DEGREE PROJECT, IN INFORMATION AND COMMUNICATION TECHNOLOGY , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

A Graph-Based Approach to Procedural Terrain

Bachelor Thesis

Written by: Vidak Mijailovic
Examiner: Christian Schulte

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

BACHELOR THESIS
DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY

A Graph-Based Approach to Procedural Terrain

Vidak Mijailovic
29th of June 2015
Stockholm, Sweden

Examiner: Christian Schulte
TRITA-ICT-EX-2015:72

Abstract

Procedural terrain generation is a field that handles procedural, not by hand, generated realistic looking terrain for use in simulations, video games, movie special effects or art. It allows for creation of vast and far more detailed terrain than humans can create by hand.

In this paper a new method for procedural terrain generation is presented. Terrain is generated in three steps. An arbitrarily shaped network oh nodes, a graph, is used as a base to design the shape and layout of terrain features. Common algorithms are used to generate custom terrain features inside the graphs sealed areas and finally the generated terrain is merged into a single piece using a new method. In this manner, more controlled and detailed terrain can be created as the layout and shape of features can be controlled.

Keywords: Terrain, real-time computer graphics, digital-cartography, graph-grammar, Perlin noise, diamond-square

Referat

Området terräng generering hanterar procedurellt, icke för hand, skapande utav realistisk terräng för användning inom simulationer, data spel, filmers specialeffekter och konst.

I denna uppsats presenteras en ny metod för procedurell terräng generation. Terrängen genereras i tre steg. Ett godtyckligt nätverk av noder skapas, en graf, och används som grund för att designa och forma utläggningen av terrängens drag. Kända algoritmer används för att skapa dragen inuti grafens tomma ytor och slutgiltigen sys den genererade terrängen ihop till en sammanhängande helhet med en ny metod. På detta vis kan man skapa mera detaljerad och bättre styrd terräng då man både kan kontrollera dragens former och utläggning.

Nyckelord: Terräng, realtid datorgrafik, digital-kartografi, graf-grammatik, Perlin noise, diamond-square

Table of Contents

1 Introduction

1.1 Background

The area of procedural terrain generation has existed since the 1980s with the purpose of procedurally, not by hand, generating vast realistic looking terrain for use with computer graphics. Its first commercial use was for creation of alien planets in the popular TV series Star Trek (Nove: Season 36, Episode 4, Hunting the Hidden Dimension 28 Oct 2008) (Lucasfilm 1982), a feat amazing the crowd at the time.

Most of the algorithms are based on fractal mathematics and are often showcased as an area promoting fractals. This is a vast area growing with the increase in modern computing power with lots of problems that still need to be solved, and lots of problems that still can be explored.

1.2 Problem Statement

Two of the most popular algorithms for generating terrain today are the Diamond-Square algorithm and the Perlin Noise algorithm. Both produce an array of numbers called a height map, sometimes also called a patch.

The Diamond-Square algorithm (Alain Fournier 1982) produces heightmaps by using fractals. At start, the patch has four preset points, a point in each corner. These preset values are called seeds. It takes two of these points and views them as one line. It then breaks the line by inserting a new point with a random offset in the middle, breaking it down into two lines. This is repeated several times. It produces realistic looking terrain but it is difficult to control the features of the resulting terrain. Scaling the algorithm to “infinite”, or actually boundless, terrain is difficult because each generated heightmap relies on the heightmap next to it for points to break. Also, the sizes of the heightmaps have to be certain fixed dimensions.

Perlin Noise (David S. Ebert 1994) is easy to scale and make infinite terrain. It produces terrain by combining waves with high or low amplitudes and short or long frequencies a few amounts of times. The only parameters it requires are the description of a wave, the rate of descent for each iteration, which is how the maximum amplitude of the wave should change on each iteration, and the number of iterations to perform.

But when one defines a wave, it will generate terrain with the same maxima and minima for each iteration in all directions. Variations such as lows and highs in mountains and valleys do not look natural, they will be evenly spread out. For example, in a patch of 500 square meters there will always be the same amount of mountains and valleys, which is not the case in natural terrain.

1.3 Problem

Instead of building a terrain system that generates the same terrain in all directions with the same formula, a graph-based system is created for designing, generating and modifying infinite terrain.

This system should be able to manually or procedurally generate a graph. All the closed areas of the graph are called fields. Each field contains terrain and is assigned properties describing it. The terrain of each field is generated based on the assigned properties. For instance, mountain properties with high peaks or flatland properties with a flat surface.

The fields should be of irregular shapes, not just squared patches and the generated terrain should be stitched together from the fields so that borders together from fields do not become visible.

Nodes of the graph can be added, removed or moved to shape the features of the terrain. An infinite world could be generated with a random graph, while allowing fine editing. For instance to place a mountain at point P and shape it while the rest of the world is auto generated.

This way one can create much more interesting and non-uniform landscapes, with a single infinite terrain graph. Naturally ranging from tall Alps like mountains to Mediterranean reaches and African steppe like flatlands, with precise control of the environment.

1.4 Purpose

The field of procedural generation is still relatively young and underdeveloped with many aspects to explore. The current methods only provide basic functionality to generate general terrain. This project aims to expand upon them and create a method that makes up on their weaknesses.

By dividing the process into multiple steps as opposed to just one, more control over the resulting terrain is introduced. Using a graph allows users to speed up the process of creating vast worlds while giving them the ability to pick certain pieces and decide their features.

Today artists spend hundreds of hours creating mountains, valleys and plains for video games before spending more time placing and merging the created assets into a natural looking solid piece for use. The automated algorithms mostly only provide repetitive terrain. Either by first generating a patch of terrain and repeating it in all directions, or by generating the same type of terrain in all direction based on some parameters. On the other hand realistic and exiting terrain in computer games or movies is often designed to be the opposite, differing from place to place to allow exploration and excitement with custom breathtaking designs.

1.5 Goals and Tasks

The main goal is to develop a system and a library in C# for creation of graph-based terrain. The library should provide all the necessary algorithms, data structures and tools one needs to generate, shape and manage graph-based terrain. Enabling the user to quickly and easily create realistic looking terrain that can be extended and shaped with great detail.

The system should be using modified versions of existing terrain generating algorithms to generate custom terrain inside of the graph fields, store it in heightmaps and stitch the different pieces into a solid terrain piece. In order to stitch the borders between the irregularly shaped fields and terrain new algorithms will be developed.

The main goals of the project

- Review current state of the art algorithms for generating terrain.
- Implement the commonly used ones efficiently and make sure they reach good performance.
- Create good data structures for containing and managing the graph and fields.
- Create a good data structure for the generated terrain heightmaps which takes into account the need for stitching irregular terrain pieces.
- Create an algorithm for stitching the terrain of irregularly shaped fields.

Additional goals are:

- Make sure the graph and the terrain can deterministically be recreated when only storing the parameters, increasing the portability of even infinite worlds.
- Make sure the produced library is external library and platform independent. Meaning that no other external libraries will be needed to compile and that the code is not tied to specific architectures.

1.6 Methodology / Methods

The project is divided into four phases, a research phase, a theory phase, an iterative prototyping phase and a testing phase.

An in-depth research phase of the current methods and algorithms used for procedural terrain generation will be conducted. During this phase necessary knowledge and theory will be gathered.

Based on the current foundations and research conducted by others, theory behind a new method will be developed. The method should meet all of the requirements and goals listed in the previous section. It is very important that the research phase is conducted with care so that the new theory does not simply try to reinvent existing theory.

An iterative and agile development process of a prototype that implements the newly designed theory will be performed. The prototype and its development will try to realize all of the goals set above and also provide feedback to the theoretical phase as new unforeseen issues and views may be found during the development. When new findings are made backtracking to the theoretical phase will be done to review what could be improved or changed.

A testing phase is to follow as conclusion to the project. The prototype will be tested to see that it functions as desired, outputs desired results and does so in reasonable execution time.

1.7 Risks, Consequences and Ethics

As in other projects, this project contains risks. The main risks of this project is that it is too big, that it would become too difficult to implement in the available timespan and that it would try to reinvent something already created. Another major risk is to fail the goal of implementing the system as a flexible independent library by accidentally making the code too complex.

The risk of having a too wide scope will be minimized by doing weekly performance and time estimates, to make sure the base of the project gets completed first. Such as terrain algorithm and data structure implementations and afterwards add upon them. This is best prevented by doing solid research beforehand.

The C# library will be created as a separate project in the solution and only include what is necessary in it to create a good API. If any external libraries have to be used or code binding the user to specific external platforms of other authors, they will be contacted to obtain approval and reference them fully. Another option is to, if possible, write and implement equivalent in the time given.

1.8 Ethics and Sustainability

In today's graphics industry hundreds of hours are spent on modeling and designing terrain. The project aims to help speed this process up by allowing for more control during the generation of the terrain so that less time needs to be spent post processing and manually shaping the terrain if it initially is not satisfactory. If the time spent shaping the terrain can be shortened it would help save resources as both economical resources and human resources could be redirected to other causes.

A main ethical issue is to make sure the project and the new method developed do not step into work already developed by others but stands on its own. As the field is still young and somewhat undeveloped there has been a lot of research done that does not surface to gain attention, making it difficult to find, bringing

with it the risk of this project being too similar. This has to be countered by extensive research prior to development. In conclusion this project will do its best to follow the IEEE Code of Ethics as closely as possible. (IEEE 2015)

1.9 Disposition

In the Background and Methods chapter the necessary background knowledge, methods and algorithms are displayed together with their respective pros and cons together. The research conducted prior to the project.

The Execution and Theory chapter presents the new methods and how they are used. The main theory and concepts developed during the project are presented.

In the Results chapter the terrain produced by the developed methods is presented and the execution time is measured. The section shows and compares the execution times of different test cases and puts them in perspective.

Finally in the Discussion chapter the resulting system and algorithms are discussed. What they achieved and what the original aim was together with possible improvements, both inside and outside the scope of this project together with a few conclusions.

2 Background and Methods

2.1 Heightmaps

A heightmap is one of the most basic and common methods for storing terrain data. The underlying data structure is a two-dimensional array or list of arbitrary size. Indices of the array correspond to X and Z coordinates of the heightmap, while values stored in the array correspond to Y coordinates, the height. For example, given a point (x, y, z), it is represented in the heightmap as a value y stored in column z and row x.

Because heightmaps are simple to implement, fast to access data from and intuitive to use, they are common and became the backbone of many algorithms. Heightmaps are used by terrain generating algorithms, such as the diamond-square (Alain Fournier 1982) and the Perlin noise (David S. Ebert 1994) algorithms.

In order to render the terrain as a surface, it has to be converted into a mesh for the graphics driver to handle. A mesh is a solid Polyhedron consisting of solid polygonal faces, straight edges and corners called vertices. Converting the terrain heightmap into a mesh is relatively straightforward. (Colin 2006) Detailed polygon meshes can be created by using all values in the heightmap while coarse polygon meshes can be created by skipping a few values instead. (Colin 2006)

The heightmap itself only contains data describing the height points, sometimes called the features, of the terrain. It does not contain any additional information. When rendering the heightmap, color, textures and other additional data is added by the rendering system, which is a separate system outside the scope of this paper. Such systems are usually based on either Microsoft DirectX (Microsoft 2015) or a version of OpenGL (Group 1997-2015).

Disadvantages of heightmaps are that they only store one Y coordinate for each pair of XZ coordinates. Only one surface can be represented meaning that for instance caves cannot be represented in such systems. For more advanced systems that include cavities and complicated landscape, voxels, volumetric-pixels, can be used instead (Eric 2010). Figure 1 shows how heightmap terrain can look like.

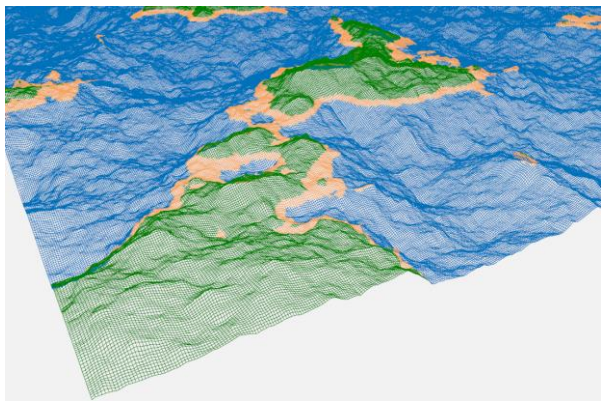


Figure 1: Example of heightmap terrain with additional color added. No point can overlap another.

2.2 Diamond-square algorithm

The diamond-square algorithm (Alain Fournier 1982) produces heightmaps by using fractals through an advanced midpoint-displacement algorithm. The algorithm uses a small piece code recursively to generate terrain. It was first introduced in 1982.

The idea behind the algorithm is to break and deform large line segments into smaller ones until they look like a mountain ridgeline. Suppose there is a line with two endpoints P1 and P2, which have Y coordinates -1 and 1. The algorithm goes to the middle of the line, measures the height at that point and adds a small random offset to the height to form a third point P3. The offset can be positive or negative. The three points can now be viewed as two line segments, (P1, P3) and (P3, P2). This is the midpoint-displacement algorithm, an algorithm for breaking line segments. If this process is repeated a large amount of times one would get shapes that are similar to mountain ridgelines. Figure 2 shows an illustration of the midpoint-displacement algorithm running three iteration.

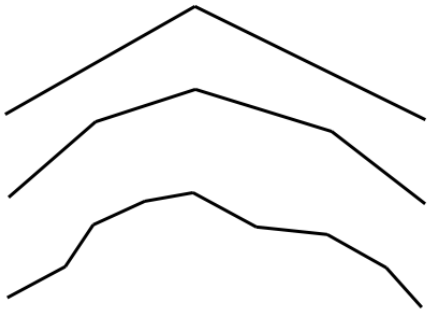


Figure 2: Midpoint-displacement in two dimensions run for three iterations on a line segment

The diamond-square algorithm is based on this idea but functions slightly differently since it is applied in three dimensions instead of two. When the principle is expanded from two to three dimensions the data structure is expanded as well. Instead of using a one dimensional array or list that is commonly done, a two-dimensional array is used instead, a heightmap.

The algorithm takes in a heightmap as a parameter and populates it with values. When the algorithm finishes, the heightmap will contain terrain data. Since the algorithm is based on the idea of large line segments being broken down into shorter line segments, the heightmap must have a few initial lines to break. A few points in the heightmap must be manually set before the algorithm can execute. Setting initial height values that dictate the terrain shape manually is called seeding. The four initial points to seed are the four corners of the rectangular heightmap. Seeds should have different values and be larger than ∅.

The algorithm then performs two steps: a diamond step and a square step.

In a diamond step, a square of four points is taken. A new value at the intersection of the points is generated. The value is an average of the four values plus a small random offset to create a more natural looking result. For instance: suppose the four corner points are (0, 0), (0, 2), (2, 0), (2, 2). The value of the middle point (1, 1) is calculated by averaging the four corner values and adding a small offset to it.

The square step functions similarly. Instead of taking a rectangle, a diamond (a cross) of four points is taken, and just like the previous step, a value is generated at the center of the diamond by averaging the four diamond points plus a small random value. Points (0, 1), (2, 1), (1, 0) and (1, 2) would be used for calculating (1, 1).

For each iteration across the heightmap more and more values are added until it eventually is full. Figure 3 illustrates how this is performed on a 5x5 heightmap. The leftmost heightmap is the initial heightmap, then the square step is performed, the diamond step performed four times, the square step performed again four times and finally the diamond step performed twelve times. In total four iterations are performed, reaching iteration level four, before the heightmap is fully populated.

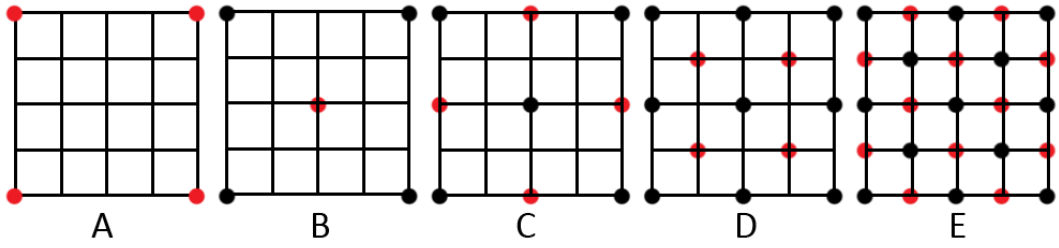


Figure 3: Example of terrain generation on a 5x5 heightmap

The algorithm performs the diamond and the square steps until a specified iteration level is reached, a specific amount of lines broken, or until the heightmap has all values set.

Because the diamond step takes a square of values with one point in the center for calculation, for instance points (0, 0), (0, 2), (2, 0) and (2, 2) when setting (1, 1) in the center, the heightmap has to have the dimensions: $(2^n) + 1 * (2^n) + 1$. The diamond step will need a point to be perfectly in the center. A few possible heightmap dimensions are then: $29 * 29$, $1025 * 1025$ or $2049 * 2049$.

The algorithm has a major feature that can both be good as bad, edge cases. If performing operations on a point close to the edge of the heightmap, for example point [0, 5] (left edge, column 5), and one would like to get the value of the point left of it for the square step, one would realize there is no point to the left. Instead there are two solutions, either use a random value, or wrap the heightmap and get the point from the opposite side. Perhaps point [16, 5], if it is a ∅ indexed 17x17 heightmap.

Because the heightmap can be wrapped like this, one can create heightmaps whose one side would fit perfectly with the opposite side if they were to be instanced, copied, and put next to each other. A heightmap whose top edge fits with the bottom edge and the left edge fits with the right edge could be copied and repeatedly placed in a grid to form a repeating infinite terrain. Video games of the 1990s and early 2000s such as Delta Force 2 (Novalogic 1999) took

advantage of this to create an illusion of infinite terrain when memory was limited.

The disadvantage is that nonrepeating infinite terrain cannot be created easily. Each generating heightmap would search for a non-existent heightmap next to it for lines near the boundaries to break. Controlling the terrain features can also be difficult. The four corner seeds will decide the basic shape of the terrain, but for more precise control one has to seed much more. If one wishes to place a mountain peak at a certain point it is most often not enough only to palace a large value at that point. Chances are that the point is at a late iteration level and that the point will not be taken into account until it is too late, creating spike like artifacts. View figure 4 for an example.

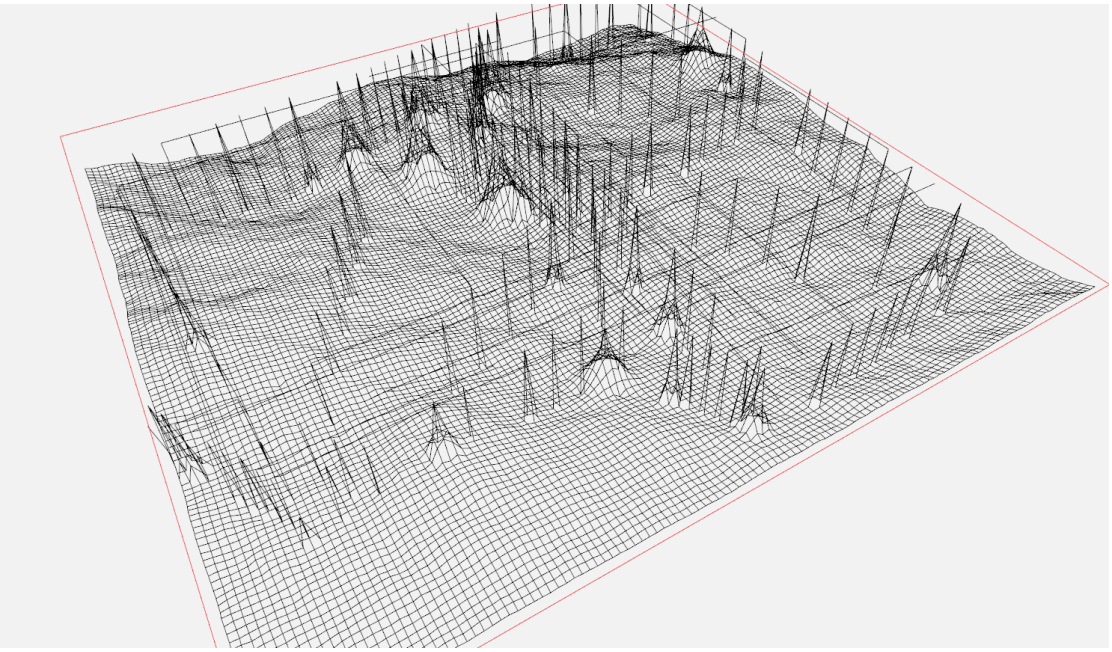


Figure 4: Example of a diamond-square heightmap seeded at a too deep level

To solve this one has to seed elements around the target point through multiple levels to ensure they are all taken into account. This can be a demanding and complicated procedure to apply if the heightmap is large as many values must be manually processed and set.

2.3 Perlin Noise

The Perlin noise (David S. Ebert 1994) algorithm is a good algorithm for generating data which should be random looking but still have some consistency, for example generating infinite terrain, rivers or textures.

The general principle behind the algorithm is to combine different wave functions with varying periods and amplitudes in order to create more detailed ones. This is done by combining the two components: a noise function and an interpolation function.

The noise function works as a seeded random number generator working with two seeds. When provided with a number as input it will return a number based on it. It will always return the same number for the same input. Additionally a number, also called a seed, can be set for the whole function, randomly changing the returning numbers. The function is called random because there should be no visible correlation between the retuning numbers.

Because the function is used to generate mountain ridgelines there should not be too steep differences between the returning values, otherwise unrealistic terrain will emerge. A maxima peak followed by minima after one meter followed by another maxima peak after another meter. Therefore a smoothing function is applied on the returning numbers bringing them closer together.

If in a XY-coordinate system, traverse through the numbers XØ to x10 and plot the resulting numbers in the Y-coordinate. It will look like random static data. But if it is interpolated between the points and drawn a graph it will resembles a ridgeline, or a wave. View figure 5.

Several waves with varying frequencies and amplitudes can be mixed together to create a ridgeline. Waves with low frequencies and high amplitudes will create the base ridgeline, while waves of higher frequency and lower amplitudes create more noisy results. When they are combined two such waves results in a wave that still has the large scale contours of the original low frequency wave, but also has more details due to the lower frequency wave. The waves are combined by adding their values together and dividing by the amount of waves. View figure 5 for an example.

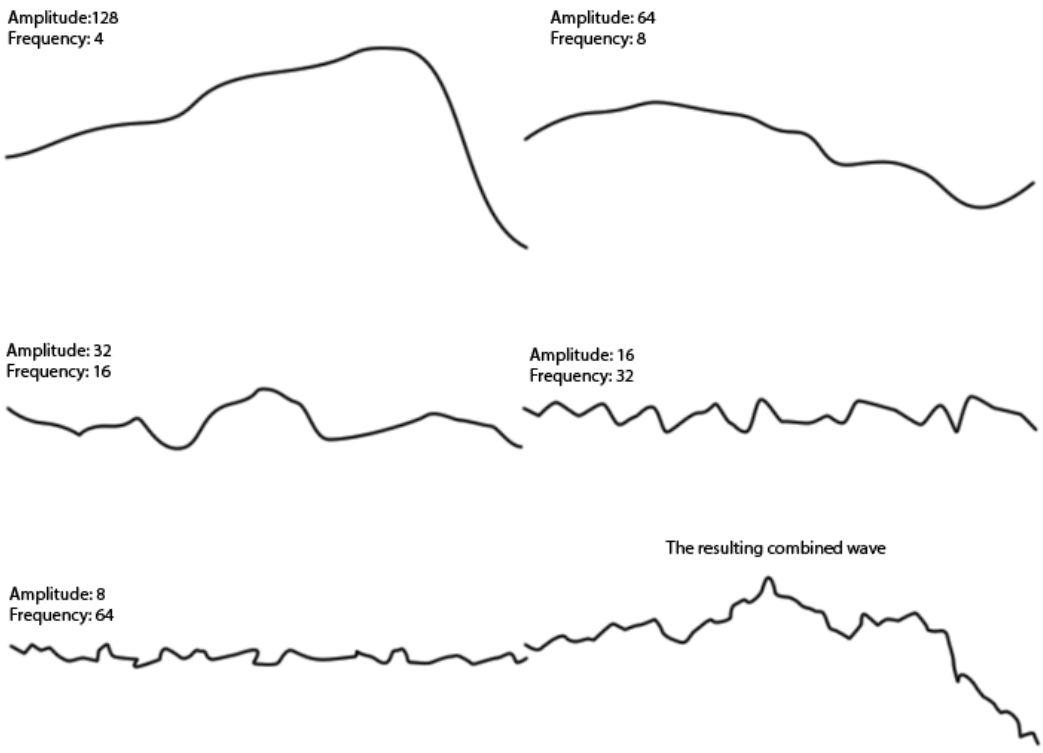


Figure 5: Example of 5 waves combined into terrain.

The main advantages of the Perlin noise algorithm is that it is relatively fast and deterministic. The algorithm only takes three parameters, the initial wave function description, the persistence variable which describes how the wave changes through every iteration and the XZ coordinate for which the Y coordinate will be generated. Typically the wave amplitude decreases by some amount for each iteration, creating smaller and smaller waves adding detail to the previous larger wave contour.

This means that if one declares a wave and a persistence, an infinite terrain can be generated by asking for the Y coordinate at any point of the world. The algorithm does not need to know the resulting values of the neighbouring patches as the diamond-square algorithm needs to.

The negative side of Perlin noise is that one cannot seed the resulting terrain at all. This means that without heavy modifications it is not possible to specify features to be generated, for example: one cannot decide where to place a mountain, only to know that there will be mountains somewhere. Also the generated terrain is of the same type in all directions as the same initial wave has to be used everywhere.

No matter how large a world is generated, there will always be a predictable amount of mountain peaks per square area. In a video game this would mean the player would experience the same type of terrain no matter where they go. View Figure 6 for a better understanding.

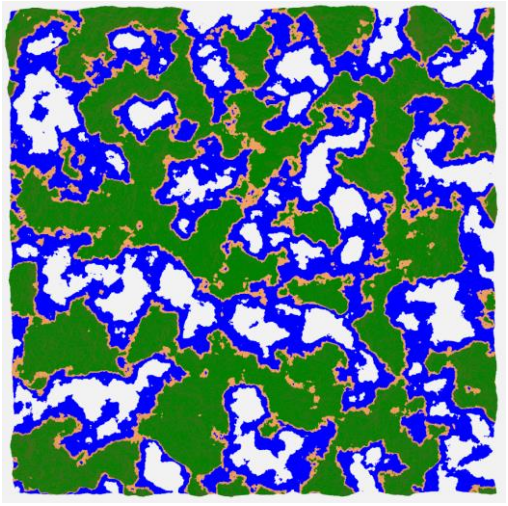


Figure 6: Top-down view of Perlin noise terrain with added color to indicate height differences. The higher points (green) outnumber the low point (white) by 3:1 evenly.

2.4 Same resolution heightmap stitching

Sometimes one wants to stitch two pieces of terrain, for example a low land heightmap and a highland heightmap with mountains. Placing an unrelated heightmap of mountains next to a heightmap of fields or lakes will look quite jarring. To solve this, one has to edit the terrain to form a good transition between the two heightmaps, this process is called stitching.

Terrain stitching is the act of stitching together different heightmaps which are either originally not thought to coexist or where the resolution differs. (Yotam Livny 2008) View Figure 7.

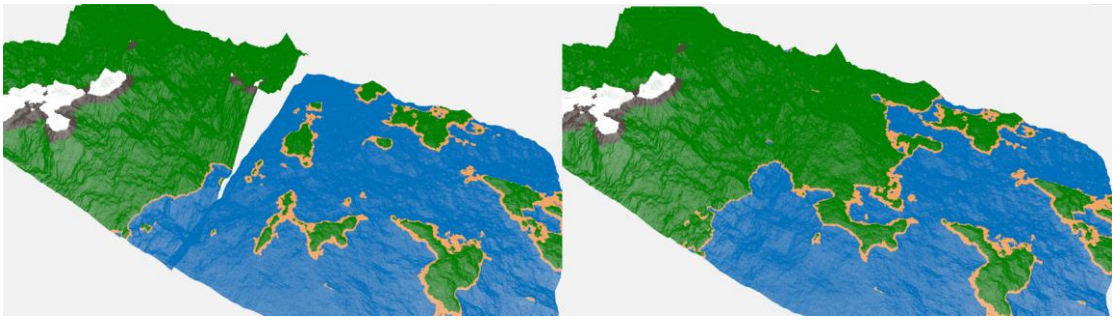


Figure 7: example of two heightmaps before and after stitching

Two patches have different resolution of their heightmaps when they have different array sizes for the same space. For example, if one array of size 10 represents Y coordinates from X world coordinate 0 to 100, while another array

of size 100 also represents Y coordinates from X coordinate 0 to 100. They both represent the same space, but the second array has 10 times more detail. This type of stitching will not be part of this project and therefore means of solving it will not be presented.

When said that two patches of terrain were not meant to coexist, it is meant they are not generated with the same parameters, or perhaps not even with the same algorithm, causing large gaps or errors where they meet.

There are multiple methods for solving this issue, the most common method is linear interpolation. This is done in a few steps.

1. Measure the height differences, Difference, between the two heightmaps at their meeting points. To know how much each heightmap should change in height in order to meet the other one, divide the Difference in two and save it as a variable: d

2. Decide how far into each of the heightmaps to interpolate, meaning how much of each heightmap should get altered in order to fit with the other heightmap. Call this amount: p. It can for instance be seven elements in.

3. To know how much each individual point should change divide the heightmap Difference with the amount of points that should change in total, call this variable: c, c=d/p

4. Go through all the points in the heightmap that should get altered. From the point most far away from the edge, to the one at the edge and keep count on how many are passed. Use a counter variable i and increment it for each element towards the edge that is passed. In the heightmap that should get lowered, subtract $c * i$ on each point that you pass and in the heightmap that should get raised add $c * i$ instead.

This will result in two heightmaps that fit together. The method is used for heightmaps whose terrain is treated as squared or rectangular. Finding a stitching algorithm that works on heightmaps of any arbitrary shape is part of this project.

2.5 Diamond-angle

The Diamond Angle Algorithm (Julian 2009) is an algorithm used for calculating the angle between two vectors connected to the same point. The algorithm is able to precisely calculate the angle between the vectors in a 360 degrees circle using very few divisions and neither sin, cos or atan2 functions, making it faster than most angle calculating methods.

Instead of using the classical Unit circle together with Sine and Cosines it views the L1 norm of the Unit circle instead. Giving it a shape of a 45 degrees flipped

cube instead, a diamond. This allows use of simpler mathematics, the straight line equation.

$$y = mx + b$$

The algorithm does not return a Euclidean angle. It returns a value between one and four instead, compared to $-\pi$ and π the Unit circle provides while retaining almost the same precision. Meaning that one can calculate an angle using the diamond angle function and convert it to a Euclidean angle if necessary.

But because the returning value should be between zero and four, and the line intersecting the diamond may do so on either the positive or negative side of the origin, the method will look a little bit more complex than just the straight line equation.

If (x, y) is a point in a Cartesian coordinate system, then the following method will return the diamond angle between the point and the origin.

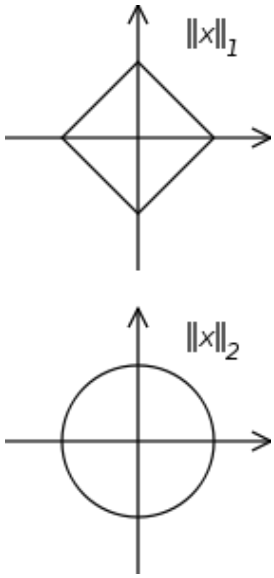


Figure 8:A classical Unit circle and a L1 Norm Unit circle

```
function DiamondAngle(x, y)
{
   if (y >= 0)
      if(x >= 0)
         return y/(x+y)
      else
         return 1-x/(-x+y)
   else
      if(x < 0)
         return 2-y/(-x-y)
      else
         return 3+x/(x-y)
}
Code from: (Julian 2009)
```

The algorithms angle calculating precision, as one will later on see, is sufficient for use in this project and because it does not use complicated methods internally, it is suitable for when often needing to calculate and compare large amounts of angles.

3 Execution and Theory

3.1 Research

The research phase began by looking into different methods for procedurally generating terrain. The two most popular methods for this were the Perlin noise and the diamond-square (Alain Fournier 1982) algorithms. Both algorithms have certain advantages and disadvantages, making each one suitable for generating specific type of terrain.

Perlin noise will generate good terrain based on a wave given as a parameter. Because it only requires an initial wave and a coordinate a terrain height for, it is well suited for generating infinite, boundless, terrain with continuous features, such as plains or hills. The diamond-square algorithm on the other is a recursive algorithm that works by breaking up lines until a ridgeline appears. View Chapter 2 for detailed explanations.

Once a few initial values or waves are given to the algorithms, they will generate terrain based on them. But neither one of these method allows for any real customization of the generated terrain. While they are useful for generating terrain of a certain type based on the parameters, they cannot change the terrain type they generate as they progress into a certain direction. They generate the same type of terrain in all directions, for example hilly terrain or lowland terrain.

Continuing the research an interesting paper was found describing how graph networks can be used to create two-dimensional worlds and levels for video games. The paper described a system in which rules could be created using grammar and an algorithm for generating graph networks out of it. (Adams 2002) (Rozenberg 1997)

For instance, one may define that node type Lake may only be connected to node type Plain or other Lakes and that the edge to the other Lake may be a river. This system allows one to dictate the way the terrain can be shaped or laid out. But the papers only focus on generating terrain layout, not generating heightmaps or other features. Graphs are generated and the hollow areas inside of them are colored in order to represent different features, such as blue lakes or dark green forests.

These combined works and approaches laid the foundation for this paper. Graphs could be used in a larger extent for terrain generation than they currently are. If one would develop a three-dimensional system that takes in a terrain layout as an undirected network graph with different terrain types assigned for each hollow area, new and detailed terrain could be created with more control than before. One could use a graph as the base for the terrain, decide which areas are lakes, plains or mountains, and generate terrain based

on that. The terrain would not fit because it is generated without consideration for each hollow area, but it could be processed to fit.

3.2 Design

The general layout of the method presented in this paper will be show here. The general idea begins with a graph, an undirected network of nodes, whose fields could be defined to contain different types of terrain with different features. Convert the graph, which dictates the terrain layout, to actual terrain using proven algorithms such as the diamond-square and the Perlin noise algorithm, and stitch the resulting terrain into a smooth single piece.

The overall steps for generating such terrain would involve:

- Define terrain type containers for terrain description. They should be able to describe terrain, how tall a mountain peak should be or how a plain should be defined. These will from now on be referred to as: terrain description.

- Generate or create an arbitrarily shaped graph, an undirected network of nodes, to represent the layout of the terrain.

- Find all the hollow closed areas in the graph, which from now on will be referred to as: fields.

- Randomly, procedurally or manually assign a terrain description to each field.

- Generate terrain for each field according to the terrain description assigned to it.

- Stitch the terrain of each field to the terrain of its neighbors, so they fit together. Two fields are neighbors if they share a common edge.

In order to perform said points, the following questions and problems need to be answered and solved:

- Define what a terrain description is and how to define it. How should it be implemented in a programming language and what information should it contain?

- What data structure should be used for the graph and how should the graph be generated?

- How does one find the fields of the graph in a fast enough manner?

- What algorithms should be used for generating the terrain in the fields and what data structure should be used for storing the terrain?

- How does one stitch arbitrarily shaped Fields with other neighboring arbitrarily shaped fields?

The following sections will give possible solutions and suggestions to these question.

3.3 Terrain Description

A terrain description is defined as a data structure that contains the name or ID of the terrain description together with all of the data necessary to generate terrain of some sort. Of some sort because this can be done in multiple ways. In this project only Perlin noise (David S. Ebert 1994) and diamond square (Alain Fournier 1982) are used for generating terrain, but any other algorithms could be used as well. As terrain description is used to generate terrain, the data it contains are the parameters for the two algorithms. Depending on which algorithm is used different data will be contained.

Please view Chapter 2 if unfamiliar with Perlin noise.
The parameters necessary to generate terrain with the Perlin noise algorithm are:

- Amplitude - The initial maximum height of the terrain. For a mountain define it as a large number for a high peak, or a low number for a valley or ocean floor.

- Frequency - The initial frequency for the random noise wave.

- Frequency change - A decimal number that describes the change of the frequency through each iteration, typical value would be two.

- Amplitude change - Persistence - A rational value that describes the change of the Amplitude through each iteration, a typical value would be 0.5. This value dictates how much the maxima of the next iteration wave should be lowered by multiplying the current maxima with it.

- Number of Octaves - The amount of iterations the Perlin noise algorithm should perform. The amount of waves it should generate and combine.

3.4 Graph

The graph is defined as an undirected network of nodes. Each node is represented by an object Node and each Node contains references to the Nodes

it is connected to as well as its coordinates in the absolute world space. Each connection is an edge.

This project will not go deep into discussing methods for generating graphs, because there is already a large amount of research done on the subject. One could for instance use Voronoi diagrams (Henrik July 30th 2005), hexagonal graphs, square graphs, or custom graphs. This project only encompasses the parts before and after a graph is obtained.

3.5 Sectioning

Once a graph is procedurally generated or manually provided, all possible fields inside of it need to be found.

This is a widely discussed and classical problem of graph theory in computer science. One solution to this problem can be to use diamond angle together with right-side walking. Please read about diamond angle in Chapter 2 if unfamiliar.

The idea is to traverse though the graph and walk in circles between the nodes across the edges connecting them. Nodes are traversed through by walking on the edges between them. Here right-side walking is used while traversing the nodes. Whenever a node is reached, the next node to visit is determined by taking the node to the right side. As a result of always taking a right side node on a closed field, the route will end at the starting node when traversing is over. One will arrive back to the node one started from.

1. Select a node of the graph.

2. Find all of the neighboring nodes it is connected to and pick the one with the smallest positive angle. The smallest angle being viewed in a counterclockwise order.

3. Continue to that node and note down the previous node. Repeat step one and two until an already passed node is reached again. This means a circle has been traversed and a filed consisting of the noted nodes is found. Figure 10 illustrates how a possible path may look like.

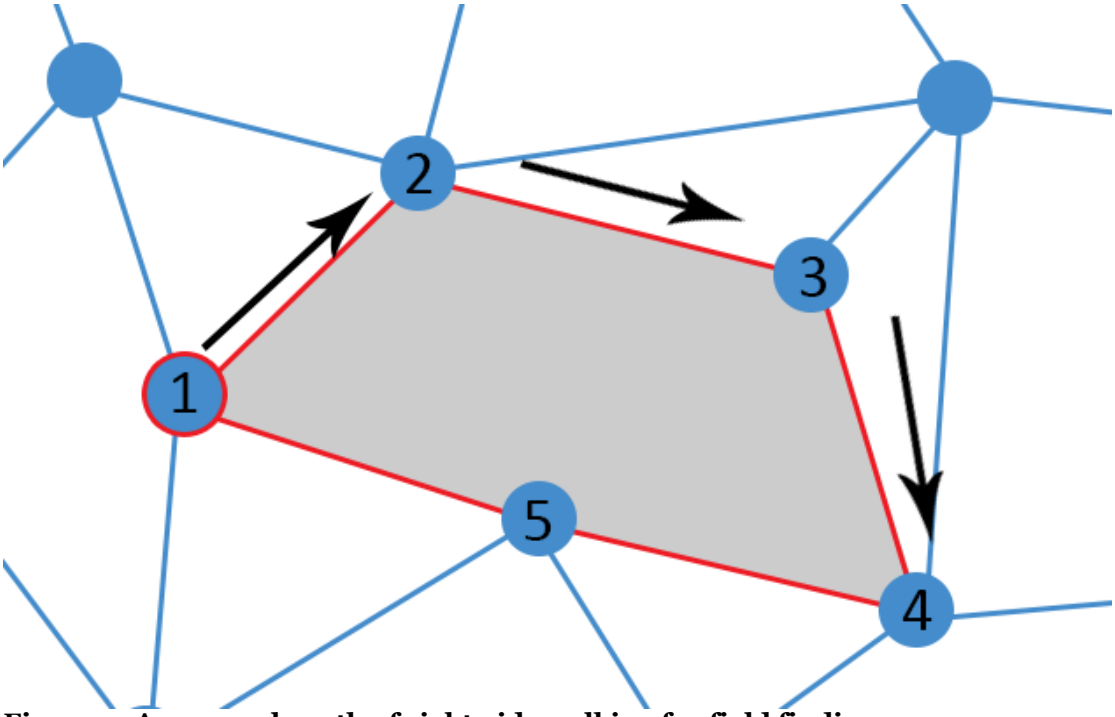


Figure 9: An example path of right-side walking for field finding

Because the diamond angle method is fast at calculating the angle and good for comparing the angles between different vectors, hence it can be used on relatively large graphs without too much computation power. The advantage of using diamond angle is visible when comparing to a few other classical methods with heavy computations using Sinus and Cosines methods.

A field is represented as a list of nodes in the order they were traversed. A field edge is the connection between two consecutive nodes in the list. After sectioning all the fields, each field is assigned with a terrain type, manually or procedurally.

3.6 Creating and populating heightmaps

The terrain in each field is stored inside of a heightmap. Even though the field itself is of arbitrary shape, the heightmap remains rectangular since it is only a simple two-dimensional array. Before the actual terrain is generated a large enough heightmap to cover the whole field needs to be created. This means that the bounding box of the field needs to be calculated, a rectangle that encompasses the whole field.

To find it one needs to know the width and height of the bounding box. A simple way to calculate them is by iterating through the fields nodes noting down the minimum and maximum X and Y coordinates. Subtracting the largest X coordinate with the smallest X coordinate gives the width of the field, and

subtracting the largest Y coordinate with the smallest Y coordinate gives the height of the field.

Now that the necessary width and height in absolute values are known, it is time to decide how many elements the actual heightmap array will contain. This depends on the area each element should take up in the absolute space and is an arbitrary value the user should set. For instance an element could take up the XZ area of $2 * 2$ world units and describe the value of a Y coordinate.

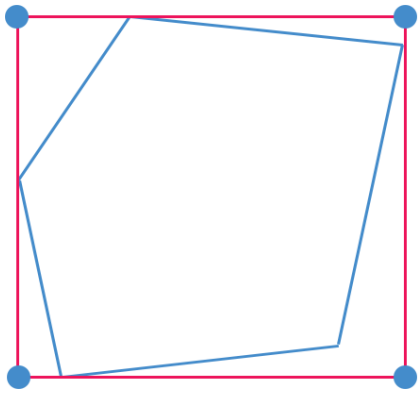


Figure 10: Red field rectangle wrapping a field

When the width and height of each element is chosen, calculate the amount of rows and columns the heightmap should have. The amount of rows is bounding box width divided by element width, and the amount of columns is bounding box height divided by element height.

Since the heightmap only contains the Y coordinates, and the indices represent XZ coordinates, each heightmap would start from the origin. In order to have varying ZX coordinates each heightmap is assigned a position offset to the ZX coordinate. The system in this paper views this offset to be at the bottom-left corner of the array. For instance its X component could be the smallest X value found in fields nodes and the Y component the largest Y value contained in its nodes.

For each field a heightmap that covers the entire field is now created. Each field now has a terrain description and a heightmap assigned to it. If the terrain description inside of a field contains Perlin noise parameters use Perlin noise and generate terrain for the entire heightmap. Likewise if diamond-square parameters were used generate terrain using diamond square.

3.7 Terrain collision detection

So far the fields are of arbitrary shape, but the heightmaps containing terrain for them are rectangular. This means that the heightmaps describe terrain that is outside of the fields. If two fields were to be placed next to each other, each field would have terrain overlapping the other ones terrain. Terrain sticking outside of the field needs to be removed.

In order to check that a heightmaps element is inside of the field use line polygon intersection methods. When generating a heightmap element one need to be able to tell whether or not it is inside of the field. If it is inside of the field keep it, if it is outside remove it.

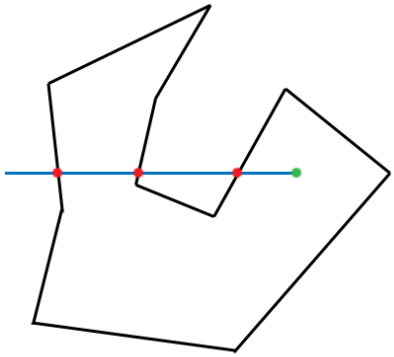


Figure 11: Collision detection for a point inside a polygon

View the field and its edges as a polygon. Draw a line outside of the field towards the element. Note down each time the line crosses an edge of the field. When the line reaches the element, count how many edges it crossed. If the number is odd the line went into the polygon and never went out, if it is even the line went in and out of the polygon again.

For each element in the heightmap, check if it is inside or outside of the field. If it is outside it needs to be removed. Because an element cannot just be deleted from a heightmap, a null value needs to be decided upon. In this paper a heightmap element is viewed as empty if its value is ∅. When rendering the heightmap no element with the height of ∅ will be rendered.

By now the terrain generated could look like in Figure 13.

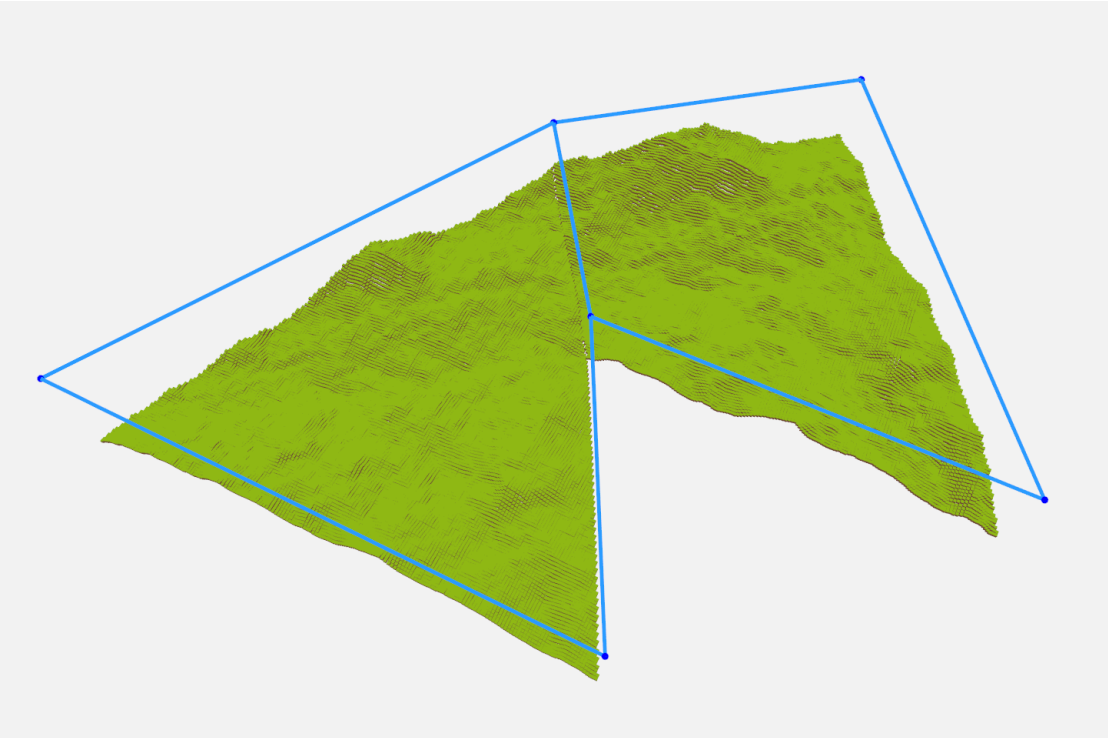


Figure 12: A pair of fields with their terrain after terrain collision has been performed.

3.8 Multiple field stitching

3.8.1 Base case

This section deals with stitching two adjacent areas together. It is closely related to the previous section, section 2.4: same resolution patch stitching. The same principal from section 2.4 can be applied when dealing with terrain and fields of arbitrary shapes.

Edges of each field are viewed as vectors. Before stitching, the algorithm will need to know how deep into each field it may go and alter the elements. Call this Distance. After deciding how much a field affects another, points within fields are parsed. One field will be stitched at a time. The field being stitched will from now on be referred to as Field A, the field stitching it with will be referred to as Field B and the edge stitched at will be referred to as the Edge.

Given a certain point, Point, inside Field A, first find out how far away it is from the Edge. If it is further away than Distance, it is outside the scope of the terrain stitching and leave it, if it is within Distance, this point should be processed.

The distance between the Point and the Edge can be found by viewing the Edge as a line segment in R2 and projecting the Point onto it. This will give the point Point On Line, shortened POL, which is the projection of Point on the Edge. The distance between the Point and the Edge is the length of the line segment POL to Point. The line segment POL to Point will now be known as the Normal.

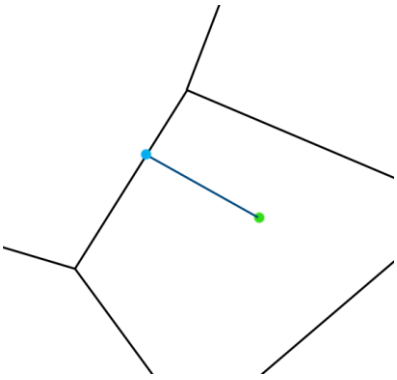


Figure 13: Showing the Normal between POL as a blue dot and Point as a green dot.

Now that it is certain there is a Point in Field A which should be changed to fit with the terrain in Field B, find out how much its value should change. Just like in section 2.4 this is done by first viewing the points at the edge between the two fields. Calculate how large the height difference, Difference, is between the two fields at that the contact point, POL.

Because one point is in Field A, the other one is in Field B and the POL can be in either one, take a step into each field from POL. By going back one element length into Field A the edge point in Field A is reached: A Point. Likewise stepping into Field B one element length the Field B edge point is reached: B Point.

To make the step normalize the Normal and multiply it by one element length. A Point is reached by adding the Normal to POL and B Point is reached by subtracting the Normal from POL. View figure 15 for illustration.

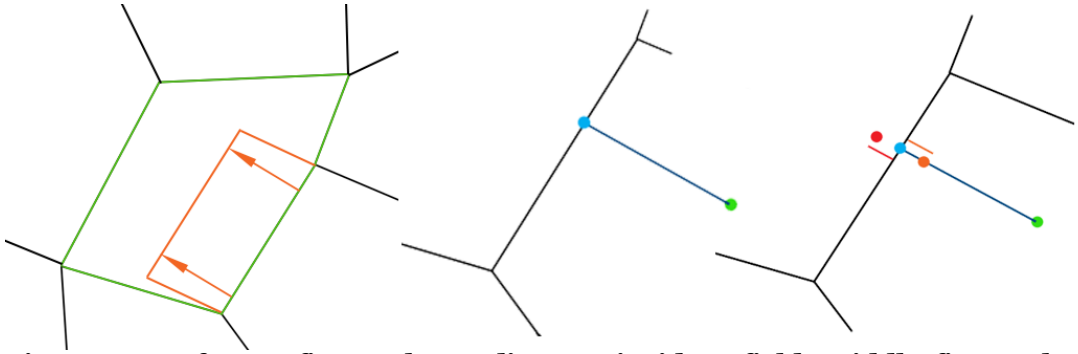


Figure 14: Leftmost figure shows distance inside a field. Middle figure shows Point, POL and the Normal. Right figure shows A Point and B Point being selected.

By subtracting B Point from A Point one will get the Difference. How much each side should change at POL in order to fit together. If both heightmaps should change so they meet in the middle, Difference should be divided by two and let each side change $(+-)\text{Difference} / 2$

Both A Point and B Point are now known. In order to calculate Difference the actual values need to be read from the heightmaps. Convert A Point and B Point to indices for accessing each heightmap.

For the X component of a point subtract it to the X component of the heightmaps position offset. This will give you the relative difference of the X components. If the difference is divided on the width of each heightmap element, the index used to access the element in the row dimension of the heightmap is obtained. The column index is calculated in the same manner. View Figure 16.

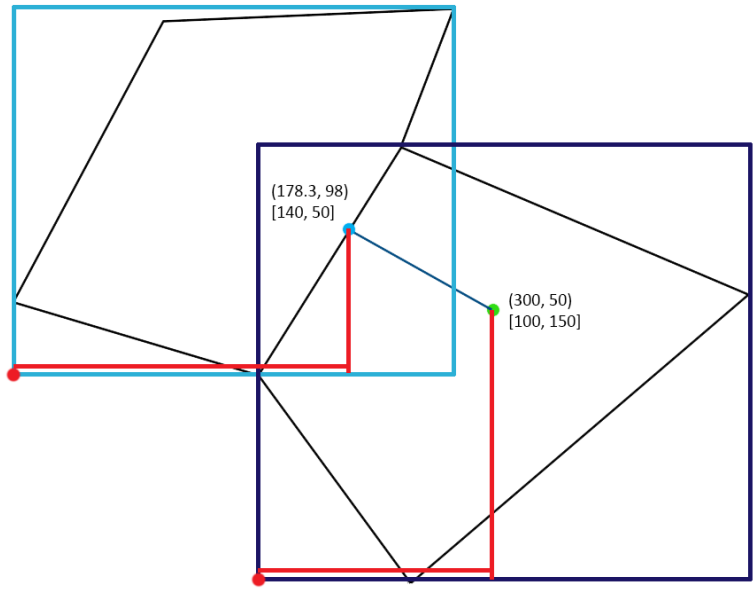


Figure 15: Converting absolute coordinates to heightmap indices

With the indices used to access the points now obtained, the values are read. With the height difference between the heightmaps at POL known, the actual change on the element at Point can be performed

The closer a point is to the Edge the more it should change and vice versa. Within Distance from the Edge, (Distance / Element Size) gives the amount of elements that could change. This amount is called Element Amount.

Difference is spread out along Distance. The gradient of the change is defined as Difference dividing Element Amount, which is also the value for the base change. Each value is changed by adding the gradient multiplied by distance from this certain point to the Edge.

Here is an example calculating the change of difference. For instance, if the element distance is 3, the Difference is 5 and Distance which defines how far away the stitching area extends is 10 then

$$(\text{Distance} - \text{Normal length}) / \text{Distance}$$

The result here shows how far away a certain point is from the Edge in percentage. Then multiply this result with Difference to get the amount of changes that needs to be applied. In this case: $(10 - 3) / 10 = 0.7$, and the change on element would be: $0.7 * 5 = 3.5$

If applied to every element within Distance in two adjacent fields, the algorithm will produce nice results. View Figure 17. If the graph is larger and there are multiple adjacent fields which need to be stitched, additional logic is needed to ensure that the stitching does not override the result of changes already made to certain Fields. The next section will discuss this.

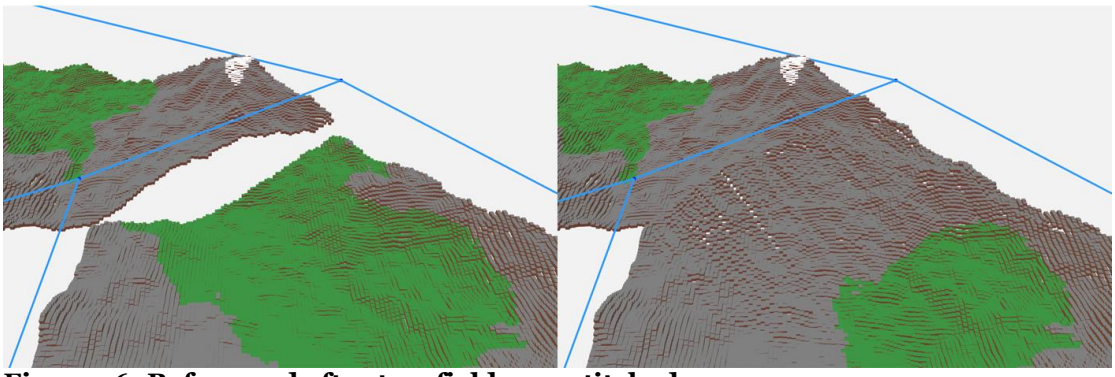


Figure 16: Before and after two fields are stitched

A key detail to notice here is that the Normal should be a Normal relative to a line segment, meaning that if the projection of the Point onto the Edge goes outside of the line segment, it should bend and become a line between the Point and the closest line segment endpoint. Due to this, the Normal either goes from POL to Point or an Edge endpoint to Point.

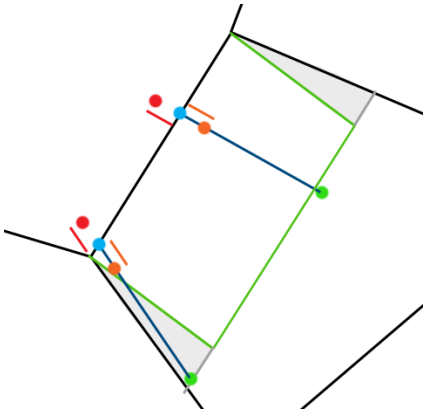


Figure 17: A bending normal example

This is very important in order to ensure that two fields do stitch together around corners also. Otherwise there will be abrupt height differences around the corner of certain endpoints when the angle between the edges is larger than 90 degrees. Figure 18 illustrates this. Notice, this is only necessary when stitching two Fields. Stitching multiple fields is illustrated at a later section.

3.8.2 Height Difference Altering

Using the above method when stitching two fields, Field A and Field B, the first thing to do is to decide at which middle ground they should meet. In the previous section it was mentioned that if one wishes to stitch two pieces of terrain and let them meet in the middle, then the following formula that should be used to calculate the difference is: $(+-)\text{Difference} / 2$. While this is true, the formula does change a little when applied in practical cases to: $(+-)\text{Difference} * \text{amount}$

Amount is a new parameter indicating how much of the possible Difference actually is to be used. In typical use-cases when both fields meet in the middle, Amount should either be 0.5 or 1. When the fields stitch, they do stitching one by one. After Field A has changed to half of Difference, Field B started to stitch. The Difference between A and B is measured again. This time B applies Difference completely.

In a case where both fields should meet halfway, Amount for Field A should be 0.5 and 1 in Field B. Depending on how much one desires each Field to change and how the change scale should be set, the values may vary. A key point to note here is that if the Amount at the first field stitching, Field A to Field B, is one, Field A will reach Field B in one go and there is no reason to stitch Field B to A also. This can be useful if performance is an issue and as little stitching as possible is desired. Though if the height differences between the fields are big, the results might look a bit too steep or crude.

3.8.3 Multiple field stitching

3.8.3.1 Multiple Field Issue

Previous sections deal with stitching two fields. While the method works well for simple cases, it has a flaw that needs to be addressed if more than two fields are to be stitched. When only stitching two fields, each field is only affected from one of its edges, but when fields in a graph are stitched they will be affected from multiple edges. This causes an interference issue close to the nodes, or close to the line segment endpoints.

When Field A is stitched in regard to Field B along an edge, Edge, the points within Distance from Edge will be affected. If then Field A is also stitched with Field C along a new edge, which shares a common node with the previous edge, the stitching will show signs of interference. As before the elements within Distance away from the new edge will be affected. Because the current edge shares a node with the previous edge, they will both alter the elements close to the common node. View figure 19 for a better understanding.

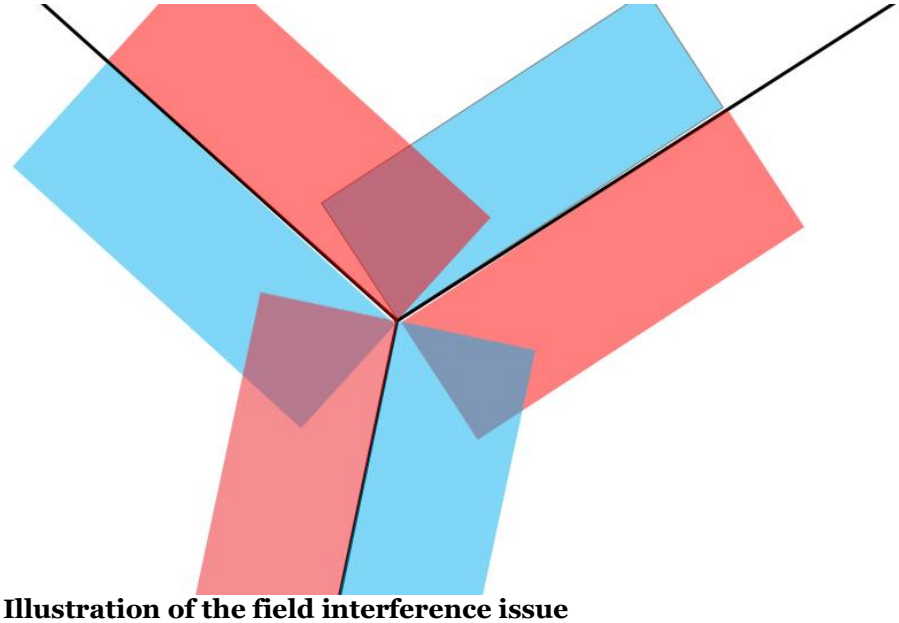


Figure 18: Illustration of the field interference issue

3.8.3.2 Point Stitch

Interference can be solved by dividing the stitching into two sub cases, the stitch in the middle of the edges and the stitching close to the nodes.

Shorten the strip of the edge currently stitched on from both sides. Instead of stitching along the entire edge, only stitch along the middle of it, leaving it un-stitched close to the endpoints, the nodes. Using a new variable called Radius that for now can have the same value as the previous value Distance. Instead of stitching between endpoint A and B of the edge, stitch only between the

segment (A + Radius) to (B - Radius), the sub segment of the edge shortened by Radius from each side.

Because stitching is only performed in the middle of the edge, there is no need to have a rotating normal anymore, as every point will have a projection on the edge and not outside of it. Also because the rotating normal does not need to be taken into account, the code can be simplified and made faster. This stitching method will be referred to as normal stitch.

The following Figure 20 shows the result of such an operation.

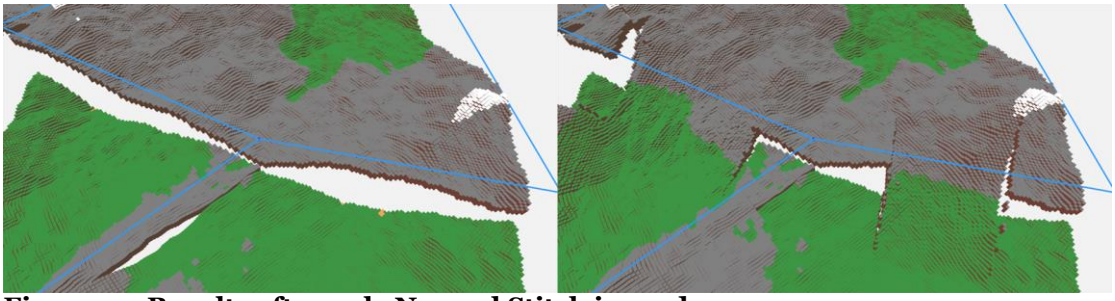


Figure 19: Results after only Normal Stitch is used

Next, stitch the elements close to the endpoints that are within the segment Radius. The segment Radius in this case does not mean the distance from the endpoint to the element, it is the distance from the endpoint to the projection of the element on the Edge, POL. If the current element is at Point, and its projection on the Edge is POL, then the distance is measured between POL and the endpoint currently stitching at. Meaning the stitch area will be the rectangular area whose one side is going from an endpoint to the POL and another side from POL to Point.

The Areas would in that case look like the illustration bellow, figure 21. The green areas are stitched with Normal Stitch, while the blue ones are stitched Point Stitch.

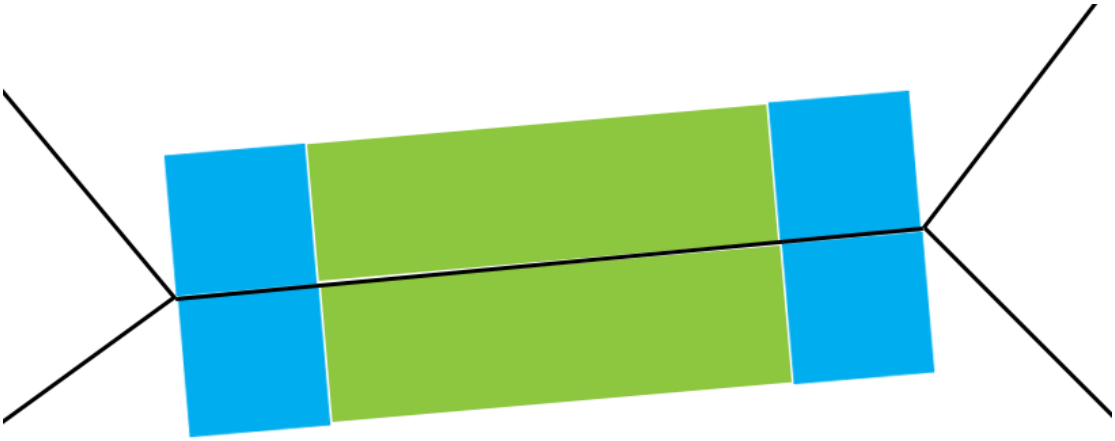


Figure 20: Example of an edge divided into six stitching areas. The blue areas use Point Stitch while the green areas use Normal Stitch.

The idea behind the Point Stitch is to only stitch the elements to their closest edge. This is to prevent the stitching with Field A at edge A to affect the stitch near the edge B with Field B close to a common node. It would distort the elements near Field B which may already have gotten stitched.

To implement this one can divide the area of the field from a nodes point of view, into two areas. If inside of Field A stitching at Node A and looking towards the center of the field, imagine an area on the left side and an area on the right side. An area separated by a vector with one side on the left adjacent to Field B and edge B and an area on the right side adjacent to Field C and edge C. The vector sought after does not need to go to the center of the field, but be the middle vector between the two edges converging in node A.

This vector can be found by finding the combined opposite vector of the node and its two connected edges. Call this vector Separator. To find it view the two edges meeting at Node A as two line segments going out from Node A. By subtracting Node As coordinates from both line segments endpoints, the segments become two vectors going out from origin. When both vectors then are added up a new vector emerges. A vector pointing in the middle between the two edges. Add the vector to the position of Node A and it will be going between the two edges into the field from Node A. This vector will be called Separator

Figure 22 shows the resulting scene. The separators are drawn as red lines for both of the endpoints and in all of the areas.

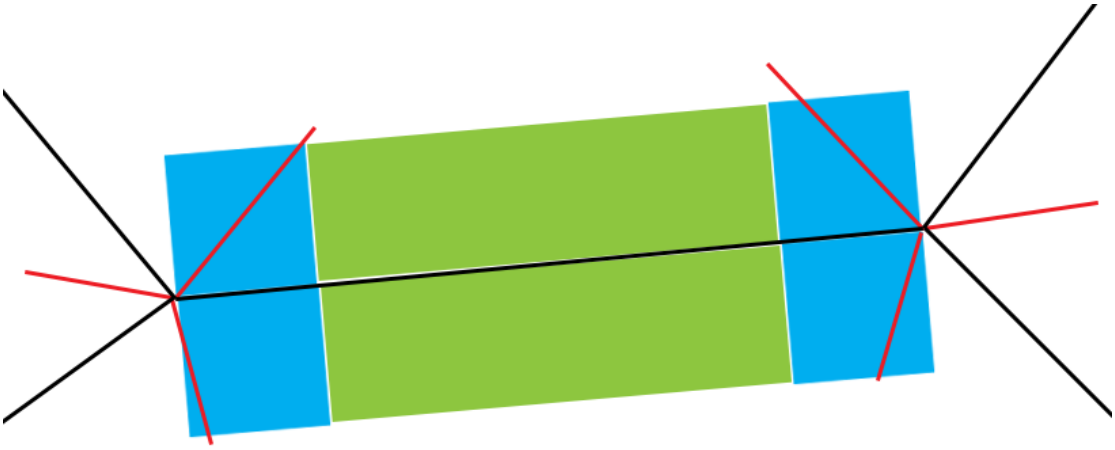


Figure 21: Showing two fields with terrain divided for Normal and Point stitch together with Separators in red

By doing this, each field will from a nodes point of view be divided into two areas, an area on the left next to Edge A and an area on the right side next to Edge B. The element stitching in one area should never be allowed to affect the elements of the other area.

3.8.3.3 One side stitch

For each element that is stitched its position, its POL, its normal, the length of the normal and the distance its projection has to the node currently at is known. Because the Separator vector is also known, it is now possible to calculate whether or not the element is on left or right side of it.

Simplify the problem and viewing it as a classical linear function with the formula $f(x) = ax + b$. For clarity the X axis can be seen as going along the vector going out from the current node along the Edge. The line gotten from the equation should be the Separator and X is a point on the Edge.

When an element is chosen use its segment Radius distance as the X parameter and find the point in which the elements Normal relative to the Edge would meet the Separator, by using the linear function $f(x) = ax + b$. The value one would get would be the distance from that POL to the Separator. Practically it would be the maximum Normal length an element at distance x from the current Node may have from the Edge and still be stitched while being inside of the correct area adjacent to the Edge. Call this distance the Maximum Distance.

If the length of the Normal is larger than the Maximum Distance, the element is not to be considered. It is then outside of its side of the field, on the wrong side of the Separator. If it is smaller than Maximum Distance, then it should be stitched. Figure 23 illustrates this.

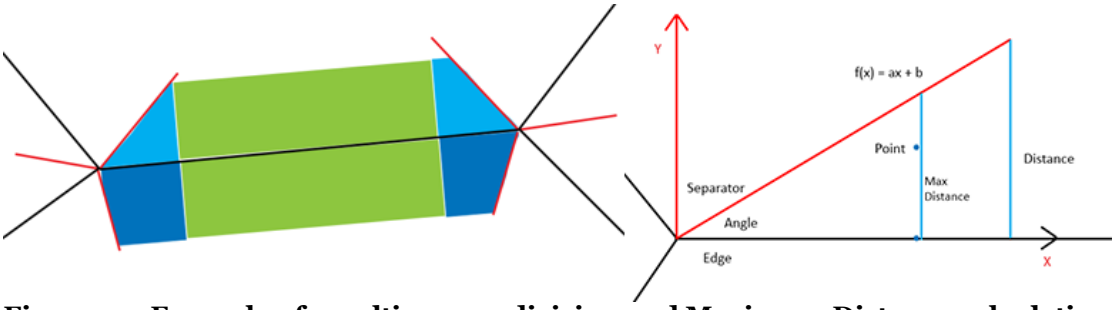


Figure 22: Example of resulting area division and Maximum Distance calculation

3.8.3.4 Dynamic Distance Value

The current element selected is then adjacent to the Edge and is on the correct side of the Separator. If this element would be stitch as before, it would produce mostly fine results, but it would still produce interference with the stitching of other edges. While the element will be on the correct side of the Separator and will never be an element another edge also tries to stitch, the elements along the Separator could look strange. There will be a height gap along the Separator. This is because the old Distance variable is still used to divide and decide how much each element should change in value. The element this Distance value will think is the element furthest away from the Edge that should change the least will be a value far away inside the other wrong side of the Separator.

In order to solve this, use the Maximum Distance variable instead of the Distance parameter.

3.8.3.5 Radius Sync

There is one last detail yet to mention which has been saved until now to avoid confusion. If the reader recalls at the start of this section it was mentioned that the variable Radius for now could be seen as having the same value as Distance. This is not really true. While the same value might be used for both variables, and in overall good stitched terrain is produced, there will be a small amount of elements with a noticeable height difference at the border between the Point Stitch and the Normal Stitch.

This is because the Normal Stitch uses Distance and counts that the element to change the least should be Distance away from the Edge and that each element should change $((\text{Distance} - \text{Normal.length}) / \text{Distance} * \text{Difference}$ .But because the angles between the edges connected to a node are arbitrary, the Separator will look different each time.

If the Edge is viewed as the floor cathetus in a right-angled triangle and the Separator is viewed as the Hypotenuse then the height cathetus can be seen as them Maximum Distance. The resulting Maximum Distance furthest away from

the node, the distance Radius, must have the same value as the Distance used in Normal Stitch. Otherwise at the border between the two they will come to different conclusions regarding how much each element should change.

The solution to this is simple. After finding the Separator measure its angle to the Edge and use Tangent to calculate the length the floor cathetus should have in order for the height cathetus to match Distance with regard to the angle.

Because one can view the edges as vectors in R2, use the Dot Product of the vectors and acos to get the angle between them. $\text{angle} = \text{acos}(\text{v1} \bullet \text{v2})$. Then the new Radius value will be $\text{Distance} / \text{Tan}(\text{angle})$ and the length of the Hypothenusa, or actually the Separator, should be $(Radius^2) + (Distance^2)$.

3.8.3.6 Pinching

In most cases the graph will by now look neat and well stitched. But there is one last corner case that should not be forgotten. If multiple fields meet in the same node, the terrain will be stitched from the edges towards the inside of the terrain until the distance Distance or until the Separator is hit. But because the values under the Separator are never stitched, one the values left or right, they will always remain in their original position.

This means that if a low lake element is close to a node and is stitched to an adjacent mountain element also on a Separator and also close to same node, neither will change and the height difference between them might be noticeably large.

To solve this do a pre-stitch node sync. Before performing any type of stitching, go through all of the nodes. In each node find out which fields are intersecting and find the element in each field closest to the node. Add up their values and find the average value. Then go into each field and sync it so that the element closest to the node reaches the average value, do this by subtracting the average value from the closest elements value. This will be the Difference in height.

Change the element according to the Difference so it meets its adjacent peers near the node at the same height, and sync the rest of the elements in the field. That is, pick an element in a field, calculate its distance to the node currently syncing at and using the same formula as previously to calculate the new value for the element.

$$\text{Difference} * ((\text{Distance} - \text{Normal length}) / \text{Distance})$$

This operation does not take very much to perform and it only deals with the elements close to the node. These fields are not stitched, but are synchronized so that their elements near the nodes meet in the same point.

After applying the whole stitching procedure from above, the result should be a nice stitched terrain graph. While some of these procedures may seem demanding, one should remember that most of them are only done two times for each edge in each field for the entire stitch of the graph. It is a small piece of the stitching time. Each edge only needs to have two Separators calculated on each side, only two Radius values calculated and each node only synchronized once. Figure 24 shows the final result.

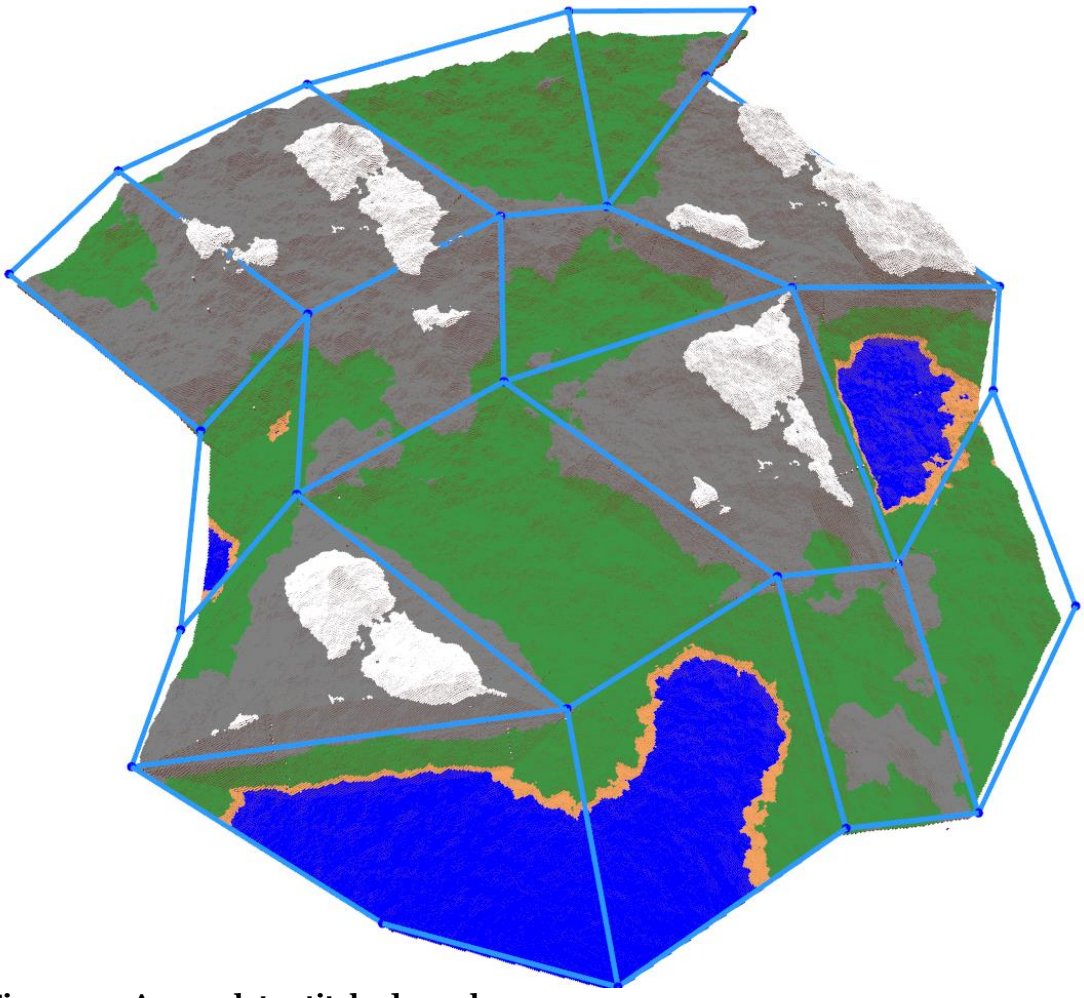


Figure 23: A complete stitched graph

4 Results

The C# implementation of the algorithm is divided into several steps, similar to how they were described in the sections above. It was mainly designed to be proof of concept code, to run decently well without being too abstract so that it also could be read and understood easily and help one grasp all of the procedures. The result is a small library containing all of the classes and functions necessary to generate graphs, terrain and to merge them for a good resulting terrain. It uses the vector classes of the OpenTK (OpenTK 2015) graphics library, but this can be changed to any other implementation of vectors and their corresponding functions such as dot product calculation, normalization etc.

Graph generation and terrain generation algorithms are part of the code, but are not the main goal of this paper. The code provides a simple graph generating algorithm, but this is only an example. Likewise the algorithm used to generate terrain, such as the Perlin noise and the diamond square algorithm, are here as examples and could be switched out for any other algorithm or implementation one sees fit. Therefore the result section will mainly be concerned with the stitching, the stitching performance and the results of it.

The different types of terrain can be described in two ways, as Perlin noise parameters or as diamond square parameters. Each field can have a terrain description assigned to it. The stitching takes in a graph including all of its nodes, edges, fields, terrain descriptions assigned to each of the fields together with a parameter deciding how much each field may change. It outputs the same graph where the terrain of each field is modified to fit that of the neighboring fields. This allows users to create different types of terrain for each field and to then merge it into one large cohesive terrain.

4.1 Performance

The performance section will mainly be concerned with the performance during stitching, as the graph and terrain generation algorithms are arbitrary and user replaceable.

When stitching a field relative to an edge, Distance indicates how far into the field elements may be affected by the stitching. A higher Distance value means that more elements in each field will be considered and that the transition between two fields will be smoother, spread out over a larger area and perhaps not as steep. One can assign a different Distance variable to each field though, allowing for more fine control over the features. If for instance a field with lower altitude is created to represent a lake bed, a larger Distance variable would produce a more dry lake while a smaller Distance would account for a steeper transition and a less dry lake.

The size of the total graph area will of course be a major factor affecting the execution time. More elements to stitch means longer execution time. The amount of fields in the graph and the amount of edges to stitch also affect performance. More edges mean larger amount of values that fall under the Distance value and more fields means more fields to stitch against each other.

For performance measuring the library containing a code implementation of the algorithm is run and the execution time of its functions are timed. The timing is performed using the built in C# .NET 4.5 Stopwatch functions and are displayed in seconds after each run. While execution time is highly relative from machine to machine, this can at least help to gain an understanding of the execution times involved for stitching. The machine used for timing is a Windows 8.1 ROG Asus G55V Laptop, using an Intel Core i5 3210M CPU with 3.0 MB L3 cache, 512 KB L2 cache, 128KB L1 cache capable of running 4 threads on 2 physical cores and 12 GB of DDR3 ram at 1600 MHz. If this was to be run on a desktop machine, the performance should be considerably better.

There are three test cases used. Each test case varies in the amount of fields and edges it contains and is tested on three different graph sizes with three different Distance values assigned. As mentioned earlier performance is most affected by the field count, edge count and the Distance value. Because the Distance value goes along each edge to check for elements within range, the general formula for the amount of elements one would pass is roughly:

$$\text{Edges passed} * \text{Average Edge Length} * \text{Distance}$$

Though depending on the graph this may not always be true since graphs with sharp corners will have very sharp Separators which will decrease the amount of elements on has to pass.

The graph is created on a rectangular plane where each side has the same length. The side lengths vary in the four cases, and thus the size of the terrain varies in four cases, as listed in the tables below. Each element size is two units.

Test one is created using a graph with side length of 2000 units, and thus considering the element size of two units, it contains about 2000000 elements. Test two has side length of 2500 and thus contains about 3125000 elements. Test three has a side length of 3000 and 4500000 elements and finally Test four has side lengths of 3500 containing 6125000 elements.

Each case is then tested running each test with three different Distance values, 60, 80 and 100. There is no point in using smaller Distance values due to the execution time being so small, and using a larger Distance value than 100 would simply cut into the opposing edges of the field. The Distance value also affects the performance in terms of milliseconds, as one will see below, and thus does not matter a great deal. It is mainly the different graph complexities that affect performance.

Each case is ran in an automated testing environment 15 times and the expected values for each dataset are calculated and displayed bellow. All measurements are performed using the built in stopwatch timer of the .NET Framework 4.5.

	Field count	Edge count
Case 1	16	40
Case 2	36	84
Case 3	64	144

Distance: 60

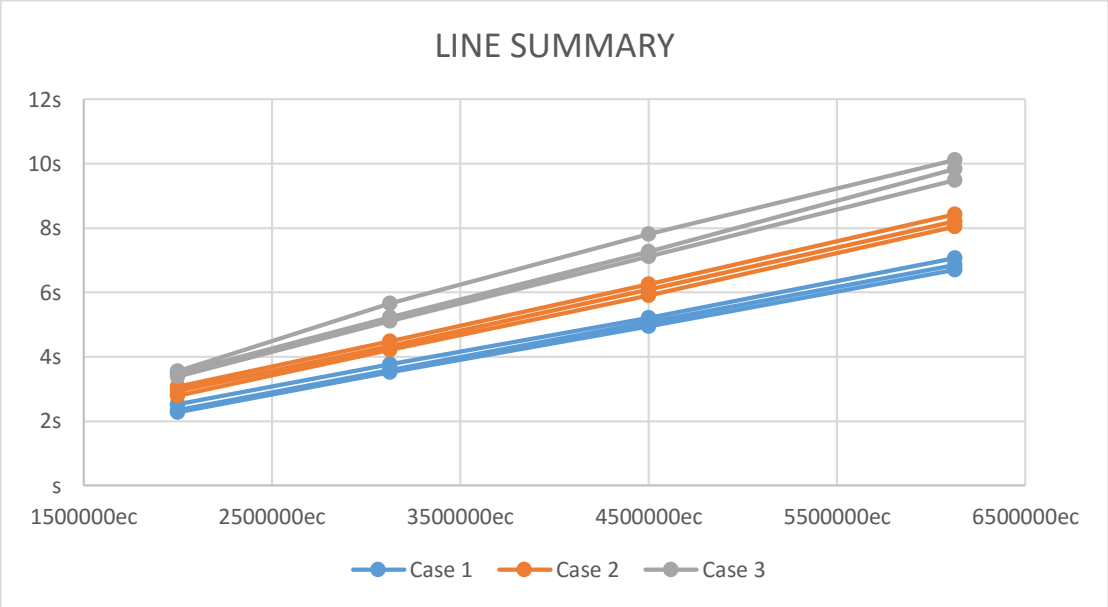
Element count	2000000	3125000	4500000	6125000
Case 1	2,27s	3,51s	4,95s	6,71s
Case 2	2,79s	4,21s	5,91s	8,05s
Case 3	3,39s	5,11s	7,12s	9,48s

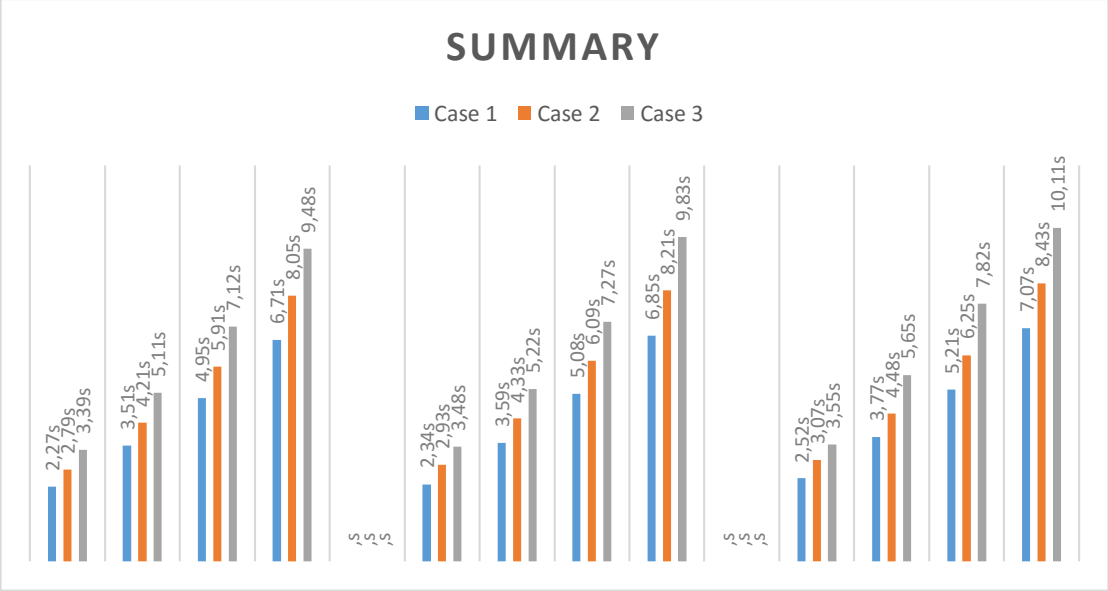
Distance: 80

Element count	2000000	3125000	4500000	6125000
Case 1	2,34s	3,59s	5,08s	6,85s
Case 2	2,93s	4,33s	6,09s	8,21s
Case 3	3,48s	5,22s	7,27s	9,83s

Distance: 100

Element count	2000000	3125000	4500000	6125000
Case 1	2,52s	3,77s	5,21s	7,07s
Case 2	3,07s	4,48s	6,25s	8,43s
Case 3	3,55s	5,65s	7,82s	10,11s





Do note that the difference in element count between the tests is not linear, thus making the plotted lines not linear as well. Test four has 1625000 more elements than Test three, which has 1375000 more elements than Test two, which has 1125000 more elements than Test one.

4.2 Renders

The following terrain and images were created using the system described in this paper together with a custom created renderer for prototyping purposes.

As a first step a graph to describe the layout of the terrain is necessary. For this a custom function that generates a random network graph was created.

The resulting graph can be seen in figure 24.

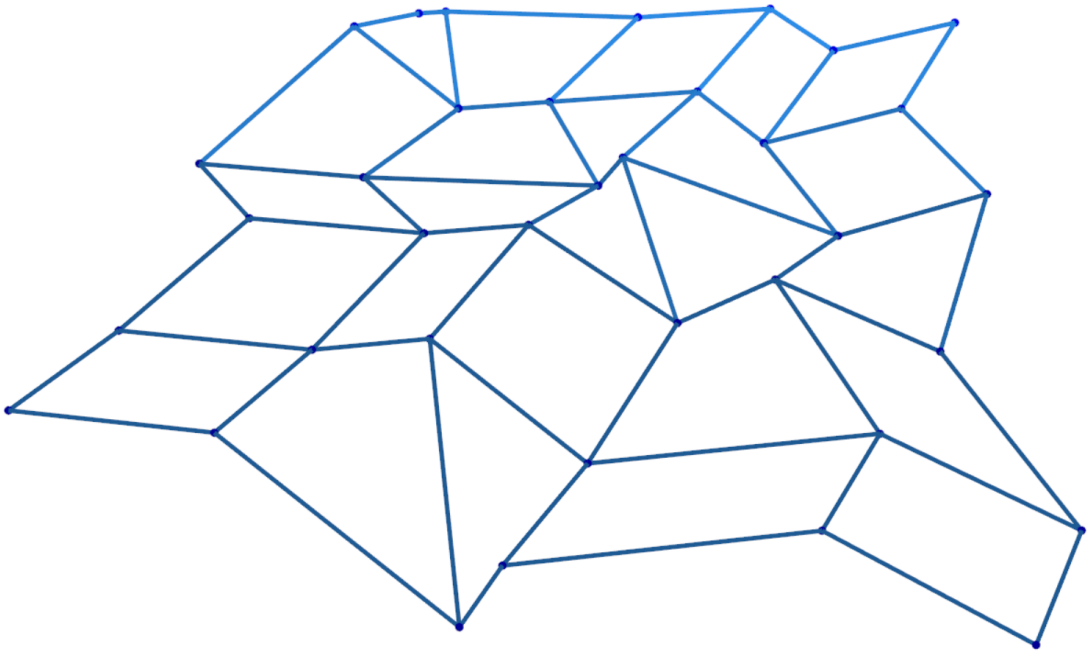


Figure 24: An empty graph before generating terrain

In this demonstration terrain will be generated using four main types of features, lake beds, low lands, fields and mountains. The algorithm used for generating the terrain is Perlin noise, so Perlin noise parameters are provided as four terrain descriptions. The descriptions can be seen in the following table. Do note that the values used here work well with this implementation of a renderer and that results may differ if used on a different renderer with different settings.

Name	Amplitude	Frequency	Persistence	Offset
Mountain	50	0.07	0.5	50
Fields	20	0.1	0.5	25
Low land	15	0.1	0.5	15
Lake bed	10	0.07	0.5	3

The fields of the graph are assigned a terrain description each. Figure 25 illustrates this by giving each field a color based on terrain assigned to it.

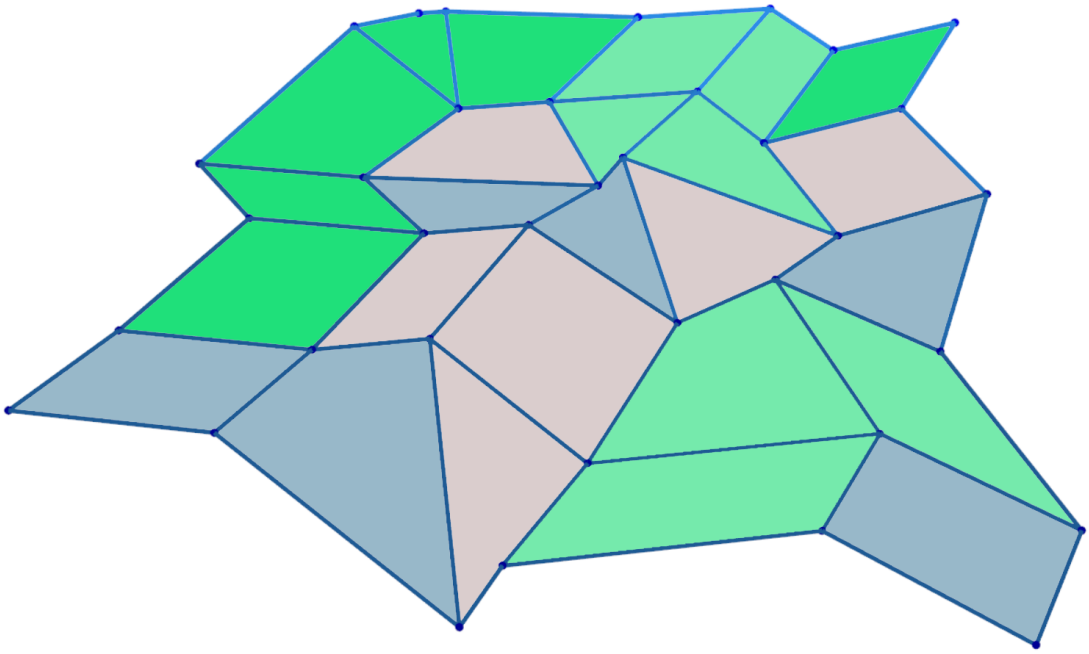


Figure 25: Illustration of a graph with its fields assigned a terrain description each. Lake beds are colored in dark blue, low land is colored in light green, fields are colored in darker green and mountains are colored in grey.

After each field has a terrain type assigned, terrain is generated for each one. Figure 26 illustrates the graph with just the raw terrain prior to the stitching. The color is added as a post-process by the rendering system in order to help illustrate the height differences. The terrain generating system in itself does not deal with visualization.

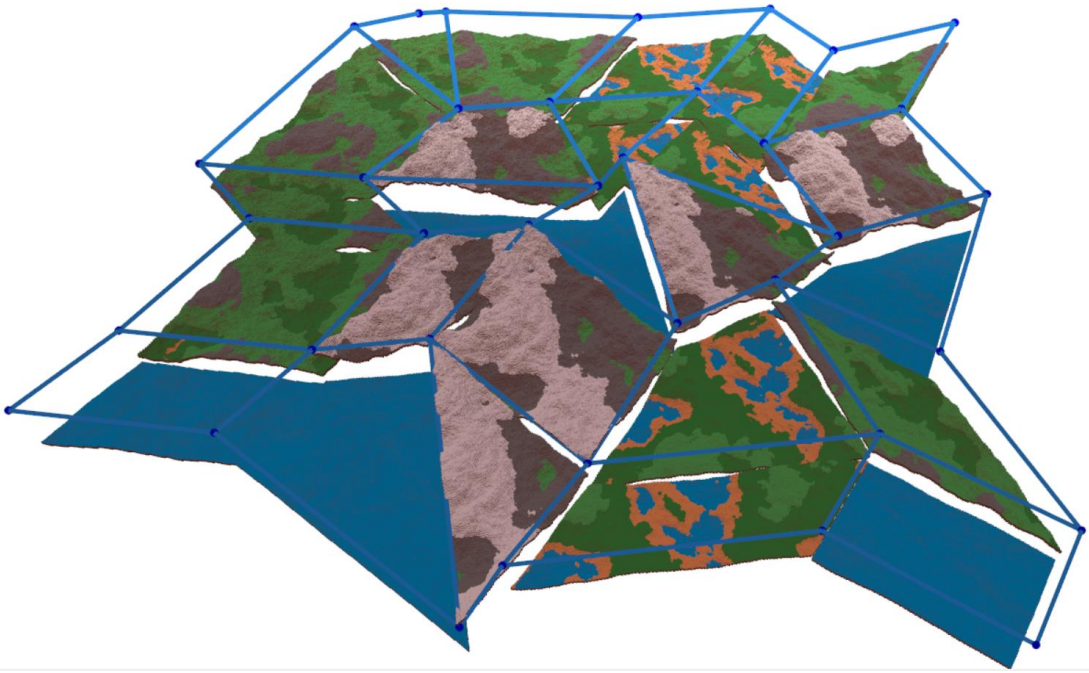


Figure 26: The graph with each field having its terrain generated.

Next the terrain pieces are stitched together. A Distance variable is chosen and assigned for use. In this case the same Distance variable is assigned to each edge of the graph. The distance variable will vary from system to system, implementation to implementation and the scale the user chooses to work with. After the stitching is performed the resulting terrain from Figure 27 is generated.

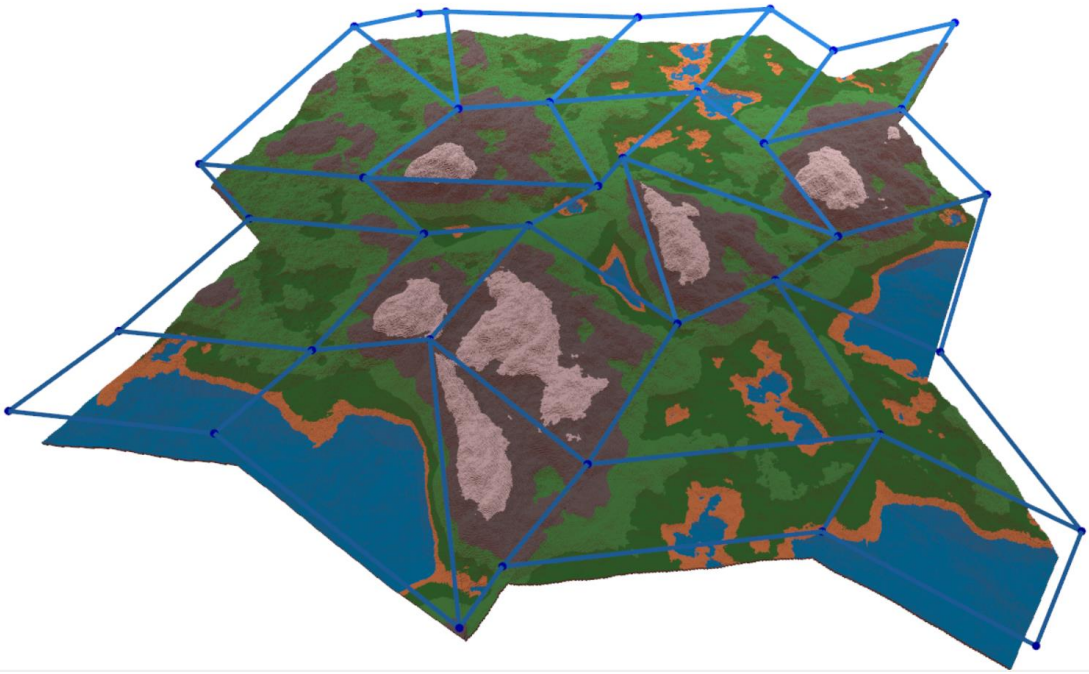


Figure 27: The final stitched graph where each terrain piece has been stitched with its neighbours.

In Figure 28 the terrain is rendered as a grid of boxes with varying height in order to show how large space each element takes up. In a real use case scenario the rendering system can render the terrain as a smooth mesh instead.

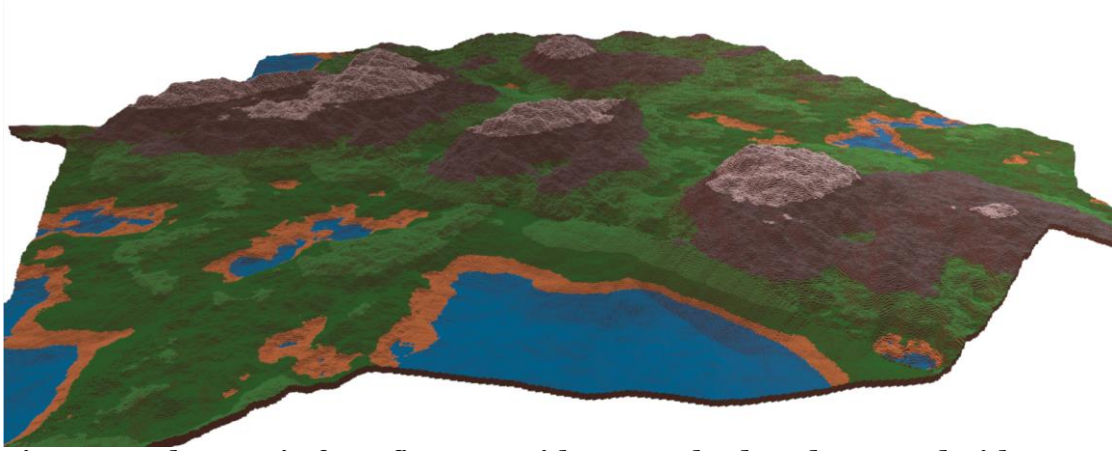


Figure 28: The terrain from figure 27 without graph edges drawn and with a different perspective

5 Discussion

5.1 Performance

In its current implementation the algorithm is running in a single thread. While there are plans to improve this aspect in the future and while the following section discusses possible implementations of concurrency, this section will show that the performance already achieved is sufficiently good. The figure found bellow, figure 25, displays a piece of terrain comprised of 3125000 elements that was stitched in 3.49 seconds on a mobile x86 CPU.

3.1 million elements is a large amount of elements for terrain used in real-time graphics. Modern game engines strive to support terrain with 2.7 million elements. (Marcin Gollent 2014) While in many heavy 3D applications the mesh of a mountain or other piece of terrain may be comprised of a similar amount of vertices, this is only a heightmap, not a mesh. Converting this heightmap to a mesh and rendering it in a real-time graphs system would be very costly per frame. Large terrain of this type are tessellated and rarely produced in such detail at once. (Mattias Widmark 2011)

In real-time graphics one would usually generate the areas far away with low fidelity, perhaps only assigning a few thousand or hundred elements for a mountain far away, and then increase the fidelity closer to the camera. Even close to the camera one would probably not generate a heightmap of 1 million elements, but would instead generate a relatively higher amount than usual and mesh it together. Add some vegetation, textures and texture bump mapping, to look like a cohesive detailed continues surface (Mattias Widmark 2011).

In non-real-time applications, such as animated movies, the meshes are very large and contain a large amount of assets being rendered. A one-time waiting fee of 3.49 seconds to stitch highly detailed and controlled terrain would be small when considering the render time of a single frame of a Pixar movie is measured in hours. (Bettinger 2012)

If one would generate multiple versions of the same graph containing the same terrain in a variety of resolutions, one could combine in them into a single mesh where the fields far from the graph center only use a few thousand elements and the fields close to the center contain tens of thousands of elements. This way far away areas would look well anyhow since they are far away, adjacent areas will look well due to their high resolution, and the terrain mesh could be even larger because faraway places do not cost much memory to produce and they would be event faster to render.

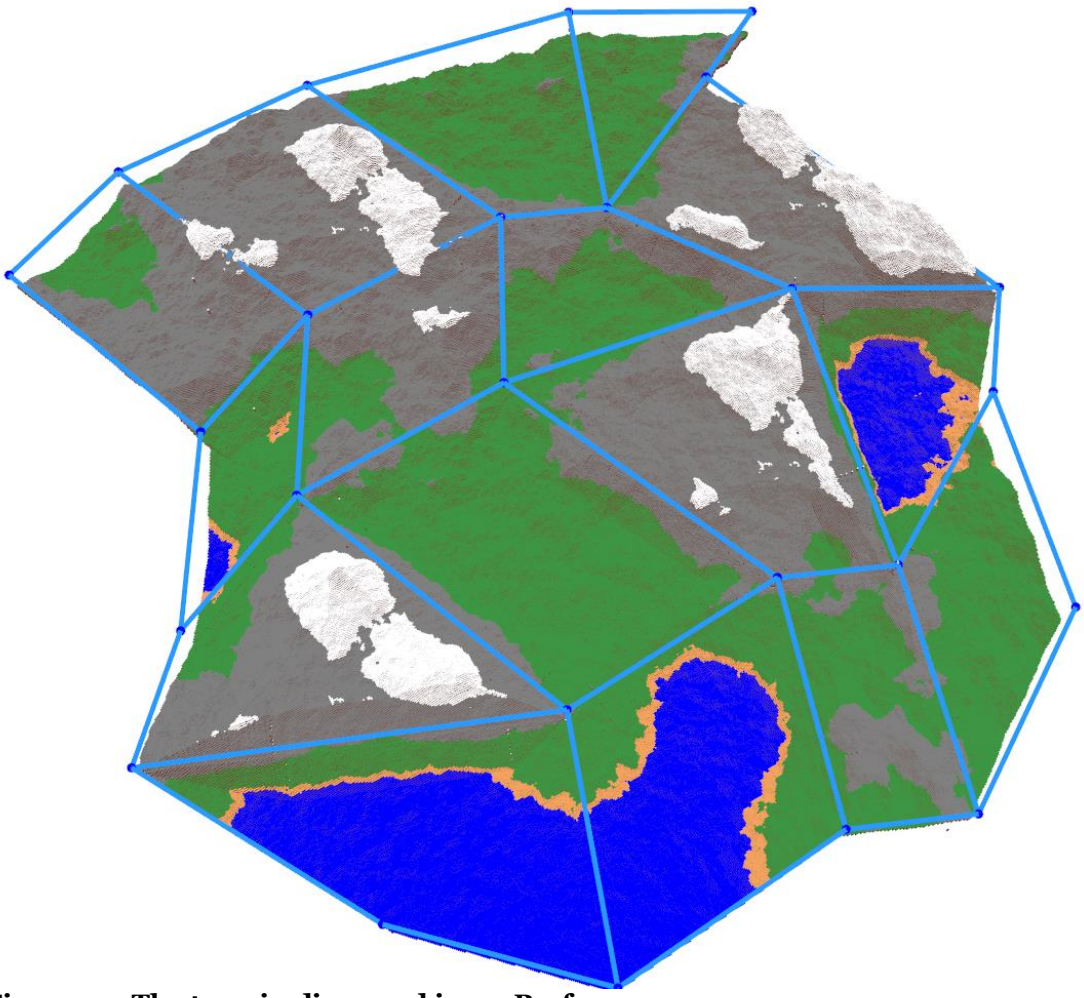


Figure 29: The terrain discussed in 5.1 Performance

5.2 Caching

A number of times during the execution of stitching, the algorithm needs to know the angle between certain edges, calculate Separator vectors together with their length and calculate the bound along each edge each stitch should be performed. Sometimes the Separator vectors of two adjacent edges will cross or overlap each other or other Distance variables bounds. In order to prevent this the algorithm makes certain calculations, as shown in each corresponding stage.

Performing these calculations when values are needed during the stitching may be simple to implement, but should be avoided in a final product. Multiple stages of the algorithm sometimes needed to access values already calculated previously, for example angles between vectors. It would be desirable to extract this code and perform all of these calculations before running the actual stitching. This is possible because it only requires one to have the graph.

If this is done only once for each edge and node before running the algorithm, and the values stored in an easily accessible and fast data structure, the

algorithm could just fetch the data when necessary and improve the execution time.

5.3 Future Improvements

5.3.1 Multithreading

The algorithm contained in this paper is currently implemented as straight code, even so it is sufficiently fast for most reasonable use cases of today. View the discussion part about performance for more details. One area of the algorithm that could see improvements though, is multithreading it.

The multithreading could be implemented in multiple ways and in several layers. One suggestion would be to first of all multi thread the creation of terrain. Create a few worker threads and assign a couple of fields to each one. Let each thread generate terrain for the fields assigned to itself. This should not be an issue to implement since the creation of terrain would only use variables private for each field, having no need for locks or other mechanisms. If the fields are large perhaps only assign one field to each thread.

Each edge and node of the graph are given a Distance value before they are stitched, indicating how far in each field should be changed. Because this variables value, or length if seen as a vector, is modified during Node Stitch, the area it covers along an edge would never overlap with the stitching from another adjacent edge. It too fits well for multithreading and could be implemented without any locks or mechanisms.

One only needs to make a slight change to the order the Node Stitch and the Normal Stitch are performed. Instead of stitching fields and imminently applying the changes, note down the changes to each element and apply them after every edge has been stitched. This way one could assign each edge to a thread for stitching or give each thread a list of edges to stitch, and could stitch them concurrently without worrying that a thread would read an already modified element and miscalculate.

When going through the elements of the fields it would be logical to do so in two nested for-loops since the data structure is a two-dimensional array of floats. And because the elements of the field being stitched inside do not affect each other during and after the stitch, one could also make this concurrent. Consider using OpenMP (OpenMP 2015) to thread the for-loops or perhaps the built in C# functions for loop threading if you are using C#.

5.4 Rounding errors

When one selects a point to stitch it is first seen as the result of two indices used to fetch data from an array. To calculate the normal from the point to the edge one views the point as a point in R2 with absolute world coordinates, instead of as a set of indices and a value. The indices are converted. After one has found

the normal one shortens it to become a normalized vector and from the point where the normal meets the edge (at POL) ones goes by one element length in each direction of the Edge. One element length into the field one stitches with regard to, and one element length into the stitched field, to get the elements that meet and measure their difference.

When one does this one ends up with two new points with absolute values in R2. In order to read the values from these points one converts the points into indices to access the element below them in the corresponding field they are found in. When converting the points one might have no issues and end up with a perfectly stitched terrain, but might other times find themselves having terrain that looks good but has a few artefacts sticking out slightly above or below the others.

This is because of the rounding errors. The array is accessed by using an index which is an integer, while the points X and Y components are probably float or double values. Based on the implementation one should consider whether to simply cast the indices to integers or whether to round them properly.

If one rounds them properly they might realize that they sometimes could get indices out of bound. If an array has 10 elements length one might sometimes get indices retrieved from converting absolute values to coordinates in the range between 9.5 and 9.99. The rounding function may then round them up to 10 instead of the intended 9 and make them go out of bound since an arrays index ranges from 0 to n - 1, where n is the number of elements. This can be easily solved by making a check to see if the index is equal to n and then shorten it by 1 to become n - 1. Though this might also be time consuming.

During the testing phase both solutions had their pros and cons. If one simply makes a cast to integer one will never have the issues of indices being out of bound, but one might have a bit crude terrain. If the element to access according to the point will be at index 4.4 it will be cast into 4, but if it is at 4.7 it will again be cast into 4 instead of 5 which may be more appropriate.

5.5 Wrong field issue

If one has a node at which multiple fields meet, and the resolution of the Fields is not very high, by which it is meant that there are few elements in each Field and their side lengths have small values, one might run into a small issue. When Field A stitches with regard to Field B, and Point is only one or two indices away from the Node. And it tries to find the element in Field B which whom it will stitch, by taking the normal from the Edge and go into Field B with the distance of one element length, that the element Field A will find will go too far through Field B and perhaps end up in Field C.

The solution to this problem could either be to check if you are one or two indices away from the node and create a special case that manually checks an element one-two indices away from the Node in Field B, or to just check which

other field the point might end up in and use that points value instead. In the tests performed during the methods development it could actually be more correct and provide a better result, but could be more time consuming and may vary from graph to graph and resolution to resolution.

5.6 Only use diamond angle

Instead of using convenient trigonometry such as Tangent and Cosines to find the angle between the Edge and the Separator, one could use the diamond angle method instead. The resulting value would be a value between zero and four as opposed to $-\pi$ to π, but it could easily be converted, as shown in section 2.5 Diamond angle, to radians.

5.7 Conclusions

The projects main goal was to introduce a new graph-based approach to generating terrain. This means that instead of only using classical terrain generating algorithm and generate terrain in all directions, graphs are used to set the layout in combination with the classical algorithms to populate them.

After first finding all of the fields in the graph, and generating terrain for each field, stitching the resulting terrain pieces together was expected to be the most difficult part to implement and the deciding factor if the method would be viable. The introduced method for stitching terrain of arbitrary shape proved to be viable, and producing correct results with reasonable execution times.

The performance and execution times measured in the test cases were surprisingly good, and the algorithm seems to be well suited for multithreading. Although multithreading is currently not implemented, it could be without too many modifications to the algorithm and mechanisms.

The drawback of the algorithm is that it may contain too many steps seen as it both requires terrain types to be described as terrain descriptions, a graph to be generated, terrain generated and then also stitched.

In conclusion the algorithm provides a good option for those who desire more control over the generated terrain and have the time and resources to implement and run it.

References

Adams David. *Automatic Generation of Dungeons for Computer Games.* The University of Sheffield, 2002.

Alain Fournier, Don Fussell and Loren Carpenter. “Computer rendering of stochastic models.” *Communications of the ACM*, June 1982: 371-384.

Bettinger, Brendan. “Pixar by the Numbers – From TOY STORY to BRAVE.” *Collider*, 2012.

Colin Smith. *On Vertex-Vertex Systems and Their Use in.* The univeristy of Calgray, 2006.

David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley. *Texturing & Modeling, A Procedural Approach.* AP Professional, 1994.

Eric Stephen Lengyel. *Voxel-based terrain for real-time virtual simulations.* University of California, 2010.

Group, Khronos. *OpenGL.* Khronos Group. 1997-2015. https://www.opengl.org/.

Henrik Zimmer. "Voronoi and Delaunay Techniques." July 30th 2005.

IEEE . *IEEE Code of Ethics.* 2015. http://www.ieee.org/about/corporate/governance/p7-8.html.

Julian. *Encoding 2D angles without trigonometry.* 5 June 2009. http://www.freesteel.co.uk/wpblog/2009/06/05/encoding-2d-angles-without-trigonometry/.

Lucasfilm. *Genesis Sequence of Star Trek 2, the Wrath of Khan.* Lucasfilm computer graphics division, 1982.

Marcin Gollent, CD Project RED. “Landscape creation and rendering in REDengine 3.” *GDC 14*, 2014.

Mattias Widmark, DICE. "Terrain in Battlefield 3: A Modern, Complete and Scalable System." *DICE 2011 GDC presentation.* DICE 2011 GDC, 2011.

Microsoft. *DirectX.* Microsoft. 2015. https://support.microsoft.com/en-us/kb/179113.

Novalogic. *Delta Force 2.* By Novalogic, 1999.

Nove: Season 36, Episode 4, Hunting the Hidden Dimension. 28 Oct 2008.

OpenMP Architecture Review. *OpenMP*. 2015. http://openmp.org/wp/.

OpenTK Group. *OpenTK*. 2015. www.opentk.com.

Rozenberg, Grzegorz. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. Leiden Univ. The Netherlands, 1997.

Yotam Livny, Zvi Kogan and Jihad El-Sana. "Seamless patches for GPU-based terrain rendering." *The Visual Computer*, March 11, 2008: 197-208.

TRITA-ICT-EX-2015:72