DE HAAGSE HOGESCHOOL
TU Delft

RUNTIME PROCEDURAL GENERATION OF ROADS

Antony Löbker
January 2013

General information

Student	Antony Löbker 10095284
Period	2012-2.1 July 2012 - January 2013
Company	Computer Graphics and Visualization Group Faculty EEMCS, TU Delft Mekelweg 5, 11th floor Delft
Client	Dr. Ir. R. Bidarra 015 - 27 84564 R.Bidarra@tudelft.nl
Supervisor	Ir. R. V. A. Lopes 015 - 27 86347 R.Lopes@tudelft.nl
School	Haagse Hogeschool Rotterdamseweg 137 Delft
Examboard	A. B. Heyer A. Andrioli

Abstract

Procedurally generating roads is not a new thing. For years, professionals and amateurs alike have been trying to generate roads for games and simulations. However, most of these tools require a certain understanding of the programming behind the code. The game-designers that use these tools are not all that programming-savvy. The other option is to use generators that require very little input. Both are not ideal. Therefore, a new kind of system must be created that allows designers to generate roads in a way that is easy to understand. Also, this system must perform at runtime, so the designer does not have to wait very long or generation can occur during the game.

To find a good algorithm, two separate options (a simple sequential algorithm and a more complex A* algorithm) have been evaluated and implemented. Control over these algorithms is handled by the cost function, which consists of these parameters: distance, slope cost, smoothness, dither and uncertainty. By setting the weights for these parameters, a desired road is created. To allow for easier control, the user can define secondary parameters that directly influence the weights for the cost function.

Preface

This report gives a detailed account of the process I went through, regarding my graduation project for the Hague University, from July 2012 until January 2013.

My last internship was in the robot-lab of 3ME of the TU Delft. There, I noticed that I feel comfortable in a university environment. This motivated me to do my graduation project at a university too, only this time something a little more in computer science, which is my background.

I have long been interested in procedural content generation, which in my eyes is a combination of a creative and a technical aspect. This, combined with the fact that I am a former computer science student at the TU Delft, caused me to apply for an internship at the Computer Graphics and Visualization Group. My goal was not this specific task, but rather the chance to work with procedural content generation.

The professor there responded very positive and I received the project proposal for this assignment. This seemed very fun to do and that is how I ended up here.

Antony Löbker, January 2013

List of terms

- 3ME – Faculty for maritime, materials and mechanical engineering.
- Connexity – The amount of neighbours a certain point has.
- EEMCS – Faculty for Electrical Engineering, Mathematics and Computer Science.
- Heuristics – An estimate cost from going from a certain point to the end of the path.
- Index mapping – Technique used to store a matrix into an array.
- Kibibyte – 2^{10} bytes. Often (incorrectly) referred to as kilobyte, which is 10^3 bytes.
- Manhattan distance – Distance between two points, measured by going along the axes.
- Non-negative least squares method – Method of finding weights for parameters in a linear equation.
- Off-line – Used in procedural content generation. Indicates that content is pre-generated.
- On-line – Used in procedural content generation. Indicates that content has to be generated when needed.
- PRM – Probabilistic roadmap. See section 6.2.2 on page 26.
- Serious games – Game worlds not intended for entertainment, but rather to educate or train.
- Tie-breaker – A small random value, added to make sure that two nodes never have the same value.

Summary

The Game Technology-department has been working with procedural content generation for several years. They now have expressed a wish to be able to generate roads on-the-fly, so that it can be used in games. The assignment is to generate a realistic road on-line and in an efficient way. A new procedural method is to be implemented, of which the output is a path in three-dimensional space. The result of this procedural method should be dependent on user defined input.

Development was done by means of an iterative development method. Here, the development of the pathplanner (for the generation of the road) and a GUI (for the visualization of the roads) was split into several iterations. The last few iterations were for fine-tuning the pathplanner.

The first step to undertake was performing a literature study.

Next, the requirements and found techniques were used to make a design. Speed is an important factor for on-the-fly generation. Unfortunately, this is often at the expense of quality. To subvert this problem, a decision had been made to develop two different algorithms. The first one is a simple sequential algorithm that keeps going forward and doesn't backtrack. The second one is a more advanced shortest-path algorithm; it will search for a global optimum. Both algorithms were developed and tested. Afterwards, these two algorithms were compared and a decision was made, which one gave the best results.

For the sequential algorithm, the decision was made to create a variation of the algorithm by Sun et al [Sun 02]. The algorithm is simple and fast, but makes some bad decisions. Therefore, a variation was made that subverts these problems.

For the shortest path algorithm, two options were available: A* and Probabilistic roadmap. Because the sequential algorithm is very speed-oriented, the shortest path algorithm can be more result-oriented. This means not only in quality of the result, but also consistency. In that respect, A* has a much more consistent result. This is in contrast to the highly fluctuating results of PRM. Aside from that, the A* algorithm can have some small optimizations. Because of all this, the decision was made to implement the A* algorithm.

When generating a road, the user wants it to behave a certain way. Therefore, actions that are desired should be encouraged and actions that are not desired should be punished. When selecting a new point, some cost calculations are done. The cost calculations are given weights and then combined, so a desired path can be created. The main parts for the cost calculation are *distance*, *slope cost*, *smoothness*, *dither* and *uncertainty*. The exact values of these costs are arbitrary, but it is mostly the proportions which determine if a point is viable or not.

Some of the cost functions are not very intuitive, but it is not easy to define more familiar terms such as fun, speed and realism into a formula. To make this easier for users, a new kind of parameter (called secondary parameter) was created that influences the weights of multiple cost function parts. The dependency between these parameters can be defined by the user.

Now that the software was working, the next iteration in the development process was to improve the efficiency of the software. An attempt was made to improve the pathplanners in both speed and quality. Setting the heuristic function is important to get good results from the star planner. However, the number of parameters needed to create an estimate has hardly any influence on the result. Another idea to increase the speed of the algorithms was to applying some simple tricks. However, this turned out to be very disappointing.

During testing, the goal was to find out two things: what is the speed and scaling of both algorithms and how do they respond to user input. The sequential planner is faster in virtually all cases. Furthermore, the sequential algorithm grows linearly, while the star algorithm is a second order polynomial. For testing the control, several different situations were taken to see how the algorithms responded. The sequential planner is much more responsive, but at the cost of optimality and realism.

Because the purpose of this assignment is to find a pathplanner which can be used in games, speed and control are very important. Therefore, the best option here is the sequential algorithm.

Contents

Chapter 1: Introduction

In the last few decades, there has been an enormous growth in the complexity of computer games. Worlds are getting ever larger and more complex and game-developers are having trouble keeping up with the immense workload. They have to rely on techniques which allow easier and faster ways to create worlds. This is called *procedural content generation*.

Despite all these improvements, creation of roads is often still done by hand. A level designer is spending weeks or even months to make a map, then roughly placing it in the level and do the finishing, like building bridges and rounding bends. The automatic generation of roads can take away a big part of that load.

Procedurally generating roads is not a new thing. For years, professionals and amateurs alike have been trying to generate roads for games and simulations. However, most of these tools require a certain understanding of the programming behind the code. The game-designers that use these tools are not all that programming-savvy. The other option is to use generators that require very little input. Both are not ideal. Therefore, a new kind of system must be created that allows designers to generate roads in a way that is easy to understand. Also, this system must perform at runtime, so the designer does not have to wait very long.

The next chapter will describe the company where this project is performed. Chapter 3 will formulate the problem and the assignment. Chapter 4 describes how this problem is approached. In chapter 5 a literature study is performed to find out what has already been done in this field. In chapter 6 the software is designed and implemented. In chapter 7 an attempt is made to find several improvements for the software. Chapter 8 will discuss testing and the results of those tests. Chapter 9 will make a conclusion about which of the algorithms is better suited and if it meets the requirements. Chapter 10 will talk about and future work.

Appendix A shows some diagrams, which are essential when developing a software system. Appendix B will show some features of the 2D viewer. Appendix C gives a full description of competencies chosen in section 3.5. Appendix D contains a developers manual, which is useful for developers that want to use this software.

Chapter 2: Company

The Delft University of Technology is an internationally known university with about 16000 students. It is divided into several faculties on a campus. One of those faculties is Electrical Engineering, Mathematics and Computer Science (EEMCS), consisting of electrical engineering, mathematics and computer science. EEMCS has about 1600 students, 400 PhD-students and 400 other staff. The building is very recognizable for the TU Delft and can be seen in figure 2.1.

Figure 2.1: View of the EEMCS building. Source: [1]

On the 11th floor of the EEMCS building is the Computer Graphics and Visualization Group. There they work in different areas, such as rendering, visualization of (scientific) data and modeling of 3D objects. Several courses related to these areas are also taught. At the moment, the most important research areas are Interactive Visualization and Virtual Reality, Game Technology and Medical Visualization. An organizational tree is shown in figure 2.2.

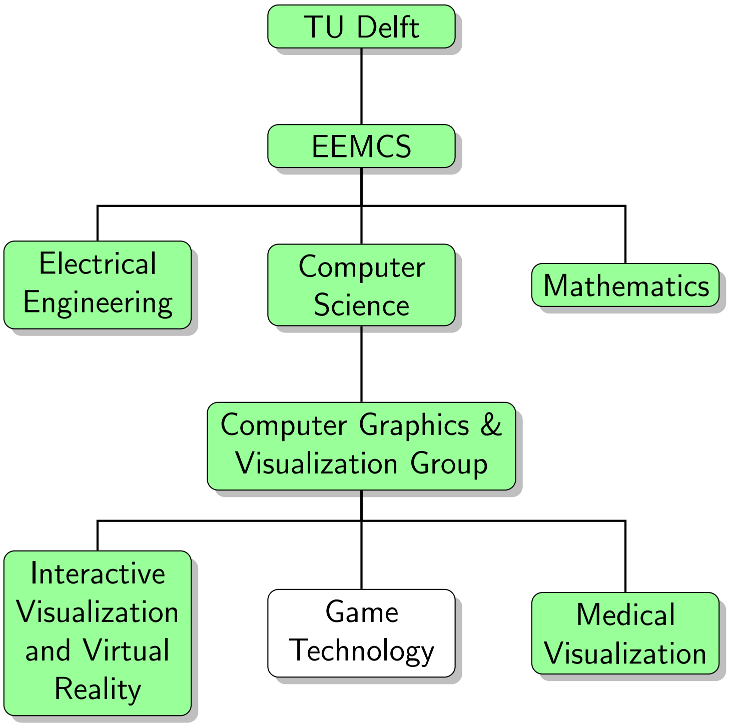


Figure 2.2: Organizational chart

This assignment was performed at the Game Technology-department, under supervision of Dr. Ir. R. Bidarra and Ir. R. Lopes.

Chapter 3: Assignment

3.1 Problem setting

The Game Technology-department has been working with procedural content generation for several years. Lots of projects have been researched, such as being able to generate terrains, rivers, buildings and cities. They now have expressed a wish to be able to generate roads on-the-fly, so that it can be used in games. These roads are predominantly highways and rural roads.

3.2 Assignment description

The objective of this assignment is to generate a realistic road on-line and in an efficient way. A new procedural method is to be implemented, of which the output is a path in three-dimensional space. For this to work, a terrain has to be supplied and a start- and endpoint have to be defined.

The client has stated that the software has to be written in C#.

Aside from the procedural method, a user interface has to be made for visualization of the road. This interface will allow for certain parameters to be set.

Where this project differs from other road generation projects, is the high degree of control over the generation. By changing several input parameters, the results can be very different. These input parameters are divided into two types:

- Geometric parameters that indicate, for example: inclination, amount and sharpness of bends, forking and possibly others.
- Gameplay parameters (based on the geometric ones) which indicate, for example: speed, visibility, realism and fun.

Typically, gameplay parameters will be used and reused often. Geometric parameters exist to allow the specification of gameplay parameters.

Additionally, the new method should be highly efficient and optimized for on-line performance.

Afterwards, the planner has to be incorporated into a DLL library, so other projects can easily use it.

3.3 Problem statement

Develop an efficient algorithm that generates a road at runtime using user defined (and gameplay-oriented) input and presented in a standardized format.

3.4 Project limitations

Generating a model and/or texture of the road falls outside the scope of this project.

The road must always be allowed to continue. Walls or otherwise blocked paths are not permitted.

The time that can be spent on this project is limited. Therefore, fine tuning the software cannot continue infinitely.

3.5 Competencies

During the graduation-process, the Hague University requires its students to select a number of aspects in their assignment, to which they give some extra attention. This is to show that they are ready for employment.

The selected competencies for this assignment are:

- C8 – Designing a technical information system
- D16 – The realization of software
- H5 – Working professionally: Working independent
- H6 – Working professionally: Working result-oriented

These are all explained in appendix C (Dutch).

Chapter 4: Approach

Now that the assignment is clearly stated, the approach can be determined.

4.1 Development method

First, it is important to determine some requirements for the development method.

- There is only one developer working on the project.
- The size of this project is fairly large.
- Delivery of the library is only needed at the end of the project.
- There will be regular feedback-moments.
- There are no set test-cases, the quality is subjective.

The feedback makes development methods such as linear programming and the waterfall model not applicable. Likewise, test-driven development is not suitable either, because the quality cannot be determined by a simple true/false check. Because the library only needs to be delivered at the end of the project, no increments are needed, so RUP and the spiral model are not the best options either.

The most promising of the remaining development methods is Agile. It is an iterative development method. This means that development is done in cycles, but the product is only delivered at the end. An example of this is shown in figure 4.1.

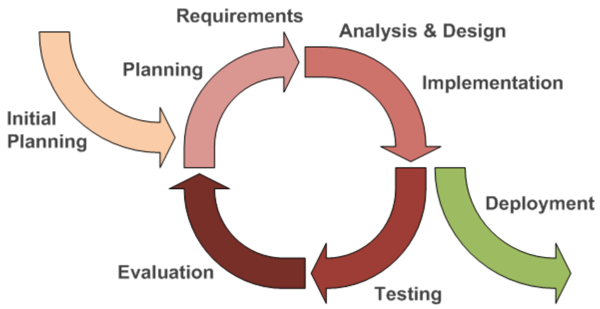


Figure 4.1: Schematic display of an iterative development method. Source: [2]

4.2 Planning

The development of this project is divided over several iterations.

First iteration

In the first iteration, a design is made and then implemented.

First, a literature study is performed to look for existing techniques that can help with the assignment. Next, the requirements and found techniques are used to make a design. It has to be determined what algorithm to use, how the cost functions are defined and how the software is built. After that, the software is implemented.

Second iteration

The second iteration will consist of building a user interface for easy visualization and testing of the algorithm.

Third iteration

In the third iteration, an attempt is made to improve the algorithm in terms of speed and accuracy.

Later iterations

Subsequent iterations will deal with feedback from the client and supervisor.

Last iteration

When the software is completely finished, the algorithm is stress-tested. This is done at the end, because measurements performed on an algorithm that is not finished can be completely different to those at the end of the project.

Chapter 5: Literature study

5.1 Procedural generation

Procedural generation is a way to automatically generate content using algorithms. The primary reason is being able to create a lot of variation without a developer spending (expensive) man-hours.

In procedural generation there is a distinction between two forms. The first form is called *online* (this has nothing to do with the Internet). On-line generation indicates that the object does not yet exist at the start of the program, therefore this should be calculated by the computer. As a result, the world looks different each time. This can also be called *runtime*. The other form is *off-line*. The idea behind this, is that the developer itself generates the content and then imports it into the program. This also means that during distribution all of the generated content has to be sent with the program. On the other hand, the game world will look more consistent.

The most well-known example is *SpeedTree*. This is a software library, developed by a company called *Interactive Data Visualization*, which allows for easy and fast generation of trees. As a result, a 3D artist does not have to spend time making hundreds of different trees. Speedtree is used in lots of commercial games. The editor can be seen in figure 5.1.

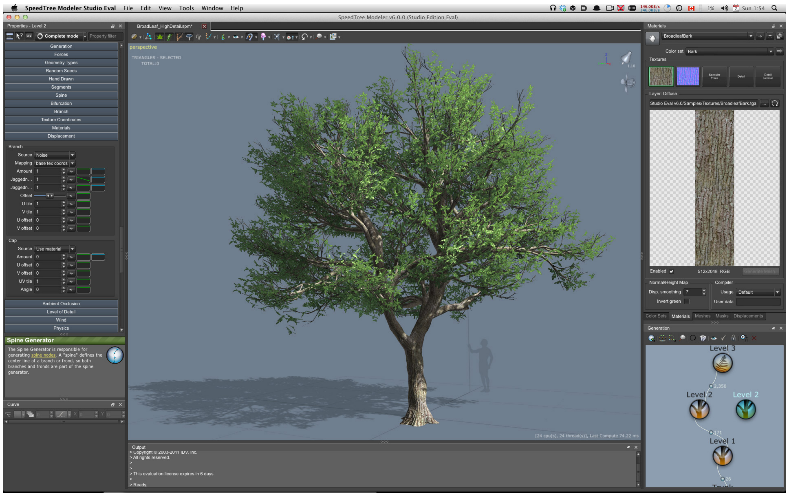


Figure 5.1: A screenshot of the SpeedTree editor. Source: [3]

Another example is the game *.kkrieger*. This is a shooter game that attempts to use procedural methods to the extreme. The most noticeable aspect of .kkrieger is that the whole game is 95 KiB (kibibytes) in size. This is a factor of 40.000 smaller than most current games, which are about 4 GB.

5.2 SketchaWorld

SketchaWorld [4] (promotional image shown in figure 5.2) is a program that allows easy game world creation. It was developed by Ruben Smelik who was in the Computer Graphics & Visualization Group (of the TU Delft) at the time. The program incorporates a large number of techniques for procedural content generation. The code of this program is a combination of C# and C++ classes, which all work together.

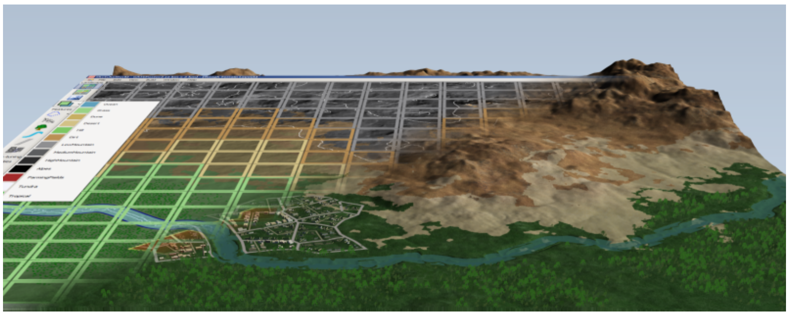


Figure 5.2: SketchaWorld. Source: [4]

Creation of a world starts with a flat terrain covered with a grid. The squares on the grid can be coloured with brushes, such as water, grass, mountains and desert. After painting, the landscape is automatically generated. After that, thing such as rivers, forests, roads and cities can be added.

At the moment, the program is used for designing *serious games*. For example, there is a project that trains levy inspectors and another where worlds are used for military simulations. SketchaWorld is not publicly available.

5.3 Papers

Template-Based Generation of Road Networks for Virtual City Modeling

The paper *'Template-Based Generation of Road Networks for Virtual City Modeling'* [Sun 02] uses templates to generate road networks. These determine where the beginning and end of a road are going to be. Generating the road itself is a fairly simple algorithm (discussed in Sun [Sun 02, Sec 3.2]). From a certain point, we look in the direction of the endpoint. We will then select a new set of points. Each new point is at the same distance and with the same relative angle between each other. Then we evaluate which of these points meets the requirements closest, and that is the new point to look at. When arriving at the end point, the path is complete.

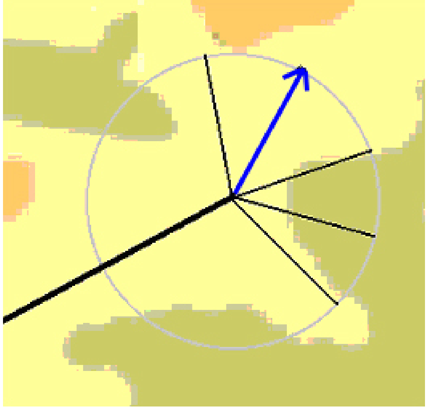


Figure 5.3: Point selection of Sun et al. [Sun 02]

Figure 5.3 shows an example situation. The black line on the left represents the road segments that have already been generated. The five lines on the right are neighbours and possible new segments. All these neighbours have the same distance from the center-point. Currently, the blue line is being evaluated. In the background, green areas indicate low terrain, yellow areas indicate medium terrain and orange areas indicate high terrain.

Citygen: An Interactive System for Procedural City Generation

'Citygen: An Interactive System for Procedural City Generation' [Kelly 07] compares the algorithm by Sun et al to several others. In a later section [Kelly 07, Sec 4.2], an extension is made to the algorithm of Sun et al by starting the path from two sides and combining them in the middle. In addition, the choice of the optimal point has been changed to selecting the point with the smoothest transition. The author also created an editor and explain how roads can be created and edited.

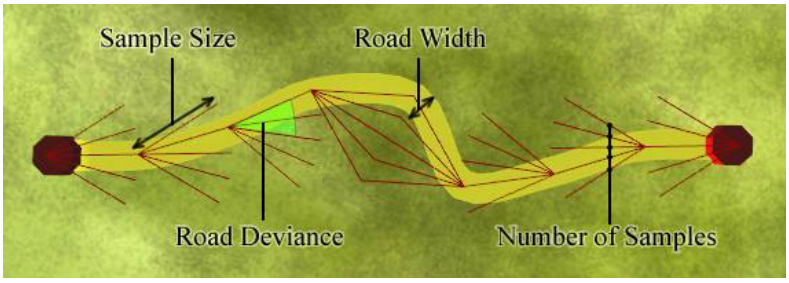


Figure 5.4: Building a path according to Kelly [Kelly 07]

Figure 5.4 shows an example of how a road could be constructed with Kelly's algorithm. Segments are created from both sides and work towards each other. In the middle they combine to form a road. The dark red points mark the start and end point. The yellow ribbon indicates the final road. Other parts have been marked in the image.

Procedural Generation of Roads

'Procedural Generation of Roads' [Galin 10] is a big inspiration for this assignment. They use an algorithm called *A** for finding a path on a grid. Each line gets a weight, which is a composite of several factors. These factors are curvature, water depth (for bridges), slope, tunnel cost, excavation and embankment. They also make use of bridges and tunnels, which gives quite realistic results. *Stochastic sampling* (removal of random points) is used to reduce the amount of computations. Another noteworthy feature is something best described as terrain modifying cost. When building a tunnel, for example, a certain amount of the terrain needs to be excavated so the tunnel has a smooth entrance and exit. This can be seen in figure 5.5.

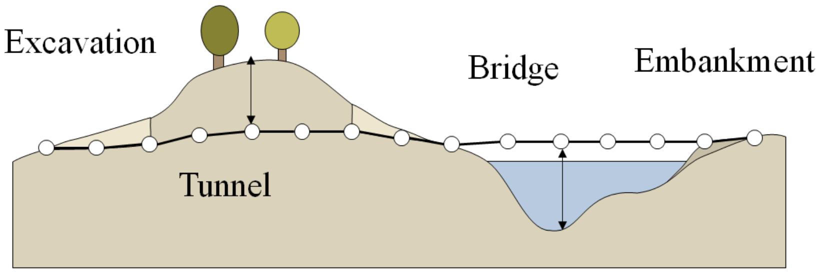


Figure 5.5: Examples of excavation and embankment by Galin et al. [Galin 10]

The paper *'Authoring Hierarchical Road Networks'* [Galin 11] is a continuation of the research in the earlier paper by Galin et al. Here they expand on the generation of roads to create an entire road network of highways and rural roads.

5.4 Bresenham's line algorithm

Selecting sections of road is a computationally intensive task. In order to limit this somewhat, large line segments are used. However, sometimes it is important to know what is happening in the intermediate squares of the line segments. Therefore, the points that lie on the line segment must be found. This is done by using Bresenham's line algorithm [Bresenham 65].

The algorithm starts with orientating the line correctly. The algorithm is designed for gradual slopes, so with a steep line, the x-axis and y-axis have to be swapped. After that, we follow the line from left to right.

We store the value of y and follow the line from left to right on the x-axis. During each step, we check whether the distance between the line and the stored y-value is less than 0.5. In case the distance is greater than or equal to 0.5, the line is closer to the pixel on the next line. In that case, we increase the y-value with 1. If not, we do not change the value. The new x- and y-coordinates are part of the line segment. An example of this can be seen in figure 5.6.

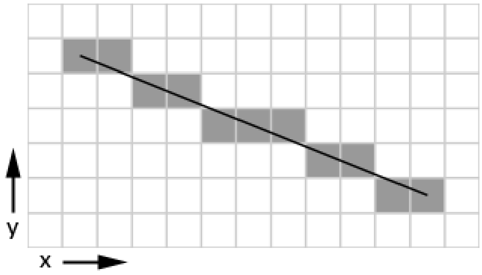


Figure 5.6: Bresenham's line algorithm. Source: [5]

Chapter 6: Designing the algorithm

Speed is an important factor in runtime generation. Unfortunately, this is often at the expense of quality. This is a problem, but it can be subverted. In the literature study, a number of different algorithms have been presented. Developing two of those algorithms, comparing them afterwards and picking the most viable option will result in a good pathplanner. However, these two algorithms will have to differ substantially if a comparison is to reveal big differences.

The first algorithm to be implemented is a simple sequential algorithm that keeps going forward and doesn't backtrack. The second one is a more advanced shortest path-algorithm, which will search for a global optimum.

Both algorithms will be implemented and tested. Afterwards, these two algorithms will be compared in terms of speed and control. A decision will be made to select the one which gives the best results.

6.1 Sequential algorithm

For the sequential algorithm, the best candidate is the algorithm by Sun et al [Sun 02], mentioned in section 5.3. The algorithm is simple and fast. However, it makes some bad decisions. Therefore, the decision was made to create a variation of the algorithm.

Problem

A great disadvantage of the current algorithm is that it makes some poor choices. When creating a new segment, a number of points are evaluated, and the best one is selected. Then, the point is moved in the direction of the end point, in order to ensure that the path does not get astray or overshoots.

Unfortunately, the newly selected point isn't necessarily the best option anymore. As a result the path may be full of steep slopes or sharp turns. For example, when moving next to a cliff, the selected neighbour might be on the same level, but the shifted point might be on top of this cliff, resulting in a big height difference. An example of this can be seen in figure 6.1.

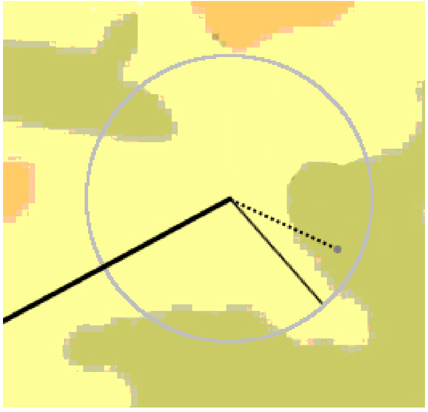


Figure 6.1: A problem in the point selection by Sun et al. The selected neighbour might not be the best option any more. The grey dot represents the shifted endpoint and the dotted line represents the new road segment.

Solution

For this implementation, the problem was solved by not adjusting the points themselves, but rather the angle in which the points can be selected. This angle is dependent on the direction and distance to the endpoint. To prevent the road from drifting away, the points that can be selected are getting closer to each other as the path approaches the endpoint. When the road is just starting, the neighbours are selected in an angle of almost 180°, near the end, this can be reduced to less than 30°. Figure 6.2 shows this.

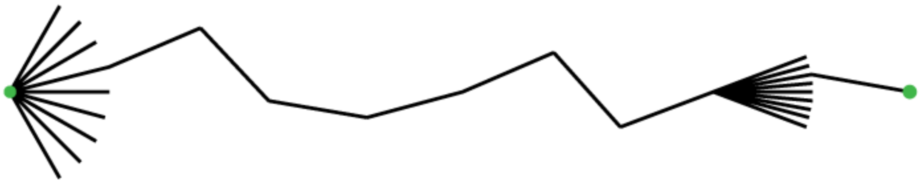


Figure 6.2: Schematic display of the point spread as you approach the endpoint. The fan on the left indicates the selection-angle when starting. The fan on the right indicates the selection-angle near the end of the road.

UML

The workings of the algorithm can now be turned into a dataflow diagram, seen in figure 6.3. First, the path array is cleared to make sure that there are no points remaining from a previous road. Then the total distance is calculated, which is needed to calculate the point spread. We get the last point, in this case is the starting point and see if we are near the end. If this is not so, we determine the angle of selection and find all corresponding neighbours. By applying the cost function to all neighbours, we find the best one and add it to the path. After that, we select our newly added point and continue from there.

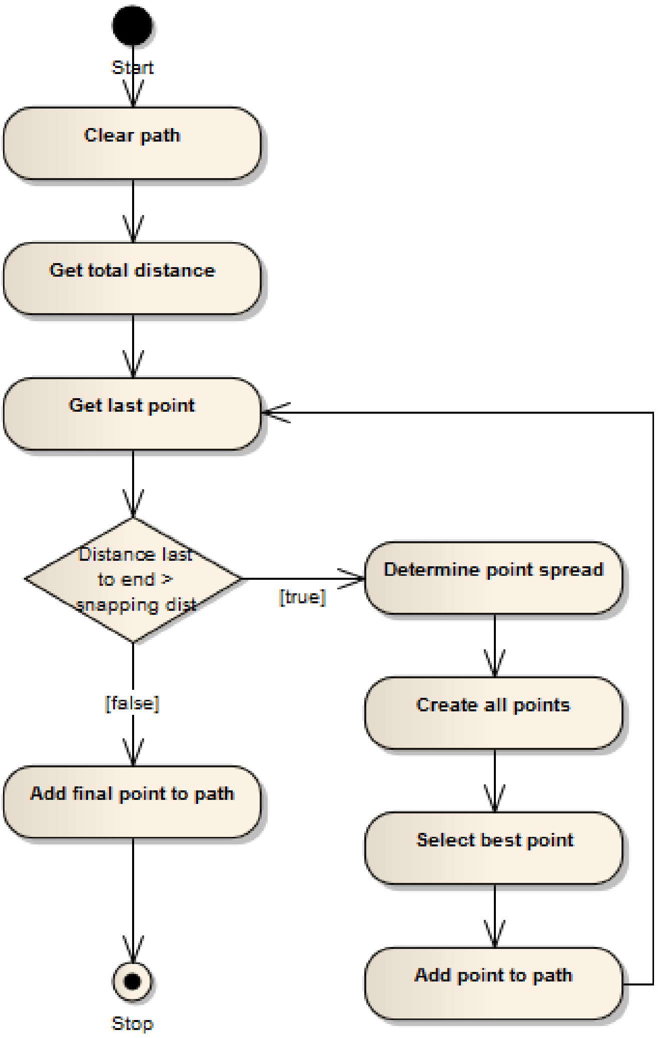


Figure 6.3: Dataflow diagram for the sequential algorithm

6.2 Shortest path algorithm

For the shortest path algorithm, there are two options: A* and Probabilistic roadmap. They will be described and then compared.

Note: The term shortest path does not describe the actual shortest path (in distance), but the path with the lowest cost.

6.2.1 A* algorithm

A* is an algorithm to find the shortest path between two nodes on a grid. It is a so-called *best-first algorithm*, because it chooses to visit the most promising nodes first, using heuristics to do this. It is an extension of Dijkstra's shortest path algorithm. It is based on the A2 path-finding algorithm, which in itself is based on the A1 path-finding algorithm.

The most important part of this algorithm is the cost calculation, consisting of three parts. Figure 6.4 gives an example of these parts. The first part is a value G, this indicates the actual cost to get from the starting point to the current node. The second part is a value H, which gives an estimate of the cost for going from the current node to the endpoint. It is important that this estimate is a good approximation of the real value, because this affects the algorithm. Finally, there is a value F, which is simply the sum of values G and H. This value is used to determine which of the nodes is the most promising.

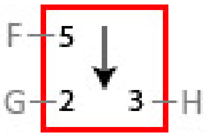


Figure 6.4: One block of the A* example. The bottom left value represents G, the bottom right H and the top left F. Source: [6]

Figure 6.5 gives an example situation. Each square contains the F, G and H values. The values are measured in *Manhattan distance*. The black dot indicates the starting point. The blue square (marked with X) indicates the endpoint. The red squares are part of the closed list and the green squares part of the open list.

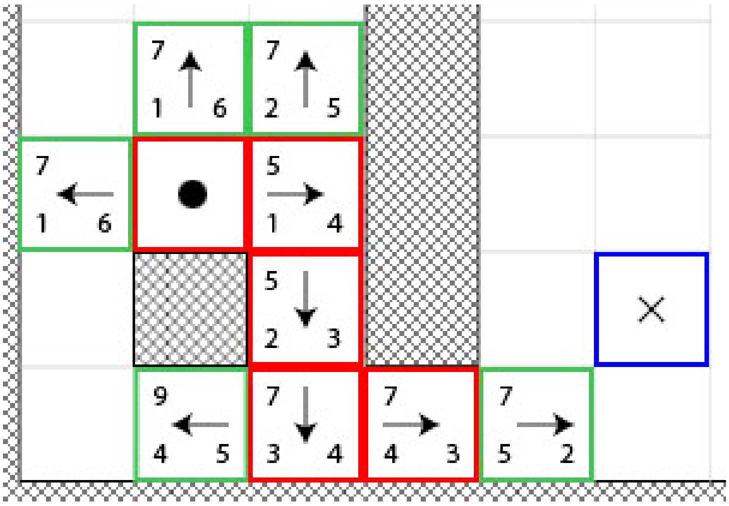


Figure 6.5: A simple example of the A* algorithm. Source: [6]

The algorithms begins by creating two lists: the closed list, which will contain nodes that have already been explored, and the open list, which will contain nodes that are relevant to evaluate. Both lists are still empty at this point. We add the starting point to the open list. The following is repeated until the endpoint is reached or the open list is empty:

We pick the node with the lowest F value from the open list and add it to the closed list. From this node, we take all its neighbours, and we check if each of the neighbours are in the closed list. If so, we have seen the neighbouring node before and can throw it away. Otherwise, we check whether the neighbouring node is in the open list. If not, we add it to the open list. If it is, we check whether the path from the current node is cheaper. A representation of handling neighbours can be seen in algorithm 1.

Algorithm 1 Pseudo code for the handling of neighbours.

```
for all $neighbours$ do
    if $neighbour$ in $closed\ list$ then
        continue
    end if

    if $neighbour$ not in $open\ list$ then
        add $neighbour$ to $open\ list$
    else
        $new\_g = my\_g + stepcost$
        if $new\_g < old\_g$ then
            update $neighbour$
        else
            continue
        end if
    end if
end for
```

In the standard form of the A* algorithm (only a grid), there often are multiple best paths. To prevent the algorithm from following each path, a small random deviation is inserted for each step. The deviation is about 0.1% of the cost. This principle is called a *tie-breaker*.

The importance of a good H-value is discussed in section 6.5.

Selection of nodes

An important consideration between quality and speed, is to determine how many neighbouring points the A* algorithm selects each cycle. This is called *connexity*. Choosing more points often produces better results, but also costs more computing power. All the points lie on a grid, so selecting the amount of points can easily be discretized.

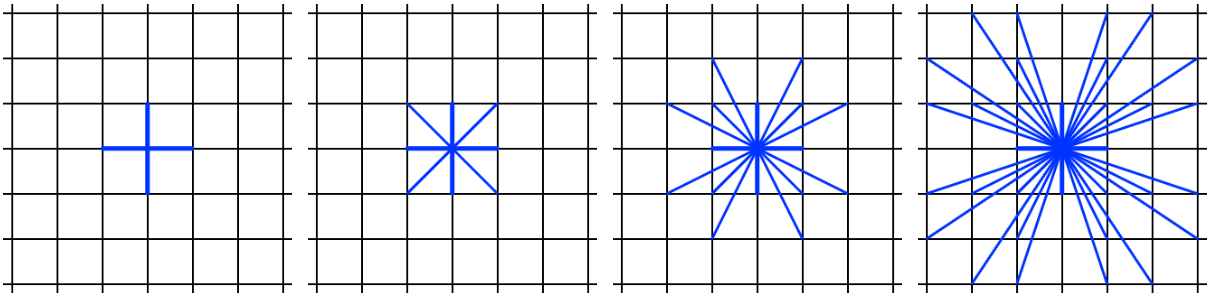


Figure 6.6: 4-, 8-, 16- and 32-connexity

The amount of connexity indicates which points are being selected. Figure 6.6 shows a number of options. The example in figure 6.5 uses 4-connexity.

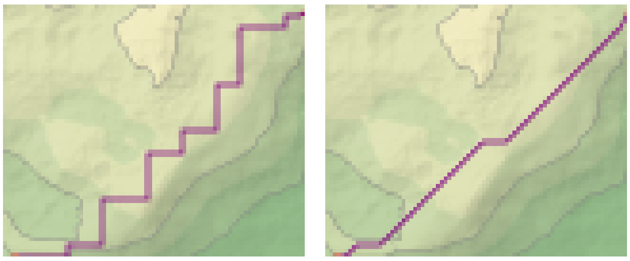


Figure 6.7: Paths with different connexity. The one on the left has 4-connexity and the one on the right has 8-connexity.

Figure 6.7 shows a clear example how the connexity influences the result. The road on the left is much longer than the road on the right. Also, it looks less realistic.

6.2.2 Probabilistic roadmap

Another option for a shortest path function is the *probabilistic roadmap*.

The probabilistic roadmap (PRM) planner is an algorithm, most commonly used in robotics, to control a robotic arm. PRM consists of two parts: a construction phase and a search phase.

In the construction phase, random points are selected in the space. If a point intersects with an object, it is removed. All remaining points are then connected to nearby points, provided the line does not intersect with an object. Thus, a graph (roadmap) is made.

In the search phase, the starting point and the endpoint are added to the graph. The path is found by applying Dijkstra's shortest path algorithm to the graph.

PRM can also be used for finding a path in games. Points are placed all over the map and connected. An example is shown in figure 6.8.

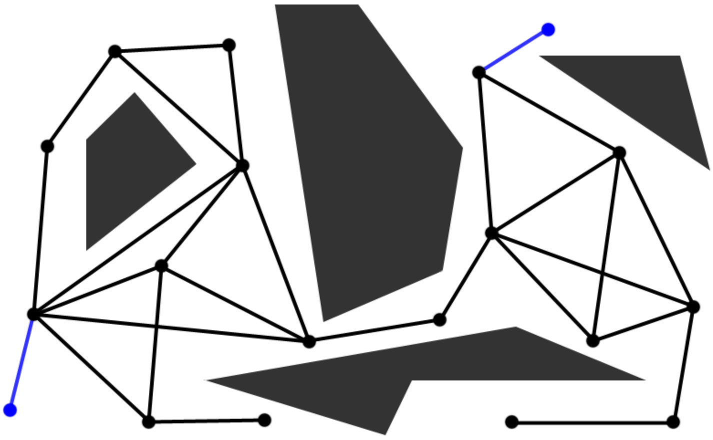


Figure 6.8: The construction phase of a probabilistic roadmap. The black areas are impassable, e.g. water.

6.2.3 Assessment of algorithms

Because the sequential algorithm is very speed-oriented, the shortest path algorithm can be more result-oriented. This means not only quality of the result, but also consistency. In that respect, A* has a much more consistent result. The grid is always in the same place and points are evenly distributed. This is in contrast to the highly fluctuating results of PRM.

Aside from that, the A* algorithm can have some small optimizations. Firstly, A* can be optimized by only creating a point-object when it is needed. This is in contrast to PRM which creates a large amount of points at the beginning. Secondly, PRM creates its point all over the terrain. If a road has to be created in a small area, a lot of points will be obsolete.

Because of all this, the decision was made to implement the A* algorithm.

UML

Now that it is clear that the A* algorithm will be implemented, some other decisions can be made.

For the A* planner there is a different way of storing its information, it uses nodes instead of points. A node has an additional property where it knows what its previous node in a path is. Additionally, these nodes are stored in a dictionary. A dictionary is a type of hashtable that has an index and an object. In this case, the index can be calculated with a process known as *index mapping*. This value is calculated like this:

$$index = y * width + x$$

Additionally, a dataflow diagram can now be made. It is presented in figures 6.9. However, because the diagram was to big, the rebuildPath-function is summarized and shown separately in figure 6.10.

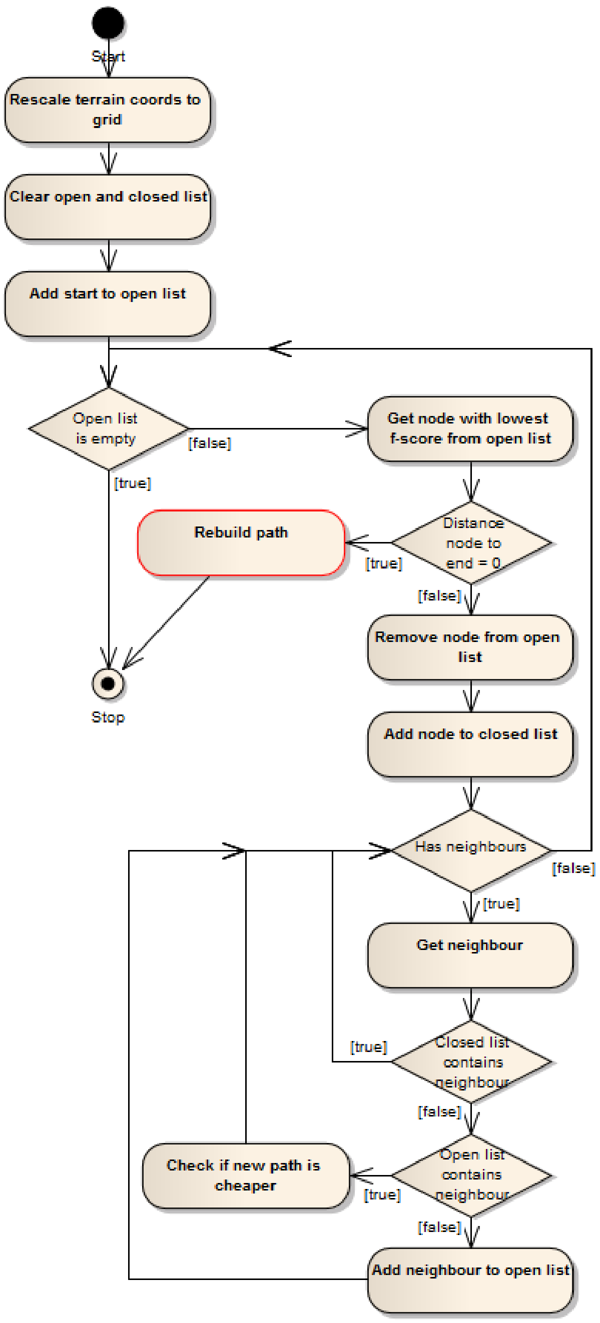

Start
Rescale terrain coords to grid
Clear open and closed list
Add start to open list
Open list is empty
[false]
[true]
Get node with lowest f-score from open list
Distance node to end = 0
[true]
Rebuild path
[false]
Remove node from open list
Add node to closed list
Stop
Has neighbours
[false]
[true]
Get neighbour
Closed list contains neighbour
[true]
[false]
Open list contains neighbour
[true]
Check if new path is cheaper
[false]
Add neighbour to open list

Figure 6.9: Dataflow diagram star

The first step is to calculate a multiplication factor to go from terrain-coordinates to grid-coordinates. Next, we make sure that the open and closed list are empty. We add the starting node to the open list. Because there is now only one node in the open list, we take it out again and see if we are near the end (probably not yet). Now we move the node to the closed list and start checking all its neighbours. For each neighbour we check if they are either in the closed list or the open list and act so accordingly. If all neighbours have been checked, we continue with a new node. This continues until we are at the end. Then the path is rebuilt and we are done.

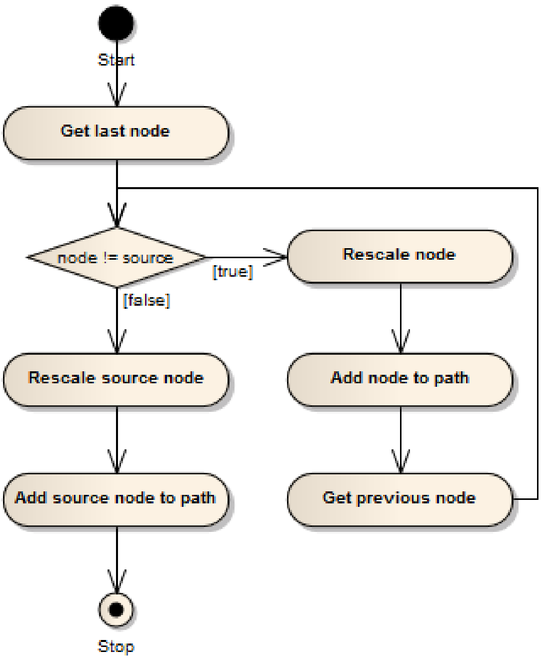


Figure 6.10: Dataflow diagram rebuild path

Figure 6.10 shows how a path is rebuilt. Nodes only have information about their previous node. Therefore, making it into a path has to be done backwards. We start by taking the last node. Currently this node is working with A*-grid-coordinates, so it needs to be turned into terrain-coordinates. Then we add the node to the path and take the previous one. This continues until the whole path has been rebuilt.

6.3 Cost function

When generating a road, the user wants it to behave a certain way. Therefore, actions that are desired should be encouraged and actions that are not desired should be punished.

When selecting a new point, some cost calculations are done. The exact values of these costs are arbitrary, but it is mostly the proportions which determine if a point is viable or not. The cost calculations are given weights and then combined, so a desired path can be created. The parts of the cost function are the geometric parameters.

Now, the parts of the cost function will be described individually. First, the desired result is stated. Then, a solution is given. After that, a formula is defined for calculating the cost. Finally, the result of the formula is shown.

6.3.1 Distance

Desired result

Going directly from A to B will often lead to some boring results. It is desirable for the road to snake its way to the endpoint and not go in a straight line. So the road is made several times longer than the actual distance and an attempt is made to squeeze it between the start- and endpoint. The distance parameter is used to define this length.

Workings

The distance parameter works by simply punishing the road if goes ahead of schedule or falls behind schedule. The actual formula, however, is a little more complicated. The road should be evenly distributed, which means that if half the road has been generated, it should be roughly halfway between the starting point and endpoint. Getting too far ahead should increase the cost and lagging too far behind should also increase the cost. An example is shown in figure 6.11.

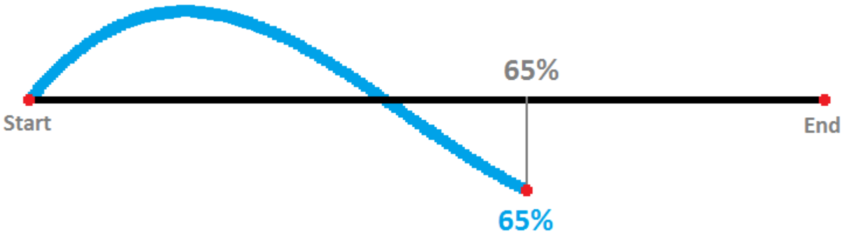


Figure 6.11: Example situation where the total length of road matches the projected progress.

In the figure, the blue line indicates the part of the road that has been built already. The length of this part is 65% of the desired path length. Also, when this value is projected on the black line, the projected point is 65% across from start to end. This is a matching situation and thus desirable.

Formula

To see how far the road is, we take the length of the road that has been generated so far and divide it by the desired distance. To find out how far the road should have been, we project the current point on the line that goes from the starting point to the end point and divide that by the length of that same line. Both these calculations return a percentage. Subtracting these gives a difference which indicates how far the road is deviating form where it should be. However, a small deviation is acceptable, while a large deviation should be punished severely. Therefore, the cost is increased by a power of 4 when going too far ahead and a power of 2 when falling behind.

The formulas for this are:

$$deviation = \frac{projection}{dist_{direct}} - \frac{road\ length}{dist_{desired}}$$

$$cost = \begin{cases} \dfrac{deviation^4}{stepsize * factor} & \text{if } deviation > 0 \\ \dfrac{deviation^2}{stepsize * factor} & \text{if } deviation < 0 \end{cases}$$

Where $projection$ is the distance from the starting point to the current point projected on the line, $road\ length$ is the length of the road until now, $deviation$ is the difference from the actual progress to the planned progress, $stepsize$ is the length of a road-segment and $factor$ is the multiplication factor.

As you can see, the deviation is being divided by two parameters. The first parameter (stepsize) is to prevent the algorithms from only choosing neighbours that are far away. Selecting a neighbour that is close will result in more line segments and therefore more calls to the cost function. By dividing the deviation by the step cost, the algorithm has to find a balance between longer and shorter line segments. The second parameter (factor) is to prevent the algorithm form taking shortcuts. Sometimes the pathplanner would take a direct road to the end with very high cost for each step, but a total cost that is lower, because there are fewer steps.

Results

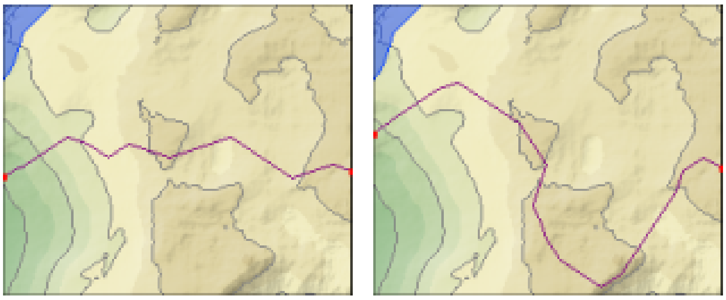


Figure 6.12: A possible result of the planner with different distance settings

Figure 6.12 shows two example situations. In the left image, the desired distance is set to 1. In the right image, it is set to 2. This example ignores the slope cost. In this case, the planner used is the star planner, because it displayed the difference best.

6.3.2 Slope cost

Desired result

The road should avoid mountain peaks, valleys and cliffs.

Workings

The line segments and the terrain have different resolutions, the terrain often has a much higher resolution. To measure the cost between two points, the intermediate points must also be looked at. These intermediate points can be found with Bresenham's line algorithm [Bresenham 65].

The cost calculation is split in to 4 sections:

- The first part (defined as p_1) is the absolute slope between the starting point and the endpoint. This was made to avoid steep slopes.
- The second part (p_2) is the difference in slope between the current line segment and the previous one. The purpose of this is to have no sudden changes in direction.
- The third part (p_3) is the average of the absolute slope of all the intermediate points. This indicates how bumpy the terrain is, which is something that should be avoided.
- The last part (p_4) is the maximum slope of the intermediate points. This works mostly the same as the first part.

A schematic representation of all these parts can be seen in figure 6.13.

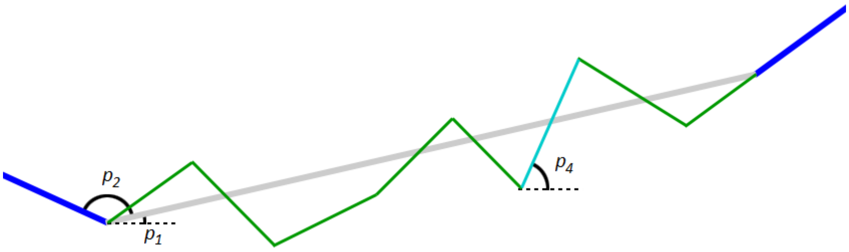


Figure 6.13: Representation of each of the parts described.

Formula

Because the costs for p_3 and p_4 are much higher, the costs for p_1 and p_2 are multiplied by the distance. This leads to the following formula:

$$dist(p_1 + p_2) + p_3 + p_4$$

where p_1 through p_4 are the respective parts and $dist$ is the distance between the starting point and the endpoint.

Result

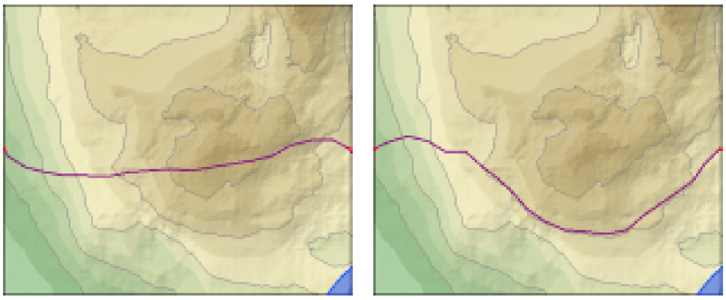


Figure 6.14: Results of the algorithm differ when the slope cost is low, to when it is high.

In figure 6.14, there are two different situations. In the first, the slope cost is very low so it will go over the mountain peak. In the second, the slope cost is much higher and it will avoid the peak. In this case, the planner used is the sequential planner, because it displayed the difference best.

6.3.3 Smoothness

Desired result

Using the smoothness, the change in direction of following line segments should be constrained more. The algorithm has to be discouraged not only to select sharp bends, but also a high amount of slightly duller bends.

Workings

This function takes the different angles between the line segments and adds these together. Figure 6.15 shows these angles.

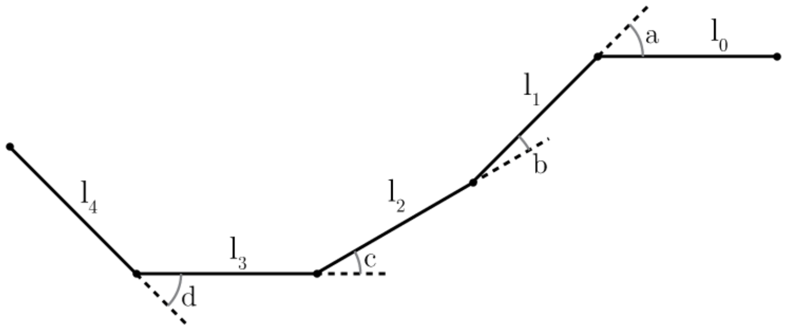


Figure 6.15: Segments l_0 to l_4 are pieces of road. Angles $\angle a$, $\angle b$, $\angle c$ and $\angle d$ are the angles between the line segments.

Formula

The cost for the angles is calculated by using the following formula:

$$a^3 + b + c + d$$

where a, b, c and d are radians in the range $[0 \ldots \pi]$.

The reasoning behind this formula is that the angle a is more important than the other angles. Also, the power of 3 insures that a value below 1 gets smaller and a value above 1 gets bigger. As a result, obtuse (dull) angles are promoted and acute (sharp) angles are punished.

Result

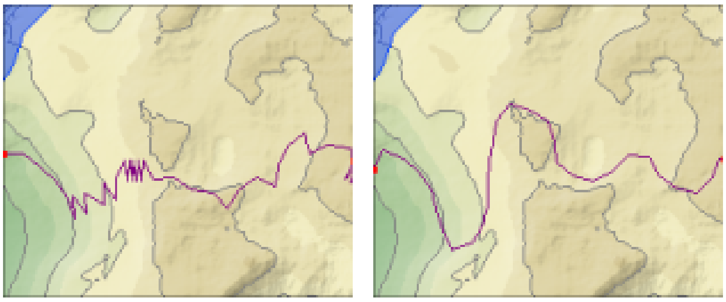


Figure 6.16: Smoothness reduces the amount and sharpness of the angles.

Two different situations can be seen in figure 6.16. The left one has several sharp turns. In the image on the right, the angles are not as sharp and there are also less of them. In this case, the planner used is the sequential planner, because it displayed the difference best.

6.3.4 Dither

Desired result

The generated roads are to be used in a gaming environment. Because of this, it can be desirable not to take the optimal route, but rather to zigzag. The dither function influences this.

Workings

Here, an attempt is made to create as much chaos as possible. Straight lines are punished and angles are promoted. For this formula uses some of the same terms as the smoothness. However, directly interfering with smoothness is not desired, so a few extra values are added.

Formula

All values range from 0 to π. Because it is desired to give a low cost to a big angle and a high cost to a small angle, this value is subtracted from π. Now the values range from π to 0.

$$(\pi - a) + (\pi - b) + (\pi - c) + \frac{2}{3}(\pi - \theta_{0,2}) + \frac{1}{3}(\pi - \theta_{0,3})$$

This can be rewritten to:

$$4\pi - a - b - c - \frac{2}{3}\theta_{0,2} - \frac{1}{3}\theta_{0,3}$$

Where $\theta_{0,x}$ is the angle between lines l_0 and l_x as displayed in figure 6.15. The reason $\frac{2}{3}$ and $\frac{1}{3}$ were chosen, is because they add up to 1, which helps in simplifying the formula.

Result

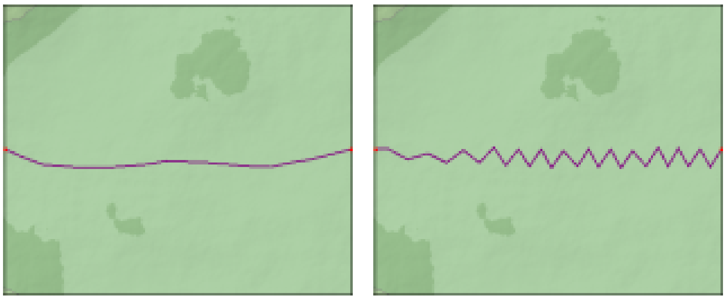


Figure 6.17: A low value produces a straight line, otherwise a zigzag pattern appears.

Figure 6.17 clearly shows a difference in results when the dither parameter is used. In this example, the sequential planner is used, because it displayed the difference best.

6.3.5 Uncertainty

Desired result

Sometimes patterns start appearing and the roads get a little boring. To fix this, a parameter was created that selects a random neighbour. This increases unexpected variability on the results.

This parameter is not part of the regular cost function, but is applied beforehand.

Workings

The user defines the chance of a neighbour being chosen randomly.

Formula

Algorithm 2 Pseudo code for determining the neighbour.

```
$r$ = random value
if $r < threshold$ then
    select random neighbour
else
    select best neighbour
end if
```

Algorithm 2 shows how a neighbour is selected. $threshold$ is a user defined value between 0 and 1.

Result

For the sequential algorithm, the effect is very visible. (see figure 6.18)

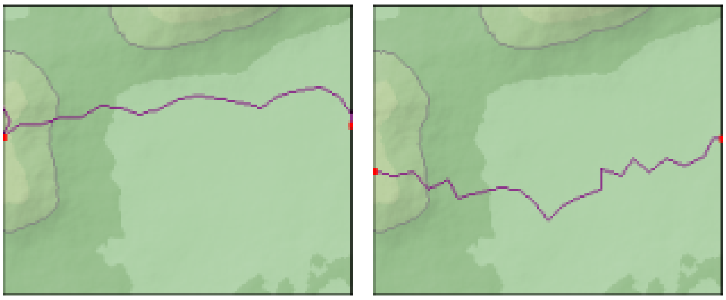


Figure 6.18: The influence of uncertainty on the sequential planner. The image on the left show a (somewhat) predictable road. The road in the image on the right is much less predictable.

However, this parameter has no influence on the star algorithm. Because of the nature of the algorithm, it cannot be told to select a certain neighbour. A neighbour can only be suggested, but the algorithm itself still has to decide if that neighbour will be chosen.

6.4 Secondary parameters

The parameters described in section 6.3 are not very intuitive, but it is not easy to define more familiar terms such as fun, speed and realism into a formula. Also, they are subjective. To make it easier and more flexible for users, a new kind of parameter was created that influences the weights of multiple cost function parts. This is called a secondary parameter and is equivalent to the gameplay parameter mentioned in the assignment description (section 3.2).

Defining a secondary parameter can be done by the user. However, a user might not know what settings will work. Therefore, a number of parameters have been predefined. These predefined ones can be edited if the user disagrees with the developers settings.

An additional benefit is that in certain cases the user only has to change one value, in stead of three or four.

An example: Speed directly influences slope cost and smoothness. When building a highway, we would like it to go straight through the landscape and not make any sharp turns. So if the value for speed is increased, the value for slope cost goes down and the value for smoothness goes up.

6.5 A*: Heuristic cost

The A* algorithm uses a heuristic cost to make an educated guess what the cost is going to be if a piece of road has to be generated. There are two important requirements for the heuristic function:

Firstly, the estimation should take very little processing time. The function is called for each node, so for a long path this can be very often. Therefore, it is important that valuable processing time is not wasted.

Secondly, the estimate should be as accurate as possible. The outcome has to be a realistic representation, because it determines which nodes are selected.

A cost estimate with a too low value causes too many alternatives to be evaluated, resulting in the algorithm taking a long time. A cost estimate with a too high value causes too few alternatives to be evaluated, which can cause the chosen path not to be the optimal path. Examples of this can be found in figure 6.19.

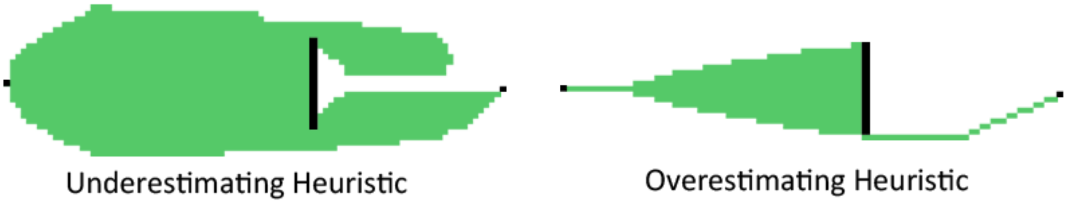


Figure 6.19: Consequences of an incorrect cost estimation. When moving from left to right (the black dots), a wall is hit halfway (the black bar), so a direct path is not possible. Source: [7]

6.5.1 Non-negative least squares method

A good way to get an accurate cost estimate, is to see if some of the parameters can be used to create a linear function. These parameters include: total distance, height difference of start- and endpoints and settings of the weights. The resulting function is of the form:

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n = v$$

where $w_1 \ldots w_n$ are the weights.

Creating this linear function can be done by using the *non-negative least squares method*. As input values a large dataset with actual statistics of a large number of paths is taken. These values are put into a new matrix Q. The final costs of these paths are put in a vector $\mathbf{p}$.

A vector $\mathbf{x}$ can now be calculated, which gives values needed to make an accurate heuristic function.

$$Q\mathbf{x} = \mathbf{p}$$

The speed can be further improved by removing the factors in $\mathbf{x}$ that are almost 0, and repeating the process.

6.6 Post-process smoothing

When building a road, the result might not always be a nice flowing road. Sometimes, there could be some sharp turns in the road that the user does not want. To solve this problem, some smoothing needs to be applied. Because this is done after the road has been generated, this is called post-process smoothing.

There are various techniques for doing this. The technique chosen for this assignment is the *cardinal spline*. This was chosen because it is easy to understand and easy to implement. For each segment, it calculates two virtual points and then draws a Bézier curve. The results can be seen in figure 6.20.

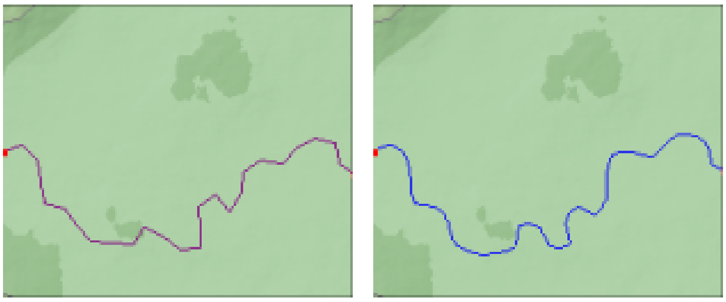


Figure 6.20: Differences in result with post-process smoothing. The left image shows a road without and the right image shows a road with filter applied.

6.7 Organizing the software

Now that it is clear what software is to be created, the next step is to organize it into a coherent software system.

When integrating the software with another project, only the pathplanners are needed. The software has therefore been split into two parts. The first part is a DLL which contains the pathplanners, the RoadHelper class and a class for the secondary parameters. The other part is an executable which contains both the user interface and the terrain loading. A schematic representation of the classes are shown in figure 6.21.

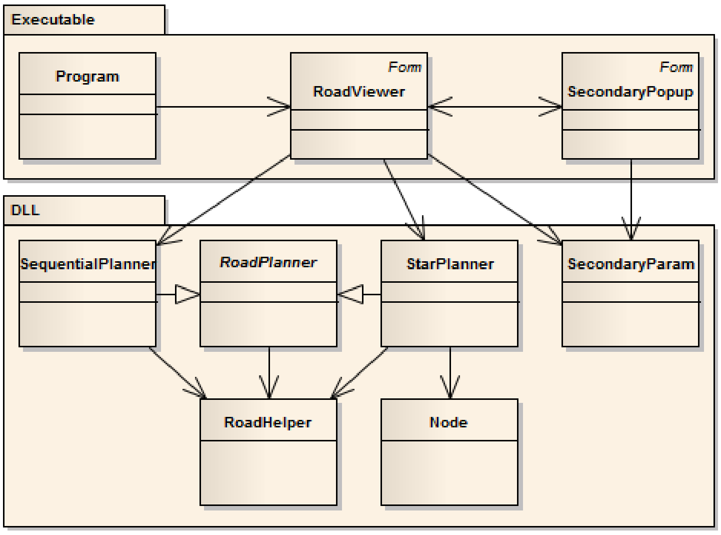


Figure 6.21: Simplified class diagram of the software. For the complete version, see appendix A.

The planner for the modified Sun algorithm has been named SequentialPlanner. The planner for the A* algorithm has been named StarPlanner, as the asterisk (*) is a reserved character. To make sure the two different pathplanners are interchangeable, both classes inherit from an abstract class RoadPlanner. This class also contains the cost functions, so results are more consistent. Some common functions are used by several classes. To keep the code short, consistent and orderly, a static class RoadHelper is created. This contains a number of functions such as, for example, point to index, index to point and distance between two points.

6.8 Implementation

The software for this assignment is written in C# and developed in Microsoft Visual Studio 2010. Most of the software is a direct implementation of the designs described earlier.

The terrain used was made in SketchaWorld. The software can handle terrains of different sizes. However, SketchaWorld currently has a limitation that it can only create square terrains. Support for other ratios could therefore not be tested.

Calling the software is explained in appendix D.

Scrapped 3D image exporter

Originally, the idea was to create a 3D representation of the terrain. This would result in nice images for the report and presentation. However, creating a custom 3D viewer was not realistic, so a decision was made to work with an existing program, Blender. This program has a feature where gray-scale heightmaps can be converted into a 3D terrain. An exporter was created to generate these heightmaps. The result is shown in figure 6.22. Unfortunately, working with Blender proved too difficult. As a result, the exporter became obsolete.

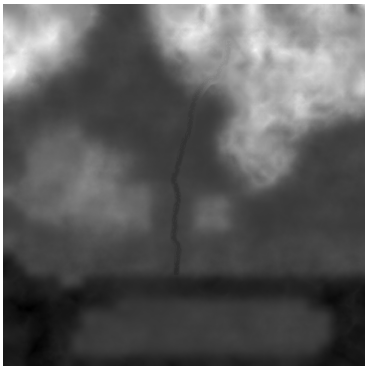


Figure 6.22: An example elevation map. White is high and black is low.

6.9 Testing

To test the control over an algorithm, a user interface had to be created. Testing with this was done continuously during development.

6.9.1 2D viewer

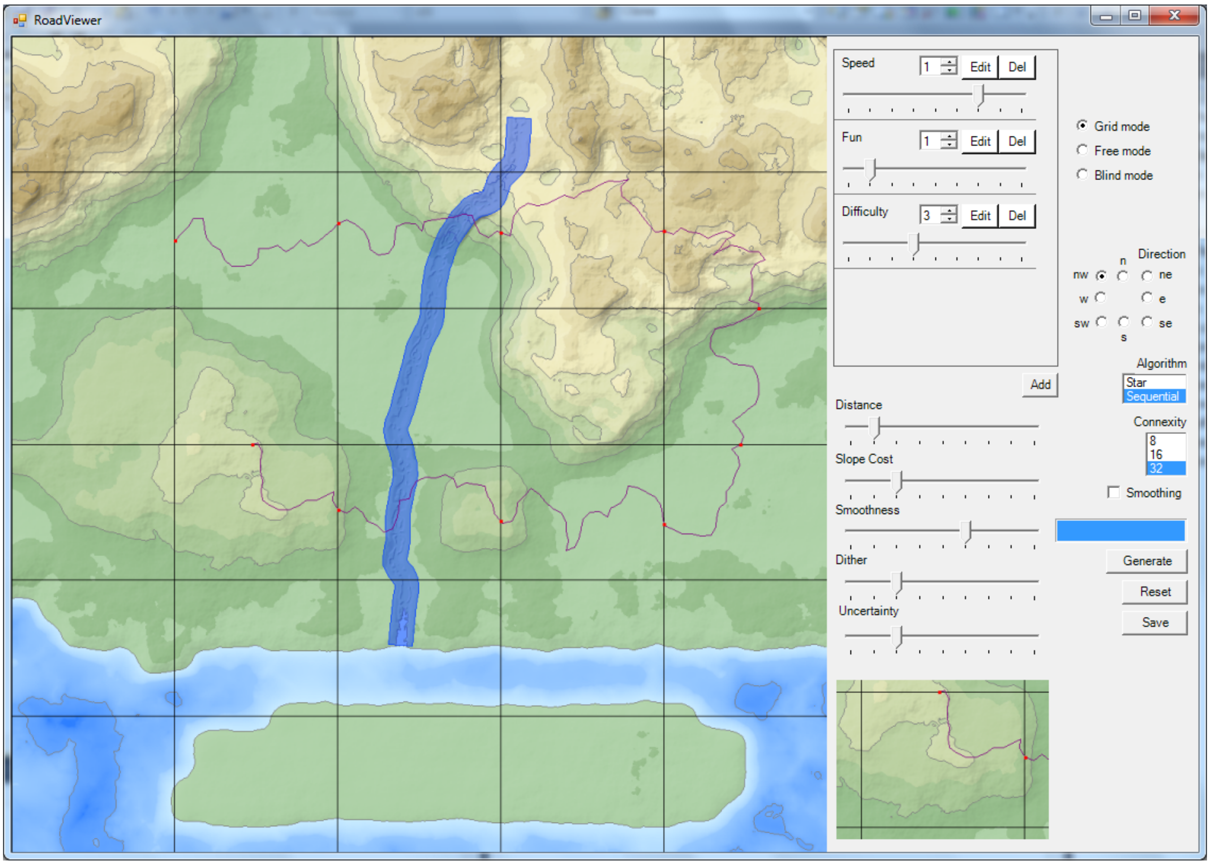
The 2D viewer was developed to make the generating visible and more insightful. People might wonder why the existing code of SketchaWorld was not used as a basis. This is because SketchaWorld is very unstable and often crashes. In addition, this solution allows for more freedom in the development of specialized features. An overview can be seen in figure 6.23.

The most important and obvious part of the interface is the terrain overview. The colours used are the same ones used in some real maps. Their purpose is to indicate height more intuitively. Here, blue is water, green is low terrain and yellow and orange are high terrain.

The sidebar starts at the top with a section for secondary parameters. Parameters can easily be added and removed. By moving the sliders, the sliders for the primary parameters are automatically adjusted, these can be found below the secondary parameters. The primary parameters have a direct influence on the behaviour of the algorithms.

To the right there is a selection of different modes. These modes are to represent certain situations, discussed in section B.8. Next is a selection in which direction the path should go. Starting and endpoints are on the grid. Below that is a selection box that chooses which of the algorithms is used. Below that, the connexity, which is only used when the selected algorithm is the star planner. A higher value gives smoother roads, but also costs more processing time. After that comes an option for post-process smoothing. Below that are a progress bar and some buttons. The progress bar is mainly to indicate whether the algorithm is doing anything, or if it is done. The buttons are respectively “Generate”, “Reset” and “Save”.

Generate causes the selected algorithm to generate a piece of road in the chosen direction, *Reset* removes the current road and *Save* exports the road to an XML-file.



Speed 1
Fun 1
Difficulty 3
Distance
Slope Cost
Smoothness
Dither
Uncertainty

● Grid mode
○ Free mode
○ Blind mode

Direction
nw ● n ○ ne ○
w ○ e ○
sw ○ s ○ se ○

Algorithm
Star
Sequential

Connexity
8
16
32

□ Smoothing

Figure 6.23: Screenshot of the developed viewer.

Finally, there is a small window at the bottom which is meant to show the user how the terrain would work in a game environment. Only the most recent part of road is displayed.

Other features of visualization tool are explained in appendix B.

Chapter 7: Fine tuning

The software is now working, so the next iteration in the development process is to improve the efficiency of the software. An attempt is made to improve the pathplanners in both speed and quality.

7.1 Accurate heuristic cost

Section 6.5.1 mentioned how the heuristic cost can be calculated by using the non-negative least squares method. Finding the actual values is done by using Matlab.

Matlab has a built in function for calculating these factors. This function is called *lsqnonneg* and has two parameters, a matrix A with statistics from a generated path and a vector $\mathbf{b}$ with the total cost of that path. The data is obtained automatically when the pathplanner is used, and stored in a file. After a while, the contents of the file can be loaded into Matlab and the new weights are calculated by executing the following code:

```
X = lsqnonneg(A, B)
```

The vector X indicates which statistics are relevant. The distance cost are often the most important one, while the costs for the height-difference between the start and endpoints are negligible, because of the large distance.

7.2 Heuristic function tests

The importance of a good heuristic function has been mentioned previously. Four variants have to be evaluated to see how big the impact of the heuristic function is. All variants are of the form

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n = v$$

and vary by the amount of parameters.

- The first formula has a large number of parameters (9) which do not all make sense. The goal here is mainly to see if it produces the best results.
- The second formula is slightly more selective. Now there are 5 parameters, which are all useful (or at least can be).
- The third formula is a slightly further optimized version. By taking the previous version and removing the values that are almost 0, there are only two parameters left.
- The fourth and last one has only one parameter. This is just the distance between the two points. This one is not expected to perform well, but will be very quick.

To test this, a starting point and an endpoint are selected and the star planner is executed with the different versions.

The code for the first formula can be seen in algorithm 3, where dx, dy and dz are the respective differences in x, y and z between the starting point and the destination.

Algorithm 3 Code for one of the options for the heuristic function.

```
$total = 0$;
$total += 8.75 * factorDist *$ sqrt$(dx^2 + dy^2)$;
$total += 0.01 * factorDist *$ sqrt$(dx^2 + dy^2 + dz^2)$;
$total += 0.52 * factorSlope *$ sqrt$(dx^2 + dy^2 + dz^2)$;
$total += 1.0 * factorSlope *$ abs$(Z_{start}$ - $Z_{destination})$;
$total += 1.21 * factorSlope *$ abs$(Z_{start}$ - $Z_{middle})$;
$total += 1.03 * factorSlope *$ abs$(Z_{destination}$ - $Z_{middle})$;
$total += 0.43 * factorSmooth *$ sqrt$(dx^2 + dy^2)$;
$total += 0.3 * factorDither *$ sqrt$(dx^2 + dy^2)$;
$total += 0.2 * factorSmooth * factorDither *$ sqrt$(dx^2 + dy^2)$;
```

7.2.1 Time

First, the time it takes to create the path was measured. The results are shown in figure 7.1.

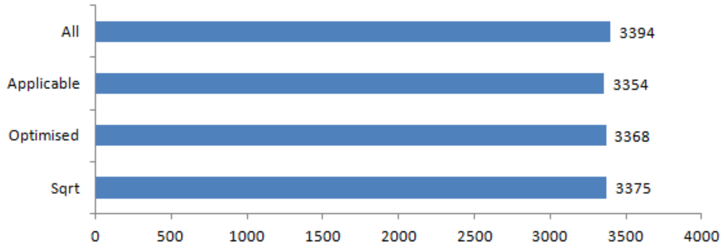


Figure 7.1: Values in ms. Average of 10 measurements.

Here the values hardly differ. The difference between the longest and shortest time is 1.2%.

7.2.2 Cost

The cost for each of the variations were also measured. The results can be seen in figure 7.2.

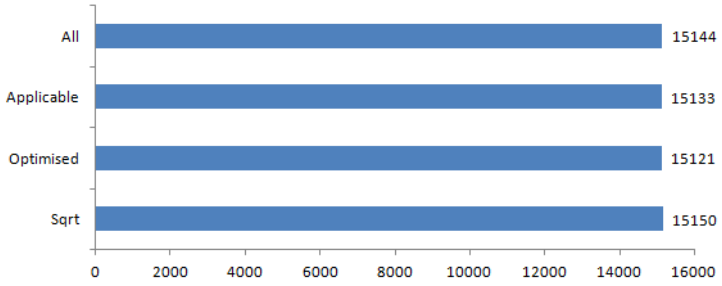


Figure 7.2: Cost of a path. Average of 10 measurements.

Again, the values are almost the same. The biggest difference is now 0.2%.

7.2.3 Assessment

Initially, the expectations were that fewer parameters would generate faster, but have longer paths. So each variation would take less time than the previous one, but the roads would be longer.

However, the observed results are very different from the expected results. There are two possible explanations for why this is. First, calculating the heuristic function is of very little impact to the algorithm. Slight differences can be chalked up to varying processor load. Secondly, the heuristic function does make calculations faster, but returns incorrect results. This causes the star planner to take more wrong side paths, which it gives up on.

Whichever of the two is true, changing the heuristic function has very little impact on the execution time of the algorithm.

7.3 Optimization tests

Speed is an important factor in on-line generation. To see if the speed can be improved further by changing a number of standard functions, some tests were performed.

Keep in mind that the times are very hardware-dependent, this is mostly about ratios. Also, this was tested in C#. Other programming languages may yield different results.

Multiply vs Bitshift

A large part of the multiplications in the code are in multiples of two. As an alternative for this, a bitshift can be used.

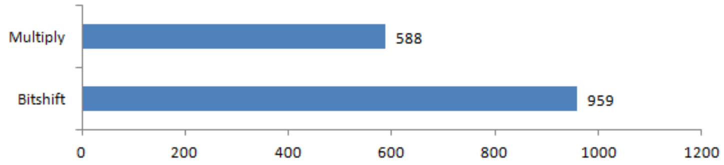


Figure 7.3: Values in ms. Number of times executed: 100 million

Figure 7.3 shows that the multiplication takes 588 ms and the bitshift takes 959 ms. The bitshift takes 63% longer and is therefore slower than an ordinary multiplication.

Divide vs Bitshift

A large part of the divisions are in multiples of two as well. Here a bitshift can be used too, just in the other direction.

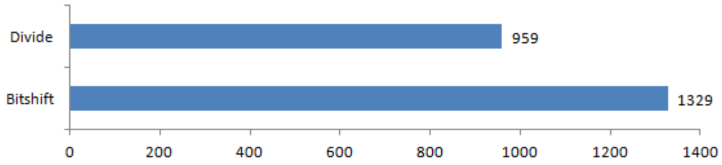


Figure 7.4: Values in ms. Number of times executed: 100 million

In the case shown in figure 7.4, the bitshift takes 39% longer.

Modulo vs &-operator

Just as multiplications and divisions, most of the modulo operations are in multiples of two. A great advantage is that now they can be rewritten to:

$$x\%2^n = x\&(2^n - 1)$$

where $\%$ is a modulo operator (not a percentage) and $\&$ is a bitwise AND-operator.

The principle here is that the bit representing the 2^n and those to the left have to be removed. By performing an &-operation, values on the left are removed and values on the right are kept.

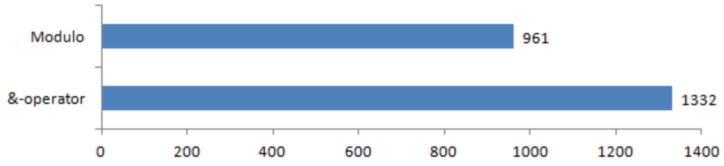


Figure 7.5: Values in ms. Number of times executed: 100 million

When comparing the two values in figure 7.5, it is apparent that the &-operator takes 39% longer.

Faster square root

Finding the distance between two points is a commonly used function. If some speed can be gained here, the execution-time can be reduced. A lot of alternatives are available on the Internet. The tested variant is:

$$\frac{1007}{1024}max(|x|,|y|)+\frac{441}{1024}min(|x|,|y|)$$
$$-\begin{cases} \frac{40}{1024}max(|x|,|y|) & \text{if } max(|x|,|y|) < 16min(|x|,|y|) \\ 0 & \text{otherwise} \end{cases}$$

which was developed by Rafael Baptista (source: [8]).

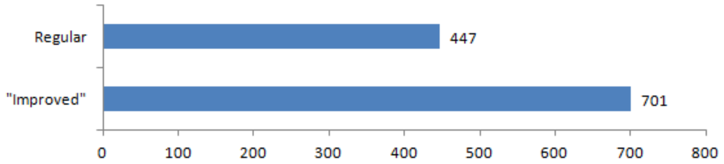


Figure 7.6: Values in ms. Number of times executed: 10 million

Strangely, the "improved" version takes 57% longer, as shown in figure 7.6.

Floats vs Doubles

Using floating point numbers is often a question of precision. It is important to know what the possible gain in speed is between single precision and double precision.

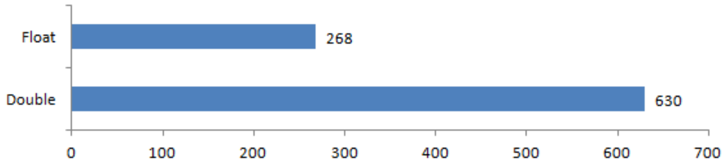


Figure 7.7: Values in ms. Number of times executed: 20 million

Figure 7.7 shows that operations with double precision take 120% longer (so 220% in total) than single precision. This is a clear indication that there needs to be careful consideration about the significance that is needed.

Cast to int

Casting a floating point number to an integer is also a common operation. All cost calculations are in floating point, as well as the selection of points. A major disadvantage of this cast, is that a value such as, for example, 0.999 is converted to 0. In order to prevent this, the value can be rounded, but the drawback is that this takes extra processing time. Both these options are compared in figure 7.8.

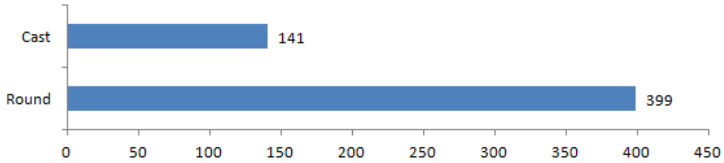


Figure 7.8: Values in ms. Number of times executed: 10 million

The rounding gives an extra time of 183% (almost 3 times longer). These measurements also showed that 4999166 of 10000000 (random) values did not match. With a difference in approximately 50% of the cases, this is a clear indication that the rounding is indeed useful.

7.3.1 Assessment

Initially, it was expected that a large part of the operations could be improved by applying some simple tricks. This turned out to be very disappointing, probably because all bitwise operators are optimized in C#. So, few speed-improvements could be achieved, but the importance of rounding before a cast has become clear.

Chapter 8: Testing and Results

Testing the software is divided into two parts: speed and control.

8.1 Speed

For an on-the-fly algorithm speed is the most important factor. A player is not supposed to notice that another stretch of road is generated and certainly does not want to wait for that.

To measure the speed, a number of fixed points are put all over the map. Between these points, a path is generated and the time is measured. Furthermore, not only is the speed considered, but also how an algorithm scales if the distance is doubled. The pseudo code for doing this is shown in algorithm 4.

Algorithm 4 Pseudo code for testing the speed of the pathplanners

```
Make a list of points spread all over the map
for $i = 0 \rightarrow listsize$ do
    for $j = i \rightarrow listsize$ do
        Start timer
        Generate a path from point $i$ to point $j$
        Stop timer
        Get total distance
        Get elapsed time
        Write data to log
    end for
end for
```

This script is executed 5 times and the results are averaged to filter out any varying processor load. The following graphs show these values. The x-axis shows the distance in pixels, the y-axis is time in ms.

8.1.1 Sequential planner

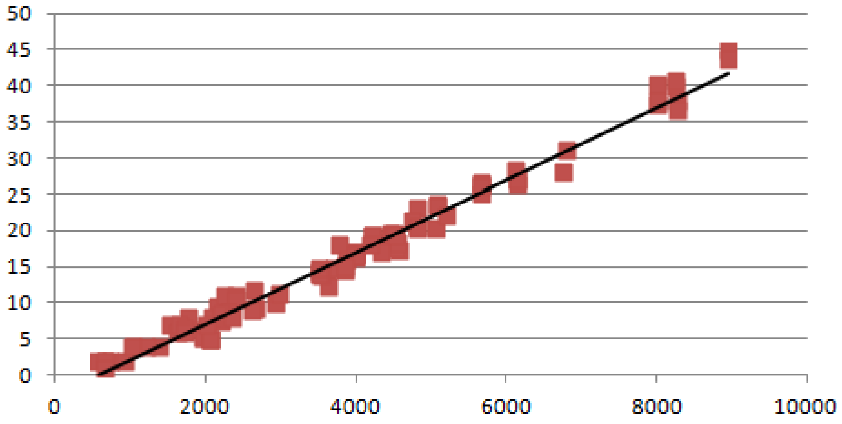


Figure 8.1: Time per distance of the sequential algorithm.

Figure 8.1 shows the results for the sequential planner. The black line is a trendline which gives an indication how long the algorithm will run given a certain distance. It is clearly visible that this algorithm grows linearly.

8.1.2 Star planner

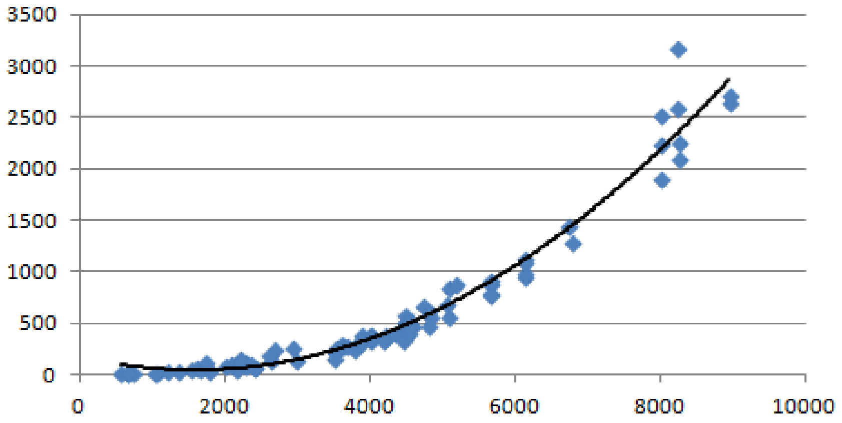


Figure 8.2: Time per distance of the star algorithm.

Figure 8.2 shows the results for the star planner. Again, the black line is a trendline. In this case, the algorithm is a second order polynomial.

8.1.3 Assessment

Comparing the two planners, it is clear that they differ a lot in terms of speed. Not only because of the order of magnitude (sequential is linear, star is polynomial), but also time spent on building actual roads.

For example, taking a distance of 4000, the sequential planner takes about 15 ms while the star planner takes 375 ms. This is a factor of 25.

Therefore, one can conclude that the pathplanners are only comparable (in terms of speed) at short distances.

8.2 Control

Being able to influence the generation is important for users, they want to be in control. So several different situations are looked at to see how the algorithms respond.

Neutral

In this situation, all the sliders are low. This is the default behaviour of the pathplanners. The sequential planner has a simple straight line. The star planner zigzags slightly. This is because it prefers long segments, the direct segments are much shorter.

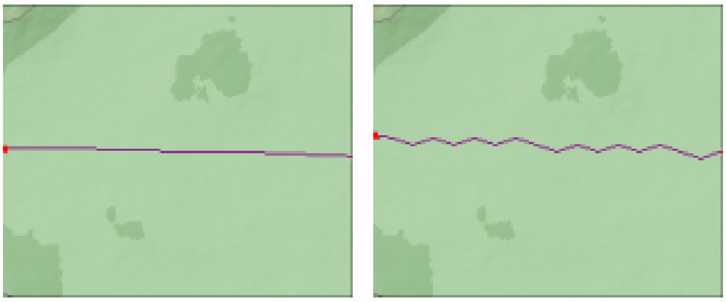


Figure 8.3: Results for the sequential planner and the star planner, respectively.

Distance

When increasing the desired distance, this is the result. The sequential planner makes a large sinusoidal wave, while the star planner is more restrained.

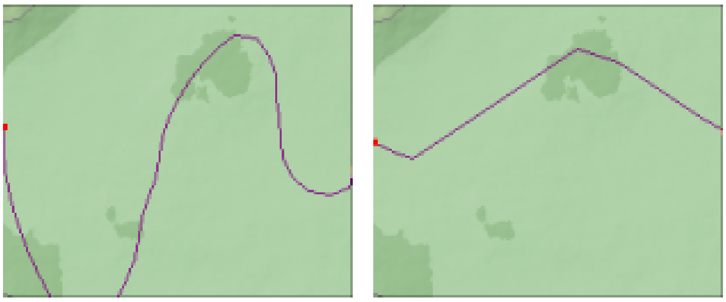


Figure 8.4: Results for the sequential planner and the star planner, respectively.

Slope cost

The slope cost is increased in this situation. The sequential planner fits itself very much to the slope, such as making hairpin turns on the steep parts. The star planner is less conforming, but it still avoids the high peaks.

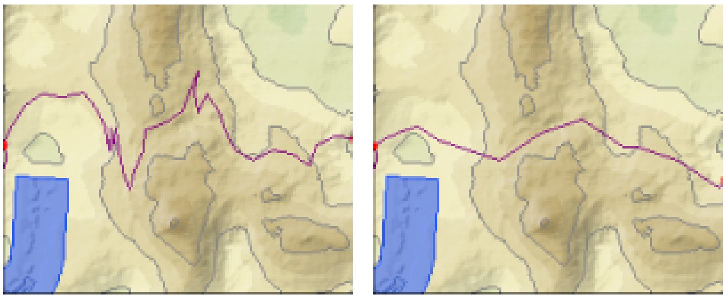


Figure 8.5: Results for the sequential planner and the star planner, respectively.

Here, the difference between the pathplanners is clear. The star planner crosses the high ridge in the middle, while the sequential planner continues on the same side until it is too late.

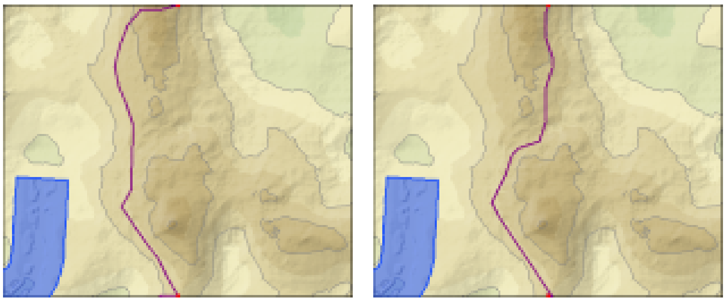


Figure 8.6: Results for the sequential planner and the star planner, respectively.

Smoothness

Smoothness is dependent on other parameters. So to show differences is smoothness, first a situation with some sharp turn has to be found. In the images, both planners use the same settings.

The hairpin turns in the left image disappear when the smoothness is increased.

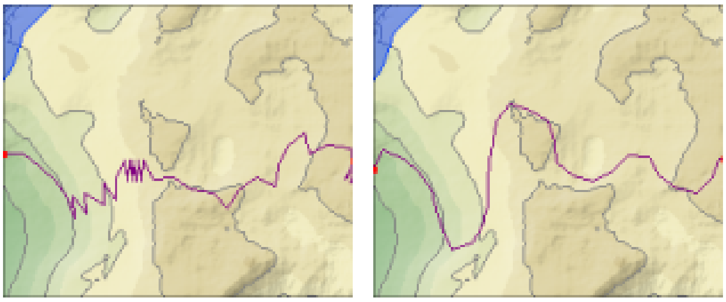


Figure 8.7: Results of increasing smoothness for the sequential planner.

The star planner has a less obvious example, but the differences are still visible.

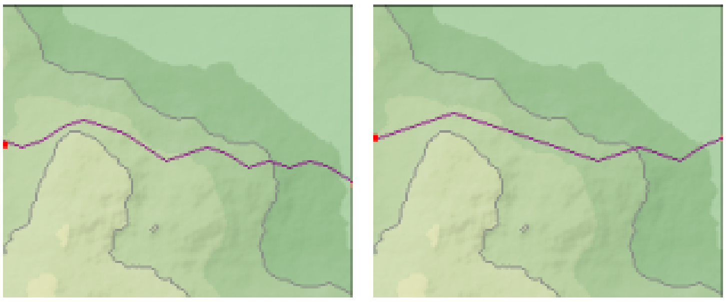


Figure 8.8: Results of increasing smoothness for the star planner.

Dither

In this situation, the dither is increased. The effects can be seen, but star less prominently. Sequential starts to fade near the end, but this is because of the nature of the algorithm.

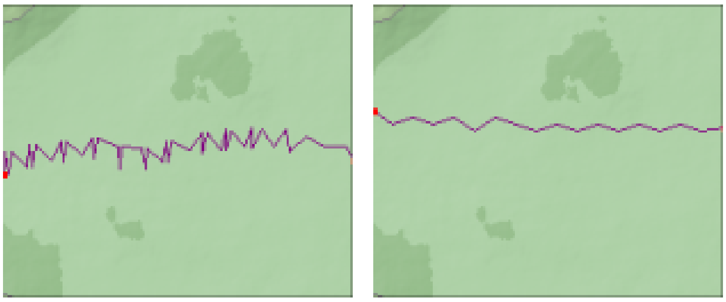


Figure 8.9: Results for the sequential planner and the star planner, respectively.

Uncertainty

As mentioned in section 6.3.5, this parameter does not influence the star planner.

For the sequential planner the results are obvious. The image on the left show a (somewhat) predictable road. The road in the image on the right is much less predictable.

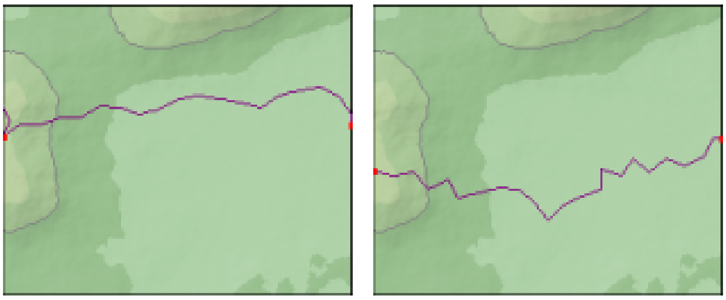


Figure 8.10: The influence of uncertainty on the sequential planner.

Combinations

When combining a high value for dither with a high value for smoothness, a low frequency wave appears.

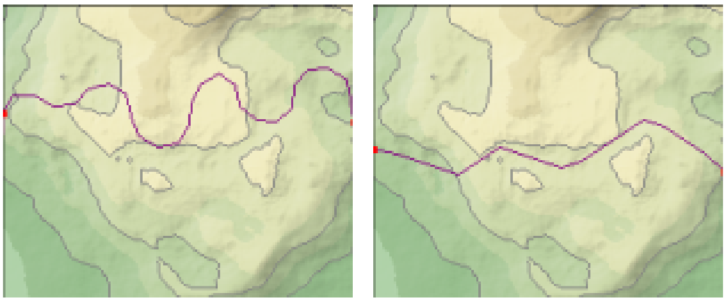


Figure 8.11: Results for the sequential planner and the star planner, respectively.

This is the result when taking the default secondary parameters and setting *speed* low, *fun* high and *difficulty* halfway.

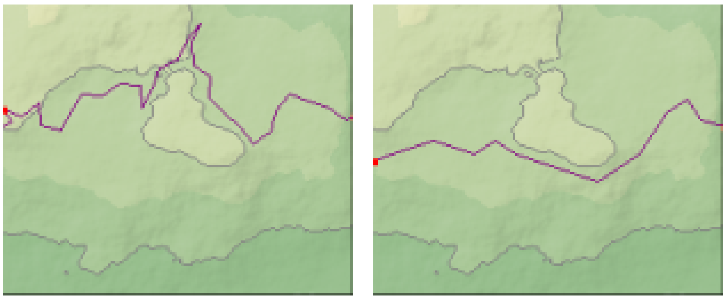


Figure 8.12: Results for the sequential planner and the star planner, respectively.

8.2.1 Assessment

It is clear that the sequential planner is much more responsive to the controls than the star planner. However, the sequential planner doesn't look ahead and takes the best option locally, while the star planner takes the best option globally. Also, the results for the star planner are slightly more realistic.

With some tweaks to the cost functions, the star algorithm could be made more responsive. However, there was no time left.

Chapter 9: Conclusion

Comparing the two pathplanners has lead to some interesting results.

The sequential planner is much faster and it responds very well to control. However, the algorithm does not plan ahead, which causes some strange results. This algorithm is most useful for non-serious situations.

The star algorithm has much more consistent results, it also plans ahead and does not do anything strange. However, it is much slower than the sequential algorithm and can therefore only be used in short distances or when the user is expecting a (minor) delay. Additionally, the uncertainty parameter has no influence on the results. This algorithm is most useful for realism.

Another downside of the star planner is that it has a tendency to ”beat the system”. In some cases, a few high-cost segments are cheaper than a lot of low-cost segments. So a short, straight road is built instead of a long, winding road.

Because the purpose of this assignment is to find a pathplanner which can be used in games, speed and control are very important. Therefore, the best option here is the sequential algorithm.

Going back to the assignment, there now is an on-line road generation algorithm. The outcome of the algorithm is a road in 3D. The generation can be influenced by the weights of the cost function (the geometric parameters) or by the secondary parameters (the gameplay parameters). Its efficiency is debatable, but is has been improved (in chapter 7). All of this has been turned into a DLL-library.

Additionally, a 2D visualization tool has been built for easy control and testing.

If the library is to be used for other projects, an explanation of how to call the software is given in appendix D.

Chapter 10: Future work

This chapter summarizes some suggestions for improvements in the future.

The most obvious continuation is using the code in a race game. Both the terrain and the road can be generated while driving.

One of the papers mentions using bridges and tunnels. By using bridges, rivers are no longer obstacles, but also ravines can be crossed. Tunnels can be used to go through mountains, in stead of around them. Expanding even further, both can be combined to go down a cliff.

To make the A* algorithm a more viable option, some of the node's neighbours can be randomly thrown away. This is called *stochastic sampling* an can reduce calculation time drastically. However, this also affects results.

Right now, all the functions that handle the secondary parameters have at least 12 inputs. This is because the primary parameters have several ways to be influenced. By creating a new class for the primary parameters, the code will look more organized. This has no benefit for the user, but might be convenient for developers.

Making the heuristic function dependent on the terrain. When loading the terrain, some sort of hash function can be applied, which gives some statistics about the terrain. These can then be applied to the heuristic function, which could improve the results.

Combining features of the sequential planner and the star planner would have interesting results. This hybrid would be mostly like the sequential algorithm, but uses heuristics to determine which of the neighbours is best. This could improve the quality.

Another possibility is to see if Probabilistic Roadmaps are a viable option. As it has not been thoroughly researched, not much can be said about it.

In order to further increase the on speed multi-core processors or distributed systems, the calculations of the cost function can be distributed over several threads. Also, looking at multiple neighbours at the same time can be possible.

Another idea is to use something like attractor points. Here, the algorithm is rewarded by moving along the coastline or through a city. This allows for more realistic roads.

If further research is carried out into the feasibility of the A* algorithm, there are a few possible extensions. Firstly, the size of the grid can be made dependent on the distance between the start and endpoint. If these are close together, it could provide a better road. Secondly, the grid can also be made non-uniform. In the open plains or at the top of a mountain, there is less need for precision. This could make a big difference in computation time.

The computer graphics group has a semantic database, where they try to save abstract terms. This is sort of the same that has been tried with the secondary parameters in this project. If the semantic database is integrated, there is a standardized way to adjust settings.

The supervisor came up with an idea to further improve the quality of the sequential algorithm. He suggested that the algorithm could generate multiple large pieces of road and then compares these. This would increase the computing-time a lot, but it would still be a linear growth.

The last suggestion is having support for different types of road. Rural roads will just be built on the terrain, but highways will have to be smoother. Hills and terrains can be smoothed by removing bumps and holes. Also, the road can bank in the corners for and extra effect.

Afterword

First and foremost, I want to thank Ricardo De Vasconcelos Abreu Lopes, my daily supervisor, for his continuous support and feedback. Secondly, I want to thank Rafael Bidarra for giving me the opportunity to do this assignment.

For their continued technical support, I want to thank Ruud de Jong and Bart Vastenhouw.

Furthermore, I want to thank all the roommates I had during this time for being able to share a joke, play a game or bounce ideas off each other. In particular Anne-Marijn, Berend, Cees-willem, Christian, Hugo, Lars, Marnix, Rick and Roland.

I also want to thank the teachers who gave me feedback on the documentation and report, namely mr. Heyer, mr. Andrioli and mr. Visser.

Lastly, I want to thank you, the reader, for taking the time to read this report.

Reflection

This has been an interesting and educative assignment. I've learned a lot about procedural content generation. Personally, I am very satisfied with the results. The planner responds very well to user input and is very fast. In my opinion, taking two algorithms and comparing them was a good idea. This allowed me to go one step further than just working with hypothetical situations.

However, there are some things that did not go over so well. Initially, I misunderstood the assignment. I was under the impression that the pathplanner for SkechaWorld had to be replaced by a new one. Luckily, most of the code could be salvaged, but an extra iteration was needed.

Something I learned during this project is working with LaTeX on an advanced level. Before this project, I had worked with LaTeX, but it was all very basic. This report is much more complicated, with design-templates, a bibliography and cross-referencing.

Nearing the end of the project, a big problem occurred. When a demo was given to the client (the person who gave the assignment), he was not pleased at all. The idea that he had in his mind didn't match with what he saw in front of his eyes. This is always a problem with transferring an idea; each person has a different interpretation and creates a different mental picture. He wanted me to completely change the core software. However, because this was about four weeks before the deadline, I had to make an assessment of time spent redesigning and building versus the expected result. I decided not to change the software. The client was not happy with this, but understood why I did it. Personally, I think that the library can still be used.

References

Papers

[Bresenham 65] J. Bresenham. *Algorithm for Computer Control of a Digital Plotter*. IBM Systems Journal, pages 25–30, 1965.

[Galin 10] E. Galin, A. Peytavie, N. Maréchal & E. Guérin. *Procedural Generation of Roads*. Computer Graphics Forum: Proceedings of Eurographics 2010, vol. 29, pages 429–438, 2010.

[Galin 11] E. Galin, A. Peytavie, E. Guérin & B. Beneš. *Authoring Hierarchical Road Networks*. Computer Graphics Forum: Proceedings of Eurographics 2011, vol. 30, pages 2021–2030, 2011.

[Kelly 07] G. Kelly & H. McCabe. *Citygen: An Interactive System for Procedural City Generation*. Proceedings of GDTW 2007: The Fifth Annual International Conference in Computer Game Design and Technology, pages 8–16, 2007.

[Sun 02] J. Sun, X. Yu, G. Baciu & M. Green. *Template-based Generation of Road Networks for Virtual City Modeling*. Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pages 33–40, 2002.

Websites

[1] F. T. Ndonga, "Figure eemcs-building." `"http://fndonga.weblog.tudelft.nl/2011/12/09/tick-tock-where-s-my-o-clock"`, December 2011.

[2] Agile-development-tools.com, "Figure iterative development." `"http://agile-development-tools.com"`, 2009.

[3] D. Girard, "Figure speedtree editor." `"http://polygonspixelsandpaint.tumblr.com/post/8609829411"`, August 2011.

[4] R. Smelik, "Figure sketchaworld." `"http://graphics.tudelft.nl/Game_Technology/Smelik"`, March 2012.

[5] Crotalus Horridus, "Figure bresenham." `"http://en.wikipedia.org/wiki/Bresenham's_line_algorithm"`, December 2007.

[6] R. Wenderlich and A. Löbker, "Figure a* (modified)." `"http://www.raywenderlich.com/4946/introduction-to-a-pathfinding"`, September 2011.

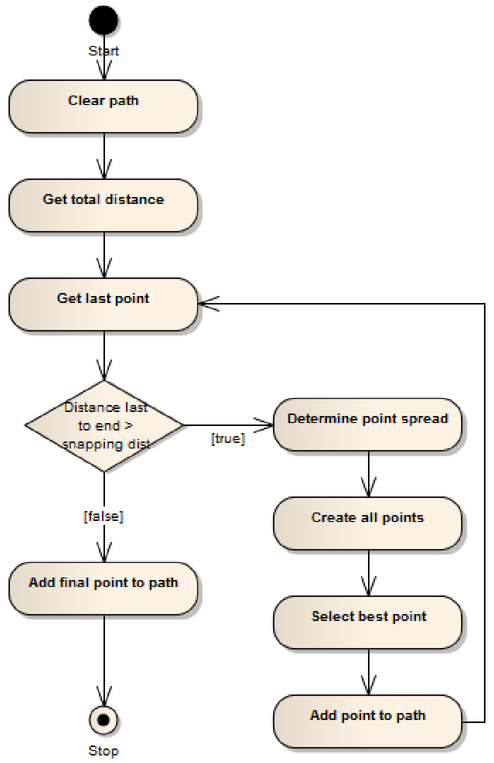
[7] B. Anguelov, “Figure heuristics.” "http://takinginitiative.net/2011/05/02/optimizing-the-a-algorithm/", May 2011.

[8] R. Baptista, “Faster square root.” "http://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml", June 2003.

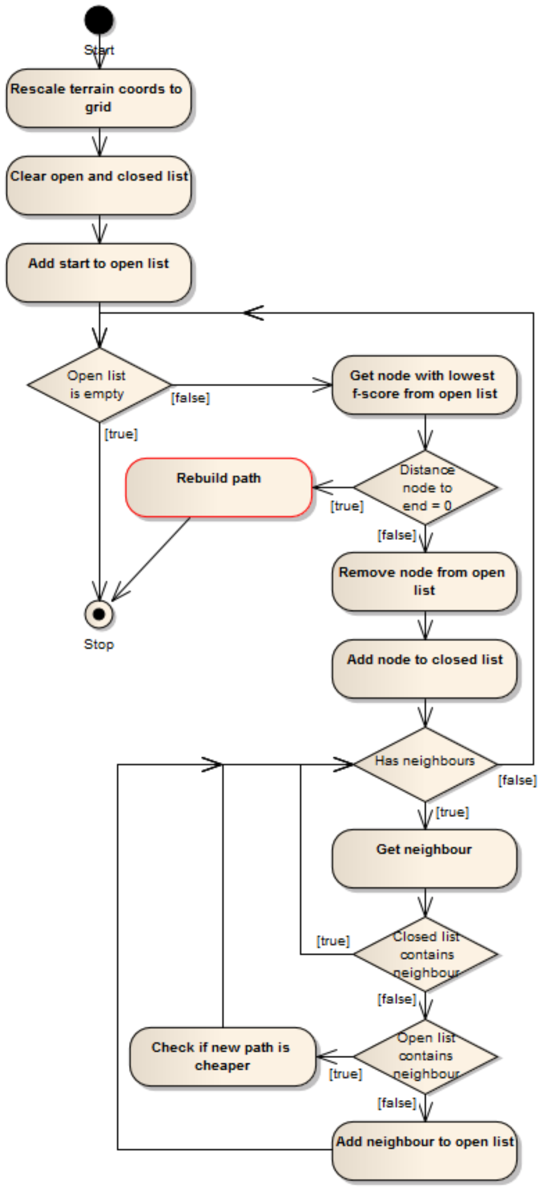
Appendix A: Diagrams

A.1 Dataflow diagrams

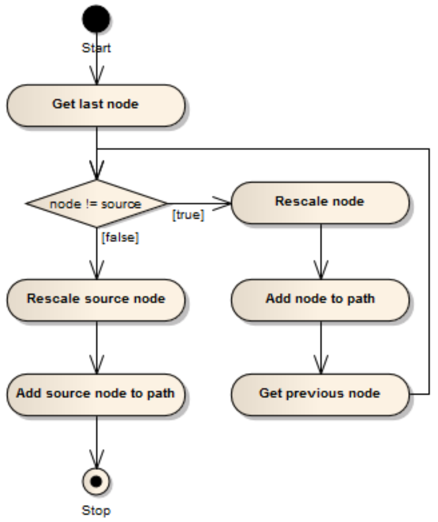
Sequential



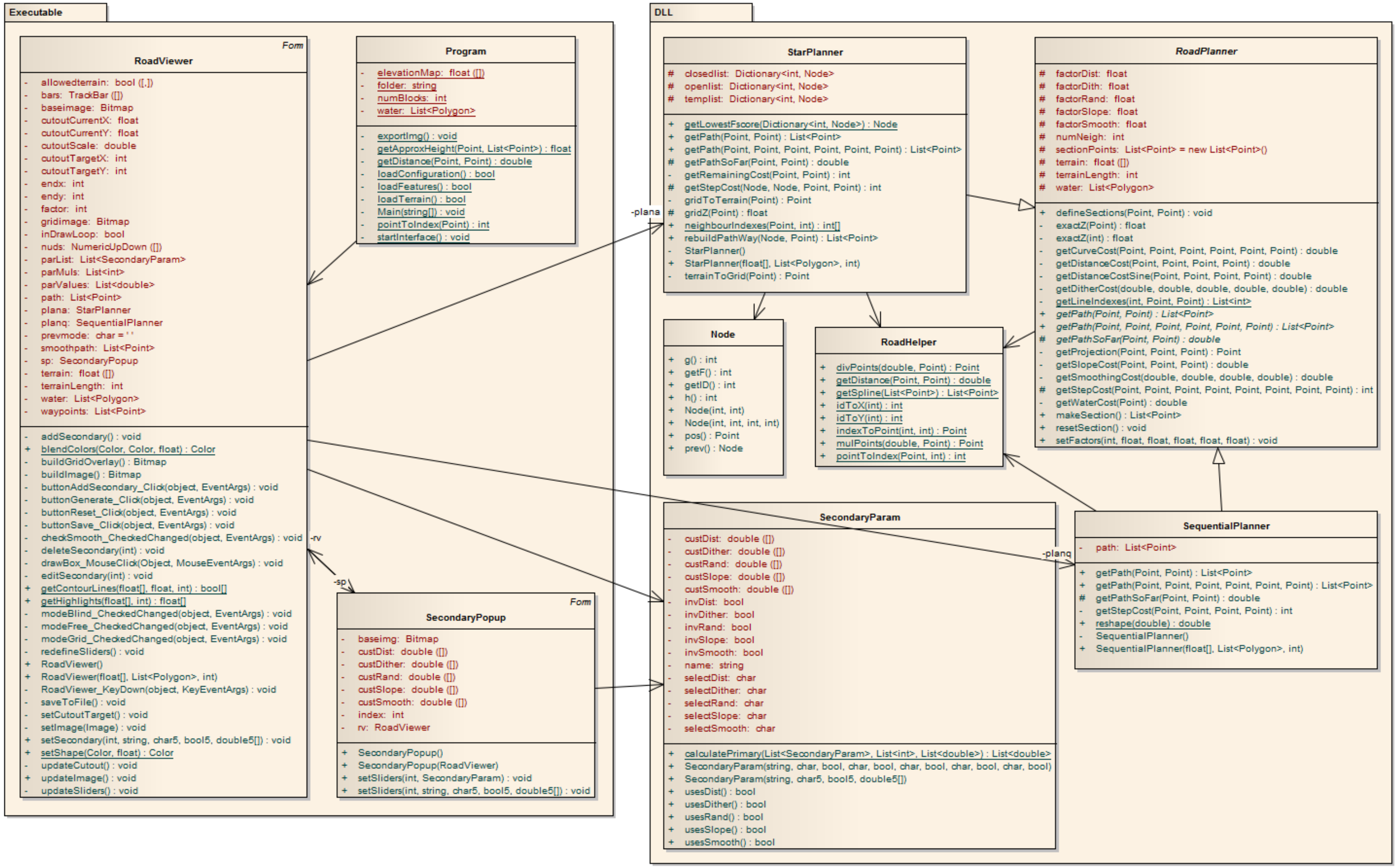
Star


Start
Rescale terrain coords to grid
Clear open and closed list
Add start to open list
Open list is empty
[false]
Get node with lowest f-score from open list
[true]
Distance node to end = 0
[true]
Rebuild path
[false]
Stop
Remove node from open list
Add node to closed list
Has neighbours
[false]
[true]
Get neighbour
Closed list contains neighbour
[true]
[false]
Open list contains neighbour
[true]
Check if new path is cheaper
[false]
Add neighbour to open list

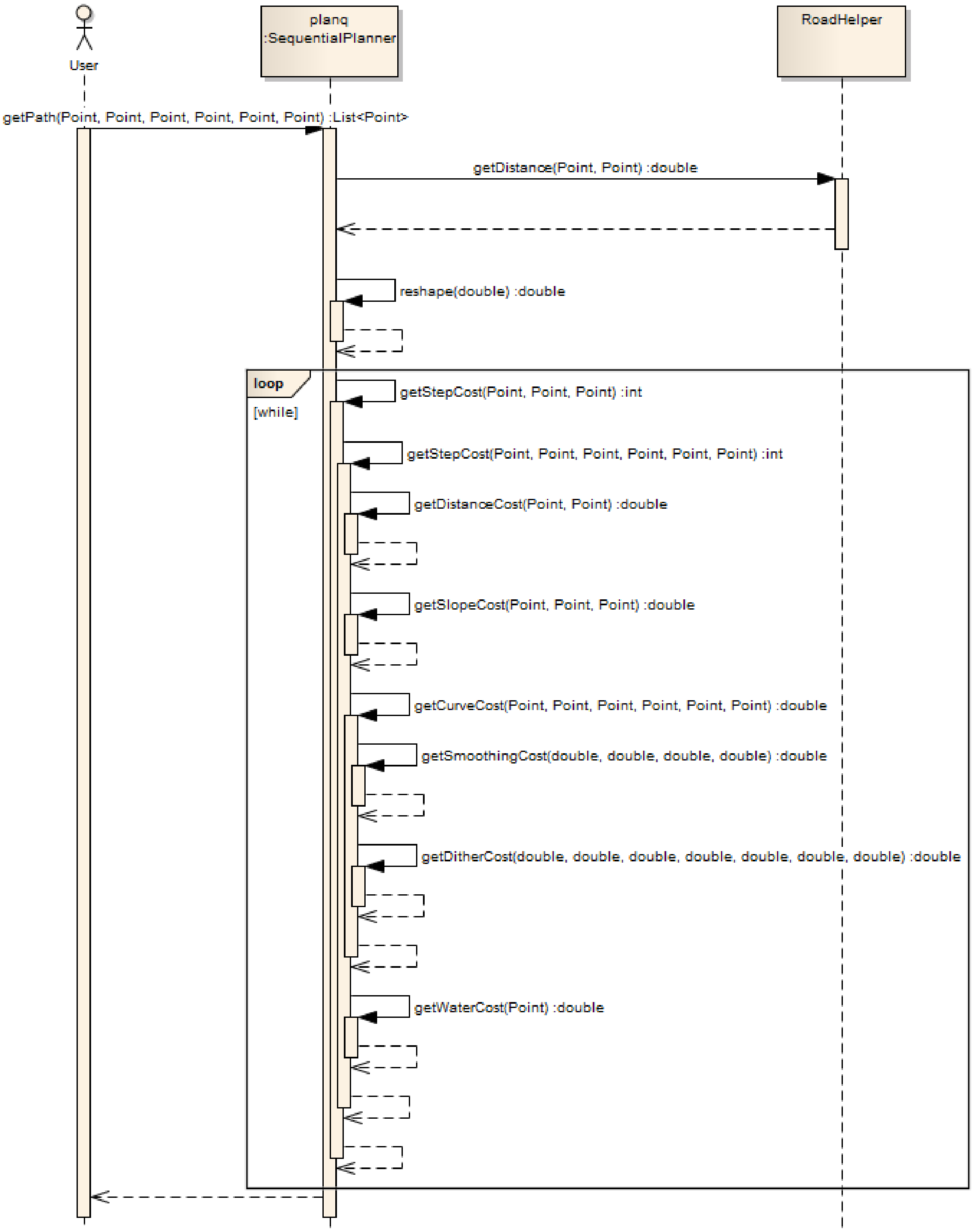
Rebuild path function

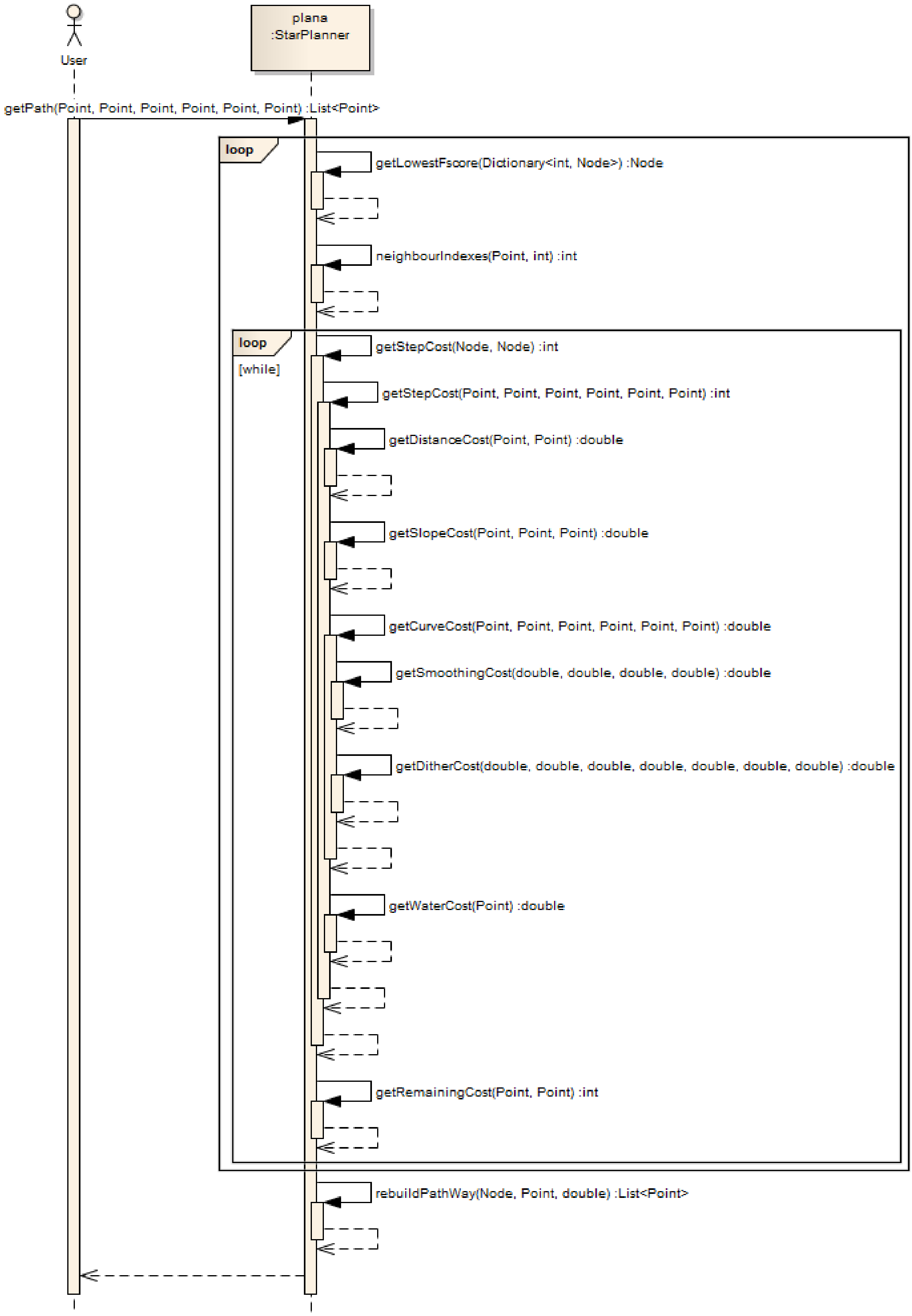


A.2 Class diagram


Executable
RoadViewer
Form
- allowedterrain: bool ([,])
- bars: TrackBar ([])
- baseimage: Bitmap
- cutoutCurrentX: float
- cutoutCurrentY: float
- cutoutScale: double
- cutoutTargetX: int
- cutoutTargetY: int
- endx: int
- endy: int
- factor: int
- gridimage: Bitmap
- inDrawLoop: bool
- nuds: NumericUpDown ([])
- parList: List<SecondaryParam>
- parMuls: List<int>
- parValues: List<double>
- path: List<Point>
- plana: StarPlanner
- planq: SequentialPlanner
- prevmode: char = ' '
- smoothpath: List<Point>
- sp: SecondaryPopup
- terrain: float ([])
- terrainLength: int
- water: List<Polygon>
- waypoints: List<Point>
- addSecondary() : void
+ blendColors(Color, Color, float) : Color
- buildGridOverlay() : Bitmap
- buildImage() : Bitmap
- buttonAddSecondary_Click(object, EventArgs) : void
- buttonGenerate_Click(object, EventArgs) : void
- buttonReset_Click(object, EventArgs) : void
- buttonSave_Click(object, EventArgs) : void
- checkSmooth_CheckedChanged(object, EventArgs) : void
- deleteSecondary(int) : void
- drawBox_MouseClick(Object, MouseEventArgs) : void
- editSecondary(int) : void
+ getContourLines(float[], float, int) : bool[]
+ getHighlights(float[], int) : float[]
- modeBlind_CheckedChanged(object, EventArgs) : void
- modeFree_CheckedChanged(object, EventArgs) : void
- modeGrid_CheckedChanged(object, EventArgs) : void
- redefineSliders() : void
+ RoadViewer()
+ RoadViewer(float[], List<Polygon>, int)
- RoadViewer_KeyDown(object, KeyEventArgs) : void
- saveToFile() : void
- setCutoutTarget() : void
- setImage(Image) : void
+ setSecondary(int, string, char5, bool5, double5[]) : void
+ setShape(Color, float) : Color
- updateCutout() : void
+ updateImage() : void
- updateSliders() : void
Program
- elevationMap: float ([])
- folder: string
- numBlocks: int
- water: List<Polygon>
- exportImg() : void
- getApproxHeight(Point, List<Point>) : float
- getDistance(Point, Point) : double
- loadConfiguration() : bool
- loadFeatures() : bool
- loadTerrain() : bool
- Main(string[]) : void
- pointToIndex(Point) : int
- startInterface() : void
-rv
-sp
SecondaryPopup
Form
- baseimg: Bitmap
- custDist: double ([])
- custDither: double ([])
- custRand: double ([])
- custSlope: double ([])
- custSmooth: double ([])
- index: int
- rv: RoadViewer
+ SecondaryPopup()
+ SecondaryPopup(RoadViewer)
+ setSliders(int, SecondaryParam) : void
+ setSliders(int, string, char5, bool5, double5[]) : void
DLL
-plana
StarPlanner
# closedlist: Dictionary<int, Node>
# openlist: Dictionary<int, Node>
# templist: Dictionary<int, Node>
+ getLowestFscore(Dictionary<int, Node>) : Node
+ getPath(Point, Point) : List<Point>
+ getPath(Point, Point, Point, Point, Point, Point) : List<Point>
# getPathSoFar(Point, Point) : double
- getRemainingCost(Point, Point) : int
# getStepCost(Node, Node, Point, Point) : int
- gridToTerrain(Point) : Point
# gridZ(Point) : float
+ neighbourIndexes(Point, int) : int[]
+ rebuildPathWay(Node, Point) : List<Point>
- StarPlanner()
+ StarPlanner(float[], List<Polygon>, int)
- terrainToGrid(Point) : Point
RoadPlanner
# factorDist: float
# factorDith: float
# factorRand: float
# factorSlope: float
# factorSmooth: float
# numNeigh: int
# sectionPoints: List<Point> = new List<Point>()
# terrain: float ([])
# terrainLength: int
# water: List<Polygon>
+ defineSections(Point, Point) : void
- exactZ(Point) : float
- exactZ(int) : float
- getCurveCost(Point, Point, Point, Point, Point, Point) : double
- getDistanceCost(Point, Point, Point, Point) : double
- getDistanceCostSine(Point, Point, Point, Point) : double
- getDitherCost(double, double, double, double, double) : double
- getLineIndexes(int, Point, Point) : List<int>
+ getPath(Point, Point) : List<Point>
+ getPath(Point, Point, Point, Point, Point, Point) : List<Point>
# getPathSoFar(Point, Point) : double
- getProjection(Point, Point, Point) : Point
- getSlopeCost(Point, Point, Point) : double
- getSmoothingCost(double, double, double, double) : double
# getStepCost(Point, Point, Point, Point, Point, Point, Point, Point) : int
- getWaterCost(Point) : double
+ makeSection() : List<Point>
+ resetSection() : void
+ setFactors(int, float, float, float, float, float) : void
Node
+ g() : int
+ getF() : int
+ getID() : int
+ h() : int
+ Node(int, int)
+ Node(int, int, int, int)
+ pos() : Point
+ prev() : Node
RoadHelper
+ divPoints(double, Point) : Point
+ getDistance(Point, Point) : double
+ getSpline(List<Point>) : List<Point>
+ idToX(int) : int
+ idToY(int) : int
+ indexToPoint(int, int) : Point
+ mulPoints(double, Point) : Point
+ pointToIndex(Point, int) : int
SecondaryParam
- custDist: double ([])
- custDither: double ([])
- custRand: double ([])
- custSlope: double ([])
- custSmooth: double ([])
- invDist: bool
- invDither: bool
- invRand: bool
- invSlope: bool
- invSmooth: bool
- name: string
- selectDist: char
- selectDither: char
- selectRand: char
- selectSlope: char
- selectSmooth: char
+ calculatePrimary(List<SecondaryParam>, List<int>, List<double>) : List<double>
+ SecondaryParam(string, char, bool, char, bool, char, bool, char, bool, char, bool)
+ SecondaryParam(string, char5, bool5, double5[])
+ usesDist() : bool
+ usesDither() : bool
+ usesRand() : bool
+ usesSlope() : bool
+ usesSmooth() : bool
-planq
SequentialPlanner
- path: List<Point>
+ getPath(Point, Point) : List<Point>
+ getPath(Point, Point, Point, Point, Point, Point) : List<Point>
# getPathSoFar(Point, Point) : double
- getStepCost(Point, Point, Point, Point) : int
+ reshape(double) : double
- SequentialPlanner()
+ SequentialPlanner(float[], List<Polygon>, int)

A.3 Sequence diagrams


User
planq
:SequentialPlanner
RoadHelper
getPath(Point, Point, Point, Point, Point, Point) :List<Point>
getDistance(Point, Point) :double
reshape(double) :double
loop
[while]
getStepCost(Point, Point, Point) :int
getStepCost(Point, Point, Point, Point, Point, Point) :int
getDistanceCost(Point, Point) :double
getSlopeCost(Point, Point, Point) :double
getCurveCost(Point, Point, Point, Point, Point, Point) :double
getSmoothingCost(double, double, double, double) :double
getDitherCost(double, double, double, double, double, double, double) :double
getWaterCost(Point) :double


User
plana
:StarPlanner
getPath(Point, Point, Point, Point, Point, Point) :List<Point>
loop
getLowestFscore(Dictionary<int, Node>) :Node
neighbourIndexes(Point, int) :int
loop
[while]
getStepCost(Node, Node) :int
getStepCost(Point, Point, Point, Point, Point, Point) :int
getDistanceCost(Point, Point) :double
getSlopeCost(Point, Point, Point) :double
getCurveCost(Point, Point, Point, Point, Point, Point) :double
getSmoothingCost(double, double, double, double) :double
getDitherCost(double, double, double, double, double, double, double) :double
getWaterCost(Point) :double
getRemainingCost(Point, Point) :int
rebuildPathWay(Node, Point, double) :List<Point>

Appendix B: Features of the 2D viewer

2D viewer was designed for visualization and testing.

B.1 Elevation lines

The 2D viewer uses elevation lines. Initially, these were generated by simply taking height values within a certain range. Unfortunately, this led to very inconsistent results, such as large blobs at gentle slopes and very thin (sometimes even no) lines at steep slopes. To solve this problem, a better algorithm had to be developed.

This algorithm iterates over all pixels. For each pixel, the one on the left side and the one on the right side are taken. For these, they are checked if they are in a different height-group (i.e. should there be an elevation line between them). If this is true, they are checked if the transition is between this pixel and the one on the left or between this one and the one on the right. After that, a check is performed which of the two remaining pixels is closest to the transition. If that is the currently selected pixel, then it should be part of the elevation line and thus greyed out. This check is repeated in the vertical direction.

B.2 Highlighting

By using colours and elevation lines, it is starting to get clear what is high and what is low. However, there is a simple trick which causes the human eye to perceive depth: lighting. If certain areas are highlighted and others darkened, the illusion of depth is created. In figure B.1 the light appears to be coming from the top left corner.

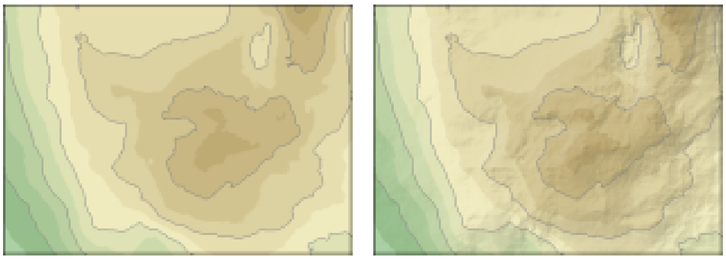


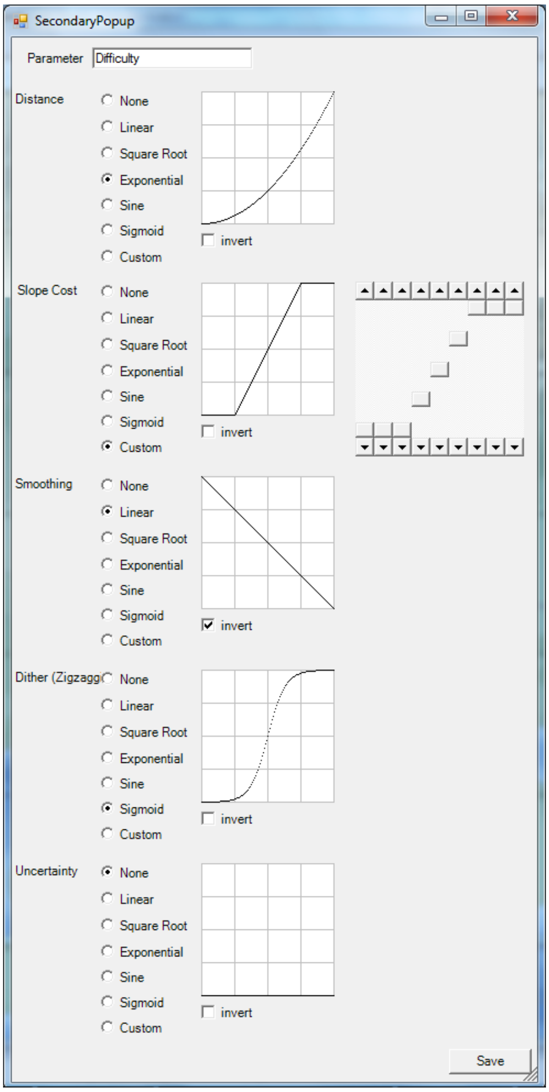
Figure B.1: Without and with highlights.

The calculation is quite simple: depending on the slope of the terrain, make the colour lighter or darker.

B.3 Secondary parameters

Secondary parameters were already mentioned in section 6.4. To make these, a window was created, where they can be defined. Users who want to define or change parameters have the possibility to do so. Additionally, these parameters need only be set once.

The new parameter can influence the primary parameters in different ways. For each of the primary parameters, a different dependency can be chosen. Be it linear, quadratic, sinusoidal or a custom setting. Additionally, the secondary parameter can be given a name to keep some overview.



SecondaryPopup

Parameter: Difficulty

Distance: ○ None ○ Linear ○ Square Root ● Exponential ○ Sine ○ Sigmoid ○ Custom — ☐ invert

Slope Cost: ○ None ○ Linear ○ Square Root ○ Exponential ○ Sine ○ Sigmoid ● Custom — ☐ invert

Smoothing: ○ None ● Linear ○ Square Root ○ Exponential ○ Sine ○ Sigmoid ○ Custom — ☑ invert

Dither (Zigzagg: ○ None ○ Linear ○ Square Root ○ Exponential ○ Sine ● Sigmoid ○ Custom — ☐ invert

Uncertainty: ● None ○ Linear ○ Square Root ○ Exponential ○ Sine ○ Sigmoid ○ Custom — ☐ invert

Figure B.2: Screenshot of the settings of the secondary parameters.

Figure B.2 shows what a custom setting might look like. The user has nine control points and the values in between are linearly interpolated.

B.4 Cutout

As mentioned earlier, the interface has a small window at the bottom, which shows a little part of the map. This is meant to show the user how the generation would work in a 2D game. After building a new piece of road, the image makes a smooth transition to the new square.

B.5 Hotkeys

For ease of using and testing, some hotkeys were created. This reduced the amount of clicking necessary. The numpad keys 1 through 4 and 6 through 9 are used for generating a piece of road in the respective direction. Numpad key 5 is used for switching between the pathplanners and numpad key 0 is used as a reset button.

B.6 Water and out-of-bounds check

The program has two checks built in. The first one makes sure that the path cannot go outside the map. If the user gives a command which would cause the road to leave the map, he or she gets an error message. The second check is to make sure the road doesn't go into the water. Currently, bridges are not available. Therefore, it was decided that building on water should not be available. When a user attempts to build a road with the endpoint in the water, the user receives an error message and the algorithm stops.

B.7 XML config file

A number of settings do not have to be changed often. Therefore, a config-file was created. When starting the viewer, this file is automatically loaded.

B.8 Modes

The algorithms should work in some hypothetical situations. To show this, a few modes were created that emulate these situations.

B.8.1 Grid mode

This is the default mode for the viewer. The map has been divided into several large blocks. The purpose of this is to simulate generation for 2D games.

For example, take the situation where you are playing a little flash-game where you have to race. While you are driving, the road that falls outside of your screen is being generated. It does so constantly and you should not notice that this is happening.

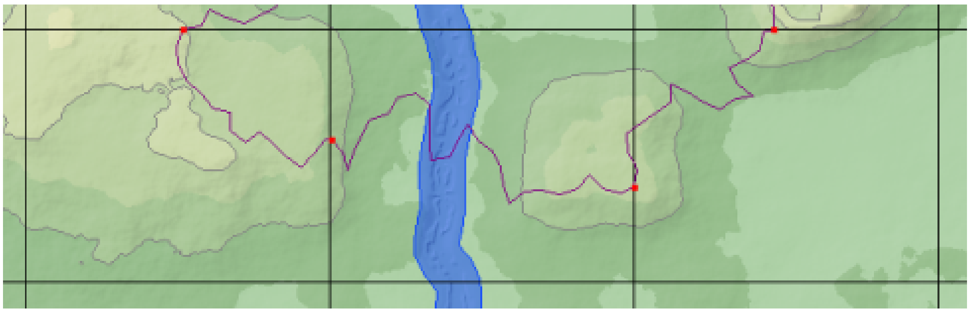


Figure B.3: An example road in the grid mode

B.8.2 Free mode

When building a road, the user might want to vary between settings for different parts. If that is the case, the parts need to be connected properly.

When the user clicks on two places on the map (start and end), some subpoints are automatically created. The location of these points are dependent on the settings for distance and uncertainty. Every time a section is generated, the settings can be changed and a different road is generated.

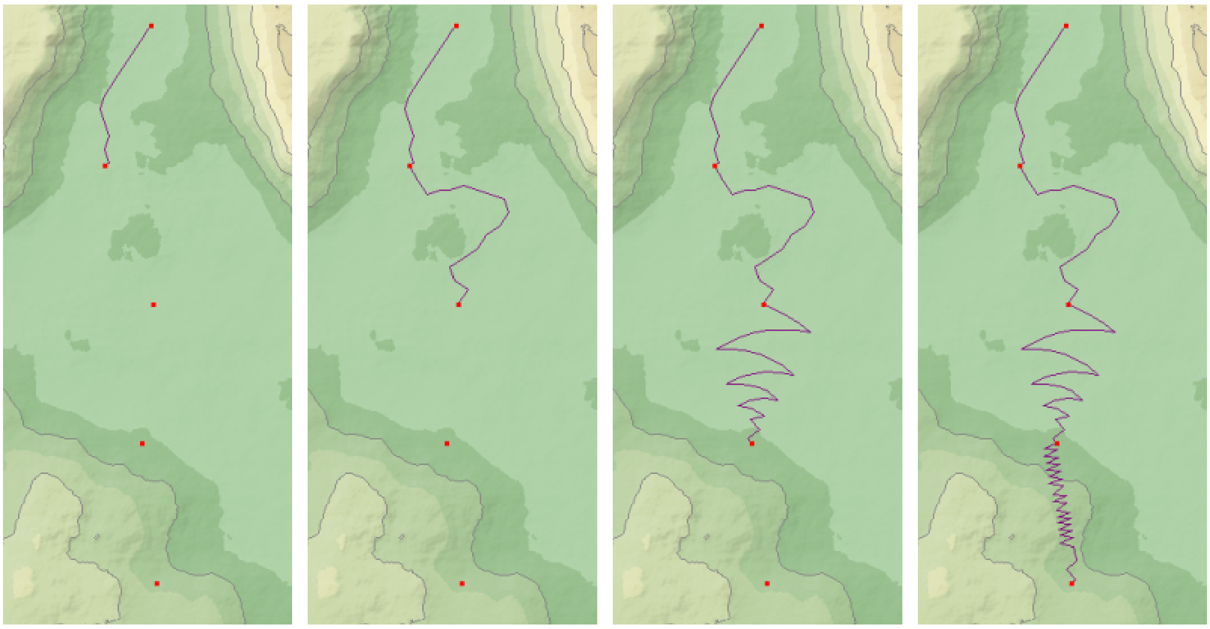


Figure B.4: An example situation where the road is divided into four sections. Each new section has different settings.

B.8.3 Blind mode

This mode is mostly the same as the grid mode. The only difference is that the darkened areas indicate that the terrain has not yet been generated. This was made to show that the terrain can be generated when needed.

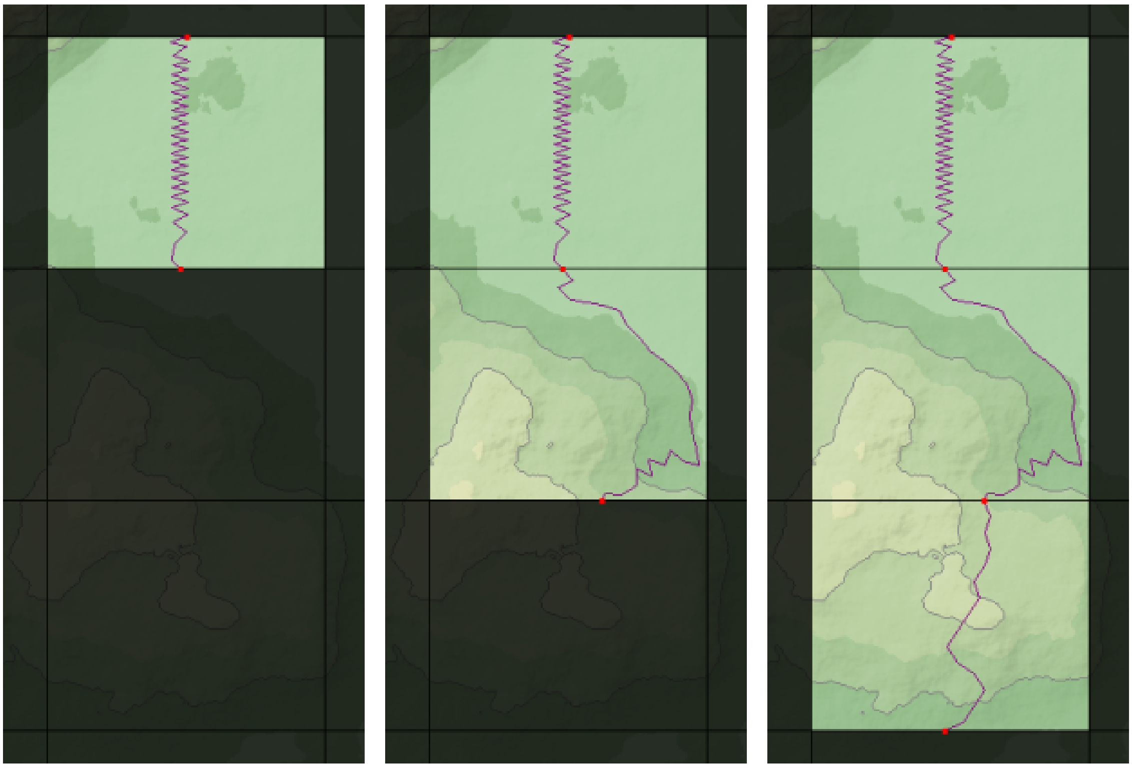


Figure B.5: Example situation where the terrain and road are generated when needed. Between blocks the settings, and therefore the roads, can change.

Appendix C: Competencies

C8	**Ontwerpen van een technisch informatie systeem**
Activiteiten	Het uitwerken van de systeemeisen (zie A5) in een ontwerp rekening houdend met de randvoorwaarden van de architectuur van het systeem (zie C10). In het ontwerp is voorts rekening gehouden met de algemene softwareontwerp principes, en die van het gebruikte ontwerpparadigma.
Product	Een ontwerp van het technisch informatiesysteem in een formele taal.
Toelichting	De formele taal is niet per definitie een (inter)nationale standaard, maar kan ook een bedrijfsstandaard zijn. Formele talen zoals UML, SFC, PSDs, flowcharts, OCL ... Het ontwerp omvat een Technisch Informatie Systeem (uit een TI-context). In het ontwerp is rekening gehouden met de algemene ontwerpprincipes van het gebruikte paradigma. Bijvoorbeeld: Bij het OO paradigma, let je op de koppeling en cohesie tussen bv packages, maak je zinnig gebruik van het Model View Controller pattern, houd je rekening met het Liskov substitution principe, ... Het technisch informatiesysteem kan ook een databasecomponent bevatten, bijvoorbeeld volgens het relationele model.
Toetscriteria	1. Beschrijvingen zijn zoveel als mogelijk in formele talen gemaakt. 2. De formele talen zijn correct toegepast. 3. De keuze voor de gebruikte techniek (de gekozen taal en de gekozen technieken binnen de taal) is verantwoord en correct. 4. De specificaties (zie A5) zijn aantoonbaar en correct verwerkt in het ontwerp. 5. Het ontwerp past binnen en houdt rekening met een ontwikkelstrategie. 6. Het ontwerp moet overdraagbaar zijn door onder anderen relevant commentaar. 7. Het ontwerp voldoet waar mogelijk aan de algemene ontwerpprincipes, en die van het gebruikte paradigma.

D16	**Het realiseren van software**
Activiteiten	Het programmeren van componenten van software.
Product	Broncode en uitvoerbare code.
Toelichting	Hoewel het eindresultaat van de taak de uitvoerbare code is, beschrijven we de taak aan de hand van de geschreven broncode. Afhankelijk van de mate van detaillering van het ontwerp (C8) kan er een overlap met die beroepstaak bestaan. Bijvoorbeeld stroomdiagrammen voor het ontwerp van de code. Zowel imperatieve als declaratieve talen (bijvoorbeeld een database vraagtaal) kunnen gebruikt worden. Het gebruik van de rijkheid van de taal uit zich in het de keuze van juiste instructies bijvoorbeeld foreach in plaats van for en enum in plaats van constanten.
Toetscriteria	1. De broncode sluit aan bij het ontwerp dat volgt uit C8 2. De broncode is syntactisch correct 3. De broncode is netjes georganiseerd a. Layout is leesbaar b. De naamgeving is zinnig en zinvol c. De stijl is consequent doorgevoerd d. Er is defensief geprogrammeerd e. Er is efficiënt geprogrammeerd f. Het commentaar is relevant g. Er is minimaal gebruik gemaakt van sequentie, selectie en iteratie 4. Er is gebruik gemaakt van de rijkheid van de taal.

H5	**Professioneel werken: Zelfstandig werken**
Activiteiten	Voorbeelden: initiatief vertonen, weinig sturing nodig hebben, niet snel in paniek raken, zelf taken signaleren, een eigen inbreng leveren, je eigen werk overzien als deel van een groter geheel, hoofd- en bijzaken scheiden, anticiperen op ontwikkelingen, dingen zien in hun context, e.d.
Product	Het product van zelfstandig werken is een vorm van gedrag van de student, dat kan worden toegepast op allerlei verschillende ICT-producten.
Toelichting	Professioneel werken houdt in, dat je opdrachtsituaties goed kunt beoordelen en dat je zelfstandig binnen de gestelde tijd en tegen beperkte kosten een prestatie met een hoge kwaliteit kunt leveren, die getuigt van creativiteit.
Toetscriteria	1. Er is gewerkt zonder dat derden vooraf de werkzaamheden gepland hebben (muv. de einddatum) 2. Er is gewerkt zonder dat projectmatige en inhoudelijke methoden en technieken de totstandkoming van het product op detailniveau zijn vastgelegd (vrijheid van handelen). 3. De student heeft zijn taak breder opgevat dan de letterlijke opdracht die hij heeft meegekregen (hij is in staat mee te denken met de opdrachtgever). Dit kan de student illustreren bijvoorbeeld dmv. adviezen die hij gegeven heeft om zaken anders aan te pakken of adviezen over vervolgstappen die genomen kunnen worden. 4. In de planning van het op te leveren product is aantoonbaar rekening gehouden met de hoofd- en bijzaken die betrekking hebben op de totstandkoming van het product.

H6	**Professioneel werken: Resultaatgericht werken**
Activiteiten	Voorbeelden: een prestatie leveren, werken volgens planning, doorzettingsvermogen tonen, werken met deadlines, slaagkans en leveren vooropstellen, maatregelen nemen die de kwaliteit van werkwijzen en producten verhogen, degelijk en accuraat te werk gaan, kosteneffectief te werk gaan, etc.
Product	Een geprioriteerd programma van eisen zodat voldaan is aan de kwaliteitseisen van de de opdrachtgever. Een planning gebaseerd op deze prioritering. Een product dat volgens planning opgeleverd wordt.
Toelichting	Professioneel werken houdt in, dat je opdrachtsituaties goed kunt beoordelen en dat je zelfstandig binnen gestelde tijd en tegen beperkte kosten een prestatie met een hoge kwaliteit kunt leveren, die getuigt van creativiteit.
Toetscriteria	1. Het product is op tijd afgeleverd en voldoet aan de kwaliteitseisen van de opdrachtgever. 2. Als het product niet het volledige product is, dan wordt verantwoording afgelegd over de gemaakte keuzes. 3. Er is aangetoond dat er bij het maken van keuzes rekening gehouden is met de beperkingen van tijd en budget. 4. Er is aangetoond dat de kwaliteit van het product en de planning van het proces bewaakt is.

Appendix D: Developers manual

Description of classes

RoadPlanner is an abstract class which was made to make both planners consistent. It contains the cost functions and an abstract version of the getPath method, as well as the methods for using sections.

Both SequentialPlanner and StarPlanner inherit from RoadPlanner. They have an override for the getPath function, which is implemented differently for both of the planners. Because they both inherit from the same class, they are interchangeable.

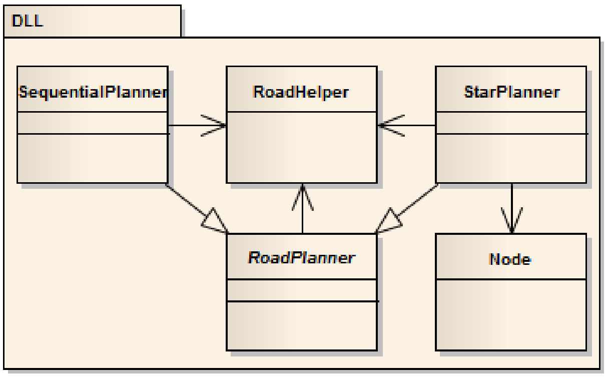


Figure D.1: Simplified class diagram of the DLL.

RoadHelper is a static class which contains some functions used by several other classes. This was created to keep the code short, consistent and orderly. This class contains a number of functions such as, for example, point to index, index to point, distance between two points and making a cardinal spline.

The Node class is used to represent a point with some additional properties. It contains a G- and H-value and can calculate the F-value. Additionally, it has a previous node stored, which allows the user to find a path.

Calling a pathplanner

A simple call to the pathplanner is done like this:

```
SequentialPlanner plan = new SequentialPlanner(elevationMap, water, terrainLength);
plan.setFactors(neighbours, factorDist, factorSlope, factorSmooth, factorDither, factorRandom);
List<Point> path = plan.getPath(start, end);
```

When calling the constructor, three parameters are needed:

- An array of floats containing the heights of the map. Currently, only square maps are supported.
- A list of polygons with all the bodies of water.
- An integer which indicates both the length and width of the map.

The function setFactors has several parameters. The first one is the amount of neighbours to check each cycle. For the SequentialPlanner any value between 5 and 15 is recommended. The StarPlanner, however, works only with powers of two. Only the values 4, 8, 16 and 32 are accepted. The other parameters indicate the weights of the cost function. The value for distance can be any value from 1 and higher. The uncertainty weight (*factorRandom*) can have any value between 0 and 1. The other parameters accept values from 0 and higher.

Finally, the getPath function can be called. When starting a new road, only the start- and endpoint are needed. To expand existing roads, it is preferable to give some of the previous points, so a better transition can be made.

Other features

The XML-config file can be loaded by putting *settings.xml* in the same folder as the DLL and calling: `Globals.loadXML();`

If the segment that has to be generated is not the first segment, it might need to account for a smooth transition. In that case, the call to the getPath-function changes:

```
List<Point> path = plan.getPath(start, end, prev1, prev2, prev3, prev4);
```

The post-process smoothing should be called like this:

```
List<Point> smoothpath = RoadHelper.getSpline(path);
```

Making sections for the free mode discussed earlier (in section B.8.2) can also be called. A call to this function would look like this:

```
List<Point> nodes = plan.makeSection();
if (nodes.Count == 0)
{
    plan.defineSections(start, end);
    nodes = plan.makeSection();
}
```

If you want to start over with defining sections or if you are done, a simple reset will suffice:

```
plan.resetSection();
```