MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

TERRAIN SYNTHESIS FROM CRUDE HEIGHTMAPS

LICENSE THESIS

Graduate: **Alexandre Philippe MANGRA**

Supervisor: **Prof. dr. eng. Dorian GORGAN**

2016

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,	HEAD OF DEPARTMENT,
Prof. dr. eng. Liviu MICLEA	**Prof. dr. eng. Rodica POTOLEA**

Graduate: **Alexandre Philippe MANGRA**

TERRAIN SYNTHESIS FROM CRUDE HEIGHTMAPS

1. **Project proposal:** *Developing and implementing an algorithm which would allow the synthesis of terrain models from insufficiently-detailed input heightmaps.*

2. **Project contents:** *Introduction, Project Objectives, Bibliographic Research, Analysis and Theoretical Foundation, Technological Considerations, Detailed Design and Implementation, Testing and Validation, User Manual, Conclusions, Bibliography, Appendix 1.*

3. **Place of documentation**: Technical University of Cluj-Napoca, Computer Science Department

4. **Consultants**: As. Prof. dr. eng. Adrian Sabou

5. **Date of issue of the proposal:** November 1, 2015

6. **Date of delivery:** June 30th, 2016

Graduate: ______________________________

Supervisor: ______________________________

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY

OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind
autenticitatea lucrării de licenţă

Subsemnatul **ALEXANDRE PHILIPPE MANGRA**, legitimat cu **C.I.** seria **KX** nr. **815097**
CNP **1930514125795** autorul lucrării **TERRAIN SYNTHESIS FROM CRUDE HEIGHTMAPS** elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea **CALCULATOARE, ENGLEZA** din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea **IULIE** a anului universitar **2015-2016** declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data	Nume, Prenume
30.06.2016	**ALEXANDRE PHILIPPE MANGRA**

Semnătura

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Table of Contents

Table of Figures

Chapter 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.

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 (TSCH: Terrain Synthesis from Crude Heightmaps) and its implementation 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.

Note: this paper elaborates on and reuses parts of a paper currently under consideration for the RoCHI conference, "Terrain Synthesis from Crude Heightmaps". This is justified by the complete overlap in what each of these papers describe with the mention that this particular one will expand and detail more on the implementation and testing.

Chapter 2. Project Objectives

This project started off as a research project trying to discover new and / or easier ways of procedurally generating terrain. The algorithm research converged towards a handful of specific procedures, namely Perlin noise and Worley noise applied to user input. A crude vision of the TSCH (Terrain Synthesis from Crude Heightmaps) algorithm was thus born. The project as a whole has, thus, two main objectives.

2.1. Develop the TSCH algorithm – Solve the crude user input issue

The algorithm needed to be developed up to a point of usability. Whilst the algorithm may not present any true complexity through its structure, it is a good contender for the solving of the user input issues described in the previous chapter. This means that by developing the algorithm, I also solve those problems in one go. As the lack of user experience with using terrain generation software can prove very difficult and the learning curves are usually high, solving these problems algorithmically and then presenting this procedure to the end-user means creating an entire new branch of terrain generation, namely one focused on allowing the layman to participate in terrain generation without much time spent on learning how to use the specific tool while also obtaining decently generated results with it. As a whole, this would represent a step forward in the domain of procedural generation by lowering the experience threshold for new users and non-developer people.

2.2. Implement the algorithm in a simplistic desktop application

By itself, the TSCH algorithm is useless for the end-users. This raises a point in the usefulness of a software implementation which utilizes the algorithm in order to make it available to the layman. This means the second objective of the project is to provide a working software system based on the algorithm which should provide at least a modicum of functionality.

The application is not supposed to be a high-end procedural terrain generation software system with a large number of available algorithms and other tools to use. It is supposed to be demonstrative to the purposes of this algorithm’s ease of use and the potential it bears on the development of easy user input options regarding procedural terrain generation.

Chapter 3. Bibliographic Research

Every kind of well-designed system starts off with a solid design foundation, detailing every aspect of the application. However, that design foundation needs a foundation itself. That is the role of the research done prior to starting work on the design. Research fulfills the important function of providing all the necessary information and discovering the relevant algorithms needed for structuring a complete solution.

This chapter highlights the research work done to accommodate the goal of this application: developing an algorithm for terrain synthesis based on crudely designed heightmaps. It will also present the decision steps taken from the inception of the project to the current status. The focus is on terrain synthesis, following the presentation from the previous chapters regarding a short introduction to procedural generation as a whole and its role and place in today’s world.

3.1. 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.

Following in the next chapter 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.

3.1.1. Software

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 [1]. 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 [2], 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.

3.1.2. Research

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. [3], 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. [4], 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.

3.2. 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 Voronoy, Russian and Ukrainian mathematician who defined this type of diagrams [5]. 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" [6]. Because of this, the regions compounding the diagram are called Dirichlet regions and the diagrams may be called Dirichlet tesselations. The diagrams have applications in science and technology, with the added capacity of aiding visual artists. An example diagram can be seen in Figure 3.X.

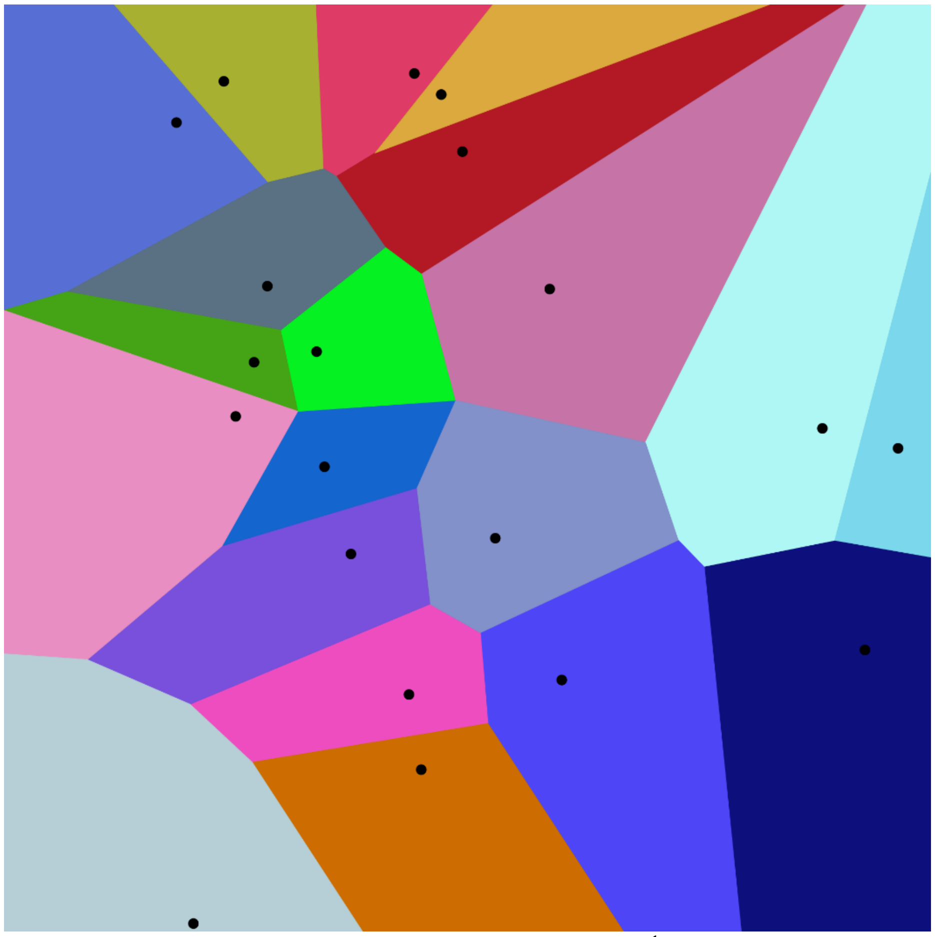


Figure 3.1 Voronoi Diagram [1]

[1] Courtesy of Wikipedia: https://en.wikipedia.org/wiki/Voronoi_diagram

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.

3.3. Worley Noise

Worley noise [7] is a type of cellular noise. This means that the noise takes form of „cells” inside the given plane. 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.

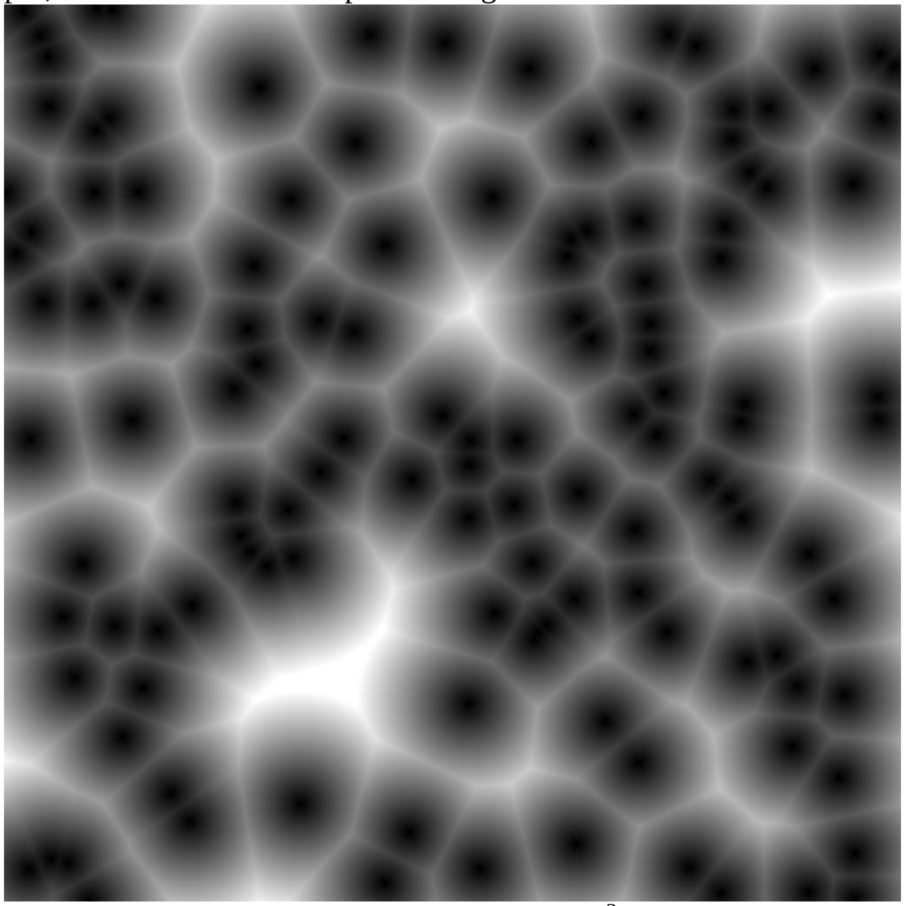


Figure 3.2 Worley Noise[2]

[2] Courtesy of Wikipedia: https://en.wikipedia.org/wiki/Worley_noise

3.4. Perlin Noise

Noise, per se, is usually associated with completely random values appearing within data and can occur at any value across the entire spectrum the data can be represented in. As far as random goes, „normal" noise is not very useful. It is simply a totally or pseudo-random distribution of values which does not conform to any rules. Any two neighboring values have no relationship whatsoever. There is no algorithm. No rule system which would allow one to generate „normal" noise. It is, hence, not usable in any practical application. Furthermore, it is usually removed by the use of various digital filters, exactly because it acts as an obstacle in the way of clarity.

Perlin noise was created by Ken Perlin in 1982 for the movie Tron because he was unhappy about how CGI looked at that time [8]. 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.

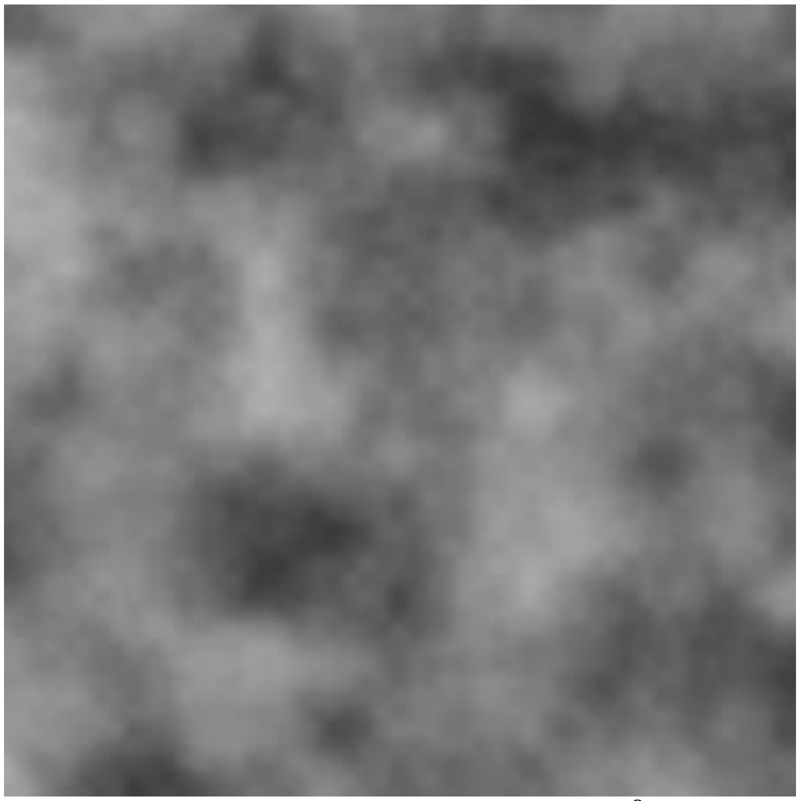


Figure 3.3 Perlin Noise 2D Example [3]

One can easily notice that despite the random appearance of the regions, everything is smoothed out. There is no dissonance between any pair of points whatsoever. The transition between values is done smoothly, over an expanse of space, never suddenly, like the case of true random noise. The algorithm itself will be presented later in this paper.

3.5. Digital Filters

Digital filters are algorithms which exact mathematical operations on the problem space, usually represented by signals, to modify different aspects of it. In image processing, the filters are usually used to modify various aspects of the image, like isolating the low or high frequencies of the color values, smoothing it, enhancing it or possibly detecting edges. The first filters were implemented in 1965 by Whitehouse and Reason [9] by simulating a two-resistor-capacitor network filter digitally.

The filters used in this project are making heavy use of sliding screens over the source input to apply the desired changes. The filters to be used are the median [10] and mean [11] filters.

[3] Courtesy of Wikipedia: https://en.wikipedia.org/wiki/Perlin_noise

Chapter 4. Analysis and Theoretical Foundation

As stated before, unrefined heightmaps are, through their very own nature, hard to use. The lack of detail means there is little to infer from whatever existing information is available. However, on the other hand, one cannot completely ignore the given heightmap, since it contains the user’s requirements, crude as they may be, for the terrain to be synthesized. The situation is thus, that the user presents enough input to form a general idea of what he or she desires as a result but does not give abundant details. Furthermore, one may encounter heightmaps with a detail level so scarce, there is almost nothing to gain from analyzing it. This requires one to find balance between the creation of new features and merging with the initial information, representing the user’s actual intent.

A theoretical solution would be one which is able to both keep the main guidelines of the input heightmap and, at the same time, detail it in such a way as to resemble terrain. Working under the assumption that predominantly there will be crude heightmaps offered as inputs, finding a way to create detail is slightly more important than respecting the features exactly as drawn. More so as the features may be too badly drawn to have any use being used as definite guidelines.

From all the problems a simplistic input may have, a few can be singled out as the most pressing of all and then focus can be expended on prioritizing the solving of them. These problems appear in most, if not all, of the heightmaps presented by people who lack the needed artistic skills to draw elaborate maps and lack the patience and knowledge to find already existing maps that conform to their set of specific requirements.

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. These changes are unaesthetic to the eye and are ignored by most algorithms, plaguing the output even after several algorithms have been applied. There is little amelioration possible by instructing the user to use intermediary grayscale values around the points of interest inside the heightmap but for smooth transitions a large number of these values would have to be used; something a regular, non-artistic user will be unable to accomplish without great effort on their behalf. This issue is, thus, realistically something which will appear in all provided crudely-drawn heightmaps and becomes a top priority issue to be solved if anything pleasing to the eye should be output and labeled as a ‘terrain’.

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. The random, inexperienced user will try to use tools available to him in common painting software, like brushes and pre-made shapes. This will cause straight or curve lines to occur in most instances of user input. The skill required to create something more detailed takes so much time to master that it is best left to the algorithm to solve.

Because of the previously presented issues, I have formulated an algorithm that tries to work with insufficiently-detailed heightmaps and provide the user a terrain model which is passable in terms of both quality and visual similarities to real-world terrain. The scope of this algorithm is not geomorphically correct terrain nor overly-detailed terrain.

The scope is restricted to synthesizing a model that is aesthetically pleasing and can be used as a final model in the desired environment or presented as a more-detailed input for other complex terrain synthesis algorithms. Following are three sub-sections which present, in order, a detailed description and walkthrough of the algorithm itself; the conceptual architecture of a system designed to support this algorithm and the most important use-cases for this system.

4.1. 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 4.1 presents such a grayscale heightmap example. To illustrate what information heightmaps contain, Figure 4.2 presents the same heightmap, rendered in three dimensions based on the heightmap values.

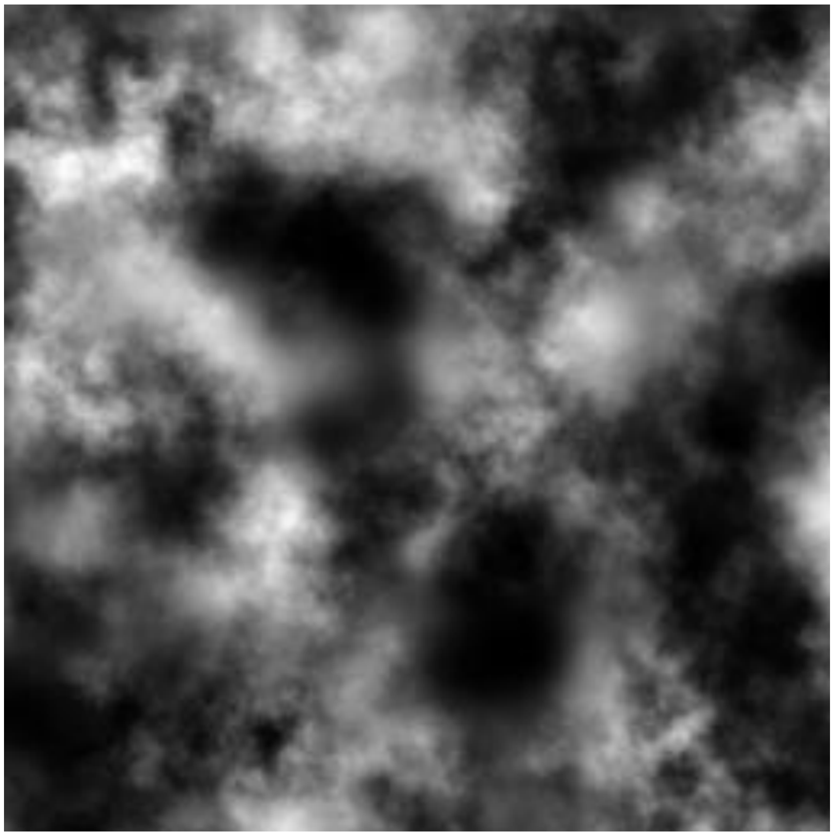


Figure 4.1 Heightmap – Grayscale

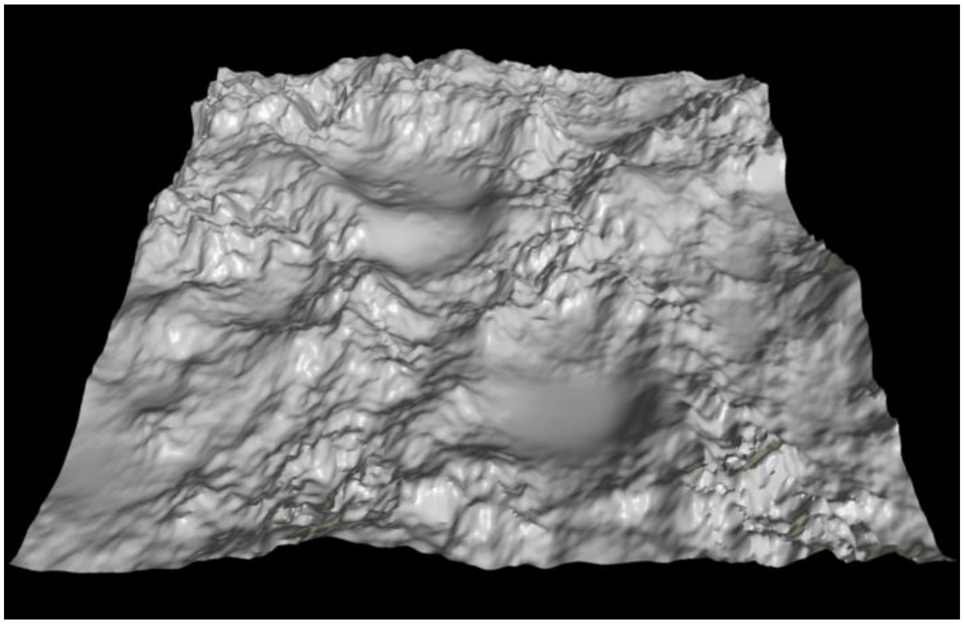


Figure 4.2 3D Rendered heightmap

4.2. Existing Algorithms Description

This section elaborates on the algorithms described in the “Bibliographic Research” section by presenting the actual algorithm description. As opposed to simply describing the origin and possibly use of said procedures, this section enables the future developer to implement the algorithms in their true form without forcing them to resort to third-party implementations or having to access multiple knowledge repositories in order to obtain the necessary information.

4.2.1. Perlin Noise

Perlin noise divides the space up in equally 90-degree-cornered segments (i.e. a grid in two dimensions, cubes in three dimensions). 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.

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 4.3 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).

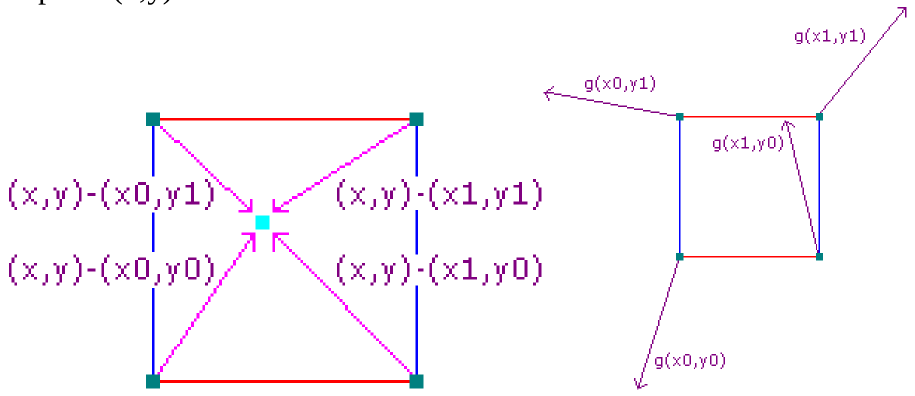


Figure 4.3 Distance Vectors (left) and Gradients (right) for a given two-dimensional segment

The value of the Perlin noise at point (x,y) is calculated by interpolating the surrounding gradient vectors using the distance vectors as weights. This means that all points inside the region will be a cummulation 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.

4.2.2. Worley Noise

Worley noise is a noise variation of the classical Voronoi diagrams also based on partitioning the problem space into areas based on a number of seed points distributed across the entire space. The created forms also bear a strong resemblance to the aforementioned diagrams, even though the rule system used to create them is different. The main concept is identical, though. The concept of using the distance from those points to determine the value of every point inside the affected plane.

This noise is generated by applying a transformation function to the Nth nearest neighboring seed point of the given spatial point. Voronoi diagrams are computed by giving that point the value of the closest seed point (f(x) = x and N = 1). Worley noise, however, permits one to use any kindof transformation function and it changes the seed point the value is rapported to, meaning one can arrive to drastically different results by simply varying N. The noise appears when the seed points are randomly distributed across the plane before the values are taken into consideration. This algorithm is presented because it served as inspiration for the algorithm upon which this paper is based upon.

4.2.3. Digital Filters – Noise Reduction

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 are 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).

The image processing filters considered here work by utilizing a sliding window that goes over the entire input, modifying it as configured. 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 4.4, 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.

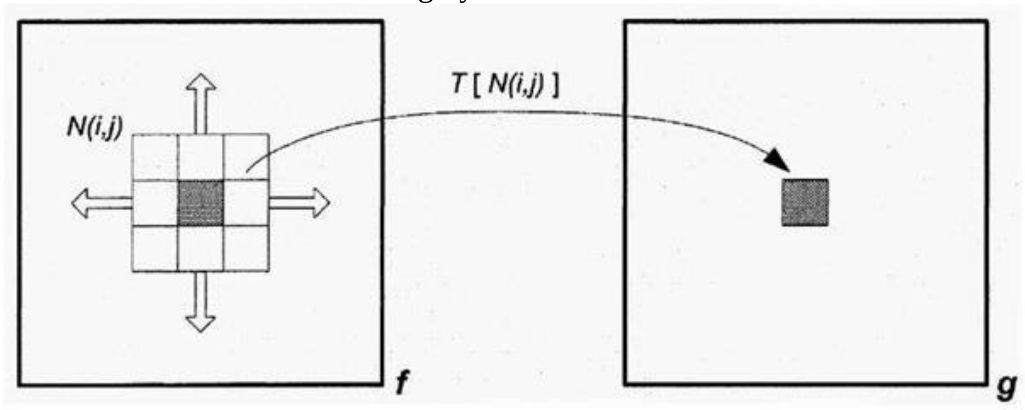


Figure 4.4 The Sliding Window and the Filter Function[4]

The two filters whose use is to be considered in this software implementation are the mean filter and the median filter. 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

[4] Courtesy of The-Crankshaft Publishing: http://what-when-how.com/embedded-image-processing-on-the-tms320c6000-dsp/image-enhancement-via-spatial-filtering-image-processing-part-1/

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.

4.3. Terrain Synthesis from Crude Heightmaps Algorithm

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 a 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.

Following is the algorithm I propose 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 any kind of specific detail, though it should be noted that it will work on all kinds of heightmaps, not only crude ones. The algorithm is composed of three main steps: edge smoothing, adding detail and detail smoothing. The focus lies on the first step, it being the one solving the two main issues presented by inexperienced users: sudden altitude increases and the complete lack of edge detailing.

4.3.1. Edge smoothing step

The first step of the algorithm is to take the brunt of the issue-solving by using a single sub-algorithm to solve both the sharp elevation level transition issues and the overly-simplistic edge definition issue. It is not too ambitious, however, since, as we shall see, choosing the correct procedure does indeed allow the solving of these problems.

4.3.1.1. Sharp elevation level transitions

The neophyte user will provide a heightmap where one elevation level ends and another beings 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). 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.

4.3.1.2. 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 1.

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.

4.3.1.3. Solution

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

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. This interpolation creates transition zones between elevation levels. These zones become broader as more neighboring seed points are taken into consideration for interpolation.

4.3.1.4. 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 4.5. 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”.

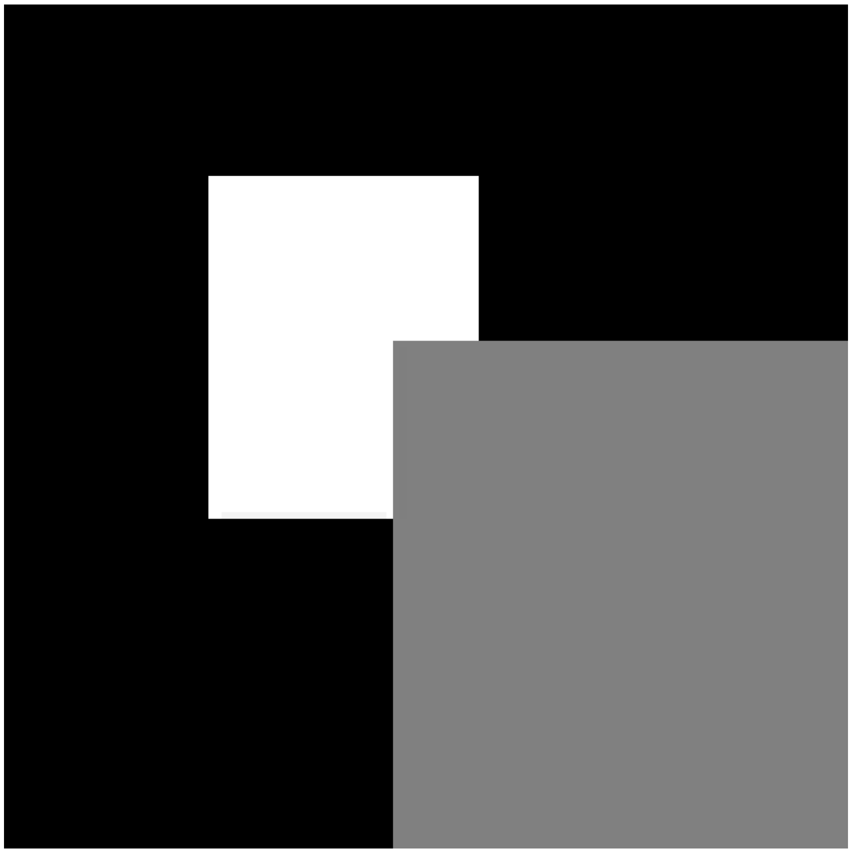


Figure 4.5 Example input

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 4.6 shows the seed points scattering. No color has been added to emphasize the randomness yet equality of distribution.

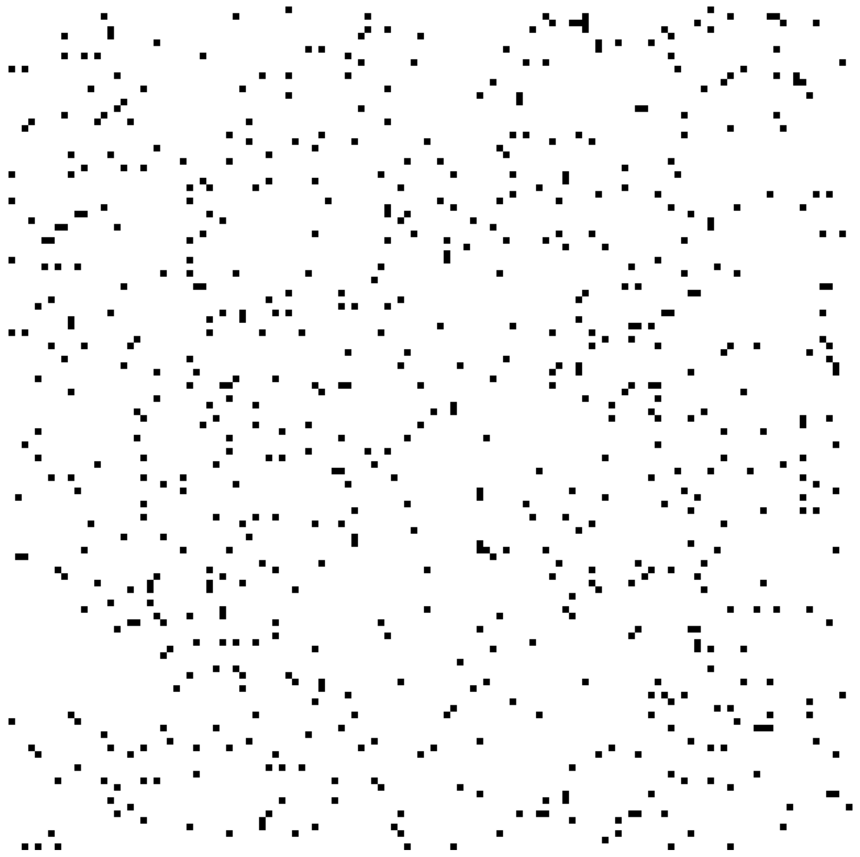


Figure 4.6 Seed points over the surface

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 4.5. The next figure, figure 4.7, is a superposition of figures 4.5 and 4.6, 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 because no grayscale value would have been appropriate since it would possibly blend with some of the seed points.

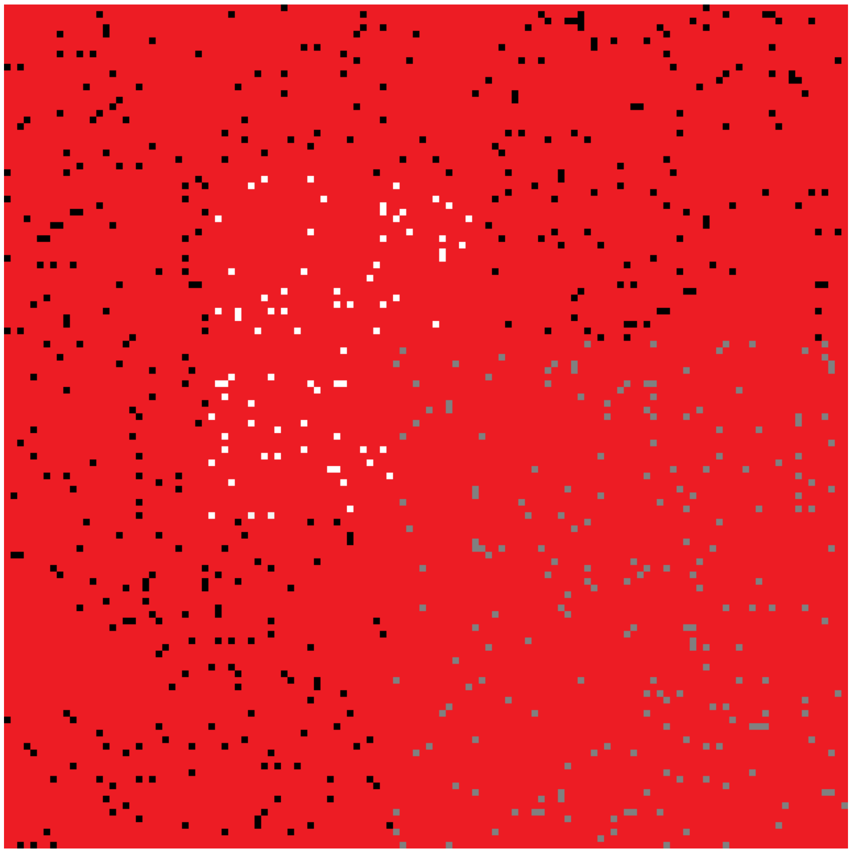


Figure 4.7 Seed point values

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 4.8. 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.

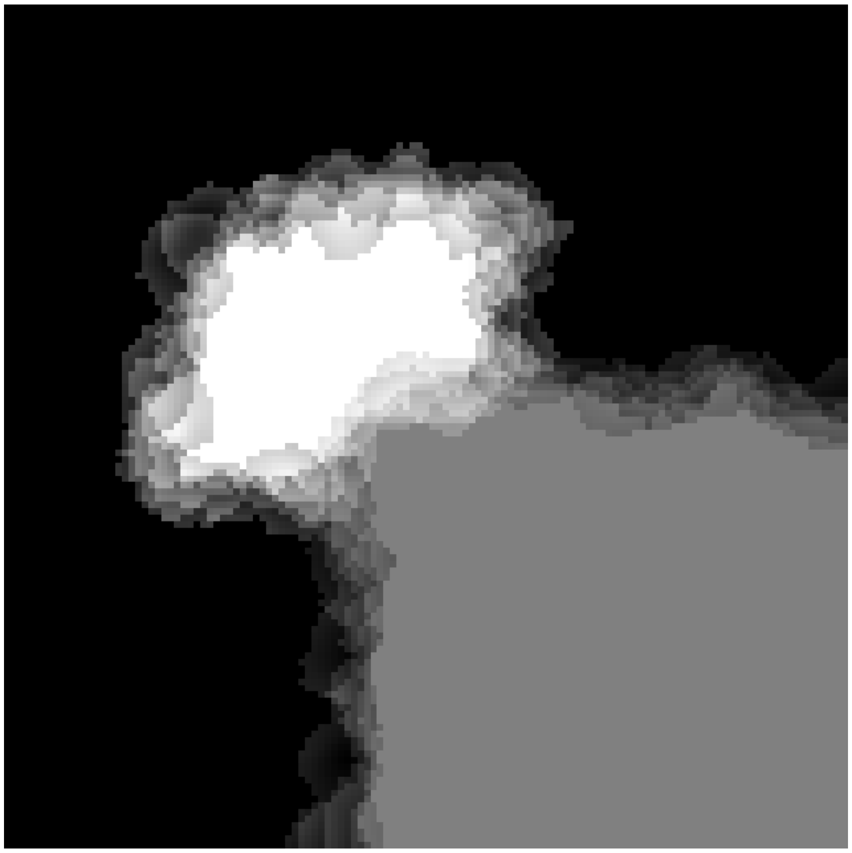


Figure 4.8 Interpolation Result

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.

4.3.2. Adding detail step

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. In the previous example, these areas are predominantly the large, black one on the left- and top-hand of the heightmap and the gray one on the bottom-right side of the heightmap.

In nature perfectly flat terrain is very, very rare. This is something to be fixed in the detail addition step by overlapping a layer of noise over the entire heightmap. This step can, of course, be replaced with any other detail-adding algorithm, allowing for deep customization of the procedure.

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. I have chosen Perlin noise for this purpose, which is a homogenous noise implementation created by Ken Perlin in the early ‘80s. 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.

I 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.

4.3.3. Detail smoothing step

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. 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. The need for smoothing also appears because the initial step of edge smoothing may still result in sharp and contrasting elevation levels inside the transition areas. These also need smoothing. 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. The smoothing is also kept to a minimum in order to not actually remove useful detail.

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.

4.4. Conceptual Architecture

The algorithm presented in chapter 4.1 is highly useful for the layman desiring inexpensive terrain models with at least average detailing obtained with minimal effort. The algorithm itself is not presentable to the hobbyist as-is. Instead, a software system must be built around it as a shell, to allow the users interaction with the working components and achieve their desired results. This interaction is conditioned by the existence of both a supporting system which replicates and manages the algorithm in such a manner that it can be used at its full extent without repercussions and a user interface simple enough not to flabbergast the user into not exerting influence upon the software application.

First off, the application needs to be able to replicate the algorithm in its entirety in such a manner that efficiency is maintained and complexity is kept as low as possible. Because the algorithm does not come with any kind of implementation details, the developer of the application is free to optimize to his or her heart's content. Due to time constraints, I have chosen not to heavily optimize the application and focused instead on the next point, making it as user-friendly as possible. It is by no means a perfect approach since this means a definite increase in execution time. However, it is a decent trade-off between the needed development time and the ease of use of the application itself. The algorithm implemented must also be properly created as to not have in reality the implementation of a different algorithm than the intended one.

Then the application needs to have a user interface designed in such a way as to not take away the usability and the user-friendliness that the algorithm itself allows. It needs to convey its properties and parameters to the layman as clearly and, most importantly, as easy to understand, as possible, in order for him or her to be able to correctly interpret the interface and manipulate the application in such a way as to lead towards the completion of the user's end goals. This means the user experience needs to be flawless and as smooth as possible, with as little domain-specific terms as possible, enabling the average user access to the software implementation without the need for a dictionary to provide linguistic support.

This chapter elaborates on the proposed conceptual architecture of the software system designed to replicate the previously described algorithm and make it available to the layman in such a way as to ease the use and minimize the learning curve. The application should, thus, be made both solid and minimalistic in order to avoid unnecessary complications on the user's side. All the necessary information will be presented to create a general perspective of the entire application and to allow one to form an opinion regarding the software implementation that will house the terrain synthesis from crude heightmaps algorithm. The detailing of such information is useful in terms of conceiving a detailed implementation of the software application described within this document and its sole purpose is precisely that: to ensure a detailed implementation is possible and that there are no important details lacking regarding the structure of such an application necessary to build the system itself.

Various concepts will be presented through the use of diagrams, pictures and/or descriptions, believed to be absolutely necessary for a successful implementation which both abides by the design set by the conceptual architecture and by the simple requirements described earlier in this chapter.

4.4.1. System Overview

In today's market, there is a need for quality software systems that deliver exactly the features one requires from them. It is unacceptable to create a software product which underperforms yet still label it wrongly as overperforming. Hence, there is a constant push towards quality and fully-featuring which not all applications are capable of delivering. Such high standards demand improved design specifications to complete and create around. The following sub-chapter was created to provide enough details for a quality software implementation of the proposed algorithm. A working shell capable of providing smooth and seamless user interaction and provide flawless user experiences.

Such high standards bring their own requirements. One such requirement is having a general view of the application. This system overview will be presented in the following paragraphs, offering a birds-eye view of the entire system, followed by a detailed explanation of each component composing the application. This kind of overview permits the developer to plan ahead by having the entire software system exposed at once and available for his or her scrutiny.

To this end, I have created a conceptual diagram showcasing the most important component of the application. This diagram will describe the major software nodes and the manner in which they interact with one another. Figure 4.9 presents this conceptual diagram with the system overview.

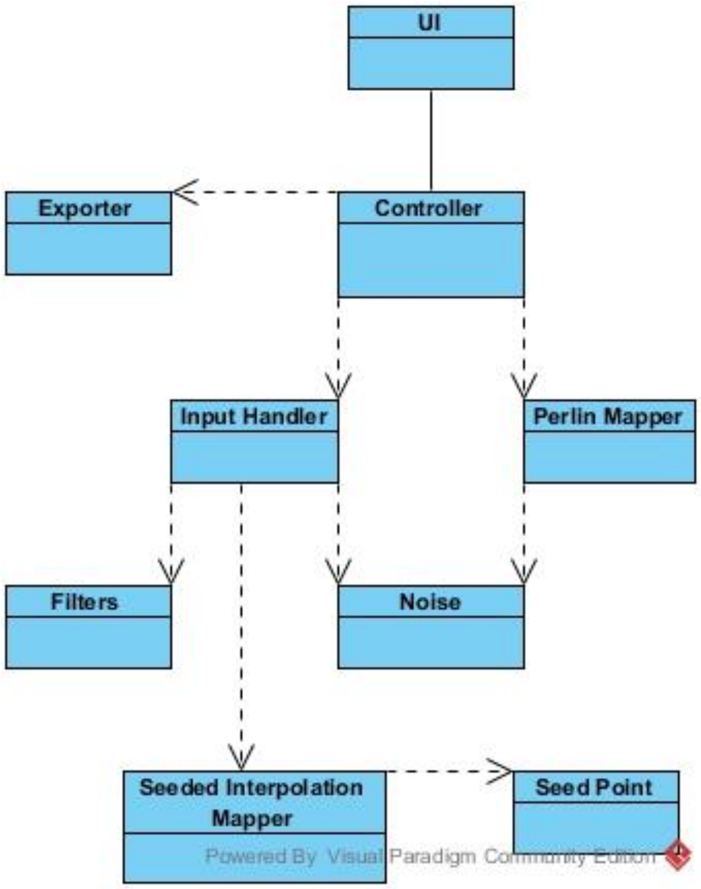

UI
Exporter
Controller
Input Handler
Perlin Mapper
Filters
Noise
Seeded Interpolation Mapper
Seed Point
Powered By Visual Paradigm Community Edition

Figure 4.9 Conceptual Diagram – System Overview

The diagram presents the main components which will comprise the entire application:

- The UI, grouped together for clarity reasons;
- The Controller, used to control application logic and interact with the user interface;
- The Exporter, used to export the heightmaps such that the user may actually use them elsewhere;
- The Perlin Mapper, used to handle a Perlin noise implementation which can be used both as a stand-alone albeit completely random terrain model or to detail the terrain obtained from parsing and interpreting the user input crude heightmap;
- The Input Handler, used to interpret the heightmap images the users offer as input for terrain synthesis, containing all the logic described in the algorithm in the previous sub-chapter;
- The Noise class, containing the actual Perlin noise module, isolated from the rest of the software implementation;
- The Seeded Interpolation Mapper, the class responsible for enforcing the first step of the algorithm: the edge smoothing step, containing from the first to last sub-steps involving the random seed distribution and nearest-N neighbor interpolation logic;
- The Filters class, a repository for digital filters used to separate them from the rest of the application out of modularity concerns;
- The Seed Point class, a class defined inside the Seeded Interpolation Mapper, purely auxiliary, used for storing relevant information about seed points, mainly the position x and y coordinates and the height value.

The aforementioned components interact with one another in such a way as to ensure the smooth workings of the application. Following is a short description of how this component interactivity works on a theoretical basis regarding the design of this software system. This short description precedes the in-depth analysis of the components in order to create a top-down view of the system and the way every component works with the others.

4.4.2. Component Overview

This segment presents each component in a detailed manner, focusing on its role in the overall architecture, the way it functions and the algorithms it entails, if any. The descriptions here form the groundwork for all the implementation work done within the scope of this project and can be considered the design foundation of the software system as a whole. The interactions presented are relevant to the final form of this project and do not serve only as proposals but as definitive guideposts to help shape the application.

4.4.2.1. UI

The UI component performs the important task of binding together all UI elements, holding references to them in order to be alerted whenever the user interacts with the interface. This enables the application to swiftly respond to any kind of given input, as set up by this class.

The sole responsibility of the UI component is to gather user input and forward it to the Controller component in order to be processed. The UI class exists only because keeping track of all the user interface elements becomes tiresome as more elements are added to the interface. This way one can refer to the interface as a whole when structuring the software system.

4.4.2.2. Controller

The Controller component, named after the controller in the Model-View-Controller pattern, is responsible for the implementation logic of the entire application. It maintains a direct connection to the user interface and receives all input from there. At the same time it sends out commands to the UI, changing it as necessary based on the user input, if any such changes are required. The second part of this class' job requirements is to handle the rest of the components and send commands down the line based on what feedback is received through the user interface.

It is responsible for handling switching between the user-input more of the application and the simple Perlin noise generated terrain. This entails handling activation and deactivation of user interface components like buttons and sliders and changing the input handling logic to cater to the user's choice. It also handles delivering the heightmap to the Exporter and retrieving it from the relevant currently-active class: the Perlin Mapper for the from-scratch terrain generation mode, and the Input Handler, for the terrain synthesis from user-provided crude heightmaps mode.

Being the „hub" of the entire application, this component is the most important, function-wise, to the user. Correct handling of thedifferent modes of the application, switching between them and relaying data and information between components and the user interface makes or breaks the entire software system hence utmost care needs to be taken when implementing this class.

4.4.2.3. Exporter

The Exporter component is one of the three main functional classes being handled by the Controller. Its role is simply providing the heightmap and possibly three-dimensional model exporting functionality for the rest of the application to take advantage of. It should serve as a one-way data processing class by receiving the three-dimensional model to export into an appropriate file format or the heightmap to be exported the same way but in a different format, more suitable for two-dimensional heighmap storing.

4.4.2.4. Perlin Mapper

The Perlin Mapper is the second of the three main functional classes being handled by the Controller. This class utilizes the Noise class to create a three-dimensional terrain model from scratch. It is not a requirement of the main algorithm used by this application, but it is somewhat useful to fully utilize the Perlin noise implementation. Not only to detail the synthesized terrain but also to create terrain from nothing, as other implementation do. This class is more of an added bonus to the normal user than a definite requirement imposed by the software system's functionality.

4.4.2.5. Input Handler

The Input Handler is the last of the three main functional classes being handled by the Controller. This class is the most complex one of the three, due to the fact that it is supposed to handle the entire implementation logic of the terrain synthesis from crude heightmaps algorithm. This component should be able to receive as input the parameters provided by the user, including the crude heightmap, and manipulate the three-dimentional model in such a way as to completely process is according to the terrain synthesis from crude heightmaps algorithm presented earlier.

Despite the simplistic description, this class bears the entire complexity of the implementation of the algorithm. This lets the Controller delegate the responsibility of morphing the three-dimensional model of the terrain into a finite product to this class.

It will interact with the Seeded Interpolation Mapper to compose the initial terrain model, then with the Noise class to provide detailing of the terrain. Finally, there will be interaction with the Filters class in order to eliminate any anomalies of the terrain which may have been caused by the layering of algorithms.

4.4.2.6. Noise

The Noise component is the class which handles the storing of Perlin noise implementation logic. It serves as a functionality provider only, since the only methods available are related to creating Perlin noise, but it is probably unwise to create it as a global or static class since there is a state being considered when providing noise values based on given parameters. The different data values that make up a state include things like the gradient vectors being used in computation, hash maps which should simplify the processing and other such pseudo-random values which are the same for a certain instance.

4.4.2.7. Seeded Interpolation Mapper

The Seeded Interpolation Mapper is the class responsible for implementing the seeded interpolation algorithm used in the first step of the described process to synthesize terrain from crude heightmaps. It bears the weight of being input a model and warping it based on the seed points generated. There should be functional encapsulation, such that no part of the generation process is left to the outside.

This component should be input all the necessary information, like width, height and the like and the actual heightmap and model in order to warp the model based on the seed points. It is useful if the model being input is a plane with no altitude changes whatsoever, since this is the first step of the synthesis process and is designed to provide a basis for future adjustments. There is also the possibility of internally creating the model and serving the warped one as an output, but this case binds a responsibility, the creation of the model, to a class which should not have anything to do with that and should, hence, be avoided at all costs. This class is better used as an independent functional class, offering methods for model mesh manipulation based on the given input parameters.

4.4.2.8. Filters

The Filters component is the class responsible with storing all used digital filters. It serves as a repository for the noise-reducing filters used, mainly the mean filter and the median filter. Its use is highly beneficial because it allows further development in the area of used filters. The repository can be extended any time to provide more varied filters for the other classes to use without any kind of programming overhead. This extensibility is precisely what the class has been designed for.

This component, because it strictly offers behavior in the form of various digital filters, can be designed as static, or global, for the purpose of being accessible everywhere. The algorithms that are currently to be included are the mean digital filter, presented earlier, and the median digital filter, also presented in a previous section. Both of these filters are noise-reduction filters and are to be used by the Input Handler for the final step of processing to smoothen out the terrain and eliminate any blatant anomalies.

4.4.2.9. Seed Point

The Seed Point class is the smallest used class and it exists solely inside the Seeded Interpolation Mapper as a way to encapsulate the information required for seed points:

- Value / height. Needed for the actual interpolation between multiple points
- Position. Registered as a coordinate pair (for two-dimensional use, as the case is for this application). Used for the weighted interpolation as weights used in the computation.

The approach to create an auxiliary class to encapsulate this information, as opposed to simply using a matrix and noting inside the value, effectively intrinsically memorizing the position, was preffered because the number of seed points may be drastically lower than the total number of pixels and thus a lot of space would be wasted. Moreover, this encapsulation allows data handling and grouping in such a way that makes iterating over the seed point space and / or searching for specific points much easier than having to parse an entire matrix for each query.

4.5. Use Cases

The use cases present the ways the user interacts with the software application. Through them, developers can better form the user interface and design the system behavior to mirror that of the one presented through the use cases. Hence, these cases are essential for the proper development and should be taken into consideration throughout the entire development process.

4.5.1. Use Case Diagram

Presented in Figure 4.10, the Use Case Diagram presents graphically the most important use cases for this application. It serves as a developer entry point into the use case definitions and helps him or her familiarize with how the application should work and what the user should be able to do as an interaction with the user interface. The following use cases are pictured:

- Switch between Perlin and Input modes
- Input Heightmap
- Alter Parameters
- Force Perlin Redraw
- Export Model

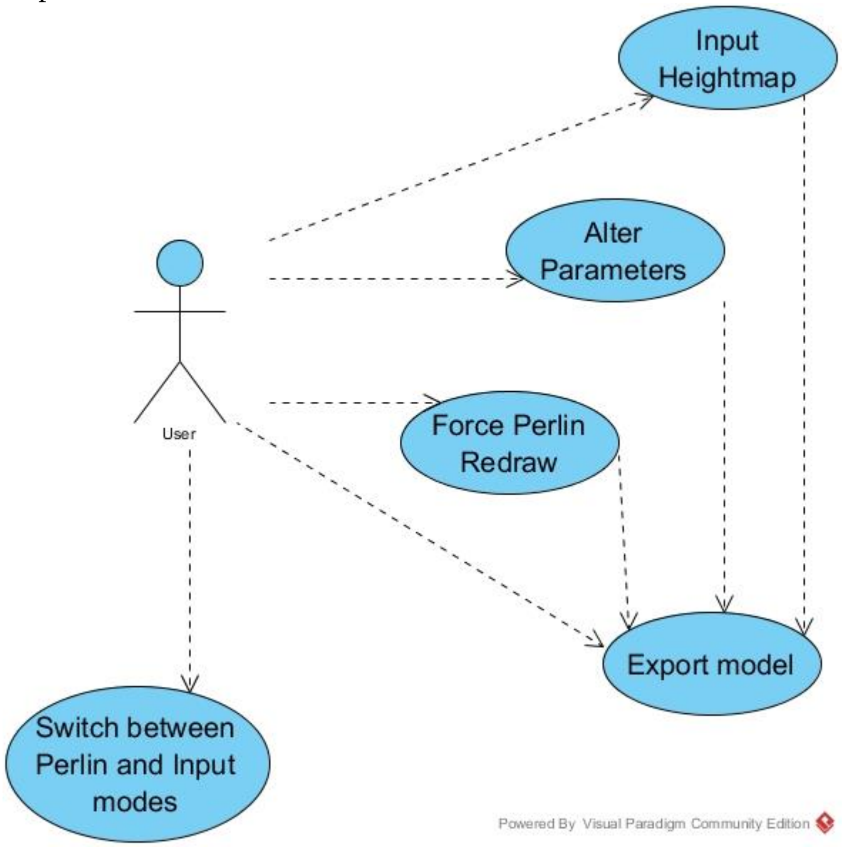

Input Heightmap
Alter Parameters
Force Perlin Redraw
User
Export model
Switch between Perlin and Input modes
Powered By Visual Paradigm Community Edition

Figure 4.10 Use Case Diagram

4.5.2. Switching between Perlin and Input modes

Main actor: the user
Purpose:

- User wants to switch between the two main application modes: Perlin and Input.

Preconditions:

- The application is running.

Postconditions:

- The application is in the mode opposite to the one it started in.

Main flow:

1. The user toggles the „User input” checkbox.
2. The system changes the user interface to reflect the change.

Extensions (alternate flows):

- none

Special requirements:

- none

4.5.3. Input Heightmap

Main actor: the user
Purpose:

- The user wishes to input a heightmap to the application.

Preconditions:

- The application is running.
- The application is in Input mode.

Postconditions:

- The file is loaded into memory and a terrain based on that input is shown on the screen.

Main flow:

1. The user inputs a file path into the path box.
2. The application reads the file from disk.
3. The application creates a terrain using the existing parameters.

Extensions (alternate flows):

- 2.a The path is invalid. No file is read. An error is displayed.
- 2.b The path is valid but the file is not. An error is displayed.

Special requirements:

- none

4.5.4. Alter Parameters

Main actor: the user
Purpose:

- The users wishes to change the parameters of the application to alter the existing terrain model.

Preconditions:

- The application is running.
- A terrain is being shown on screen.

Postconditions:

- The terrain model shown is altered, reflecting the parameter changes.

Main flow:

1. The user uses the interface to modify a parameter (through slider or textbox).
2. The user presses the „Apply” button.
3. The application alters the terrain based on the new parameter.

Extensions (alternate flows):

- none

Special requirements:

- The system graphics card must support 3D rendering at a fast-enough pace to not slow down the real-time operation.

4.5.5. Force Perlin Redraw

Main actor: the user
Purpose:

- The user wishes to see another Perlin noise generated terrain, different from the current one without changing the parameters.

Preconditions:

- The application is running.
- The application is in Perlin mode.

Postconditions:

- The displayed terrain is looking different than it did at the start of the flow. Parameters are kept the same since just the seed of the terrain was changed.

Main flow:

1. The user presses the „Redo” button.
2. A new seed is chosen and the new terrain is displayed on the screen.

Extensions (alternate flows):

- 2. The new seed is too close to the previous one and the terrain remains the same (the new terrain is identical or almost identical to the previous one).

Special requirements:

- The system graphics card must support 3D rendering.

4.5.6. Export Model

Main actor: the user

Purpose:

- The user wants to export the model currently being displayed into a format which can be used for external purposes, as originally intended.

Preconditions:

- The application is running.
- A model is being displayed on the screen.

Postconditions:

- A file containing the model or the heightmap of the model is present on the file system.

Main flow:

1. User presses the „Export” button.
2. The application saves the terrain data into a file.
3. A file is created on the disk and the data is copied into it.

Extensions (alternate flows):

- 3. No file is created due to lack of application privileges.

Special requirements:

- The application must have write-to-disk privileges.

Chapter 5. Technological Considerations

5.1. Unity Game Engine

The Unity game engine is a powerful tool for creating two-dimensional and three-dimensional applications. It is a software development IDE which focuses mostly on games and gaming-related applications. The interface is highly user-friendly, allowing newcomers to learn and handle it. It offers a large number of readily-built components and provides its users with a base rendering engine such that one is no longer concerned with the rendering pipeline. It is possible, though, to affect it but it is not necessary, like if building a graphic application from scratch. Figure 5.1 presents Unity's interface with an example project being already loaded and displayed on-screen.

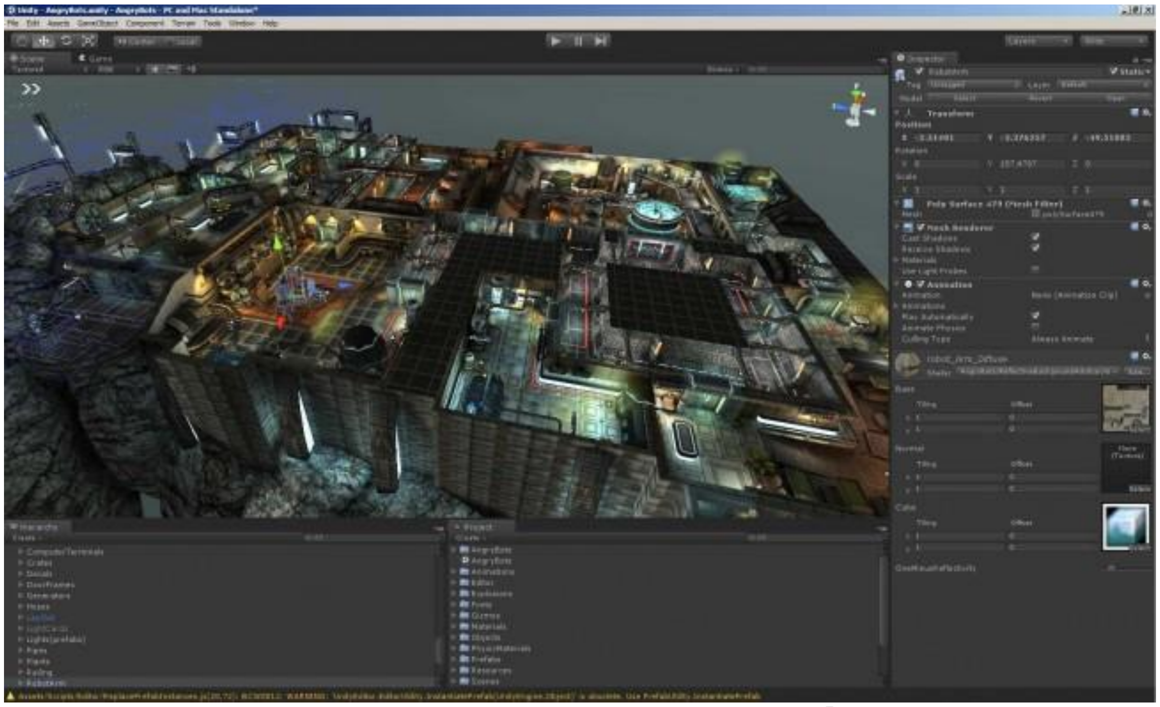


Figure 5.1 Unity Game Engine [5]

One can easily see the potential of such an application. Instead of visualizing only lines upon lines of code, the user is able to inspect and modify the entire three-dimensional space in real-time. This is also a useful feature when considering building the interface for the software system since it is possible to model and add to the interface on-the-go, without worrying about the code related to placing the interface objects.

[5] Courtesy of Liberty Voice: http://guardianlv.com/2014/10/unity-game-engine-reportedly-going-up-for-sale/

The main reasons for choosing Unity as the development engine behind the application are as follow:

- Ready-to-use cross-platform development. The Unity game engine offers project export into a wide array of executables suited for various operating systems and gaming platforms. I can easily export the application for systems like Windows, OSX and Linux without changing anything in the application itself. Unity will handle all compatibility issues. The only exception is using platform-specific code, like certain libraries available only on one operating system and not the other but this should pose no real issue if one avoids using such code.
- Available graphics processing pipeline. The rendering system is incorporated as part of the engine. This means there is no development effort to be spent on simply rendering the terrain model and interface on-screen. This is completely handled by the Unity Game Engine. It uses Direct3D for Windows and Xbox, OpenGL ES for Android and iOS, OpenGL for Mac and also for Windows and whatever proprietary APIs each system developer provides for gaming platforms and consoles.
- A scripting and component system which greatly simplifies the application fragmentation into objects, even though it is not fully utilized in this software system, it helped by removing much of the dreaded file manipulation existent in regular editors.

The main drawback of using Unity is the fact that three-dimensional model meshes are limited to 16000 vertices. On a square mesh, this is slightly less than a resolution of 256x256. This implicitly limits the resolution of input heightmaps in order to allow 1:1 processing of the image files.

5.2. GPU Computing

General-purpose computing on graphics processing units refers to using the graphics processing unit in processing of data outside the rendering pipeline. A graphics processing unit is different from a central processing unit by having lower frequencies but a large number of cores. Significantly larger than that of central processing units. This makes certain data processing in parallel much, much faster if the processing algorithm respects the requirements given by the graphics processing unit the work will be placed on.

The Unity game engine allows one to harvest the power of the graphics processing unit through compute shaders. This kind of shaders is not yet generally available on most systems, but it facilitates parallel data processing. Depending on the algorithm, parts or even the entire processing block can be moved to the graphics card in order to be parallelized. This leads to experimental code which tries to take advantage of the GPU processing power.

Chapter 6. Detailed Design and Implementation

This chapter describes the actual implementation of the application, using the design mapped onto the Unity game engine specifications.

6.1. System Architecture

6.1.1. Class Diagram

The class diagram shows the software system structure moulded on the Unity game engine scripting system. It is a more detailed diagram than the conceptual architecture since it displays the components in their final state, alongside the more important methods available inside each class. Figure 6.1 illustrates this diagram.

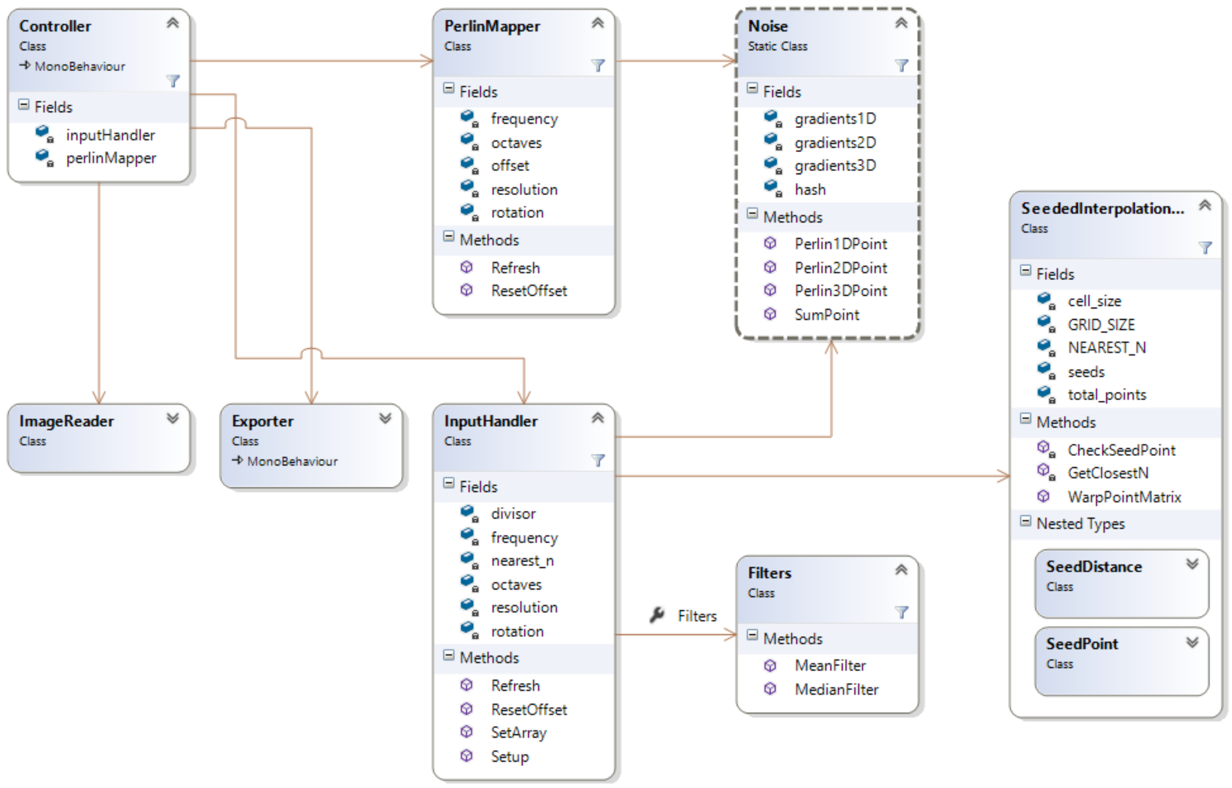


Figure 6.1 Class Diagram

Note the lack of user interface components. This is an artifice implemented in order to reduce wasted space due to the large number of interface elements like sliders, text boxes and buttons. One may refer to the conceptual diagram to find out that the lynchpin of the system is the Controller class, which binds all user interface elements together and sends the commands and parameters down to the other classes, like the Perlin Mapper, the Input Handler, the Exporter and the Image Reader.

6.2. Component Implementation

This sub-chapter describes the implementation of the components shown in the class diagram (Figure 6.1) and elaborates on the functionality of each class independently and on the relationships between them. Briefly, there is also a motivation description which argues why the class exists and is useful as a part of the whole software system.

6.2.1. UI

The user interface is an umbrella term which gathers all the user interface classes together. These classes are used, sometimes in multiples, to provide the user means to configure the application parameters. Composing the user interface, the following classes are being used:

- Button: used for the implementation of buttons like „Load file”, „Redo”, „Apply Changes” and „Exit”.
- InputField: used for allowing the user to input various strings or values, like the file path or various parameter values.
- Slider: used to visually represent the parameter values and mirror them alongside the input fields.
- Panel: used to visually delimit various spaces, like the one which contains general interface elements, the one containing user interface elements pertaining to the Perlin noise and the one containing user interface elements containing elements controlling the seeded interpolation algorithm parameters.
- Text: used to visually transmit various static messages to the user. The text location is of relevance to the pertinence of the message by binding the text message to a particular panel or another.
- Toggle: essentially check boxes, they allow the user to control features which only have two states, like the User Input / Perlin modes.

All the user interface elements are of a type presented in the aforementioned list. Most of them are active throughout the application’s lifetime but a select few, representing the User Input mode or the Perlin terrain mode, can be activated or deativated based on the currently active generation mode.

6.2.2. Controller

The Controller class is the most important class, interaction-wise, of the entire software system. It makes the connection between the user interface elements and the functional components of the application. This class contains a reference to all relevant user interface element instances, like input fields, buttons and sliders, and maintains the logic of input handling. It contains all the methods pertaining to handling the “OnClick” and “OnValueChanged”events for all those user interface elements. Whenever the user clicks one of the buttons or changes the value of one of the sliders the relevant method is called (a “callback”) and the Controller handles the values by sending them to the classes which do all the important processing, like the Exporter, Image Reader, Perlin Mapper or the Input Handler.

The Controller contains references to all important user interface elements but, most importantly, it contains actual instances of the aforementioned important processing classes, most notably the Input Handler and the Perlin Mapper. It relays parameter information to the Input Handler and the Perlin Mapper depending on the active mode through the use of conditional coding inside the callback methods for the user interface components. It also retrieves information from the Perlin Mapper and the Input Handler and relays it to the Exporter when the user signals the want to export the existing model. Moreover, it sends instructions to the Image Reader to read the given input image in the case of the Input Handling mode of the application from the given file path.

There is also the entire implementation code for the mode switching flow and data throughput. This is, ultimately, what the Controller class is for: being the main manager for the application, handling all the important logic, making sure the resources are properly managed and everything is processed in the way the user wishes it to be.

6.2.3. Image Reader

The Image Reader is an auxiliary class whose only main purpose is reading images. The two main methods it contains are ResourceImageToArray and FileSystemImageToArray. ResourceImageToArray is a method which reads images from the project’s resources. It is mainly used to load images during testing, since then is the easiest time to load images to the resources of the application. The user will probably never trigger this method, since the main usage for users is to load files from the filesystem, loading images into the application’s resources would be quite hard and useless, since the effect would virtually be the same as loading the image directly. The image is loaded and transformed into an array of float values, retaining the entire image pixel values into the floats.

The method FileSystemImageToArray is a method used to read files from the filesystem using the C# methods from System.IO. It will be invoked to read .png and .jpg files from disk; any other kind of image file type is not suited for the methods used. These files are handed over by the user through their absolute paths on their filesystem. This is the primary class which makes the user input part of the application work by pinpointing the files on the user’s filesystem. It also produces arrays of float values which mirror the pixel value of the input image one-to-one, provided that the image was provided in the required format and color scheme.

6.2.4. Exporter

The Exporter is another auxiliary class whose main purpose is the reverse one to that of the Image Reader. It is a class which receives the data of the generated terrain model and creates a file where it writes the data. Currently the exporter only exports heightmaps as .jpg or .png files but it is entirely possible to extend the class to allow the export of three-dimensional models. The exporter receives a matrix of float values as input data and an absolute path representing the final destination and filename of the exported heightmap. It does not modify the data in any way, preserving a one-to-one relationship between the input floating point value matrix and the output pixels.

6.2.5. *Perlin Mapper*

The one and only job of the Perlin Mapper is to shape the mesh based on the local implementation of the Perlin noise algorithm. It does this by preserving a reference to the rendered mesh and modifying the height values of all vertices based on the given Perlin noise value for that specific set of coordinates. Since the mesh reference is kept throughout the application lifecycle, there is no need for heavyweight communication with the Controller class. Only the important parameters are communicated and stored inside the Perlin Mapper:

- Frequency – defines the number of "segments" traversing the mesh, just like the frequency for a wave function. The Perlin noise can, in fact, be described like such a function. This parameter influences how "much" noise appears on the terrain. Low values result in large-ish shapes molded onto the terrain while greater values increase the noise granularity as distributed across the mesh
- Octaves – the octaves represent how many "sub-layers" the Perlin noise will have. Each octave after the first halves the effect is has on the terrain but doubles the frequency. This way several Perlin layers can overlap, creating a better and more detailed version of noise.
- Offset – the coordinate offset added to all points. Since Perlin noise is the same at the same point in the plane, this offset permits the "redraw" of Perlin noise by its change and, hence, the virtual repositioning of the model mesh on the plane.
- Rotation – fulfills the same role as the offset by rotating the Perlin noise plane in three dimensions. This rotation obviously drastically changes the available values. Used together with the offset it allows for powerful reusability options regarding the same Perlin noise plane, sampled at different point and from different angles.
- Resolution – the resolution of the mesh. Currently only square meshes are considered, hence the total number of vertices to be used will be resolution * resolution. This parameter is stored inside the Perlin Mapper because it morphs the mesh and should, hence, be able to also modify the total number of available vertices.

These parameters control the Perlin noise itself and, consequently, the shape of the mesh being rendered. To aid in the real-time modification of the mesh, two methods are being used as ways of interfacing with the Controller.

- Refresh – this method is used as a signal to redraw the entire model based on the new requirements. The Refresh method receives as input the full set of parameters, such that no matter which parameter is being changes, one does not need to vary the method used. The changes are reflected instantly on the model, since there is little processing to be done.
- ResetOffset – this method is called whenever the Controller signals a need to change the position and rotation of the Perlin noise. The change is immediate, since a new pair of position and rotation values are created and it is reflected in the mesh instantly.

6.2.6. Input Handler

The Input Handler is the “sister class” of the Perlin Mapper and handles the responsibility of morphing the terrain based on a given input matrix of floating point values. This input matrix is the result of processing the input file given by the user and read by the ImageReader. This matrix of floating point values provides a base upon which the class improves upon using the Terrain Synthesis from Crude Heightmaps algorithm.

This class implements and applies all the steps of the algorithm to the model, from the randomly seeded linear interpolation to the filtering effects. Just like the Perlin Mapper, this class also keeps a reference to the model mesh and modifies it on command. Because it incorporates Perlin noise as the detailing step, there is some overlap regarding the important parameters it acts upon:

- Divisor – refers to the number of seed points being used by the interpolation algorithm. The divisor is the divisor by which the total number of pixels is divided to obtain the number of seed points. Hence, a divisor value of 25 refers to using 1 seed point for every 25 pixels (Nr. pixels / 25).
- Nearest_n – refers to the number of nearest neighbors taken into consideration during the linear interpolation step. N is the number of seed points which participate in the interpolation using their distances to the target point as weights for the calculation.
- Frequency – defines the number of “segments” traversing the mesh, just like the frequency for a wave function. The Perlin noise can, in fact, be described like such a function. This parameter influences how “much” noise appears on the terrain. Low values result in large-ish shapes molded onto the terrain while greater values increase the noise granularity as distributed across the mesh
- Octaves – the octaves represent how many “sub-layers” the Perlin noise will have. Each octave after the first halves the effect is has on the terrain but doubles the frequency. This way several Perlin layers can overlap, creating a better and more detailed version of noise.
- Offset – the coordinate offset added to all points. Since Perlin noise is the same at the same point in the plane, this offset permits the “redraw” of Perlin noise by its change and, hence, the virtual repositioning of the model mesh on the plane.
- Rotation – fulfills the same role as the offset by rotating the Perlin noise plane in three dimensions. This rotation obviously drastically changes the available values. Used together with the offset it allows for powerful reusability options regarding the same Perlin noise plane, sampled at different point and from different angles.
- Resolution – the resolution of the mesh. Currently only square meshes are considered, hence the total number of vertices to be used will be resolution * resolution. This parameter is stored inside the Perlin Mapper because it morphs the mesh and should, hence, be able to also modify the total number of available vertices.

The class also has some overlap with the Perlin Mapper regarding the public methods used for controlling the parameters and the model generation. These methods are invoked by the Controller to reshape the terrain model. Care must be taken not to invoke these methods too often, since redrawing the entire terrain model by applying the terrain synthesis algorithm is a costly operation, notably the randomly seeded linear interpolation with its nearest N neighboring seed points search. This is why it is recommended to let the user consciously direct the software system to "apply the changes". Following is a list of the main methods shared with the other components:

- SetArray – this method is used to shape the base terrain using the input floating point value matrix representing the user file. This crude heightmap provided by the user is memorized inside the class for easy access to its data. It is also useful because of how it triggers the mesh formation, paving the way for future Refresh method calls.
- Setup – this method serves as an initial entry point for the usage of the Input Handler. Setting up the parameters allows the software system to manage the terrain model mesh better and to avoid pitfalls like trying to refresh inexistent models with blank parameter values.
- Refresh – this method is used as a signal to redraw the entire model based on the new requirements. The Refresh method receives as input the full set of parameters, such that no matter which parameter is being changes, one does not need to vary the method used. The changes are reflected instantly on the model but it is a costly operation and it takes time to reshape and redraw the model. This means that, unlike the Perlin Mapper, the Input Handler's Refresh method needs to be called sparingly.
- ResetOffset – this method is called whenever the Controller signals a need to change the position and rotation of the Perlin noise. The change is immediate, since a new pair of position and rotation values are created and it is reflected in the mesh instantly. This method also takes a larger amount of time to be executed. Thus, it should also be called sparingly.

6.2.7. Noise

The Noise class contains the implementation of Perlin noise used by both the Input Handler and the Perlin Mapper. It works with a predefined list of gradients and is capable of providing Perlin noise values in three dimensions (one-dimensional noise, two-dimensional noise and three-dimensional noise). Only the three-dimensional noise is used by the software system. Likewise, it contains linking to compute shaders which allow the computation of multiple Perlin noise values in parallel on systems with DirectX11 or greater and Windows OS but since it is restricting it is *not a feature* used by the application. This feature is powerful enough, though, that it bears mentioning since this class can be reused in other projects to its full potential.

The four main methods offered by this class as ways of obtaining Perlin noise values are:

- Perlin1DPoint – provides noise values when given one-dimensional coordinates as a parameter.
- Perlin2DPoint – provides noise values when given two-dimensional coordinates as a parameter.
- Perlin3DPoint – provides noise values when given three-dimensional coordinates as a parameter.
- SumPoint – provides a composite, "layered" three-dimensional noise value with multiple octaves.

These methods all take the full range of necessary parameters in addition to the coordinate, like the frequency and the octaves values needed as part of handling the Perlin noise implementation itself.

Most of the code for this class has been adapted from Catlike Coding's Unity tutorial [6] on Perlin noise because that implementation's quality is unquestionable, doubled by the powerful parametric variation controls the code allows. Moreover, a good implementation of the Perlin noise saves time if checked for errors and handled properly, as this one was by me.

6.2.8. Seeded Interpolation Mapper

The Seeded Interpolation Mapper class is responsible for managing the randomly seeded linear interpolation of the first step of the terrain synthesis from crude heightmaps algorithm. It implements the entire step, from the generation of seed points across the entire image up to warping the mesh based on those seed points. The seed points are, as per the algorithm, placed randomly across the entire surface and they receive their value from the underlying image given as a two-dimensional matrix of float values. As an efficiency improvement, the seeds are assigned to "cells" which are nothing more than divisions of the problem space to allow a faster nearest-neighbor search among the seed points.

Conducive to the algorithm, the following parameters are used within the class:

- Cell_size – used as an auxiliary value to determine the size of the cells containing seeds. Allows for faster distribution and selection of seends inside cells.
- Grid_size – the number of cells across the grid. The cell size can be computed after finding out the grid size. This effectively represents the size of the cell matrix.
- Nearest_n – the number of neighbors to search for during the interpolation step. This value is respected as is: the "nearest_n" seeds will be searched for and used in the computation of the target point's value using the distance from the target point to these seed points as weights for determining the influence of each seed's value towards the final value of the target point.
- Seeds – the actual seed point matrix, characterized by grid_size, defining the size in number of cells across, considering a square grid of cells. The cell_size value represents the width of a contained cell, summing up to a total of 1.0. This repository of seed points is parsed for every target point

[6] Catlike Coding, "Noise" : catlikecoding.com/unity/tutorials/noise

on the image plane, meaning this is a section which warrants the most optimization effort.

- Total_points – is simply the number of total seed points to be generated and distributed across the problem space. This number may vary as set by the classes using the Seeded Interpolation Mapper.

There are few methods of control, mainly just setting up the class and then invoking the proper method to warp the mesh into a new form by using the linear interpolation between the existing seed point distribution. Some of the most important and relevant methods of this class are:

- CheckSeedPoint – a method which, given a seed point, compares it to the other N seed points to see if it is closer than any of them, then proceeds to replace the seed point if one such point is found in the seed point group being saved at that moment.
- GetClosestN – a function which, given a target point, returns a list of the closest N seed points to that point, where N and the target point are given. This function calls CheckSeedPoint on all seed points being considered, delegating the actual check and replace actions to a lower level.
- WarpPointMatrix – this function is a public method which triggers the Seeded Interpolation Mapper class to modify the given mesh by an interpolation of the generated seed points. It calls the GetClosestN method for each point on the image then interprets the resulting list by linearly interpolating the values of the seed points based on their distance to the target point.

This class declares two sub-classes which help it keep the data organized and avoid using ambiguous tuple variable declarations. This helps keeping the code integrity and cleanliness. The auxiliary classes are:

- SeedDistance – this class contains a reference to a seed point and the distance from it to the current target point being evaluated. This class keeps the developer from tracking two separate arrays containing the seed points and their distances and allows for easier sorting, if the need ever comes.
- SeedPoint – the actual seed point. This class contains a value variable, which is simply the value the seed point takes based on the underlying image, and the position of the seed point in two dimensions. The third dimension is irrelevant for the purposes of this algorithm hence only two dimensions are taken into account and used for computing the interpolation.

6.2.9. Filters

The filters class implements the digital filters used by this project to smoothen out the terrain once all details have been placed and computed. It is a repository class which has no state and only static methods, hence it is more like a collection of filters one can use in the detailing of the terrain. The available filters are:

- MeanFilter – a filter which replaces the center pixel with the mean (average) of the surrounding nine pixels.
- MedianFilter – a filter which replaces the center pixel with the fifth pixel as ordered ascendingly

6.3. User Interface

This section describes the user interface and its elements. A user interface is one of the building blocks of a good application and one cannot underestimate the need for a good interface which allows the user to interact with the application itself and provide instructions on how the software system should operate. The interface was built to be as simple as possible such that the layman can easily use it without any training or too great of a learning curve.

6.3.1. Interface as a whole

The interface as a whole is presented in a simplistic manner with as few controls as possible. The main objective is to not flabbergast the user needlessly, but to let him or her experience the terrain generation with as little confusion as possible. The controls are grouped together in groups which try to represent the common functionality as well as possible. Figure 6.2 displays the interface in the user input mode, with all the sliders abd buttons. Note the existence of the „Load File" area and the lower „Apply Changes" button and the associated sliders. These will be unavailable in the Perlin mode but since the user input mode is the main mode, these controls are important to the general presence of the application.

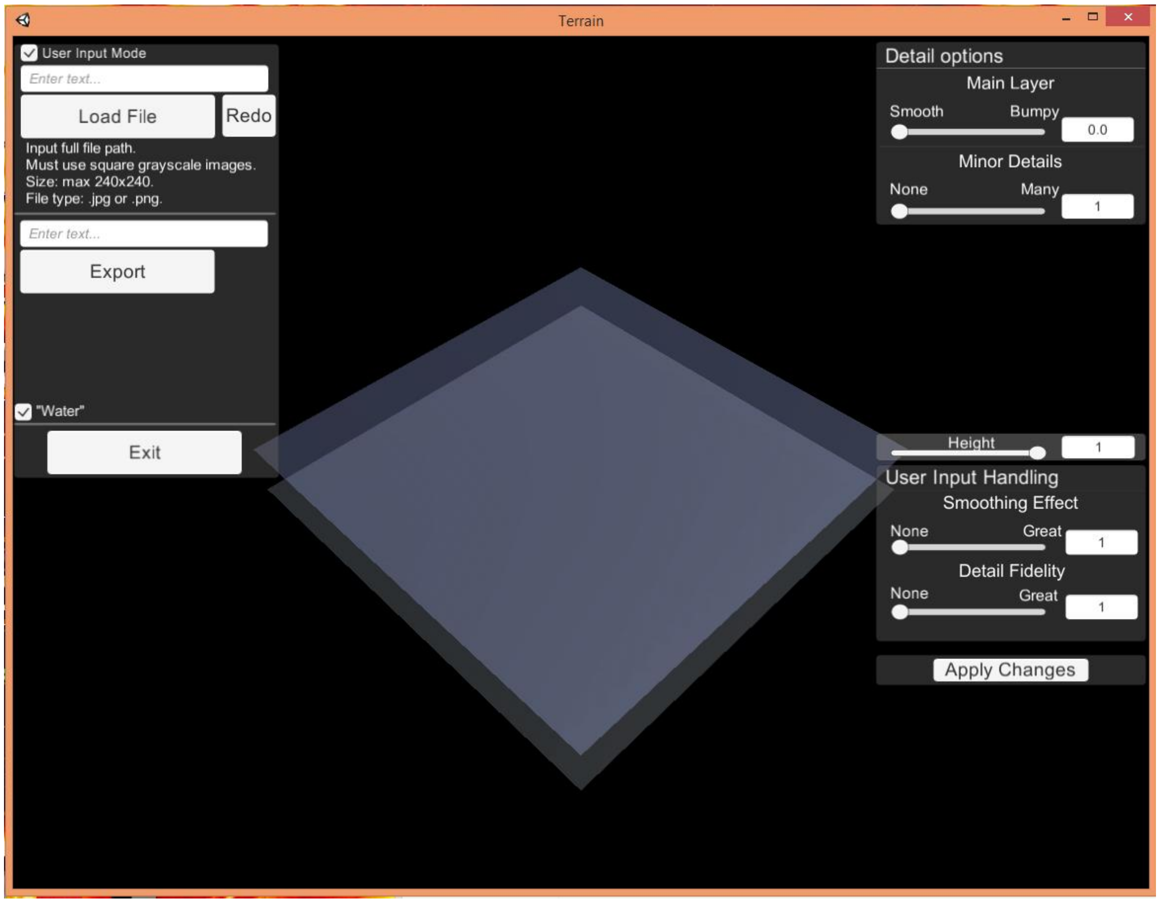


Figure 6.2 User Input Mode

One will notice that the Perlin mode of the application does not change many of the controls. The control overlap is intentional, as it allows the user to remain familiar with the interface, even when switching between modes. There are still enough differences to let the user be aware of a change of functional modes. Notable changes include the disappearance of the bottom-right panel with sliders and the complete change of the top-left panel. Instead of having a file import control, it now simply allows the user to modify the resolution of the current terrain.

As one may infer from the control overlap, Perlin noise is used as the last step in the algorithm for shaping user input into a working terrain model. Thus, the terrain model presented in the Perlin mode is simply an isolation of the Perlin noise values across the entire plane, magnified to look as terrain. This was a design choice in order to reuse the Perlin noise functionality and allow users to work with something simple like a Perlin terrain generator. This idea is neither new nor implemented to the best of its capabilities. It is just a by-product of the main application functionality and presented as such. Hence it only occupies a secondary position for the user, being inactive at the start of the application. Figure 6.3 presents the full Perlin mode interface.

It should be noted that the user may switch between these two modes by toggling the „User Input Mode” toggle control in the top-left corner of the interface and all the interface changes are made instantly, together with wiping the terrain clean.

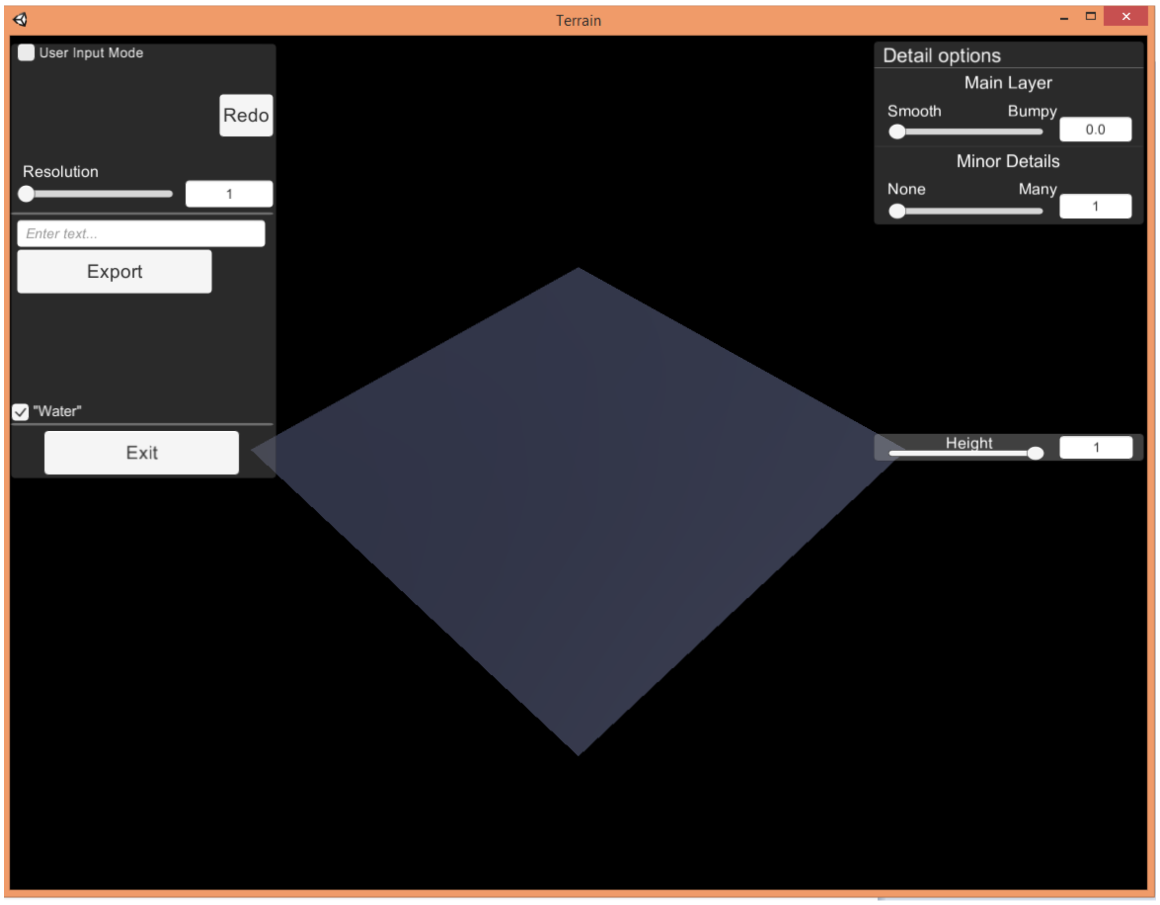


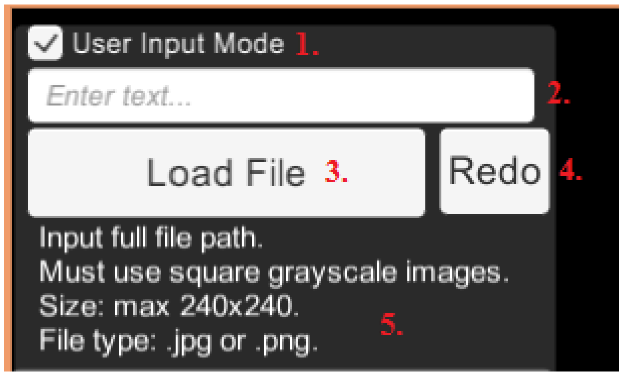
Figure 6.3 Perlin Mode

6.3.2. Interface Regions

This sub-section describes each section in slight detail, such as to acquaint the prospective developer with the different control elements from the user interface regions. This allows for a deeper understanding of how each region works and what its general purpose is, even if some are quite vague by nature. The vagueness occurs because not all controls are tightly-bound together purpose-wise yet one should not scatter mindlessly them across the entire screen, but group them in such a way as to still allow the user ease of understanding and, most importantly, of use.

6.3.2.1. User Input Region

The User Input Region is the region responsible for allowing the user to present his input heightmap image file to the application in order to be processed. Hence, it is the first stepping-stone of the User Input mode of the software system and needs to be lightweight and easy to use. Do note the lack of a file browsing capability. The choice of an absolute file path was selected due to some of Unity game engine's restrictions, getting over which would have increased exponentially the development time. This is, thus, a middle-ground between the best solution, a file browser, and the worst solution, no user input capabilites, which tries to alleviate the development effort. This region is exemplified in Figure 6.4.



Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.

Figure 6.4 User Input Region

The following elements make up the User Input Region:

1. The mode toggle control – it allows the user to effortlessly switch between the two main functional modes of the application: User Input mode and Perlin mode. User Input mode is activated whenever the checkbox is checked and, thus, the toggle is active. Perlin mode is likewise active only whenever the checkbox is not checked and the toggle is inactive.
2. The input file path control – it servers as the focal point for the user's input introducing actions by offering a space into which the user can input the file path of the file to be analysed. The text box is actualised in real-time however the string is not read until the „Load File" button is pressed. The text box can safely and gracefully handle space characters, unlike some other software.
3. The „Load File" button – used by the user to signal that he or she would like to load the file positioned at the absolute path contained within the input file

path control. This button, alongside the input text box, disappears whenever the application switches to Perlin mode.

4. The „Redo” button – can be used by the user to redo the Perlin noise. This is the only way the application allows the user to reseed the noise. What this means, as described in the previous chapters of this paper, is simply changing the point position offset and the rotation of the points being considered for Perlin noise.
5. The information box – contains useful information for the user to properly choose and submit the heightmap image files.

6.3.2.2. Detail Layer Region

The detail layer region is a region responsible for handling input regarding the Perlin noise layer, regardless of the mode. This region, however, masks the actual purpose of the input with more user-friendly descriptions of what the Perlin noise parameters actually do. This region is presented in Figure 6.5.

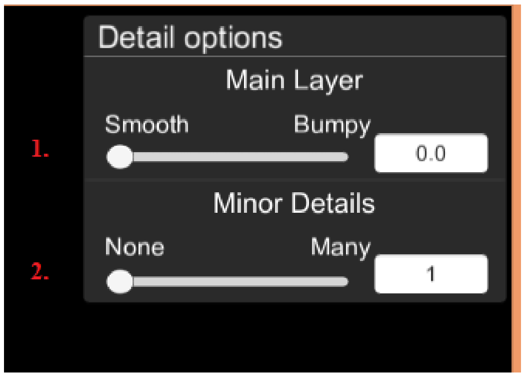


Figure 6.5 Detail Layer (Perlin) Region

The Detail Layer region is composed of the following controls:

1. The Main Layer sub-region – contains a slider and a text box mirroring each other which control the frequency of the Perlin noise. The ends of the spectrum are labeled „Smooth” for a lack of Perlin noise, or low-frequency noise, which translates to large, wavy forms, and „Bumpy”, which translates to a high frequency noise and smaller yet more numerous forms. The description text, appropriately labeled „Main Layer” is called so due to the frequency controlling the underlying detail layer of the Perlin noise.
2. The Minor Details sub-region – contains a slider and a text box mirroring each other which control the octaves of the Perlin noise. The ends of the spectrum are labeled „None”, since one octave represents just the top frequency of the noise, and „Many”, since each octave after the first means adding another Perlin noise layer at half strength but double frequency.

6.3.2.3. Input Handling Region

The Input Handling region is the region responsible for adjusting parameters relevant to the user input mode. These parameters are those directly influencing the randomly seeded linear interpolation step, namely one of the parameters represents the number of neighboring seed points being taken into consideration for the interpolation and the other parameter represents the total number of seed points when compared to the total number of pixels. The Input Handling region also contains the height control for the image. This control remains active during both of the application's functional modes while the rest of the input handling-specific controls become inactive when Perlin mode is toggled on.

The control elements of the Input Handling region, shown in Figure 6.6, are:

1. The height control – a slider with its mirroring text box controlling the maximum height of the terrain. Varying this parameter effectively scales up or down the terrain on the vertical axis, turning it flat on a low setting and curvy again on a high setting. This is the only control to remain active during the Perlin mode's activation.
2. The Smoothing Effect region – it vaguely describes something as „smoothing effect”. This is, in fact, a direct correlation to the number of seed points taken into consideration for the linear interpolation algorithm. As the number of seed points increases, so does the smoothing effect. The two different labels attached to the slider accompanied by a text box are called „None” and „Great” since it is an uncountable value parameter.
3. The Detail Fidelity region – it is associated with the number of total seed points used in the linear interpolation. It is labeled „Detail Fidelity” because as the number of seed points increases, more detail from the input heightmap image is preserved. The two tags are, again, „None” and „Great”.
4. The „Apply Changes” button – this button is used to confirm a terrain model update with the new options. Since the transformation is costly from an operational perspective, the design decision to prevent instant updates was taken, and this button was introduced.

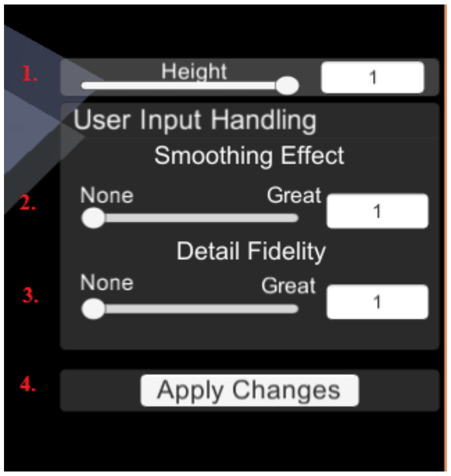


Figure 6.6 Input Handling Region

6.3.2.4. Auxiliary Region

The Auxiliary region is a region bearing no strong cohesion between elements. It is a leftover region used to gather and group together the elements which would otherwise have to be delegated into standalone regions. The Auxiliary region, as presented in Figure 6.7 currently contains the following controls:

1. The export file path field – this is the control where the user specifies the destination file for the heightmap export. This field is the counterpart of the input file path field used by the user to specify the input heightmap image used as a base for the terrain model to be synthesized.
2. The „Export” button – the control used by the user to signal that he or she successfully wrote the absolute file path of the export heightmap image and that the software system can now create that file and export the heightmap.
3. The „Water” toggle – this control is an auxiliary control which toggles a superficial water place on or off. This was added as a purely aesthetic addition to the application to signal in a very rough manner where a water level might be located. Since it may or may not be desired, the option to hide it was added through this toggle control.

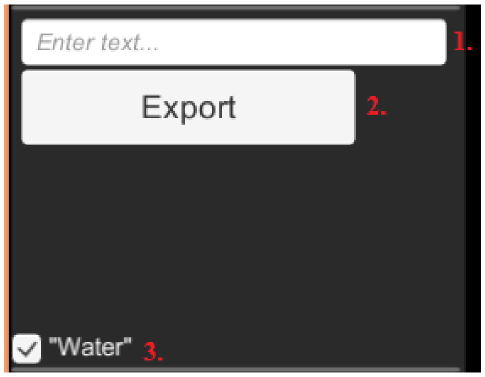


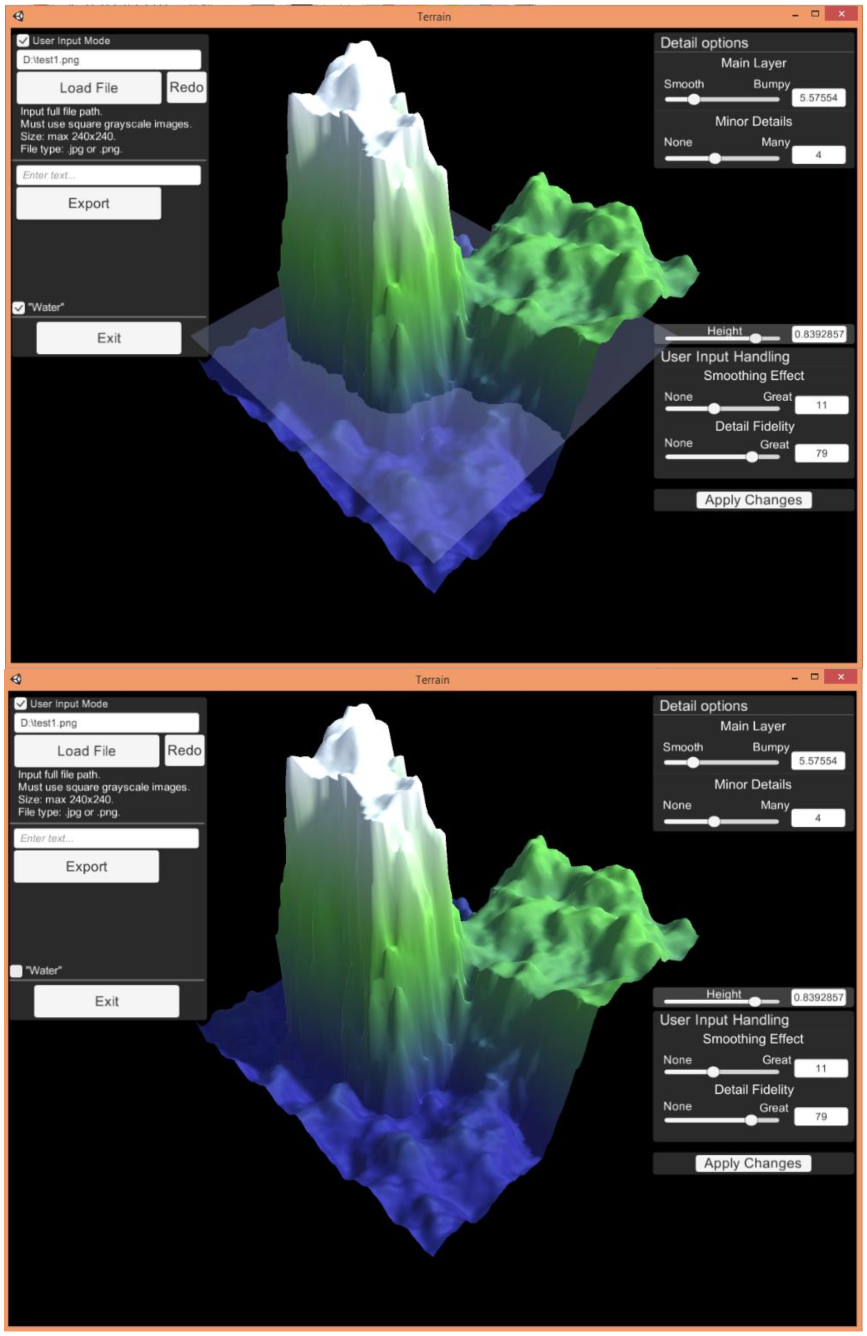
Figure 6.7 Auxiliary Region

6.3.3. Usage Examples

This part of the paper exemplifies how the interface looks during the different modes of the application, showcasing a terrain model to display exactly what a true user would experience while using this software system. This can be considered as a very brief walkthrough through some of the presented features of this application. The modes are specified in the sub-chapter titles. These have no direct influence over what is described. They simply validate the active interface elements. Since User Input mode is the main mode, most examples have been taken from a User Input mode flow.

6.3.3.1. Water Layer (User Input Mode)

The water layer is used to simulate the look of water such as to help the user visualize how his or her terrain would look with the lowermost parts submerged underwater. Figure 6.8 displays the same user interface with an identical terrain model. In the top picture, the water layer is active and one can see the transparent bluish layer near the bottom of the picture. The bottom image displays the same terrain model, only with the water toggle deactivated. One can now see that the transparent water later near the bottom is no longer visible. This allows the user to choose how he or she should see the terrain: with or without water pockets. The feature may help tremendeously should the user ever try to display marine landscapes, such as islands or water-bordering mountains.



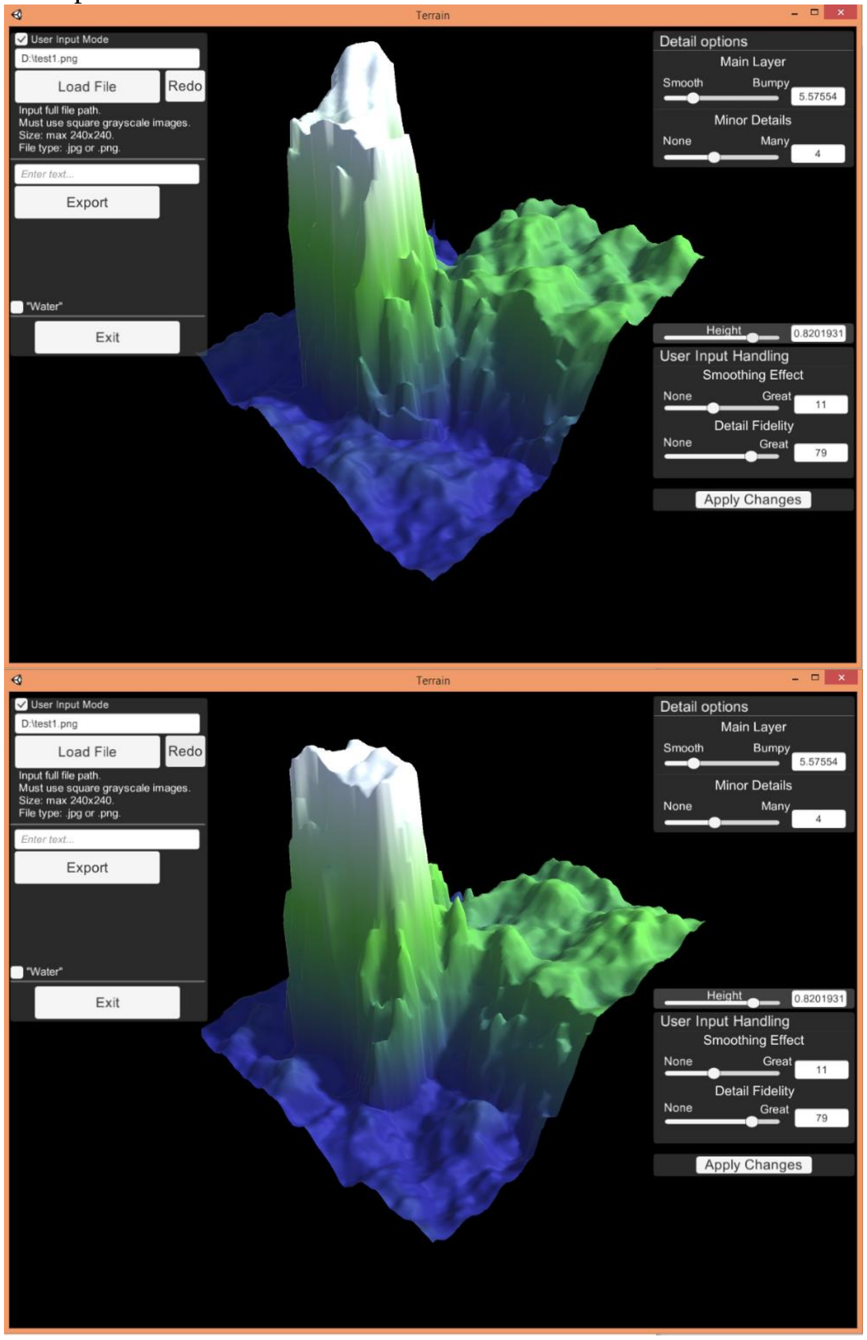
[✓] User Input Mode
D:\test1.png
Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.
Enter text...
[✓] "Water"
Main Layer: Smooth – Bumpy 5.57554
Minor Details: None – Many 4
Height 0.8392857
Smoothing Effect: None – Great 11
Detail Fidelity: None – Great 79

[✓] User Input Mode
D:\test1.png
Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.
Enter text...
[ ] "Water"
Main Layer: Smooth – Bumpy 5.57554
Minor Details: None – Many 4
Height 0.8392857
Smoothing Effect: None – Great 11
Detail Fidelity: None – Great 79

Figure 6.8 Example User Input - Water

6.3.3.2. Redo Button (User Input Mode)

The „Redo” button is a button allowing the user to reset the randomness of the terrain model. Whilst one cannot in this current implementation specify the seed, this gives the user some control over the overall look, allowing him or her to „try again”, hoping the next random values will suit his or her purpose better. Figure 6.9 demonstrates the use of the „Redo” button. The top picture presents the model beforehand, the bottom picture presents the terrain model after the button press. One can notice the difference in details between the two illustrated models, even though the parameters on the right-hand side are identical in both pictures.



User Input Mode ✓
D:\test1.png
Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.
Enter text...
"Water"
Detail options
Main Layer
Smooth Bumpy 5.57554
Minor Details
None Many 4
Height 0.8201931
User Input Handling
Smoothing Effect
None Great 11
Detail Fidelity
None Great 79

User Input Mode ✓
D:\test1.png
Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.
Enter text...
"Water"
Detail options
Main Layer
Smooth Bumpy 5.57554
Minor Details
None Many 4
Height 0.8201931
User Input Handling
Smoothing Effect
None Great 11
Detail Fidelity
None Great 79

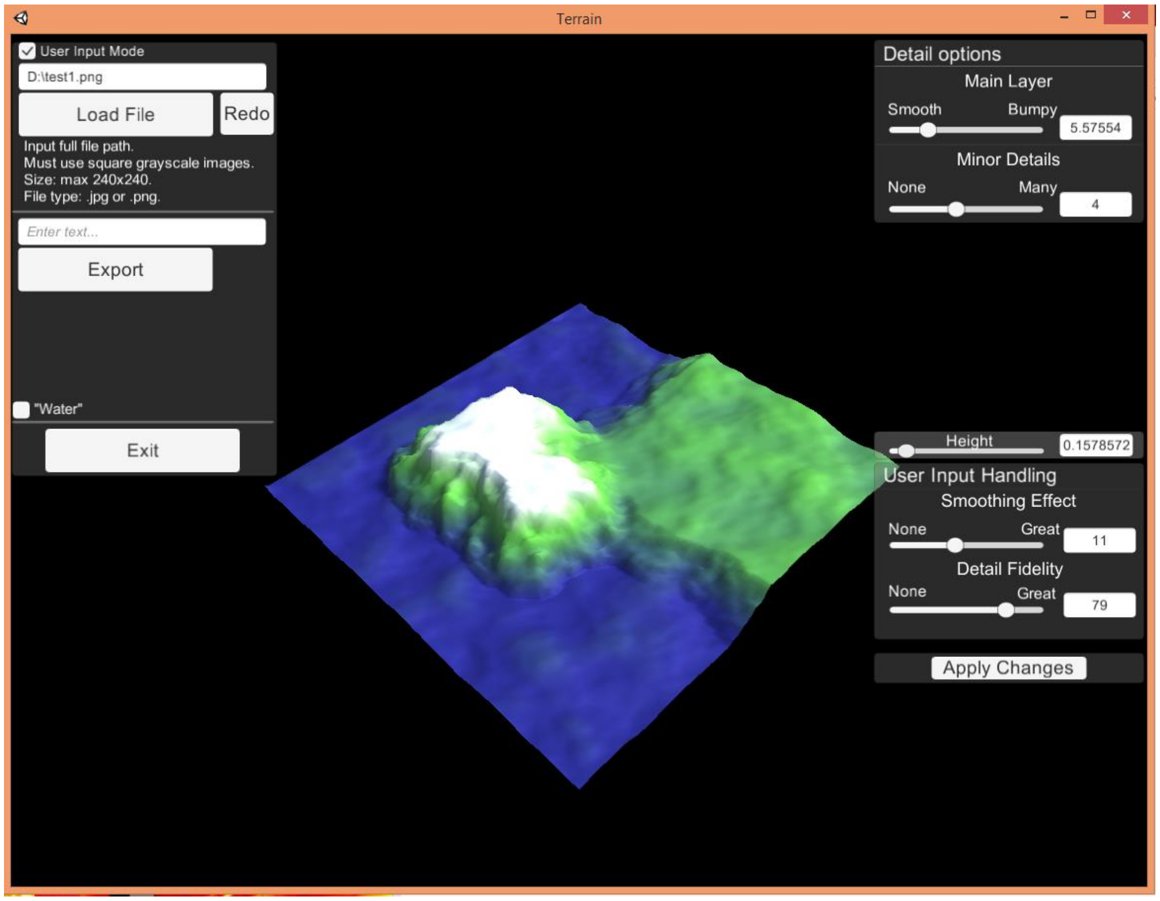
Figure 6.9 Reference Input - Redo

6.3.3.3. Height control (User Input Mode)

The height control, represented by the slider coupled with a textbox mirroring it, control the height difference between the top-most and the lowest points of the terrain model. As one would have noticed, in the previous examples the “Height” value was set at 1.0, displaying the default height property value. Figure 6.10 shows the same terrain model, but with a 0.15 height value. It is very noticeable that the model now appears flattened, yet it still looks and feels like a terrain model, albeit of a different type. Thus, the height value, while seemingly innocuous, brings a new dimension to the application, completely changing the generated terrain look and feel.

The height control permits any user to fine-tune the obtained terrain and get more detail across the height value spectrum from black through the shades of gray to white and focus that on a shallow region. Hence, the user does not need to use shades of gray close to black to model a low-height model and can focus on the overall shape and appearance by flattening the terrain model afterwards, through the use of this very control.

This makes the application more versatile and user-friendly, allowing one to adjust the vertical scale and change the overall appearance of their synthesized terrain model. This happens through a procedure so simple, the returns are amazing, considering the extremely low cost of changing the model scale. The Unity game engine makes this task ridiculously easy and efficient, hence, this task can be performed real-time with no need for user change approval beforehand.



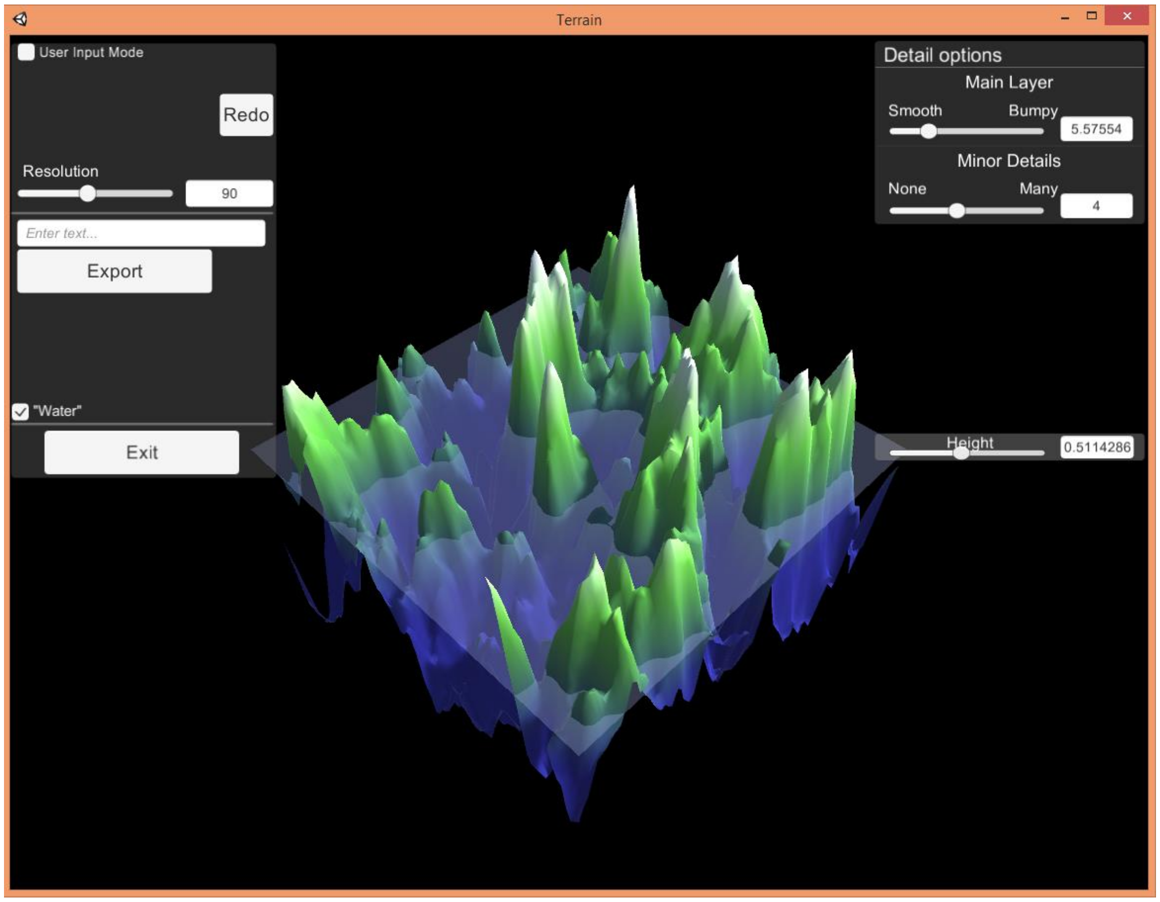
☑ User Input Mode
D:\test1.png
Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.
☐ "Water"

Main Layer: Smooth – Bumpy: 5.57554
Minor Details: None – Many: 4
Height: 0.1578572
User Input Handling
Smoothing Effect: None – Great: 11
Detail Fidelity: None – Great: 79

Figure 6.10 Input Sample – Low Height

6.3.3.4. Perlin Mode

This section displays in Figure 6.11 the application executing in Perlin mode. This is the secondary more which allows the user to play around with a terrain model generated strictly through Perlin noise rendering. This, as opposed to the User Input mode, permits anyone to somewhat utilize this software system to create terrain even if a heightmap image is not available to use at that moment in time. Once can easily notice the lack of the Input Handling controls on the lower right-hand corner of the screen. Moreover, the file input text box and „Load" button have vanished, making place for the resolution slider which controls the resolution of the perlin terrain.



Resolution: 90
Main Layer: 5.57554
Minor Details: 4
Height: 0.5114286

Figure 6.11 Example Perlin Mode

Chapter 7. Testing and Validation

The testing phase of a software product begins alongside the development and does not stop until the product is finished. Discreet component testing and overall testing of the application allow the developers to readily fix any issues which might appear during the development process. The components have been tested as they have been developed, first individually, then their interactions were tested and finally their place inside the final application. Finally, after the application reached a reasonable shape and functionality, feedback from users was sought to at least partially validate the usefulness of the TSCH algorithm. The two sub-chapters describe these two types of testing: Software Testing and Validation through user feedback.

7.1. Software Testing

Software testing is the testing of all software components, individually and as a whole, throughout the development process. It is essential to discover potential bugs and weaknesses as early as possible to prevent issues later on which would be harder to fix as more behavior is implemented and components become used together as a whole.

7.1.1. Code/Component Testing

Component testing refers to the individual testing of the components forming the final software system. This kind of testing ensures that any component which will be used as part of the whole is correctly working by itself. This means the behavior of the component should be exactly as expected. Modularity of the component system means this kind of testing is easy to complete by using each component alone and testing its public methods, verifying the behaviour for each and every one of them as they are built and then one last time to make sure they can be successfully used later on duringt he development process. This sub-chapter briefly describes how each of the components was tested to make sure it behaved properly and was ready to use as an integrated part of the application. The components are listed in the order of their implementation and, hence, initial testing.

7.1.1.1.Noise Component

Testing the Noise component was relatively straightforward. The only thing it is supposed to do is provide Perlin noise values for given coordinates coupled with noise parameters like frequency and octaves. Extracting values from the methods was easy. The second part of testing involved making sure the values obtained are actually part of a Perlin noise plane. This was done by having an external class paint an image based on the values obtained. This way one could easily check the effects of the Noise component by looking at the produced images, fixing any computation errors and bugs along the way. In much the same way were the shaders tested for all three dimensions. However, since they are too restricted for general availability, even though functional, they have been removed from the final application’s modus operandi.

7.1.1.2. Seeded Interpolation Mapper Component

The Seeded Interpolation Mapper component was quite similar to the Noise component with regard to testing it during its coding. The testing for this component was done in several steps, using mostly debug breaks and debug logs to print out values from the seed matrix and then from the image being created.

The first step of testing was done to ensure the seed points are created and distributed uniformly and randomly across the problem space. This meant checking the point distribution in the seed grid, checking how many seeds had been assigned to each grid cell and checking that the points fill up randomly and evenly every grid cell. Grid cells, in this context, are the subdivisions of the grid of seed points used to speed up seed point searching.

After it was ensured that seed points were being generated both randomly and evenly across the plane, testing was performed to check if the seed points can correctly warp a given matrix using linear interpolation. This part of the testing was done in conjuncture with the Image Reader so as to work on actual images and not just some random test data.

7.1.1.3. Image Reader Component

The Image Reader component was easy to test, since its main functionality is just reading images from the project resources and from the filesystem. Each of these two file sources was tested independently and the ease of Unity game engine’s rendering allowed me to display the picture on the screen with a minimal amount of code. The actual test procedure was simple: I simply created images then loaded them using the Image Reader’s methods followed by displaying them on-screen by converting the resulted floating-point array back to an image.

7.1.1.4. Filters Component

The Filters component also presented trivial testing, since digital filters are a tried and true method of filtering digital images. The filters were applied on a small matrix to verify the values returned are indeed the right ones. Testing was quick and effortless and resulted in two working digital filters: the mean filter and the median filter.

7.1.1.5. Exporter Component

The Exporter component was also easy to test, being functionally-opposite to the Image Reader component. Its sole job is to receive a floating point value matrix and transform it into an image, saving it to the filesystem at te given absolute path. This was tested paired with the Image Reader, making sure that the output image was identical to the given input image. Testing was completed in a short amount of time, validating the usefulness and proper functioning of this component.

7.1.1.6. Perlin Mapper Component

The Perlin Mapper component is a mesh-altering class which directly influences the underlying mesh attached to the object this component is on. It utilizes the Noise class to produce the height values which will be transfered onto the model mesh. Testing was done by replacing the image output of the Noise component testing with a mesh. The same array which painted the image now modified the height values of a mesh of the same size as that previous image.

Auxiliary testing was done to enforce the support of various minor functionality, like the refresh capabilities and changing parameters on-the-fly. This was possible by utilizing the Unity game engine's editor as a testing tool by enforcing automatic model updates during runtime while looking at the editor screen. Thus, no user interface was needed because the editor's interface was used instead.

7.1.1.7. Input Handler Component

The Input Handler component is a parallel component to the Perlin Mapper, since it also modifies the underlying mesh of the object. This component, however, is influenced by a given input heightmap and operates alongside the other components to implement the TSCH algorithm. The testing was done step by step as each step from the algorithm was developed.

Firstly, the Image Reader functionality was added and testing was done to assure the floating-point array is correctly being accessed. Then an instance of the Seeded Interpolation Mapper was added to handle sampling the array and morphing the model mesh. This testing was also done through the editor's interface, modifying the sampled image and making sure it is properly modified by the Seeded Interpolation Mapper. The next step was adding Perlin noise to the implementation. The testing of this was also done easy through the model visualization of the Unity game engine editor and varying parameters to see the effects on the model. The last step was adding the Filters to the implementation and checking that they do affect the model as intended.

7.1.1.8. Controller Component

The Controller component was designated to link the user interface to the Input Handler and Perlin Mapper and implement top-level application logic. The implementation was done in parallel to the development of the user interface in order to make sure the Controller does what it is supposed to. The testing was done by adding controls to the user interface and linking them through callback functions to the other software system components. Testing was thus done on the last time period of the development. Each method was individually tested to make sure the control triggers the proper reaction.

7.1.1.9. Exporter Component

The Exporter component, being the functional equivalent of the Image Reader, was just as easy to test by making sure the images loaded were the same. Testing was thus done by using the Image Reader to retrieve an image and then using the Exporter to reproduce the image.

7.1.2. *Functional Testing*

Functional testing was done to test the way the components behave in order to ensure a better control of the application. Step by step testing allowed to closely observe the application processing results and plan out the interface based on the behavior and responsiveness to parameters.

7.1.2.1. *Testing the TSCH algorithm steps*

One of the most important functional testing procedures was to visually observe the results of each of the three main TSCH algorithm steps. Testing was done using the Unity game engine editor without a full user interface implementation, relying on the editor interface to vary parameters and trigger each algorithm step. Figure 7.1 presents a simple map being taken through the steps. The succession of pictures, left to right and top to bottom presents: no transformation, after seeded interpolation, after Perlin noise addition and after a filter was applied.

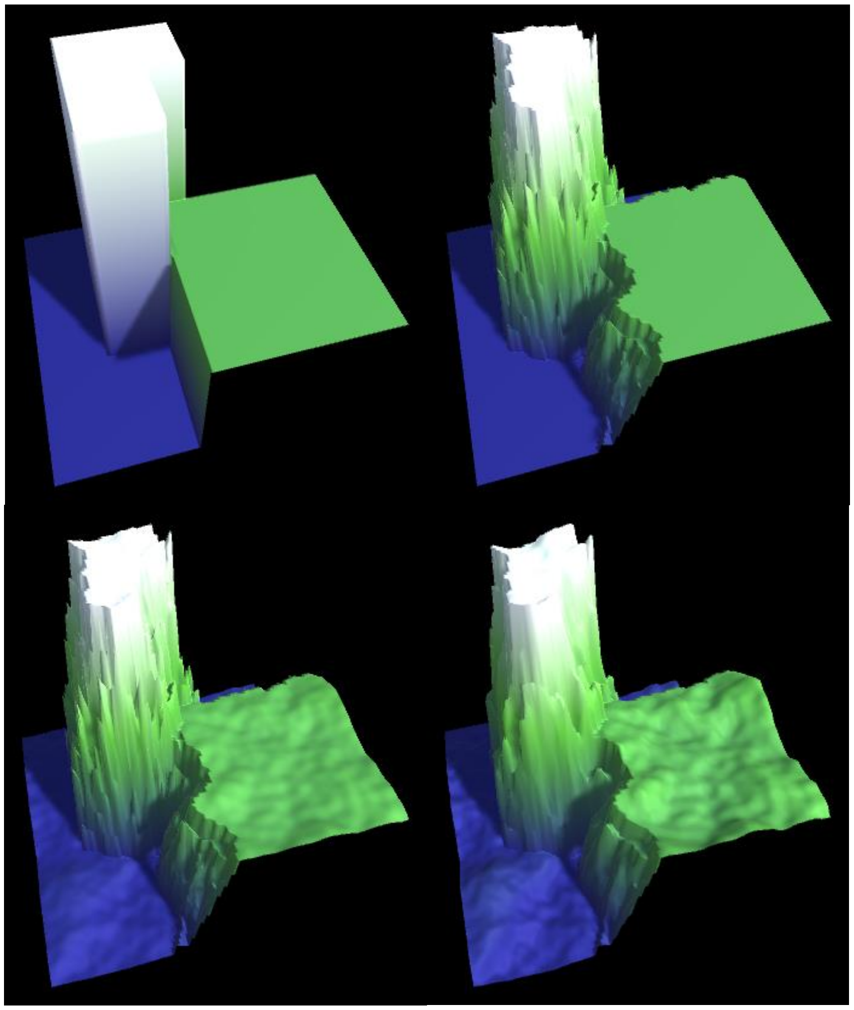


Figure 7.1 Simple Map - Steps

Figures 7.2 and 7.3 present the same four steps applied to a more complex map and a very complex map, to watch the differences in behavior when one applied the same parameters to differently complex heightmaps.

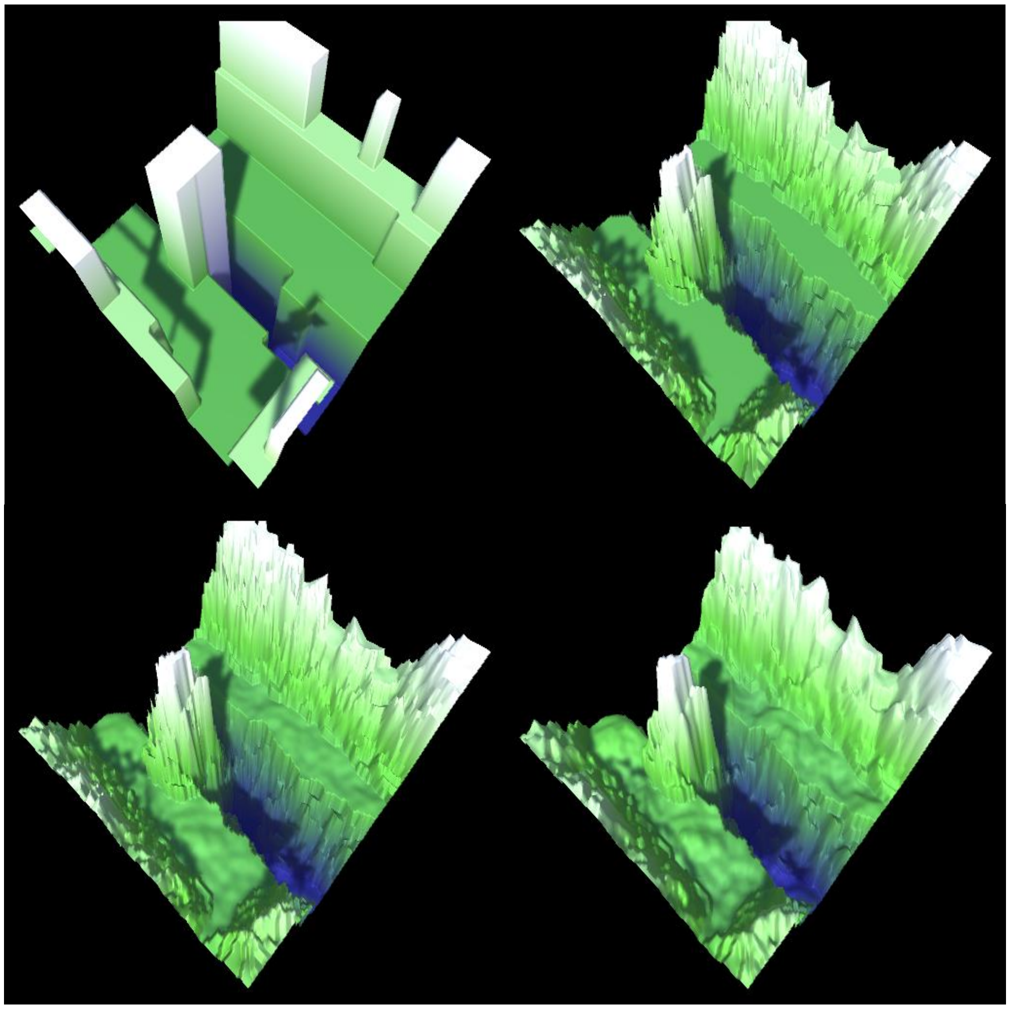


Figure 7.2 Complex Map - Steps

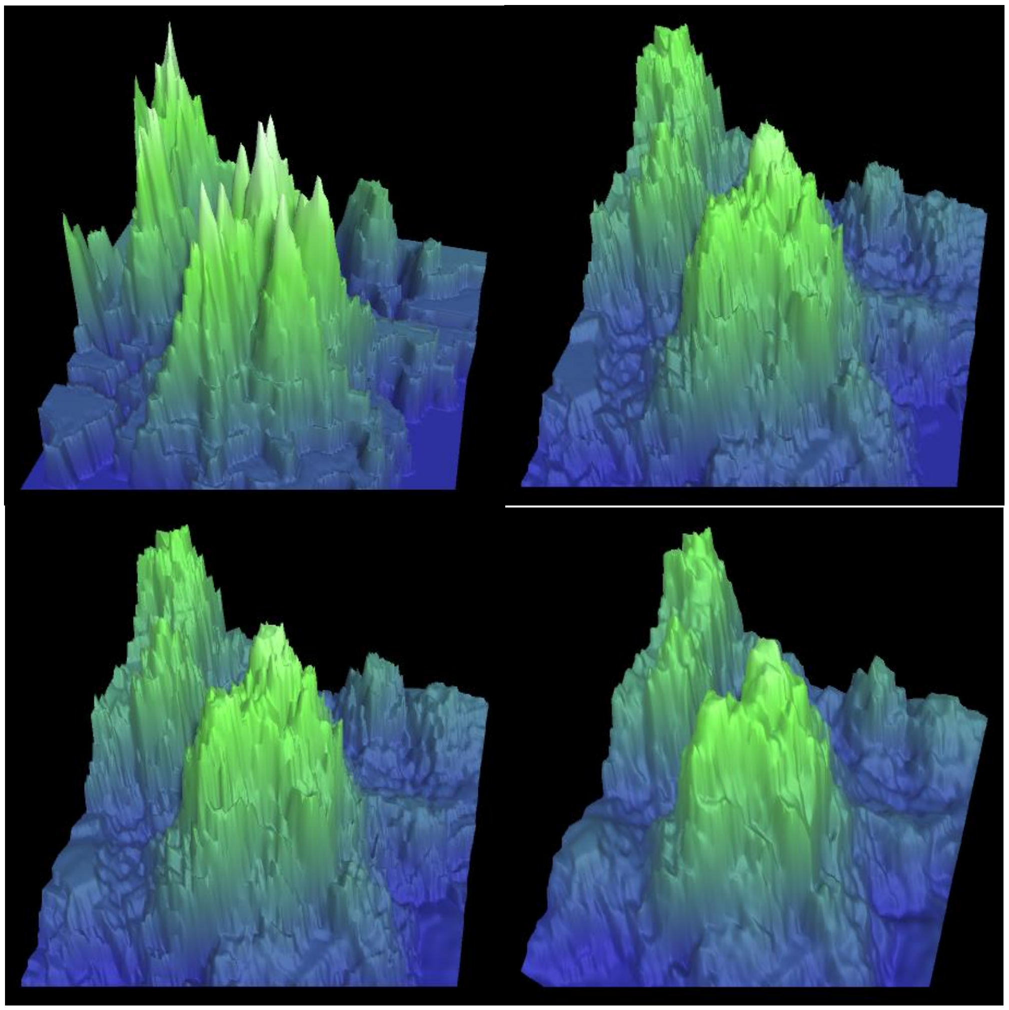


Figure 7.3 Very Complex Map - Steps

7.1.2.2. Testing Interpolation Parameters

The Seeded Interpolation algorithm’s parameters, namely the number of seed points to be taken into consideration when interpolating and the total number of seeds scattered across the image. Testing these parameters was done independently. First one parameter is kept constant and the other is varied, then the reverse happens.

Figure 7.4 illustrates the variation of the number of seed points taken into consideration while the total number of seeds is kept at one seed for every twenty-five pixels (1:25). From top to bottom and left to right, the number of neighbors taken into account for the linear interpolation is five, fifteen and twenty-five. It can be easily noticed that keeping a low number of neighbors means a lot of te initial detail of the application is preserved, which is not necessarily a good thing. However, too many neighbors means the image comes out highly smoothed, negating some of the detail. Hence, empiric evidence suggests using numbers between ten and fifteen neighbors for a decently-looking resulting terrain model.

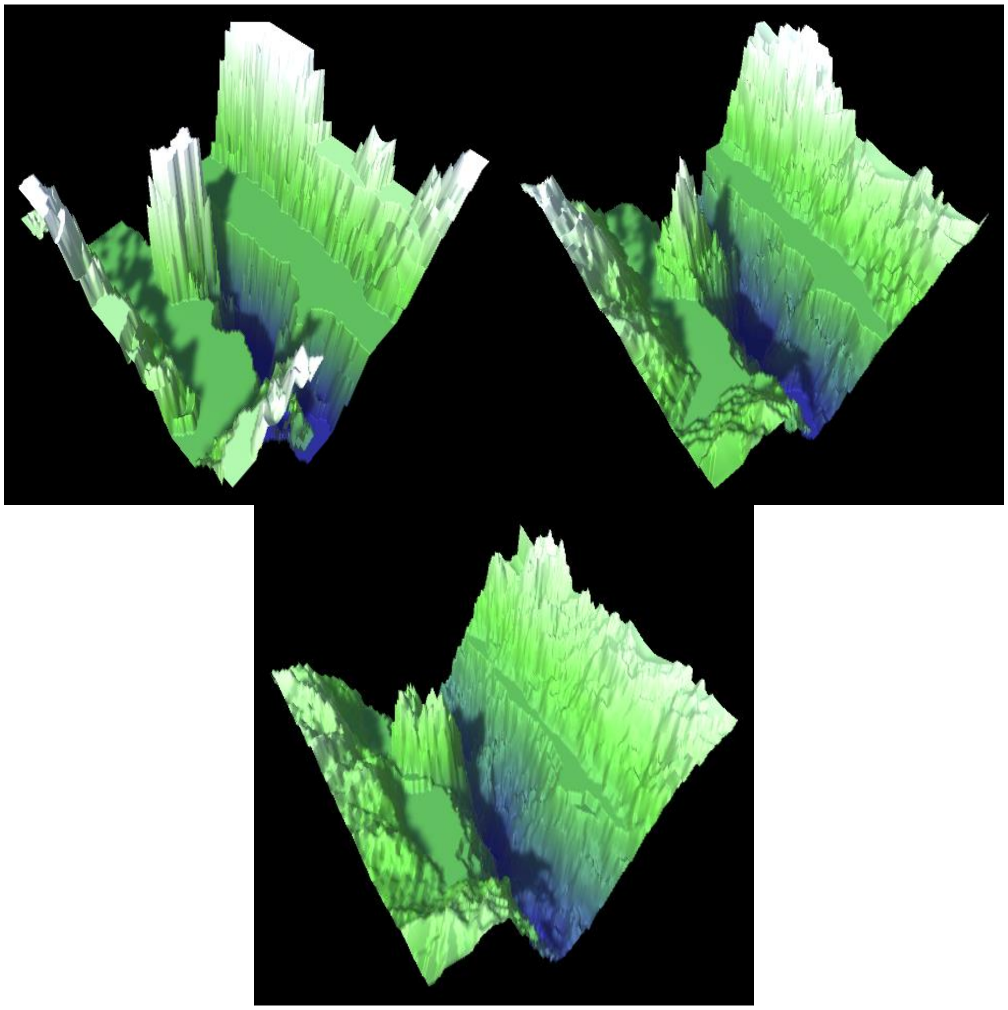


Figure 7.4 Nearest N Neighbors Variation

Figure 7.5 presents variations of the total number of seed points when the nearest neighbors taken into consideration for the linear interpolation is kept at fifteen. As presented from left to right and from top to bottom, the ratio of seed points to pixels is: 1:1, 1:20, 1:50, 1:150 and 1:250. 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 evidentiates 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 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.

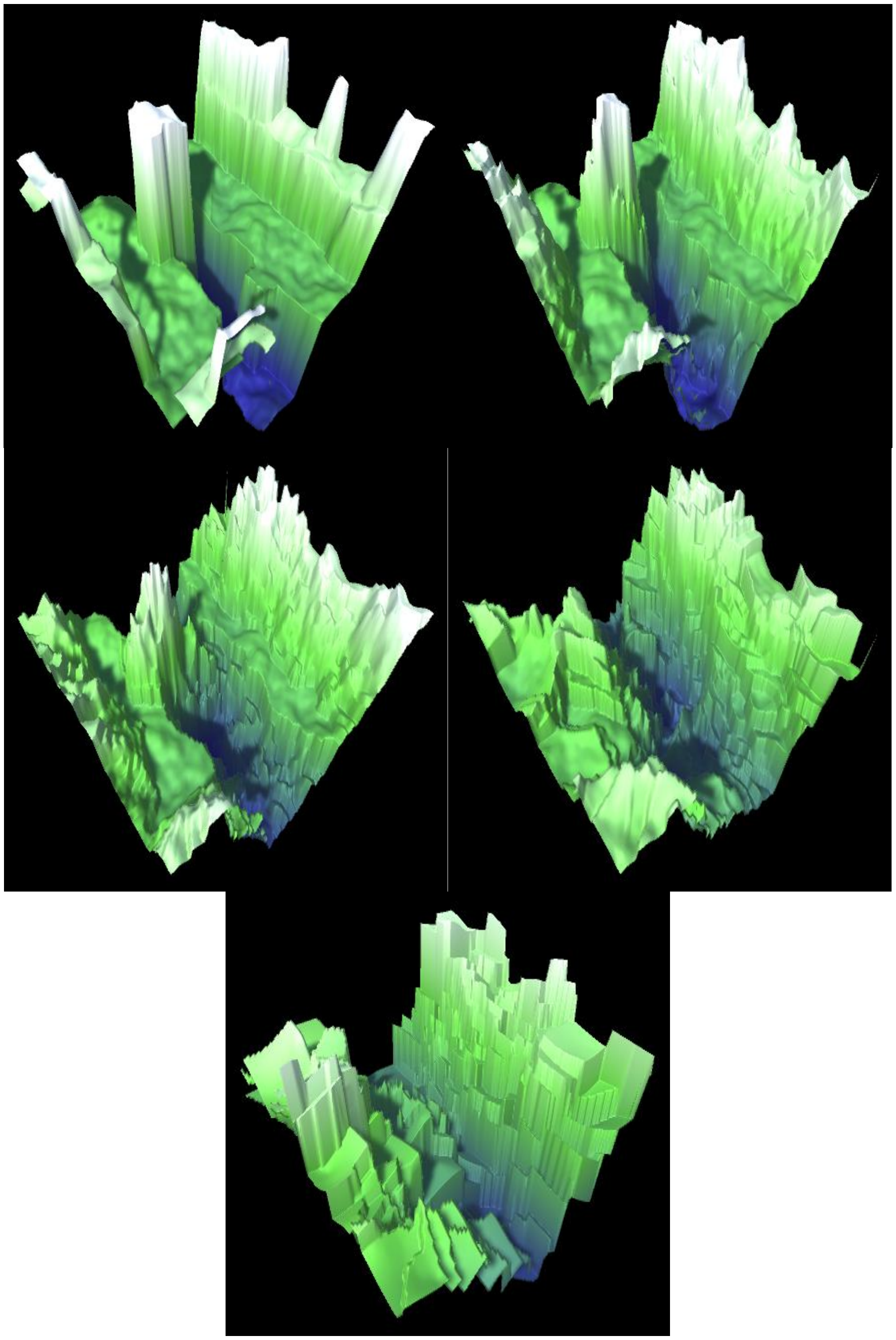


Figure 7.5 Seed Point Total

7.1.3. Application Testing

Application testing was done to verify that the user interface behaves and reacts just as planned. Planning was done in parallel with the development of the user interface and the Controller component. Testing was done for each region to make sure it works correctly and is bound to the right algorithm parameters.

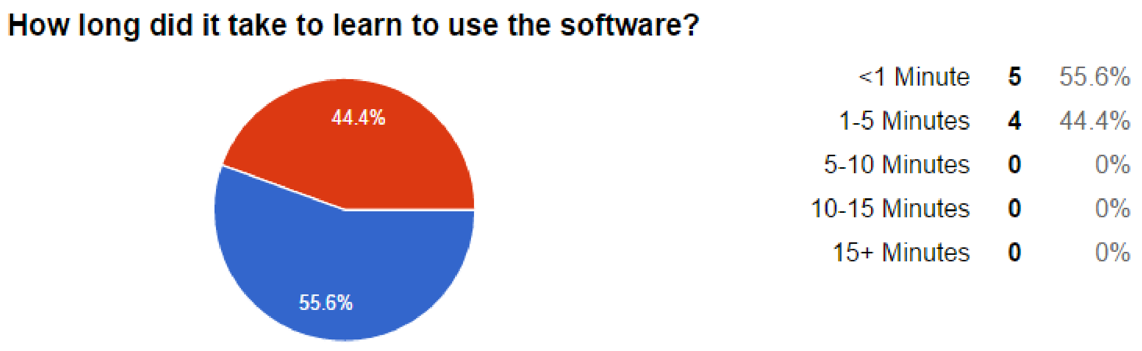
7.2. Validation – User Feedback

User feedback was sought in order to validate the algorithm as working. The application was distributed and a Google form was created to capture the user feedback. Roughly 35 people have expressed an interest in the application and, out of them, 9 have completed and submitted the feedback form. The majority of the people was centered in the game development area, having had mostly game developers comment and show their interest. This is a biased experiment, though, since one of the main distribution points for the application was in a game development group. A surprising number of the interested people also had come before in contact with other terrain generation software, making their feedback even more relevant and valuable.

Moreover, all feedback was, from a purely subjective point of view, positive. The commentary given ranged from „an unique twist” to „extremely useful” (actual quotations). The following sub-chapters rely on the 9 actual feedback forms which were submitted and which shape a slightly more objective view on this project's results.

7.2.1. Ease of Use

Since this algorithm is supposed to make easier the introduction of people into the world of terrain synthesis, the ease of use was of utmost importance to be registered. A point to be made is that the user interface closely influences the ease of use and it is nigh-impossible to separate these two elements. Thus, the realization of the user interface needs to be taken into account when considering the following results. The following figures present the analytics data retrieved from the form, alongside an answer count. This sub-chapter ignores the „figure mentioned before appearance” due to the nature of the figures including all relevant information within and being followed by commentary.

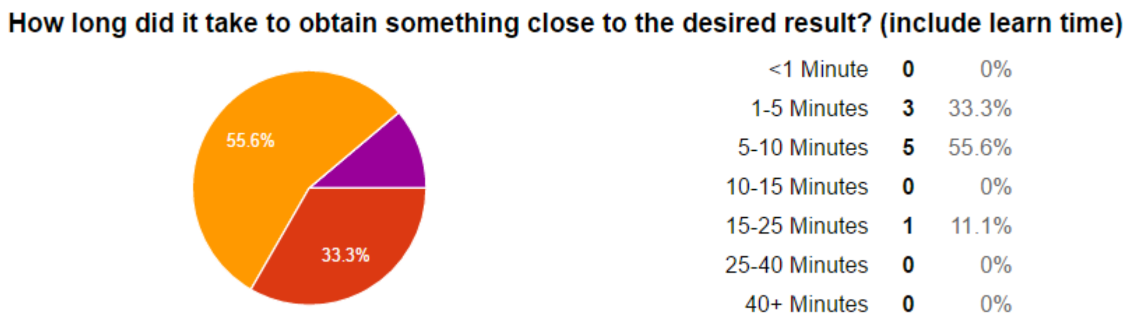


How long did it take to learn to use the software?

<1 Minute	5	55.6%
1-5 Minutes	4	44.4%
5-10 Minutes	0	0%
10-15 Minutes	0	0%
15+ Minutes	0	0%

Figure 7.6 Time to Learn

Figure 7.6 presents TTL (time-to-learn) results. All users took 5 or fewer minutes to learn how to handle the application. This is a great result pointing towards an intuitive interface which anyone can use and profit from. Furthermore, it is extremely fast compared to other software products due to its simplicity, meaning the ramp-up time is so low, one can almost instantly start working on their own projects by using this application. Over half of the users took less than a minute to acquaint themselves with the application. This is an achievement few software products may be able to pride themselves with and can serve as a pivotal strong point of this specific product. Should the algorithm be improved in any way, it can only benefit the overall application by keeping things simple yet increasing the power and quality.



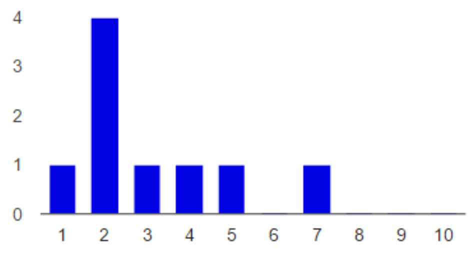
How long did it take to obtain something close to the desired result? (include learn time)

<1 Minute	0	0%
1-5 Minutes	3	33.3%
5-10 Minutes	5	55.6%
10-15 Minutes	0	0%
15-25 Minutes	1	11.1%
25-40 Minutes	0	0%
40+ Minutes	0	0%

Figure 7.7 Time to Result

Figure 7.7 presents the time required for the user to reach a result which is at least close to the desired one. Time to learn is included in this since I wanted to highlight how fast completely new users can go from nothing to a valid result, without having experienced the application before. Again, the results are spectacular. 8 out of 9 users have reported being able to produce a terrain model close to what they were expecting within the first 10 minutes of use. This is a fast result compared to almost any other kind of visual production software. The TTL is thus followed by roughly 5 or less minutes of adjustments to obtain the desired result.

How easy was it to use this software?



Very easy (a child could do this): 1	**1**	11.1%
2	**4**	44.4%
3	**1**	11.1%
4	**1**	11.1%
5	**1**	11.1%
6	**0**	0%
7	**1**	11.1%
8	**0**	0%
9	**0**	0%
Very difficult (it takes a lot of practice): 10	**0**	0%

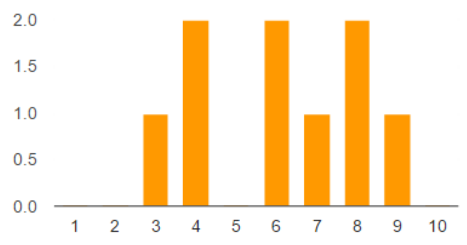
Figure 7.8 Ease of Use – User-reported

Figure 7.8 presents precisely the results of the ease of use question posed to the user. It is important to consider the user's viewpoint when trying to form an overall picture regarding the product and its use. As one can notice in the results, the vast majority of users (almost 90%) found the product relatively easy to use. On a scale from 1 to 10, where 1 is the easiest and 10 is the most difficult, the answers are concentrated on the 1-5 area and, within it, the majority leans towards a 2 (44% of the total). This is another powerful indicator of how easy the application is to use, opinionated by the users themselves. Of course, the fact that it is spread out may mean that the interface is not at its best and that improvements can be made, to shift the results further towards the 1-2 region.

7.2.2. Closeness to Expected Result

The closeness to an expected result is the other important part of the user feedback. This is the result which the application produces. The following figures display results regarding the feedback about the resulting terrain model synthesized from the user-provided heightmap.

Did the resulting terrain model meet your expectations?

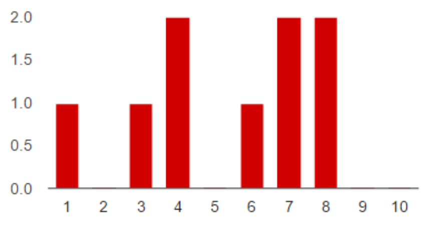


No. Far from it...: 1	**0**	0%
2	**0**	0%
3	**1**	11.1%
4	**2**	22.2%
5	**0**	0%
6	**2**	22.2%
7	**1**	11.1%
8	**2**	22.2%
9	**1**	11.1%
Totally! It's everything I wanted!: 10	**0**	0%

Figure 7.9 Meeting Expectations

Figure 7.9 presents how close the terrain models the users were able to create resemble what they started out to achive. How much of their expectations were met by this application. The results are more divided about this. The trend is positive, as 67% of the users were at least halfway satisfied with their resulting model. The other 33% of users with lower levels of satisfaction still gravitate towards the halfway point, even if their grade was lower. A way to interpret these results is that the application produces noticeably good results but not great ones. The algorithm can benefit from improvements to make the result better looking and the application can be developed to allow more user control over the terrain. This potentially raises the user experience complexity but also improves the end result by better allowing the user to shape the terrain according to his or her requirements.

Did the result look like... actual terrain?



No. The result was a hideous abomination!: 1	**1**	11.1%
2	**0**	0%
3	**1**	11.1%
4	**2**	22.2%
5	**0**	0%
6	**1**	11.1%
7	**2**	22.2%
8	**2**	22.2%
9	**0**	0%
Totally. It looked like it could exist in nature.: 10	**0**	0%

Figure 7.10 Realistic Terrain

Figure 7.10 presents the users' opinion on how realistic the created terrain was. The results are unsurprisingly close to those of the previous user feedback question: the majority of users thought it was at least halfway decent as far as terrain models go while close to half of the users thought the result was not good enough as far as aesthetics go. This points out and validates the fact that the algorithm was not made to provide geomorphically correct terrain. This is a definite area where the algorithm can be improved, by making the end result even more terrain-looking.

Chapter 8. User Manual

8.1. System Installation

The installation of the application is incredibly easy. Since Unity it being used, it can be compiled for any operating system as long as it is not for commercial use. This particular application has been compiled for the Windows, Linux and OS X operating systems. The application comes bundled with all the needed dependencies, placed in the _Data folder. As long as the folder is placed near the executable, the application should run perfectly fine.

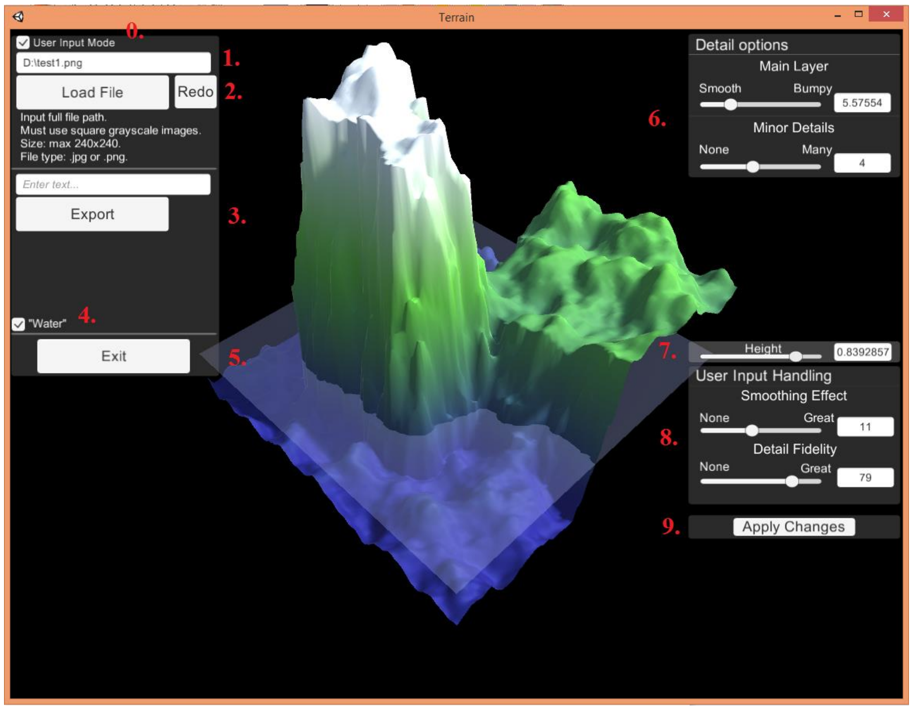
Installation steps:

1. Unpack the .zip archive.
2. A) Windows – copy the _Data folder and the .exe file to the desired final location.
2. B) Linux – copy the _Data folder and the .x86 file to the desired final location.
2. C) OS X – drag and drop the .app file to the desired final location.

8.2. Usage

Starting the application is as simple as starting any executable on that specific operating system. Using it is almost as simple. Figures 8.1 and 8.2 display the user interface when in User Input mode and Perlin mode, respectively. The interface elements, presented previously in chapter 6.3 are numbered and will be presently briefly described.

0. Toggle control – used to switch between User Input mode and Perlin mode
1. File Load control – used to signal the location where the input heightmap will be loaded from
2. Redo button – used to get a new random value for the terrain model details
3. Export control – used to choose a destination image file for the current model's heightmap export
4. Water toggle – used to toggle the water simulacrum plane on or off
5. Exit button – used to exit the application. No changes are saved.
6. Detail options region – used to control the detailing layer
7. Height control – used to control the total vertical scale of the model
8. Input handling region – used to control the TSCH algorithm's interpolation step in a user-friendly manner
9. Apply changes button – used to signal the application that the changes need to be applied to the on-screen terrain model
10. Resolution control – used to change the resolution of the Perlin noise mesh

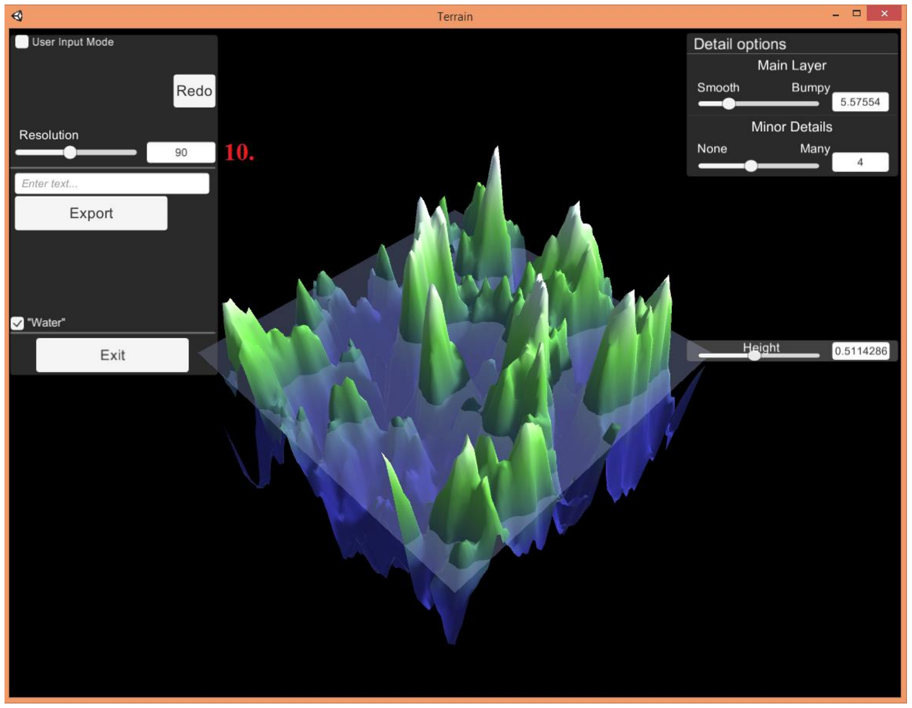


D:\test1.png

Input full file path.
Must use square grayscale images.
Size: max 240x240.
File type: .jpg or .png.

Smooth — Bumpy: 5.57554
None — Many: 4
Height: 0.8392857
Smoothing Effect: None — Great: 11
Detail Fidelity: None — Great: 79

Figure 8.1 User Input Mode



Resolution: 90
Smooth — Bumpy: 5.57554
None — Many: 4
Height: 0.5114286

Figure 8.2 Perlin Mode

Chapter 9. Conclusions

9.1. Contributions and Achievements

Through this project's completion, I have successfully introduced a new algorithm for terrain synthesis. The TSCH algorithm is, as far as my research tells, new to the field since most of the current algorithm research effort revolves around improving the generated terrain's quality and maleability at the cost of user input requirements expanding proportionally. The TSCH algorithm thus fills a niche in the procedural generation domain by lowering the entry point for users trying out terrain synthesis from heightmaps. This is possible by the researched method to detail said heightmap to a more suitable state, even if the input detail is kept to a minimum. This contribution to the field of terrain synthesis and procedural generation is the main achievement of the project.

9.2. Result Analysis

In the beginning, two goals were set:

1. Develop the TSCH algorithm
2. Implement it in an application

Both these goals were successfully reached. The first goal, developing the algorithm was reached after the research phase, when comparing different procedural generation algorithms for their efficiency. Multiple algorithms were considered but Perlin and Worley noise came on top due to their simplicity. Some of the others considered were good but would have exponentially increased developing time. Thus, this algorithm is a lightweight one, perfect for fast deployment and supports great further customization, since only the first step, the randomly seeded linear interpolation, is essential to TSCH, the rest of them being adaptable to other algorithms in a mix-and-match fashion. The algorithm is described in detail in Chapter 4 of this paper.

The second goal was reached through the writing of an actual implementation inside the Unity game engine. The technological choice was made to simplify the rendering output and take advantage of the component based system. The implementation was modularized through the use of C# scripts within Unity. After the behaviour was coded, an interface was added, alongside a controller to act as the bridge between the functional part of the application and the user interface. The implementation is described in Chapter 6 of this paper.

9.3. Further Development

There is a plethora of future development possibilities all across both the TSCH algorithm and the application:

- The seeded interpolation algorithm can be optimized
- The algorithm can be improved to work with consecutive segments of terrain to be „stitched" together in one big terrain model
- The application can have more user input options added for more control over the generation
- Code can be written to control model segmentation in sub-models to allow big terrains to be rendered while avoiding the 64k vertex limit of models

Chapter 10. Bibliography

[1] „Discover,” World Machine Software, LLC, [Interactiv]. Available: www.world-machine.com/about.php?page=features.

[2] „Introducing Gaia,” Procedural Worlds, 2015. [Interactiv]. Available: www.procedural-worlds.com/gaia/.

[3] A. H. Zhou, J. Sun, G. Turk şi J. M. Rehg, „Terrain synthesis from digital elevation models,” *IEEE Transactions on Visualization and Computer Graphics,* 2007.

[4] L. Cruz, F. Ganacim, D. Lucio, L. Velho și L. H. de Figueiredo, „Exemplar-based Terrain Synthesis,” 2013.

[5] G. Voronoi, „Nouvelles applications des paramètres continus à la théorie des formes quadratiques,” *Journal für die Reine und Angewandte Mathematik,* nr. 133, pp. 97-178, 1908.

[6] G. L. Dirichlet, „Über die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen,” *Journal für die Reine und Angewandte Mathematik,* nr. 40, pp. 209-227, 1850.

[7] S. Worley, „A Cellular Texture Basis Function,” în *SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques,* 1996.

[8] K. Perlin, „Making Noise,” 2002. [Interactiv]. Available: www.noisemachine.com/ talk1/.

[9] D. J. Whitehouse și R. E. Reason, The Equation of the Mean Line of Surface Texture Found By an Electric Wave Filter, Leicester: Rank Taylor Hobson, 1965.

[10] R. Fisher, S. Perkins, A. Walker și E. Wolfart, „Spatial filters - Median filter,” 2003. [Interactiv]. Available: homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm.

[11] R. Fisher, S. Perkins, A. Walker și E. Wolfart, „Spatial filters - Mean filter,” 2003. [Interactiv]. Available: homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm.

Appendix 1 – RoCHI submitted paper

Terrain Synthesis from Crude Heightmaps

Alexandre Philippe Mangra
Technical University of Cluj-Napoca
Str. G. Barițiu 28, 400027, Cluj-Napoca, România
alexandre.mangra@gmail.com

Adrian Sabou
Technical University of Cluj-Napoca
Str. G. Barițiu 28, 400027, Cluj-Napoca, România
adrian.sabou@cs.utcluj.ro

Dorian Gorgan
Technical University of Cluj-Napoca
Str. G. Barițiu 28, 400027, Cluj-Napoca, România
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 parametrized to show how the user can influence the resulting model.

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

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Algorithms; Terrain models.

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.

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 [1]. 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 [2], 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. [3], 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. [4], 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.

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.

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.

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 beings 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 1. 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.

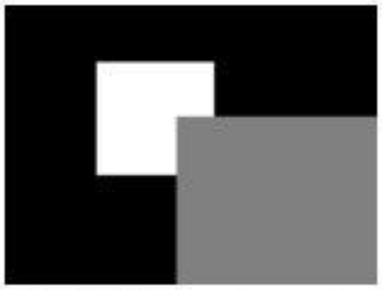


Figure 1 – Sharp transition example

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 1.

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 [5].

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 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 2. 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.

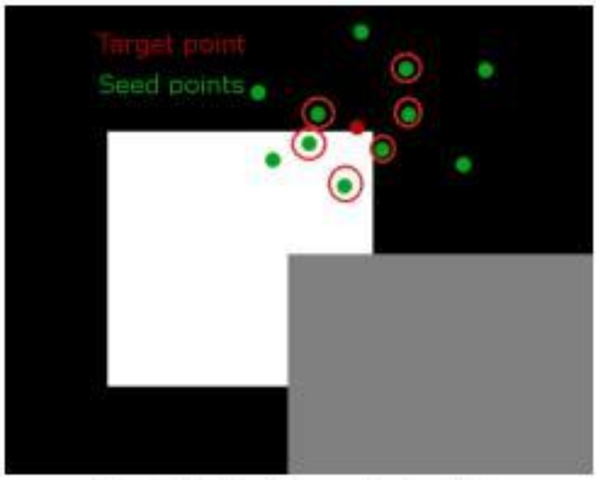


Figure 2 – Seed points around a fixed point

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.

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 [6] for this purpose, which is a homogenous noise implementation created by Ken Perlin in the early '80s. 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.

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 [7], [8] 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.

IMPLEMENTATION

The Unity game engine

For implementing and testing, the Unity [9] 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.

TESTING

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 two were made by hand, the third one is sampled from the Internet [10]; presented in Figure 3.

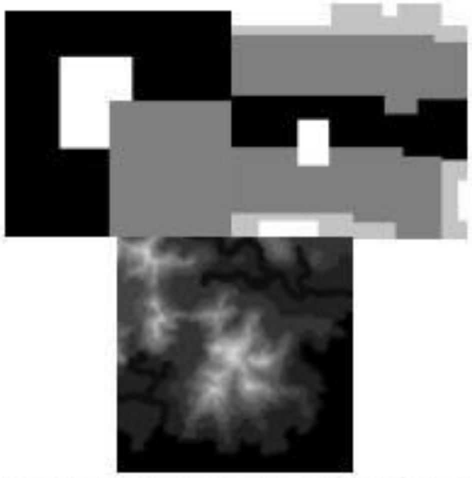


Figure 3 – Left: Simple Heightmap; Right: Detailed Heightmap; Bottom: Complex Heightmap

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 4a and 4b show how the model advances from its initial state to the final, more detailed output.

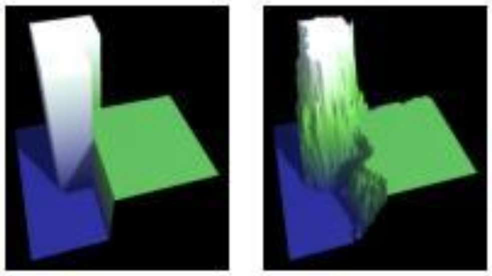
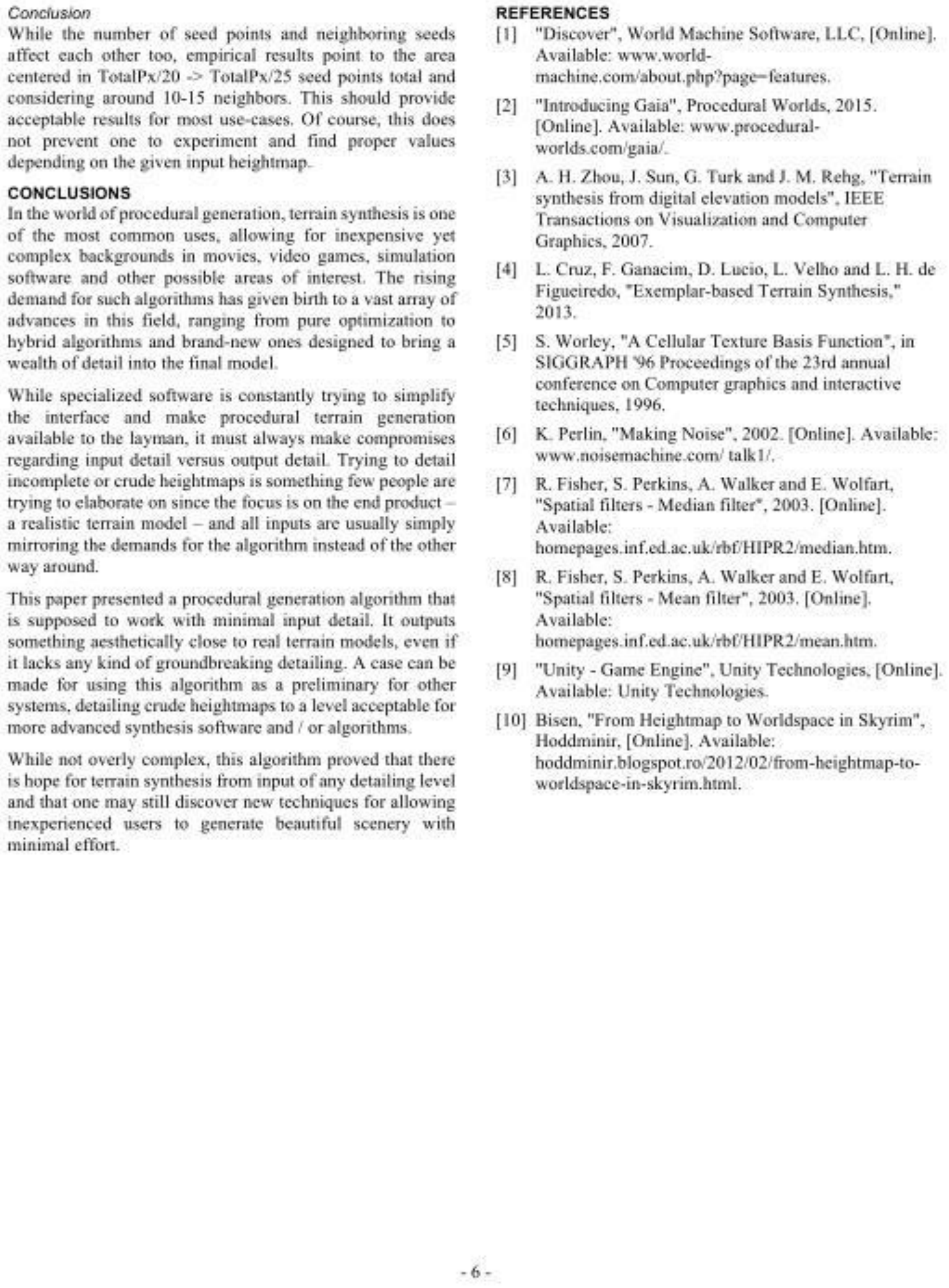


Figure 4a – Left: Original State; Right: Edge Smoothing



Conclusion

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.

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.

REFERENCES

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

[2] "Introducing Gaia", Procedural Worlds, 2015. [Online]. Available: www.procedural-worlds.com/gaia/.

[3] A. H. Zhou, J. Sun, G. Turk and J. M. Rehg, "Terrain synthesis from digital elevation models", IEEE Transactions on Visualization and Computer Graphics, 2007.

[4] L. Cruz, F. Ganacim, D. Lucio, L. Velho and L. H. de Figueiredo, "Exemplar-based Terrain Synthesis," 2013.

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

[6] K. Perlin, "Making Noise", 2002. [Online]. Available: www.noisemachine.com/ talk1/.

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

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

[9] "Unity - Game Engine", Unity Technologies, [Online]. Available: Unity Technologies.

[10] Bisen, "From Heightmap to Worldspace in Skyrim", Hoddminir, [Online]. Available: hoddminir.blogspot.ro/2012/02/from-heightmap-to-worldspace-in-skyrim.html.