 997-1007.

[Saupe, 1988] Saupe, Dietmar. 1988. *Algorithms for random fractals*. In The Science of Fractal Images, Springer-Verlag, p. 71–136.

[Smed & Hakonen, 2017] Smed, Jouni; Hakonen, Harri. 2017. *Algorithms and Networking for Computer Games*. Wiley.

[Stolfi, 2013] Stolfi, Jorge. 2013. An image of white noise. Accessed 24.2.2023. *https://upload.wikimedia.org/wikipedia/commons/f/f6/White-noise-mv255-240x180.png*. License https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[Thedarkb, 2021] Thedarkb. 2021. An image of Rogue video game. Accessed 24.2.2023. *https://upload.wikimedia.org/wikipedia/commons/0/0c/Rogue_Screenshot.png*. License https://creativecommons.org/licenses/by-sa/4.0/deed.en.

[TinyKeep, 2013] Reddit post where developer describes TinyKeep dungeon generation process. *https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/*.

[Wijk, 1991] van Wijk J. J. 1991. *Spot noise texture synthesis for data visualization*. Computer Graphics, vol. 25, no. 4, p. 309– 318.

[Voss, 1988] Voss, Richard F. 1988. *Fractals in nature: from characterization to simulation*. In The Science of Fractal Images. Springer-Verlag, p. 21–70.

[Worley, 1996] Worley, Steven. 1996. *A cellular texture basis function*. In Proceedings of ACM SIGGRAPH, p. 291–294.

[Wyvill & Novins, 1999] Wyvill, Geoff; Novin, Kevin. 1999. *Filtered noise and the fourth dimension*. In Proceedings of the ACM SIGGRAPH Conference Abstracts and Applications, p. 242.