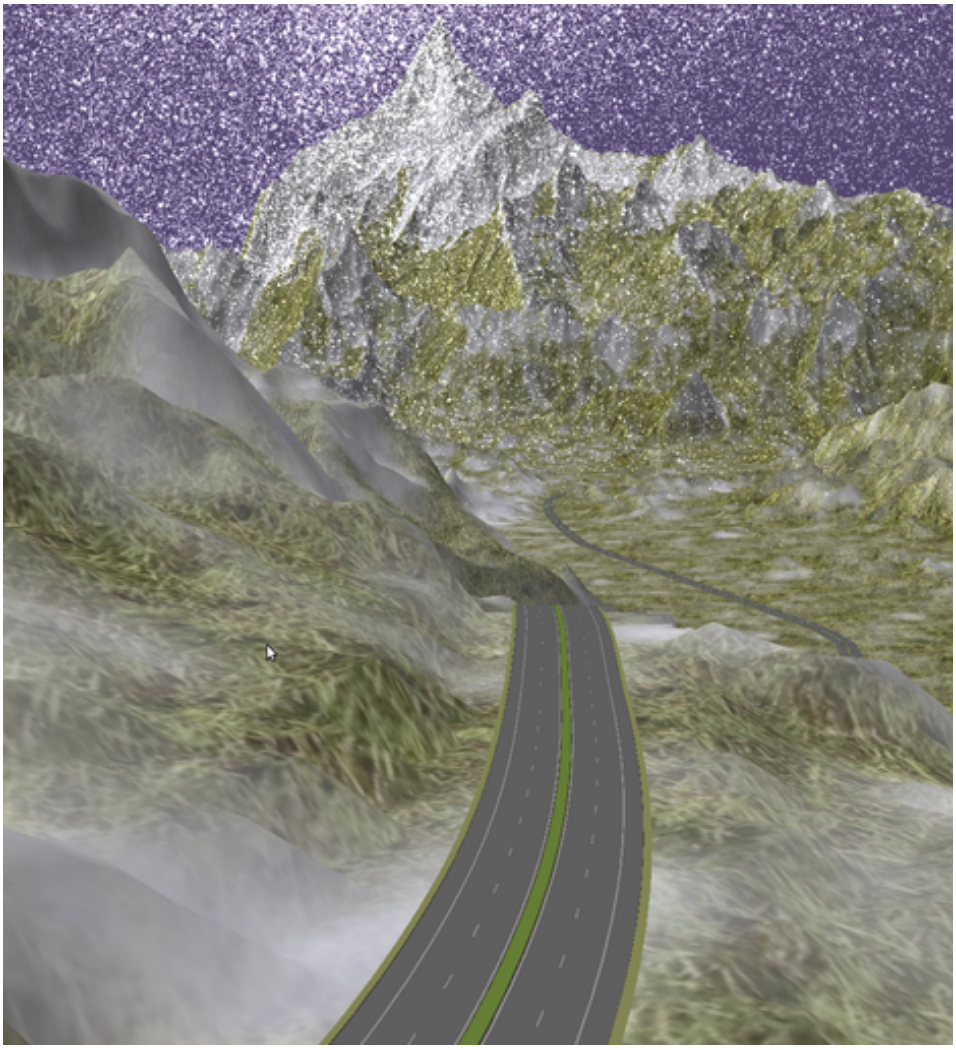
Procedural Generation of Roads for use in the Snow Simulator



Hallgeir Lien

NTNU – Trondheim
Norwegian University of
Science and Technology

Fall 2011

Problem Description

Earlier master students at NTNU have implemented a snow simulator, which models each snow flake as a particle, and also models wind as a fluid, which is simulated using computational fluid dynamics. The original snow simulator has been modified to also use Lattice-Boltzmann methods for simulating the wind field. In this project, we look at integrating the snow simulator with procedurally generated roads. I implement the procedural road generation algorithm from [1], and import this as a mesh into the scene. I also look at importing real-world terrains by implementing a USGS DEM file format parser that outputs a 16-bit height map.

Abstract

In this report, I present an integration of a road generation algorithm, described in [1], with the GPU accelerated snow simulator developed at the HPC lab at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). I also present a USGS DEM to RAW height map converter, for using real-world maps in the snow simulator.

Given a height map of a terrain, my road generation application generates a road that minimizes a cost function that depends on the length, slope and curvature of the road. This is accomplished using an optimized A* search, generating a discrete shortest path represented by a series of nodes.

The road trajectory is computed by using the nodes of the discrete shortest path as control points in a clothoid spline, which has a curvature that varies linearly with the length of the curve. The trajectory is used to generate a RoadXML file, which is a description of the road (trajectory, road profile, textures, etc.); this RoadXML file is then used in SCANeR Studio developed by Octal[2] to generate a triangle mesh of the road.

The mesh generated by SCANeR Studio is stored in the Wavefront OBJ format, and this file is read to load the mesh into the snow simulator for rendering. Materials defined in the OBJ file is also loaded. The terrain is adjusted to accommodate the road model as necessary, by constructing a distance map for each vertex in the road mesh, then adjusting the terrain elevation according to the closest vertices.

I present visual results showing the roads as they appear in the terrain. I also present performance results where different grid resolutions are tested; these results show that for many terrains, a lower resolution may not give a much higher cost of the trajectory, while giving much lower running time on searches. In addition to this, I present performance results on using Dijkstra's algorithm vs an A* search for generating the roads. The results show a clear difference, A* being up to 2.3 times faster, averaging at 1.7 times faster using the four test maps.

Acknowledgements

I would like to thank Dr. Anne C. Elster for being my primary advisor for this project. A special thanks is due for allowing us to present this project as well as the earlier work on the snow simulator at the Supercomputing 2011 conference in Seattle, which was a great experience. Her help and advice I got on this project have been very valuable.

I would also like to thank Dr. Jo Skjermo at Vegdirektoratet for both giving me a chance to work on an exciting project like this one, as well as being my co-advisor on this project, giving me a lot of helpful tips and resources on optimizing A* searches, as well as giving valuable feedback on my report, and other very appreciated helpful tips.

Lastly, I want to thank PhD candidate Rune E. Jensen for spending quite a bit of time with me debugging the snow simulator code as well as retrieving the original source code for the snow simulator from gigabytes of an unorganized pile of old HPC lab backups.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this report, I present my work of integrating procedurally generated roads with a snow simulator developed at the High Performance Computing lab at NTNU. I implement the road generation algorithm from [1], output a RoadXML file[3], use an external tool to generate a road mesh, and finally import this mesh into the snow simulator.

The first part of this project, generating the road trajectory, is done using an A* graph search with points on a height map as the nodes, i.e. we perform a discrete shortest path search over the (potentially) continuous domain that is the terrain. Then, in order to generate a RoadXML file, the actual trajectory is computed from the discrete shortest path by using the nodes along the path as control points for a clothoid spline. The clothoid spline is constructed using the method described in section 2.4.

A road mesh is generated using SCANeR Studio from Octal, by importing the RoadXML file. A triangle mesh is output in the Wavefront OBJ format, which is then parsed by the snow simulator. Finally, the terrain is adjusted so it will match the road model; this is necessary due to the fact that the road trajectory elevation is sampled at discrete intervals when generating the RoadXML file, and that the road has a width which is not accounted for in the RoadXML.

1.1 Project scope

The scope of this project is first and foremost to implement the road generator in [1], and load these roads as meshes into the snow simulator. The project aims to test the performance of the road generation algorithm at different grid resolutions as well as comparing it with Dijkstra's algorithm, and present visual results showing roads rendered in the snow simulator. It is beyond the scope of this project to fully integrate the road generation algorithm, as proposed in chapter 5, Future Work. Instead, an application external to the snow simulator is developed that generates roads and outputs a RoadXML file.

1.2 Report organization

First, the report will go through some of the background material used in this project in chapter 2; this includes solving shortest path problems, heuristic search, an overview on the NVIDIA GPU architectures and road generation. Then, the implementation of the road generator (which includes the A* search), integration with the snow simulator and the USGS DEM parser is presented in chapter 3. Visual results as well as performance results are presented in chapter 4. Finally, chapter 5 summarizes some of the possible improvements and future projects, and chapter 6 concludes.

Chapter 2

Background

This chapter will give some background on important algorithms and file formats used in this project, like the A* graph search algorithm and a method for creating clothoid splines given a set of control points. It will also give a short overview of the snow simulator and of GPU computing.

2.1 Shortest Path Problems

Finding the shortest path between nodes in a graph is an old and heavily researched problem. The problem can be stated like this: Given a weighted graph $G = (V, E)$ with edges E and vertices V, and a weighting function $w(e) = w(v_i, v_j)$ of edge $e = (v_i, v_j)$, find the path from v_a to v_b, that solves the minimization problem

$$\min_{\mathbf{P}} \left\{ \sum_{e \in \mathbf{p}} w(e) \right\} \tag{2.1}$$

over all paths $\mathbf{P}$ between nodes v_a and v_b.[4]

2.1.1 Solving shortest path problems

There are many algorithms to solve the minimization problem of finding the shortest path. There are two main categories: Single-source, and all-pairs. Single-source solves the shortest path algorithm for a single source node a, to one or all other nodes. All-pairs finds, as the name suggests, the shortest path between all the nodes in the graph. In this project, we are concerned about the single-source shortest path algorithms. In particular, we are interested in finding the shortest path between one node v_a to another single node v_b.

Dijkstra's algorithm and A* search

One of the most well known and efficient single-source algorithms when we assume that we have no knowledge of the graph, is Dijkstra's algorithm. Dijkstra's algorithm

is described in algorithm 1. The basic idea behind the algorithm is to always choose to look at the neighbors of the node currently closest to the starting node that has not yet been explored. To retrieve the actual path, we keep a pointer to the predecessor of each node, which is updated once a better path to that node has been found.

Algorithm 1 Pseudo code for Dijkstra's algorithm

```
Initially, set the distance to the source d(v_a) = 0 and for all other nodes d(v_i) = ∞.
Let π(v) be the predecessor to node v; initially, ∀v(π(v) = none)
Put all nodes v_i in a priority queue, sorted on d(v_i).
while |Q| > 0 do
  v := next node in Q
  Q := Q − {v}
  for all neighbors w of v do
    if d(v) + w(v, w) < d(w) then
      d(w) := d(v) + w(v, w).
      π(w) := v
    end if
  end for
end while
```

After the algorithm finishes, we have the total cost of the shortest paths from the source v_a to all other nodes. The time complexity, when implemented efficiently with a fibonacci heap, is $O(|E|+|V|\log|V|)$[5]. One useful observation is that the shortest path to the node we pick from the queue is optimal; this is due to the fact that all other nodes in the queue has a larger or equal distance from v_a, and thus, assuming non-negative edges, we can not hope to find a better path.. Because of this, if we are only interested in the path from v_a to v_b, we can stop the algorithm once we pick v_b from the queue.

Utilizing a priori knowledge about graph topology: A*

In many practical cases, we have specific knowledge about the graph topology. For instance, if we seek to find the shortest path from the city of Oslo to Trondheim, we know, for instance, that if we move in the right direction (i.e. north), we are most probably closer than we were, and we know that we need to travel AT LEAST the straight line distance from Oslo to Trondheim. This kind of knowledge can be utilized in making more efficient single-source, single-destination shortest path algorithms, by computing heuristics on the cost from each node to the goal. One such algorithm is A*.[6]

A* is an extension of Dijkstra's algorithm. We define $g(x)$ as the distance from the start node v_a to node x ($g(x)$ is the same as $d(x)$ in Dijkstra's algorithm, but it is common to denote this function as $g(x)$ in literature). We also define the heuristic $h(x)$ as an *admissive* (optimistic) and *consistent* (monotone) estimate of the cost from node x to the goal node v_b. That the heuristic is optimistic simply means that it never over-estimates the distance left to the goal, i.e. $h(x) \leq h^*(x)$, where $h^*(x)$ is the *o*ptimal heuristic function, that give the exact cost from node x to the goal node v_b. That the heuristic is consistent, means that the search will never take a step back. This is described later.

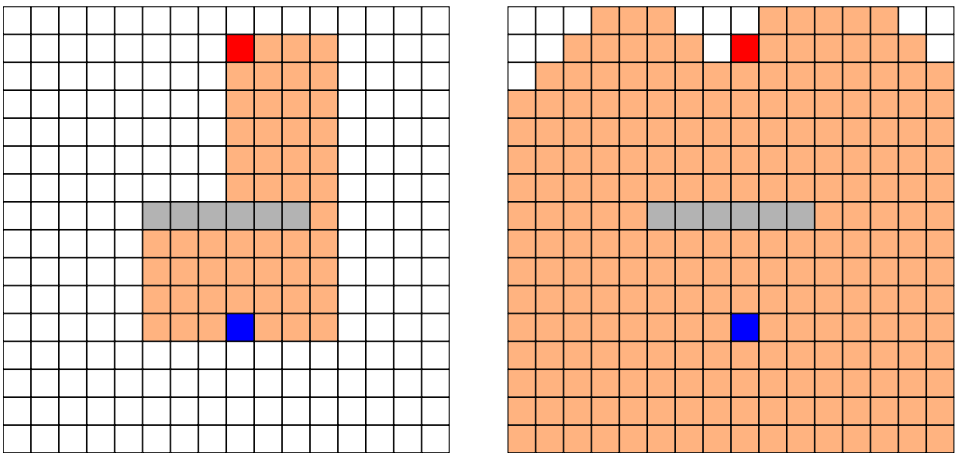
Combining these values gives the *f*-cost $f(x) = g(x) + h(x)$, which is the actual

estimate on the distance from the start node v_a to the goal node v_b. The algorithm is outlined in algorithm 2.

Algorithm 2 Pseudo code for A* shortest path algorithm

Maintain a queue $OPEN$ and a set $CLOSED$.
Initially, $OPEN = \{v_a\}$ and $CLOSED = \emptyset$.
Let $\pi(v)$ be the predecessor to node v; initially, $\forall v(\pi(v) = \text{none})$
while $|Q| > 0 \wedge v_b \notin CLOSED$ **do**
  $v :=$ node with smallest f-cost in OPEN
  $OPEN := OPEN - v$
  $CLOSED := CLOSED + \{v\}$
  for all neighbors w of v **do**
    if $w \notin CLOSED \wedge w \notin OPEN$ **then**
      Put w in the $OPEN$ queue.
    else if $w \in OPEN \wedge g(v) + w(v, w) < g(w)$ **then**
      $g(w) := g(v) + w(v, w)$
    end if
  end for
end while

The main difference from the regular Dijkstra's algorithm, is how nodes are picked from the queue. Instead of picking the node with the lowest $g(x)$ as we do with Dijkstra, we pick the node with the lowest f-cost, $f(x) = g(x) + h(x)$. This way we prefer nodes that are "closer" to the goal. As a side note, if we set $h(x) = 0$, the A* algorithm is reduced to Dijkstra's algorithm. If we use the example of finding the shortest route from Oslo to Trondheim, if we use Dijkstra, nodes will be expanded in all directions, i.e. Dijkstra will search both south, west and east, which is unnecessary work. If we instead use A*, nodes south of Oslo may still have a small distance from the start, but will have a higher $h(x)$ than those to the north, and thus may not be selected for expansion. Figure 2.1 shows the expanded nodes for a simple shortest path problem where $w(v, w) = 1$ for all pairs of neighbors v, w, and the neighbors are the neighboring four nodes.



(a) A* expansions (b) Dijkstra expansions

Figure 2.1: Expanded nodes in A* vs. Dijkstra

Designing the heuristic. Admissibility and consistency

In order to guarantee that the A* algorithm is optimal, the heuristic function $h(x)$ must be admissible and consistent[6]. That $h(x)$ is admissible means that $h(x) \leq h^*(x)$ where $h^*(x)$ is the optimal heuristic that gives the exact distance to the goal from a node x. More intuitively, this means that $h(x)$ is an *o*ptimistic estimate of the cost: the estimate is at most the actual cost. For path finding in euclidean space, e.g. in $\mathbb{R}^{\nVDash}$, an admissible heuristic is the straight line distance to the goal; the actual cost to the goal may, however, be higher due to obstacles.

A consistent, also called monotonic heuristic is a heuristic that fulfills the requirement $h(x) \leq w(x, y) + h(y) \Leftrightarrow h(x) - h(y) \leq w(x, y)$. This means that we will never take a step back, because the cost of getting to y from x is greater (or equal to) the difference in the estimates. This implies that if x is the node being explored, we can never find a better $g(x)$ by going through y from x; we will always have a monotonically increasing total cost for the best path, as the search goes on. The straight line distance in euclidean space $\mathbb{R}^{\ltimes}$ is a consistent heuristic for the shortest path between two points, because moving in any direction will give a greater or equal total f-cost for the path. It is equal if there is no obstacles, and greater if there are obstacles, because then we are no longer moving in a straight line.

What heuristic to choose is highly dependent on the problem. For finding the optimal driving route, the euclidean distance to the goal is sufficient. This may also be true for the problem of finding a trajectory through a terrain, as is the case in this project. For solving the 8-puzzle, the sum of the Manhattan distances of each brick may be an acceptable heuristic. In general, a more accurate heuristic is more expensive to compute, but will also most likely be more efficient in terms of the number of nodes expanded.

2.1.2 Applications of shortest path algorithms

In general, shortest path algorithms solve the minimization problem of finding the path of least cost from a start node v_a to v_b. Applications for such algorithms are numerous. Examples are: finding the fastest driving route given a map, finding the best route for a network packet, find the optimal solution to a particular game (e.g. Rubik's cube), and, part of what this project is about, find the best path through a terrain for generating a road trajectory. All that is required, is to express the problem as a weighted graph, and also if using A*, design an admissible and consistent heuristic.

2.2 GPU computing

Although this project is not particularly about GPU computing, the snow simulator is real-time much because the heavy computation of simulating the wind field and updating the particle velocities and positions are implemented on the GPU. Because of that, the topic of GPU computing is still relevant for this project. This section will describe the capabilities of the GPU, its architecture, and logical layout when used for general

purpose computing. We will mainly focus on NVIDIA GPUs here because that is what has been used traditionally in the snow simulator, but the concepts are transferable to GPUs from other vendors.

2.2.1 Physical layout of a GPU

GPUs (Graphics Processing Units) are accelerators originally intended for highly efficient rendering of 3D graphics on a computer. Rendering is a highly parallel process, where each pixel can be rendered more or less independently from every other pixel. Because of this, GPUs are designed to be highly parallel devices with hundreds of simple vector cores where each vector core typically work on a single pixel. All vector cores may access a global memory with a capacity of hundreds of megabytes up to some gigabytes of memory, but there is also a faster cache available which is shared among groups of size 8 to 32 vector cores, depending on the architecture.

A GPU has hundreds of vector cores called *s*treaming processor cores (SPs). These cores are grouped into *s*treaming multiprocessors (SMs); each SM has between 8 to 32 SPs depending on the architecture; pre-Fermi architectures had 8, while Fermi-GPUs have 32 SPs per SM. Each of those SPs are a very simple processor, that may perform common floating point and integer operations. All of the SPs in a SM must perform the same instruction, but they may be performed on different data (SIMD, Single Instruction Multiple Data). In other words, each SM can be seen as a vector processor processing a vector of length 8 or 32 each.

Memory model

All vector cores may access a global memory, analogous to a regular computer's main memory. However, even though the bandwidth is huge (> 100GB/s for modern NVIDIA Tesla GPUs)[7], the latency is very large, 400-800 clock cycles.[8, p. 87] This necessitates faster on-chip memory.

NVIDIA GPUs have an on-chip memory is called *shared memory*. The difference between shared memory and a traditional cache is that shared memory must be managed manually; i.e. movement of words in and out of shared memory is done explicitly in code.

In addition, a texture cache is available, that in addition to caching, can perform interpolation and wrapping. The texture cache is read only.

In addition to shared memory and the texture cache, each thread running on the GPU has access to a very fast register memory. This is analogous to the CPU registers, in that it's used to store local variables, intermediate results and so on. A third type of fast memory is the constant memory. This is basically a cached read-only memory.

Figure 2.2 shows an overview of the memory model of a pre-Fermi GPU.

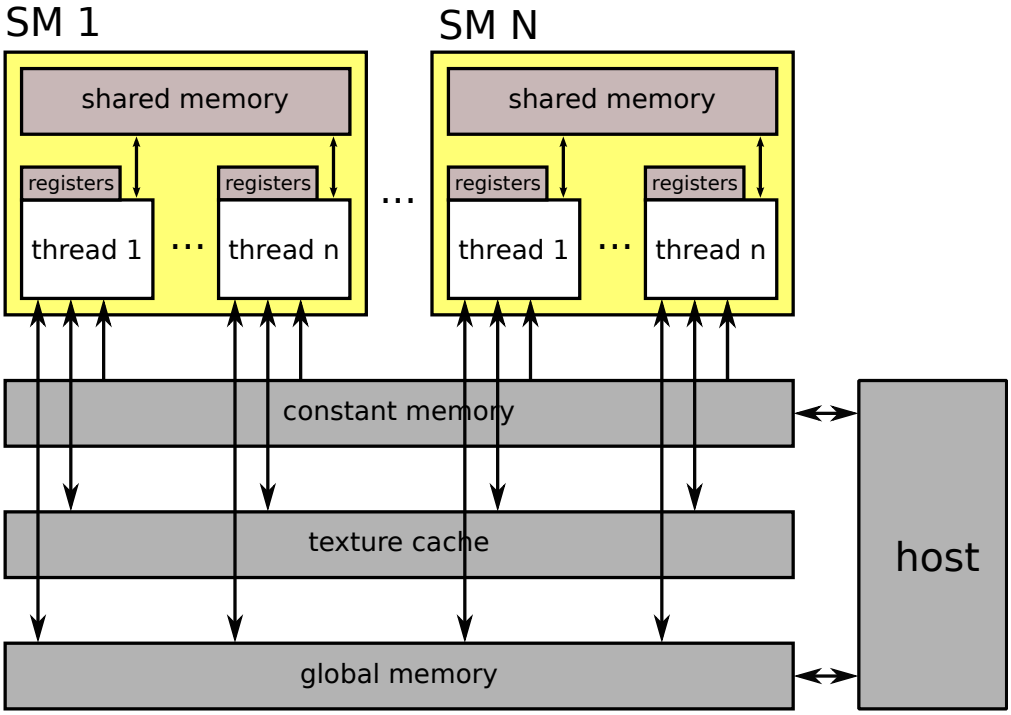


Figure 2.2: GPU memory model

2.2.2 General Purpose GPU Computing and CUDA

General purpose computing on GPUs are today an active topic of research. One reason why it's attractive is the enormous peak GFLOPS rate; Intel Core i7 980 XE, a high-end CPU at the time of writing this report, may perform at up to 109 GFLOPS[9] in double precision. As a comparison, a NVIDIA Tesla C2070, a high-end GPU specifically designed for general purpose computing, have a peak performance at 515.2 GFLOPS in double precision, a factor of 4.7 higher. In single precision it achieves 1.03 TFLOPS. However, despite the higher GFLOPS rate, GPUs are not well suited for solving all classes of problems.

GPUs get their power from large amount of data parallelism, as each SP in an SM must perform the same instruction at any given time; this means, if an application does not have any, or very little parallelism, GPUs are inferior to CPUs, which is built to be very fast on serial tasks. One can perform any task on the GPU but serial tasks is very slow because of lack of architectural features like pipelining, a slower clock rate, a smaller cache and a higher memory latency.

General purpose computing on GPUs is typically performed using a massive amount of very light weight threads with virtually no scheduling overhead; in other words, the GPU can context switch between threads at little to no cost. Each thread typically handles one or more data element. Threads are grouped into thread blocks, which is an 1D, 2D or 3D structuring of threads. Thread blocks are again grouped into a 1D or 2D grid. A thread block is executed on one SM, which means it may utilize the fast on-chip memory in an SM for thread communication and data sharing. A thread is executed on one SP. Figure 2.3 shows the logical thread organization in CUDA; this matches closely

to the physical layout on the GPU.

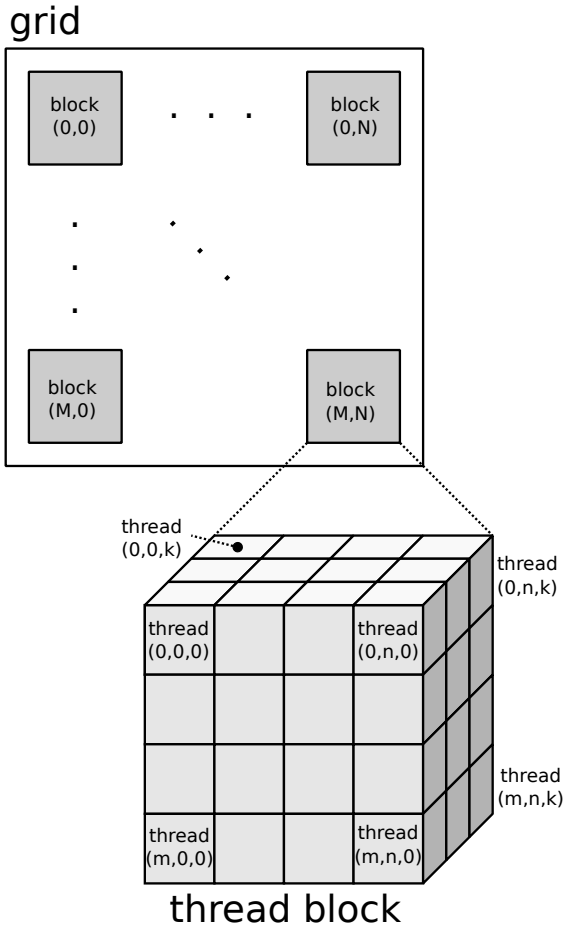


Figure 2.3: GPU thread model

A procedure that is executed on the GPU is called a kernel. A kernel is called from the host code that runs on the CPU; the host code also specifies the number of thread blocks to execute and how many threads to use in each thread block. The total number of threads executed is then grid size $\times$ block size.

CUDA is an extension to the C++ language developed by NVIDIA, for NVIDIA GPUs for general purpose computing. A CUDA program consists of host code and a device code; the host code is mostly regular C++: program initialization and memory allocation (for both host and device) goes in the host code. Device code is code that runs on the "device", i.e. the GPU in this case. Although the device code is syntactically C, the execution model is different, as the function is executed on an enormous number of threads at once. Each thread gets a thread ID, and each block gets a block ID; the thread ID and block ID together defines the global thread ID, i.e. the position of the thread in the grid. Typically, this global ID is directly mapped to one or more indices in the data array being processed.

Improvements with Fermi

In GPUs built on pre-Fermi architectures like NVIDIA Tesla C1060, the only fast on-chip memory available was constant memory, a read-only texture cache, and shared memory. Shared memory demands quite a bit of the programmer, as the programmer must recognize the locality properties of the memory accesses in each program. Both constant memory and the texture cache is limited both in size, and in that both are read only from a device kernel.

Fermi introduced a cache that resembles the traditional cache in a CPU. The cache is handled by the GPU, and is transparent to the programmer. However, the programmer may still want to utilize techniques like blocking for cache, but this is typically less verbose than explicitly reading and writing to shared memory.

In addition to introducing a cache, Fermi has increased double precision performance over previous architectures.[10, p. 9] In pre-Fermi architectures, like Tesla C1060, double precision peak performance was $1/8$th of peak single precision performance in terms of GFLOPS, while on Fermi it's up to about $1/2$ of peak single precision performance.[10, p. 9]

2.3 Newton's method

Newton's method is a method for solving equations of the form

$$f(x) = 0$$

for some function $f(x)$. It takes as input an initial guess for the value of x, x_0, and then computes

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.2}$$

This process is repeated until $|f(x_n)| < \epsilon$, where ϵ is the tolerance of error.

The way Newton's method works is that for every iteration, it draws a straight line from the current estimate of x, x_n, in the direction of the tangent of the curve at x_n, and computes the intersection with the line $y = 0$. x_{n+1} is then the value of x at that intersection point. Then the process is continued until convergence.

Newton's method typically has quadratic convergence[11]. However, note that Newton's method may not always converge, depending on the choice of the initial guess x_0, and the properties of the function. For instance, if a function has a derivative of zero, or if the derivative is otherwise badly behaved near the root, we may fail to find a root using this method. An example is the function $f(x) = \cos(x)$; an initial guess of 0 gives division by zero, and other guesses causes the next guess to overshoot due to the nature of the function and its derivative.

2.4 Clothoids and clothoid splines

Clothoids, or Euler spirals or Cornu spirals as they are also called, is a class of parametric curves that is often used in road modelling and planning due to their pleasant

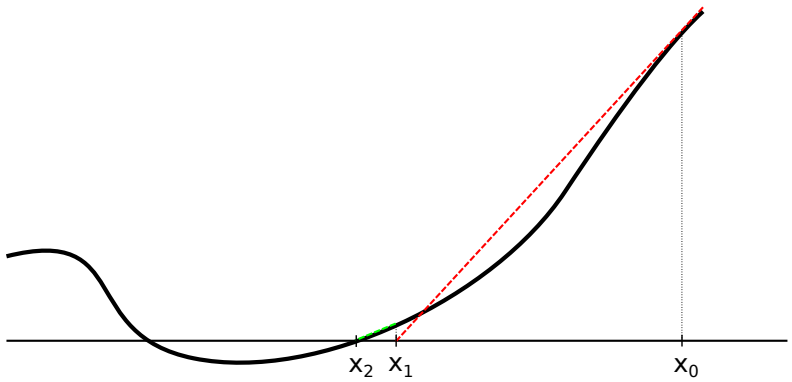


Figure 2.4: Newton's method. x_0 is initial guess, and it converges after two iterations to one of the roots x_2.

properties with regards to curvature. The curvature changes linearly with respect to the position along the curve, which gives smooth transitions into curves in the road. A clothoid is described by the parametric curve

$$x(t) = aC(t),\ y(t) = aS(t) \tag{2.3}$$

where a is a scaling factor, and $C(t)$ and $S(t)$ are the Fresnel integrals

$$C(t) = \int_0^t \cos\left(\frac{\pi}{2}s^2\right) ds \tag{2.4}$$

$$S(t) = \int_0^t \cos\left(\frac{\pi}{2}s^2\right) ds \tag{2.5}$$

and the curvature of a clothoid curve is given by

$$\kappa(t) = \frac{\pi t}{a},\ \kappa(\theta) = \frac{\sqrt{2\pi\theta}}{a} \tag{2.6}$$

A plot of a clothoid is shown in figure 2.5, plotted from $t = 0$ and upwards. Note that in many applications of clothoid curves, in particular in road trajectory computation, only the first part of the curve is used, as we will see later on.

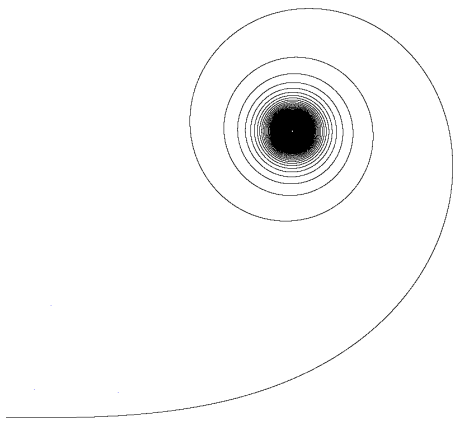


Figure 2.5: Plot of a clothoid for $t \in [0, \infty)$

2.4.1 The Fresnel integrals

A commonly used form of the Fresnel integrals is given by equations 2.4 and 2.5, where the parameter is the position along the curve t. Alternative forms of these integrals are given by using a point's tangent angle deviation θ from the beginning of the curve at $t = 0$ as a parameter instead of t, and is given as[12][13]

$$C_2(\theta) = \frac{1}{\sqrt{2\pi}} \int_0^\theta \frac{\cos(u)}{\sqrt{u}} du \tag{2.7}$$

$$S_2(\theta) = \frac{1}{\sqrt{2\pi}} \int_0^\theta \frac{\sin(u)}{\sqrt{u}} du \tag{2.8}$$

These two forms are equivalent, with[13]

$$C(t) = C_2\left(\frac{\pi}{2}t^2\right) \tag{2.9}$$

$$S(t) = S_2\left(\frac{\pi}{2}t^2\right) \tag{2.10}$$

The Fresnel integrals can be efficiently computed within an error of less than $\epsilon = 5 \cdot 10^{-10}$ without numerical integration, by using polynomial approximations of two functions $f(t)$ and $g(t)$, and then rewriting the integrals as[14]

$$C(t) = \frac{1}{2} + f(t)\sin\left(\frac{\pi}{2}t^2\right) - g(t)\cos\left(\frac{\pi}{2}t^2\right) \tag{2.11}$$

$$S(t) = \frac{1}{2} - f(t)\cos\left(\frac{\pi}{2}t^2\right) - g(t)\sin\left(\frac{\pi}{2}t^2\right) \tag{2.12}$$

where

$$f(t) = \sum_{n=0}^{11} f_n t^{-2n-1}, g(t) = \sum_{n=0}^{11} g_n t^{-2n-1} \tag{2.13}$$

where values for g_n and f_n is given in appendix A, as well as in [14]. This allows for efficient computation of the Fresnel integrals.

2.4.2 Notation

The notation used for this section is based on [12]. The points where clothoid pairs meet with zero curvature will be called *connection points* or *connection vertices*. A clothoid pair is made up by three points; connection points $\mathbf{P}_0$ and $\mathbf{P}_1$, and control vertex $\mathbf{V}$. The vertices are labeled such that $||\mathbf{P}_0 - \mathbf{V}||_2 \geq ||\mathbf{P}_1 - \mathbf{V}||_2$. The vector $\mathbf{V} - \mathbf{P}_0$ is called $\mathbf{g}$ and $\mathbf{P}_1 - \mathbf{V}$ is called $\mathbf{h}$; the lengths are referred to as g and h, respectively.

The clothoids starting at $\mathbf{P}_0$ and $\mathbf{P}_1$ are labelled A_0 and A_1 respectively; any straight line segment that has been appended before A_0 is referred to as S. The lengths of these curves are $|A_0|$, $|A_1|$ and $|S|$, respectively. The control vertices in a spline with n vertices are referred to as $\mathbf{v}_0, ..., \mathbf{v}_n$, and the connection points are referred to as $\mathbf{p}_0, ..., \mathbf{p}_{n-1}$. Note the difference between $\mathbf{p}_0$ as the first connection point in the spline, and $\mathbf{P}_0$ as the point local to a clothoid pair.

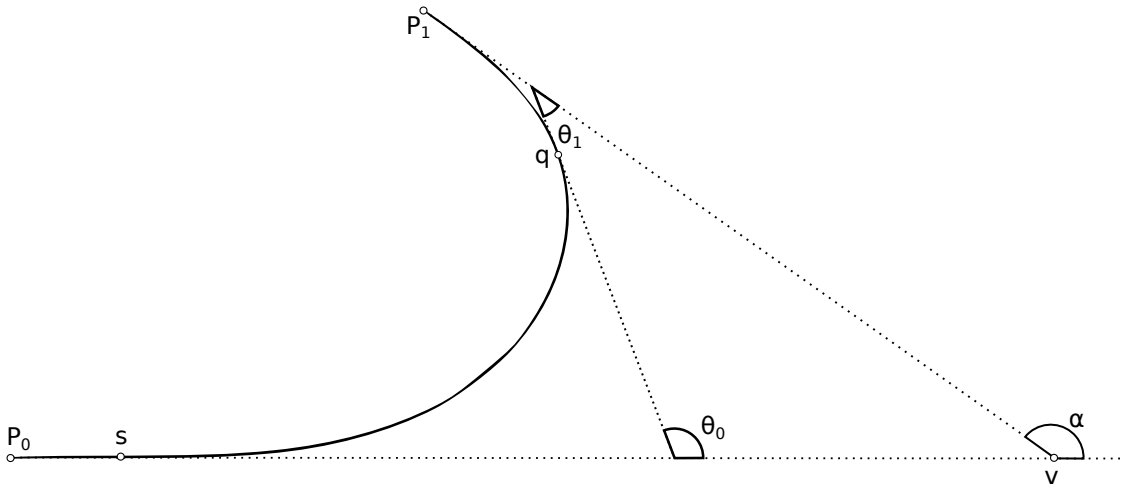


Figure 2.6: Clothoid pair that can be used as a spline segment. $\mathbf{q}$ marks where the clothoids are joined, and $\mathbf{s}$ marks where a straight line segment is appended.

2.4.3 Clothoid splines

Clothoids can be used to form splines, given a set of control points $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$.[12] One way of achieving this is first by noting that clothoids have a curvature in the beginning of the curve $\kappa(0) = 0$. Assume that we are given three points $\mathbf{V}$, $\mathbf{P}_0$ and $\mathbf{P}_1$, and the tangent vectors of $\mathbf{P}_0$ and $\mathbf{P}_1$ are $\mathbf{T}_0$ and $\mathbf{T}_1$, respectively. If we can find a pair of clothoids A_0 and A_1 that start at $\mathbf{P}_0$ and $\mathbf{P}_1$, respectively, in the direction of their tangents, so that they are joined at some point $\mathbf{P}$ with equal curvature, then we can construct a spline using three or more control points.[12]

The first step in constructing the spline is to define the vertices where the clothoid pairs meet with zero curvature. One way of choosing these is to use the first and last control vertex as the first and last vertex, and use the midpoint between two control vertices for all the middle points. This is a typical way of doing it, although there may be other ways of choosing the connection vertices. The tangents of each point is chosen so that the last tangent in a clothoid pair points in the direction of the control vertex $\mathbf{v}$ from $\mathbf{P}_1$, and the very first tangent is chosen in the direction of $\mathbf{v}$ from $\mathbf{p}_0$.

Finding clothoid pairs

Given the three points $\mathbf{V}$, $\mathbf{P}_0$ and $\mathbf{P}_1$ where the tangent vectors of $\mathbf{P}_0$ and $\mathbf{P}_1$ are $\mathbf{T}_0$ and $\mathbf{T}_1$, respectively, how can a pair of clothoids be found that satisfies the continuity of the curvature at the meeting point, and the initial direction given by the tangent vectors? Label $\mathbf{P}_0$ and $\mathbf{P}_1$ so that $||\mathbf{P}_0 - \mathbf{V}||_2 > ||\mathbf{P}_1 - \mathbf{V}||_2$, and call the longer edge g and the shorter one h. Let $k = g/h$ be the ratio of their lengths. Let α be the angle between these two edges, as shown in figure 2.6. Then, if

$$\frac{k + \cos\alpha}{\sin\alpha} < \frac{C_2(\alpha)}{S_2(\alpha)} \tag{2.14}$$

then such a pair of clothoids exist[12]. Finding this pair is then done by computing the tangent angle deviation θ for the meeting point for both clothoids, θ_0 and θ_1, or in other

words, find θ_0 and θ_1 such that $\theta_0 + \theta_1 = \alpha$ and $\kappa_{A_0}(\theta_0) = \kappa_{A_1}(\theta_1)$. This is done by solving the equation

$$\sqrt{\theta}[C(\theta)\sin(\alpha) - S(\theta)(k + \cos(\alpha)] + \sqrt{\alpha - \theta}[S(\alpha - \theta)(1 + k\cos(\alpha)) - kC(\alpha - \theta)\sin(\alpha)] = 0 \tag{2.15}$$

for θ_0[12]. For solving this equation, [12] suggests Newton's method or the bisection method. Finding θ_1 is then simply $\theta_1 = \alpha - \theta_0$.

Finally, we must compute the scaling factors for A_0 and A_1, that is, a_0 and a_1, respectively. Again referring to [12], this is done by solving the equation

$$a_0 C(\theta_0) + a_0\sqrt{\frac{\alpha - \theta_0}{\theta_0}}C(\alpha - \theta_0)\cos(\alpha) + a_0\sqrt{\frac{\alpha - \theta_0}{\theta_0}}S(\alpha - \theta_0)\sin(\alpha) = g + h\cos(\alpha) \tag{2.16}$$

for a_0. This is a simple rearrangement of terms since θ_0 and all other values are known at this point. To find a_1, we use the formula[12]

$$a_1 = a_0\sqrt{\frac{\theta_1}{\theta_0}} = a_0\sqrt{\frac{\alpha - \theta_0}{\theta_0}} \tag{2.17}$$

Now we have all we need to define the clothoid pair given by the three points: The scaling factors a_0 and a_1, and the tangent angle deviation of the curves at their endpoints θ_0 and θ_1.

Appending straight line segments

There may be a significant asymmetry between the lengths of $\mathbf{g}$ and $\mathbf{h}$ (i.e. g and h). Given a control vertex $\mathbf{V}$ and two points $\mathbf{P}_0$ and $\mathbf{P}_1$, $||\mathbf{P}_0 - \mathbf{V}||_2$ and $||\mathbf{P}_1 - \mathbf{V}||_2$ may be very different. It is easy to see that if this is the case, the inequality 2.14 may no longer hold and we may not be able to find a clothoid pair for these three points. Because the curvature at the joints between clothoid pairs are zero, the same as a straight line, this can be solved by simply "moving" $\mathbf{P}_0$, i.e. the point that is furthest away from $\mathbf{V}$, closer, and then append a straight line from the old position of $\mathbf{P}_0$ to the new position, so that the asymmetry is reduced. Moving it so that $g = h$ results in a symmetric pair of clothoids, i.e. one clothoid is simply a reflection of the other.[12]

To achieve this, assume that the point furthest away from $\mathbf{V}$, i.e. the point defining g, is labelled $\mathbf{P}_0$. Then we may define a symmetry parameter $\tau \in [0, 1)$ that determines how close to $\mathbf{V}$ we wish to move $\mathbf{P}_0$. $\tau = 0$ gives perfect symmetry, while any other value of τ gives different degrees of asymmetry, up to the limit determined by 2.14. The limit of the size of g is given by rearranging the inequality 2.14, and changing the inequality sign with an equality sign:

$$g_{lim} = h\left(\frac{C(\alpha)}{S(\alpha)}\sin(\alpha) - \cos(\alpha)\right) \tag{2.18}$$

Then, $\mathbf{P}_0$ is moved to

$$\mathbf{P}_0' = \mathbf{v} - ((1 - \tau)h + \tau g_{lim})\mathbf{T}_0 \tag{2.19}$$

and the region of the curve between $\mathbf{P}_0$ and $\mathbf{P}_0'$ is now described by a straight line between these two points.

2.5 Road generation

Roads can be generated in a number of ways; city street networks can be generated using using Lindenmayer systems[15], and shortest path algorithms like A* can be used to generate roads through a terrain, and this method may even be extended to generating hierarchical roads.[1][16] Generating realistic roads is more often than not non-trivial due to the many variables that determine if a road is "good" or not. For instance, different types of roads (e.g. highways, country roads) may have different tolerance to changes in elevation and curvature, and the trade-offs of going up a steep slope or making a longer, more curvy road around that slope must be weighted. In this section we will first look at how we can formulate the problem of generating a road through a terrain as a shortest-path problem as described by [1], and lastly we will discuss the details about the trajectory computation and road modelling given a discrete shortest path.

2.5.1 Formulating road generation as a shortest-path problem

Road generation can be formulated as a shortest-path problem in the following manner. First, discretize the terrain into a grid of elevation points. Each of these points is then a node in the graph. Conveniently enough, these points are also in $\mathbb{R}^3$ $(x, y, height)$, which makes designing heuristics for A* easier.

Next, we must define the edges of the graph. Those edges define, naturally, which nodes that can be reached from each node. The most naive ways of defining this neighborhood is either to choose ALL nodes to be reachable from every other node, or only the immediate eight neighbors. The first approach would be computationally infeasible due to the sheer amount of nodes in the graph. The other approach has a severe restriction on angles (the so-called limit on direction problem[1]), as the path finding algorithm can only take turns in multiples of 45 degrees, resulting in a fairly jagged path.

The solution to this problem of limit on direction, as presented in[1], is to not only use the immediate neighbors, but use a neighborhood mask to define the neighbors. This mask is defined as follows: The neighbors of a point $p_{a,b}$ is defined as all points $p_{a+i,b+j}$ where i, j are integers, $gcd(i, j) = 1$ and $i, j \in [-k, k]$. That $gcd(i, j) = 1$ requirement is not a requirement for correctness, but to avoid unnecessary cost function evaluations for overlapping paths. Figure 2.7 illustrates the neighborhood of a grid point with a mask size of $k = 2$. Figure 2.8 shows the effect this mask has on the shortest path. The gray box is an obstacle. We see that both the curvature and total length of the path is reduced.

In order to define the weights of each edge, we define the cost of building an infinitesimal portion of road from point $\mathbf{p}$, in the direction $\dot{\mathbf{p}}$ and rate of change of the direction $\ddot{\mathbf{p}}$ as $c(\mathbf{p}, \dot{\mathbf{p}}, \ddot{\mathbf{p}})$. Now, in order to evaluate the cost of a particular trajectory, simply integrate over the curve as shown in equation 2.20. Here, t_1 and t_2 define the segment end points of the parametric curve defined by the whole trajectory, as illustrated in figure 2.9. t_1 and t_2 correspond to the grid points $\mathbf{p}_{i_1,j_1}$ and $\mathbf{p}_{i_2,j_2}$, respectively.

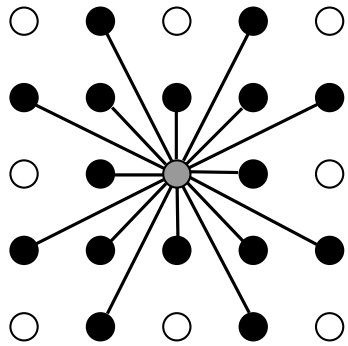
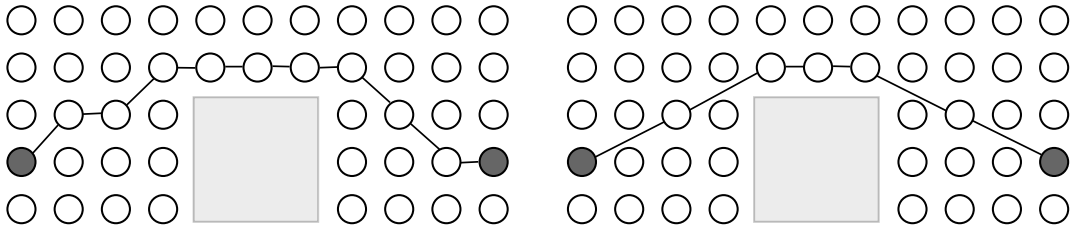


Figure 2.7: Neighborhood of grid point (middle).



(a) Only eight neighbors (b) Neighborhood mask with $k = 2$

Figure 2.8: Effect of neighborhood mask

$$cost(\mathbf{p}_{i_1,j_1}, \mathbf{p}_{i_2,j_2}) = \int_{t_1}^{t_2} c(\mathbf{p}(t), \dot{\mathbf{p}}(t), \ddot{\mathbf{p}})(t)dt \tag{2.20}$$

Evaluating this integral can be done numerically by discretizing the curve into intervals of length Δt and using some numerical integration method like the midpoint rule or Simpson's rule.

2.5.2 Cost functions

There are many potential contributing factors to the cost of a trajectory. One of them is of course road length, where the cost typically increases linearly with the length. In order to get realistic roads, however, using only road length as a cost function is not sufficient, because that would simply generate a nearly straight line from start to goal, which is suboptimal in e.g. a mountainous terrain. Adding an extra cost for slopes, or change in elevation, will cause the road to have curves around steep mountain sides, and roads will generally avoid bumpy terrain. However, this may result in sharp turns because curvature is not taken into account when attempting to avoid elevation changes. This may be alleviated by also taking curvature into account.

Let each of the cost functions, for road length, elevation changes and curvature, and possibly more, be numbered $1, ..., n$ where n is the number of cost functions. Also, let μ_i be a *t*ransfer function that maps the measured values of slope, curvature and so on, to the actual cost value. It may be helpful to think of μ_i as a weighting function,

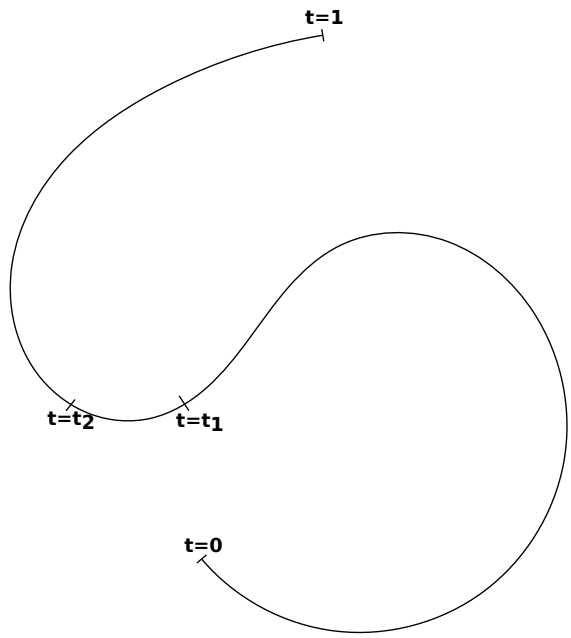


Figure 2.9: Trajectory segment

although it may also be nonlinear. Then, the cost function can be defined as

$$C(\mathbf{p}, \dot{\mathbf{p}}, \ddot{\mathbf{p}}) = \sum_{i=1}^{n} \mu_i(c_i(\mathbf{p}, \dot{\mathbf{p}}, \ddot{\mathbf{p}}))$$

2.5.3 Trajectory computation

A piecewise linear curve, like the one given by the discrete shortest path A* algorithm described above, is not a good description of the road trajectory, as this gives very sharp turns. Instead, the road trajectory can be modelled as a clothoid spline[1], which is described in section 2.4. This gives a pleasant curvature which varies linearly with the position along the curve[12]. Given the discrete shortest path represented by the points $\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_n$, a clothoid spline can be constructed using these points as control points. The spline starts and ends at the end points ($\mathbf{p}_0$ and $\mathbf{p}_n$).

2.6 The Snow Simulator

A major part of this project is to extending the snow simulator developed at the HPC lab at NTNU, to support loading meshes (mainly roads, but may be arbitrary meshes). In order to fully integrate the contributions from this project into the snow simulator, it is important to have some background on the way the snow simulator works. Figure 2.10 shows the snow simulator in action. This section will first describe the terrain format. Then, wind simulation and particle updates are described briefly.

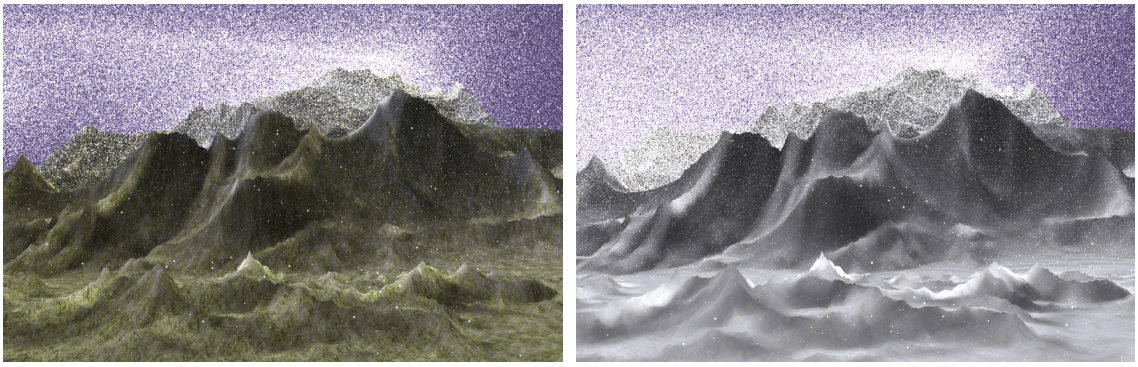


Figure 2.10: A scene before and after being covered with snow

2.6.1 Terrain

Terrains are in many applications represented by a height map, which is nothing more than a bitmap of a certain width and height, where each element represent a height in the terrain. In the snow simulator, a 16 bit integer height map is used; this means, each height value is an integer of 16 bits, which is then mapped to an actual height value in meters in the snow simulator.

The height map is stored as raw data, with no header, in row-major order; that is, the elements within a row is stored sequentially in the file, so row 1 is stored first, followed by row 2, etc. The map is loaded by first reading all the values of the file, then generating vertices for a triangle mesh that will be used as rendering. Figure 2.11 shows a visualization of a height map.

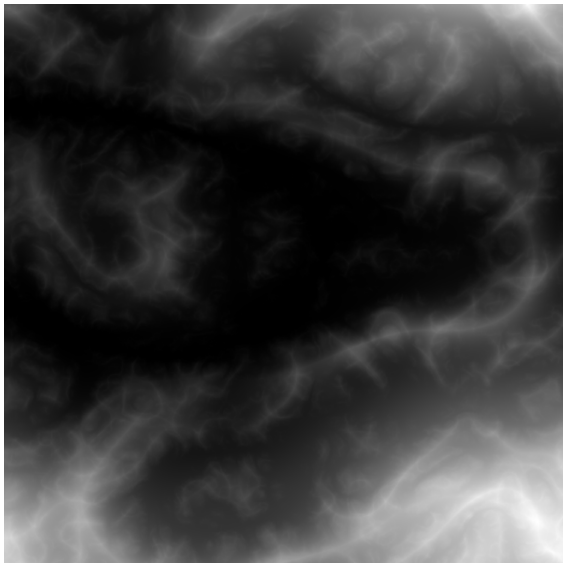


Figure 2.11: A height map of a terrain

In the original snow simulator the integers are mapped to a floating point value between 0 and 64. This means that a value of zero represents an vertex y-position $y = 0$, and a value of $2^{16} - 1$ is mapped to $y = 64$. For this project, for the terrains to have

the correct proportions when loaded from real-world data, and also for the roads to be laid out correctly in the terrain, the 16 bit height values are in principle mapped to an actual height value between h_{min} to h_{max}, where h_{min} and h_{max} is the lowest and highest points in the terrain, respectively. In practice though, it is convenient to have the lower point at $y = 0$ so that the simulation box does not have to be translated to cover the terrain. So instead, the height values are mapped to a value between 0 and $h_{max} - h_{min}$.

2.6.2 Wind simulation

Wind in the snow simulator is modelled by the incompressible Euler equations, which is derived from the Navier-Stokes equations by setting the viscosity to zero and density to one,

$$\nabla \cdot \mathbf{u} = 0 \tag{2.21}$$

$$\frac{\delta \mathbf{u}}{\delta t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla \mathbf{p} \tag{2.22}$$

where $\mathbf{p}$ is the pressure, and $\mathbf{u}$ is the velocity vector field.

This equation is simulated numerically by first solving for advection (movement of the fluid along its own motion), then solving for pressure which involves solving the Poisson equation $\nabla^2 p = \nabla \cdot \mathbf{u}^*$ (where $\mathbf{u}^*$ is a temporary velocity field generated from the advection step), and finally projecting these velocities into a divergence-free (mass conserving) field[17].

2.6.3 Particle updates

Snow particles are updated according to their current velocity, wind velocity at their position and gravity.[17] First, the wind velocity is interpolated from the wind field by a texture lookup in the GPU. Then, a circular velocity of the snow particle is computed, which causes them to rotate around a central axis. Then the acceleration is computed from the drag (a function of the difference between the velocities of the wind and snow particles), and gravity. Each particle is updated by exactly one thread on the GPU.

2.7 USGS DEM Format

USGS DEM is a file format for storing elevation data. It is a format in human readable ASCII characters, and the files are segmented into blocks of 1024 bytes. There are three types of records in a DEM file: one A-record, which is the header; multiple B-records, where each B-record store one column of elevation data and one C-record, which stores statistics of errors and accuracy of the data.[18]

The data in a DEM is defined for a quadrangle; i.e. an arbitrary polygon of four vertices. The quadrangle may or may not be an axis-aligned rectangle, but most likely it will be close to a rectangle. Oftentimes, the quadrangle attempts to follow the curvature of the earth mapped onto a plane, which gives slightly rotated and skewed rectangles.

Next, I will give an overview of the A and B records of a DEM. The C-record is not relevant for this project.

2.7.1 A-record overview

The A-record is the header of the DEM file, containing data such as the vertices of the quadrangle in world coordinates, total number of columns of data, minimum and maximum elevation in the DEM, units used for the quantities in the file (feet, meters, arc seconds), and so on. It also stores information about resolution in all three dimensions.

2.7.2 B-record overview

Each B-record stores one column of data for the height map along with a header for this data. However, since the quadrangle stored in a DEM file may not be quadratic, each column may have different number of elements. The position of the bottom-most point is specified in the header in world coordinates, along with the number of points. One B-block may span more than one 1024-byte block.

Each elevation data point in the B-record is stored as an integer, and may be converted to an actual height value h_i (in meters) by using the following formula for data point y_i:

$$h_i = ((e_{max} - e_{min}) \cdot y_i + d_y) \cdot res_y \cdot unit$$

where d_y is the height datum and e_{max} and e_{min} is the maximum and minimum elevation, respectively. $unit$ is the number of meters per unit specified in the A-record; $unit = 1$ for meters, or 0.3048 for feet.

Details about the layout of the USGS DEM file format can be found in the specifications[18].

2.8 RoadXML

The RoadXML format is an XML based file format used to describe road networks. In addition to describing the trajectories, it also describes the profile of the road (i.e. width, markings, number of lanes, ...) and textures used for rendering.[3] RoadXML is an open format, developed in collaboration with the driving simulation industry and universities. Although RoadXML is a very powerful and extensive format used to describe complex systems with intersections, traffic information, grip, and much more, it is in this project only used as a road trajectory descriptor suitable for generating the road model.

Chapter 3

Implementation

This chapter will describe my implementation of the procedural road generation algorithm, how the roads produced here is integrated with the snow simulator with all the changes to the snow simulator this entails, and lastly it describes the details about my implementation of the USGS DEM to height map converter application.

3.1 Procedural road generation

For this project, I implemented the road generation algorithm described by [1] for generating a road through a terrain described by a discrete height map. The road generator is a separate program written in C++ implementing an A* search algorithm which takes in a terrain in form of a height map, and outputs a file in the RoadXML format (see [3]). This section will first present the particular A* implementation, then it will describe how the clothoid spline is generated. Lastly, the RoadXML export code is described.

3.1.1 Implementation of A*

The A* search used for generating roads in this project first generates a discrete shortest path, described by n points in space. When computing costs, it is assumed that these points form a piecewise linear curve where each point along the curve $(x(t), y(t))$ is a function of parameter t, which is the length of the curve. Eventually, a clothoid curve is generated from the same points, but this is not used for path finding.

The A* algorithm is implemented as described by algorithm 2, with the edge weight w being equal to the cost function described in the section "Cost functions". The following section will also describe the data structures used, and why they were chosen.

Choice of data structures

The time complexity of A* is highly dependent on the data structures used in the implementation. In particular, the data structure used for the $OPEN$ and $CLOSED$ lists

must be chosen carefully. In this implementation, a minimum heap with a supplementary hash map is used for the $OPEN$ list, and a hash map is used for the $CLOSED$ list.

An A* search often has a very large amount of insertion operations to the $OPEN$ set, as it is used in algorithm 2. In addition, every iteration the node with the lowest f-cost must be retrieved from this set. Using an unsorted list gives a constant insertion time, but a really bad lookup-time when searching for the node with the lowest f-cost of $O(n)$. In the other extreme, a sorted list gives constant lookup-time, but insertion takes $O(n)$ time.

Instead, typically a min-heap is used. A min-heap has a constant lookup-time of the "smallest" element (smallest in this case being the node with the smallest f-cost) because the next element to be extracted is always on top, but in addition the element must be removed, which takes $O(lg(n))$ time. Insertion takes in theory $O(lg(n))$ time, but due to the structure of the heap, in practice this cost is constant on average.

However, the elements in a min-heap generally have no specified order other than the smallest item being on top, and the second smallest item being one of its children. Searching for an arbitrary item in a min-heap therefore requires us to search through the whole heap. Because this is something that is done regularly, to check for existence of an item in the $OPEN$ list (see the inner for-loop in algorithm 2), a supplementary hash map is used to keep track of which nodes exists in the heap.

A hash map has constant lookup and constant insertion time, although the items are in general unordered. Therefore, the $CLOSED$ list is implemented as a hash map, because all that list is used for, is inserting nodes, and looking up existence of nodes. This is also the reason why a hash map is well suited for augmenting the heap for the $OPEN$ list, to keep track of existence of nodes in the heap.

Cost functions

Three separate cost functions were implemented based on slope, curvature and road length. Given two points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, the cost to go from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$ is defined as

$$C(\mathbf{p}_i \to \mathbf{p}_{i+1}) = c_{length}(\mathbf{p}_i, \mathbf{p}_{i+1}) + c_{slope}(\mathbf{p}_i, \mathbf{p}_{i+1}) + c_{curvature}(\mathbf{p}_{i-1}, \mathbf{p}_i, \mathbf{p}_{i+1}) \tag{3.1}$$

These cost functions attempt to approximate the actual cost of the road on a piecewise linear curve formed by the points defining the discrete shortest path.

The cost for road length for the path between two points is given as

$$c_{length}(\mathbf{p}_i, \mathbf{p}_{i+1}) = w_{road}||\mathbf{p}_{i+1} - \mathbf{p}_i|| \tag{3.2}$$

where w_{road} is the weight for road length; this weight is typically set to one in this project.

Let $z(\mathbf{p})$ be the elevation in the terrain at some point $\mathbf{p}$. For the slope cost, we need to numerically solve the integral

$$c_{slope}(\mathbf{p}_i, \mathbf{p}_{i+1}) = w_{slope} \int_0^1 |\dot{z}((1-t)\mathbf{p}_i + t\mathbf{p}_{i+1})| \, dt$$

where $\dot{z}(\mathbf{p})$ is the slope of the terrain along the curve at $\mathbf{p}$. This can be approximated by using the midpoint rule, with a discrete summation as

$$\begin{aligned} c_{slope}(\mathbf{p}_i, \mathbf{p}_{i+1}) \approx & w_{slope} h \sum_{j=0}^{N-1} \frac{|z((1-t_{j+1})\mathbf{p}_i + t_{j+1}\mathbf{p}_{i+1}) - z((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1})|}{h} \\ = & w_{slope} \sum_{i=0}^{N-1} |z((1-t_{j+1})\mathbf{p}_i + t_{j+1}\mathbf{p}_{i+1}) - z((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1})| \end{aligned} \tag{3.3}$$

where h is the step length used in the approximation, and $t_i = ih$, and $N = 1/h$. The smaller h is, the better an approximation to the actual integral is achieved. h should be chosen so that at least every terrain height point along the path is sampled; i.e. $h \leq$ (resolution)$/||\mathbf{p}_{i+1} - \mathbf{p}_i||$. $(z((1-t_{j+1})\mathbf{p}_i + t_{j+1}\mathbf{p}_{i+1}) - z((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1}))/h$ is the forward difference approximation to the slope $\dot{z}((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1})$.

Lastly, curvature at a point $\mathbf{p}_i$ can be estimated as[19]

$$\kappa(\mathbf{p}_i) \approx \frac{2\sin\left(\frac{\theta}{2}\right)}{\sqrt{||\mathbf{v}_1|| \cdot ||\mathbf{v}_2||}} \tag{3.4}$$

where $\mathbf{v}_1 = \mathbf{p}_i - \mathbf{p}_{i-1}$ and $\mathbf{v}_2 = \mathbf{p}_{i+1} - \mathbf{p}_i$; i.e. $\mathbf{v}_1$ and $\mathbf{v}_2$ is the vectors between the neighboring points, and θ is the angle between these points, i.e.

$$\theta = \cos^{-1}\left(\frac{\mathbf{v}_1}{||\mathbf{v}_1||} \cdot \frac{\mathbf{v}_2}{||\mathbf{v}_2||}\right)$$

This is made into a cost function by applying the curvature weight to it:

$$c_{curvature} = w_{curvature}\kappa(\mathbf{p}_i) \tag{3.5}$$

Now using the total cost function given in equation 3.1, and substituting each cost for the results from equations 3.2, 3.3, 3.4 and 3.5, we get the final cost function

$$\begin{aligned} C(\mathbf{p}_i \to \mathbf{p}_{i+1}) = & w_{road}||\mathbf{p}_{i+1} - \mathbf{p}_i|| \\ & + w_{slope} \sum_{i=0}^{N-1} |z((1-t_{j+1})\mathbf{p}_i + t_{j+1}\mathbf{p}_{i+1}) - z((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1})| \\ & + w_{curvature} \frac{2\sin(\frac{\theta}{2})}{\sqrt{||\mathbf{v}_1|| \cdot ||\mathbf{v}_2||}} \end{aligned} \tag{3.6}$$

Choice of heuristic function

The choice of the heuristic function can greatly influence the time it takes to find a path from the start node to the goal node. The more accurate the heuristic is, the fewer nodes are expanded in the search; however, of course, as mentioned in section 2.1.1, more accurate heuristics are typically more expensive to compute.

In the road generator, assuming zero curvature, we can find a lower bound for the cost between two points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ using the cost due to the distance between the

points plus the cost due to the elevation difference:

$$\begin{aligned} C(\mathbf{p}_i \to \mathbf{p}_{i+1}) =& c_{length}(\mathbf{p}_i, \mathbf{p}_{i+1}) + c_{slope}(\mathbf{p}_i, \mathbf{p}_{i+1}) + c_{curvature}(\mathbf{p}_{i-1}, \mathbf{p}_i, \mathbf{p}_{i+1}) \\ \geq& c_{length}(\mathbf{p}_i, \mathbf{p}_{i+1}) + c_{slope}(\mathbf{p}_i, \mathbf{p}_{i+1}) \\ =& w_{road}||\mathbf{p}_{i+1} - \mathbf{p}_i|| + w_{slope} \sum_{i=0}^{N-1} |z(t_{i+1}) - z(t_i)| \end{aligned}$$

Because $|a + b| \leq |a| + |b|$, it follows that

$$\sum_{i=0}^{N-1} |z(t_{i+1}) - z(t_i)| \geq \left| \sum_{i=0}^{N-1} (z(t_{i+1}) - z(t_i)) \right| = |z(t_N) - z(t_0)| = |z(T_2) - z(T_1)|$$

This gives a lower bound for the cost of moving from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$:

$$C(\mathbf{p}_i \to \mathbf{p}_{i+1}) \geq w_{road}||\mathbf{p}_{i+1} - \mathbf{p}_i|| + w_{slope}|z(T_2) - z(T_1)| \tag{3.7}$$

Substituting $\mathbf{p}_{i+1}$ for the goal node $\mathbf{v}_b$ in equation 3.7 gives the heuristic used in this project, i.e.

$$h(\mathbf{p}_i) = w_{road}||\mathbf{v}_b - \mathbf{p}_i|| + w_{slope}|z(\mathbf{v}_b) - z(\mathbf{p}_i)| \tag{3.8}$$

where $z(\mathbf{p})$ is the elevation at point $\mathbf{p}$.

In order to guarantee that the path that is found is optimal, the heuristic used must be consistent and admissible. h as defined above is definitely admissible, because it is a lower bound of the cost, as proved above. It is also consistent, because moving from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$ gives a change in h that is less or equal to the actual cost for moving from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$, and thus the total cost $f(x) = g(x) + h(x)$ is always increasing. Using h in $A*$ therefore results in an optimal path.

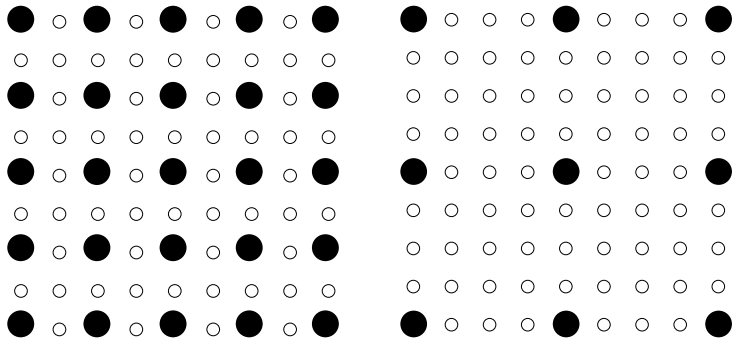
Grid coarsening

Typically in an A* search every grid point is used as a node in the graph. For huge maps, say 4096×4096, this approach becomes very slow due to the complex cost function used for procedural road generation. A more complex cost function makes it harder to make a heuristic that is both efficient and accurate for the distance to the goal, which causes A* to expand more nodes, and thus scale poorly with larger problems. A solution to this is to coarsen the grid used, so that instead of using every elevation point in the height map as a node, we use every m nodes, as shown in figure 3.1. This significantly reduces the number of nodes, making it viable to create paths through larger terrains.

Neighborhood mask creation

A typical A* implementation for use with a grid like the discretization of a terrain often use eight neighbors. [1] propose that not only the immediate eight neighbors are used per node, but instead use a larger neighborhood as described in section 2.5.1, and as shown in figure 2.7.

The naive implementation of utilizing this would be to for each node loop over each neighboring node within a distance k, check if $gcd(i, j) = 1$, and if so, compute the



(a) $m = 2$ (b) $m = 4$

Figure 3.1: Grid coarsening with grid points every 2 and 4 elevation point

cost to that node and add it to the $OPEN$ list. This does a lot of unnecessary checks as many of those neighbors will not be selected for expansion because $gcd(i, j) \neq 1$. Instead, the neighborhood mask are precomputed before the search starts and put in an array. This allows us to instead loop over the array, with exactly as many elements as there is neighbors that will actually be expanded. The pseudo code for the generation of neighborhood masks are given in algorithm 3.

Algorithm 3 Pseudo code for constructing the neighborhood mask

```
Let M be the neighborhood mask, initially empty.
for i := 0 → k do
  for j := 0 → k do
    if gcd(i, j) ≠ 1 then
      continue
    end if
    M := M + (i, j)
    if i ≠ 0 then
      M := M + (−i, j)
    else if j ≠ 0 then
      M := M + (i, −j)
    else if i ≠ 0 ∧ j ≠ 0 then
      M := M + (−i, −j)
    end if
  end for
end for
```

3.1.2 Trajectory generation

A clothoid spline can be used to model the road trajectory, as this gives smooth curves with linearly varying curvature. The method described in [12] is used (see section 2.4). A class `ClothoidPair` was written to take care of constructing each clothoid pair in the spline, while the class `ClothoidSpline` contains one or more clothoid pairs that makes up the spline.

Constructing the clothoid spline

After the control points have been generated by the A* search as described above, these are passed on to the ClothoidSpline constructor. This constructor will first determine the connection points $\mathbf{p}_0, ..., \mathbf{p}_n$, i.e. where the clothoid pairs begin and end; this is done as described in section 2.4.3. The end points are used as connection points as well. These points are then removed from the control vertex list. The tangents are then determined as

$$\mathbf{T}_i = \begin{cases} (\mathbf{v}_i - \mathbf{p}_i)/||\mathbf{v}_i - \mathbf{p}_i||_2 & \text{if } i = 0 \\ (\mathbf{p}_i - \mathbf{v}_{i-1})/||\mathbf{p}_i - \mathbf{v}_{i-1}||_2 & \text{if } i > 0 \end{cases} \tag{3.9}$$

The next step is to construct a clothoid pair that begins and ends at the connecting vertices for each control vertex, such that the tangent of the clothoid at the endpoints are the same as the tangent for the endpoints themselves.

If the control points and its corresponding connection points are co-linear, we simply use a straight line between the points. Otherwise, note that the starting point of clothoids have a tangent of $\mathbf{T} = (1, 0)$ and position $(0, 0)$; this means that the clothoids must be translated and rotated rotated to coincide with the points and tangents at the end points. Note also that we do not need to rotate and translate the points themselves, only the final clothoids.

In each clothoid pair, one clothoid must be flipped; this is because the clothoids have different signs on the derivative of their curvature; i.e. $Sgn(\frac{d\kappa_0}{dt}) \neq Sgn(\frac{d\kappa_1}{dt})$. To test for which clothoid needs flipping, it is sufficient to check the sign of the third component in the cross product $\mathbf{g} \times \mathbf{h}$ where $\mathbf{g}$ and $\mathbf{h}$ are the vectors representing the longest and shortest vectors, respectively, of $\mathbf{V} - \mathbf{P}_0$ and $\mathbf{P}_1 - \mathbf{V}$. If the sign is negative, the clothoid starting in $\mathbf{P}_0$ must be flipped; otherwise, the other clothoid must be flipped.

Finally, the scaling factors a_0 and a_1, and the tangent angle deviation for the first clothoid at the meeting point θ_0 are computed. Note that θ_1 is computed using $\theta_1 = \alpha - \theta_0$ because $\theta_0 + \theta_1 = \alpha$[12]. θ_0 is found by using Newton's method described in 2.3 to solve equation 2.15 in section 2.4.3. An initial guess of $x = \alpha/2$ is used, which is reasonable because θ_0 is somewhere between 0 and α, and is exactly $\alpha/2$ for symmetric clothoid pairs.[12]. The scaling factors is found by solving equation 2.16 for a_0, also in section 2.4.3. a_1 is found using the formula given in equation 2.17. The complete pseudo code for constructing the clothoid spline is seen in algorithm 4.

Generating Cartesian points from the spline

Given a clothoid spline constructed using the methods above, we need to be able to look up the position in Cartesian coordinates for each value of the parameter t of the curve. This allows us to render the spline, by starting at $t = 0$, ending at $t = t_{max}$ and incrementing t with sufficiently small increments. It also allows us to look up height values at every point of the spline, which is important when generating a RoadXML file, as described in section 3.1.3.

Algorithm 4 Algorithm for constructing a clothoid spline

Require: Control points $W = \{\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_n\}$
Require: Connection points $Q = \{\mathbf{v}_0, (\mathbf{v}_1 + \mathbf{v}_2)/2, (\mathbf{v}_2 + \mathbf{v}_3)/2, ..., (\mathbf{v}_{n-2} + \mathbf{v}_{n-1})/2, \mathbf{v}_n\}$
Require: Tangents T_i according to equation 3.9
  for $i := 1 \rightarrow n - 1$ **do**
    $C :=$ new clothoid pair
    $\mathbf{P}_0 := \arg\max_{\mathbf{p}}\{||\mathbf{v}_i - \mathbf{p}_{i-1}||_2, ||\mathbf{v}_i - \mathbf{p}_i||_2\}$
    $\mathbf{P}_1 := \arg\min_{\mathbf{p}}\{||\mathbf{v}_i - \mathbf{p}_{i-1}||_2, ||\mathbf{v}_i - \mathbf{p}_i||_2\}$
    if $\mathbf{P}_0, \mathbf{v}_i, \mathbf{P}_1$ is co-linear **then**
      $C.length := C.|S| := ||\mathbf{P}_1 - \mathbf{P}_0||_2$
      $C.t_0 := C.t_1 := 0$, $C.a_0 := C.a_1 := 1$
    end if
    $\mathbf{g} := \mathbf{v}_i - \mathbf{P}_0$, $\mathbf{h} := \mathbf{P}_1 - \mathbf{v}_i$, $g = ||\mathbf{g}||_2$, $h = ||\mathbf{h}||_2$
    $\alpha := \cos^{-1}(\mathbf{g}/g \cdot \mathbf{h}/h)$
    $g_{lim} := h(C_2(\alpha)/S_2(\alpha) * \sin(\alpha) - \cos(\alpha))$
    if $g > \tau * g_{lim} + h(1 - \tau)$ **then**
      $C.|S| := g - (\tau * g_{lim} + h(1 - \tau))$
      $g := g - C.|S|$
    else
      $C.|S| := 0$
    end if
    $k := g/h$
    $C.flipA0 := (\mathbf{g} \times \mathbf{h}).z < 0$
    $\alpha_0 := atan2(\mathbf{T}_0.y, \mathbf{T}_0.x)$
    $\alpha_1 := atan2(\mathbf{T}_1.y, \mathbf{T}_1.x)$
    $\{\alpha_i$ is the rotation angle to align T_i to $(1, 0)\}$
    $\theta_0 :=$ root of equation 2.15 found with Newton's method, $0 < \theta_0 < \alpha$
    $\theta_1 := \alpha - \theta_0$
    $C.a_0 :=$ solution to equation 2.16, solved for a_0
    $C.a_1 := a_0\sqrt{(\theta_1/\theta_0)}$
    $C.t_0 := \sqrt{(2\theta_0/\pi)}$
    $C.t_1 := \sqrt{(2\theta_1/\pi)}$
    Add C to the list of clothoid pairs
  end for

Looking up a point in the spline is done in three steps. First, find the clothoid pair that corresponds to the parameter t as given. Then, it must be determined which clothoid or straight line segment of the clothoid pair corresponds to t. Finally, the point is computed by evaluating the Fresnel integrals, and then rotate, flip and translate the point to the correct position.

Finding the clothoid pair corresponding to t is done using a binary search into a precomputed `lengths` array. This array contains the cumulative lengths of the curve before each clothoid pair has been added. As an example, say a spline has three clothoid pairs with the lengths 140, 90 and 125. The `lengths` array then contains the values 0, 140, 230 and 355, in that order. The index of the largest length $l \in lengths$ such that $l < t$ then is the index of the clothoid pair corresponding to t. In general, given n clothoid

pairs $C_1, ..., C_n$, the length entries is given as

$$l_i = \begin{cases} 0 & \text{if } i = 0 \\ l_i - 1 + |C_i| & \text{otherwise} \end{cases}$$

where $|C_i|$ is the length of C_i.

A clothoid pair consists of two clothoids A_0 and A_1, and possibly a straight line segment S. S connects A_0 to P_0, and A_1 connects A_0 to P_1. Figuring out which part of the clothoid pair t corresponds to is more tricky, because it is unclear which clothoid comes first in the curve of A_0 and A_1, as A_0 always starts at $\mathbf{P}_0$, i.e. the endpoint of the longest segment. This information is recorded when the clothoid pair is constructed in form of a reversed-flag.

We construct a `local_lengths` table with the cumulative lengths of all the parts of the clothoid pair (S, A_0 and A_1). If the clothoid pair is not reversed, that is, A_0 comes first in the curve, we get the cumulative lengths 0, $|S|$, $S + |A_0|$ and $S + |A_0| + |A_1|$ in this order, where $|X|$ means the length of $|X|$. If the clothoid pair IS reversed, we instead get 0, $|A_1|$, $|A_1| + |A_0|$ and $|A_1| + |A_0| + |S|$. Using this, we can determine which clothoid/straight line segment corresponds to t, by finding the "local" $t_l = t - l_i$ and using this the same way as we found the correct clothoid pair (however, a binary search is not necessary here).

The last step is to compute the position from the clothoid or straight line we found in the previous step. The parameter t_l must be transformed into t_{ll} local to the clothoid or straight line. How this is done, depends on which segment of the clothoid pair we are in, and whether or not the clothoid pair is reversed. Table 3.1 shows all possibilities.

Part of curve	**Reversed**	$\mathbf{t}_{ll}$
S	No	t_l
S	Yes	$\|S\| - (t_l - \|A_1\| - \|A_0\|)$
A_0	No	$(t_l - \|S\|)/a_0$
A_0	Yes	$t_0 - (t_l - \|S\|)/a_0$
A_1	No	$(t_l - \|S\| - \|A_0\|)/a_1$
A_1	Yes	$t_1 - t_l/a_1$

Table 3.1: Transformation of t_l into a parameter t_{ll} local to the segment

3.1.3 RoadXML generation

After the discrete shortest path has been found, represented by the points $\mathbf{v}_0, ..., \mathbf{v}_n$, a RoadXML file can be created. A RoadXML file contains information about road profiles, ground materials, the road trajectory, banking curves and elevations, and more.[3] For this project, only the road trajectory and elevation curves are defined in the RoadXML file; the rest is given in a static header, which defines the road profile (number of lanes, width of the road, ...).

The road trajectory is defined in the XYCurve tag[3]. A curve of type Clothoid-Spline is defined, with the control points found from the discrete shortest path algorithm. The control points are specified with the Vectord2 tag, from the first to last control

point, as shown in figure 3.2. The x and y values of each Vectord2 element is *relative* to the x and y values specified in the XYCurve tag, which is the starting point of the curve. The direction of the XYCurve specifies rotation; in this project, direction is always zero.

```
<XYCurve direction="0" x="1480" y="10000">
  <ClothoidSpline type="spline">
    <Vectord2 x="200" y="-80" />
    <Vectord2 x="400" y="-160" />
    <Vectord2 x="560" y="-200" />
    <Vectord2 x="680" y="-240" />
    <Vectord2 x="880" y="-320" />
    (...)
  </ClothoidSpline>
</XYCurve>
```

Figure 3.2: Example of a trajectory definition in the RoadXML file

Next, the elevation curves are defined. In RoadXML, these are referred to as SZ curves; S refers to the position along the curve, and Z refers to the elevation at that point. SZ curves in RoadXML actually defines a series of third-degree polynomials that gives a pleasant entry and exit to a slope. Figure 3.3 shows an example of an SZCurve definition.

```
<SZCurve>
  <Polynomial>
    <begin direction="-0.125207" x="0" y="131.439" />
    <end direction="0.0514822" x="50" y="125.178" />
  </Polynomial>
  <Polynomial>
    <begin direction="0.0514822" x="50" y="125.178" />
    <end direction="-0.119704" x="100" y="127.752" />
  </Polynomial>
  <Polynomial>
    <begin direction="-0.119704" x="100" y="127.752" />
    <end direction="-0.0555701" x="150" y="121.767" />
  </Polynomial>
  (...)
</SZCurve>
```

Figure 3.3: Example of elevation curve definitions in the RoadXML file

Each Polynomial element defines a third-degree polynomial, ax^3+bx^2+cx+d. The coefficients are derived by the applications using the file from the beginning and ending direction (both in radians, value of zero is flat), and the beginning and ending elevation values. It's easy to see that we may construct a linear equation with four unknowns by inserting the values for x and y into the equation $ax^3 + bx^2 + cx + d = y$, and doing the same for the derivative using the slope (derived from direction) as the right-hand-side.

For this project, a polynomial is generated for intervals of length 50 as this seems to give good enough resolution, while not giving a very bumpy road. The beginning direction for polynomial p_i are taken as the average slope between x_i and x_{i+1}. The ending direction are the next polynomial's beginning direction, in order to have continuity for the slope.

3.2 Integration with snow simulator

This section presents the modifications done to the snow simulator to accommodate road models and real-life terrains. First, the scaling of the terrain was modified, and this is described in section 3.2.1. Then, the mesh loader is described. Finally, the terrain adjustments necessary to accommodate the road models are described.

3.2.1 Scaling of terrain and model

The original snow simulator used a fixed value for the maximum elevation given by a height map, without taking into consideration the scale of the map. In fact, there was no parameters describing the physical dimensions of the map; a height map was simply taken in, placed within a square of a certain dimension, and the height of each point was scaled statically to a value between 0 and 64. In reality, we want the height of each point in the height map to be proportionally correct to the actual widths and heights of the map. In other words, if we have a map of some mountain, say mt. Everest, and the height map defines an area of $10km \times 10km$, and is contained within a square of 512×512 in the snow simulator's coordinate system, we want the y coordinate of the highest peak to be $8.848 \cdot 512/10 = 453.0176$, again in the snow simulator's coordinate system.

To achieve this, a few extra parameters was introduced to the snow simulator:

- Resolution of the height map, in meters, in the X and Z direction (R_x, R_z).
- Minimum and maximum elevation defined by the height map, in meters (e_{min}, e_{max} (for mt. Everest, this could be 0 and 8848)

Using this we can now find the elevation e at each height map point, which is as given by equation 3.11. R_x is the resolution of the height map in the x direction, and S_x is the size of the scene in the x dimension. E is the elevation computed from the height map. d is the 16-bit integer stored in the height map; $d \in [0, 2^{16} - 1]$. n is the number of elevation points in the x direction.

$$E = \frac{d(e_{max} - e_{min})}{2^{16}} + e_{min} \tag{3.10}$$

$$e = \frac{(E - e_{min})S_x}{nR_x} = \frac{d}{2^{16}} \cdot \frac{(e_{max} - e_{min})Sx}{nRx} \tag{3.11}$$

3.2.2 Model loader and format

The models used for roads and other meshes in the snow simulator is the Wavefront OBJ format, which is a plain text format containing information about the material used, positions of vertices, texture coordinates, faces and normals. In the snow simulator, the vertices, normals and texture coordinates are stored in flat arrays, V, N, T, respectively, as they are loaded from the OBJ file. Faces are stored as an array where each element has nine indices into these arrays $(i_{[vnt]}, j_{[vnt]}, k_{[vnt]})$. A triangle's vertices are then $V(i_v)$, $V(j_v)$ and $V(k_v)$, and for normals it is similar, but with the normal indices instead. This format saves quite a bit of memory, as vertices can be shared between multiple faces, and a face takes up one third of the space of the vertices, normals and texture coordinates that define it.

Wavefront OBJ files has a very simple structure. The first word on each line indicates what data is contained on that line, e.g. vertex coordinates, texture coordinates, or similar. The rest of the line has the data itself; for a vertex, this would be three floating point values indicating the vertex' position. A list of "keywords" (i.e. words that begin a line) is listed in table 3.2. The data format in this table is the expected format of the data following the keyword. Note that the list is not a complete reference of the OBJ format; faces may have more than three vertices, and there are more keywords supported, but for the snow simulator, this is sufficient.

Keyword	Meaning	Data format
#	Comment	No data.
v	Vertex	$x \quad y \quad z$
vt	Texture coordinates	$s \quad t$
vn	Vertex normal	$x \quad y \quad z$
f	Face	$i_v \quad j_v \quad k_v$
		OR $i_v/i_t \quad j_v/j_t \quad k_v/k_t$
		OR $i_v//i_n \quad j_v//j_n \quad k_v//k_n$
		OR $i_v/i_t/i_n \quad j_v/j_t/j_n \quad k_v/k_t/k_n$
mtllib	Load material library	Filename
usemtl	Use material	Material name

Table 3.2: Wavefront OBJ keywords identifying the type of data on each line

Materials are loaded through material libraries; these are basically a list of named materials with textures and reflectance values for specular and diffuse reflection. A new MaterialCollection class instance is created when an "mtllib" keyword is encountered; then, the file specified on that line is opened and parsed. When a "newmtl" line is encountered, a new Material instance is added to the MaterialCollection, and the following lines up until the next "newmtl" line is interpreted as parameters for that material. Table 3.3 shows the different keywords for these files.

3.2.3 Terrain adjustments

The generated road model may not fit the terrain perfectly due to various reasons. For instance, when sampling elevations along the curve when generating the RoadXML,

Keyword	Meaning	Data format
#	Comment	No data.
newmtl	Create new material	Material name
Ka	Ambient color	$R \quad G \quad B$
Kd	Diffuse reflectance	$R \quad G \quad B$
Ks	Specular reflectance	$R \quad G \quad B$
map_Kd	Diffuse reflectance map (texture)	Filename

Table 3.3: Material library keywords

bilinear interpolation is used to look up a height value because a point on the curve may not correspond exactly to a grid point. The elevation given by this interpolation does not correspond exactly to the height of the point on the corresponding triangle in the triangle mesh that is generated in the snow simulator. In addition, the road has a width as well, while the trajectory generated by the road generator has an infinitesimal width, which causes many vertices to be placed above or below the terrain.

When adjusting the terrain, we want to adjust the height map elevation points so that if grid point (x, z) is near the road surface in the xz-plane, the elevation at that point should be set to the elevation of the closest vertex in the road mesh. Figure 3.4 shows this idea. The small dots represent grid points in the terrain, and the large dots are road mesh vertices. The color of the grid points indicate from which road vertex their height is extracted from. So a gray grid point's height is set to the nearby gray road vertex's y-value. In reality, of course there are a much higher density of vertices in the road model.

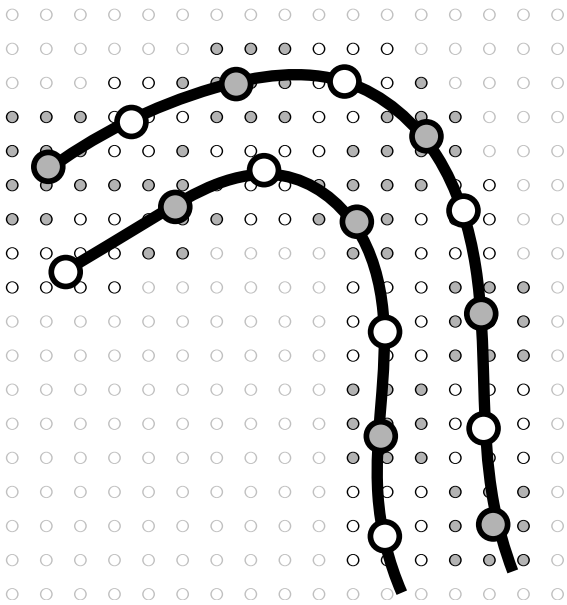


Figure 3.4: Terrain adjustment. Small points are terrain grid points. Large points are road mesh vertices.

It is not sufficient to go through each road vertex and set the closest grid point's elevation to that vertex' y-value. The reason is that if we first set grid point (x, z)'s

height to $v_1.y$, we may later find a new vertex v_2 that are further away, but still has (x, z) as the closest grid point, and thus will overwrite the previous height value $v_1.y$ with $v_2.y$. It is also not a good solution to go through all the grid points close to the trajectory and check the distance to all road vertices, as there may be hundreds of thousand of vertices, and hundreds or thousands of grid points, giving a very inefficient solution.

Instead, terrain adjustment may be done in two passes. In the first pass, a distance map is created. In the second pass, the information stored in this distance map is used to determine what height each grid point should have. The distance map is simply a 2D array of the same dimensions as the terrain height map, where each element contains two values: the distance to the closest road vertex, and that road vertex' elevation.

Initially, all distances are set to infinity. Then, we traverse the vertex list of the mesh, and for each vertex, check the distance to the closest grid point and the immediate eight neighboring grid points to that vertex, and update the distance map accordingly. Adjusting the heights is then done by simply iterating through the distance map, and if an element has a non-infinite distance, set the height of each grid point to the height specified in the distance map. This algorithm, described in algorithm 5, has an asymptotic running time of $O(|V| + mn)$ where $|V|$ is the number of road vertices, and m and n is the size of the height map.

Algorithm 5 Terrain adjustment algorithm using a distance map

Require: Uninitialized distance map $D(\mathbf{p}) = (s_{ij}, e_{ij})$; s_{ij} is the distance to the closest road vertex, e_{ij} is the elevation of that vertex
Require: Grid $G(i, j) = e_{ij}$ of height values
Require: Set of road vertices V

```
Initialize D such that ∀_i ∀_j (d(i, j) = (∞, 0)).
for all v ∈ V do
    Closest grid point p := (round(v.x), round(v.z))
    for all i ∈ {−1, 0, 1} do
        for all j ∈ {−1, 0, 1} do
            q := p + (j, i)
            if q is outside of G then
                continue
            end if
            if ||q − v||_2 < D(q).s then
                D(q) = (||q − v||_2, v_y)
            end if
        end for
    end for
end for
for all d = (x, z, s, e) ∈ D do
    if s < ∞ then
        G(x, z) := e
    end if
end for
```

3.3 USGS DEM converter

Real-world height maps are often stored in the USGS Digital Elevation Model (DEM) format. Among the users are Kartverket in Norway, who provides digital elevation maps of Norway. There is also a considerable amount of elevation data in the DEM format for the United States, although it has been mostly superseded by the SDTS format, also developed by the USGS[20]. However, the snow simulator uses raw 16 bit integer height maps, for simplicity. Converting a USGS DEM map to 16 bit RAW takes time, and as such, it was decided that instead of directly supporting DEMs in the snow simulator, the best approach was to create a stand-alone converter from DEM to RAW.

3.3.1 Parsing the DEM file format

USGS DEM is a format in human readable ASCII characters, and the files are segmented into blocks of 1024 bytes. There are three types of records in a DEM file: one A-record, which is the header, contained in one block; multiple B-records, where each B-record store one column of elevation data and one C-record, which stores statistics of errors and accuracy of the data. We are here concerned about the A and B records since these contain the actual data and meta data.

Each data element in the DEM format is contained within two offsets in a record. For instance, the spatial resolutions are contained 816 bytes into the A-record, and ending at byte 851. This area contains three values; the x, y and z resolution of the DEM. Since the format is so regular, parsing each field is very straight forward. First, read the block from file (1024 bytes), and store it in a string. Then, this string is easily sliced at the field boundaries and then split into separate elements, and finally each element is parsed as numbers.

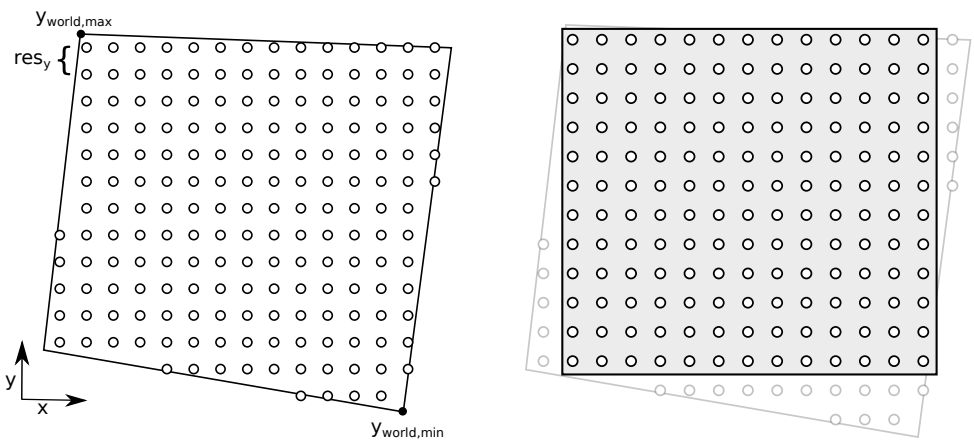
For the elevation data in the B-record, the following conversion must be done, in order to map the data to actual heights, just as described in section 2.7.2:

$$h_i = ((e_{max} - e_{min}) \cdot y_i + d_y) \cdot res_y \cdot unit$$

where d_y is the height datum and e_{max} and e_{min} is the maximum and minimum elevation, respectively. $unit$ is the number of meters per unit specified in the A-record; $unit = 1$ for meters, or 0.3048 for feet.

3.3.2 Placing elevation data points and cropping

The DEM file format allows quadrangles that are not square or axis aligned. For instance, a DEM file may describe a height map of the shape shown in figure 3.5a, or the quadrangle may be rectangular, or many other polygons described by four vertices. The goal is to convert the DEM to a rectangular height map that can be used in other applications, like the snow simulator; in other words, the data points from the height map must be aligned in a grid, and finally cropped, as shown in figure 3.5b. In this figure, the gray box represents the final grid. The points outside of this box are removed in the cropping process.



(a) Example quadrangle, unprocessed (b) Example quadrangle, cropped

Figure 3.5: USGS DEM quadrangles; the small circles represent elevation data points

The first step of generating the cropped height map is to align each column from the B-records in a grid. In order to do this, we first must compute the dimensions of the bounding box of the quadrangle in order to hold all the data points. The number of B-records (width) is given by the A-record, but each B-record may differ in size, and even if they don't, data point i from B-record 1 may not correspond to the same vertical position as data point i in B-record 2.

The corners of the quadrangle in world coordinates is described in the A-record (see [18]). Since we know the spacing between points, we can therefore compute the number of rows m as

$$m = \left\lceil \frac{y_{world,max} - y_{world,min}}{res_y} \right\rceil$$

where $y_{world,max}$ and $y_{world,min}$ is the maximum and minimum quadrangle y-values in world coordinates, and res_y is the resolution in the y direction in meters. This is illustrated in figure 3.5. Now, after generating a grid of $m \times n$ elements that can hold all data points, for each B-record, we compute the starting row as

$$row_{0,j} = \left\lfloor \frac{y_{0,j} - y_{world,min}}{res_y} \right\rfloor$$

where $y_{0,j}$ is the y-position in world coordinates of the first data point in the j'th B-record. After the first point, all other points come sequentially, so

$$row_{k,j} = row_{k-1,j} + 1$$

where $i_{k,j}$ is the row of the k'th data point in the j'th B-record.

If the height map from the DEM is not rectangular, it must be cropped in order to produce a rectangular map, as illustrated in figure 3.5, or else we end up with a lot of bogus data points where the DEM does not contain any data, or a more complex data format would be required to describe the shape of the boundaries of the terrain. Another consideration is that we want to remove as few real data points as possible. For instance, in figure 3.5a, in order to create a rectangular map, we may

1. Crop the left-most and right-most columns, and the two bottom-most rows (removing 23 data points)
2. Or, crop the 8 bottom rows (removing 96 data points)
3. Or, crop the top 7 rows, and bottom 2 rows (removing 113 data points)
4. Or any other combination that produces a rectangular map.

Here, alternative 1 is clearly preferable. One way of finding the optimal way of cropping is representing the problem as a search in state space, and find a shortest path through the graph generated by the nodes (state, i.e. the intermediate grid) and the edges (removal of one row or column) with weights equal to the number of real data points being removed. However, this is time consuming, so a simpler method was used, where the row or column with the highest ratio of real data points over the length of the row or column was used. This appears to work very well in practice.

The pseudo code for placing the DEM data points in a grid and outputting a RAW can be seen in algorithm 6. Since the parsing itself is trivial (it's simply a string slicing, splitting and casting operation), this is assumed to be done already.

Algorithm 6 Complete algorithm for aligning and cropping a USGS DEM

Require: One parsed A-record A, and a list of n parsed B-records B
  Get res_y and quadrangle vertices from A-record
  $y_{world,min} \leftarrow$ smallest y-coordinate of all quadrangle vertices
  $y_{world,max} \leftarrow$ largest y-coordinate of all quadrangle vertices
  $m \leftarrow \lceil (y_{world,max} - y_{world,min})/res_y \rceil$
  $H \leftarrow$ empty $m \times n$ matrix
  for all $b \in B$ **do**
    Get $y_{0,j}$ from b
    $col \leftarrow$ column index of b
    $row \leftarrow \lfloor (y_{0,j} - y_{world,min})/res_y \rfloor$
    for all elevation $h \in b$ **do**
      $H(row, col) \leftarrow h$
      $row \leftarrow row + 1$
    end for
  end for
  $x_{min} \leftarrow 0$, $y_{min} \leftarrow 0$
  $x_{max} \leftarrow n$, $y_{max} \leftarrow m$
  while there are rows or columns left to crop **do**
    $n_{left} \leftarrow$ # of missing data points in the left column
    $n_{right} \leftarrow$ # of missing data points in the right column
    $n_{top} \leftarrow$ # of missing data points in the top row
    $n_{bottom} \leftarrow$ # of missing data points in the bottom row
    $r_{left} \leftarrow n_{left}/(y_{max} - y_{min})$, $r_{right} \leftarrow n_{right}/(y_{max} - y_{min})$
    $r_{top} \leftarrow n_{top}/(x_{max} - x_{min})$, $r_{bottom} \leftarrow n_{bottom}/(x_{max} - x_{min})$
    if r_{top} is largest **then**
      $y_{min} \leftarrow y_{min} + 1$
    else if r_{bottom} is largest **then**
      $y_{max} \leftarrow y_{max} + 1$
    else if r_{left} is largest **then**
      $x_{min} \leftarrow x_{min} + 1$
    else if r_{right} is largest **then**
      $x_{max} \leftarrow x_{max} + 1$
    end if
  end while
  Write $H(y_min : y_max, x_min : x_max)$ to file.

Chapter 4

Results

In this chapter I present visual results for the roads that are imported into the snow simulator, and some performance analysis of the road generator. I evaluate the quality of the heuristic used in the A* search by comparing the A* search to Dijkstra's algorithm, and I compare the performance and cost impact of varying the resolution (i.e. coarsening) of the height map.

4.1 Test bench hardware

All tests and renderings were performed on a desktop computer with the specifications given in table 4.1. The Tesla C1060 device was used for all GPU computing tasks. The

Component	Specifications	Remarks
CPU	Intel Core 2 Quad Q9550 2.83GHz	
L1 Cache	64KB	32KB data, 32KB instruction
L2 Cache	12MB	Unified; two cores share 6MB
Memory	4GB	
GPU 1	NVIDIA Quadro FX 5800	
GPU 2	NVIDIA Tesla C1060	GPU used for the GPU computing tasks

Table 4.1: Hardware specifications for test bench

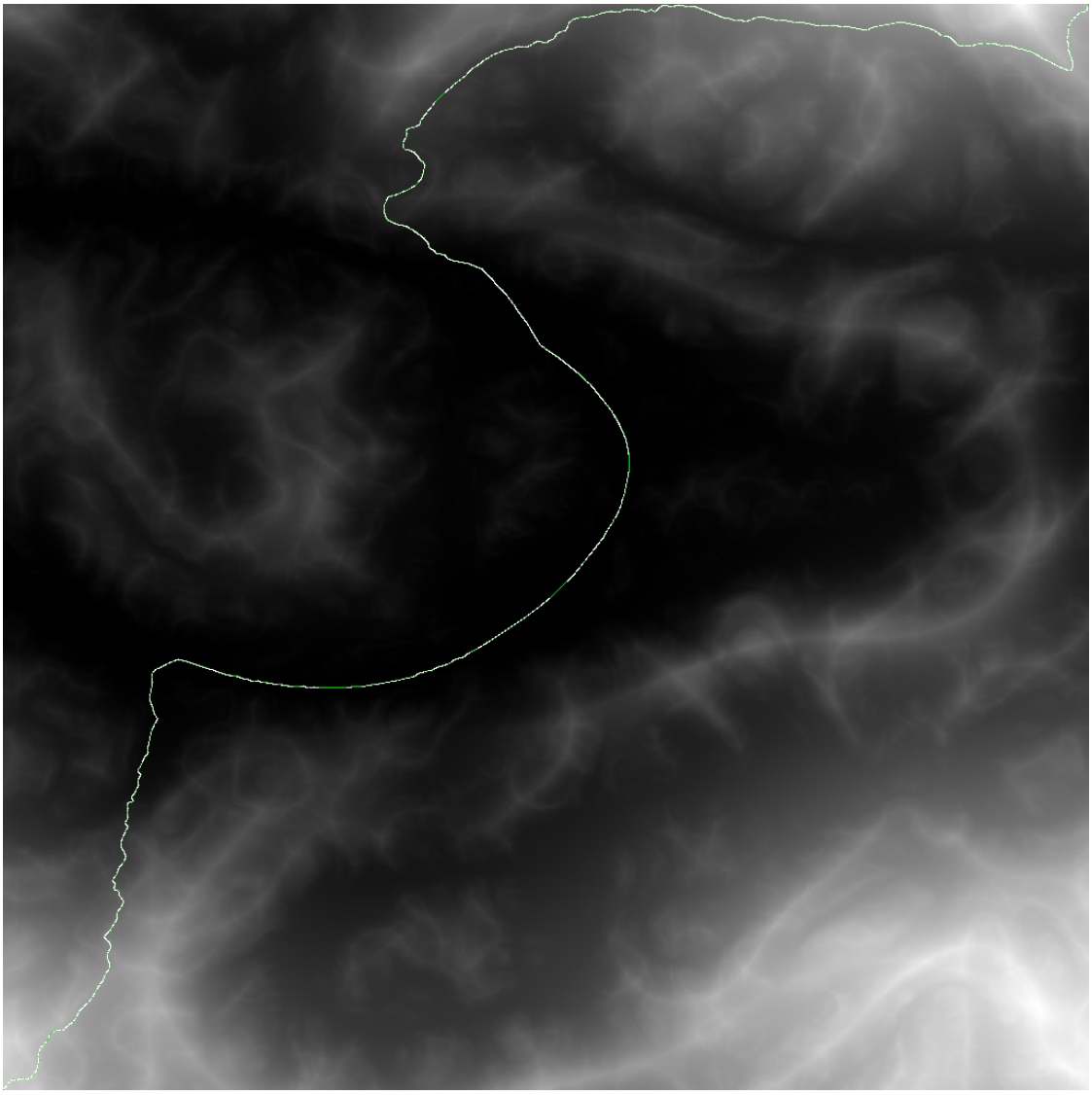
GeForce Quadro FX 5800 was used for rendering.

4.2 Visual results

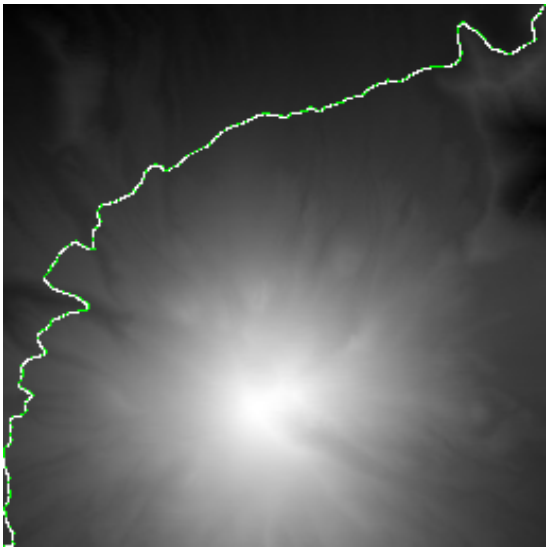
In this section, we will look at visual results for the generated roads; both as an overlay to the height map, and some of the generated roads imported into the snow simulator.

First, we look at the road trajectories that are generated for different height maps. These are shown in figure 4.1. Figure 4.1a shows an automatically generated terrain (a fractal terrain) representing a mountainous terrain, and a valley. Figures 4.1b and 4.1c is height maps representing Mt. St. Helens before and after the eruption; this is terrains

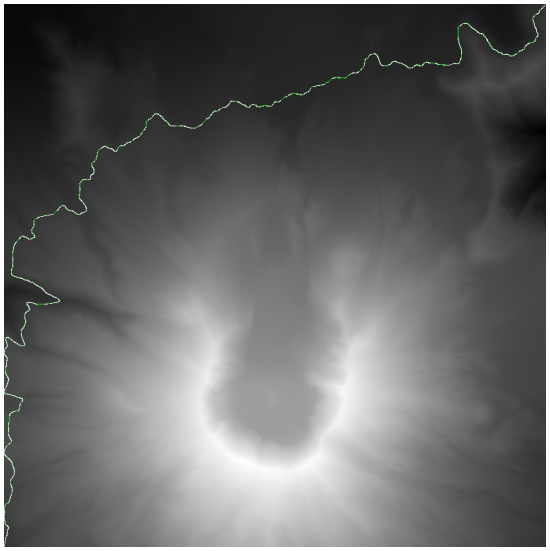
converted from actual digital elevation maps (see [21] for the source DEMs). We see that generally, the trajectories seem to follow the terrain contour lines, as this minimizes the slope cost of the path, which is what was expected.



(a) Auto-generated mountainous terrain



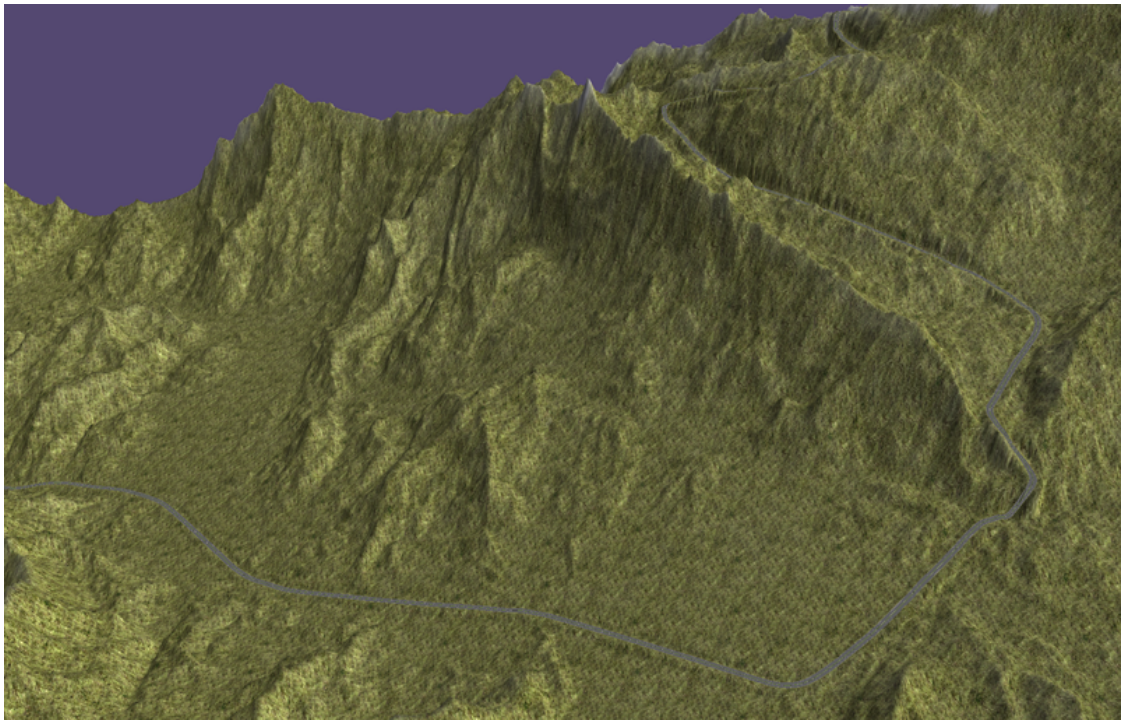
(b) Mount St. Helens (before eruption)



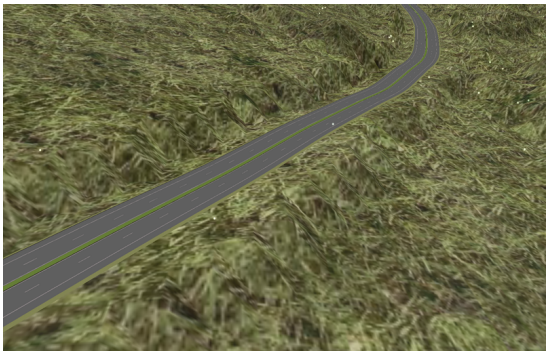
(c) Mount St. Helens (after eruption)

Figure 4.1: Procedurally generated road trajectories for different height maps

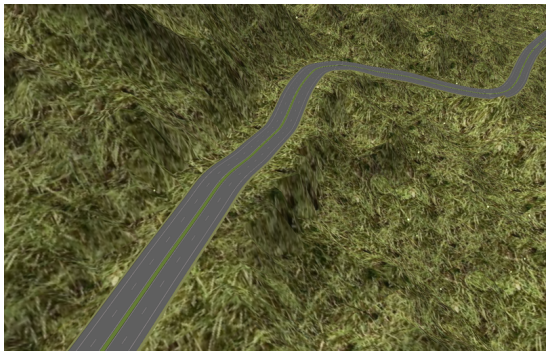
After loading the model into the snow simulator and adjusting the terrain, we can see the roads as it curves through the terrain. Figure 4.2 shows the road and the terrain without snow cover for the random generated fractal terrain. We clearly see the effects of the terrain adjustments in the bottom two screenshots, i.e. figures 4.2b and 4.2c. Note that snow rendering was disabled on the overview image (figure 4.2a) in order to improve visibility of the road.



(a) Terrain overview



(b) Detailed view of road

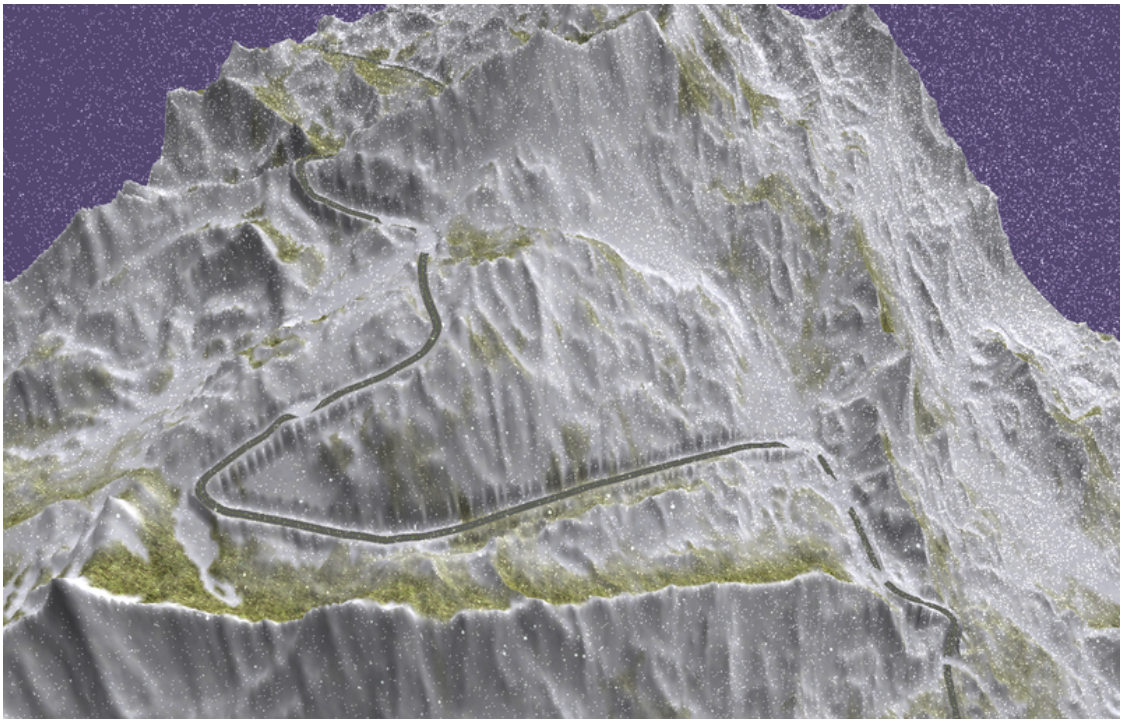


(c) Significant terrain adjustments

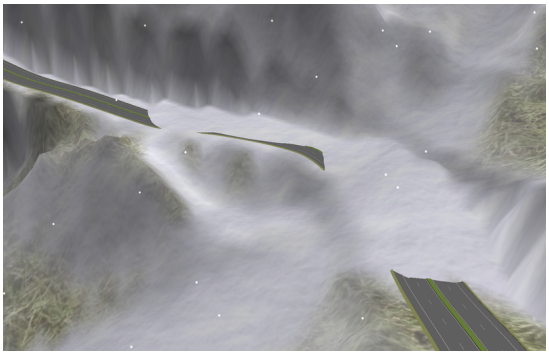
Figure 4.2: Road and terrain before snow cover

After some snow has fallen, as in figure 4.3, the terrain has been more or less covered in snow except for some shadowed areas. The snow height is continuously "smoothed", so that snow that falls in areas with a steep gradient in the terrain, is moved downwards along the gradient, simulating sliding snow (think of it as a mini avalanche). We clearly see this effect on the road in figures 4.3b and 4.3c.

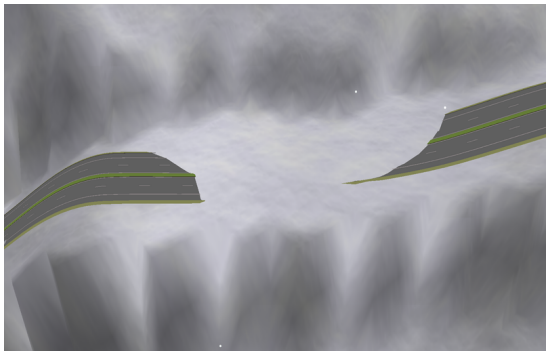
Another map that was used was Mt. St. Helens before and after the eruption.



(a) Snow covered terrain overview



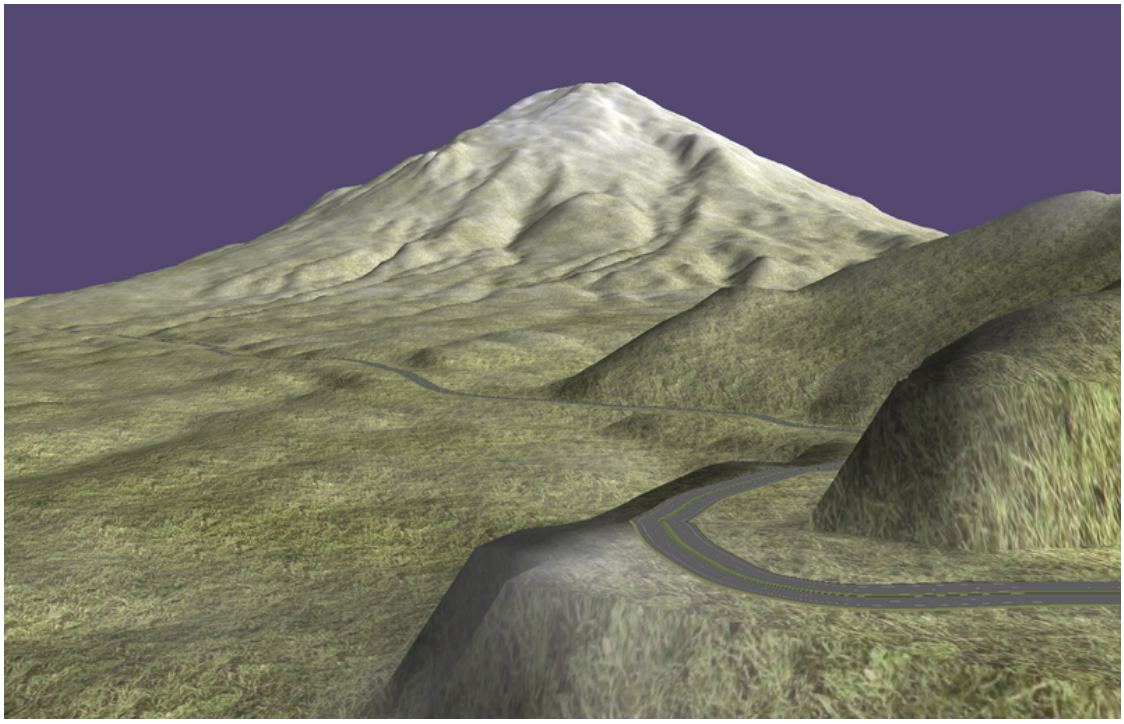
(b) Snow from higher up covering road



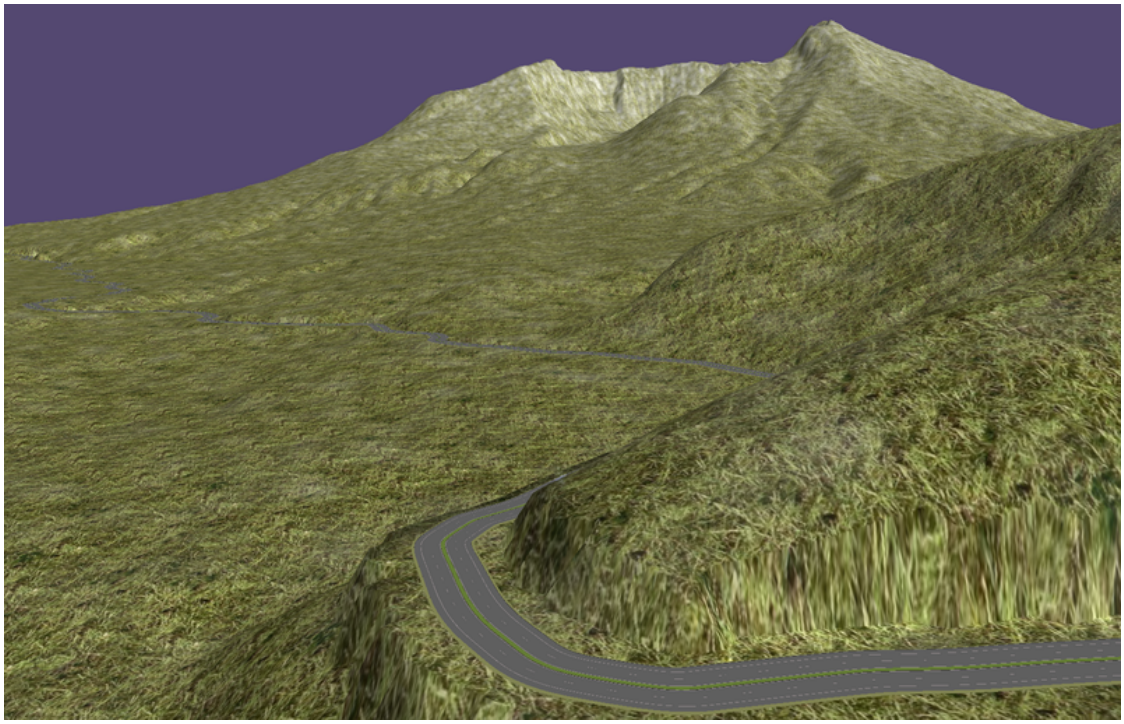
(c) Another case of snow sliding down on the road

Figure 4.3: Road and terrain with snow cover

These are shown in figures 4.4 and 4.5.

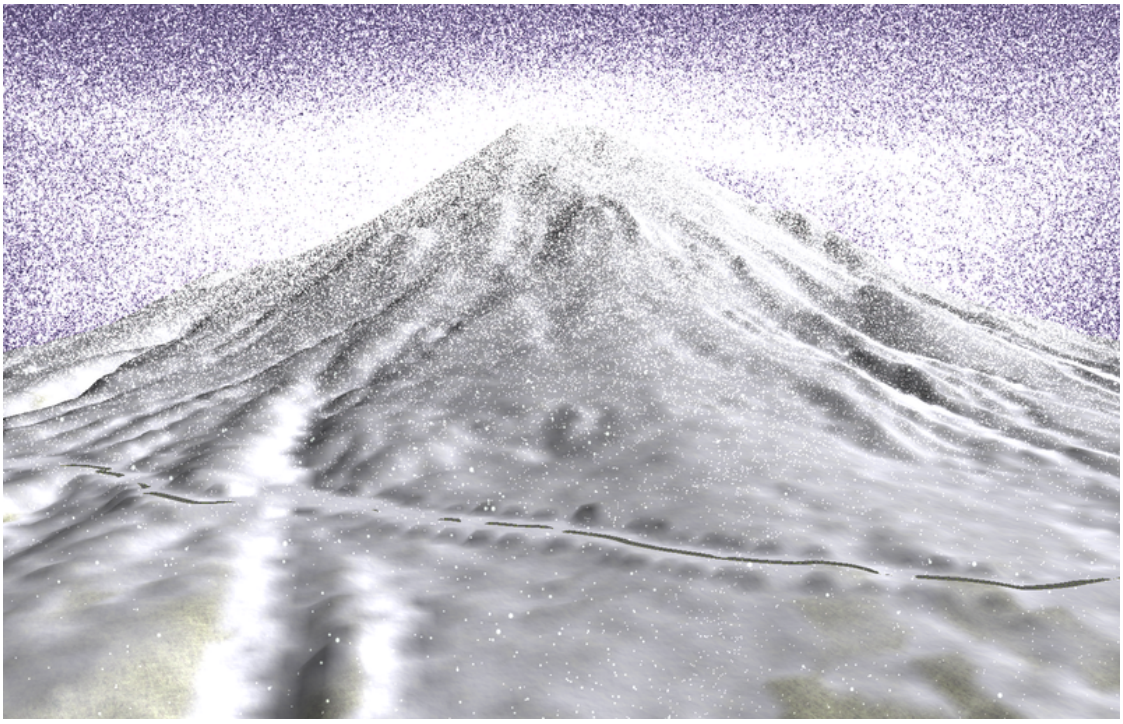


(a) Before eruption

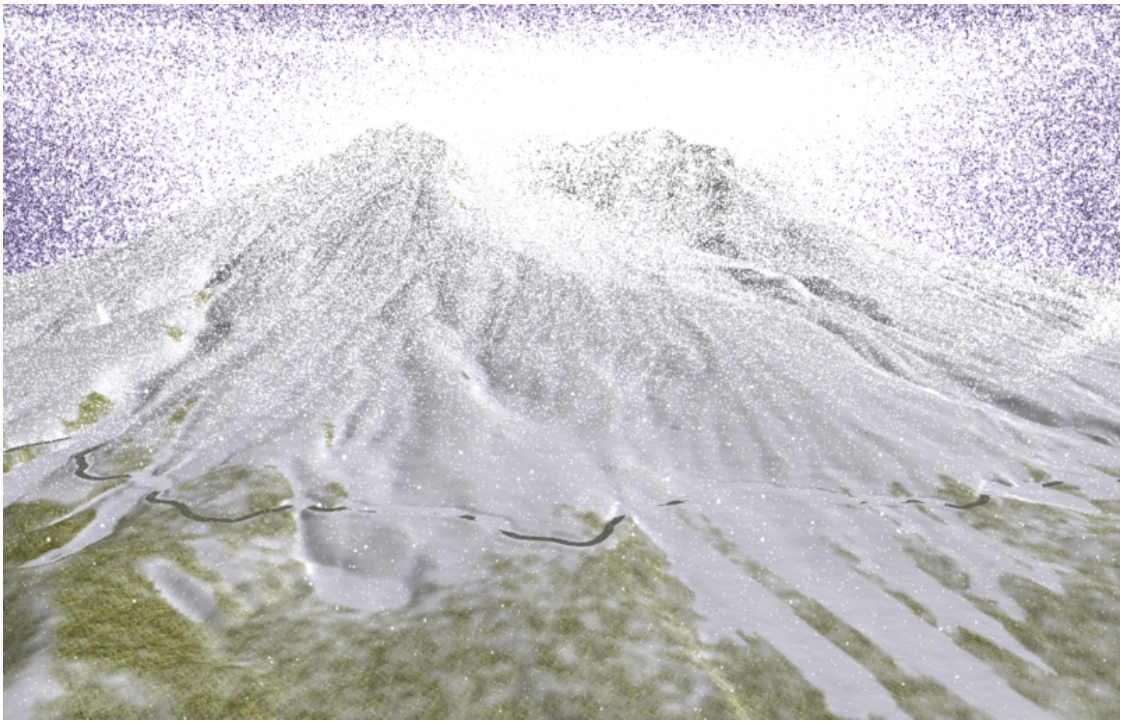


(b) After eruption

Figure 4.4: Closeup of road in the Mt. St. Helens terrain



(a) Before eruption



(b) After eruption

Figure 4.5: Overlooking of Mt. St. Helens with heavy snow

4.3 Performance of road generator

In this section, I present results from performance testing of the road generator.

4.3.1 Testing parameters

Four different height maps have been tested: digital elevation data from Mt. St. Helens before and after the eruption (from [21]), a random fractal terrain representing a valley surrounded by mountains, and Trondheim. These maps range from very small to very large, with different resolutions. Table 4.2 gives an overview of these maps.

Map	Shorthand	Dimensions	Resolution (m)
Mt. St. Helens before eruption	helens_before	256×256	30
Mt. St. Helens after eruption	helens_after	768×768	10
Random fractal terrain	mountains	1024×1024	10
Trondheim	trondheim	4096×4096	20

Table 4.2: Height maps used for performance testing

The weights for road length, slope and curvature is set to 1, 100 and 100, respectively. A threshold of maximum curvature was set to 0.03, which is roughly the curvature of a circle with a radius of 33 meters. Paths with an estimated curvature of more than 0.03 is not considered, because these would give unrealistically sharp turns. A neighborhood of $k = 5$ is used for all test cases. This gives 30 neighbors per node, except for the edge nodes which have less.

4.3.2 Performance of Dijkstra vs A*

The main difference between Dijkstra's algorithm and the A* algorithm is that A* takes an estimated guess of the distance to the goal, and uses this, together with the cost to reach the next node from the start, in deciding which node to expand next. With a good heuristic, A* may expand far fewer nodes than Dijkstra's algorithm, typically resulting in a lower search time.

In this section I present a comparison of the performance using Dijkstra's algorithm, and the A* algorithm with the heuristic in equation 3.8 in section 3.1.1. Since the algorithms are identical except for the heuristic, Dijkstra's algorithm is simply A* with $h = 0$. In the Dijkstra runs, the heuristic is therefore simply set to zero, but the algorithm is identical otherwise.

	A*		Dijkstra		
Map	**Running time**	**Cost**	**Running time**	**Cost**	**Speedup**
helens_before	2.7	104265.54	5.42	104265.5	2.01
helens_after	16.64	79424.38	38.96	79424.4	2.34
mountains	71.86	357921.25	87.07	357921.2	1.21
trondheim	1449.4	232596.41	1765.92	232596.4	1.22

Table 4.3: A* vs. Dijkstra's algorithm

The starting points for all was chosen so that a solution to the shortest path was non-trivial; a trivial setup is when the actual cost to the goal is close to the heuristic estimate of the cost to the goal; in this case, the comparison is unfair and unrealistic, as

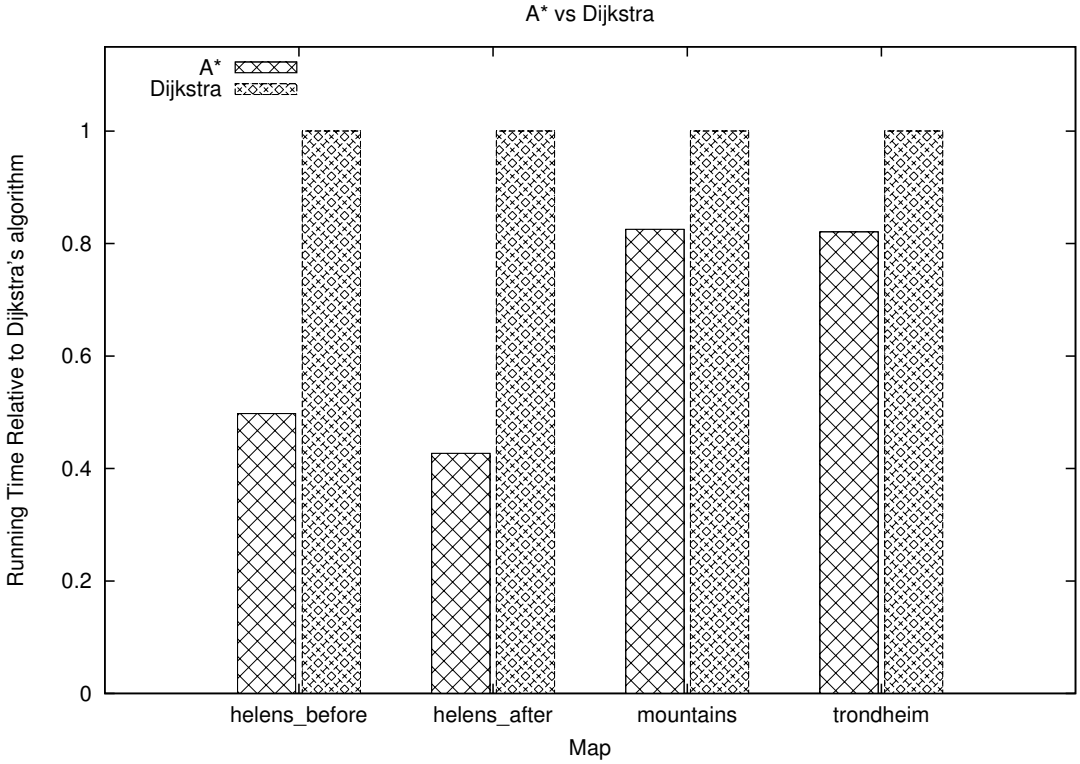


Figure 4.6: A* vs. Dijkstra's algorithm. Running times are relative to that of Dijkstra's algorithm

A* will almost immediately find the goal node by expanding only those nodes that the shortest path contains.

Figure 4.6 shows the running time of Dijkstra's algorithm relative to A*, with the running time of Dijkstra's algorithm defined to be one. Table 4.3 shows the results in tabulated form. The costs of the road generated by Dijkstra and A* is included as a confirmation that the heuristic used indeed results in an optimal trajectory.

We see that for Mt. st. Helens, there is more than two times speedup for using A*, while the Trondheim and fractal maps show more modest improvements. The Mt. St. Helens maps are generally simpler maps with only one major elevation peak, which makes the heuristic more accurate. The heuristic does not account for a rough terrain, but rather the difference in elevation between the start and end node. Both the mountains and Trondheim maps contain significant elevation changes throughout the terrain, which makes the heuristic less accurate.

4.3.3 Effect of grid coarsening

In this section I present performance results for different densities of grid points. The time used to generate the shortest path and also the optimality of the path, compared to using a density of 1, i.e. where every height point is considered a node in the search graph. The results are presented in table 4.4, and a plot of the relative cost increase due to grid coarsening with respect to the density is presented in figure 4.7.

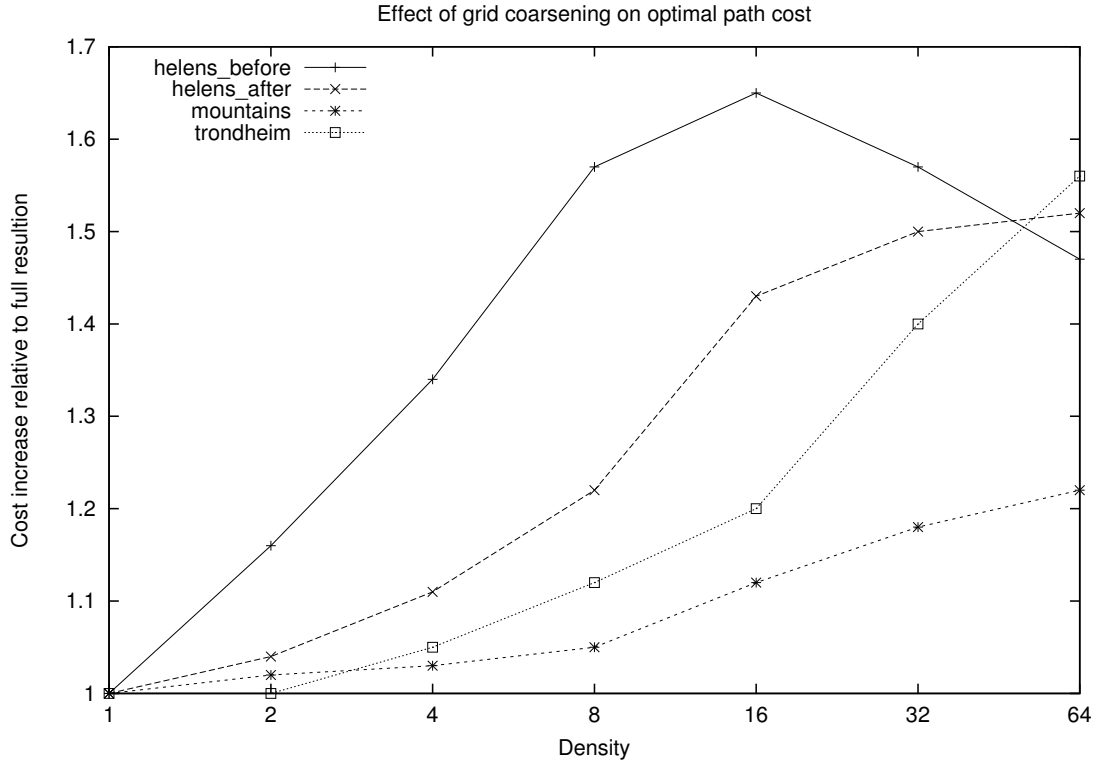


Figure 4.7: Effect of grid coarsening on optimal path cost

We see here that even though a density of one is optimal with regards to cost, which is not surprising because this gives the highest resolution in finding a path, the running time increase for using such a fine resolution is rather large.

It is interesting to note that the real-world maps take a higher cost-penalty when the resolution is decreased, while the auto-generated fractal terrain (Mountains) is at most 1.22 times worse with a grid density of one node each 64×64 height points than with one node for every height point.

Fractal terrains typically have a low frequency on large elevation changes due to the way they are usually generated. Start with a flat terrain, and until some recursive depth d is reached, recursively subdivide the terrain, and perturb the elevation of each subdivision; the amplitude of the perturbation typically decreases with recursion depth, giving high-frequency changes of small amplitude, and low-frequency changes of big amplitude. This means that using lower resolution on a fractal map does not necessarily give a huge increase in cost, because the costly elevation changes usually have a low frequency, so it can be dealt with at a lower resolution.

Real world terrains also have fractal-like geometry[22], but these terrains may (and often will) exhibit (relatively) high-frequency elevation changes with a large amplitude also. This of course gives a higher penalty for using a lower resolution as these high amplitude elevation differences may not be circumvented by building the road around them.

Density	Running time	Speedup	Road cost	Difference from "optimal"
Mt. St. Helens (before eruption)				
1	2.79	1.0	127811.4	1.0
2	1.26	2.22	148230.5	1.16
4	0.61	4.62	171457.2	1.34
8	0.28	9.95	200554.1	1.57
16	0.11	24.78	211139.1	1.65
32	0.04	73.22	200443.2	1.57
64	0.0	857.34	188397.3	1.47
Mt. St. Helens (after eruption)				
1	27.15	1.0	124590.4	1.0
2	11.6	2.34	129981.5	1.04
4	5.14	5.29	138310.6	1.11
8	2.41	11.28	152238.6	1.22
16	1.22	22.28	177559.5	1.43
32	0.53	50.92	186755.2	1.5
64	0.19	140.54	189069.5	1.52
Mountains				
1	94.53	1.0	481881.1	1.0
2	37.8	2.5	489985.2	1.02
4	16.75	5.64	497859.8	1.03
8	7.61	12.42	507023.9	1.05
16	3.5	26.99	540848.4	1.12
32	1.55	61.17	570683.6	1.18
64	0.6	157.56	588097.1	1.22
Trondheim				
1	1775.99	1.0	327124.9	1.0
2	668.83	2.66	328696.4	1.0
4	282.29	6.29	344508.6	1.05
8	128.03	13.87	365704.2	1.12
16	60.86	29.18	393396.3	1.2
32	29.36	60.5	458742.5	1.4
64	14.02	126.65	509933.1	1.56

Table 4.4: Effect of density of grid

Chapter 5

Future work

There are several ways the work from this project could be extended, both in the snow simulator and in the road generation tool. In this section I present some possible future projects for the snow simulator and the path finding application. Two of the major topics of these projects is more flexible and more efficient road generation, and a higher degree of integration of the road generator into the snow simulator.

5.1 Waypoints, multiple destinations and influence maps in road generator

In this project, roads are generated from two endpoints with no way of adding multiple destination, or otherwise modifying the path other than adjusting the weights of the cost functions. A future project could be to implement waypoints in the road generator, so that we can force the road to go through certain points, e.g. for connecting cities or making roads that go up to landmarks.

The RoadXML format already supports road intersections. This could be used with an extended version of the current road generation algorithm to add multiple destinations. Given multiple end points, find a "good" road that connects all end points. This could be modelled as a minimum spanning tree over the same graph as used in this project for path finding

Another interesting project would be implementing procedurally generated hierarchical road networks as described in [16]. Here, there is a certain number of "levels" of roads: Highways, connecting big cities, major roads connecting big and small cities, and minor roads connecting small cities. Each of these types of roads have different requirements. For instance, highways may have a smaller tolerance for curvature due to the higher speed limits than major and minor roads.

5.2 Parallelization of cost computation

Each node used in the A* search has a significant number of neighbors, the number of which is dependent on the value of k when constructing the neighborhood mask, as shown in algorithm 3 in section 3.1.1, and also described in section 2.5.1. [1] uses $k = 5$, which results in 30 neighbors per node. Computing the cost functions described in 3.1.1 then becomes fairly expensive, partly because of the sheer number of neighbors, but also because computing the cost to each neighbor involves doing numerical integration over the line segment connecting the two nodes.

It becomes obvious that if we were to compute the cost between all nodes and their neighbors, this would be an embarrassingly parallel problem, and thus may be well suited for parallelization on e.g. a GPU. We could then pre-compute the cost to every neighbor, store this cost in a two-dimensional array, and then simply look up the cost in the array when computing the shortest path.

Pre-computing the costs does imply that some extra work is done, namely computing the costs for nodes that would not be expanded by the A* algorithm in the first place. These costs would not be used. The assumption necessary for this approach to be viable is that the GPU has enough computing power to keep the cost of doing this extra work lower than computing the cost serially for each expanded node.

5.3 Using data from snow simulator as a cost function

When building roads, it makes sense to avoid building it where snow buildup is significant. This may for instance be right below mountain sides where snow would fall down on the road. Using data generated from snow simulations, we could use the snow depth at each point in the terrain as a cost function for where to put the road.

Given two points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, this cost can be expressed mathematically as

$$c_{snow}(\mathbf{p}_i, \mathbf{p}_{i+1}) = w_{snow} \int_0^1 s((1-t)\mathbf{p}_i + t\mathbf{p}_{i+1})dt$$

where $s(\mathbf{p})$ is the snow height of the terrain at point $\mathbf{p}$. This can, as with the cost for slope described in 3.1.1, be solved numerically as

$$c_{snow}(\mathbf{p}_i, \mathbf{p}_{i+1}) \approx w_{snow} \sum_{j=0}^{N-1} hs((1-t_j)\mathbf{p}_i + t_j\mathbf{p}_{i+1})$$

where $t_j = jh$ and $N = 1/h$.

This feature would require us to first run the snow simulator for a given amount of time, extract the snow height map to a file, and then load this file into the road generation tool, causing there to be an extra step in the road generation process, but in turn it would give more realistic trajectories based on real concerns.

5.4 Integration of USGS DEM parser into the snow simulator

In this project, the USGS DEM parser was developed as a standalone Python script. Python, since it is an interpreted language, is generally slower than a compiled language like C++. However, it could be viable to implement the USGS DEM parser in C++, and then integrate it with the snow simulator in order to be able to load DEM files directly, without having to convert them to 16 bit RAW first as an intermediate step. The viability of this is limited by the performance of the DEM parser because if parsing takes an unreasonable amount of time, this becomes an annoyance to the users of the snow simulator.

5.5 Texture blending in snow simulator

With the support for real-world digital elevation maps through parsing of USGS DEM files, it is natural that additional realism is added by implementing texture blending in the terrain. Texture blending can be used, for instance, to have a different texture on steep hills (like mountain sides), plains (like grass plains or fields) or high mountain tops. In the real world, mountain sides and summits are not covered by grass, like in the snow simulator, which makes it look unrealistic.

5.6 Road model generation in snow simulator

In [1] they generate the road model directly from the control points given by the discrete shortest path (see section 3.1) without going through a third-party application like Octal SCANeR that was used in this project. This is also desirable for the roads in the snow simulator, but this was not the focus of this project due to time constraints. Performance is of course a concern here as unreasonable startup times of the snow simulator may cause annoyance among the users.

5.7 Integration of road generator in the snow simulator

Right now the road trajectory generator, i.e. the shortest path algorithm to generate the control points for the clothoid spline used to define the road trajectory, is implemented in a separate C++ application. This application could be eliminated completely by doing the path finding in the snow simulator itself. Combined with integration of the USGS DEM parser (see section 5.4), road model generation in the snow simulator (see section 5.6) and accelerated cost function evaluation using GPUs (see section 5.2), the snow simulator would be much more flexible in that it would not need any intermediate steps in order to load a DEM, generate a road through the terrain, and finally generate the road model.

Chapter 6

Conclusion

In this project I integrated the procedural road generation algorithm described in [1] with the snow simulator developed at the HPC lab at NTNU. I developed a C++ program for finding the trajectory using an A* search, which outputs a RoadXML file that is imported into a third-party tool called SCANeR Studio by Octal. The model from SCANeR, in the Wavefront OBJ format, was then imported as a triangle mesh in the snow simulator for rendering.

Results showed that the generated roads seem to have a natural look, with smooth curves that goes along the contours of the terrain instead of taking the shortest route over steep hills. The terrain was also adjusted using a method where each terrain vertex' y-component is set to the y-component of the closest road vertex; this seem to work well, as the roads seem to match the terrain almost perfectly.

Performance results show that generally, using a slightly lower grid resolution (e.g. half, or one quarter of the resolution of the height map) for the A* search seem to be preferable because the cost penalty for doing so is small compared to the performance increase. However, it is always a judgement call whether or not a full-resolution search should be performed; in some cases, running time does not matter that much as long as it is "reasonable", while other times, if for instance the road generator is integrated fully with the snow simulator, running time of the road generator matters more.

I also performed tests on using Dijkstra's algorithm instead of an A* search. Results show that for the Mt. St. Helens maps, the speedup was between 2 and 2.31, while the Trondheim and the procedurally generated mountains map showed more modest improvements, due to a more complex terrain. However, they did show a 1.22 and 1.21 times speedup, respectively, which is fairly good given the complex terrain.

Lastly, I presented possible future projects. One of these involved better road generation algorithms that supports waypoints, multiple destinations, hierarchical road networks and influence maps for allowing higher degree of control of how the road trajectory should be laid out. Another project involves using snow depth data from the snow simulator in a cost function for the road generator. Other future potential project include better integration of the road generator with the snow simulator, direct loading of USGS DEM files in the snow simulator and using texture blending for more realistic

rendering of the terrain.

Appendices

Appendix A

Computation of Fresnel integrals

The Fresnel integrals can be approximated by the formulas

$$C(t) = \frac{1}{2} + f(t)\sin\left(\frac{\pi}{2}t^2\right) - g(t)\cos\left(\frac{\pi}{2}t^2\right)$$
$$S(t) = \frac{1}{2} - f(t)\cos\left(\frac{\pi}{2}t^2\right) - g(t)\sin\left(\frac{\pi}{2}t^2\right)$$

where

$$f(t) = \sum_{n=0}^{11} f_n t^{-2n-1}, g(t) = \sum_{n=0}^{11} g_n t^{-2n-1}$$

and where f_n, g_n is given in table A.1. This gives an error of less than $5 \cdot 10^{-10}$. For $t < 1.6$, a good approximation with an error of less than $6 \cdot 10^{-10}$ can be found by using the truncated Taylor series.[14]

n	$\mathbf{f_n}$	$\mathbf{g_n}$
0	0.318309844	0
1	9.34626E-08	0.101321519
2	-0.09676631	-4.07292E-05
3	0.000606222	-0.152068115
4	0.325539361	-0.046292605
5	0.325206461	1.622793598
6	-7.450551455	-5.199186089
7	32.20380908	7.477942354
8	-78.8035274	-0.695291507
9	118.5343352	-15.10996796
10	-102.4339798	22.28401942
11	39.06207702	-10.89968491

Table A.1: g_n and f_n values for computing the Fresnel integral

Appendix B

USGS DEM to RAW converter reference

Synopsis

```
./usgstoraw.py [OPTIONS]
```

Description

usgstoraw is a program for converting USGS DEM files to 16 bit RAW files. Without any options, usgstoraw.py reads a DEM file from standard input and outputs a RAW file to standard input.

Options

The various options that can be given to usgstoraw.py is listed below. Many options have both short and long versions, e.g. the input file can be specified by both `-f` and `--inputfile`; in these cases, the alternative options are separated by a comma.

`-f FILE, --inputfile=FILE`
: FILE is name of USGS DEM file to read as input. If not set, usgstoraw.py reads from standard input.

`-o FILE, --outputfile=FILE`
: FILE is name of file to write the output to. If not set, usgstoraw.py writes to standard output.

`-x N, --cropwidth=N`
: N is the final width of the output RAW file. The DEM will be cropped to this width. If not set, the DEM will be cropped minimally, as described in algorithm 6 in section 3.3.2. It is an error to specify a size larger than the width given by the minimally cropped DEM.

`-y N, --cropheight=N`

N is the final height of the output RAW file. The DEM will be cropped to this height. If not set, the DEM will be cropped minimally, as described in algorithm 6 in section 3.3.2. It is an error to specify a size larger than the height given by the minimally cropped DEM.

`--cropx=N`

N specifies which side that should be cropped if the width of the DEM should be cropped. A value of 0 crops both sides, alternately, 1 crops the left side and 2 crops the right side.

`--cropy=N`

N specifies which side that should be cropped if the height of the DEM should be cropped. A value of 0 crops both top and bottom, alternately, 1 crops the top side and 2 crops the bottom side.

Examples

`./usgstoraw.py < test.dem > test.raw`

reads test.dem, passes it to standard input of usgstoraw.py and outputs raw data through standard output, which is redirected to test.raw.

`./usgstoraw.py -f test.dem -o test.raw`

does the same as above, but reads test.dem directly through file IO inside the script, and writes directly to test.raw. This is preferred.

`./usgstoraw.py -f test.dem -o test.raw -x 512 -y 512 -cropx=2 -cropy=1`

does the same as above, but crops the final RAW to 512×512 by cropping the right and top sides.

Appendix C

A* Road Generator reference

Synopsis

```
./pathfind [OPTIONS] <heightmap filename> <output filename w/o extension>
```

Description

pathfind is a program for creating a road trajectory through a terrain using an A* search. The output is a RoadXML file, as well as a simple list of control points. The terrain is in form of a height map read from a RAW file. The time spent finding the path as well as the total cost of the path is output to standard output right before the program exits. On completion, pathfind outputs the following files, assuming the output filename was set to "out".

out.rnd
: The RoadXML file describing the trajectory.

out.rd
: The control points of the road trajectory in an easily parsed format.

Options

The various options that can be given to pathfind is listed below. The options `-x` and `-y` are required.

`-x N`
: *REQUIRED*. Width of the height map in the number of points.

`-y N`
: *REQUIRED*. Height of the height map in the number of points.

-a N
Height (in meters) represented by the value 0 (zero) in the height map. Default is 0.

-b N
Height (in meters) represented by the value 65535 ($2^{16} - 1$) in the height map. Default is 2700.

-s N
Resolution of height map, in meters between height points. For instance, if this value is 10, then there are 10 meters between two neighboring values in the height map.

-d N
Grid coarsening factor. Setting this to anything larger than 1, will in practice give a lower resolution height map, where only one point every N will be used as nodes in the search.

-h N
Weight of the heuristic used in A*. Setting this to 0 gives Dijkstra's algorithm. Setting it to 1 gives regular A*. Setting it to a value > 1 may result in a path faster, but it might be sub-optimal with regards to cost.

--startx=N
N specifies the starting X position for the search. This is relative to the top-left corner of the height map, and is in number of height points from the left edge.

--starty=N
N specifies the starting Y position for the search.

--endx=N
N specifies the ending X position for the search.

--endy=N
N specifies the ending Y position for the search.

Examples

./pathfind -x 768 -y 768 -a 0 -b 1500 -s 10 test.raw output
reads test.raw as the height map, maps each of the height values to a value between 0 and 1500, and outputs output.rd and output.rnd. A resolution of 10 meters is assumed. The full height map resolution is used. The default starting and ending coordinates is used, i.e. start in the top-left corner and end in bottom-right corner.

References

[1] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, "Procedural generation of roads," *Eurogrpahics*, vol. 29, no. 2, 2010. [cited at p. iii, v, 1, 15, 17, 21, 24, 52, 53, 55]

[2] "Scaner studio." http://www.scanersimulation.com/. Viewed 27.10.2011. [cited at p. v]

[3] P. Ducloux, J. Chaplier, and G. Millet, *[RoadXML 2.2.1] Road Network Description - XML Format Specification*, jul 2009. [cited at p. 1, 20, 21, 28]

[4] "Shortest path problem." http://en.wikipedia.org/wiki/Shortest_path_problem. Viewed 10.10.2011. [cited at p. 3]

[5] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 338–346, 1984. [cited at p. 4]

[6] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, pp. 100 –107, july 1968. [cited at p. 4, 6]

[7] "Tesla c2050/c2070 gpu computing processor." http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf. Viewed 21.11.2011. [cited at p. 7]

[8] NVIDIA, *CUDA Programming Guide*, oct 2010. [cited at p. 7]

[9] "Wikipedia article with gflops specs for intel i7." http://en.wikipedia.org/wiki/FLOPS. Viewed 18.11.2011. [cited at p. 8]

[10] Nvidia, *Fermi Compute Architecture Whitepaper*. [cited at p. 10]

[11] C. T. Kelley, *Solving Nonlinear Equations with Newton's Method*. Philadephia, PA: Society for Industrial and Applied Mathematics, 2003. [cited at p. 10]

[12] D. M. D.J. Walton, "A controlled clothoid spline," *Computers & Graphics*, vol. 29, pp. 353–363, 2005. [cited at p. 12, 13, 14, 17, 25, 26]

[13] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, 1970. [cited at p. 12]

[14] K. D. Mielenz, "Computation of fresnel integrals. ii," *Journal of Research of the National Institute of Standards and Technology*, vol. 105, jul 2000. [cited at p. 12, 59]

[15] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in *in Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 301–308, Press, 2001. [cited at p. 15]

[16] E. Galin, A. Peytavie, B. Beneš, and E. Guérin, "Authoring hierarchical road networks," *Eurographics*, vol. 29, no. 2, 2010. [cited at p. 15, 51]

[17] I. Saltvik, "Parallel methods for real-time visualization of snow," Master's thesis, NTNU, Trondheim, jun 2006. [cited at p. 19]

[18] U. S. D. of the Interior, *Standards for Digital Elevation Models, part 2: Specifications*, jan 1998. Specifications for the USGS DEM format. [cited at p. 19, 20, 35]

[19] J. McCrae and K. Sing, "Sketching piecewise clothoid curves," *Eurographics*, 2008. [cited at p. 23]

[20] "Usgs dem." http://en.wikipedia.org/wiki/USGS_DEM. Viewed 13.10.2011. [cited at p. 34]

[21] "Mt. st. helens usgs dems." http://ned.usgs.gov/Ned/historic.asp. Viewed 31.10.2011. [cited at p. 40, 46]

[22] B. B. Mandelbrot, *The Fractal Geometry of Nature*. New York: W. H. Freedman and Co., 1983. [cited at p. 48]