Revista Romana de Interactiune Om-Calculator 9 (2) 2016, 119-142 © MatrixRom

TSCH Algorithm - Terrain Synthesis from Crude Heightmaps

Alexandre Philippe Mangra, Adrian Sabou, Dorian Gorgan

Computer Science Department, Technical University of Cluj-Napoca
Str. G. Baritiu, Nr.28, 400027, Cluj-Napoca
E-mail: alexandre.mangra@gmail.com, {adrian.sabou, dorian.gorgan}@cs.utcluj.ro

Abstract. This paper presents an approach to terrain synthesis from minimal-detail user-provided heightmaps. There is no assumption regarding the level of detail provided, in order to allow users without access to powerful heightmap tools and/or resources to generate useable terrain based on a self-provided crude feature plan. We present the issues stemming from a lack of detail in user input, notably sharp altitude increases and oversimplified feature edges, and proceed to elaborate on using the terrain synthesis algorithm to solve the issues and create a level of detail that more closely resembles realistic terrain models. The algorithm pipeline is presented and parameterized to show how the user can influence the resulting model.

Keywords: Terrain synthesis; Heightmap; Worley noise; Perlin noise; Filters.

1. Introduction

Over the past decades, computational power has become less expensive and more powerful thanks to technological advances. Alongside it, computer generated imagery (CGI) increased in availability and potential. CGI is one of the mainstays of technology, used in various fields like video games, movies, art, simulation software and anywhere else image generation is beneficial. One of the reasons for its popularity is the artistic freedom it entails, coupled with the potential to mimic something that does not exist in the real world.

When compared to physical props and background, computer generated imagery becomes evidently advantageous. There is a large number of images unable to be accurately reproduced without the use of computers, be it spaceships, hellish creatures, otherworldly plants or simply vast, expanding landscapes. Creating a quality physical replica would incur costs unreasonable for any budget, not to mention unfeasible if we're considering the entire landscape of an alien planet. A virtual reproduction's costs can easily be quantified in the artist's and/or programmer's work and the

required hardware.

Thus, the industry's needs have fueled the development of an array of algorithms and generation software tailored specifically at creating this kind of models. Among them, terrain generation is one of the most used fields due to it contributing significantly at reducing production costs of backgrounds.

The terrain model can be created procedurally (using a set of rules) or based on a set of given input data. The latter is usually combined further with algorithms to refine the given data and produce something usable. Purely procedural terrain suffers from restricting the user control over the final location of terrain features like mountains, hills, plains, rivers or islands. On the other side of the spectrum, some synthesis algorithms working with input data such as heightmaps, feature graphs or guides require at least part of the data to be highly specific. This can prove inconvenient to the casual user, forcing him to spend increasing amounts of time researching the software used and finding ways of creating the necessary input.

The casual user, then, raises new issues when trying to create terrain with specific features. He will, in most cases, be unable to properly form input data relevant for the application that would lead up to the desired terrain model. It can become tedious and time-consuming to master a new skill or software in order to obtain decent results.

One issue will be the sudden altitude increases caused by the user creating the input heightmap by hand. Painting the heightmap with only a handful of colors or grayscale values leads to the creation of a layered terrain which does not conform to reality nor has any kind of transition between layers, hence the sharp, perfectly vertical, altitude changes.

Another issue is the lack of detail on such models. The layman will have neither the time nor experience to paint "rough" edges, as seen in nature at the delimitation of two differently elevated areas. There is a high probability of encountering very uniform edges, if not downright straight, thus breaking the illusion of natural, chaotic, form.

This paper elaborates on a simple algorithm which tries to solve these issues by detailing very crude input to a point where it becomes usable, either as the final terrain model or as a more precise input heightmap for more complex algorithms. It also reflects the extended content of another paper (Mangra et al., 2016), published by the RoCHI2016 conference.

2. Related Works

The high demand makes it such that a variety of techniques for procedural or user-guided generation already exists. As information becomes increasingly available, more and more people try to expand the horizons by either improving existing methods or finding new ones. Terrain synthesis is one of those expanding domains, with entire companies being built around terrain generation software and a growing number of research papers detailing new algorithms.

One such example is World Machine Software, LLC and their sole product: World Machine (Discover, 2016). An immensely powerful terrain generation software which allows users to create terrain from scratch by layering algorithms and directing data through the pipeline they create as they see fit. It also supports user-guided generation, by allowing the input for algorithms to be provided through external files. At first glance, however, it does not offer ways to process low-detail input. Elevation discrepancies remain sorely visible throughout the processing pipeline. A person trying to control the features will be unable to do so unless he or she invests enough time in learning how to use this complex software. If such procedures exist, they are unintuitive at best.

A case should be made for Gaia (Gaia, 2015), procedural terrain software created by Procedural Worlds, which, among other capabilities, allows users to define where they want certain features to be placed by inserting specialized markers called “stamps”. This greatly alleviates the input issues but restricts the user to the set of available stamps (currently over 150) as an advantageous tradeoff between control and power. The only downside is its reliance to the Unity game engine since it is provided as a Unity “asset”, a plug-in of sorts.

Aside from terrain synthesis software, the number of papers detailing new and experimental algorithms for synthesis is on the rise. The focus is on giving as much power as possible to the user, creating new ways of synthesizing terrain from different input data-sets. Somewhat unsurprisingly, the tendency is to reach for improved reproduction quality. To give the end-user the power to remake relief forms based on certain patterns and to do so at the best quality level possible.

For example, one of the more well-known papers on the topic is the work of Zhou et al. (2007), describing an algorithm to map a relief style onto a simplistic user-provided sketch. As long as the user finds a heightmap describing the desired relief shape and pattern, he can utilize the algorithm to great results. This only partially solves the issue of low-detail user

guidance. Firstly because only one type of pattern can be applied at a time, preventing, for instance, both a mountain range and a river or lake to be mapped in the same map instance. Secondly, because obtaining such patterns may or may not prove difficult, depending on what the user intends to obtain and the available patterns on the internet. These problems arise, of course, because of the high specialization of the algorithm and are perfectly acceptable in the context of the goal set for this procedure.

Another such work is that of Cruz et al. (2013), with an objective similar to that of Zhou et al.: user-guided terrain synthesis. This paper focuses on having an input graph besides the simplistic sketch, called “guide” here. They try to create geomorphically correct terrain from a collection of real-world data. Very similar in both scope and surfacing issues to the previously presented work: it requires information the layman may not immediately have available and it becomes hard to model several terrain features at once.

In the quest for improving the obtained terrain, most researchers specialize their work, leaving the inexperienced user dead in the water. Even when a software product implements something with general availability, the learning curve is almost never shallow. Large amounts of time must be invested for the average user to obtain usable results from most of today’s software implementations.

3. User-Guided Terrain Synthesis

While procedurally generating terrain has plenty of advantages, such as speed of generation, variety and realistic detailing, the main drawback is the lack of user involvement in the placement of terrain features. This makes it hard to create something specific and which conforms to the user’s requirements.

As described in the previous section, this gave birth to a series of algorithms and software which do exactly that: create terrain based on a set of specific user input data. They give more freedom to affect the end product and allow one to model the shape of the terrain based on their own wishes. Artists and designers gain tremendous power by being able to create terrain in drawing that is then converted to a highly-realistic 3D model. Researchers and other technical-oriented people gain an equal amount of power by being able to convert data obtained from the real world to create incredible virtual replicas.

For the hobbyist, however, or any other inexperienced user the challenge becomes much greater. One has no use for powerful tools if they require

large time investments to master. Furthermore, there is a definite possibility that said user is not interested in highly-detailed or geomorphically correct terrain. The main interest point is the creation of a terrain model simulacrum that abides by the user's requirements. Most of the people interested in terrain synthesis will not be artists, capable of creating detailed heightmaps to provide to the software nor experienced enough to find the other resources needed as input, such as real-world data, formatted in a way which the software expects.

Figure 1a. Example crude highmap.

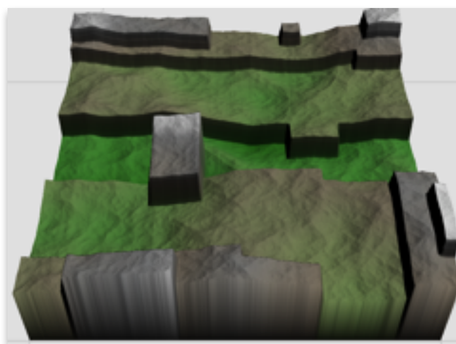

Figure 1b. World Machine 15 minute test.

Figures 1a (the input heightmap) and Figure 1b (the resulting model) illustrate the problem with a hands-on example using the World Machine software. We have created a relatively simple test image and loaded it into the software. It was an attempt to synthesize a terrain model based on a given input which lacks detail. The resulting model is the result of approximately 15 minutes of experimenting. It is painfully obvious that it is not a good result. The model lacks any kind of detail with regard to the edges, which remain almost perfectly straight. All the added detail circumvents these sharp altitude changes and regular shapes and there was no visible way of altering the model in such a way as to make these edges better looking.

Following is the algorithm proposed to solve this problem by interpreting low-detail heightmaps and synthesizing a terrain model that, while not necessarily accurate from a realistic point of view, meets the requirements set by the user through the input heightmap and places the terrain features where they are expected. It is assumed that an input is provided in the form of a crudely-drawn heightmap, lacking detail.

3.1 Related algorithms and concepts

Heightmaps

Heightmaps are exactly what their name suggests: images which store values representing height data encoded in color values. The simplest way of storing this kind of data is through grayscale information. Grayscale is simpler than true color because it only has two poles: black and white, and everything in between. A heightmap also only stores a scale of values, from the lowest point to the highest one. Hence, the coupling is nigh-perfect, limited only by the range of grays the image file can support. A color image could theoretically improve upon the range of values, depending how each color encodes height information. One disadvantage to this kind of data storing is that it can only store height value information. This means no vertically concave terrain can be represented. Heightmaps are essential to this project's existence, since they provide a reliable input option and also a sturdy output configuration possibility. Figure 2 presents such a grayscale heightmap example. To illustrate what information heightmaps contain, Figure 3 presents the same heightmap, rendered in three dimensions based on the heightmap values.

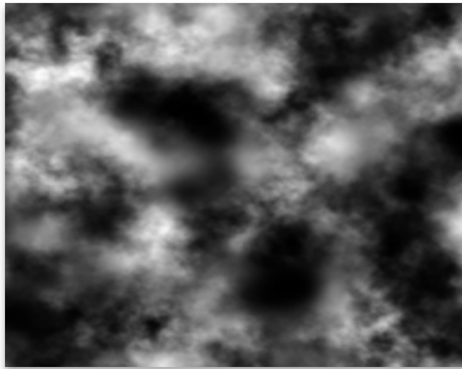

Figure 2. Highmap, greyscale.

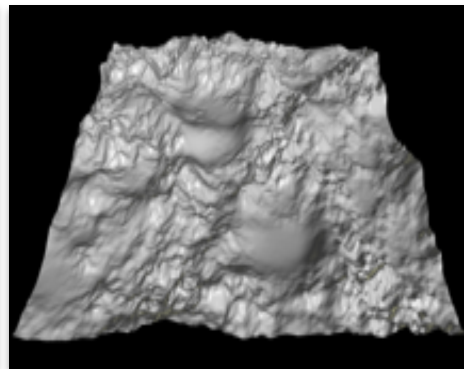

Figure 3. 3D rendered highmap.

Voronoi diagrams

A Voronoi diagram is a distribution of segments in a plane based on a number of seed points distributed across said plane. Every point inside a specific region is closest to the seed point inside it. In other words, a point inside the plane will become part of the region surrounding the closest seed point. They were named after Georgy Voronoi, Russian and Ukrainian

mathematician who defined this type of diagrams (Voronoi, 1908). The first considerations of them were recorded as early as 1644 by Descartes then later by Dirichlet in 1850 in his work „Über die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen" (Dirichlet, 1908). Because of this, the regions compounding the diagram are called Dirichlet regions and the diagrams may be called Dirichlet tessellations. The diagrams have applications in science and technology, with the added capacity of aiding visual artists (Figure 4).

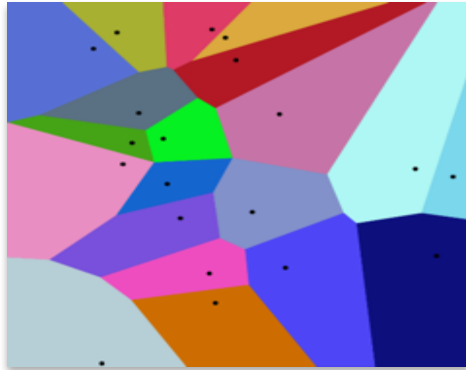

Figure 4. Voronoi diagram.

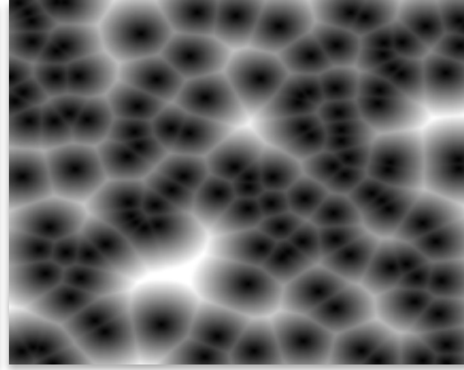

Figure 5. Worley noise.

There are many uses for Voronoi diagrams. They can produce various textures when used in 2D and abstract models in 3D. In 3D they can also be used for realistic fracture simulation animation of models, like glass shattering and wood splintering patterns. They can also be used for 1-NN classifiers in machine learning and finding clear routes in some AI applications, among other interesting uses. In the context of this paper, they are presented in order to introduce the next segment: Worley noise, which has semblances to this kind of diagrams.

Worley noise

Worley noise (Worley, 1996) is a type of cellular noise. This means that the noise takes form of „cells" inside the given plane (Figure 5). Practically, it divides the problem space into regions. This partitioning is based on a scattering of seed points, just like in the Voronoi diagrams. The way these regions are defined is by a function which takes the N-th closest seed point in consideration. What this means is that when N is 1, the cellular noise looks exactly like a Voronoi diagram mapped onto the seed points if the

function is a simple value assignment function. The noise can look and behave very differently depending on what the chosen function does and which N has been selected. This type of noise will prove relevant later during this paper, when the custom interpolation algorithm is discussed.

Perlin noise

Perlin noise was created by Ken Perlin in 1982 for the movie Tron because he was unhappy about how CGI looked at that time (Perlin, 2002), (Adrian's, 2016) and (Perlin, 2016). Essentially, it is a sort of gradient noise used as a primitive in more involved processes. In time, Perlin developed a better version of the noise, called Simplex noise. This improves upon the former by simplifying the generation process and reducing complexity and any leftover visual artifacts.

Unlike the usual kind of noise, Perlin noise is homogenous. This means that any two neighbors have very close values to one another compared to the breadth of the available spectrum. This leads to values that „flow" over the length of the entire area by never having sudden value changes between regions and, more importantly, between neighbors. Figure 3.3 presents a 2D representation of the Perlin noise. Another important fact to mention is that Perlin noise can be extended to any number of dimensions. The algorithm is perfectly suited for 1D, 2D, 3D, 4D, etc. Furthermore, one can use the next dimension to animate the model. For instance, one can model 3D noise to represent a cloud formation and then use the 4D values to chain „snapshots" of that cloud and effectively animate it, the frames of the animation being given by the algorithm itself, guaranteeing smooth transition between one another.

Perlin noise divides the space up in an n-dimensional grid. Then, for each segment, it produces pseudorandom gradient vectors on its corners. This kind of generation allows the noise to be coherent, meaning the transition from a point to its neighbor is greatly reduced. The coherence is usually achieved through linear interpolation of the previously-mentioned vectors. Each segment's value is such an interpolated value, given by dot products of distance vectors to a point randomly chosen inside the segment and the surrounding gradient vectors. Adjacent segments share gradient vectors on the common side.

The first step, that of spatial division, simply creates equally-sized regions which are delimited by gradient vectors. One vector is assigned to each of the corners of the region. Each region will have two corners in one dimension, four corners in two dimensions, eight corners in three

dimensions and so on. Consequently, each corner will have its own pseudorandom gradient having the same number of dimensions as the space being divided.

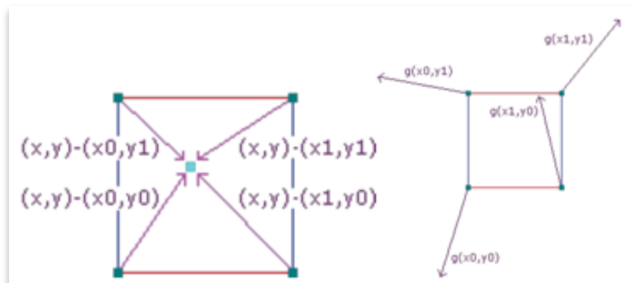

Figure 6. Distance vectors (left) and Gradients (right) for a given two dimensional segment.

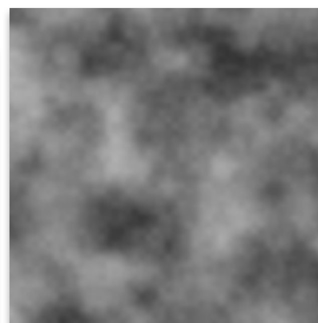

Figure 7. Perlin noise.

This grid of gradients is what the algorithm uses to compute noise. The randomness is given by the vectors themselves. For each point in the plane, the value of the Perlin noise can be computed by relating the give point to the corners of the region the point is in. Figure 6 presents a graphical representation of those relations and the set of gradient vectors g. The point is evaluated as the set of distance vectors originating in the corners and pointing at the given point (x, y).

The value of the Perlin noise at point (x,y) is calculated by interpolating the surrounding gradient vectors using the distance vectors as weights (Figure 7). This means that all points inside the region will be an accumulation of the gradients weighted by the distance of the point to the corners. Because it is an interpolation, the transition of values between points is smooth inside the region. When crossing regions, the transition is also smoothened by having a number of gradient vectors common on the borderline corners of the regions.

The actual interpolation may involve a number of additional steps, like passing the point through another value function to change the way position influences the final value. Furthermore, the interpolation can be done in a number of ways to customize the algorithm or simply to make value handling more efficient. Usually, the values are first interpolated on one plane (horizontal, for instance, in a two-dimensional example) then in the other plane (vertical, in the same example). This is repeated for all

dimensions until one single processed value is left, which will be interpreted as the Perlin noise value of that specific coordinate set. Because of this, the algorithm is also highly parallelizable, being able to compute the noise value for all points in the plane at the same time, given the availability of the entire set of gradient vectors.

Digital filters

Digital filters are algorithms (or hardware implementations) which reduce or amplify signal features. They iterate over the input and change it based on certain restrictions set up from the beginning. In this particular case, the interest lays on noise reduction filters. This type of filters is specially designed to algorithmically remove noise from a give dataset by comparing all values to the neighboring ones, trying to determine where anomalies are present and replacing them with a value considered to be more fitting in that particular area (the comparisons made differ from implementation to implementation).

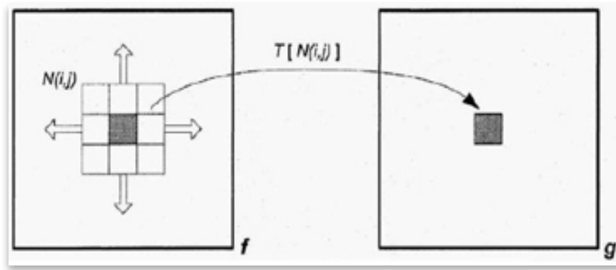

Figure 8. The Sliding Window and the Filter Function.

Figure 9. Sharp transition example.

The image processing filters considered here work by utilizing a sliding window that goes over the entire input, modifying it as configured (Crankshaft, 2016). The size of the window depends on the requirements and implementation specifics, but in this case a three-by-three sliding window moving inside a two-dimensional plane should suffice. This window replaces the value in the center (the fifth value, counted from top-left to bottom-right) with a value chosen through specific algorithmic variations. The window is presented in Figure 8, where N(i,j) represents the set of N points to be considered and T is the filter function which chooses the new value for the grayed-out cell.

The two filters whose use is to be considered in this software

implementation are the mean filter (Fisher, 2003a) and the median filter (Fisher, 2003b). These are both noise-reduction filters which would be used to smoothen out the heightmap outputs. The mean filter works by computing the mean of all the encompassing neighboring cells and assigning it to the center cell. This obviously eliminates spikes by having the other points even out the values. However, this also means that noise does affect the final value, since it is taken into account when computing the mean.

The second filter to be taken into consideration is the median filter. This noise reduction filter works by replacing the value of the middle cell with the median value of all nine cells. The median value is the fifth value taken from the ordered list of values composing the sliding window for a particular point. This approach ignores spikes and the resulting value is not influenced by them. Hence, this filter may prove more useful.

3.2 Edge smoothing

The first step is to solve two issues in the same pass. The issues being:

Sharp elevation level transitions

The neophyte user will provide a heightmap where one elevation level ends and another being with a drastic difference in value/height. Best example would be the user wanting a mountain surrounded by sea and drawing with a high value in an area of very low values. This will cause a vertical drop (value change of 100%) between the value level represented by the "mountain" (white) and the one of the "sea" (black), as can be seen in Figure 9. Going straight from perfect gray to white or black is also not a good use-case, since the value switches by 50% of the total. A lesser but still perfectly vertical drop.

Simplistic edge definitions

The layman will not have the art skills or appropriate resources to paint better edges. He or she will resort to basic straight or curved lines as shape delimiters, also exemplified on Figure 2.

Both of these problems can be solved by a single, well-chosen algorithm which creates a transitory area between levels and breaks the edges up in a rougher contour. In this case it is a custom-made algorithm inspired by Worley noise.

Solution

The first step consists of scattering seed points randomly but evenly across the surface of the heightmap, taking the equivalent height values from the input. I. e. if point X's location is above a black pixel, its value will be 0.0. If it's above a white pixel, its value will be 1.0. The number of seed points is proportional to the number of pixels in the heightmap and can be adjusted for different end results.

The second step is parsing the entire mesh and adjusting the heights of points based on the nearest N seed points as a linear interpolation of their

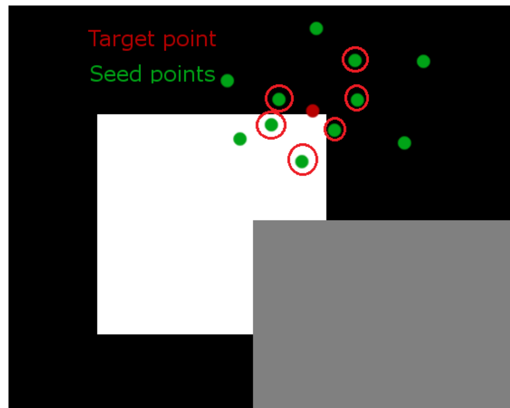

Figure 10. Seed points around a fixed point.

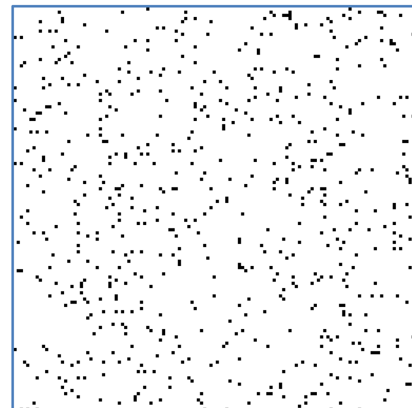

Figure 11. Seed points distribution.

assigned height values using the distance between the affected point and the seed as a weight. This differs from classical Worley noise, where only the N-th closest point is considered in the rendering function. Increasing N enlarges the area which affects the mesh point, meaning the transitional area between altitudes becomes wider. Exemplified in Figure 10. Circled are the seed points used in computing the new point's height when N = 6. The target point will be affected by 4 perfectly black seeds and 2 perfectly white ones, meaning it will end up closer to, but not perfectly, black.

The randomness of the seed point location creates the rough, natural edges that users expect to see and permits infinite variations on the exact contour when randomly redistributing the seed points again: at one time, the mesh point is affected by 6 low-height points and 3 great-height points then, during another run, the same point is affected by only 2 low-height points and 7 great-height points because in the new seed distribution, the positions change and, hence, distances are altered.

The issue of sharp elevation transitions is solved by the linear interpolation between neighboring seed points by creating transition areas which are equally random in appearance, even if this detail is less noticeable.

3.3 Edge smoothing – step-by-step example

Following is a detailed description of how this algorithm works on a given example. The input consists of a crude heightmap drawn in grayscale using MS Paint and shown in Figure 9. This kind of input is illustrative for what lack of detail really means. White represents the highest points and black the lowest, with gray representing an approximately in between height. This particular heightmap represents a plateau on the lower-right side of the map, culminating in a peak towards the center of the map. The rest of the map is drawn at its lowest point, representing either something akin to a valley or the sea bottom. This is, of course, very far from a realistic representation of the desired map. It contains a bare-bones representation of the desired terrain features: a “plateau”, a “mountain” and the “sea-level”.

If one were to interpret the heightmap as-is, the issues presented previously become painfully obvious. Since black is the lowest height and white is the greatest height, the transition creates a sharp drop, sharper than a natural one. The transition created here would be a perfect straight drop, unlike anything seen in the natural world. One may argue that normal smoothing would work, creating a transitionary area between the altitude zones. However, that will simply alleviate the issue, not remove it, since the crude shapes will remain. Moreover, the regular way normal smoothing works means the edges will remain the same, utterly un-natural-looking. Straight edges will remain straight and all curves will retain their shape. This means that all circles, ovals and other standard editor shapes will preserve the shape and their distinct edges, making the heightmap unusable nor recognizable as a terrain heightmap.

Both problems can be solved by randomly sampled linear interpolation.

This happens by distributing across the entire input heightmap a number of seed points which take the height value of the pixel situated at their position on the input image, this being a value between 0.0 for the color black, bottom altitude level, and 1.0 for the color white, top-most altitude level.

For exemplifying the procedure, the number of seed points was chosen to be one for every 25 pixels and distributed evenly across the surface. I.e. All sub-sections should have the same average number of seed points contained inside them. Figure 11 shows the seed points scattering. No color has been added to emphasize the randomness yet equality of distribution.

As it can be seen, there is no inherent pattern to the seed points. Any kind of random distribution algorithm works as long as the distribution is uniform across the entire surface. Uneven distributions will distort the space and, while it may be a useful experiment to test the effects of uneven distributions on the actual synthesis of the terrain, it is not within the scope of this project and will, hence, be avoided.

The seed points are, however, useless without some kind of data attached to them. This data is represented by the value of the underlying pixels by mapping the seed point distribution onto the provided crude input heightmap. Remember the previously presented example input, showcased in Figure 9. The next figure, Figure 12, is a superposition of figures 9 and 11, showing how the seed points get their values from the underlying input pixels. Seed point coloring has been preserved to reflect their grayscale values. A red background has been used for contrast with the seed points

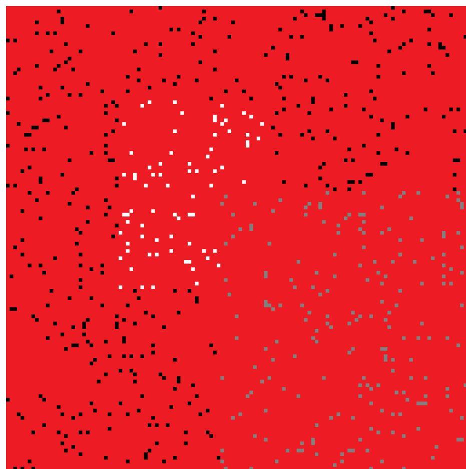

Figure 12. Seed point values.

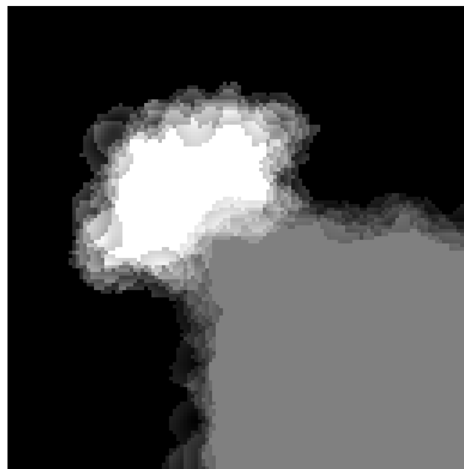

Figure 13. Interpolation result.

because no grayscale value would have been appropriate since it would possibly blend with some of the seed points.

Visually, the previous picture already displays random characteristics of the edges between different altitude areas. Even though there is a clear spatial delimitation of seed point values, provided by mimicking the exact pixels from the input, which is delimited very clearly, the edges become less defined by the nature of the randomness of seed point positions. Every pixel is going to be assigned a value representing the linear interpolation using distance as weight between the nearest N neighbors. A point with many black neighbors and a few gray ones will be decidedly darker than one with many white neighbors and a few gray ones. A point with only white/gray/black neighbors will also have that exact color, through the nature of interpolation.

This interpolation process' result is presented in Figure 13. Notice that because of the seed points, the edges are no longer straight nor are the transition zones even. Since they are influenced by the random placing of seeds, there are plenty of transitory areas between the large, flat altitude zones.

It can be easily seen that between the flat areas (white, gray and black) now smoother transitions are being made. Several different shades of gray have been added to the edges and they create transitory areas. Moreover, these transitory areas are very uneven, reducing the synthetic appearance of predefined image editor shapes. These edges present a much more palatable terrain simulacrum, even if it is not a perfect or geomorphically correct one. This step is, thus, quintessential for the efficacy of this algorithm and can be separately implemented for various related purposes. It represents the backbone of the entire system. The rest of the steps simply improve upon this model and refine it.

3.4 Adding detail

After creating transitory areas at the edges, the model is left with large expanses of flat terrain. This happens because all throughout the same level of value, encompassing seed points will all have the same value, hence linear interpolation produces that value repeatedly.

At this point, another procedural generation algorithm can be used and overlaid on top of the model in order to create rough detail across the flat areas. We have chosen Perlin noise for this purpose, because it is a homogenous noise. The fact that this noise is homogenous means that any

two neighboring points have close values and create a pleasing, flowing aspect, unlike true random noise.

The noise will be added as a small increase in height across the entire terrain model. This means it will affect both the large areas of flat terrain and the previously created transitory ones. This will improve the aspect of the terrain and make it more palatable for the human eye. As a side-effect, it adds randomness throughout the model, increasing reusability and the diversity of potential outputs. However, since the detailing is small compared to the overall scale of the terrain, this remark is not of such great importance.

We should add that this step may be replaced by another way of imprinting a more realistic texture to the terrain. The caveat is that one should take care not to add complexity to the user interaction, like needing a secondary input, such as a realistic texture for imprinting upon the model or an extended number of added parameters.

3.5 Detail smoothing

After applying the Perlin noise as a means for detailing the terrain model, the shape of the model needs to be smoothened to eliminate any kind of sharp peaks that may occur near the edges. This step also helps make the terrain more pleasing to the eye.

Odd peaks and shapes

This phenomenon may happen because as Perlin is applied uniformly across the mesh, it also affects the slopes previously created. The points on these slopes will be displaced and sometimes the displacement goes against the desired shape, i.e. a point will increase in height whilst it would be aesthetically pleasing to remain fixed or decrease in height.

Digital filters – image processing

The chosen solution to the previous problem is to run the whole model through a digital noise reduction filter. This will effectively remove any "noise" which, in this case, is represented by those seemingly random shapes.

A median or mean filter with a 3x3 kernel is perfectly reasonable to solve this issue and any other oddities the terrain model may show. It is applied to the entire model. One run through should suffice, since over-applying a filter will reduce the level of detail, counter to the initial purpose of this algorithm. Likewise, care should be exercised with more powerful filters,

some of which will strip too much detail even with a single pass.

After the completion of this step, one should be left with a reasonably detailed terrain model which respects the initial feature placement requirements provided by the user through a crudely-drawn heightmap. Needless to say, this algorithm will work just as well with more complex input, meaning it is suitable for the entire range of possible heightmap detail.

Figure 14. Left: Simple Heightmap; Middle: Detailed Heightmap; Right: Complex Heightmap.

4. Implementation

4.1 The Unity game engine

For implementing and testing, the Unity (2016) game engine was chosen because of its existing rendering engine and ease of programming using self-contained scripts. All steps have been converted into C# scripts and linked together.

The algorithm is implemented using operations on a float value matrix representing the terrain model then said matrix is applied onto the heights of a mesh, effectively rendering the result onscreen.

The following testing section has been fully realized using the Unity implementation. Due to mesh restrictions, the size of the samples has been reduced to under or at 128x128 pixels.

5. Testing and Validation

For the purposes of testing, only the aesthetics of the final terrain model have been taken into consideration. Completely ignoring the performance aspect, since it is reliant on implementation, the testing focuses on confirming that the before-stated issues are solved and that the final model is at least partially resembling a natural form of terrain. The three heightmaps used for testing are: an overly simplistic one, a slightly detailed one and a very detailed one. The first twos were made by hand, the third one is sampled from the Internet (Hoddminir, 2012); presented in Figure 14.

5.1 Validating result

Initial testing was done to prove the algorithm does indeed end with a detailed model of plausible terrain. It bears mentioning again that the end goal was not realistic terrain. Instead, it was to create a level of detail that more closely resembles realistic terrain models. Any sufficiently detailed model which may pass for terrain is good enough for confirmation. Figures 15 shows how the model advances from its initial state to the final, more

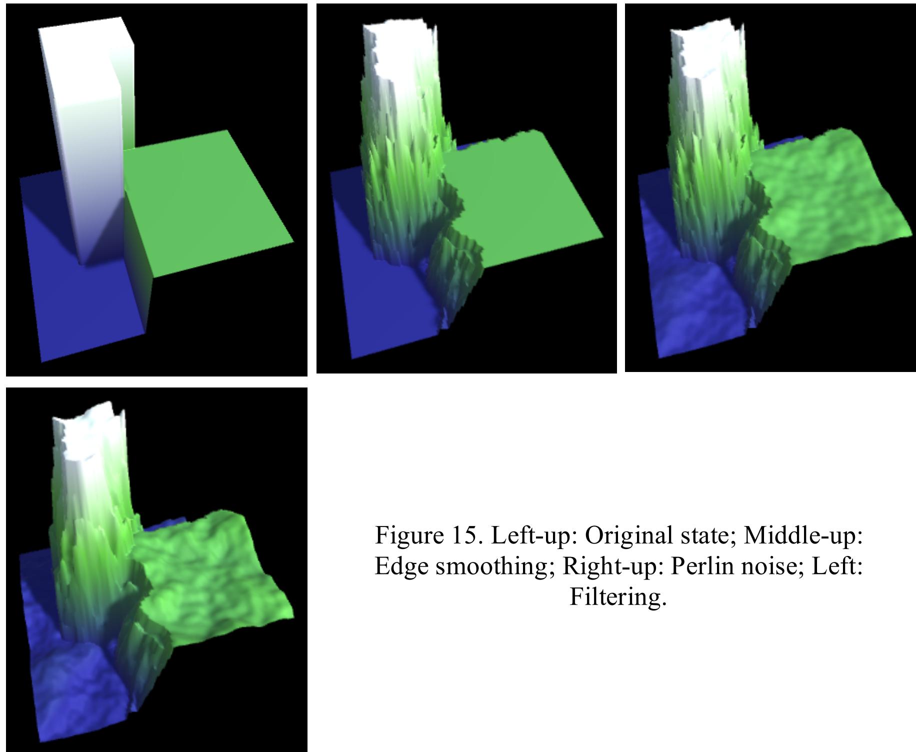

Figure 15. Left-up: Original state; Middle-up: Edge smoothing; Right-up: Perlin noise; Left: Filtering.

detailed output. Figure 15 also showcases visual artefacts (sharp edges, noticeable on the "mountain" edge) remaining from previous steps and how they are eliminated through filtering.

Figures 16 shows the effect on more detailed heightmaps. The result is proof that when confronted with too much detail, the algorithm overrides part of it with its own edge smoothing.

5.2 Varying parameters for edge smoothing

These tests have been done to empirically find reasonable value ranges for the number of nearest seed points and the total number of seed points by altering one of them and keeping the other constant.

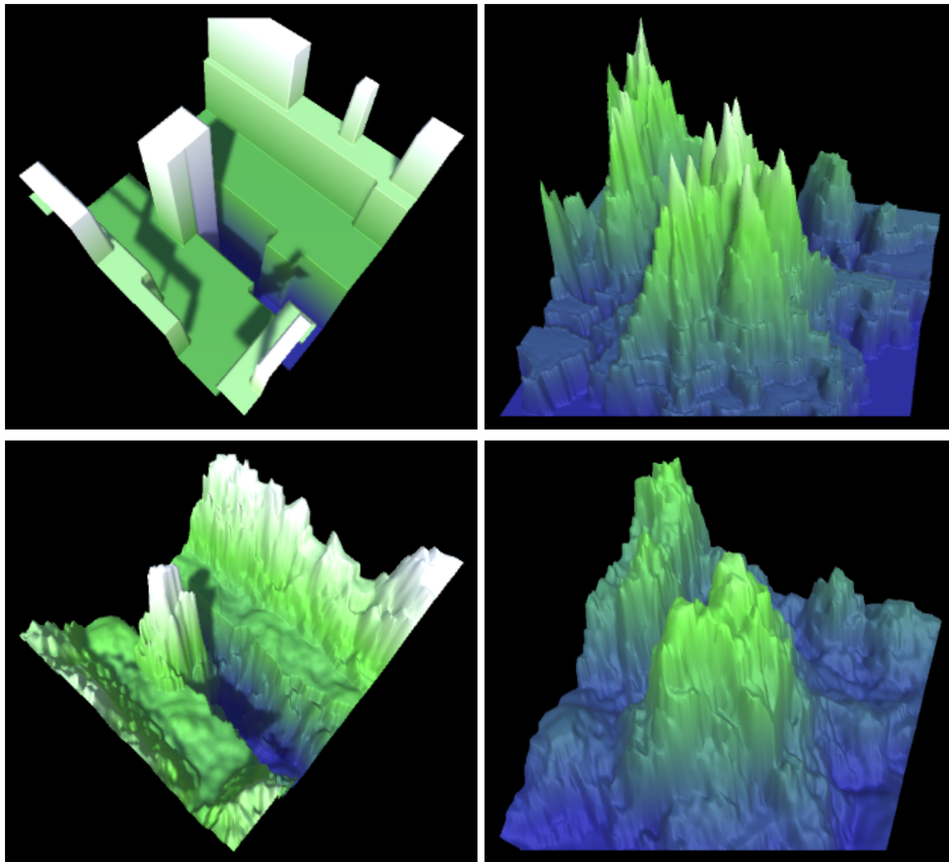

Figure 16. Original and final model. Left: Detailed highmap.
Right: Complex highmap.

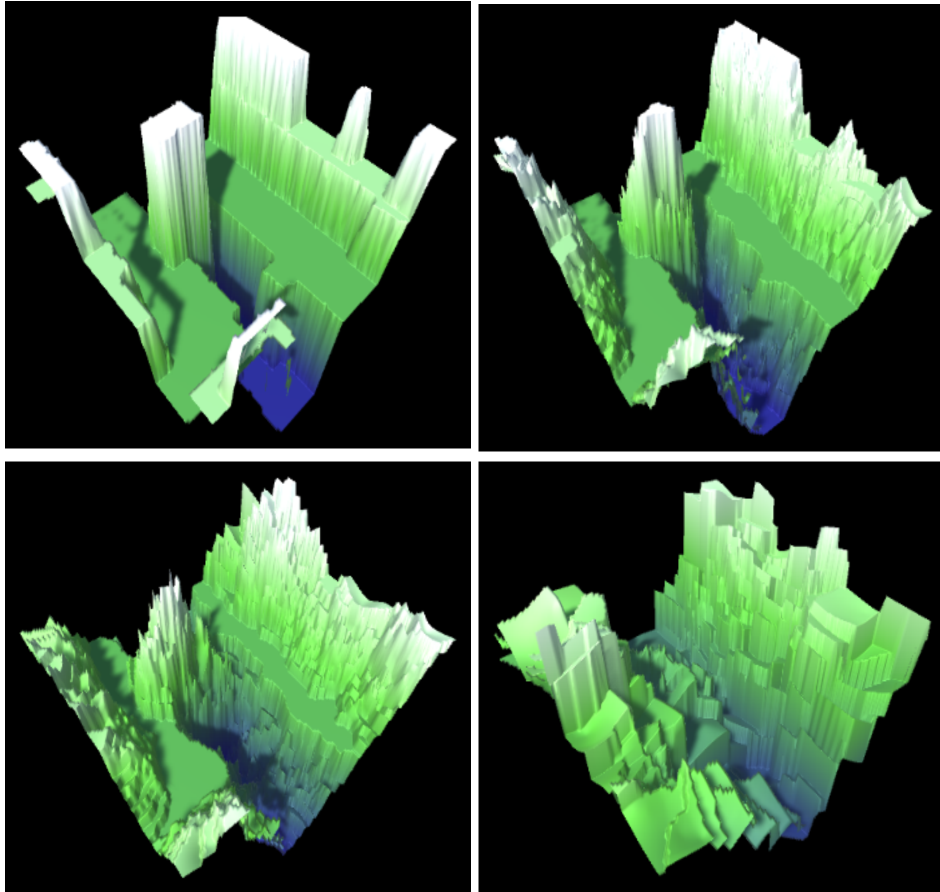

Figure 17. Seed Point Number. Top-left: 1:1; Top-right: 1:20;
Bottom-left: 1:50; Bottom-right: 1:250.

Number of seed points

This number dictates how many seed points in total are scattered throughout the plane. The number is related to the total number of pixels available (height x width). We shall use the seed point to pixel ratio to mark the total number of seeds. Hence, a 1:10 ratio would mean there is exactly one seed point for every 20 pixels. As we grow the number of seeds, the area of influence for each point in the model decreases, as more seeds are found in its direct vicinity. This preserves more of the initial detail of the image, counter to what edge smoothing is supposed to do. On the other hand, too

few seeds mean that the terrain will no longer respect all the details provided. There may be entire areas uncovered by seeds and, thus, initial detail is not preserved enough. Figure 17 presents a succession of models, showcasing this effect.

There are a few things to be noticed from this empirical evidence. As the number of seed points increases, so does detail fidelity, since the saturation of seeds is becoming evident, especially when there is one seed of every pixel of the image. This is an unwanted result, since this step is supposed to add detail to the edges. The first picture highlights this shortcoming. As the number of seeds decreases, an increase in variation of detail can be observed, however, after a point the detail becomes scarce, since too few

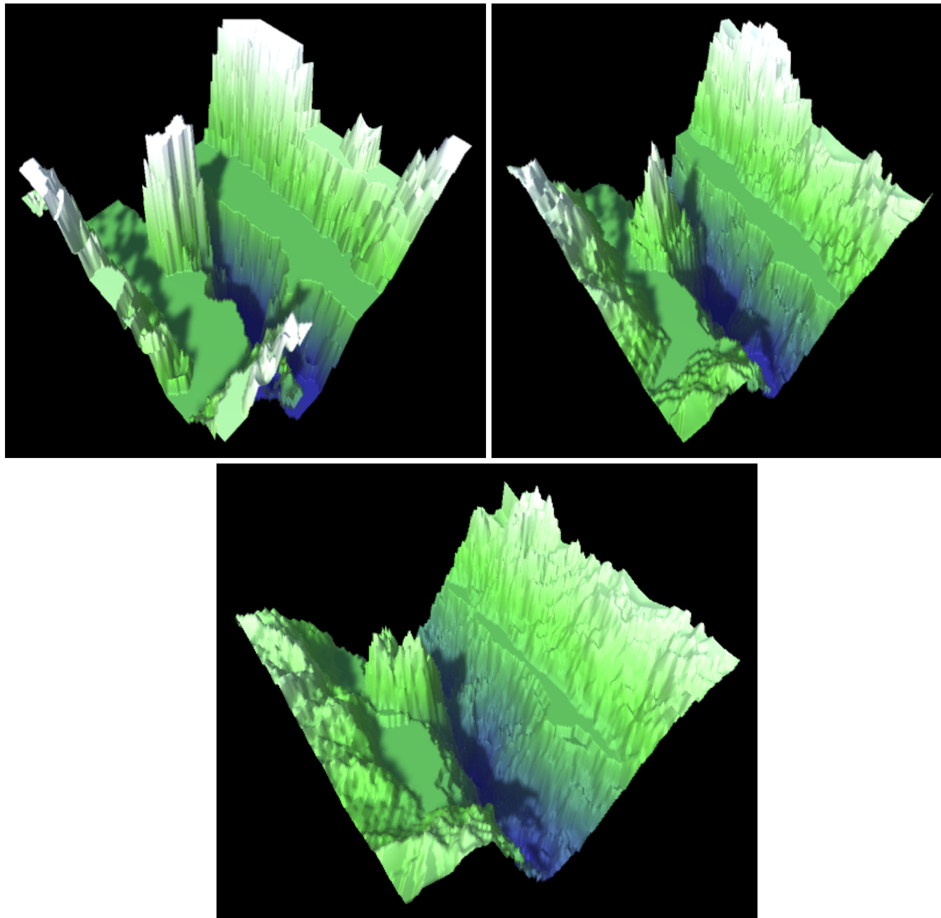

Figure 18. Neighbor number. Left: 5; Right: 15; Bottom: 25.

seed points are left to capture the initial values of the image. This test suggests that going with less than 1:100 seed points per pixels is bad for keeping the detail needed to maintain the useful terrain features displayed in the input heightmap. The sweet spot is observed to be somewhere between a ratio of 1:10 and 1:50 seeds per pixel, anything more than that risking to steal precious detail from the given input.

Number of nearest seed points

Testing the number of nearest N points taken into consideration is also a worthwhile experiment, to showcase how different values affect the outcome. For this test, the number of seed points is TotalPx/25. Results empirically show that as the number of neighbors decreases, detail fidelity increases, up to a point where the desired smoothing effect is cancelled. When there are too many neighbors taken into consideration, the smoothing is too strong and the entire map becomes flattened. Figure 18 shows this effect.

Conclusion on the seed point based approach

While the number of seed points and neighboring seeds affect each other too, empirical results point to the area centered in TotalPx/20 -> TotalPx/25 seed points total and considering around 10-15 neighbors. This should provide acceptable results for most use-cases. Of course, this does not prevent one to experiment and find proper values depending on the given input heightmap.

6. Conclusions

In the world of procedural generation, terrain synthesis is one of the most common uses, allowing for inexpensive yet complex backgrounds in movies, video games, simulation software and other possible areas of interest. The rising demand for such algorithms has given birth to a vast array of advances in this field, ranging from pure optimization to hybrid algorithms and brand-new ones designed to bring a wealth of detail into the final model.

While specialized software is constantly trying to simplify the interface and make procedural terrain generation available to the layman, it must always make compromises regarding input detail versus output detail. Trying to detail incomplete or crude heightmaps is something few people are trying to elaborate on since the focus is on the end product – a realistic

terrain model – and all inputs are usually simply mirroring the demands for the algorithm instead of the other way around.

This paper presented a procedural generation algorithm that is supposed to work with minimal input detail. It outputs something aesthetically close to real terrain models, even if it lacks any kind of groundbreaking detailing. A case can be made for using this algorithm as a preliminary for other systems, detailing crude heightmaps to a level acceptable for more advanced synthesis software and / or algorithms.

While not overly complex, this algorithm proved that there is hope for terrain synthesis from input of any detailing level and that one may still discover new techniques for allowing inexperienced users to generate beautiful scenery with minimal effort.

Acknowledgment

This paper is supported by the Sectoral Operational Programme Human Resources Development POSDRU/159/1.5/S/137516 financed from the European Social Fund and by the Romanian Government.

References

Adrian's soapbox, *Understanding Perlin Noise*, (2016) [Online]. Available: http://flafla2.github.io/2014/08/09/perlinnoise.html.

Crankshaft Publishing, *Image Enhancement Via Spatial Filtering (Image Processing) Part 1*, (2016) [Online]. Available: http://what-when-how.com/embedded-image-processing-on-the-tms320c6000-dsp/image-enhancement-via-spatial-filtering-image-processing-part-1/.

Cruz, L., Ganacim, F., Lucio, D., Velho, L., and de Figueiredo, L. H. (2013) *Exemplar-based Terrain Synthesis*.

Dirichlet, G. L., (1850) Über die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen, *Journal für die Reine und Angewandte Mathematik*, no. 40, pp. 209-227.

Discover, (2016) *World Machine Software, LLC*, [Online]. Available: www.world-machine.com/about.php?page=features.

Fisher, R., Perkins, S., Walker, A., and Wolfart, E., (2003a) *Spatial filters - Mean filter* [Online]. Available: homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm.

Fisher, R., Perkins, S., Walker, A., and Wolfart, E., (2003b) *Spatial filters - Median filter*, 2003. [Online]. Available: homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm.

Gaia, (2015) *Procedural Worlds*, [Online]. Available: www.procedural-worlds.com/gaia/.

Hoddminir, B., *From Heightmap to Worldspace in Skyrim*, (2012) [Online]. Available: hoddminir.blogspot.ro/2012/02/from-heightmap-to-worldspace-in-skyrim.html.

Mangra, A. P., Sabou, A., and Gorgan, D. (2016) Terrain Synthesis from Crude Heightmaps, in *Proceedings of RoCHI2016*, Iasi.

Perlin, K., *Making Noise*, (2002) [Online]. Available: www.noisemachine.com/ talk1/.

Perlin noise, Wikipedia, (2016) [Online]. Available: https://en.wikipedia.org/wiki/Perlin_noise.

Unity - Game Engine, Unity Technologies, (2016) [Online]. Available: Unity Technologies.

Voronoi, G., (1908) Nouvelles applications des paramètres continus à la théorie des formes quadratiques, *Journal für die Reine und Angewandte Mathematik*, no. 133, pp. 97-178.

Zhou, A. H., Sun, J., Turk, G., and Rehg, J. M. (2007) Terrain synthesis from digital elevation models, *IEEE Transactions on Visualization and Computer Graphics*.

Worley, S., (1996) A Cellular Texture Basis Function, in SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques.

.