Natural Path Generation

Patch-based terrain optimization for use in games

Stef Velzel
Master's Thesis at *Utrecht University*

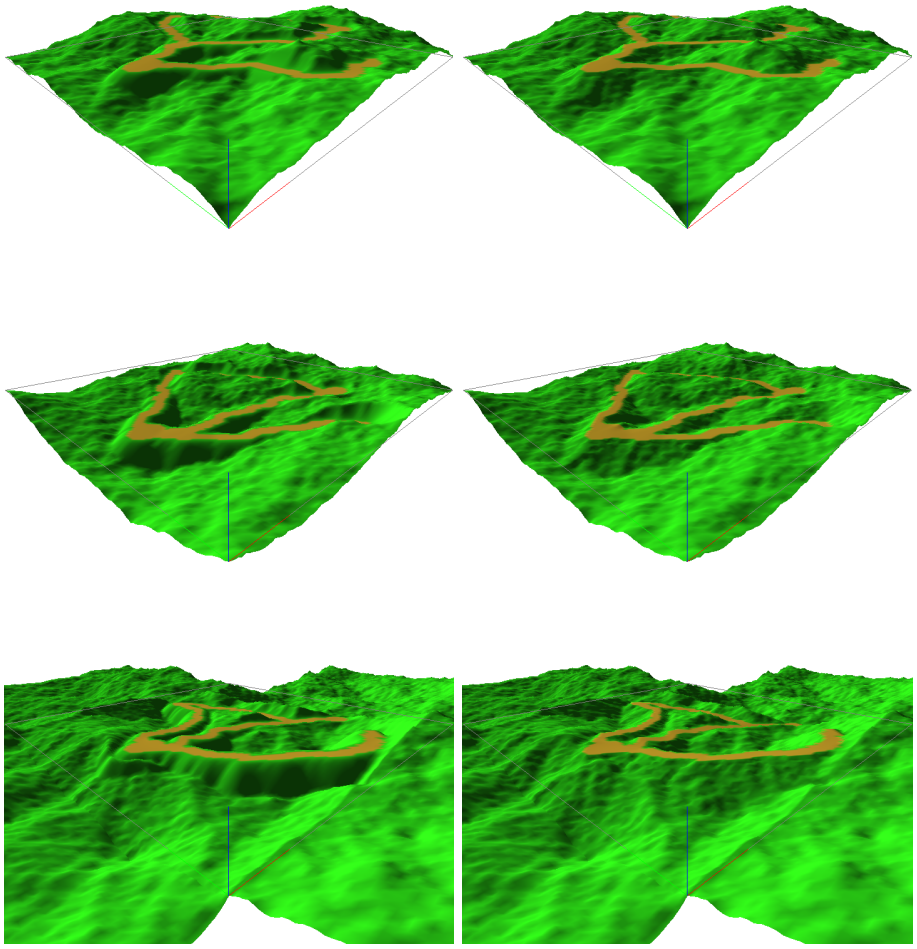

Figure 1: *Sample outputs of the proposed iterative relaxation algorithm. The left column uses directional derivative constraints for connecting terrain to the path, the right column uses roughness constraints.*

1 Introduction

Procedurally generating content is an increasingly important topic within the simulation of virtual worlds such as in computer games; it aims to automate the creation of a huge variety of content. Not only terrain, but also vegetation, buildings, characters or even rule-based missions set in procedural environments and spaces [12]. Research has even ventured into human-built structures such as roads [17] and entire cities [19, 23] that could be placed upon a terrain. This makes the development process of the designers more efficient and it creates more unexpected content for the player to keep exploring. Procedural generation can be further expanded by modifying the generated content based on the desired player experience [29] or even tailoring the generated content to the player during *actual play* based on the progress and behavior of the player [6].

Methods for procedural generation are now sophisticated enough that some games start to rely on them extensively. In these cases it is vital that the generated content is varied enough to let the player keep exploring new configurations. The world in these types of games is finite, yet there is an infinite number of variations and often there is no way to 'complete' the game. It can be observed that no manual content creation can ever achieve this, as game designers can only deliver a finite number of variations. Examples of games that rely on content that is infinitely random are Rogue [44] and Spore [46]. Some games take this to the extreme and procedurally generate infinite terrains or environments, examples are Minecraft [30] and No Man's Sky [25].

A common trope of infinite games is that they enable exploration play, which is a type of play that is recently being studied more [5]. Conversely, these games do not allow easy integration of a story line or a progression of sequential missions as seen in many adventure type games such as Magicka [31] or Tomb Raider [13]. This disadvantage is due to the lack of ability to constrain linearity in a world where the player can go anywhere. This work introduces a procedural terrain modification algorithm that is tailored to adventure type games that want to introduce an infinite terrain to emphasize exploration play. The challenge is to embed a 'natural' path the player walks along into an infinite natural terrain that forces a certain degree of linearity. This allows the introduction of (semi-)sequential missions or story lines into the game.

1.1 Proposed Algorithm

The algorithm that this work will propose works under the assumption that infinite terrain is greedily created by generating patches of terrain. All patches are strictly square so that they can be stitched together to form a seemingly infinite terrain in all four cardinal directions (north, east, south, west), see Figure 2 for an illustration. Such a patch of terrain will be represented as a finite two-dimensional heightmap, which is a widely used way to define terrains in computer games or simulations. Similar approaches of dividing up a terrain into square patches are occasionally used in other works [32, 47], however not much material exists that uses such a system to greedily generate worlds of unknown size (or theoretically infinite).

The proposed algorithm is a post-processing step that takes a square patch of terrain as input and produces an output patch of equal dimensions that can be seen as a modification of the input. In the most basic sense, the problem it tries

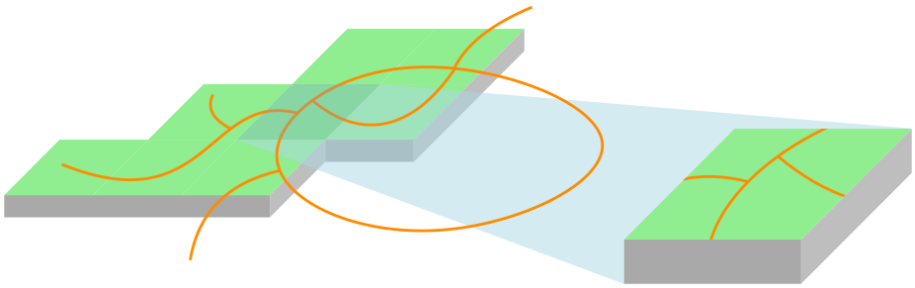

Figure 2: *Illustrated example of eight generated patches that are stitched together to form a bigger terrain (left). A graph that covers multiple patches (and even not yet generated patches) is cut so that the input graph G for a single patch (right) is well defined.*

to solve is as follows: *given a square patch of terrain, denoted by L, transform it into a new patch of terrain of equal dimensions, denoted by H, by modifying L such that a given planar input graph, denoted by G, is embedded into H as a path that can be walked upon.* This means the player, or any character, can walk on a path through output terrain H defined by input graph G. This in turn means the slope, or more precisely; the norm of the *gradient* of every point of the path in any direction must not exceed a given maximum value.

A crucial constraint needs to be considered: the *realism* of the path through the terrain. The goal is to generate a realistic looking terrain, naturally the path through it should look realistic as well. It is, for example, undesirable to simply move all vertices of L that touch the embedding of G downwards so that they can be walked upon. The path should follow the shape of the actual base terrain. More precisely, the terrain connected to the boundaries of the path must connect in a 'natural' way, preserving the realism of the terrain. We want to *maintain the 'realism' of output terrain H by imposing mathematical constraints based on graph G.* We can attempt to formulate what 'realistic' means in the design of the constraints.

Within the patch-based model, the boundaries of H need to align to its neighboring four patches seamlessly, in other words: *given set height values for the border of H, modify its interiors such that it 'realistically' connects to its borders.* A method will be discussed that flows from the proposed constraints. Input graph G can be obtained from any source; one can imagine that if we have a graph that is bigger than one patch, it could be cut up into squares, as illustrated in Figure 2, such that each patch has its own input graph. If the total size of the world is unknown, we could procedurally expand this graph. This patch system could then easily facilitate greedily generating entire new areas of the world.

Compare this problem to real life; when humans make a path, a reasonable assumption is that they want to exert the least amount of work possible, therefore we want to modify the input terrain as little as possible; *graph G should be embedded in output terrain H such that it has changed as little as possible from input terrain L.* The Earth Mover's Distance (EMD) [36] is chosen as a metric to evaluate the distance between the input and output terrains. The precise definition used in this work is given later on in the mathematical problem

description. In short: the EMD from L to H, denoted as $\mathbf{EMD}(L, H)$, is the minimum cost of turning L into H. It measures the amount of material moved and weighs it by the distance it needs to be moved, which has nice similarities with what we would expect in reality.

At this point, we can define the algorithm as an optimization problem, where the objective is to minimize the cost of transforming L into H. This problem can be solved directly to obtain a global optimum H_0. However, methods for solving such a problem optimally don't scale well with the input size. Therefore this work proposes an iterative relaxation method that approximates this optimal solution. Further work may focus on expanding this algorithm to generate a more appropriate base terrain L, procedurally generating the graph to embed in the terrain or generating the actual missions and rules [12] to augment this algorithm.

2 Related Work

There are many different approaches to procedurally generate environments. Take for example dungeon games; these take place in a labyrinth of rooms, corridors, hidden places, treasures and monsters. Dungeons are an easy way to conceptually think of a world with characters in it, you do not need an outside world with distant mountains or a sky above. The places characters can move to are naturally restricted by the layout of the dungeon. Consequently, dungeons are one of the first procedural environments that were used in games. These environments often rely on a constructive method using some space partitioning algorithm to make a subdivision of space, yielding rooms, corridors or caves [42]. Similar yet distinct processes can be performed to achieve all kinds of dungeon-like structures. A common approach is to use generative grammars, which are strings of symbols that are rewritten according to predefined rules until the string only contains terminal symbols. These symbols can represent graphs, to generate the structure of a dungeon [3], but they can also represent the geometry of the individual spaces [12]. Other methods include (but are not limited to) cellular automata, genetic algorithms and, for example, using agents that define the shape of the dungeon by their movements [45].

However when trying to generate three-dimensional terrains with hills, mountains and other natural phenomena that believably mimic the outside world, dungeon generation techniques fall short. Many algorithms that do produce such terrain exist such as the uplift model [34], the hill algorithm, the fault algorithm and various post-processing techniques [20]. Although they are useful algorithms, the most powerful and extendable approaches appear to be fractal landscape modeling, physical erosion simulation, genetic terrain programming and methods to constrain the generated terrain such as synthesis based on input features or synthesis from sample terrain patches.

2.1 Fractal terrain

Noise is a widely used technique to generate natural terrain [20, 28, 33, 37, 43]. To get visually believable slopes we generate random values on a coarser lattice than the heightmap and interpolate between them. This way the terrain height gradually increases or decreases. However, this approach produces sharp peaks

and valleys connected with straight slopes. Instead of generating the height values directly, we can also generate random slope (also called gradient) values and infer height values from that. Since we are now generating gradients, i.e. rate of change, we have an extra level of smoothness; the gradients are interpolated, so the gradient of the terrain will change gradually. This is also called perlin noise [20], which is a widely used primitive in many facets of procedurally generating content.

The method above still has a major visual drawback, namely that it only oscillates at one frequency. In nature, landscapes have the same kind of variations at multiple scales. This self-similarity is the basis of fractals, and generated terrain with this property is called fractal terrain [43]. A conceptually simple way to introduce such self-similarity is by generating multiple heightmaps, each with a different frequency, and adding these different frequencies together to form a single terrain. However, more efficient and straightforwards methods exist; perhaps the most well known algorithm to produce fractal terrain is the midpoint displacement algorithm [33]. Instead of calculating each height value for each frequency, the midpoint displacement methods works by recursively calculating the missing values halfway between already known values and offsetting the new values with a random value whose range is determined by the depth of the recursion. A few variations of the midpoint displacement algorithm exist such as the triangle-edge subdivision or the diamond-square subdivision. These variations differ in what lattice points are selected for each recursive step, resulting in slight differences in the resulting overall shape of the the terrain.

One of the major advantages of the midpoint displacement method is that in each recursive step, adjustments can be made to the random offset to better reflect the terrain type. For example, the height fluctuations of lattice points can be taken into account so that the range of the random offset is smaller in flat landscapes, but greater for more rugged landscapes [28]. This technique, and all other fractal terrain techniques mentioned above, produce realistic looking terrain (some artifacts may occur). However, they all lack any form of control over the location or size of terrain features.

2.2 Erosion simulation

Erosion simulation is an approach to synthesizing terrains based on the erosion from fluid streams running through it. It is often used as a post-processing step to further refine the terrain by adding ridges and valleys to enhance realism. The two most widely used methods are based on thermal erosion [26] and hydraulic erosion [9]. Thermal erosion simulates material breaking loose and being deposited at the bottom of a slope. Hydraulic erosion is material being transported by flowing streams of water. One of the greatest weaknesses of such algorithms is the computation cost of the transportation of water. This is a complex and expensive process to perform. some effort has been made to integrate more complex fluid simulation into the erosion process [10]. On top of that, a lot of research has gone into optimization of erosion algorithms to run on the GPU [24, 27, 39].

All the above algorithms still suffer a lack of control over the shape of the terrain. Some methods are suggested that combines the actual terrain generation and the erosion simulation [11, 18]. Instead of eroding an already generated terrain, these methods allow the designer to paint a simple sketch (or similar)

that will define either the rough shape of the mountain ranges or the shape of the river system. However, there is still no control over the finer details of the terrain. Another method introduced by Olsen [28] attempts to modify erosion algorithms such that the terrain satisfies constraints with use for computer games in mind. Olsen observes that for computer games, it is desirable to have a low average slope of the generated heightmap such that buildings can be built or characters can walk upon areas of the terrain. However, some height variation is desired, so a second constraint is introduced that says the slopes should have a high standard deviation. By careful modification of the erosion algorithm Olsen succeeds in quickly generating terrains that satisfy these constraints.

The above erosion algorithms sketch ways to exert some control over the location or shape of terrain features in fractal terrains. At large scales we can guide the general shape of the terrain and at smaller scales we can make sure it satisfies certain constraints concerning the shape of the terrain. Unfortunately, large-scale control does not translate to small scale control and small scale constraints do not allow for precise control over the location or size of terrain features, which is exactly what we aim to do in this work.

2.3 Search based terrain

Another approach to generating terrain is to use genetic programming [14]. The method was inspired by the lack of control over most terrain generation algorithms. The idea behind the approach is to use interactive evolution with genetic programming to generate terrain programs. A terrain program is simply a program that can be executed to generate a new, novel terrain. Such a program is designed with specific terrain features in mind such as cliffs, corals and mountains. These terrain programs are the individuals on which genetic programming is applied, a human designer manually selects individuals for each generation. Some effort has been made to improve the generated programs such that other parameters can be imposed upon the terrain such as accessibility [15, 16]. However, these methods still have major drawbacks, namely that the search for the right terrain program is rather difficult and time-consuming. Furthermore, continuous human interactivity is needed to ensure the resulting terrains are realistic, or at least believable.

Simpler search based methods rely on a local search algorithm using heuristics to match an exact shape or exact constraints on the terrain [4, 38]. It is easy to see how these are more practical to use within computer games, they do not require an interactive process of evolving a terrain program and they give more control over the output shape. Especially the technique proposed by Stachniak and Stuerzlinger [38] provides a useful yet simple local search algorithm. Similar to erosion simulation the algorithm is essentially a post-processing step applied to already generated terrains. It modifies terrain using simple gaussian kernels and provides a very powerful model of constraining the terrain. It allows to match an exact shape, stitch terrain patches together, carve paths through the terrain and more. The major drawback of such a technique is that it is only a post-processing step, it offers little control over large-scale terrain features. Furthermore, the search space is rather large, it is very time-consuming and nowhere near real-time. However, it does output promising results.

2.4 Controlling terrain

The techniques discussed in previous sections do not allow for precise control over the location, size and shape of the generated terrain features all at once. Most techniques do allow control over one of these aspects, but are lacking in another. A lot of research has gone into giving precise control and the ability to constrain terrain as desired. Several authors have investigated sketch-based systems [2, 40, 41], these methods take a rough sketch of desired features from the designer and generate terrain based on it. Another sketch-based method utilizes real landscape data [47]. This example-driven method takes a designer-sketched feature map and combines it with patches of real heightmap data to get the desired features in the generated terrain. A different method instead takes exact vector based feature curves as input [21], which has been extended to volumetric terrain generation [7].

Another school of thought to introduce more control is to constrain points in the heightmap to be generated. One such method uses local search and is already highlighted above [38]. However, it is a very time consuming algorithm. For use in video games it is desirable to decrease the search space such that terrain patches can be generated while the player is moving towards them. Another interesting method uses simple incremental construction to get a mountain peak at an exact location [22]. Unfortunately, no research has been done to extend this to allow for entire mountain ranges. Another approach is to constrain existing generation models such as constraining the midpoint displacement algorithm [8]. This technique first fixes lattice points and then runs bottom-up (i.e. recursively, but bottom-up instead of top-down) to further expand the starting constraints of the algorithm. This allows the fixed points to naturally influence the landscape around it. Afterwards, it runs the midpoint displacement algorithm as usual. Such an approach enables to fix the exact height of given lattice points in the landscape. Unfortunately it offers no intuitive control over larger areas yet; to affect an entire group of lattice points, we have to specify each location precisely, so it would need augmentation from some other algorithm to be useful.

Finally we address seamlessly stitching different patches of terrain together. This appears to be a relatively simple problem; many of the methods discussed above can achieve this. For example, the algorithms taking vector based feature curves as input [7, 21] could define a feature curve at the boundary of a patch. Alternatively, the bottom-up midpoint displacement algorithm [8] could simply constrain the vertices at the boundary. The method by Stachniak and Stuerzlinger [38] could easily perform a local search to modify two boundaries to match. The algorithm in [32] proposes a simple cubic spline interpolation between the boundaries of two patches that slightly overlap and finally the example-driven method in [47] solves a poisson equation inspired by methods to perform pixel matching across seams in a color image.

3 Problem Description

As stated before, the algorithm is an iterative relaxation method that can be seen as a post-processing step on an already generated base terrain L, where $L = \{l_1, l_2, ..., l_N\}$ is a patch of terrain with $N = n \times n$ points arranged in a

square grid with equal sides of length n. Similarly, let $H = \{h_1, h_2, ..., h_N\}$ be the *yet unknown* output patch of equivalent arrangement and dimensions as L. The algorithm takes input patch L and transforms it into output patch H such that a given planar input graph, denoted by G, is embedded into H as a path that can be walked upon and $\mathbf{EMD}(L, H)$ is minimal. The 'walkability' of the path, among other features, is implemented as a constraint of the optimization problem. Another such feature is that terrain next to this path should connect seamlessly to it in a realistic manner.

To generate base terrain L, this work uses the midpoint displacement algorithm. Note that we can immediately solve the problem of stitching patches of terrain together by applying the method that constrains the midpoint displacement method as discussed above [8]. However, we will discuss an alternative method integrated within the proposed algorithm such that we are not confined to using the midpoint displacement method. This way we can use any source as our base terrain, even real-life data.

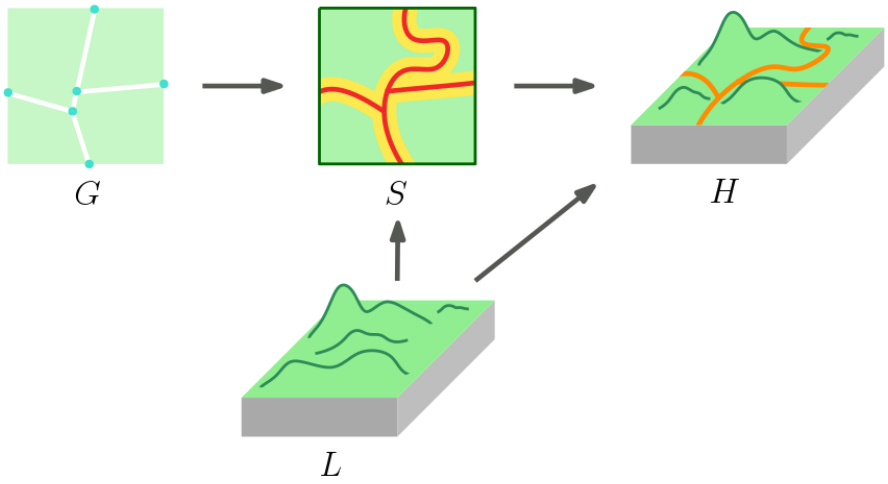

Figure 3: *Overview of the pipeline of this work. Graph G and terrain L together form the input. G is pre-processed into subdivision S where every subset $S_\nu \subseteq S$ imposes a specific constraint on output terrain H. S and L together form the input for the iterative relaxation.*

Besides the high-level realism constraint formulated as the minimization objective, we impose low-level constraints on output terrain H such that the points of H that belong to the path can be walked upon and the points that do not belong to the path still behave in a realistic way. Essentially, input graph G is pre-processed based on L to obtain a subdivision of L that tells the algorithm what parts need to change and how. This subdivision is denoted as S, where every subset $S_\nu \subseteq S$ imposes one specific constraint on every point in S_ν. Given every subset S_ν is a set of points, they can be thought of as subsets of output H as well, each containing the points their constraint applies to.

To compute S, we fix every node of G on a desired location and for every edge

we perform a path-finding algorithm on terrain L. We can interpret every point in heightmap L as a path-finding node, where each pair of directly neighboring points are connected with an edge, including diagonals. In this work, the A^* algorithm is used to find a path through heightmap L to connect all the nodes of G. After this is done, every node and found path is given a width such that all subsets S_ν can be properly defined with the correct points in them. For the A^* algorithm to work, it is required to define the cost of traveling between all the points in L. For simplicity, only directly neighboring points, including diagonals, are considered to be connected with a path-finding edge. The idea is that we want to penalize terrain with a high slope such that the path would naturally flow in-between steep terrain features like humans would.

This pre-processing of G is separate from the proposed algorithm and S is considered input to the algorithm. This allows us to swap out this pre-processing and path-finding with any process to give shape to the path and desired terrain features as we see fit. Once subdivision S is obtained, the proposed algorithm will try to compute H, solving for all constraints such that $\mathbf{EMD}(L, H)$ is minimal. See Figure 3 for a schematic overview of the pipeline.

3.1 Optimization Problem

The mathematical problem is formulated as a modification of the Earth Mover's Distance. Instead of just solving for the distance between two known terrains, one terrain is known, and one terrain will be the *unknown* output to solve. To achieve this, a term is added to the total cost to account for creation or destruction of material and the low-level constraints are added to give rise to desired features in the terrain, such as the 'walkability' of the path.

Let the ground distance matrix be defined as $D = [d_{ij}]$, where d_{ij} denotes the ground distance between points $l_i \in L$ and $h_j \in H$. The ground distance can be any metric, here the L_1 distance was chosen. Furthermore, let the height of any point l_i or h_j be denoted by itself. The additional low-level constraints to give rise to desired output features are applied to predefined regions of the terrain. Let $S_\nu \subseteq H$ be a subset of points in H that constraint ν is applied to, where $\nu(j)$ is the constraint value of constraint ν at point h_j. The discussed constraints and their associated regions are gradient constraint g in region $S_g \subseteq H$, directional derivative constraint g° in region $S_{g^\circ} \subseteq H$, roughness constraint r in region $S_r \subseteq H$ and position constraint p in region $S_p \subseteq H$.

We want to find output terrain H and flow $F = [f_{ij}]$, where f_{ij} is the flow from l_i to h_j, that minimizes overall cost of generating H. Let σ denote the cost of creating or destroying material:

$$\min \left(\sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} d_{ij} + \sigma \left| \sum_{i=1}^{N} l_i - \sum_{j=1}^{N} h_j \right| \right)$$

subjected to the following constraints:

$$f_{ij} \geq 0 \qquad 1 \leq i \leq N, 1 \leq j \leq N \tag{1}$$

$$\sum_{j=1}^{N} f_{ij} \leq l_i \qquad 1 \leq i \leq N \tag{2}$$

$$h_j \geq \sum_{i=1}^{N} f_{ij} \qquad 1 \leq j \leq N \tag{3}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} = \min\left(\sum_{i=1}^{N} l_i \ , \ \sum_{j=1}^{N} h_j\right) \tag{4}$$

and the following additional constraints:

$$|\nabla H(h_j)| \leq g(j) \qquad \forall h_j \in S_g \tag{5}$$

$$|\nabla H(h_j) \cdot D(j)| \leq g^{\circ}(j) \qquad \forall h_j \in S_{g^{\circ}} \tag{6}$$

$$R(h_j) = r(j) \qquad \forall h_j \in S_r \tag{7}$$

$$h_j = p(j) \qquad \forall h_j \in S_p \tag{8}$$

where constraint value $g(j)$ is the maximum slope (i.e. magnitude of the gradient) the terrain may take at point h_j in *any* direction and ∇H is the 2-d vector field with partial derivatives of H at all points (i.e. the gradient of the terrain). Constraint value $g^{\circ}(j)$ is the maximum slope in the direction denoted by 2-d unit vector $D(j)$ that the terrain may take at point h_j. Constraint value $r(j)$ is the roughness the terrain must have at point h_j, where $R(h_j)$ denotes the actual roughness at point h_j. Finally, constraint value $p(j)$ is the height point h_j must take. All regions S_ν and constraint values are predefined, meaning they are constants in the optimization problem.

As can be seen, there are four types of low-level constraints with which this problem models all desired features: *gradient* (5), *directional derivative* (6), *roughness* (7) and *position* (8) constraints. The gradient constraint is used to model the actual path defined by G, we need characters in a game to be able to walk upon this area, hence the slope may not exceed the maximum given by $g(j)$. The directional derivative constraint is the first option to make sure the terrain next to this path connects seamlessly to it. It is the same as the gradient constraint, except that it is only necessary to be satisfied in a single direction given by $D(j)$. In this work, this direction is always chosen to be perpendicular to the path, the further a point is from the path, the greater maximum slope $g^{\circ}(j)$ will be. Both the gradient and directional derivative constraints rely on the gradient ∇H, given our terrains are represented as finite heightmaps, we need an approximation for this derivative. The first-order finite difference was chosen for this:

$$\nabla H(h_j) = \left[\frac{h_{x(j)} - h_j}{\delta}, \frac{h_{y(j)} - h_j}{\delta}\right]$$

where δ is the distance between any point and its nearest neighbors and $x(j)$ and $y(j)$ are the closest next neighbor of index j in the x and y direction respectively. Both constraints demand the above holds in four directions, once in each 'quadrant' around point h_j, however for readability this detail is omitted

in the problem formulation. The roughness constraint is firstly designed to be an alternative to the directional derivative constraint. It proposes to maintain the *roughness* of every point, this concept is defined by Riley, DeGloria and Elliot [35] and slightly adjusted for our purposes as follows:

$$R(h_j) = \sqrt{\sum_{k \in \tau(j)} \left(\frac{h_k - h_j}{\delta} \right)^2}$$

where $\tau(j)$ is the set containing the eight direct neighbors of index j. The division by δ was added so it is invariant to the resolution of the terrain. This work tries to maintain roughness, i.e. $r(j) = R(l_j)$, in other words; the roughness of any point $h_j \in S_r$ should be equivalent to the roughness at point l_j. This makes sure no new discontinuities will be introduced in the terrain. Finally we have the position constraint, this simply forces a point to be at a given height, in combination with the roughness constraint it is used to stitch the boundaries of a terrain patch to those of its neighboring patches.

As for the EMD constraints, constraint (1) dictates that material is moved from L to H and not vice versa. Constraint (2) limits the amount of material that any point l_i can send and constraint (3) dictates the least amount of material that any point h_j must receive. Finally, constraint (4) forces to move the maximum amount of material possible, this is called the *total* flow in the original Earth Mover's Distance. Note that if we fix H to be known, the value of $\mathbf{EMD}(L, H)$ can be calculated as the minimization objective of this problem without the additional constraints (5, 6, 7, 8). In the original metric, the resulting value was divided by the total flow in order to avoid favoring smaller signatures in the case of partial matching. However L and H are of the same dimensions by definition, rendering this problem irrelevant. When H is a decision variable, the minimization objective will still yield the value of $\mathbf{EMD}(L, H)$.

4 Method

The optimization problem can be solved directly, yielding a global optimum H_0. Unfortunately this is a very time-consuming process, the original Earth Mover's Distance is a linear problem, however the added low-level constraints (5) and (7) are quadratic, making the problem even more time-consuming. On top of that constraint (7) is non-convex, which often makes the problem unsolvable. Finally it can be proven there are constraint values for which the problem is strictly unsolvable because it is over-constrained.

For all of the above reasons this work proposes an iterative relaxation method that approximates the solution. This algorithm needs certain threshold values so we can accept a 'solution' when the constraints are satisfied within a certain threshold value from their goals. This is necessary due to floating point errors, however it might also give us a reasonable output terrain even when the problem is technically infeasible. There are methods with which we could solve for a local solution with thresholds, for example a local search based method that solves for arbitrary mathematical constraints [38], however these methods still have a running time impractical for use in games.

The proposed algorithm is loosely based on an iterative method that directly generates new terrains from vector based constraints [21]. This method essen-

tially imposes constraints on a per-point basis, in the same way our low-level constraints are formulated. The idea is to 'relax' each point individually, resulting in a new terrain that is closer to a solution. We keep iterating the same relaxations until all constraints are within an acceptable threshold from a local solution. The difference is that our method allows the relaxation of a single point to affect its direct neighbors, making the process more complex, but also more versatile. Observe that the running time of such a method is inherently faster than local search based methods; we do not need to compute multiple candidates during each iteration and choose one of them. Instead we greedily compute a next candidate without computing any other candidate.

To get a sense for how well the relaxation algorithm approximates a global optimum, we do actually try to solve for H_0. This was implemented using the Gurobi Optimizer [1]. Unfortunately this can only be run for constraints values that have a feasible optimum.

4.1 Implementation

In all implementations, height values of all points $l_i \in L$ are roughly speaking in the interval $[0, 1]$, this is used as reference for choosing other constants. For all experiments, some constant values were chosen as will be discussed below. However the value of $g(j)$ for all points $h_j \in S_g$ (for all points, the same value is chosen) and the threshold values for relaxation are varied to measure the impact on performance.

Remember that input graph G is first pre-processed into subdivision S. To do this, every node of G is fixed on a desired location and for every edge we find a path between its endpoints by using the A^* algorithm on heightmap L. During the path-finding process, when we are at point l_i, we consider its eight direct neighbors as next points to visit. The cost of traveling each of those edges to the next point has to be defined for the A^* algorithm:

$$\text{cost}(l_i, l_k) = d_{ik} + d_{ik} * \alpha \left| \frac{l_k - l_i}{d_{ik}} \right|^{\beta}$$

where l_k is the next considered neighbor, d_{ik} is the L_1 distance from l_i to l_k. The idea is that we want to avoid terrain with a high slope, such that the path gently avoids mountain peaks and other terrain features that we would naturally walk around. Therefore we penalize edges with a high slope; α is the linear cost and β is the exponential cost. In this work the values $\alpha = 10000$ and $\beta = 1.8$ were empirically chosen for all experiments.

Furthermore, the directional derivative constraint is used to make sure the terrain next to the path connects seamlessly to it. It has constraint value $g^\circ(j)$ for every point $h_j \in S_{g^\circ}$ that expresses the maximum allowed slope, this value grows larger the further h_j is from the actual path and is written as follows:

$$g^\circ(j) = g(k) + \gamma * \sqrt{\frac{d_{jk}}{b}}$$

where h_k is the closest point to h_j that is on the path and γ is a constant value denoting the maximum increase in allowed slope. For this formula to work, distance d_{jk} from h_j to h_k is divided by the constant border width b such that the resulting value is in the interval $[0, 1]$. Any point with a distance

from the path greater than b is not in S_{g°. In this work, the value $\gamma = 0.05$ was empirically chosen for all experiments. The value of b can be altered until desired results are achieved.

Finally, in the objective of our optimization problem, a term was added to account for creation or destruction of material, σ denotes the cost of doing so. In the relaxation algorithm, material is only created or destroyed when a position constraint is applied, for example to seamlessly stitch the borders of a patch to that of its neighbors. All other constraints are solved such that they do not create or destroy any material. However to implement a solver to get a global optimum, we need a value for σ. Conceptually, we would rather move material to neighboring areas instead of completely removing it from the terrain. Similarly, creating more material is only desired when necessary. For this reason, the value of σ must be greater than any value d_{ij} in ground distance matrix D. In this work, the value $\sigma = 2n\delta$ was chosen, where n is the length of a single side of the terrain grid and δ is the distance between any point and its nearest neighbors.

4.1.1 Iterative Relaxation

The idea behind the iterative relaxation method is to consider all points $h_j \in H$ individually and relax them, i.e. adjust its height or the height of its neighbors until it locally satisfied the applied constraints. H will be initialized as a copy of L. One relaxation of all points in H is considered a single iteration of the algorithm. The algorithm halts when all constraints are within an acceptable threshold from their goals, or when the maximum number of iterations I_m is reached. The slope and roughness constraints need such a threshold, let these be denoted as T_s and T_r respectively. The algorithms below demonstrate how these thresholds are applied. The position constraint is always handled last in an iteration, therefore it will be satisfied exactly and no threshold value is necessary. Below we will discuss how one relaxation on a single point is performed for each of the four low-level constraints.

The gradient (5) and directional derivative (6) constraints both rely on a 1-d slope variant. In 1-d, a slope constraint can be defined that says the slope between the point h_j and its neighbor h_k may not exceed a maximum slope. The general idea is to minimize $\mathbf{EMD}(L, H)$, where moving material over the least amount of distance is preferred, therefore we only move material between directly neighboring points. To solve the 1-d slope constraint, we move material from the highest point to the lowest point, this can be expressed in an algorithm as follows:

```
1: procedure RelaxSlope1D(h_j, h_k, maxSlope)
2:     slope = (h_k − h_j)/δ
3:     if |slope| > maxSlope then
4:         move = (|slope| − maxSlope) ∗ δ/2
5:         if slope < 0 then
6:             move = −move                    ▷ Negate when h_j > h_k.
7:         h_j += move
8:         h_k −= move
```

For the 2-d gradient constraint (5), we are not concerned with the 1-d slope of a single edge between two points. Instead, we are concerned with the magnitude of the gradient at each point. The gradient is defined to always be pointing in the direction of steepest slope, where its magnitude is equivalent to the 1-d slope. Therefore, we constrain the gradient. To compute the magnitude, we not only need the neighbor of h_j in the x-direction $h_{x(j)}$, but also its neighbor in the y-direction $h_{y(j)}$. In algorithm form, this constraint is solved as follows:

---

procedure RELAXGRADIENT(h_j)
  $xSlope = (h_{x(j)} - h_j)/\delta$
  $ySlope = (h_{y(j)} - h_j)/\delta$
  $gradient = \sqrt{xSlope^2 + ySlope^2}$
  if $gradient > g(j) + T_s$ **then** ▷ Note slope threshold T_s is added.
    $f = g(j)/gradient$
    RELAXSLOPE1D(h_j, $h_{x(j)}$, $|xSlope| * f$)
    RELAXSLOPE1D(h_j, $h_{y(j)}$, $|ySlope| * f$)

At the boundaries of the terrain grid different rules apply because of missing neighbors, plus the constraint is actually applied four times, once in each direction or 'quadrant' around point h_j, however for readability these details are omitted.

Solving the directional derivative constraint (6) is almost identical to solving the gradient constraint. The only difference is that we only care for the slope in a given direction, instead of the direction of the steepest slope. It is expressed in an algorithm as follows, where $D_x(j)$ and $D_y(j)$ are the x-component and y-component of $D(j)$ respectively:

---

procedure RELAXDIRECTIONALDERIVATIVE(h_j)
  $xSlope = (h_{x(j)} - h_j)/\delta$
  $ySlope = (h_{y(j)} - h_j)/\delta$
  $derivative = |xSlope * D_x(j) + ySlope * D_y(j)|$
  if $derivative > g^\circ(j) + T_s$ **then** ▷ Note slope threshold T_s is added.
    $f = g^\circ(j)/derivative$
    RELAXSLOPE1D(h_j, $h_{x(j)}$, $|xSlope| * f$)
    RELAXSLOPE1D(h_j, $h_{y(j)}$, $|ySlope| * f$)

The roughness constraint (7) is solved by multiplying each term in $R(h_j)$ with a factor that will make the roughness equivalent to the value we want it to be. This is done by altering each of the eight neighbors of h_j, such that their height differences with h_j get amplified or suppressed as required. Note that this may create or destroy material, therefore we move all nine points up or down with the same amount such that there is no net difference in total amount of material. Again, this tries to minimize $\mathbf{EMD}(L, H)$ by *only* moving material between neighboring points. It is expressed in algorithmic form as follows, where the set containing the eight neighbors of index j is $\tau(j)$:

```
1: procedure RelaxRoughness(h_j)
2:     if |R(h_j) − r(j)| > T_r then        ▷ Note roughness threshold T_r is used.
3:         f = r(j)/R(h_j)
4:         moved = 0
5:         for all n ∈ τ(j) do
6:             move = (h_n − h_j) * f − (h_n − h_j)
7:             h_n += move
8:             moved += move
9:         for all n ∈ {j} ∪ τ(j) do
10:            h_n −= moved/9
```

Lastly we have the position constraint (8), which is always solved last, in a separate loop, such that all points in H always satisfy position constraints exactly. This is such that when we try to stitch boundaries of a patch, there is no gap in the terrain. The interior of the terrain is handled by roughness constraints, such that it naturally connects with the boundaries. Also note that this is the only constraint that creates or destroys material and the only one that does not touch any neighboring points of h_j. The algorithmic form is rather trivial:

```
1: procedure RelaxPosition(h_j)
2:     h_j = p(j)
```

4.1.2 Optimization Program

To implement a direct solver for H_0 in the Gurobi Optimizer [1], the mathematical problem formulation had to be adjusted slightly. The absolute value in the objective make it a non-linear problem that could not be implemented in Gurobi, therefore it had to be implemented using linear constraints, as did the min() operator in constraint (4). To circumvent the absolute value, two new decision variables are introduced, the so called *slack variables* s^+ and s^-. The minimization objective is changed to the following:

$$\min \left(\sigma(s^+ + s^-) + \sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} d_{ij} \right)$$

and the following two constraints are added:

$$\sum_{j=1}^{N} h_j - \sum_{i=1}^{N} l_i \leq s^+$$

$$\sum_{i=1}^{N} l_i - \sum_{j=1}^{N} h_j \leq s^-$$

This makes sure that both the positive and negative values that the difference in total material can yield will be minimized towards zero.

The min() operator in constraint (4) gives a bigger problem. The operator cannot be written differently with the same results, however the closest possibility is to split it up in two separate constraints as follows:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} = \min\left(\sum_{i=1}^{N} l_i \ , \ \sum_{j=1}^{N} h_j\right)$$

$$\Downarrow$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} \leq \sum_{i=1}^{N} l_i$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} \leq \sum_{j=1}^{N} h_j$$

Unfortunately, this allows for all values f_{ij} in flow matrix F to stay all zeros. In order to force all flow values to be used, constraint (3) is changed to a strict equality:

$$h_j \geq \sum_{i=1}^{N} f_{ij} \qquad 1 \leq j \leq N$$

$$\Downarrow$$

$$h_j = \sum_{i=1}^{N} f_{ij} \qquad 1 \leq j \leq N$$

In order words, the sum of all flow towards point h_j cannot be lower than the amount of material at h_j, forcing the solution to have non-zero flow values. This has the side effect that it also disallows h_j to grow any larger than the flow towards it. In other words, this formulation does not allow for the creation of new material. Similarly, if instead we changed constraint (2) to an equality, it would not allow for the destruction of material. Unfortunately no way was found to circumvent this problem, this means that inputs that demand creation or destruction cannot be directly solved unless you know if the total material of H_0 will be larger or smaller than L beforehand. In practice this means no input with position constraints can be solved directly.

All the other constraints could all be implemented, including the additional low-level constraints, even though constraints (5) and (7) are quadratic. However as mentioned earlier, constraint (7) is actually non-convex, any input given to the solver that uses this constraint could not be solved. In conclusion this means an optimal solution H_0 can only be calculated for inputs that exclusively use gradient or directional derivative constraints.

4.2 Evaluation

There are four configurations of types of constraints that we will discuss. The path itself will always be constrained by the gradient constraint. The directional derivative and roughness constraints are two mutually exclusive options to deal with seamlessly connecting the path to its surrounding terrain. Both of these

options have the ability to stitch the terrain's borders to those of its neighboring patches using a combination of position and roughness constraints. Besides these four distinct configurations, for any input terrain L, we run the algorithm at different resolutions (i.e. different values for input terrain size $N = n \times n$) to see how our measures are affected by input size. The same input graph G is used for all experiments.

All measurements will be run on all configurations and resolutions they can be run on. For any configuration-resolution pair, the algorithm is run for N_s samples (one sample being a randomly chosen unique input terrain L). The value for N_s is chosen based on the resolution that takes the most time to compute, the computation time for different configurations at the same resolution were comparable enough. We look at three separate measurements for all configurations and resolutions:

- Firstly, we look at the running time, expressed in the number of iterations it took to find a solution. Given we run the experiments at different resolutions of the terrain, we can estimate the asymptotic running time of the algorithm expressed in input size N.
- Secondly, we try to solve for an optimal solution. This program outputs the value of $\mathbf{EMD}(L, H_0)$. This will be compared to the value of $\mathbf{EMD}(L, H)$, where H is the output of the iterative relaxation. The ratio between the two gives us a sense for how well the algorithm approximates a global optimum.
- Lastly, for cases that cause the algorithm to never halt, we output some statistics about the resulting terrain. We look at the percentage of constraints that are not yet satisfied and the average distance from the goal for those constraints. Note that for many cases it is not certain the algorithm will never halt, however they are halted when the maximum number of iterations I_m is reached. The value for I_m is experimentally chosen.

Furthermore, the value for maximum path slope $g(j)$ for all points $h_j \in S_g$ and the threshold value T_s will be varied for a handful of configurations to measure impact on the first two performance measurements. The roughness threshold value T_r will be discussed individually, as it has some unique properties. Finally, besides numerical results, a handful of sample outputs are visualized in Figure 1 to show what types of results can be expected from the algorithm.

5 Results

Each of the aforementioned measures is plotted against terrain size $N = n \times n$. For most experiments the resolution ranges from $3 \times 3 = 9$ up to $129 \times 129 = 16641$. The value for the number of samples to take N_s was based on how long it took to compute the results for a resolution of 129×129. A value of $N_s = 10$ was chosen. Unfortunately this is a rather low sample size, however going lower in resolution was thought to be more hurting than leaving the sample size lower. A higher resolution took an unrealistic amount of time to run for any configuration that does not reliably satisfy all constraints. Only for measuring the running time, where no roughness constraint was necessary, was the experiment run

on a size of $257 \times 257 = 66049$. The maximum number of iterations I_m was experimentally chosen for lower resolutions. The values of $I_m = 100,000$ and $I_m = 1,000,000$ were chosen for resolutions 129×129 and 257×257 respectively.

All results are plotted against terrain size (i.e resolution). The dotted lines represent the average values of all $N_s = 10$ samples. These are plotted on the same x-coordinates and can be visually compared. Besides these, the plots have the actual data points visualized as circles, sometimes with a boxplot when thought useful. These circles and boxplots are displaced along the x-axis so they can be viewed next to each other, but do note that they do not align with the average lines exactly.

Any image visualizing the output of the algorithm is colored per point in the terrain. The red parts of any image is the path, where only gradient constraints are applied. Any yellow part uses the directional derivative constraint and finally any green part uses the roughness constraint.

5.1 Running Time

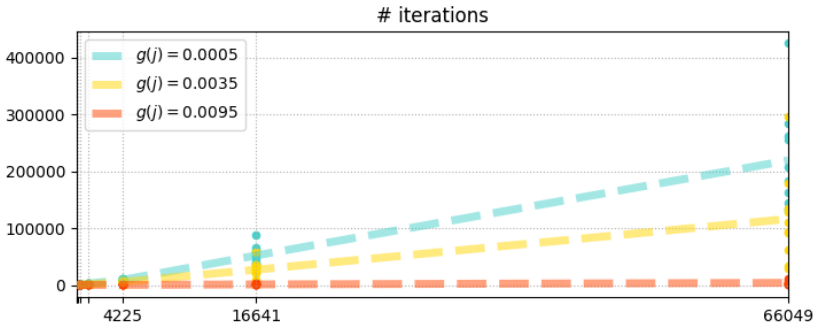

Figure 4: *Number of iterations at which the iterative relaxation terminated plotted against terrain size $N = n \times n$, i.e. the number of points in the terrain (e.g. $66049 = 257 \times 257$). Each line represents the expected number of iterations $I(N)$ for varied values of $g(j)$.*

Figure 4 shows the number of iterations it took to find a solution for increasing terrain sizes starting at 3×3 up to 257×257. We only show results for a single configuration here: using the directional derivative constraint without border stitching, this was done so we could go higher in resolution. The maximum slope $g(j)$ for the entire path through the terrain is varied to show impact on performance. The value $g(j) = 0.0005$ represents a path that is nearly flat, in this case, the terrain has to be modified a lot to satisfy all constraints and a lot of iterations are necessary. The value $g(j) = 0.0035$ represents a reasonable path gradient that you could expect in an adventure type game. Lastly for the value $g(j) = 0.0095$ it is barely required for the terrain to be modified at all, we can see the number of iterations required slowly drops to zero as $g(j)$ increases. For all other experiments the value $g(j) = 0.0035$ is chosen because this value represents a realistic path. See Figure 5 for a visualization of an example output for all three values.

We express the *expected* running time of the iterative relaxation algorithm

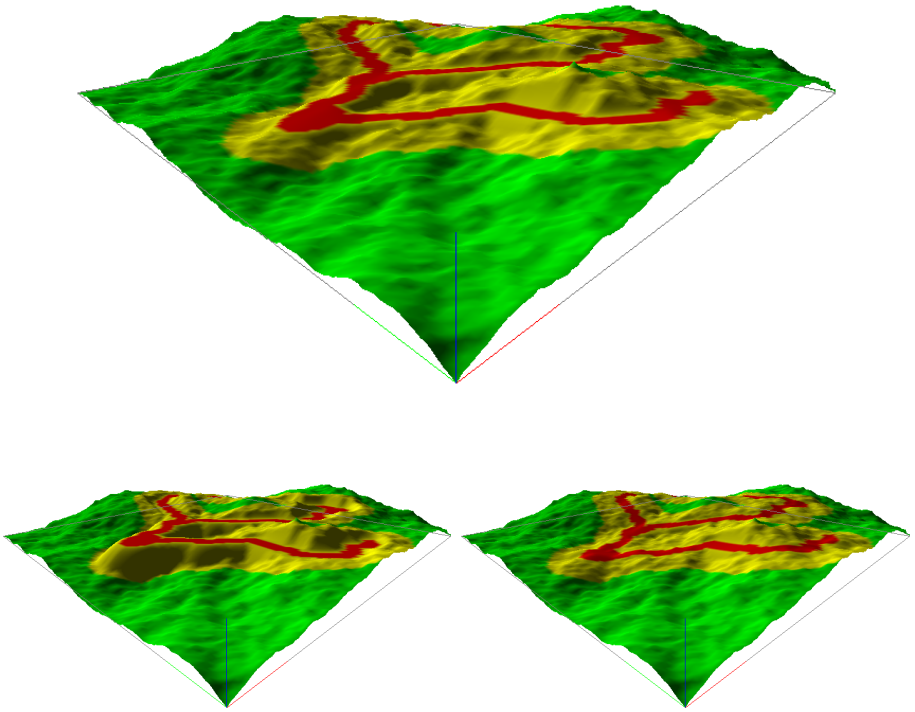

Figure 5: *The same input terrain L and input graph G with different values for $g(j)$. Left: $g(j) = 0.0005$, top: $g(j) = 0.0035$, right: $g(j) = 0.0095$.*

as $I(N) * I_s(N)$, where $I(N)$ is the expected number of iterations for terrain size $N = n \times n$ and $I_s(N)$ is the running time of a single iteration. The latter is linear in the size of the terrain, i.e. $I_s(N) = \Omega(N)$. This is easy to see; there are four types of constraints, that means each of the N points in the terrain has a constant number of constraints applied to them. Furthermore, the running time of solving a single constraint once is constant. Therefore the running time to process one iteration of all constraints on one point is constant and the running time for N points is $\Omega(N)$.

N	81	289	1089	4225	16641	66049
$g(j) = 0.0005$	82.8	385.7	2368.7	8828.1	51781.6	219010.8
$g(j) = 0.0035$	58.2	219.1	1102.5	5276.0	26898.2	116102.7
$g(j) = 0.0095$	11.3	18.3	39.4	155.6	998.3	3744.4

Table 1: *Values of $I(N)$ from $9 \times 9 = 81$ up to $257 \times 257 = 66049$.*

The value of $I(N)$ is plotted in Figure 4, it can be seen this is approximately constant in terrain size N. Unfortunately we cannot make any definitive statements, however the multiplication factor of the value of $I(N)$ appears to be fluctuating around 4, getting larger and smaller as we vary the value of $g(j)$.

This is close to the multiplication factor on the number of points in the terrain, which approaches 4. We can see this a bit clearer in Table 1. Thus we estimate the the upper bound on $I(N)$ to be $O(N)$. This gives us a rough, forgiving estimate on the total running time of the entire algorithm of $O(N)*\Omega(N) = O(N^2)$. However more data is required to draw a definitive conclusion.

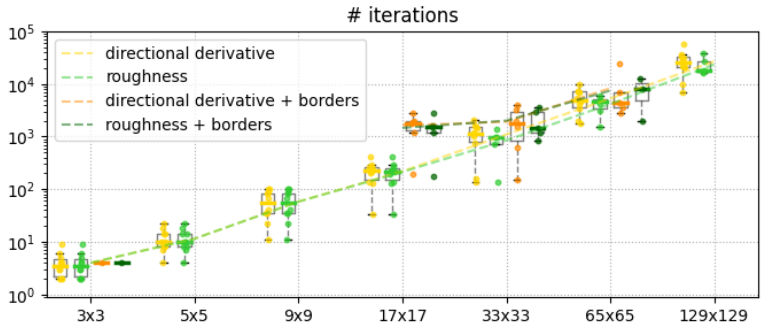

Figure 6: *Number of iterations plotted against terrain size $N = n \times n$ for all four configurations, a logarithmic scale is used to see how they compare.*

All above results only discuss a single configuration that does not use roughness constraints, in Figure 6 the results for all other configurations are shown as well, this only goes up to a resolution of 129×129. Here we can clearly see that the performance of using either the directional derivative or the roughness constraint is nearly identical. See Figure 7 for a visual comparison between the two. Unfortunately we cannot say so about when we try to stitch borders with surrounding patches. A lot of data points are missing for these configurations, this is due to the fact that they are often over-constrained and end up in a state that does not allow the algorithm to halt (i.e. not all constraints can be satisfied). Especially on lower resolution we often lack the ability to properly represent high frequency features described by the constraints, however it still happens at higher resolutions. When and how this happens will be discussed further later on. Still, for the data points that did manage to satisfy all constraints, we can see they do not lie that far from the results we've seen so far.

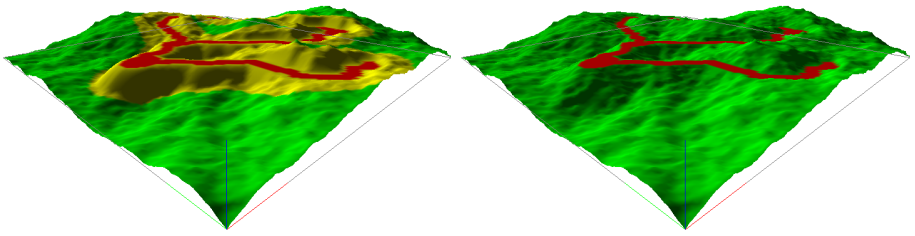

Figure 7: *Comparison of solving for the same input terrain L and input graph G using directional derivative (left) and roughness (right) constraints. A small value of $g(j)$ was chosen to make the differences visible.*

5.1.1 Parallelism

An interesting note is that the algorithm is readily parallelizable. It scales well and can, for example, be implemented on a graphics processing unit. To see this, it is necessary to observe two facts. First; that the implementation of each of the gradient (5), directional derivative (6) and roughness (7) constraints never set the value of any of the points $h_j \in H$ to a new, absolute value. Instead they always add or subtract a value. Second; the set of neighboring points of any point h_j that affect it by solving their constraints is constant. Meaning at any point, the computation necessary to get the new height of any point h_j is invariant during the entire algorithm. Knowing these two facts, we can see that for each point h_j, we can compute all necessary computations individually for the aforementioned constraints and we get a set of values that each of the constraints want to add to or subtract from the height of h_j. We then take the average of those delta values to get the actual value we will add to h_j. The maximum number of values we take the average of is 16 gradient or directional derivative constraints + 9 roughness constraint = 25 values. Finally, the position constraint (8) is solved last such that it is always satisfied exactly.

Given each of the delta values is divided by 25, it takes much more iterations to converge to a solution. On top of that, each constraint is solved multiple times for each of the points it affects. However they can now run in parallel, meaning that the running time of a single iteration $I_s(N)$ can be significantly reduced. In theory, it can be reduced to $O(1)$ by running *all* points simultaneously, meaning that the optimal running time of the algorithm could be $O(N) * O(1) = O(N)$. However in practice it would be unexpected and the value of $I_s(N)$ would likely be a in the form $O(N/p)$, where $p \in \mathbb{N}^+$ is the number of points that can be run in parallel, making a rough estimate of the optimal running time of $O(N^2/p)$.

5.2 Accuracy

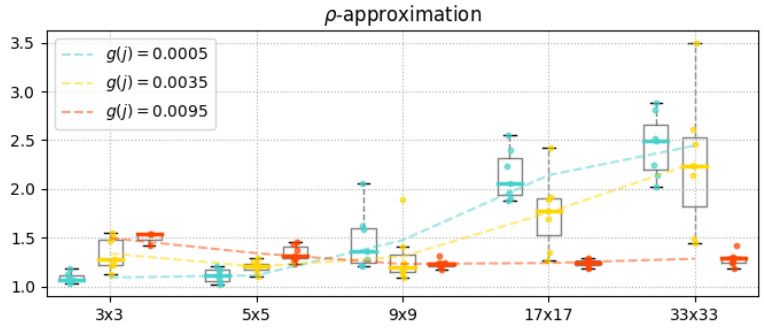

Figure 8: *The approximation ratio, defined as* $\mathbf{EMD}(L, H)/\mathbf{EMD}(L, H_0)$*, plotted against terrain size* $N = n \times n$*. The same data set as in* Figure 4 *is used, where* $g(j)$ *is varied.*

Figure 8 shows the approximation ratio (also called the ρ-approximation) of the found solutions. This value is calculated by dividing the distance from L to the found solution of our algorithm H, denoted as $\mathbf{EMD}(L, H)$ by the same value for an optimal solution found by Gurobi, denoted as $\mathbf{EMD}(L, H_0)$. As can

be seen, for a maximum slope of $g(j) = 0.0035$, representing a realistic path, the approximation ratio increases as the terrain resolution increases. However due to the lack of data for higher resolutions little conclusions can be made about the behavior of this increase. Unfortunately for any resolution above 33×33 Gurobi could not solve for an optimal solution due to running out of resources.

Still, some useful information can be seen in the plot. As explained earlier, by varying the maximum slope $g(j)$, we vary the way the output terrain is supposed to look, in other words; this plot shows how well the algorithm approximates an optimal solution for different path types. We can see that when we decrease $g(j)$, making the path flatter, it does not radically change behavior of the approximation ratio, suggesting that we cannot do much worse. Conversely, when we increase $g(j)$, allowing much of the path to be left unmodified, we can see the approximation gets much closer to an optimal solution. Conceptually this makes sense, it is likely that in this case only slight local changes are made to the terrain, therefore it is less likely to fall in a local minima.

To look a bit closer at the accuracy of the iterative relaxation, the slope threshold value T_s is also varied to measure how close we can get to an optimal solution, see Figure 9. It is crucial to understand that the implementation in Gurobi to get a global optimum does not consider any threshold value. It tries to solve for exact satisfaction of all constraints. The iterative relaxation algorithm does not. Technically, this approximates a different problem, initially this was necessary because the algorithm would never halt due to floating point errors, but as we will be discussed below, it can actually help in getting a minimal running time of the algorithm whilst maintaining a nearly identical solution.

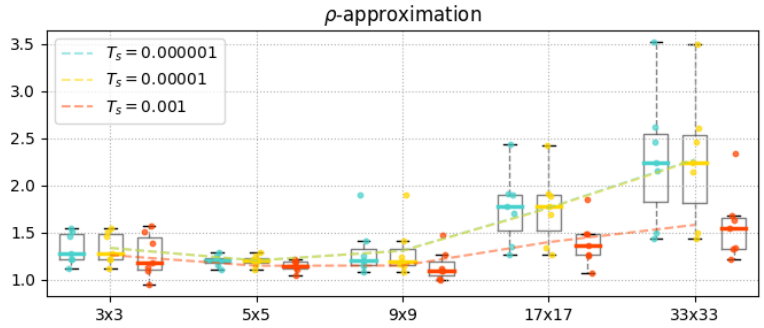

Figure 9: *The approximation ratio plotted against terrain size $N = n \times n$. The slope threshold value T_s is varied while keeping the maximum slope fixed at $g(j) = 0.0035$.*

See Figure 9 again, for comparison, the yellow plot is equivalent to the yellow plot in Figure 8. When we increase T_s we get a solution that seems to be closer to an optimal solution. However we can also see that the approximation ratio occasionally slightly dips below one. This means the value of $\mathbf{EMD}(L, H)$ is smaller than $\mathbf{EMD}(L, H_0)$, which should be impossible. This is due to the fact that the Gurobi program does not account for thresholds. Because the slope threshold is higher, any gradient or directional derivative constraint can be satisfied earlier in the relaxation program whilst they are still far away from their goal values, this gives us too many satisfactory solutions. The conclusion

is that the slope threshold T_s should be small enough in order to properly approximate the actual problem we are trying to solve. A value of $T_s = 0.00001$ seems to do the trick. When we decrease T_s even further we can see we do not get a significantly better approximation, in fact a tenth of the value $T_s = 0.00001$ appears to be yielding nearly identical solutions.

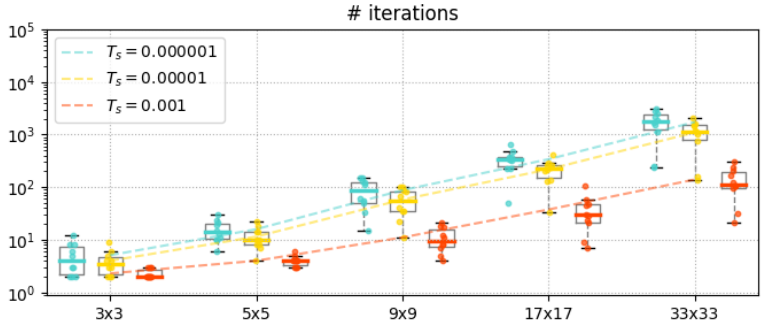

Figure 10: *Number of iterations plotted against terrain size $N = n \times n$. The same data set as in* Figure 9 *is used, where T_s is varied.*

To see how the above helps us in getting a minimal running time, we look at the number of iterations it takes to find a solution again, see Figure 10. As we can see, picking a higher value for T_s clearly results in less iterations required. But more importantly, decreasing T_s beyond $T_s = 0.0001$ still results in more required iterations, whilst not resulting in a better approximation. From this we can conclude that it is not preferable to pick the smallest value for T_s, in fact, for the configurations used here, a value of $T_s = 0.00001$ appears to roughly be the optimal value where we get the best approximation whilst keeping the number of required iterations minimal. When this algorithm is applied in practice, similar experiments should be run for the relevant configurations and resolutions to get the optimal threshold value to use in production.

5.2.1 Infeasible Constraints

All accuracy measurements so far have not discussed any configuration that uses the roughness constraint. This is because this constraint is non-convex and Gurobi could not solve for the optimal solution. On top of that, the iterative relaxation algorithm often would not halt when roughness constraints were used due to the fact that the terrain is often over-constrained, and not all constraints could be satisfied. Unfortunately the possibility that some samples would halt, but did not do so because the maximum number of iterations I_m was reached, cannot be excluded. This means we can make no definitive conclusions about when samples were infeasible.

To dive a bit deeper into what is happening in the case of infeasibility due to roughness constraints, some statistics were gathered about samples that the relaxation did not find a solution for. In Figure 11 the results for all four configurations are shown, it goes up to the highest resolution that could be run in a reasonable amount of time of 129×129. It shows the percentage of all constraints that were left unsatisfied after the algorithm hit the maximum

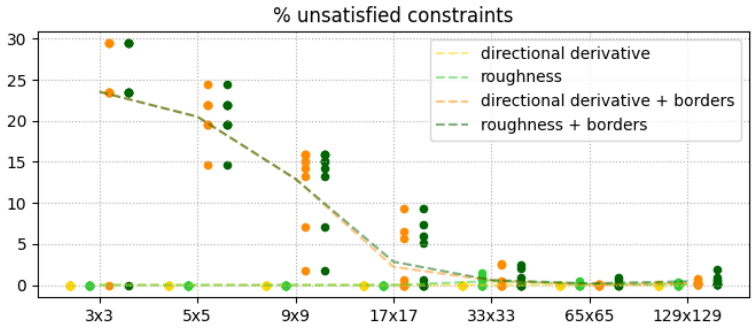

Figure 11: *Percentage of unsatisfied constraints plotted against terrain size $N = n \times n$ for all four configurations.*

number of iterations I_m. We can see that when only the directional derivative constraint was used, all solutions were completely satisfied, however when roughness constraints are introduced some errors start to appear. Conceptually, any terrain with slope constraints only can always be solved; it only imposes a maximum slope, so completely flattening the terrain will always satisfy all constraints. This is not the case for roughness constraints, given the roughness at any point needs to be an exact value (within given bounds defined by the roughness threshold T_r). When trying to stitch borders with surrounding patches, we can immediately see some major errors on the lower resolutions. As stated previously, these low resolution terrains simply lack the ability to properly represent high frequency details. The percentage of errors appears to decrease as resolution goes up, unfortunately there is no way of telling what happens at extreme resolutions. See Figure 12 for a visual demonstration of stitching together the borders of patches.

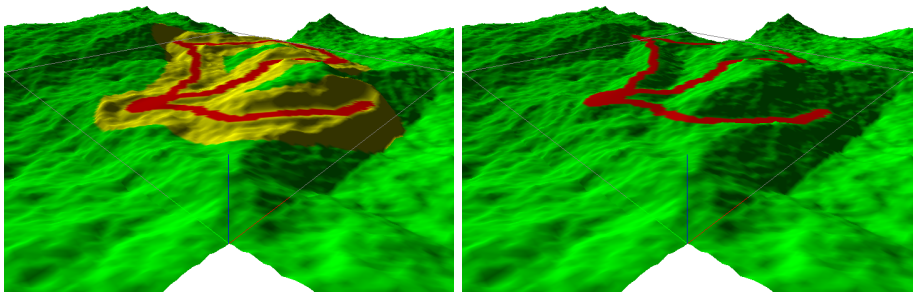

Figure 12: *Comparison of 'solving' for the same input terrain L and input graph G using directional derivative (left) and roughness (right) constraints in conjunction with position and roughness constraints to stitch the borders of the patch to that of its neighbors'.*

There are two cases of over-constraining a terrain that can be visually verified. The first is when roughness constraints conflict with a necessary slope along a stretch of terrain such that position constraints at the borders of the

patch are satisfied. This often happens in the corners of the terrain, or when the path is close to the border of the terrain. See Figure 13 for a visualization, the surrounding terrain is not modified anymore, but in the lower right corner we still see some flickering between the same states, meaning the constraints will never be satisfied.

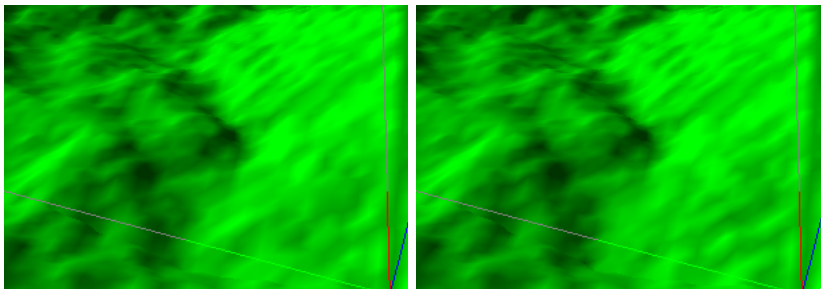

Figure 13: *Demonstration of a set of infeasible constraints due to conflicting roughness and position constraints at the border of a terrain. The two images were taken during the relaxation algorithm at different iteration numbers, some flickering can be seen.*

The second case is when roughness and gradient constraints are in conflict. Remember that gradient constraints are always used to model the path, when there is a small area in-between the path that has a roughness constraint, the two constraints form a race condition. One wants to flatten the area, the other wants to roughen it. See Figure 14 for a visualization, the area of terrain keeps rotating between two states, never allowing either constraint to be satisfied.

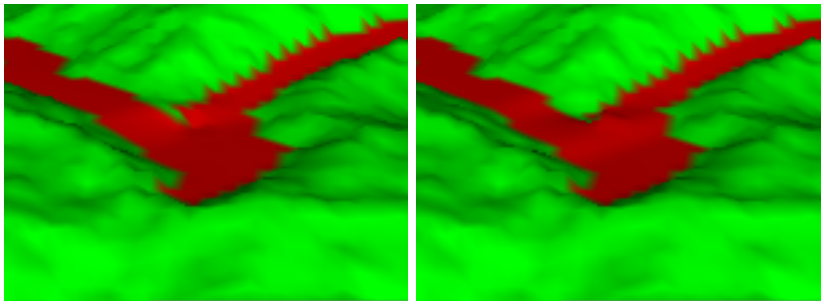

Figure 14: *Demonstration of a set of infeasible constraints due to conflicting roughness and gradient constraints along the path. A 'rotating' movement can be seen.*

Look back at the plot in Figure 11, when we use directional derivative constraints to solve the area around a path, it still shows the same percentage of unsatisfied constraints. However the case where gradient and roughness constraints are in conflict cannot happen, therefore it means the case that happens most often is the former, where roughness and position constraints are in conflict at the border of the terrain. At resolution of 65×65 and up we see that when we use the directional derivative constraints, we have slightly less errors. This decrease is explained by the lack of gradient and roughness constraints being in

conflict.

There is one property of the roughness constraint that is left untouched, which is the roughness threshold T_r. See Figure 15 for a snapshot of the terrain during the relaxation process for three different values of T_r. The value $T_r = 0.04$ is used for all other experiments. When we increase the value of T_r we can see the terrain around the path is barely modified. However when we decrease the value of T_r something interesting happens; the terrains tends to flatten and converge towards a noisy, flickering plain. Meaning that attempting to exactly match the roughness at every point in H with that of every corresponding point in L does not preserve the terrain, instead it launches it in a very unstable state. This is explained by the fact that the roughness constraint only sees high frequency details; namely those between points and their direct neighbors. Low frequency details, such as the overall curvature of a hill or mountain, are not registered by the local constraint. Therefore these features can accidentally be squeezed flat when the threshold is too tight, which unfortunately appears to always be the case. When this algorithm is applied in practice, a proper value for T_r should be empirically chosen such that it is just large enough to allow low frequency terrain features to remain.

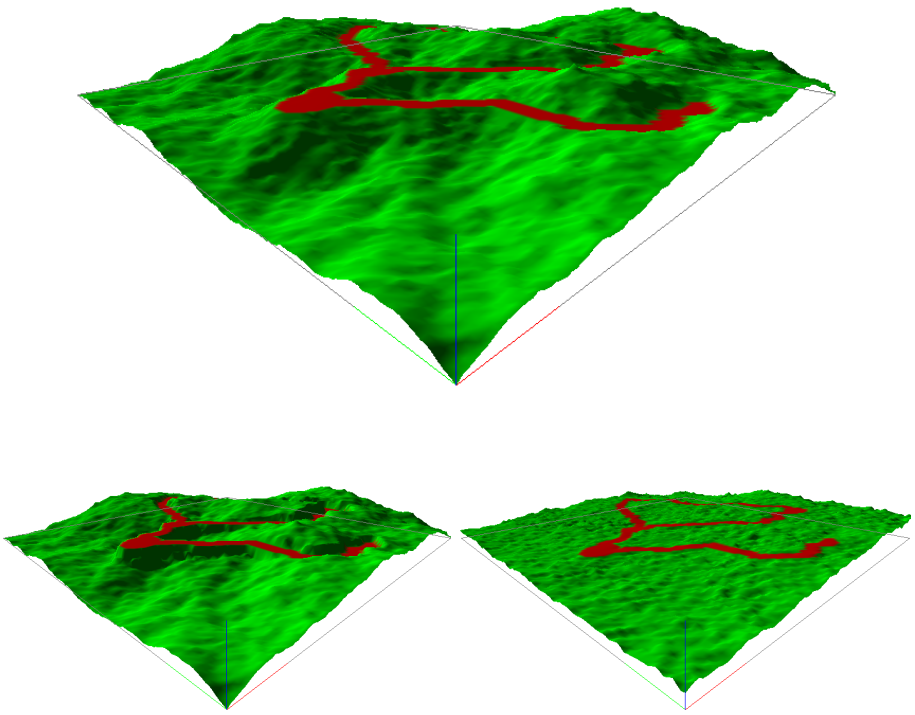

Figure 15: *Comparison of 'solving' for the same input terrain L and input graph G with different values for T_r. Left: $T_r = 0.2$, top: $T_r = 0.04$, right: $T_r = 0.004$.*

6 Discussion

The proposed algorithm attempts to solve an optimization problem based on minimizing the Earth Mover's Distance between input terrain L and output terrain H. Four constraints are added to give rise to desired features in the terrain. These constraints make it a quadratic programming problem, with one constraint being non-convex. Due to this complexity the running time would be unrealistically high for use in games, for this reason the choice of an approximation algorithm was made. Because the search space is huge, an iterative relaxation algorithm was designed that greedily picks the next successor state each iteration. The drawback being that the output is less accurate, as it can easily fall into a local minima. For a visual comparison between the solution found by our algorithm and the optimal solution found by Gurobi for the same inputs, with a terrain resolution of 33×33, see Figure 16. The optimal solution costs less, but the solution of our algorithm looks more natural. Therefore it might be worth considering that getting as close as possible to an optimal solution is not always preferred. This begs the question if there exists a more 'realistic' optimal solution, how many of those exist, and what 'realistic' really means mathematically.

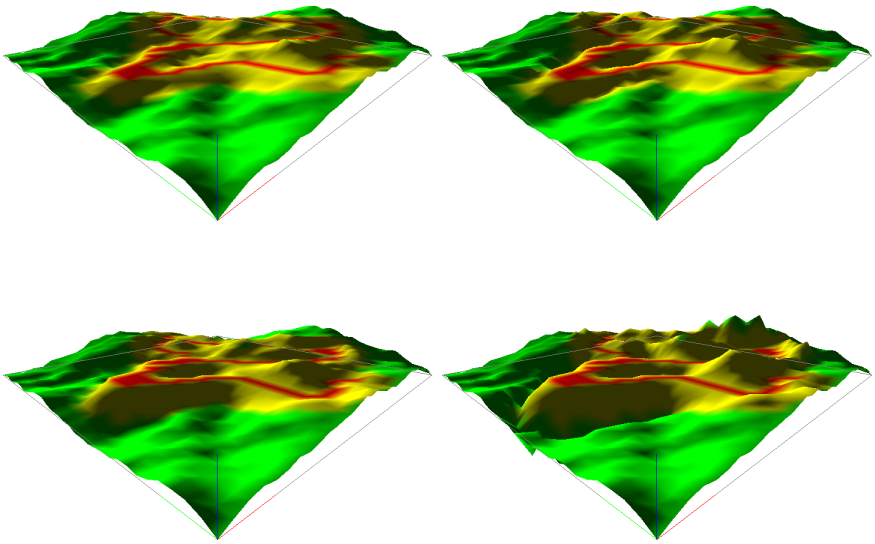

Figure 16: *Comparison of the solution H found by the iterative relaxation program (left column) and the optimal solution H_0 found by Gurobi (right column). Top row: $g(j) = 0.0035$, bottom row: $g(j) = 0.0005$.*

The individual constraints that were designed are the gradient (5), directional derivative (6), roughness (7) and position (8) constraints. The goal values for each constraint are fixed at the start of the algorithm, but they can be any value, they can even vary across the terrain. The results only show a constant maximum slope $g(j)$ across the terrain, however the algorithm can easily han-

dle paths of different slopes. Furthermore, the input graph G, and method of pre-processing it to get subdivision S can be swapped out for any process. This makes the algorithm very flexible in the variety of paths it can output.

The four constraints and their implementations are only a proposal, much more constraints could be conceived of, however manual implementation is required. Some constraints could be improved, the directional derivative constraint over-compensates for directions perpendicular to $D(j)$. Compared to the roughness constraint, it is visually less appealing, and the roughness constraint seems more promising. However, as opposed to the directional derivative constraint, the roughness constraint is very unstable, meaning that it could easily cause conflicts in constraints, causing the problem to be infeasible and the algorithm not to converge. Still, when used in practice, a reasonable output is still wanted, even though it may not satisfy all constraints exactly. Unfortunately the algorithm has no way of exiting for this case, except for waiting to run the maximum number of iterations I_m.

From the proposed constraints, a natural solution flows that solves the problem of stitching together the borders of neighboring patches. This could be very useful for use in seemingly infinite worlds. However this is entirely based on position and roughness constraints and does not always result in a realistic looking terrain. It also makes the algorithm even more unstable. Another constraint would be preferable that somehow describes how the points at the border of a terrain should behave to be 'seamless'. An experiment was done using more position constraints to match the first order derivative at the borders of neighboring terrains. However this was proven very unsuccessful, the results were almost identical and it was left out of consideration.

Lastly, results for higher resolutions were hard to gather due to the lack of resources to generate sufficient data. However from the available data some characteristics could still be deduced. A rough, forgiving estimate of the running time was given. However no definitive statements could be made. How well the algorithm approximates a global optimum and in what situations in performs best was shown, however its behavior could not be exactly defined. Still, all experimentation proved sufficient in showing what kind of threshold values are preferable and what future work might be done to improve the work.

6.1 Future Work

There are a lot of facets of the current work that could be improved given more resources to work on it. Firstly, more data could be gathered to increase the accuracy and range of the results. The experiments should preferably be run on higher resolutions so the asymptotic behavior of the algorithm can be defined more precisely. Especially for the roughness constraint more data could be gathered, unfortunately this is hard to do because of the complexity of the optimization problem. Another strategy could be to look deeper into the theoretical side of the constraints and determine when a constraint will always converge to a solution.

Besides numerical and theoretical work, the iterative relaxation easily lends itself to design of new constraints. Obviously new ideas for modeling the path and its surrounding areas could be thought up. But more specifically, the process of stitching borders of neighboring patches could be improved a lot. As mentioned earlier, an experiment was done using position constraints to main-

tain the first order derivative at the borders. A constraint that explicitly tries to model this property across a wider band of points along the border could be looked into. Furthermore, the roughness currently only registers high frequency details of the terrain. If the constraint was applied at multiple scales on the same terrain, it could also suffice as a method to maintain terrain features of all frequencies. The only downside is the danger of over-constraining the terrain even further, making the possibility of the algorithm never converging even larger. More research could go into this subject.

Besides new or better constraints, the pre-processing of graph G into subdivision S is open for improvement or even new algorithms, as it can easily be swapped out with some other process. The path finding in this work is a simple implementation of the A^* algorithm with a cost function such penalties are given for steep slopes. This could be modeled differently, catered for a specific kind of shape of the terrain. On top of that, one could think of a different way of generating base terrain L. Currently a simple midpoint-displacement method is used, however this process could also take into account the shape of G, generating a base terrain that already facilitates some of the desired features in the output terrain. This could make the job of the iterative relaxation process easier and maybe even faster.

More research could go into improving the performance of the iterative relaxation process. In its current state, the most beneficial way to try to do this is to come up with an early-exit strategy. Currently the algorithm often does not converge when roughness constraints are introduced due to conflicting constraints. If a method was devised that detects such a conflict one of two things could be done: first, attempt to modify the state of the terrain such that it does converge and second, simply exit the process earlier, such that we do not have to wait for the maximum number of iterations I_m to be reached.

Lastly, the pipeline that is shown in this work can be augmented or used as inspiration for new work. One interesting school of thought is to expand the dimensionality the terrain can express. The terrain is currently stored as a finite heightmap, very similar to the method that iteratively generates new terrains based on constraints [21]. This paper was later extended to voxels [7]. A very similar expansion could be applied to our iterative relaxation, making it deal with voxels, which allows for caves, overhangs, tunnels or even floating islands. The algorithm, whether it runs on heightmaps or voxels, was designed to be used in for adventure type games that want to introduce infinite terrain to emphasize exploration play. To work further towards this goal, this work could be combined with other work that generates missions and rules [12] that fill the world up with interesting gameplay.

7 Conclusion

Looking at the proposed algorithm from a zoomed out perspective, it attempts to solve the following problem: *given a square patch of terrain L, transform it into a new patch of terrain H by modifying L such that a given planar input graph G is embedded into H as a path that can be walked upon.* To be able to walk on a path means that the *gradient* of every point of the path in any direction must not exceed a given maximum value. This is done by introducing an iterative relaxation algorithm that does just this; it carves a path through

the terrain where the gradient at every point of that path does not exceed this maximum value. From this it flows that the entire graph is properly represented in the terrain, each node and graph can be walked upon. An argument for a rough, forgiving running time of $O(N^2)$ is shown. The algorithm is readily parallelizable, cutting the running time to a potential $O(N^2/p)$, where $p \in \mathbb{N}^+$ is the number of points that can be run in parallel. However such an implementation is not yet built and should be properly evaluated.

The gradient constraint is one of four constraints that together attempt to embed a path in such a way that we *maintain the 'realism' of output terrain H by imposing mathematical constraints based on graph G.* The iterative relaxation algorithm approximates an optimal solution of an optimization problem, the four constraints are added to this problem's own constraints. The directional derivative and roughness constraints are two distinct options to maintain the 'realism' of the terrain, however what 'realistic' means remains an open question; it is up to the designer of the constraints. Furthermore, not all constraints allow for guaranteed convergence of the algorithm, for this reason, the maximum number of iterations I_m, or an early-exit strategy should be carefully chosen.

The optimization problem itself imposes that *graph G should be embedded in output terrain H such that it has changed as little as possible from L.* It does this by minimizing the cost of transforming L into H by moving, creating or destroying material, or more formally; the Earth Mover's Distance $\mathbf{EMD}(L, H)$ is minimized. The proposed algorithm's results were compared to an optimal solution to this problem to see how accurate it is. We can conclude that the algorithm performs better when the constraints require less modification of the base terrain, however not much can be said about the asymptotic behavior as we scale the terrain resolution. To hone in on optimal accuracy, experiments should be run to find a value for the slope threshold T_s. For the roughness threshold T_r, the value should be empirically chosen.

From the proposed formulation and constraints a natural solution flows that solves the following problem: *given set height values for the border of H, modify its interiors such that it 'realistically' connects to its borders.* It combines position and roughness constraints to set the values and modify the interiors respectively. This implementation is more of an experiment to see how much the constraints can handle, more research is desired.

References

[1] Gurobi optimizer reference manual, 2020. http://www.gurobi.com.

[2] D. Adams, P. Egbert, and S. Brunner. Feature-based interactively sketched terrain. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 208–208, 2012.

[3] D. Adams et al. Automatic generation of dungeons for computer games. *Bachelor thesis, University of Sheffield, UK.*, 2002.

[4] D. Ashlock and C. McGuinness. Landscape automata for search based procedural content generation. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[5] S. Bakkes and T. Dam. The ace2 model: Refining bartles player taxonomy for creation play. *GAME-ON' 2019*, 2019.

[6] S. Bakkes, S. Whiteson, G. Li, G. V. Vişniuc, E. Charitos, N. Heijne, and A. Swellengrebel. Challenge balancing for personalised game spaces. In *2014 IEEE Games Media Entertainment*, pages 1–8. IEEE, 2014.

[7] M. Becher, M. Krone, G. Reina, and T. Ertl. Feature-based volumetric terrain generation. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–9, 2017.

[8] F. Belhadj. Terrain modeling: a constrained fractal model. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 197–204, 2007.

[9] B. Beneš and R. Forsbach. Visual simulation of hydraulic erosion. 2002.

[10] N. Chiba, K. Muraoka, and K. Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation*, 9(4):185–194, 1998.

[11] G. Cordonnier, J. Braun, M.-P. Cani, B. Benes, E. Galin, A. Peytavie, and E. Guérin. Large scale terrain generation from tectonic uplift and fluvial erosion. In *Computer Graphics Forum*, volume 35, pages 165–175. Wiley Online Library, 2016.

[12] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011.

[13] P. Douglas, T. Gard, V. Arnold, and N. McCree. Tomb raider, 1996.

[14] M. Frade, F. F. De Vega, and C. Cotta. Modelling video games landscapes by means of genetic terrain programming-a new approach for improving users experience. In *Workshops on Applications of Evolutionary Computation*, pages 485–490. Springer, 2008.

[15] M. Frade, F. F. de Vega, and C. Cotta. Evolution of artificial terrains for video games based on accessibility. In *European Conference on the Applications of Evolutionary Computation*, pages 90–99. Springer, 2010.

[16] M. Frade, F. F. de Vega, and C. Cotta. Automatic evolution of programs for procedural generation of terrains for video games. *Soft Computing*, 16(11):1893–1914, 2012.

[17] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin. Procedural generation of roads. In *Computer Graphics Forum*, volume 29, pages 429–438. Wiley Online Library, 2010.

[18] J.-D. Génevaux, É. Galin, E. Guérin, A. Peytavie, and B. Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32(4):1–13, 2013.

[19] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time procedural generation ofpseudo infinite'cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 87–ff, 2003.

[20] M. Grumet. Terrain modeling. *Report, Institute of Computer Graphics and Algorithms Vienna University of Technology*, 2004.

[21] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin. Feature based terrain generation using diffusion equation. In *Computer Graphics Forum*, volume 29, pages 2179–2186. Wiley Online Library, 2010.

[22] K. R. Kamal and Y. S. Uddin. Parametrically controlled terrain generation. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 17–23, 2007.

[23] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[24] X. Mei, P. Decaudin, and B.-G. Hu. Fast hydraulic erosion simulation and visualization on gpu. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 47–56. IEEE, 2007.

[25] S. Murray, G. Duncan, R. Doyle, D. Ream, and W. Braham. No man's sky, 2016.

[26] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *ACM Siggraph Computer Graphics*, 23(3):41–50, 1989.

[27] B. Neidhold, M. Wacker, and O. Deussen. Interactive physically based fluid and erosion simulation. In *Eurographics workshop on natural phenomena*, 2005.

[28] J. Olsen. Realtime procedural terrain generation-realtime synthesis of eroded fractal terrain for use in computer games. 2004.

[29] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):54–67, 2010.

[30] M. Persson and J. Bergensten. Minecraft, 2011.

[31] J. Pilested and E. Englund. Magicka, 2011.

[32] W. L. Raffe, F. Zambetta, and X. Li. Evolving patch-based terrains for use in video games. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 363–370, 2011.

[33] R. Ramstedt and J. Smed. Midpoint displacement in multifractal terrain generation. In *Proceedings of the International Conference on Intelligent Games and Simulation*, 2016.

[34] J. Rankin. The uplift model terrain generator. *International Journal of Computer Graphics & Animation*, 5(2):1, 2015.

[35] S. J. Riley, S. D. DeGloria, and R. Elliot. Index that quantifies topographic heterogeneity. *intermountain Journal of sciences*, 5(1-4):23–27, 1999.

[36] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[37] J. Schneider, T. Boldte, and R. Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on gpus. In *Vision, modeling and visualization*, volume 2006, pages 145–152, 2006.

[38] S. Stachniak and W. Stuerzlinger. An algorithm for automated fractal terrain deformation. *Computer Graphics and Artificial Intelligence*, 1:64–76, 2005.

[39] O. Šťava, B. Beneš, M. Brisbin, and J. Křivánek. Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 201–210. Eurographics Association, 2008.

[40] F.-X. Talgorn and F. Belhadj. Real-time sketch-based terrain generation. In *Proceedings of Computer Graphics International 2018*, pages 13–18. 2018.

[41] F. P. Tasse, A. Emilien, M.-P. Cani, S. Hahmann, and A. Bernhardt. First person sketch-based terrain editing. In *Proceedings of Graphics Interface 2014*, pages 217–224. Canadian Information Processing Society, 2014.

[42] J. Togelius, N. Shaker, and M. J. Nelson. Constructive generation methods for dungeons and levels. In N. Shaker, J. Togelius, and M. J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 31–55. Springer, 2016.

[43] J. Togelius, N. Shaker, and M. J. Nelson. Fractals, noise and agents with applications to landscapes. In N. Shaker, J. Togelius, and M. J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 57–72. Springer, 2016.

[44] M. Toy, G. Wichman, K. Arnold, and J. Lane. Rogue, 1980.

[45] R. Van Der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2013.

[46] W. Writh, A. Hutchinson, J. Chalmers, C. Gingold, S. Librande, and S. Johnson. Spore, 2008.

[47] H. Zhou, J. Sun, G. Turk, and J. M. Rehg. Terrain synthesis from digital elevation models. *IEEE transactions on visualization and computer graphics*, 13(4):834–848, 2007.