Examensarbete
LITH-ITN-MT-EX--04/072--SE

Avoiding cracks between terrain segments in a visual terrain database.

Hanna Holst

2004-12-17

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--04/072--SE

Avoiding cracks between terrain segments in a visual terrain database.

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Hanna Holst

Handledare Matt Cooper
Examinator Matt Cooper

Norrköping 2004-12-17

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2004-12-17

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ _______________	**Rapporttyp** Report category Examensarbete ☐ B-uppsats ☐ C-uppsats ☒ D-uppsats ☐ _______________	**ISBN** _____________________ **ISRN** **LITH-ITN-MT-EX--04/072--SE** **Serietitel och serienummer** **ISSN** Title of series, numbering _______________

URL för elektronisk version
http://www.ep.liu.se/exjobb/itn/2004/mt/072/

Titel
Title Avoiding cracks between terrain segments in a visual terrain database.

Författare
Author Hanna Holst

Sammanfattning
Abstract To be able to run a flight simulator a large area of terrain needs to be visualized. The simulator must update the screen in real-time to make the simulation work well. One way of managing large terrains is to tile the area into quadratic tiles to be able to work with different detailed representations of the terrain at different distances from the user. The tiles need to match with the adjacent tile in their edge points to avoid cracks. When every tile can have a number of different levels of detail this means that the different levels of details need to match too. This was previously done by having the same edge points in all levels of details, giving an unnecessarily large polygon count in the less detailed levels. The method developed in this report uses different versions of borders with their own level of detail, adapting to different detail levels at the adjacent patch, reducing the number of polygons.

Nyckelord
Keyword terrain generation, landscape generation, level of detail, tile, patch, virtual world, simulator, VR, Visualization

Abstract

To be able to run a flight simulator a large area of terrain needs to be visualized. The simulator must update the screen in real-time to make the simulation work well. One way of managing large terrains is to tile the area into quadratic tiles to be able to work with different detailed representations of the terrain at different distances from the user. The tiles need to match with the adjacent tile in their edge points to avoid cracks. When every tile can have a number of different levels of detail this means that the different levels of details need to match too. This was previously done by having the same edge points in all levels of details, giving an unnecessarily large polygon count in the less detailed levels. The method developed in this report uses different versions of borders with their own level of detail, adapting to different detail levels at the adjacent patch, reducing the number of polygons.

Appreciations

I want to thank the people at Saab for helping me. I want to thank my supervisor Andreas Ekstrand for his support and thoughts about the report. I also want to thank Mats Björkman for all the help with the programming. One may think that you were my supervisor. Thank you very much. Thanks to Andreas Ladell for fighting through the report even though busy with work and for the help with viewing the results. Also thanks to Magnus Toneby for his last minute review of the report.

Table of contents

1 Introduction

This chapter is an introduction to the rest of the report. It gives an introduction to the problem and the planning of the work. For best understanding the reader should have some knowledge in the visualization field.

1.1 Background

Simulators are used to try out new technical solutions before they are built in reality. They can also be used for people to get training in doing complicated tasks without risking themselves or their equipment. To get results out of these tests or training sessions the computer generated simulation needs to agree with the real world as much as possible.

The most important thing to simulate in a simulator is the behavior of the actual aircraft. A more or less detailed representation of the aircraft can be used. To add some realism, an outside world can be added. More or less detailed representations of the objects in the world can be used. For a close up view of a house all windows, ornaments and structure of the walls must be present for it to look real, but from a very long distance it may be sufficient to draw it as a box with a color.

There are some different ways to create a virtual world for a computer. The most common is to use polygons. A polygon can be any of many different forms but the most popular polygon to use is the triangle since it is easy to do calculations upon. A complicated world needs a lot of polygons to look real from close up. But it gives more for the computer to calculate at each time the image on the screen needs to be updated. There is a trade off between how real we want the world to look and how much triangles the computer can handle. More detail gives more calculations but increases the realism, less detail gives faster simulation but decreases the realism.

This report describes the handling of the terrain of the virtual world. In flight simulators the terrain must be very large because the aircraft can fly at high altitude and therefore can see very far. To create a terrain that matches some part of the real world a regular grid of measured height values are used and from those a surface of triangles is created. If no prioritization is done in choosing points to triangulate, all points will be used and there will be a huge polygon count. It is impossible for the computer to draw all these

polygons at every update. Therefore it is necessary to reduce the number of polygons by carefully choosing which points to triangulate.
In the DMGS (Digital Map Generating System) at SAAB AB in Linköping the triangulation is made with Delaunay triangulation [1, (14, 15)] creating a TIN (Triangulated Irregular Network). The terrain is split into a number of quadratic sections, tiles (figure 1-1a), saved in different files. Every tile is triangulated in a discrete number of LODs (Levels of detail), figure 1-1b. If the terrain is viewed from high altitude less detail is shown and more detail is shown if viewed from low altitude.

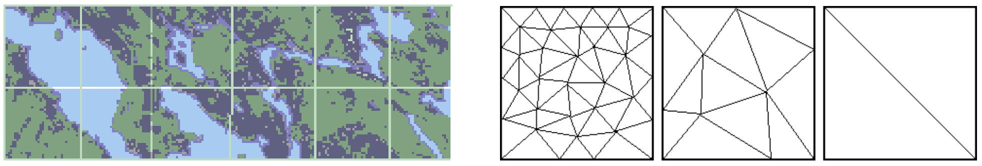


Figure 1-1: Tiles and levels of detail (LOD) (a) The world is split into a number of quadratic squares called tiles. (b) Each tile can have a number of LODs with a different number of triangles each.

The reason for using tiles is that it is the smallest entity of the terrain that can consist of different LODs. The reason for using LODs is to keep the polygon count down. If the virtual world consists of for example a whole country with sufficient level of detail for low altitude flight, the terrain would consist of a large amount of polygons, too many for the computer to handle at each frame. With the use of LODs, parts of the terrain that are further away from the user can be shown with fewer polygons. Another reason for using tiles is that if the virtual world is large the whole world can not be kept in memory at the same time. If tiles are used the computer can load the necessary parts of the terrain into memory when needed instead of loading them all at the start of the simulation.

The tiles must match at their connections, not to create any cracks in the terrain. This means that two connecting tiles must have the same points on the shared edge. Since the world is in 3D the point not having a match on the other tile can be at a different altitude from the line connecting the nearest points on the other tile, forming a triangular opening in the terrain where the background can shine through giving ugly-looking results. This is illustrated in figure 1-2.

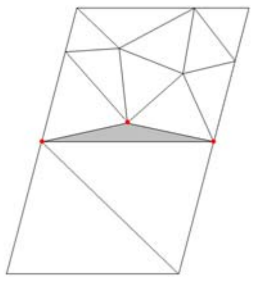


Figure 1-2: If the points along the shared edge are not the same in both tiles a triangular opening can appear through which the background can shine through.

The different levels of detail have different distances from the viewer which define when to switch in and out. Since this distance is based on the distance between the viewer and the tile center two adjacent tiles can have different levels of detail. If there are to be no cracks between the tiles this means that even the different levels of detail have to match in their connections.

1.2 Problem

The method with the terrain split in different quadratic sections gives rise to the problem of connecting these tiles to each other.

This can be done by forcing the points on one tile edge to be present also in the adjacent tile but, when the difference in detail is added, the tile must not only match the adjacent tile but also its different levels of detail. This can be solved by forcing all the levels of detail to have the same edge points. If the edge points are taken from the most detailed level it will lead to an unnecessarily large number of polygons at the least detailed level, looking like fans along the edges. This is unwanted since it slows down the application. If, instead, the edge points are taken from the least detailed level it will result in too sparse triangulation along the edges of the more detailed levels, resulting in too few polygons in that area and the tile edges may look like ditches. This is illustrated in figure 1-3.

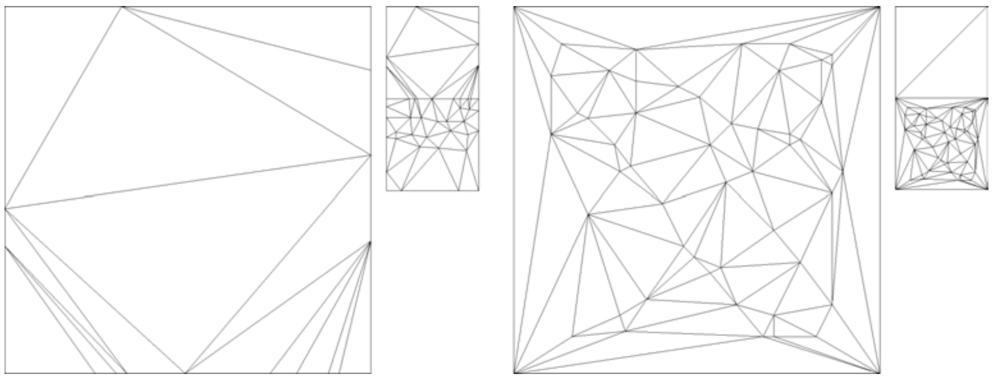


Figure 1-3: Matching edge points (a) Edge points taken from the most detailed LOD, causing fans to appear along the edges. (b) Edge points taken from the least detailed LOD, causing large polygons to appear along the edges.

One other solution, illustrated in figure 1-4, is to not match points but to push the less detailed versions of the tile some distance into the ground. By doing this you do not need the unnecessary polygons around the edges. Since you always look from the more detailed version the cracks between the tiles are hidden, but not removed. You still have to match the tile edges of the same level of detail.

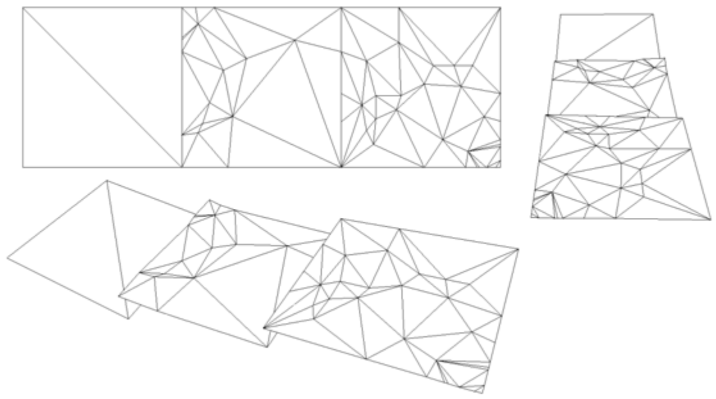


Figure 1-4: Push-down into the ground method.

The top left image is a top view. No cracks in the terrain are visible from above but notice the T-vertices. The bottom left image illustrates how the tiles are pushed down into the ground. The viewer is situated at the bottom right corner. The right view shows how the cracks are not visible since the lower tiles are further away from the viewer. The downside is that, in rough terrain, it can look unnatural since a substantial lowering is needed.

The solutions above are the ones used in the DMGS system right now. None of the above solutions is perfect, creating the need for finding better solutions. The desired method avoids cracks but does not require edge-point matching in a way that creates the problems above. Since the most commonly used method is to adapt edge points to the most detailed level this means in practice to reduce the number of polygons. This is preferably done with no or little change in the runtime systems.

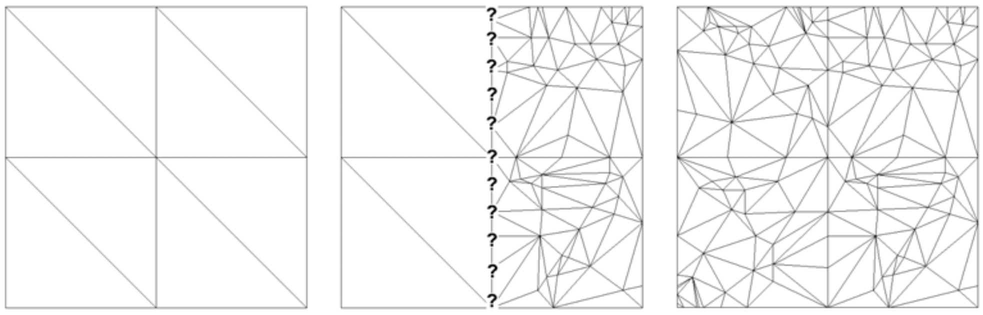


Figure 1-5: Wanted method allows less detailed tiles to meat with the more detailed tiles seamlessly.

Figure 1-5 shows the desired method. It will make it possible to have only two triangles in the terrain of the least detailed LOD, shown to the left, and at the same time be able to have more points on the edges of the most detailed LODs, shown to the right. The join between the two different levels of detail is required to be seamless. To be able to do this something has to be done in the transition between different LODs, shown in the middle image.

1.3 Method

At the outset this project was to be an evaluation project with possible implementation in the long run, but it turned out to be an implementation project with a short evaluation as foundation.

Research of known methods was done and the methods found went through a quick evaluation to decide which were worthwhile to examine further. One method was examined further and some solutions were developed based on the chosen method. One solution was then chosen for test implementation into the Saab DMGS. The implementation was carried out and the result was tested and compared to the original implementation. The comparison was made with respect to generation time, visual appearance and runtime performance.

1.4 Limitations

The solutions of this report work only on a surface that is split into quadratic tiles. No methods that require changes to the runtime system were examined.

1.5 Document overview

Chapter 2, “Important concepts”, introduces some concepts that are used later in the report. The chapter could be read before the rest of the report or be used to lookup concepts later on. Readers with little knowledge in the visualization field are recommended to at least glance through the concepts before proceeding with the rest of the report. Details can be looked up later.

Chapter 3, “Description of the system”, describes the systems used at SAAB for generating virtual worlds and running the simulation. Those systems are the starting points for the solutions and implementation presented in chapters 5 and 6. This chapter should at least be glanced to before continuing with the rest of the report to get an idea about the conditions under which the work of this report is done.

Chapter 4, “Known methods and quick evaluation”, examines some known methods used to solve or work around the problem. The chapter contains quick evaluations of all the methods making up the base for choosing methods to look deeper into. This chapter can be read independently of the other chapters but it is recommended to be read before proceeding with the following chapters.

Chapter 5, “Developed solutions”, presents the solutions created within this project. This chapter should be read carefully before reading chapter 7 to be able to understand what is done and why.

Chapter 6, “Implementation”, describes the preparation for implementing the chosen solution and the changes carried out in the implementation. To understand this chapter it is recommended that at least chapter 5 have been read before.

Chapter 7, “Results”, compares the results of the output created by the new system with the original system with respect to generation time, appearance and runtime performance.

Chapter 8, “Conclusions”, summarizes the report. It discusses the results of the report. It also discusses what could have been done differently and what is left to be done in the future. It is recommended that this chapter is read last.

2 Important concepts

This section describes some important concepts that are used in the report and/or improves the understanding. It can be read before the rest of the report or be skipped depending on the reader’s knowledge in the field. It can also be used to look up concepts when needed.

2.1 Simulator

To be able to understand the problem and meaning of the work done in this report it is important to understand where it is used. The target for the result in this report is a flight simulator. It is not specialized on any specific simulator and it could as well be used in car simulator or in a computer game. For better understanding an example of a simulator follows.

A simulator [5] of an aircraft can consist of a physical model of an aircraft, more or less coherent with a real aircraft, and some sort of screen on which a virtual world can be displayed. The behavior of the aircraft is simulated by a mathematical model. The physical simulator is a tool to let the user control the aircraft in a similar way as in the real aircraft.

Figure 2-1 shows an example of a simulator used at Saab. It consists of a cockpit and a dome onto which the virtual world is projected. The cockpit is a cabin with a stick, pedals and a throttle. It also has some displays to show the instrument readings on. This is not a physical replica of a whole aircraft, but contains the most important features needed to complete its task.

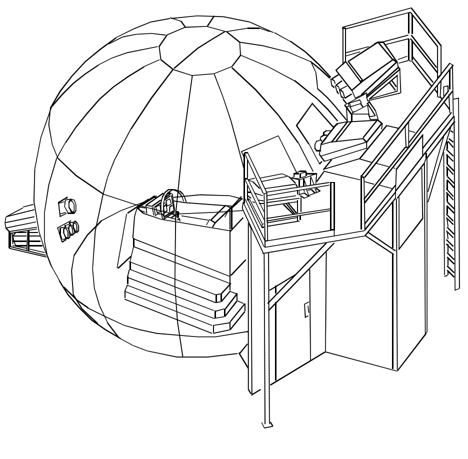


Figure 2-1: Simulator – Model of an airplane cockpit in the center of a spherical dome onto which the simulated world is projected by a number of different projectors.

The dome is six meters in diameter with the cockpit placed in the center. The virtual world is projected onto the dome by a number of different projectors situated outside the dome, projecting onto it through holes.

2.2 Virtual world

To be able to use a simulator where the user can look outside the aircraft and see what would be seen from a real aircraft a virtual world is needed. A virtual world is a replicate of the real world created by computer graphics and displayed on a screen. All features outside the physical aircraft of the simulator are part of the virtual world. The terrain, the sky and all objects like trees, houses and vehicles. The virtual world is called OTW, Out The Window.

You can separate the world into generated objects and modeled objects [2]. Generated objects are created by an automated process and the modeled objects are created by hand, in a modeling program. In the system [3] described in this report the terrain is generated and the trees, houses and other objects are modeled. The modeled objects are placed onto the terrain through an automated process. The terrain generating system at Saab is modified in this project as well as the placing of the modeled features onto the terrain.

2.3 Programs used

2.3.1 ArcMap and DMGS, Digital Map Generating System

The test data of this report was created in a program called ArcMap. It is a program for handling geographical data. It is built on the foundation of GIS (Geographical Information System) that is a way of storing and presenting geographical data. The Saab OTW generating system is built into ArcMap [6, 7] and uses its data for creating the virtual world.

The Digital Map Generating System, DMGS [3], is a system for producing all types of maps for different use in a simulator or a real aircraft. It is created by Saab and built into the program ArcMap. The input into the system consists of geographical data presented in a number of different formats and is to be the same for all types of output, making them all correlated.

The output can consist of a map consisting of only the height values of the terrain for example the instruments that measure height over ground in the simulation or the whole virtual world shown in the dome. The different outputs are produced by different modules. The virtual world is generated by the OTW (Out The Window) export module. This module is the one modified within this project. Figure 2-2 shows ArcMap and the DMGS OTW menu.

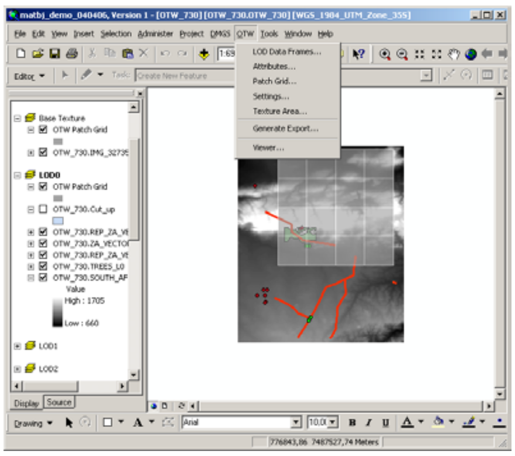


Figure 2-2: ArcMap and the DMGS OTW module

2.3.2 Multigen Creator

Multigen Creator [8] is a modeling program for constructing real-time 3D models. The program is used by Saab to create the models present in the simulations. Creator is used in this report to model simple surfaces for testing of the solutions. It is also used for viewing the result generated by DMGS.

2.3.3 GRAPE

Grape is the graphical engine constructed at SAAB in Linköping. It is used to view the results generated by DMGS. This, unlike Multigen Creator, is just a visualization system and therefore specialized for showing the graphics. Grape is the main target for the virtual environments created by DMGS. Read more about Grape in section 3.2.

2.4 Data

To create the virtual world requires data from the real world if we want it to look as much as the real world as possible. Data can be collected and represented in a lot of different ways. This chapter presents the most important forms of data used by the ArcMap and the DMGS. Most of the data is bought in from different places and not created by Saab.

2.4.1 Raster data

Raster data [7] contains values everywhere in the data. It is built up by a grid of cells. Each cell is filled with a sample of data or an interpolated value, see figure 2-3.

Raster data values are often created from discrete data, values measured only at certain positions either on a regular grid, se figure 2-4, or irregularly spaced. The cell raster data is created by interpolating values from the known position values.

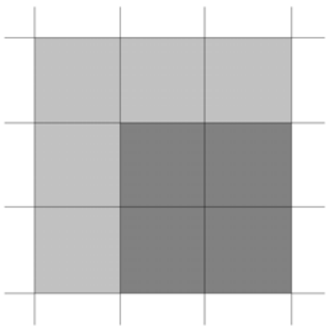


Figure 2-3: Raster grid

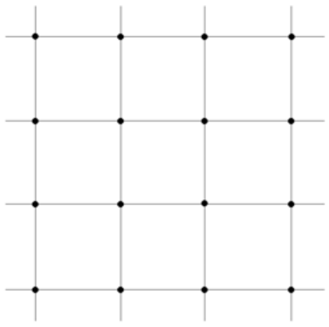


Figure 2-4: Discrete value grid

An example of point raster data is the measured elevation values used in DMGS [3] to create the terrain. It can be treated as cell raster data if values are interpolated to the cells from the corner points.

Another example of raster data is the photographs used to texture the terrain. When texturing the terrain every polygon in the terrain gets a part of a photograph attached onto it. Satellite photos are low resolution photos used to texture the terrain from high altitude and areas of little interest. Aerial photos are used for texturing the terrain when and where better resolution is needed, for example around airports and other areas of interest. The texturing process in DMGS is finished and is only needed to be modified for the solution of this report.

2.4.2 Vector data

Vector data [7] handles different features. It can represent types of terrain, roads or measurements. Vector data has a position in the world and information about the feature it represents.

Vector data can consist of lines circumscribing an area of a specific type. Examples of where this type of data is used in DMGS [3] are for lakes, forests and fields. This can be used for giving those areas special treatment in the generation process.

Line data is a type of vector data that forms lines in the landscape. Line data is used for creating, for example, power lines or roads.

Point data is also a type of vector data. It represents values that only exist at a specific position, for example the position of a house, mast or a runway in the DMGS.

2.5 Scene graph

The concept of a scene graph is the most important thing to understand for in order to be able to understand the developed solutions in this project since it is the foundation of all of the solutions.

A scene graph [1, 8] is a tool for keeping order in the 3D world. The scene graph is a tree structure that controls the rendering process. In the tree different types of objects, or nodes, are placed according to the desired drawing order. Which types of nodes exist is dependent on the scene graph used, but most scene graphs build on a similar set of nodes. Examples of nodes can be geometry nodes, graphic properties nodes, transformation nodes, light source nodes and LOD nodes. A node affects all nodes below it in the tree. This represents a way of doing things just once instead of doing it each time it is needed.

The scene graph used in this report is the one used by Creator and Perfly. The important nodes for this report are the LOD nodes. What LOD means is presented in the next section.

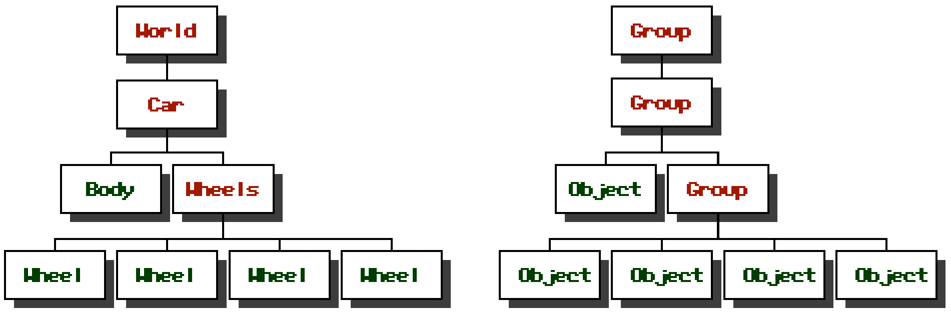


Figure 2-5: Simple scene graph example

A simple scene graph example is shown in figure 2-5. The car is in the world. Everything that affects the world will affect the car. The things affecting the car will not affect the world. The car has a body and wheels. If the car moves the wheels and the body will move. In this example only group nodes and object nodes are used. Notice how the four wheels are connected by the group wheels.

2.6 Level Of Detail, LOD, of objects and terrain

The previous section mentioned LOD (Level Of Detail), nodes as being the most important ones for this project. The concept of level of detail and the corresponding nodes in the scene graph makes up a cornerstone of both the problem and the solutions in this report. This section explains what level of detail is and why it is needed.

To represent a whole world with good accuracy a large number of polygons are needed [2], too large for the computer to keep in memory and handle fast enough. The solution is to not use the same resolution for the whole world, but there is still the need to be able to show an object with high resolution when the viewer is close to it. This can be possible by building different versions of an object with different resolutions, called *Levels Of Detail* [10, 8]. An object far away from the user only covers a few pixels on the screen and can therefore be shown with only a few polygons, while an object close to the viewer needs a lot of polygons to look good. An example of an object in different levels of detail is shown in figure 2-6.

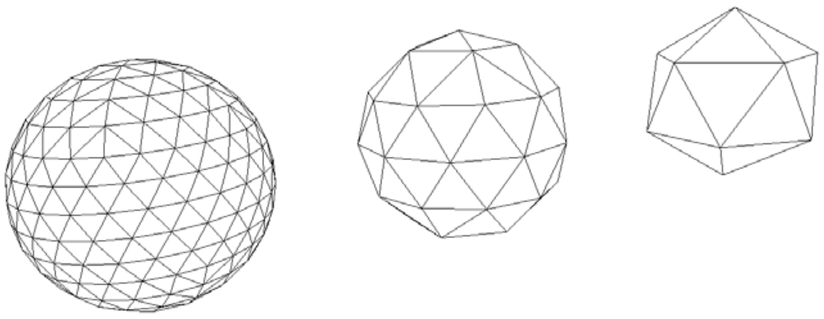


Figure 2-6: Level of detail - Different resolution versions of the object can be used depending on the distance to the viewer.

Levels of detail in terrain generation are more complicated since it covers an area from close to the viewer all the way to the horizon [13]. To handle level of detail in terrain the terrain is split into quadratic parts called tiles. Each tile has different levels of detail.

The LOD of a tile (or an object) is determined by three parameters [8], illustrated in figure 2-7 and explained below.

Near range –	How close to a tile (or an object) you can get before it changes to a more detailed level.
Far range –	How far from a tile (or an object) you can come before it changes to a less detailed level.
LOD-center –	The point in the tile (or object) from which the distance are measured. Does not necessarily need to be in the center of the tile (or object).

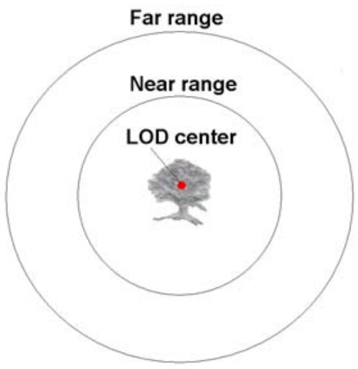


Figure 2-7: Level of detail parameters

A fourth parameter called *transition* may be used to hide the visible effects when a LOD is switching. The *transition* is a distance around the near and far ranges within which fading between the two LODs occur. More about fading can be read in section 2.9.

2.7 Triangulation

To create the terrain raster data is used [3]. From the data the triangulation process creates a surface of triangles. The triangulation in the SAAB generation system has already been done before this project and no changes were made to the triangulation structure or algorithm but to make the changes some understanding of the features presented in this chapter was needed. There is no need for the reader to know all the details of this section but it can increase the overall understanding of this project.

The raster data can have values in two or three dimensions and be manually created or surveyed from the real world. For the terrain in a 3D flight simulation we want three dimensions and real world data to make it as realistic as possible.

There are several more or less automatic algorithms for choosing which points to triangulate. The different algorithms give different results and which you use depend on what result you want.

2.7.1 Triangulation structures

There are several types of structures [11, 2, 1, 10]. The most commonly used is the polygon mesh. It is not used in the SAAB system. Instead a more flexible solution is used, called TIN, Triangulated Irregular Network.

The decision to use a TIN instead of a polygon mesh affects the search for a method that solves the problem attacked in this report because of it is a more complicated structure.

2.7.1.1 Polygon mesh

A polygon mesh [11, 2, 1, 10] is a regular structure built up of triangles from a regular grid of points. An example of a polygon mesh is shown in figure 2-8.

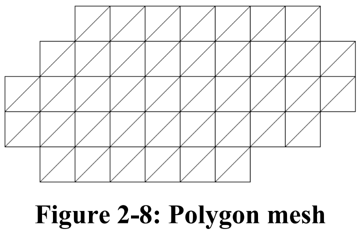


Figure 2-8: Polygon mesh

Because of its regular structure the polygon mesh is easy to construct. The grid of points is easily triangulated but to look good the result contains a large number of polygons, hard for the computer to handle. To produce a smooth simulation the number of triangles has to be reduced, for example by triangulating only a portion of the points in the grid. This result gives fewer polygons but often lacks detail in more complicated areas.

2.7.1.1 TIN – Triangulated Irregular Networks

A triangular irregular network [11, 1, 10] is, as it sounds, an irregular structure. All the triangles can be of different sizes and shapes. The advantage of a TIN over a polygon mesh is that you can triangulate sparsely over areas with small height differences and more densely in rough terrain. In that way you can save polygons in plain areas and use them where they are needed more. This means that you can get a nicer terrain with fewer polygons. An example of a TIN is shown in figure 2-9.

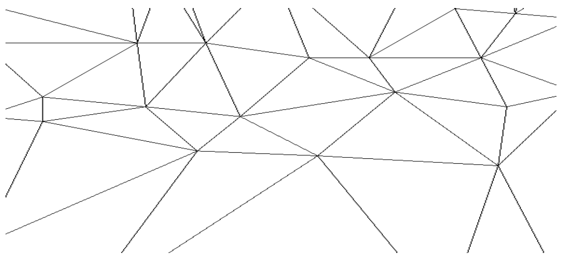


Figure 2-9: TIN – Triangulated Irregular Network

The downside is that the TIN is more difficult to generate. You need an algorithm that chooses a portion of the points to triangulate to give the result you need. There are two ways to solve the triangulation. You can start with a lot of polygons and gradually remove the points that give the least error when removed. Or you can do it the opposite way, adding the points that differ the most from the surface you already have. SAAB uses the second of these methods.

2.7.2 Delaunay triangulation

The triangulation in the DMGS is done by progressively adding points that differ the most from the surface already present. To make the triangulation as good as possible Delaunay triangulation [1, (14, 15)] is used. The goal is to avoid sharp angles, making all angles as similar as possible. Sharp angles can give horrible looking results when the surface is shaded. Very slim triangles are called slivers and are also not appreciated. Read more about slivers later in this chapter. The Delaunay triangulation is a widely used method known to give good results.

When a triangle is created there must not be any other point inside the circle that is formed between the three points of the triangle. This is called the Delaunay criterion and is shown in figure 2-10.

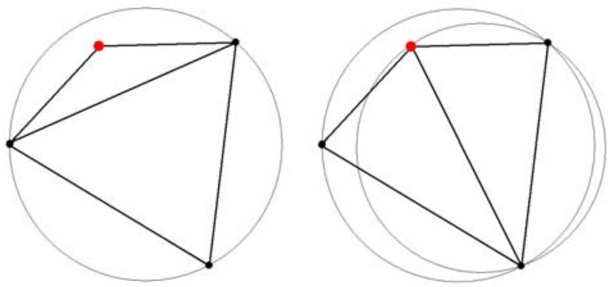


Figure 2-10: Delaunay criterion (a) The new point violates the Delaunay criterion. (b) The point is retriangulated so that the Delaunay criterion is fulfilled.

The points are added depending on their elevation values. (This is not really part of the Delaunay triangulation, but is often used together with the Delaunay triangulation to add a third dimension to the otherwise 2D Delaunay algorithm.) The point where the surface differs the most from the data is inserted first. When the point is chosen in 3D space the graph is projected down to 2D (the z component is thrown away), the Delaunay triangulation is performed and the z component is inserted again.

2.8 Artifacts

2.8.1 Slivers

Slivers [16, 17, 18] are very sharp triangles that are undesirable in triangulation. The slivers can be seen as bad looking, sharp areas brought out by the shading. Even a good implementation of the Delaunay triangulation does not completely prevent slivers.

The slivers, since they are so thin, do not contribute much to the area of the terrain, giving an unnecessarily large number of triangles, and are therefore seen as ineffective. The slivers have too many negative effects and so are unwanted. Figure 2-11 shows an example of slivers.

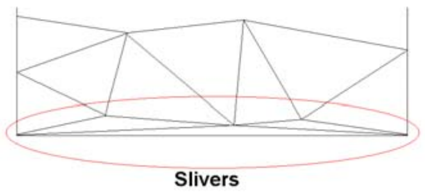


Figure 2-11: Slivers

2.8.2 T-vertices

T-vertices, or t-junctions [11, 19], occur when a point lies on a triangle edge instead of meeting the triangle at a corner point. They can give rise to cracks in the terrain. The t-vertices should not give any cracks if the points are at the same elevation as the polygon edge, but due to rounding issues in reality they may cause tiny cracks and are therefore unwanted.

T-vertices, figure 2-12, occur when points of one tile edge are not present on the adjacent tile edge. This is the main part of this project. There must under conditions be any T-vertices in the solution provided by this report.

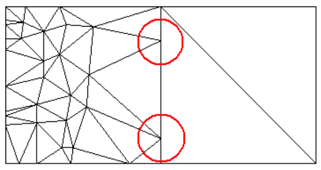


Figure 2-12: T-vertices

2.9 Tiles and paging

The reason for dividing the terrain into squares, called tiles, is that the computer needs to handle large terrain. If the terrain is too large the computer can not fit the whole terrain into memory at once. When tiles are used the system can load the tiles successively when needed, instead of loading them all at once at startup. This is called paging [4]. Tiles also make it possible to have different levels of detail depending of how far from the viewer the terrain is.

The tiles have to match at their edge points, avoiding t-vertices, not to give cracks in the terrain. This is especially tricky when using LOD and this is what the main part of this report is about, explained in the problem description.

2.10 Quad-tree

Quad-trees are used in the DMGS, but they do not play a major part of this report. But since the concept is mentioned it may be interesting to have an idea of what it is about.

A quad-tree [3] is a typical approach for speeding up a spatial search. The area, in this case the terrain, is split into smaller parts by splitting each part into four new parts progressively forming a tree. When searching for a particular position in the tree it is not necessary to search through the whole area as would be needed if no tree is used. Only the branch leading down to the leaf containing the position needs to be searched since it is easily decided at every division of the tree in which quadrant the position is situated. Subdivision into a quad-tree is shown in figure 2-13.

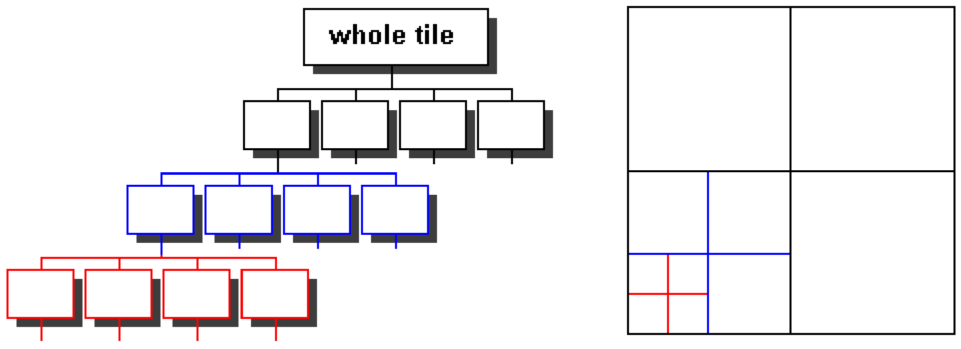


Figure 2-13: Quad-tree subdivisions

2.11 Popping, fading and morphing

When an object or terrain tile is switched to another LOD the geometry changes. If the switching is done too close to the viewer you can see the terrain changing from one second to another. This phenomenon is called popping [20]. One solution is to move the popping so far away that the user is not able to notice it, but most of the time we want the switching to happen as close as possible for speed reasons.

To be able to switch as close to the viewer as possible, fading and morphing can be used. Fading [8] shows both LODs, fading between the two for some time. Morphing [1, 20, 21] slowly moves the points of the second LOD from a position on the first LOD surface until they have reached their final positions. Fading is used in the SAAB simulation and looks good as long as the computer can handle the transparency blending.

2.12 Naming conventions for this report

To get structure in the report some naming conventions were made. To not be confused when reading further, memorize the difference between borders and edges.

2.12.1 Borders or edges

In this report the word edge is used to describe the outline of a feature such as a terrain tile. Border is used to describe an area, especially the polygon collection around the edge of the tile used to form the solution of this report. See figure 2-14 for better understanding.

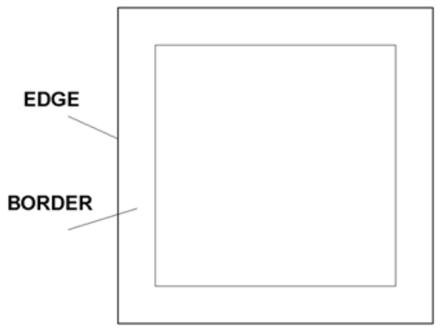


Figure 2-14: Borders and edges

2.12.2 Directions in the tile

Different parts of a tile are referred to by their relations to the tile center – west (W), north (N), east (E) and south (S), shown in figure 2-15. These names have nothing to do with direction in the real world. West part is simply the one to the left of the patch center. (This is used for the borders in the solution of this report.)

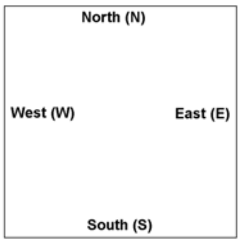


Figure 2-15: Directions

2.12.3 Level of detail number

To keep track of the different levels of detail they are enumerated. The most detailed LOD is called 0 and increasing number means decreasing detail, illustrated in figure 2-16.

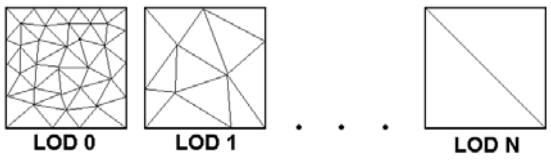


Figure 2-16: LOD number convention

3 Description of the system

The system for creating and displaying the OTW view can be divided into two parts. The first is the generation of the environment and the second is the runtime system. The modifications proposed and implemented in this report operate only on the generation part, but to understand the conditions for finding the solutions some knowledge of the runtime system is also needed.

3.1 Generation system, DMGS – Digital Map Generating System

This chapter handles the OTW, Out The Window, export module in the DMGS, Digital Map Generating System [3]. This is the program that has to be adapted to introduce the changes proposed in this project.

The DMGS OTW export module collects different kinds of geographical data and previously created models and generates a virtual world, written to files in the OpenFlight format [9] that can be loaded into a real time 3D engine for simulation. Figure 3-1 shows DMGS with the OTW menu.

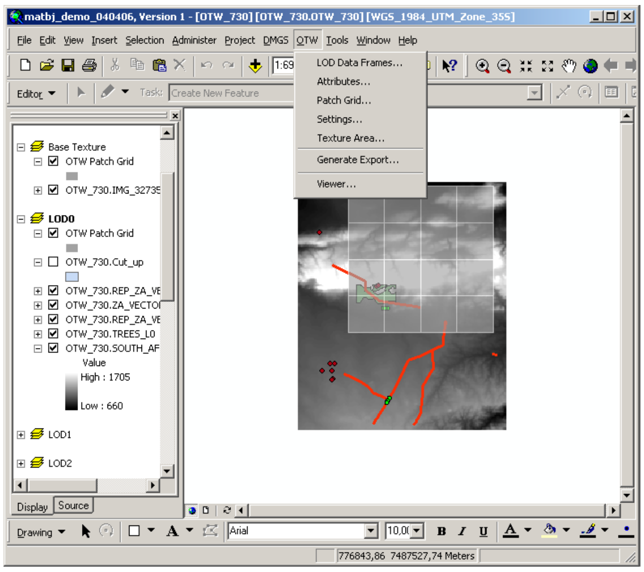


Figure 3-1: DMGS and the OTW module

3.1.1 Overview of the system

To be able to follow how the generation process proceeds, an overview of the system is presented in this section. More detailed descriptions of some concepts are presented in next sections. Figure 3-2 shows a diagram of the generation process. It is further explained in the text.

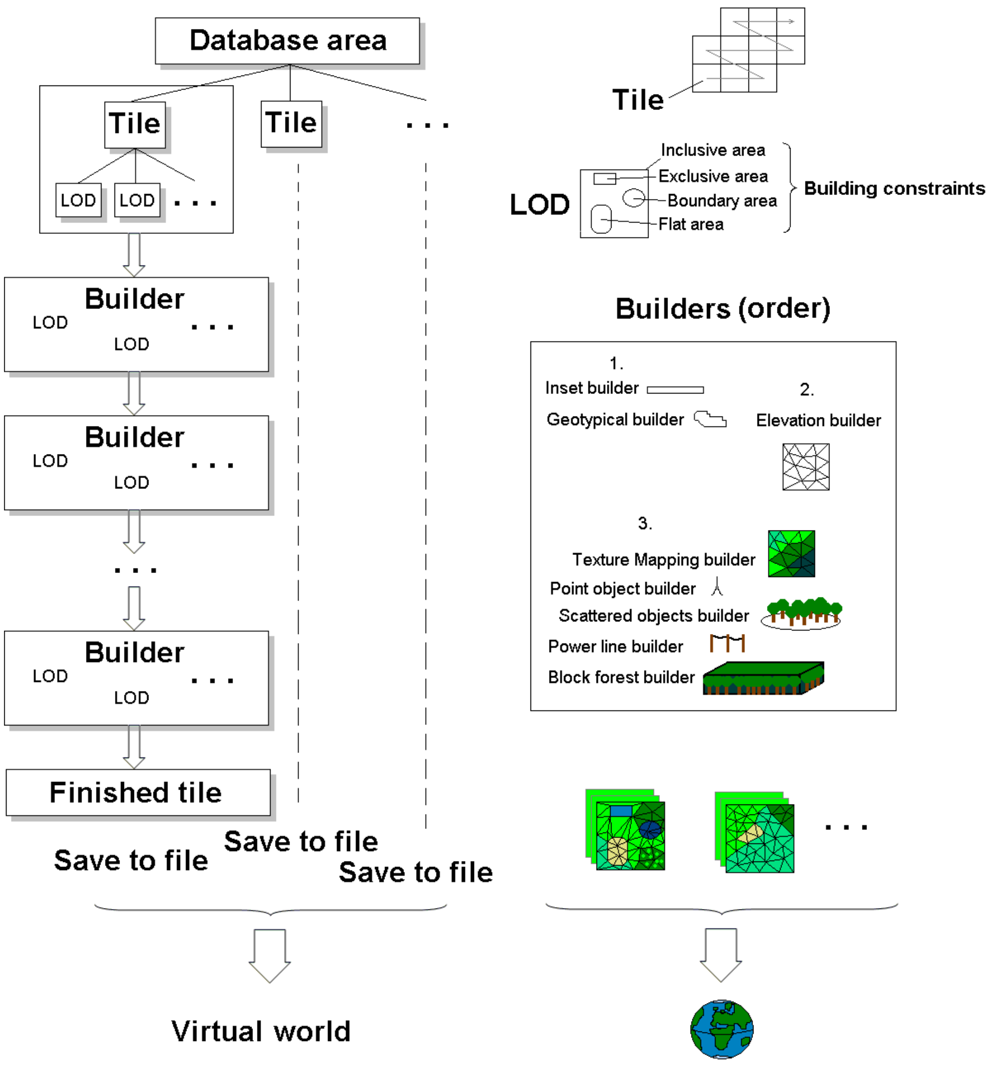


Figure 3-2: Overview of the DMGS.

An area in the database is chosen to be generated. The area is split into a number of equally sized quadratic *tiles*. The tiles are then generated from bottom to top and from left to right. The different tiles are generated separately, but their edges need to match to avoid cracks.

Each tile is generated into a number of different LODs. Typical numbers of levels are three or four. Every LOD has its own *building constraints*. The *inclusive area* is the outline of the tile indicating that the area inside should be triangulated. Inside the other building constraints special types of terrain are to be built, like seas, fields or no terrain is to be created. More about the different building constraints can be read later in this chapter.

The tiles are constructed one at the time. The construction is made in different *builders*. The builders specialize on creating a special part of the world, like triangulating the terrain or placing a special type of object onto the terrain. *Inset builder* and *Geotypical builder* are activated first because they affect the triangulation of the terrain. Thereafter the terrain is created by the *Elevation builder*. The order of the rest of the builders is not important. They all place different types of object onto the terrain. The builders operate on one LOD at the time.

When the tile has gone through all the builders it is saved to file in OpenFlight format and the process creates the next tile. A number of the OpenFlight files are later loaded into the runtime system to form the virtual world.

3.1.2 Scene graph structure

The LODs are organized in a staircase structure. The stair nodes are also LOD nodes. With this staircase structure there are only two different areas in each level. LOD 3 is placed at the first level since it is most likely to be shown. The staircase structure is shown in figure 3-3.

To prevent all tiles from switching at the same time the switching distances are not multiples of each other. The distances are shortened a little to spread the calculations over the time. An example of LOD distances are shown in figure 3-4.

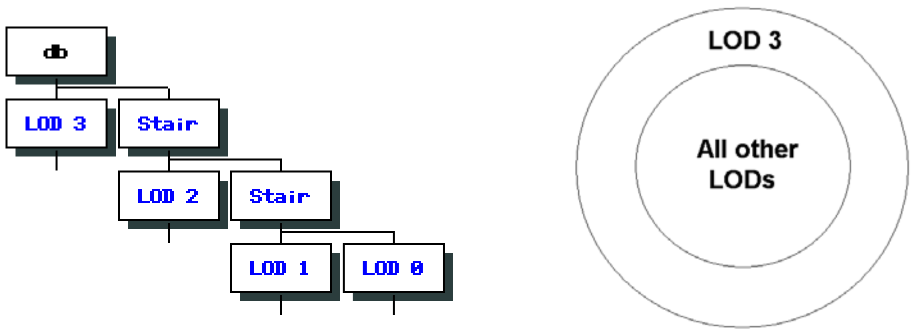


Figure 3-3: Staircase structure (a) Staircase scene graph (b) top step in the staircase structure

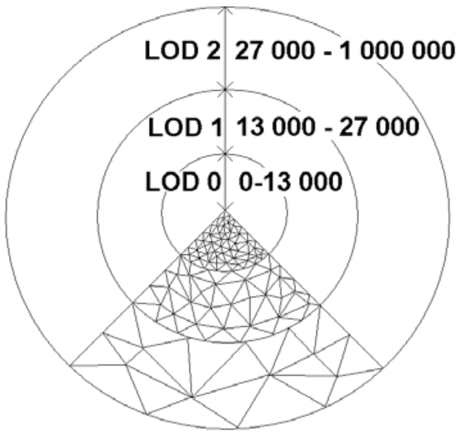


Figure 3-4: LOD distances

The terrain is split into a quad tree to speed up the search for the parts of the terrain to be shown in the runtime system. In each level the terrain is split into four parts. A quad tree split is shown in figure 3-5.

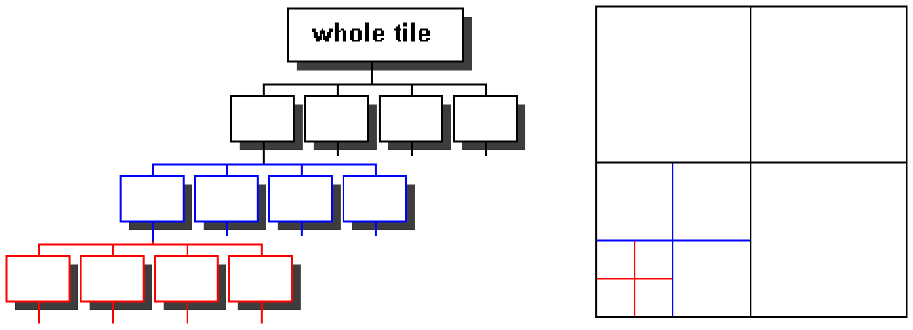


Figure 3-5: Quad-tree split

3.1.3 Building constraints

Building constraints control the different types of areas to take into consideration when constructing the terrain. Different types of areas give different conditions for the terrain generated in the area. Depending on what type of area a part of the terrain is it should, for example, be triangulated differently or a special type of object is to be placed there.

The different types of building constraints are:

3.1.3.1 Inclusive area

An inclusive area is the area in which the triangles are to be placed. For a tile this means the area of the whole tile. No triangles can be placed outside the inclusive area by the triangulator. Every tile needs to have exactly one inclusive area. The inclusive area for a tile is the four corner points and the lines connecting them. If the tile is to be connected to other already generated tiles, the points from those tiles edges are also included in the inclusive area.

3.1.3.2 Exclusive area

An exclusive area is an area where no triangles can be placed. Exclusive areas are placed in inclusive areas to create holes in the triangulation where an inset, a modeled object, can be placed.

3.1.3.3 Boundary areas

Boundaries can be forced to exist inside an inclusive area. Triangles can be placed both inside and outside a boundary area, but the edges have to be present. A boundary area can have a special texture. This is used, for example, to create fields.

3.1.3.4 Flat area

Flat areas have much in common with boundary areas. The edges have to be present but the area also has to be totally flat. A flat area can also have a special texture. This is used for example to create lakes, preventing them to slope.

3.1.3.5 Elevation raster

The elevation raster contains height values in a grid and is used for the triangulation of the terrain.

3.1.4 Preparing building constraints and triangulation

One tile is fully generated before the process proceeds to the next one. The whole tile is sent to the builder and the builder then handles the LODs one at a time. For all builders but the elevation builder the order of the LODs are of little importance. But for the elevation builder the order is vital.

For each LOD a border LOD is chosen. The border LOD tells the generation from which LOD to fetch the edge points. These points are found and put into the inclusive area of the LOD.

When a tile is created the most detailed LOD having itself as border LOD is triangulated first. If the adjacent tiles are already generated, edge points from the correct LOD of those tiles are included in the inclusive area of the LOD. This is shown in figure 3-6. The triangulation is handled by the elevation builder.

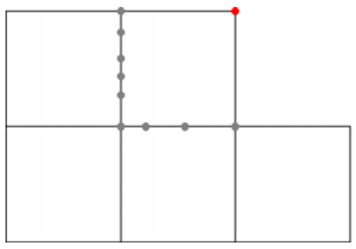


Figure 3-6: Edge points are fetched from previously generated tiles.

In the elevation builder the triangulation is made by sending the building constraints into a triangulator. The triangulator creates triangles with Delaunay triangulation taking the building constraints into account.

When the first LOD is done the other LODs are created with the edge points from the LOD chosen as their border LOD.

If all the LODs are set to have the same border LOD the same points will be present in all levels and no cracks will appear in the connection between the tiles. Most of the time the most detailed LOD is chosen as border LOD. This is the method usually used today.

If the push-down terrain method is used then all LODs are generated with no points from other LODs. This means that all LODs have their own border LODs. Instead of all edges matching in their points, the less detailed LODs are pushed down some distance into the ground. This makes the transition from lower to higher LOD visible, but since the terrain closest to the viewer always is the more detailed the cracks will not be visible. This method is not often used.

3.1.5 Builders

As mentioned in the overview section the builders construct different parts of the world. Inset builder and geotypical builder have to be activated first, then the elevation builder and, finally, the rest of the builders in any order. Inset builder and geotypical builder insert features into the terrain, elevation builder constructs the triangulation and the rest of the builders are specialized on placing different objects onto the terrain.

The builders have to be modified to be able to handle the solution developed in this project.

3.1.5.1 Inset builder

The Inset builder places pre-made models into the terrain. It handles the models that need to be incorporated into the terrain, for example an airbase model where the runway needs to be seamlessly put into the ground. The models that are placed onto the terrain are placed by other builders. The insets are put inside exclusive areas.

3.1.5.2 Geotypical builder

The Geotypical builder creates areas that can be seen as large areas of similar looking terrain, such as lakes and fields. The areas are built inside boundary areas or flat areas.

3.1.5.3 Elevation builder

The Elevation builder is the builder that creates the triangulation of the terrain from the height values in the elevation grid. It takes the building constraints as input and triangulates the terrain using a triangulator. The following settings from the user control the triangulation.

- Maximum number of polygons for the tile for the current LOD.
- Maximum wanted height error for the tile for the current LOD.
- Minimum distance from edges for a vertex in the current LOD.
- Maximum distance between vertices on edges in the current LOD.
- LOD from which to get edge vertices for the current LOD.
- Terrain pushed down or not.
- Distance to push down.

3.1.5.4 Texture Mapping builder

The texture on the terrain is constructed by the texture mapping builder. The builder goes through every LOD and puts textures on all the triangles belonging to the terrain that is not inside a boundary area.

3.1.5.5 Point Object builder

Models placed on top of the terrain are placed by the point object builder. The builder calculates the elevation at the position where the model is to be placed and sets the model's origin at that position. It also scales and rotates the models. To avoid cracks between the model and the terrain a foundation should manually be added to the bottom of the 3D model. Examples of point objects are houses, masts and bridges.

3.1.5.6 Scattered Objects builder

The scattered objects builder randomly places objects into an area. Attributes specify the density of the scattered objects and which models to use. The objects can be randomly positioned, rotated and scaled. The random process is controlled by attributes. Trees are a typical example of scattered objects.

3.1.5.7 Power Line builder

The power lines should follow a line through the landscape. The power line builder places modeled pylons on even spaces along that line. If a power line crosses the edge of the tile a pylon is placed on the edge if it is not present already in the adjacent tile. The pylons are connected by wires. Additional pylons are placed to prevent the wires from intersecting the ground.

3.1.5.8 Block Forest builder

The block forest builder calculates where to put a forest onto the terrain. It creates a wall of a specified height surrounding the forest area, following the terrain elevation, and creates the top of the forest block by using a triangulator. If there are terrain polygons hidden under the forest block those can be found and removed. Extra trees can be placed around the block forest to hide the sharp wall.

3.2 GRAPE, Graphics engine

The Grape (Graphics engine) software package, [4], is a runtime system for real-time simulations developed at SAAB Aerospace in Linköping. It can be used in different simulators with different types of display hardware.

Grape supports terrain databases in the OpenFlight format [9]; the format generated by the DMGS and used also by Multigen Creator [8]. It uses a paging technique that loads the parts of the terrain when they are needed, instead of loading them all at the same time. This procedure saves computer memory.

The scene graph structures implemented in the DMGS, described in 3.1.2, also help to optimize the performance of Grape at runtime. The quad tree structure helps to speed up the visualization by providing a quick way of deciding which features of the tile are necessary at a specific moment. It is unnecessary to make calculations on features outside the view field since they will not be visible anyway. The staircase structure of the LOD tree performs a similar task. It makes the search for the active LOD faster.

Which tile to load and which LOD to activate at a certain moment is based only on the distance between the viewer and the tile center. To avoid the popping effect when a new LOD is activated, fading is used in a transition phase.

4 Known methods and a quick evaluation

There is a big interest in terrain visualization. Most research handles continuous terrain generation in real time. This report presents a more specialized problem that divides the terrain into tiles and the problem becomes that of binding tiles with different resolution together without using too many polygons and preventing cracks.

4.1 Simple solutions

In some cases a simple solution can be the best solution. It depends on how tolerant the audience is.

4.1.1 Vertical polygons

Add vertical polygons to fill the cracks that appear [22], as shown in figure 4-1a. If tiles are used and no information is available from the nearby tiles a skirt of vertical polygons can be added to the edges of the tiles [23, 10, 12], see figure 4-1b. The skirt should be as long as needed to cover the openings in the cracks, for example from the highest point on the tile down to the lowest point on the near tiles.

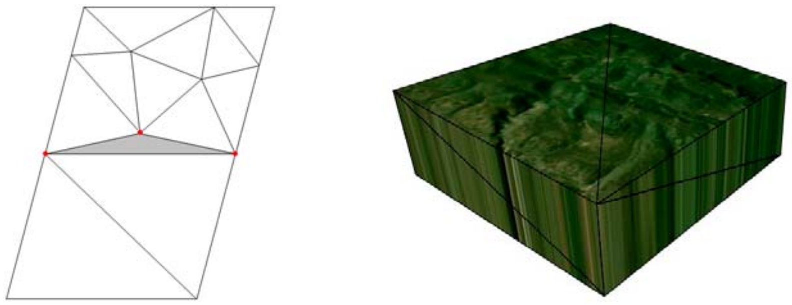


Figure 4-1: (a) Vertical polygons (b) Skirt of vertical polygons

This method does not take away the cracks but covers them with vertical polygons. The cracks will probably be very slim making the vertical polygons behave like vertical slivers and giving bad shading results [24]. Another problem is that textures are most of the time taken as areal photos, from straight above. This means that there are no textures for vertical polygons. How should the vertical polygons be textured? One way is to take the color of the edge of the tile and smear it down the vertical polygon. This may work with very slim polygons, but if more is shown of the vertical polygons they will look ‘striped’ [24].

This method also eliminates the possibility to have only two polygons at the least detailed level since the extra polygons are needed on the sides of the tile. When a tile is created it is not certain that the surrounding tiles are yet created. Therefore only the approach with a vertical skirt can be considered. For this it assumes that that each side of the tile has at least two polygons each to form the skirt. This gives a minimum of ten polygons for the least detailed level. At more detailed levels more polygons are needed for the skirt.

Since this can give visible artifacts and also does not make it possible to have only two polygons in the least detailed level this is not a solution to consider for SAAB.

4.1.2 Background color

Put a background color under the terrain or clear the screen with a color that makes the cracks less visible [24].

It is only possible to choose one color for the background so it is obviously not going to look correct for some parts of the world. If a green color to match the ground is chosen the green will be visible also when cracks appear in water. This can look very bad. Every color you choose will look terrible in some part of the terrain.

This is the simplest solution and may be good in some cases, but does not really solve the cracking problem. SAAB already uses this method already just in case some mistakes are made, but it is not seen as a solution.

4.1.3 Save old screen color

Do not clear the screen [24]. Instead keep the old color on positions where there is nothing new to draw.

This can work well in slow moving systems since the crack-filling color continuously adapts to the surrounding terrain. But combat airplanes make quick turns and rolls causing quick transitions between sky and land appear from frame to frame and that can obviously mean that the color has changed too much from one frame to the other to give the wrong color in the cracks.

Since this will look bad in fast motion and changes between land, sky and water this will not be a solution for SAAB since their aircraft move quickly most of the time.

4.1.4 Flat terrain with models of mountains

Make a flat terrain with few polygons and add models of the mountains [24], shown in figure 4-2.

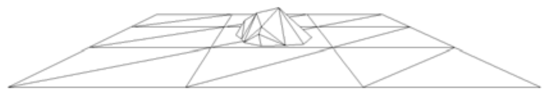


Figure 4-2: Flat terrain with modeled mountains

Natural environments are seldom totally flat. There is always some curvature to the landscape. It may look as if the ground is flat from high altitude, and in that case the ground may be treated as flat and the highest mountains can be modeled. But the models to add are seldom constrained to a small area and as you come closer to the ground the ground would look strange with a flat landscape. Very close to the ground, the whole surface must be modeled to capture the important curvature. It will simply be too much curvature to be handled as models.

This can maybe work in very flat environments, like old computer games, but natural terrain is seldom totally flat so this method is not an option for SAAB.

4.2 More complicated methods

The simple solutions above do not really take care of the cracks. The more complicated methods below do.

4.2.1 Dynamic terrain generation

Generate the terrain at runtime from a height grid [10, 25, 26, 24].

To do this you need to use a regular grid, polygon mesh, to be able to do the triangulation fast enough. The large area needed for a flight simulation takes huge amounts of power to calculate at runtime.

There are numerous methods for dynamic terrain generation and is a large research area. There may be a solution that will work for SAAB, but it will probably give more calculations at runtime and slow down the simulation. In any case this will cause the need for changes in their runtime system and therefore has low priority in this project.

4.2.2 Match edge points

Match the edge points to adjacent tiles and make sure that all LODs have the same edge points [3, 13, 21, 24], illustrated in figure 4-3.

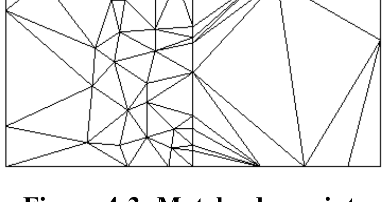


Figure 4-3: Match edge points

This method prevents cracks but there is a choice of adapting to the most detailed LOD or least detailed, or somewhere between. Each choice has its advantages and disadvantages. Adapting to the most detailed LOD gives unnecessarily large numbers of polygons in the least detailed LOD and the opposite gives rise to the problem with too sparse triangulation along the edges in the most detailed LOD making the edges show.

This is the main method used by SAAB at this moment.

4.2.3 Push-down the distant tiles

Force the edges of the further away tiles to be lower than the lowest point on the edge of the nearer tile [3, 24]. This is done by pushing the less detailed tile some distance into the ground. Since the tiles closer to the viewer are always at the same or higher detail as the more distant tiles, the cracks will not be visible, even though they are actually there. You still have to match the edge points for the same LODs.

This method does not take away the cracks but hides them. But if terrain is rough and there is a difference in LOD it can look strange. The method is illustrated in figure 4-4.

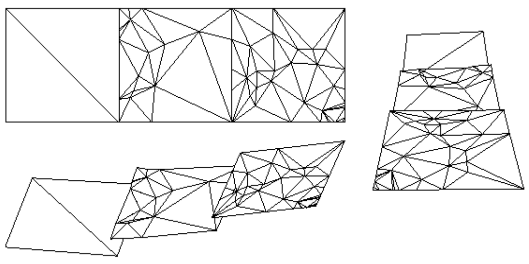


Figure 4-4: Push-down into the ground method

Top left view is a top view. No cracks in the terrain are visible from above, but notice the T-vertices. Bottom left view illustrates how the tiles are pushed down into the ground. The viewer is here situated at the bottom right corner. The right view shows how the cracks are not visible since the lower tiles are further away from the viewer.

This method is already used by SAAB as their second choice.

4.2.4 Pregenerated transition tiles

Use transition tiles with higher resolution on one side and lower on the other to form a bridge between different LODs [24], illustrated in figure 4-5. (The shapes of the transition tiles are unclear.) The transition tiles can be generated off-line and switched between in runtime, depending on the LODs of the tiles on the different sides.

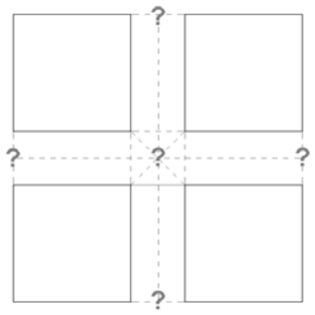


Figure 4-5: Transition tiles

The method eliminates the cracks and takes away the need to match edge points between different LODs, which in practice for SAAB means that it reduces the polygon count.

This actually solves both the criteria for the method that SAAB is looking for and is, therefore, worth exploring.

4.2.5 Real-time stitching

The real-time stitching was one of the first to come to mind when this project started. The method is used in [27]. The method leaves a space between the tiles and stitches the points together at runtime, forming the surface between the tiles, illustrated in figure 4-6.

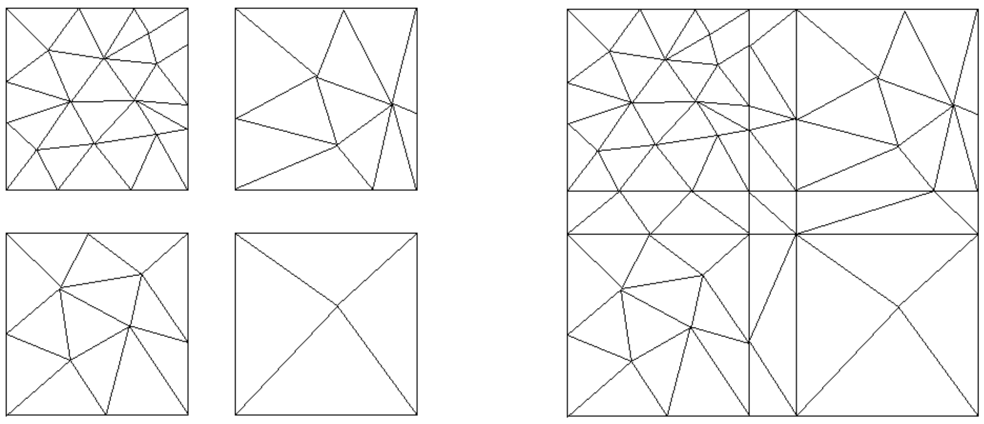


Figure 4-6: Real-time stitching

As the name implies, this method requires work to be done in the runtime system. Since the sought method must work offline this method has low priority in this project.

4.2.6 Adapting edge points

Move the higher resolution edge points in z-direction to lie on the surface of the lower resolution LOD [24]. This method requires that all the points (their x,y-positions) in the lower detail LOD be present in the higher detail LOD. When a higher resolution LOD is activated its edge points lie on the surface of the lower resolution LOD and are moved to their original position when the next tile activates its more detailed LOD. This method is illustrated in figure 4-7a.

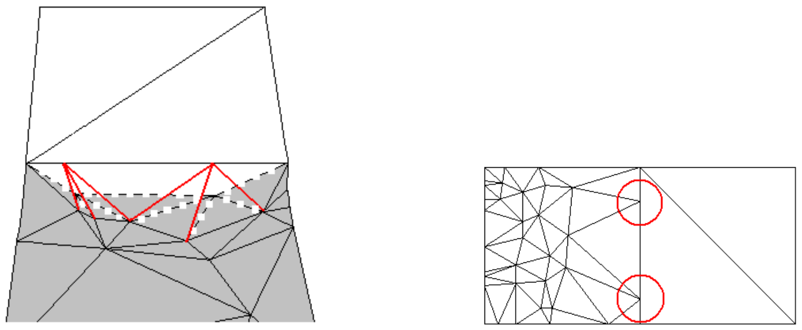


Figure 4-7: Adapting edge points (a) Points moving in runtime (b) T-vertices

This method must be done in runtime as a form of morphing and will still give rise to T-junctions. It can also give rise to cracks if, for example, the rounding of numbers is slightly off. This could be explored if there is a way to do it outside runtime, but it will still give T-vertices and that is not preferable, shown in figure 4-7b) Therefore this is not explored further in this report.

4.3 Conclusion

All methods in section 4.1 were found not to be worth exploring further either because they did not solve the problem, gave more polygons than necessary or were not suitable for realistic world generation. The methods in section 4.2 were more promising. The method of matching edge points and the one that pushes down the terrain are already used by SAAB and are therefore not necessary to examine. The dynamic terrain generation, real-time stitching and the adapting edge point methods were interesting but would require changes to the runtime system and, therefore, have low priority. The most interesting method was method 4.2.4, with transition tiles. This method was the first choice for further examination.

5 Developed solutions

The method shown in section 4.2.4, with transition tiles, was explored further since it was found the most suitable to solve the problem for SAAB. Since the solution described below was found the need for examining any other method requiring changes in the runtime system was unnecessary.

The solutions in this chapter are the ones created within this project. They all build on method 4.2.4, but they all use borders inside of the tiles instead of special tiles between. This is explained further in solution one. The solutions are not all different solutions as much as they are as they are improvements of the same idea and show the process of developing the final solution used for implementation.

Solution one was the initial solution and an improvement was found in solution two to keep the total nodes in the scene graph down. The third solution is an attempt to minimize the triangle count down in runtime. The fourth and last solution works on the same problem as the third solution.

5.1 Solution 1

The idea brought up in this project was to use special transition tiles between the regular tiles. This idea was solved by using borders inside the tiles instead and a method was developed from that idea.

5.1.1 Method

To create the transition tiles the tile is divided into center and four borders, illustrated in figure 5-1.

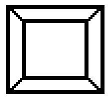


Figure 5-1: Divide the tile into center and four borders.

The center is to be the same while a certain LOD is active. The center will be of one version for the LOD and the borders will be of a number of different versions, adapting to different LODs of the adjacent tiles. Which version of the border that is active at a certain moment depends on which LOD the adjacent tile is in.

The adapting borders are shown in figure 5-2. Darkest gray represents the most detailed LOD and the white represents the less detailed LOD. The gray scale in between represent the adapting borders.

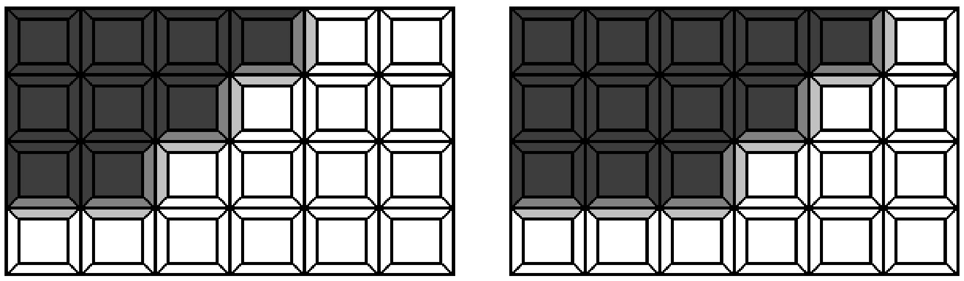


Figure 5-2: Use borders that are different depending on the LOD of the adjacent tile.

The adapting borders will match the edge points in the center of the LOD on the inside and the edge points of the LOD it is adapting to on the outside, preventing cracks in the terrain. See figure 5-3. The corner points of the whole tile are always present and, therefore, not from any particular LOD. It is important that the corner points are at the same positions in 3D in all LODs, or else cracks may appear when a border is adapting to another LOD.

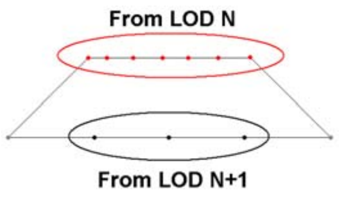


Figure 5-3: Use edge points from LOD N on the inside and points from LOD N+1 of the adjacent tile to form the adapting border.

To create the scene graph for handling the borders the process starts by placing a LOD-node for the whole geometry. Its center is in the middle of the tile. Directly under this node the center of the tile is placed. The resulting tree is shown in figure 5-4a.

Since the borders belong to one of the tiles it is necessary that the borders of one LOD are switched in and out at the same time as the center of the tile, otherwise the borders will not match the new center. This is solved by placing the border nodes beneath the main LOD-node. Since the borders must be in more than one version their geometry can not be placed directly under the main LOD-node since both the geometries would be visible all the time the LOD is active. Instead the different borders and their versions are put under their own LOD-nodes, with the LOD centers of the adjacent tile.

In this way the borders will change when the adjacent tile is changing. The LOD ranges of the LOD for the border that matches between two tiles of the same LOD is set to be the same as the ranges for the center of the tile. This step is shown in figure 5-4b.

New copies of the borders that match other LODs of the adjacent tiles are made and put under LOD nodes with the LOD ranges same as the LODs they adapt to. This step is shown in figure 5-4c. If the LOD would have been LOD 1 instead of 0, one more LOD that adapts to a more detailed LOD would have been necessary, giving three versions of the border for each border.

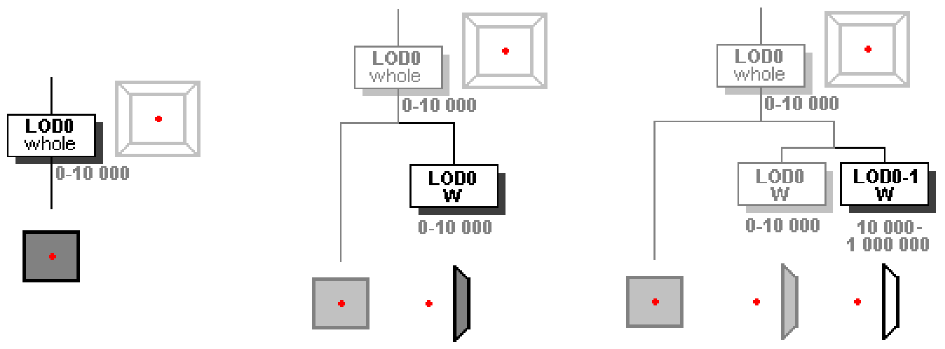


Figure 5-4: Creation of the scene graph step by step. (a) Add whole tile LOD (b) Add border under a LOD node (c) Add adapting border under its own LOD node

The whole tree will consist of a number of different LODs for the whole tile, all with a center and four borders. Each border consists of a non-adapting border and a number of adapting borders, show in figure 5-5.

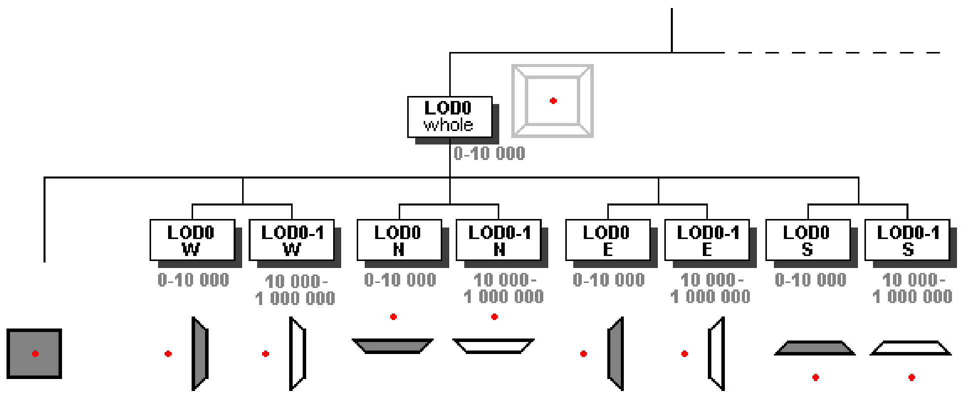


Figure 5-5: The whole scene graph for one LOD of a tile

This will give as result that the borders will follow the center of the tile but also change when the adjacent tiles are changing.

The tiles need to be matched in the generation even with this solution, but now only to the same LOD. This means that the least detailed LOD can have less detailed edges and the most detailed can have more detailed edges.

5.1.2 Evaluation

If the difference between two adjacent tiles is forced to be no more than one level, this method gives one LOD node for the center and two different versions of the borders for the least and most detailed levels and three for the levels in between. For a three LOD solution this would give 9(=1+4*2) different LOD nodes for the most detailed level, 13(=1+4*3) LOD nodes for the next level, and 9(=1+4*2) LOD nodes for the least detailed level. This gives a total of 31(=9+13+9) different LOD nodes. If this is divided for the tree LOD nodes this would give approximately 10(=31/3) LOD nodes for each LOD. The number increases for solutions with more levels since a larger portion of the LODs would need 13 LOD nodes.

The method will prevent cracks in the terrain but will give many copies of the terrain resulting in a growth in geometry. It will also reduce the number of polygons in the least detailed level compared to the method used today (where the most detailed level is used as border LOD) but will still give ten polygons, see figure 5-6, instead of the wanted two for the least detailed level.

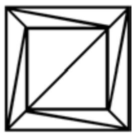


Figure 5-6: Borders give rise to a minimum count of triangles of ten instead of two.

The increased number of LOD nodes may slow down the application at runtime since LOD calculations is not that fast. This must be explored in tests.

The method was tested in Multigen Creator [8] on simple terrain, as above and the switching of the nodes worked well. A series of images from this test is shown in appendix A.

It would be possible to implement this method in the system. Still, a method that will not require the extra polygons caused by the borders (especially in the least detailed LOD) would be preferred.

5.2 Solution 2

Solution 1 was a possible solution but it still had some issues to resolve like the high polygon count in the least detailed level and the large number of LOD nodes desired. These problems are attacked in developing this solution.

5.2.1 Method

This solution improves solution 1 by only adapting to a lower level of detail, never to a higher. This gives that the least detailed level do not need borders at all, illustrated in figure 5-7.

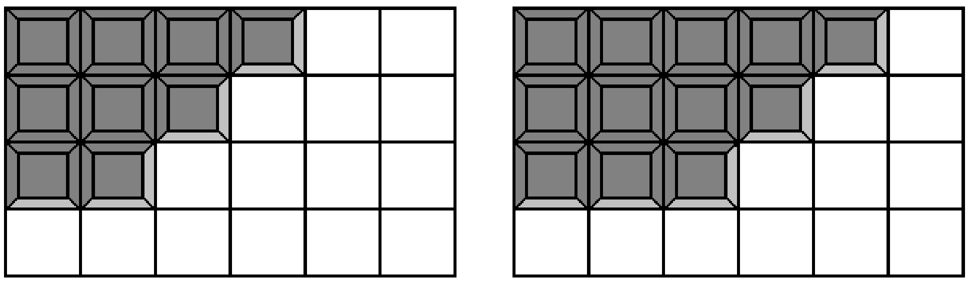


Figure 5-7: Adapt only to less detailed level of detail

The reason for adapting from higher to lower detail is that no borders are needed in the least detailed LOD, giving the possibility to only have two polygons there, meeting the goal of this project.

Since no border exist that adapts to a more detailed LOD, the border with the same LOD as the tile will need to be there even when the border meets a more detailed LOD. Otherwise a hole would appear in the terrain where the border should be. Therefore those borders need a near range that is the same as the near range of the one step more detailed LOD. This is illustrated in figure 5-8.

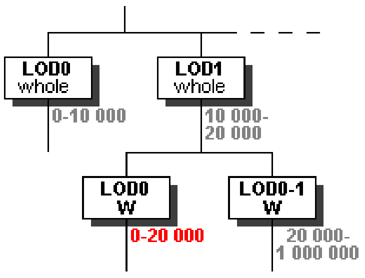


Figure 5-8: Keep border active even when border meets a more detailed LOD.

5.2.2 Evaluation

This method makes it possible to have only two triangles at the least detailed level. The borders in the other levels will not make that much difference since they contain more triangles anyway.

Since the borders only need to be in two versions instead of three the geometry and required number of LOD-nodes will decrease making the simulation faster than in solution one. The number of LOD-nodes for a three LOD solution will be one for the center and only two for each border in all LOD levels except the least detailed that only require the LOD for the center since all geometry is put there. This gives 9(1+4*2) for the most detailed level, 9(1+4*2) for the second level, and only 1 for the least detailed level. This sums up to a total of 19(=9+9+1) LOD nodes, which gives about 6(=19/3) LOD nodes per LOD compared to the ten nodes of the previous solution.

There still is the problem with the borders needing to be present, giving unnecessary triangles. If it is possible to create a method that does not need to show the borders when no adjacent tile is of another LOD, only if one or more of them are, it would be preferable.

5.3 Solution 3

In solution 2 the borders needs to be present at all times, meaning that the borders will cause more polygons in the higher detailed LODs than necessary. Even though it may not be of great influence on the whole, the question of improvement was raised. Is it possible to create a solution where the borders do not need to be present if all borders are at the same level as the center? The answer to this question is presented in this solution.

5.3.1 Method

Solution 3 is an improvement of solution 2, not showing the borders at all times. The tiles meeting tiles on all sides that are at the same LOD do not need any borders. The borders will only be needed on the tiles that meet a tile of another LOD on any of the sides. Figure 5-9 represents this idea.

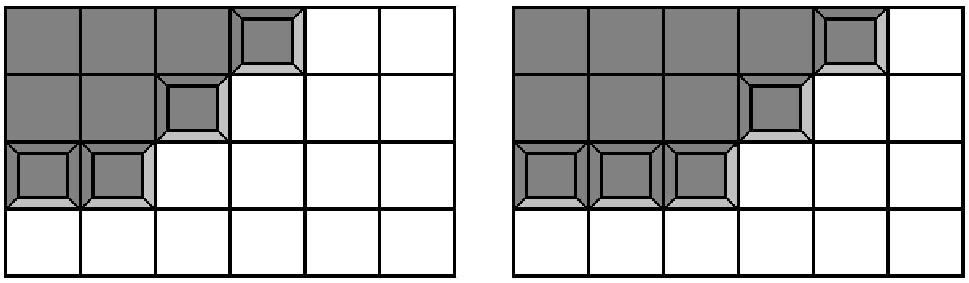


Figure 5-9: No borders are needed when no adjacent tile is at a different LOD.

All borders have to be shown if one border is needed, otherwise it would be necessary to have different centers for all possible combinations of borders as shown in figure 5-10.

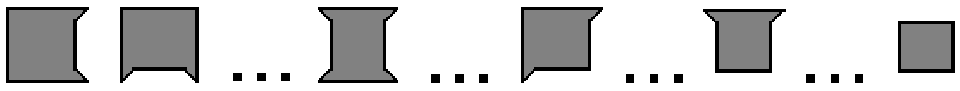


Figure 5-10: Centers at different combinations of borders, far too many.

A more complicated tree than in the previous solutions is needed to make this solution to work.

1. When all borders are at the same LOD as the center part of the tile no borders are needed. The large center can be shown, covering the whole tile.

2. If a border is at the same LOD as the tile, it must be checked if any of the other borders are at another level. If there is one or more then the borders and the small center are to be activated. If no border is at another level then neither the border nor the small center are to be shown.

3. If a border is at another level than the tile the adapting border and the small center are shown.

Point 1-3 must be done for all the borders.

The resulting tree is shown in figure 5-11.

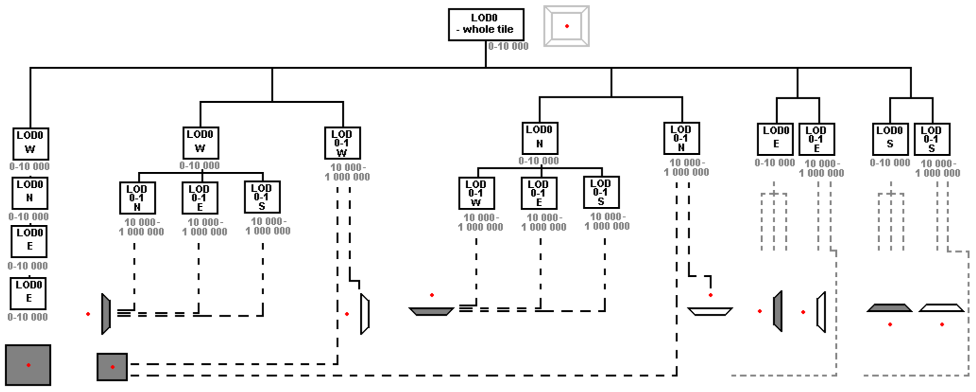


Figure 5-11: Scene graph for one LOD for the solution where borders only are needed in tiles with at least one adjacent tile at a different LOD.

5.3.2 Evaluation

This method will give rise to a very large tree with a lot of LOD nodes. Since instances of the geometry can be used the need for copying of the geometry is removed, but there will still be more geometry than in solution 2 since we have added the large center. Less triangles will be displayed at any one time, however.

How much extra geometry is needed is dependent on how the system handles instances. If the tile is separated and placed into a quad-tree the geometry may be duplicated even if instances are used.

This method will probably be slower because of the large number of LOD nodes. It is also more complicated to create the tree. The number of LOD nodes for a three level solution will be 21+21+1. A total of 43 LOD nodes, which means about 14 LOD nodes per LOD compared to the 6 of the previous solution. But the number of triangles shown at one time will be smaller than in solution 2 since we do not need any borders if they are not necessary where no side needs to adapt.

5.4 Solution 4 – final solution

Solution 3 was rejected due to it involving too many LOD nodes and too complicated a structure. The problem with the extra triangles in the more detailed LOD caused by the borders remained. Was there another way of solving this? A simple but effective way was developed in the final solution presented in this chapter.

5.4.1 Method

This method is almost exactly the same as solution 2 except that the borders do not need to have the exact shape of a trapezoid. Instead the whole tile is first triangulated, as usual, and then the borders are made by taking the triangles lying on each edge as the borders, illustrated in figure 5-12. The edge triangles are removed from the center and separated into the four borders. The adapting borders can then be constructed in the same way as described in solution 1.

This means that no extra triangles are needed because of the borders and still there is no need for the extra complexity of the tree in solution 3.

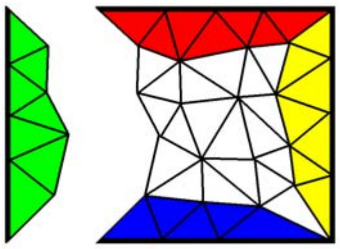


Figure 5-12: The borders are made from already present triangles.

5.4.2 Evaluation

This method is simple and makes it possible to have only two polygons at the least detailed LOD. Matching all LODs to one of the levels is no longer needed, only matching to the same level, and cracks are still avoided. The adapting borders build a bridge between the different levels of detail. No extra polygons are added to the solution by the borders since they are found from the already present geometry. The number of LOD nodes needed is minimized by only adapting to a one step less detailed LOD. The solution solves the problems of this project.

6 Implementation

The solution finally chosen for implementation was solution 4. The implementation was done by modifying the terrain generating module, OTW, in the DMGS, at SAAB in Linköping. DMGS is implemented in the programming language C# .net.

6.1 Pre implementation

This section presents some issues that had to be resolved before the implementation could start and how they were solved.

6.1.1 Choosing border polygons

How do we choose which triangles belong to which border? How many polygons should be part of the border? How are they found?

The simplest solution for choosing which polygons belong to the border is to take all polygons that have a point on the particular edge. But polygons can have a point on more than one edge and, therefore, polygons that suppose to lie on another edge may be included. To solve this problem it is not enough that the polygon has a point on the edge; it has to have at least one point on the edge that is not one of the corner points. Or in the case when only one polygon is at the edge, the polygon that contains both corner points will make up the whole border. How to choose the border polygons is illustrated in figure 6-1a and b.

The downside with this solution is that it does not prevent slivers. If the more detailed tile has a lot of polygons the borders become very slim. If then the one step less detailed tile has few points on the borders it will give rise to slivers in the border, see figure 6-1c.

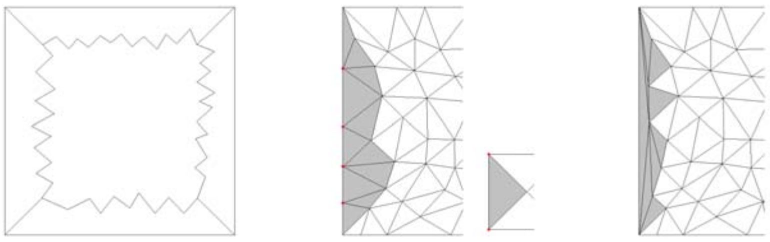


Figure 6-1: How to choose border polygons (a) Which polygons should make up the border? (b) – All polygons with a point that is not a corner point on the edge, or contains both corner points. (c) Slivers can occur in the adapting borders.

The problem with slivers in the borders is ignored in the first approach, but as the work went on the problem was found not to be that big. If the minimum distance between a vertex and an edge setting and the maximum distance between edge vertices is used, slivers can be prevented in the same way as they were in the original code.

6.1.2 How to divide corners

To be able to form the borders according to the method presented above we need to guarantee that the triangles along the edges can not touch more than one edge, illustrated in figure 6-2. Otherwise there will be problems when a tile changes LOD but the border follows another tile and does not change. This means that the edge points of the border may not match the tile that had changed. See figure 6-3 for illustration.

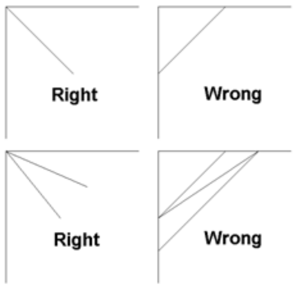


Figure 6-2: No polygons around the edge are allowed to touch more than one edge.

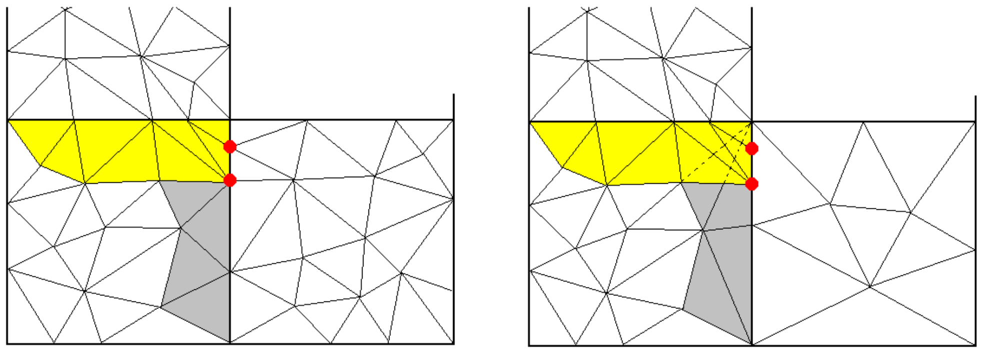


Figure 6-3: Problems occur when border polygons touch more than one edge. (a) No problem when all tiles are at the same LOD. (b) Problems occur when right tile has changed LOD. Nearest border has followed but the other border follows the top tile that has not changed LOD and has therefore not changed either. This means that points are present in the border that are not present in the changed tile. There will also be double geometry in the problem area since the adapting border has new terrain there.

Preventing polygons from touching more than one edge can be done during the triangulation process or handled afterwards. The simplest solution in the SAAB system is to use the building constraints.

By forcing a cross to be present in every LOD, except the least detailed, the problem with triangles touching more than one border will never occur. This is done by placing four triangular boundary areas in each LOD, as shown in figure 6-4.

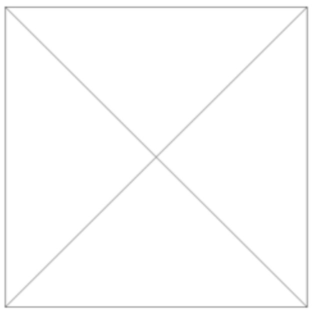


Figure 6-4: Insert four boundary triangles to prevent polygons from touching more than one edge.

Forcing the boundaries to be present will give extra triangles along the cross and is therefore not an optimal solution, but it was seen as a simple work around of the problem. The gain in fewer triangles in the less detailed LODs was also seen to be enough so that the extra triangles along the boundaries could be disregarded within the scope of this project.

6.1.3 Prevent more than one LOD difference between two tiles

To keep the number of LOD nodes down we need to prevent more than one level difference in LOD between two tiles. It is possible to implement a solution that could adapt to more than one LOD difference, but this would give more LOD nodes since every tile has to have a larger number of adapting borders. More LOD nodes mean more calculations in the runtime system.

If it is presumed that a tile only needs to adapt to a one step less detailed tile, but the difference in LOD is larger, there will be no border handling this and there will be a hole in the ground where the border should be. This must be prevented.

The solution to this problem depends on the difference in distance between the tile centers. The biggest difference in distance is between two tiles on the diagonal. But since we have a 3D world it also depends on the difference in elevation between the two tiles. The maximum elevation difference, z_{max}, is a value that has to be changed depending on the roughness of the terrain. How to calculate the minimum distance between near range and far range is shown in figure 6-5.

$$d_{\min} = \sqrt{tilediagonal^2 + z_{\max}^2}$$
$$= \sqrt{\left(\sqrt{tilesize^2 + tilesize^2}\right)^2 + z_{\max}^2} = \sqrt{2 \cdot tilesize^2 + z_{\max}^2}$$

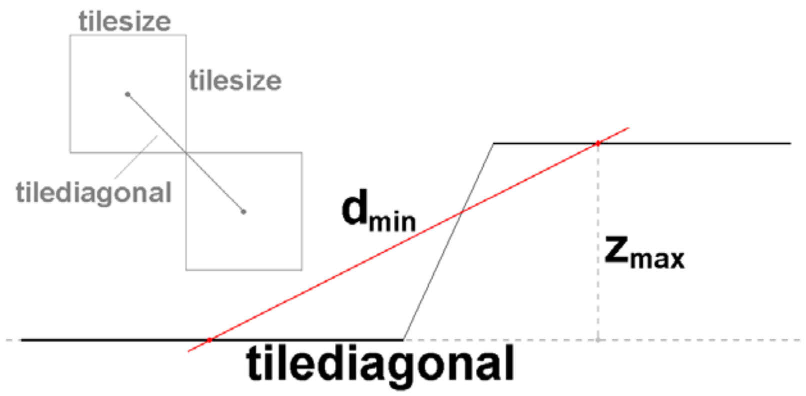


Figure 6-5: How to calculate minimum distance between near range and far range to prevent more than one level difference between two adjacent tiles.

The settings of LOD ranges and other are set in a GUI by the user before the generation is started. The choice of handling the problem above was to incorporate it into the GIU. When a user wants to save new settings, the differences between near and far ranges are checked. If any LOD has too narrow settings (less distance in between near and far ranges than d_{min}) a message is displayed, informing the user and asking if the settings should be saved anyway. Since the maximum elevation distance between two points is different for every database and difficult to calculate the user has to write it in by hand. The default value is 2200, chosen to assure no holes to appear in Sweden (The highest mountain is Kebnekaise with 2111 meters above sea level)[1] The value chosen is unnecessarily large since the elevation happens gradually and there are no huge steps from the highest mountain to the sea level, but it makes it impossible to get holes in the terrain due to more than one LOD difference.

6.1.4 Keep corner points the same for all LODs

The corner points are the only points on the edge that also belongs to another border and must therefore be the same in all LODs, not to cause any cracks. There will be no problem when there is the same data in all LODs, as there usually is, since the points in all LODs will get the same value from the raster. The problem occurs when different data is used in the different LODs. Different data may give different height values and cracks can appear if the corner points get different heights.

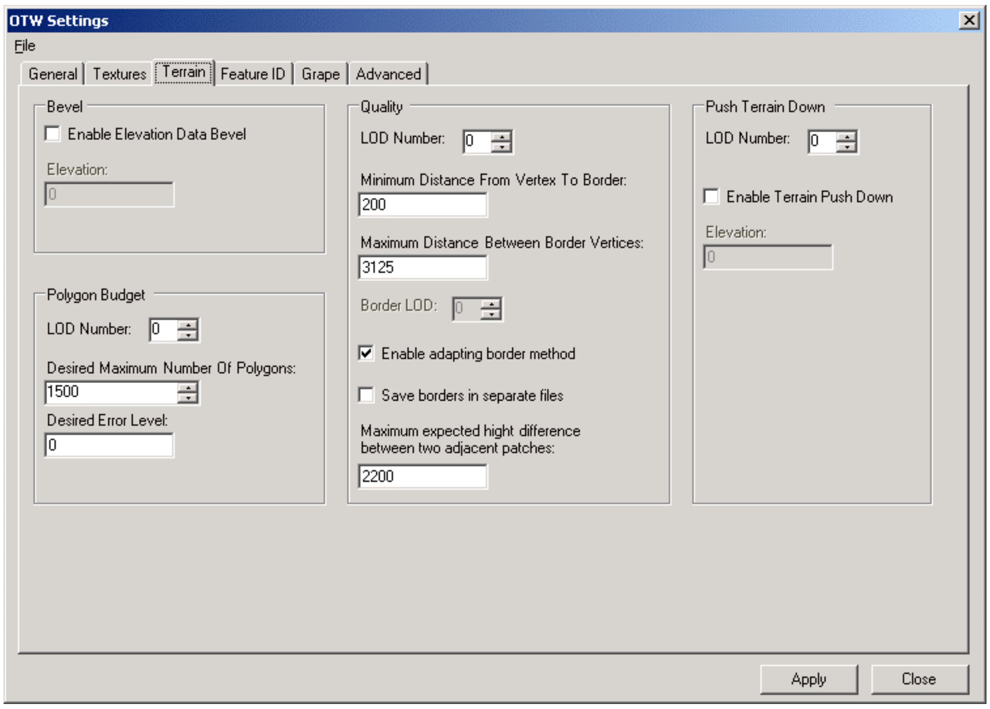
Since the same raster is used for all LODs most of the time by Saab, the problem was not seen as a big one. Also since it is only a programming issue and does not affect the theory of the solution this was left for future work.

There should be a number of ways of solving this. For example the corner points could be 3D in the inclusive areas or the no-data-raster may be created in a way that it contains the correct values at the corner points.
´

[1] Lantmäteriet, www.lantmateriet.se (2004-10-22)

6.2 GUI settings

Settings were added to give the user a choice of using the border method or not. If the border method is chosen all LODs get their selves as border LODs. The choice of saving borders in separate files becomes active for the user to choose. A text field also becomes visible to allow the user to change the minimum distance between near and far range to prevent more holes in the terrain due to more than one LOD difference between two adjacent patches, see section 6.1.3. The settings added to the settings GUI are shown in figure 6-6.



Bevel
[ ] Enable Elevation Data Bevel
Elevation: 0

Polygon Budget
LOD Number: 0
Desired Maximum Number Of Polygons: 1500
Desired Error Level: 0

Quality
LOD Number: 0
Minimum Distance From Vertex To Border: 200
Maximum Distance Between Border Vertices: 3125
Border LOD: 0
[✓] Enable adapting border method
[ ] Save borders in separate files
Maximum expected hight difference between two adjacent patches: 2200

Push Terrain Down
LOD Number: 0
[ ] Enable Terrain Push Down
Elevation: 0

Figure 6-6: Added settings to the quality parameters of the terrain generation in the DMGS.

6.3 Detaching, triangulating and organizing the borders

The creation of the borders is done after the elevation builder is finished. Elevation builder creates terrain covering the whole tile. This terrain becomes the center of the tile when the borders are removed in the way explained later in this section.

To handle the borders a new class was inserted into the OTW project. It takes the whole tile as input and goes through it one LOD a time, modifying the borders.

For each LOD all polygons belonging to a specific border are found, put under a LOD node and removed from the center. The detachment of the border is shown in figure 6-7.

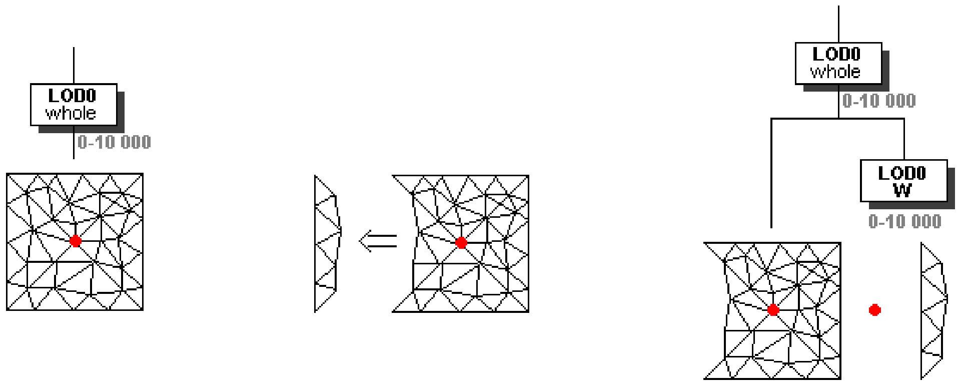


Figure 6-7: Detach the border and put it under its own LOD node under the tile LOD. (a) Work on one LOD of the whole tile (b) Find the border polygons and detach them (c) Put the border polygons under its own LOD with center in the adjacent tile.

The near and far ranges of the border LOD node are the same as for the main LOD and the center the same as the center for the adjacent tile. Since the tile adjacent to the border may not have been created yet, the center of the adjacent tile has to be calculated. The x- and y-values are calculated by adding or subtracting the tile size from the center of the present tile. The z-value is measured from the terrain raster data at the position (x,y). It is important that these calculations are done in the same way as it is done for the tile when it is created, to get the correct values. It is also important to freeze the LOD centers for all LODs. Otherwise the centers will be calculated in the runtime visualization system.

The polygons are found by traversing the polygons in the tile LOD. For every polygon all points are traversed until all points are searched or the polygon is added to the border. For each point the distances to the edge line and to the corner points are checked. If the distance is smaller than some small number it is assumed to lie on the edge. The three different cases below can occur:

1. If the point is found to lie on the edge, but not found to be one of the two corner points, its polygon is added to the border. If a polygon is added, its remaining points do not need to be examined further. If the polygon belongs to an exclusive area it is not removed from the center and not put into the border.

2. If a corner point is found, the polygon is further examined to check if it also contains the other corner point. If that is the case it is added to the border. If the polygon belongs to an exclusive area it is not removed from the center and not put into the border.

3. If the point is not found to lie on the edge, the test proceeds with the remaining points in the polygon.

At the same time as the polygons are found a union of all the polygons on the border is created. This gives a polygon containing all the exterior points of the border, shown in figure 6-8a. Polygons inside exclusive areas are added to the union. A union of the polygons of the one level less detailed LOD is also created, shown in figure 6-8b.

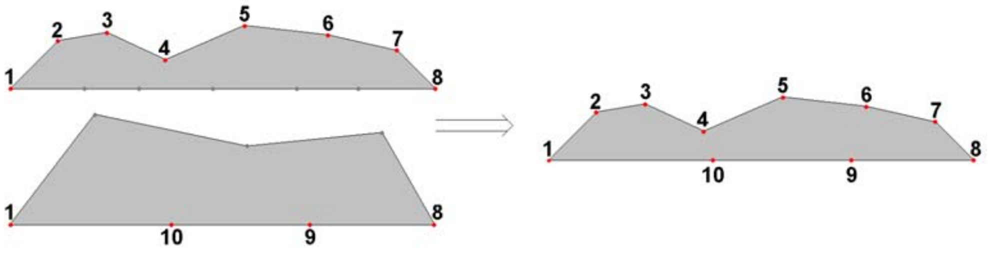


Figure 6-8: Inclusive areas. (a) (top left**) Inclusive polygon for LOD N (b)** (bottom left) **Inclusive polygon for LOD N+1 (c)** (right) **Inclusive area for the adapting border.**

The two union polygons are then used to find the inclusive area to use when triangulating the new adapting border. This is done by first traversing all points in the more detailed union polygon to find the first corner point (point 1 in figure 6-8a). When that point is found it is added to the inclusive area polygon (figure 6-8c) and the more detailed union polygon points are traversed and added until the second corner point is found and added (points 2-7 until point 8 in figure 6-8a).

Thereafter the union polygon of the less detailed union polygon is traversed to find the second corner point (point 8 in figure 6-8b). That point is already added from the more detailed union polygon, and does not need to be added. The less detailed union polygon points are continued to be traversed and added until the first corner point is found (points 9-10 to point 1 in figure 6-8b). The corner point does not need to be added since it is also already added.

The inclusive area created is put into a triangulator. To keep the exclusive, boundary and flat areas the same as for the original borders, the building constraints of the whole tile (for the correct LOD) is also put into the triangulator (not the inclusive area since we want the new one). The triangulator creates a triangulation of the points in the adapting inclusive area, taking into consideration the building constraints. Since no elevation raster is put into the triangulator, no extra points are added due to terrain curvature.

From the triangulator the new polygons can be recovered and put under a separate LOD. The LOD distances should be the same as the distances of the LOD the border is adapting to and the center the same as the center of the tile adjacent to the border. The addition of the adapting border is shown in figure 6-9.

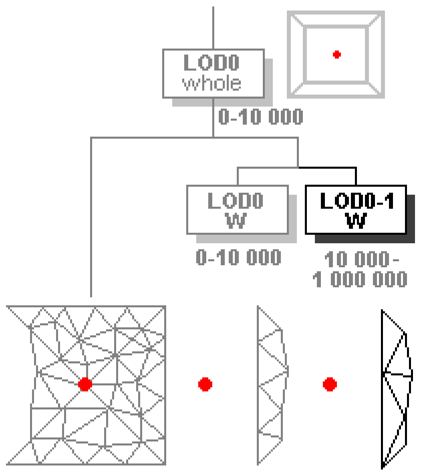


Figure 6-9: Adapting border triangulated and added to its own LOD node.

A choice of saving center and borders in one OpenFlight file or separate files was implemented. This choice came up for performance reasons. With the extra geometry added by the borders the file may become too large for the system to handle efficiently. To be able to do everything in the same way all the way up to the saving to OpenFlight files, a temporary tree structure was used during the whole building process. The temporary tree structure is shown in figure 6-10. The left part of the structure in figure 6-10 is the real structure and all the border groups are temporarily placed directly under the database header.

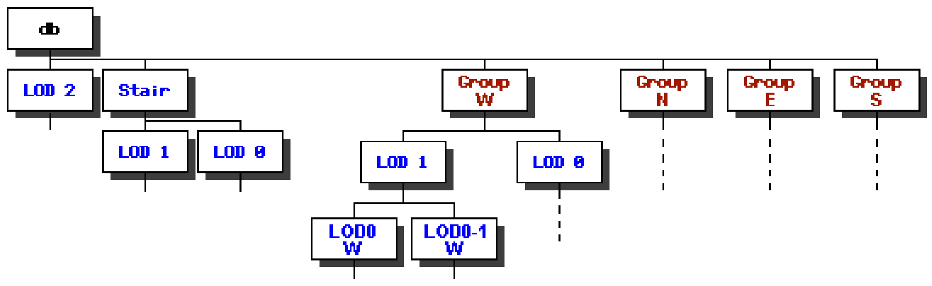


Figure 6-10: The temporary tree structure used during the building process.

The borders and adapting borders are returned from the border constructor in a list and put into the temporary tree structure.

After all builders are completed the borders are put in their final positions either in their correct positions in the tree and saved in the same file as the rest of the tile, or under a header node and saved in a separate file.

6.4 Modifying the builders

The builders had to be modified to be able to handle borders. They originally assumed that the whole tile LOD geometry was placed under a single LOD node in one version and filled the whole quadratic area of the tile. To be able to handle borders it must be able to work on the borders as well and not assume that the whole quadratic tile is filled with polygons under the same node.

All builders take a number of arguments. These arguments are which tile, the tile building constraints, a list of the LODs in the tile and a copy of the list with no features, called the nude list. The list is called nude because it only contains the naked ground polygons with no features on them. It is used to be able to do fast searching for polygons only on the terrain, even after features are added to the tile. Every nude triangle contains a reference to its counterpart in the real terrain list so that a polygon can be found in the nude list and the operation can be performed on the real terrain.

To make the modifications to the builders necessary for the improvement in this project a list containing all borders and one for its nude counterpart was added to the input arguments. The list is a list of headers, one for each LOD except the least detailed, which points to the borders. The list structure is shown in figure 6-11.

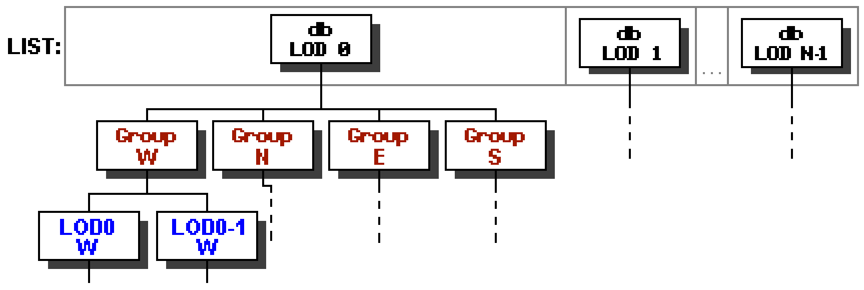


Figure 6-11: List of borders

To be able to adapt the power line builder the inclusive areas for the adapting borders were also added, in an array list, to the input arguments. The first position is a list that contains all the inclusive areas for the west adapting borders from LOD 0 and increasing, next all for the north borders and so forth.

The elevation builder did not require any changes since it is executed before the borders are created and is not used for building the new adapting borders.

6.4.1 Inset builder and Geotypical builder

Insets are models put into exclusive areas and geotypical terrain is constructed inside flat areas or boundary areas. Both builders are activated before the borders are created and therefore no modifications were needed in those builders. But insets and geotypical terrain still needs to be present in the borders. This means that the exclusive areas, flat areas and boundary areas need to be considered also in the triangulation of the adapting borders. This was solved by sending the building constraints of the whole tile into the triangulator, with the inclusive area switched to the inclusive area of the adapting border, see figure 6-12.

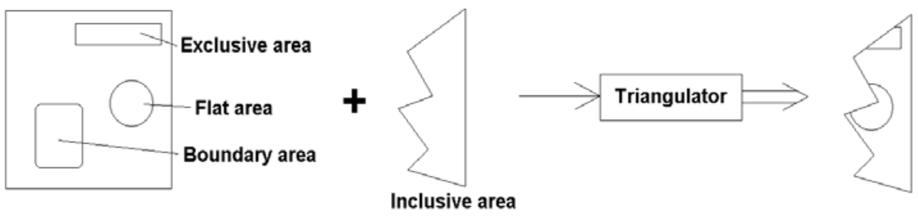


Figure 6-12: Triangulate the adapting border with building constraints

In this way the areas are inserted into the border. But no model is put into the exclusive area since that is handled by the inset builder. To be able to do this without having to run the inset builder again, the model for the original border is kept in the center part of the tile and not removed from the border. Since the model is a part of the center of the tile it will not switch with the border, instead it will fill the hole in the terrain in both versions of the borders.

The snag with this approach is that when no inclusive areas are present in the less detailed LOD, the points in the exclusive area on the tile edge will be present but may not get the same z-value as the model points. This will give the result that the model will not fit seamlessly into the terrain and cracks may appear, see figure 6-13.

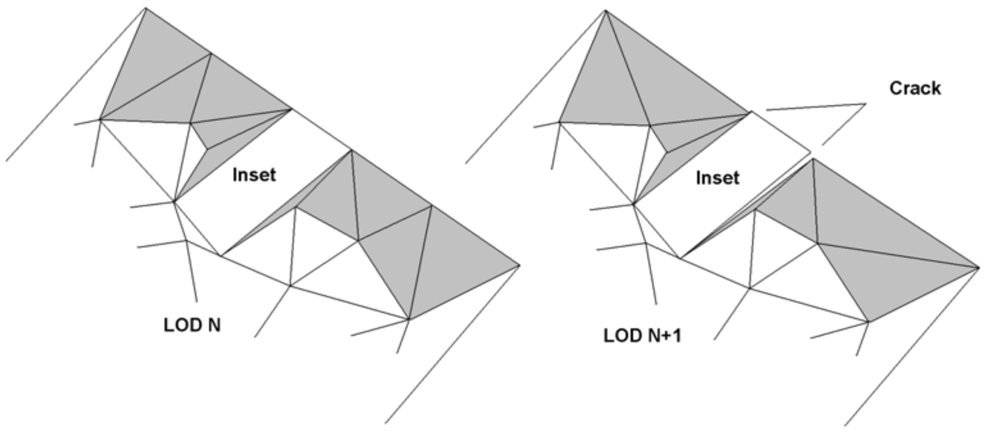


Figure 6-13: Cracks may appear between inset and adapting border terrain.

The crack problem is not impossible to solve. It can be done by forcing the missing points to be present in LOD N+1 if LOD N contains insets. Since this meant major changes to the preparation of inclusive areas it was skipped in the scope of this report.

6.4.2 Texture mapping builder

Not much was needed to be changed in the texture mapping builder to make it operate also on the borders. The builder initially worked on all terrain under the LOD nodes in the input argument terrain LOD list, going through all polygons and texturing them. The addition made to make it work even on the borders was to make it operate even on the LOD nodes of the borders in the border list.

6.4.3 Point object builder and Scattered objects builder

To make the point object builder and scattered objects builder work for borders is a similar problem. Both builders assume that all polygons are under the whole tile LOD and cover the entire tile area. Instead the builders need to place the object either on the center of the tile or in a border.

The objects on the borders need to be placed in two versions, one on the original border and one on the adapting border. Those two objects are the same object and therefore the preparation of the object can be done just once for the two objects, like finding rotation and scaling of the object, and then just placing the object twice. Both builders place the objects by taking the known x and y coordinates and finding a z-position on the terrain.

The main difference between the builders is that point object builder only places one object and the scattered object builder first distributes a number of objects over an area, represented by vector data and then places the object in a way that is very similar to the way the point object builder does. Since the preparation was to be done just once for each object position the scattering of the scattered object could be done once in the same way as it did when no borders were used.

Since the builders worked only on the whole tile LOD, assuming that all geometry was situated beneath and in one version, no terrain was found when the object should be placed on a border and the object was skipped. A new test was inserted to search through the original borders for terrain. If both borders were to be searched at once the algorithm would be confused since it assumed that it would only place one object but found two different z-values. Therefore the original border was searched first. If terrain was found in the original border there should also be terrain in the corresponding adapting border. Then the object can be placed once for each border. The placing of object on a border is illustrated in figure 6-14.

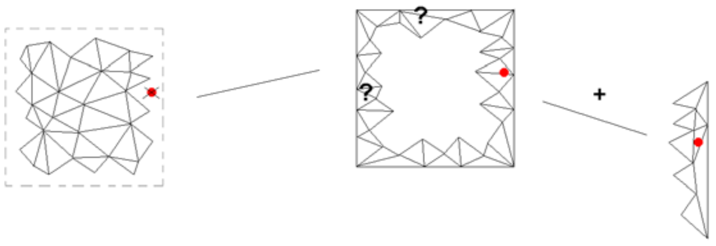


Figure 6-14: If no terrain to place the object on was found in the center of the tile, search the original borders for terrain. The west border is checked first, then the north, the east and at last the south. If terrain is found in an original border the object is placed on the adapting border as well.

Each time the object is placed it is cloned to its own structure that is placed under the correct node. This means that one copy is made for the original border and one for the adapting border if the object is placed on a border. For the features belonging to the center of the tile, the quad tree structure needs to be searched to put the feature under the right quad-tree group. This needed no modification since it only worked on the actual polygons under the LOD node. But an addition was needed for when an object is placed on a border. The borders are not divided into quad tree groups and the features are therefore put directly under the right LOD. The borders do not contain that many polygons and therefore the implementation of quad-trees in the borders was ignored in the scope of this report.

6.4.4 Power line builder

To construct a power line the pylons are placed along a line (line data), representing where the power line should be. At the same time a pylon is placed wires are constructed that connect to the previously placed pylon. This can cause a problem along the tile edges. The problem is illustrated in figure 6-15.

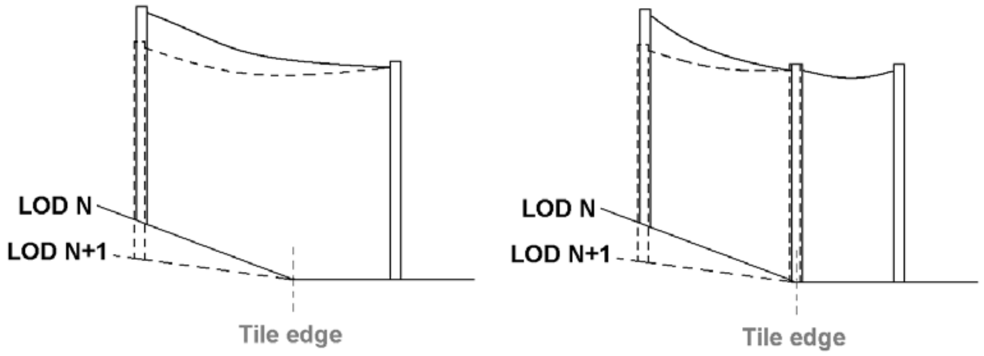


Figure 6-15: Problem with power line builder along tile edges (a) Problem occurs when a pylon is to adapt to a pylon on the other side of the tile edge since the other tile can have different LODs. (b) The solution is to place a pylon on the tile edge.

The pylon on the left tile has one version for each LOD. The pylon on the right tile needs to be in only one version, but connects with different wires depending on which LOD the left tile is in. Since a pylon can only bind wires in one version, the solution to this is to place a pylon on the edge between the tiles, one for each of the two LODs. The pylon on the edge will be of two versions, but they will be placed at the same elevation meaning that the wire from the right pylon will bind to the correct point on the pylon on the edge at all times.

The same problem as around tile edges occurs around the border edges, and it is solved in the same way. Pylons are placed where the power line crosses a border edge. Since the pylons exists two versions, the pylon on the border edge needs to be able to appear in two versions and is therefore placed on the border.

Since the border edges can have a very jagged structure, the worst case occurs when a power line is parallel to a tile edge and crosses the edge of the border a several times. This will give rise to a large number of pylons placed close to each other. This is shown in figure 6-16. It is obviously not a perfect solution, but sufficient in most cases. Most of the time the polygons defining the jagged edge of the border will be too large in comparison to the maximum distance wanted between two pylons, for the problem to occur.

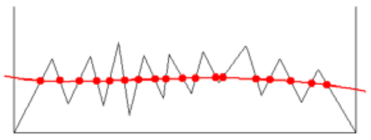


Figure 6-16: Power line crossing border edge multiple times can cause a great number of pylons close to each other.

When the positions where the pylons are to be placed are found it is time to place the pylons. This is done in a similar way as the point object builder and scattered objects builder. A test for finding the elevation on the terrain for placing the pylon is conducted on the list of LODs and the addition made was if no terrain intersection was found, to search through even the borders. If the terrain was found in one border, terrain will be found in the adapting border.

The pylons are planted one by one, binding its wires to the previous pylon. When a pylon is placed on an original border, one is placed at the adapting border too. Every time a pylon is placed it binds wires to the pylon placed before it. This means that the previous pylon has to be kept in memory. To make both the pylon on the original and on the adapting border to bind its wires to the correct heights on the pylons, a previous pylon for the adapting border as well as the one for the original border and center has to be kept in memory. Which one to bind to has to be tested for each pylon.

When a pylon is placed, a check is made on the x,y-position of the previous pylon (it is the same for the original and the adapting). If the previous pylon is found to lie inside the center of the tile or on the center edge the wires for both edges are to be connected to the previous pylon. If on the other hand the previous pylon was found inside the border or on the tile edge, the previous pylon should be used for the pylon on the original border and the adapting previous pylon should be used for placing the pylon on the adapting border. This is illustrated in figure 6-17.

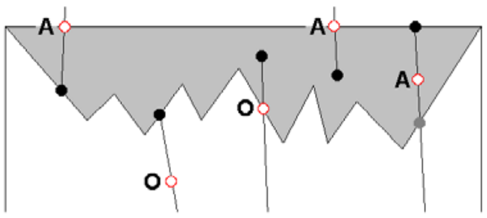


Figure 6-17: When a pylon is placed (●) anywhere inside the border or on a border edge, the previously placed pylon (○) needs to be checked to find which version – adapting previous pylon (A) or original previous pylon (O) – the present pylon should bind its wires to.

This test is only needed when a pylon is to be placed on a border. If it is to be placed in the center, it only needs to be placed in one version and to connect to the previous pylon.

Problems can occur along the inside edge of the border since a pylon that is placed on the edge can find its terrain in the center. This is solved by checking if the pylon is on a border edge and in that case placing the pylon on that position, but putting it in two different versions under the LOD nodes of the border.

6.4.5 Block forest builder

Block forests are placed inside an area circumscribed by vector data. The area is cut along the border edges. If the user wants to place extra trees around the block to get a more realistic look, the area where the block forest is to be placed is shrunk down to give space for the extra trees inside the area specified by the vector data.

The block forest creation is illustrated in figure 6-18. The block forest is created by first creating the roof. It is created by triangulating the forest area to follow the terrain and then raised to a user specified height over the ground. Thereafter walls are created around the whole forest area, except on the tile edges. The roof polygons and the wall polygons are textured with different textures.

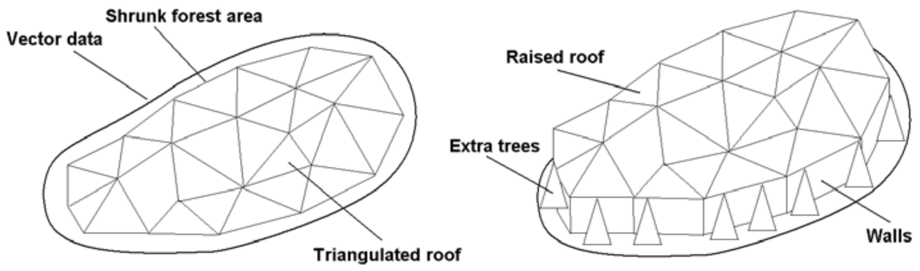


Figure 6-18: Creating a block forest. (a) The block forest roof is triangulated inside the area created by shrinking the vector data area. The roof follows the terrain. (b) The roof is raised and walls are created around the forest area. Extra trees can be placed around the block forest but inside the vector data.

To be able to create block forests when the border method is used the forest area has to be divided into different parts on the center and each border, shown in figure 6-19. The area is shrunk before the division. The split forest areas are then created separately. The forest parts on the borders are created in two versions, one that follows the original border terrain and one that follows the adapting border terrain.

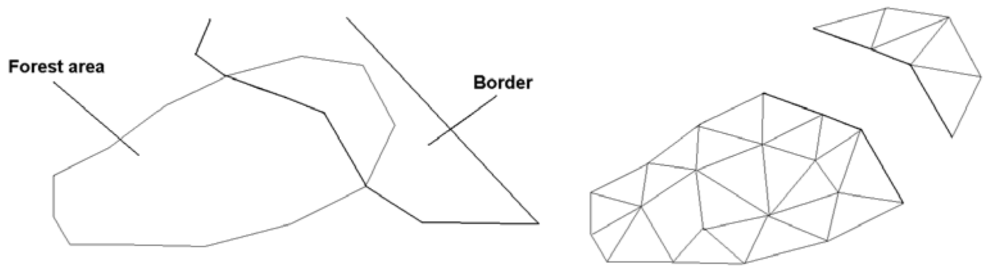


Figure 6-19: Split forest at border edges and create the parts separately.

When the walls are created walls are only created on the exterior of the original shrunk forest area, not between the parts.

The roof triangles are attached in the tree at the positions where the triangle in the terrain lying beneath is attached. In the original code the whole geometry was under one LOD and it was searched through to find the terrain polygon. To place the block forest triangles that are on the borders, the correct border LOD is searched instead of the whole tile LOD. The wall polygons are placed at the position as the touching roof polygon.

The extra trees around the forest are placed as in scattered objects builder.

7 Result

This section contains tests conducted on the resulting DMGS created in this project and a comparison with the system as it were before the modifications started.

7.1 Creation of test data

To test on as different terrain as possible the choice was to choose one flat and one rougher area. To fully test the advantage of the solution created in this project a large area had to be used as would be the case in a generation for production. The solution has its greatest advantage in a large area since most of the terrain in that case will be of the least detailed level.

Due to a server change, not much data was available that was no classified. Therefore an area of the north of Sweden was chosen. The area reaches from the north east cost, around Luleå, where the landscape is flat to the rougher area near the Norwegian border. Around Luleå all types of data is available like elevation data, satellite imagery, aerial photos, vector data for lakes, power lines etc and point data for other features. Satellite imagery is also available in an area near the Norwegian border. The rest of the area contains only elevation data. Since we want a large area this area is also used. The area covered by the data is shown in figure 7-1.

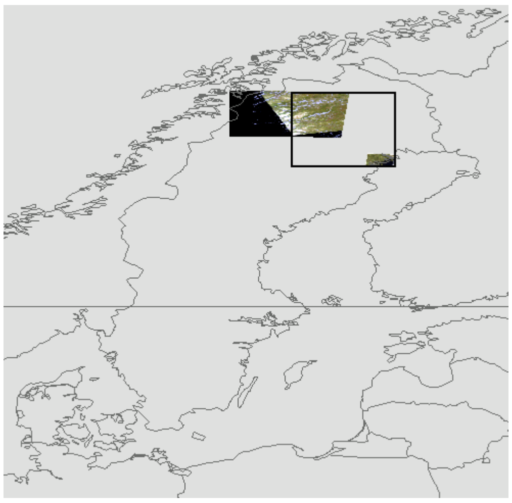


Figure 7-1: Data area in north of Sweden. The area generated is the area inside the black rectangle. Elevation data is present inside the whole area but images for texturing are only present in the two corners.

7.2 Test setup

To do comparison between the border method developed during this project and the original method, one generation with the border method and two generations with the original method, one with the most detailed LOD as border LOD and one with the middle detail, was conducted for every test. The least detailed LOD is never used as border LOD by Saab today and was therefore and for time reasons not used in any test.

7.2.1 Test 1

The whole area of 187.5 x 300 km^2, 15 x 24 = 3600 tiles, was generated with only elevation data and texture. This test may be used for both performance and visual appearance measures.

The tile size used was 12.5 x 12.5 km^2, a typically used setting. The maximum polygon count for the different LODs were 1500, 800 and 50 and the maximum allowed height error was set to 0, 2 and 10. The near and far ranges settings were 0-20 000, 20 000-45 000 and 45 000-100 000 000. The rest of the settings were kept at their default values. The near and far ranges were chosen to prevent more than one level difference between two adjacent tiles, se section 6.1.3 and 45 000 was chosen to not be twice as big as 20 000 to keep all LODs to switch at the same time as described in section 3.1.2. A summary of the settings is shown in table 1.

	LOD 0	**LOD 1**	**LOD 2**
Maximum number of polygons	1500	800	50
Minimum deviation error	0	2	10
Minimum distance between border vertices	2500	2500	2500
Maximum distance to border	200	200	200
Near range	0	20000	45000
Far range	20000	45000	100000000
Transition	500	500	500

Table 1: Settings for test 1

7.2.2 Test 2

Next test was designed to assure fewer polygons in the less detailed LODs. The least detailed polygon count was lowered to 10 and the maximum deviation set to 50 and for the next least detailed LOD the polygon count was lowered to 400 and the maximum deviation set to 10. The maximum difference between border vertices, used to prevent slivers, were set to 3125, 625 and 12500, meaning at leas three extra points added on each edge in LOD 0 and one in LOD 1. The rest was kept as it were in test 2. A summary of the settings is shown in table 2.

	LOD 0	LOD 1	LOD 2
Maximum number of polygons	1500	400	10
Minimum deviation error	0	10	50
Minimum distance between border vertices	3125	6250	12500
Maximum distance to border	200	200	200
Near range	0	20000	45000
Far range	20000	45000	100000000
Transition	500	500	500

Table 2: Settings for test 2

7.2.3 Test 3

This test is similar to the previous test. The result may be used in the same way.

All settings but the maximum polygon count was kept the same as in test 2. The maximum polygon counts were set to around one fourth of the counts of test 2, giving 375, 100 and 10. This was chosen to examine how the result would be affected when the border makes up a larger portion of the tile. A summary of the settings is shown in table 3.

	LOD 0	LOD 1	LOD 2
Maximum number of polygons	375	100	2
Minimum deviation error	0	10	50
Minimum distance between border vertices	3125	6250	12500
Maximum distance to border	200	200	200
Near range	0	20000	45000
Far range	20000	45000	100000000
Transition	500	500	500

Table 3: Settings Test 3

7.2.4 Test 4

The last test generates the area around Luleå with all possible data activated. This allows for all features of the builders to be generated. This is done to compare how the terrain created by the border method looks with features compared to the terrain generated by the original method. This can be used for performance tests but the main goal with this generation is to look at the visual result. It was also a test of checking which builders that worked in the new implementation. The settings were kept as they were in test 2. A summary of the settings is shown in table 4.

	LOD 0	LOD 1	LOD 2
Maximum number of polygons	1500	400	10
Minimum deviation error	0	10	50
Minimum distance between border vertices	3125	6250	12500
Maximum distance to border	200	200	200
Near range	0	20000	45000
Far range	20000	45000	100000000
Transition	500	500	500
Inset builder	X	-	-
Geotypical builder	X	X	X
Point object builder	X	-	-
Scattered objects builder	X	-	-
Power line builder	X	-	-
Block forest builder	-	-	-

Table 4: Settings Test 4

7.3 Computer resources

The hardware used for the terrain generation is a client running the DMGS application getting data from a server via the network.

The client is a standard Dell Precision Workstation 650 running windows 2000 with a single Intel Xeon 2.8 GHz processor, 1 GByte of memory and 73 GByte of SCSI disk.

The server is running ESRI ArcSDE and Oracle DB on top of SPARC Solaris on a Sun Fire V440 Server with four 1.28 GHz ultraSPARC IIIi processors, 8 GByte of memory and 3 TByte of fibre channel disk from redundant RAID level 5 controllers with 1 GByte of ECC cache memory.

The workstation running the simulation, with GRAPE 3.1, is a standard Dell Precision Workstation 650 running RedHat Enterprise Linux 3 with dual Intel Xeon 3.2 GHz processor, 4 GByte of memory, NVIDIA Quadro FX 3000 graphics, a color calibrated 22" CRT monitor, a 21" LCD TFT monitor and 600 GByte of fast SCSI disk on a RAID level 5 cache controller for scratch storage.

7.4 Performance

The results from the tests are presented in this section.

7.4.1 Generation times

The generation times are measured for the whole area generated and are presented in table 5. The table also presents the time it would take to generate one tile of size 12.5 x 12.5 km^2 based on the resulting time. The test setups are presented in section 7.2.

	Test 1	Test 2	Test 3	Test 4
Original method (border LOD 0)	3h 43m 30s = 37s/tile	3h 6m 14s = 31s/tile	51m 6 s = 9s/tile	48m 26s = 2m 50s
Original method (border LOD 1)	3h 7m 1s = 31s/tile	2h 8m 58s = 21s/tile	39m 46s = 7s/tile	–
Border method	4h 37m 49s = 46s/tile	3h 34m 28s = 36s/tile	1h 13m 34s = 12s/tile	1h 7m 55s = 4m 15s/tile
Increase in time with borders compared to:				
Original method (border LOD 0)	24%	15%	44%	49%
Original method (border LOD 1)	51%	66%	85%	–

Table 5: Generation times

The generation times increase for the border method compared to the original method. The degree of time increase depends a lot on the settings from the user. In the best case above the time only increased by 15% and in the worst 44% (for only terrain) compared to using the most detailed LOD as border LOD in the original method. Compared to the original method with LOD 1 as border LOD the time increase was from 50% and up to almost twice the original time. With features the increase in time was almost 50%. This may suggest that the extra work done by the builders affects the generation time more than the creation of the borders.

7.4.2 Polygon count

The different tests produce different number of polygons in each level of detail. The number of polygons produced depends on the settings chosen for the test but also on the method used. Table 6 shows the number of polygons produced by the different methods for the different LODs.

		Test 1	Test 2	Test 3	Test 4
LOD 0	**ORG_0**	1502	1502	358	35963
	ORG_1	1501	1500	375	–
	BORDER	1504	1507	385	35950
LOD 1	**ORG_0**	800	397	100	343
	ORG_1	803	395	102	–
	BORDER	810	404	106	368
LOD 2	**ORG_0**	74	73	31	51
	ORG_1	51	27	14	–
	BORDER	63	9	2	14
Overall gain with border method compared to:					
ORG_0		14% (in least detailed LOD, none in the others)	88% (in least detailed LOD, none in the others)	94% (in least detailed LOD, non in the others)	73% (in least detailed LOD, none in the others)
ORG_1		-25% (in least detailed LOD, none in the others)	67% (in least detailed LOD, none in the others)	85% (in least detailed LOD, non in the others)	–

Table 6: Polygon count (ORG_0 = Original method with border LOD 0, ORG_1 = Original method with border LOD 1, BORDER = Border method)

For all of the tests above, all points in the least detailed level were placed on the tile edges when the original method was used. When those points were placed the entire polygon budget was consumed. The border method allowed more points to actually get to use inside the tile. Still less polygons were needed in the least detailed LOD compared to the original method (in all tests but test 1 with border method 1). There was a slight increase in the other LODs. This may be due to the forced cross that prevents the triangles around the edge to touch more than one edge (see section 6.1.2). The gain in polygons can be optimized by trying different settings for the LODs.

The image in appendix B-2 shows the difference in polygon count between the border method and the original method.

7.4.3 Update rate

No differences in the update rate could be detected since all tests could run at maximum update rate. A slower computer may have given more difference, or a larger terrain. But there was not enough time to test this. More tests are needed in the future for testing this.

The main reason for looking at the update rate is to see how the extra LOD nodes produced by the border method affects the update rate. Maybe they slow it down and a great gain in triangles are needed to compensate for this. To give a fair test the same path must be flown in all versions of the terrain produced by a test.

7.5 Visual appearance

The visual tests were done in GRAPE 3.1 on the third computer presented in section 7.3 by Andreas Ladell. He is used to looking at simulator terrain. Images from the tests are shown in appendix B.

He could not see any holes in the terrain in the original nor the border method. He could not see any particular difference between the methods. Especially in the flat area around Luleå there was no visible difference. In the rougher area near the Norwegian border a difference could be seen when viewing test 2. In the original border the least detailed LOD became visible as a flat area with strange looking peaks around. An image of this is available in appendix B-1. In the border method slivers appeared between LOD 1 and 2. This can be helped by using the sliver preventing settings better, but it still looked better than the problem in the terrain created by the original method.

8 Conclusion and future work

The goals of this project were to find a method that did not mean any changes to the runtime system, Grape, and to improve the performance by allowing a smaller number of polygons to be shown further away from the user than today.

A method using transition borders between different LODs was developed allowing for less triangles in the less detailed LOD. The function was relatively easy to implement and worked well. It clearly shows that the method found solves the problem and is possible to implement.

The generation time was increased by 15-45% when only terrain was generated. The increase depended on the settings. The increase in generation time was larger when features were added, almost 50%. The increase in time was found acceptable since it is possible to enhance the performance at runtime instead and the generation time increase was found reasonable and not too large. The code is at this moment not optimized, and doing so the generation times should be reduced noticeably.

The possibility to have less polygons in the more distant tiles will increase runtime performance. The larger area visualized the greater the gain would be. In the long run this will mean that more features can be generated in the terrain without slowing the simulation down. Or more polygons can be used in the more detailed terrain LODs.

An even better result would have been able to be achieved by removing the need to have the cross forced by the building constraints in all but the least detailed LOD. Further work might solve this through a more sophisticated triangulation algorithm instead.

The polygon count did not decrease as much as one could think. The reason to this is the setting of maximum polygons. It tells the system to create a number of polygons. Most of the time it is that condition that stops the triangulation. Instead of giving fewer polygons, the points around the borders decreased making better use of the triangles inside the tile where they are needed. This gives a better looking result, correlating better with the real world. Therefore it is possible to reduce the number of wanted polygons in the less detailed LODs to get the polygon count down instead without any loss in appearance.

In the future it would be interesting to test whether the extra LOD nodes produced by the border method influences the update rate negatively or not and make sure that the whole gain in triangles was not all consumed.

The person doing the appearance test did not see that much difference between the original method and the border method. The biggest difference was the one mentioned in the section above: polygons were used where they were needed in the least detailed LOD giving no flat areas in the rougher area as it were with the original method. It would be nice to see terrain produced by the method of this project working in the simulator.

Further work with this project would mean to fix the builders that do not work properly. The builders that do not work properly are the inset builder that gives cracks around the tile edge and the block forest builder that gives incorrect results. The inset builder problem can be solved by forcing the missing points on the edge to be present in all LODs. The block forest builder was not finished due to lack of time, but the overall structure is finished.

Further work would also be to incorporate the changes made within this project in the more developed version of the system used at SAAB at this date. There has been a great deal of development since the version used here.

Only simple testing of the builders was conducted. The goal with this report was to see if the necessary changes were possible and this is clearly proven, even though the code is not properly tested. More extensive tests need to be done when the code is officially brought into the system. There is no point in testing the solutions when the code is out of date.

One minor issue to look into is the power lines that follow parallel to a border and crosses the edge between the border and center multiple times. This can happen, but most of the time the borders are not that rough and the case will not appear.

References

1. The Visualization Toolkit, 3rd edition, W Schroeder, K Martin, B Lorensen, Kitware, ISBN 1-930934-07-6, 2002

2. 3D Computer Graphics, 3rd edition, Alan Watt, Addison-Wesley, ISBN 0-201-39855-9, 2000

3. EBS DMGS OTW/IR/RR Export module Subsystem Design Description, SAAB Andreas Ekstrand, Reg. No. FTM-02.088 issue 4, 2004-06-08

4. Grape IG software description, SAAB Anders Wallerman, Reg. No. FTM-03.013 issue 3.0, 2004-05-16

5. Beskrivning av PMSIM, SAAB Andreas Ladell, Reg. No. FG-98.259, 2001-02-13 1

6. What is ArcGis?, ESRI, ArcGis 8.2, 2002 (http://www.esri.com/software/arcgis/concepts/overview.html, 2004-11-18)

7. The ESRI Guide to GIS Analysis, vol 1, ESRI Andy Mitchell, ISBN: 1-879102-06-4, 1999

8. Multigen Creator – The Desktop Tutor, version 2.3, Multigen-Paradigm Inc, 1999 (http://www.multigen.com/products/database/creator/index.shtml, 2004-11-22)

9. OpenFlight Scene Description Database Specification, Multigen-Paradigm, Issue 15.7.0, April 2000 (http://www.multigen-paradigm.com/support/dc_files/of_spec_15_7.pdf, 2004-12-02)

10. An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains, David Hill, University of Toronto Department of Computer Science, 2002 (http://www.magma.ca/~dhlf/downloads/thesis.pdf, 2004-10-26)

11. Landscape modeling: Digital Techniques for Landscape Visualization Stephen M. Ervin, Hope H. Hasbrouck, McGraw-Hill Professional Publishing (c) 2001, ISBN: 0-07-135745-9 (http://www.landscapemodeling.org, 2004-11-18)

12. A Hybrid, Hierarchical Data Structure for Real-Time Terrain Visualization, Konstantin Baumann, Jürgen Döllner, Klaus Hinrichs, Oliver Kersting, IEEE Computer Graphics International, 1999. Proceedings, June 1999

13. Real-time visualization of big 3D City models, Michael Beck, ViewTec AG Zurich, 2003

14. Computing Diriclet Tessellations, The computer Journal 24(2):162-166, 1981

15. Computing the n-Dimensional Delaunay Tessellations with Application to Voroni Polytopes, DF Watson, The Computer Journal 24(2):167-172, 1981

16. Sliver Exudation, Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunners, Michael A. Facello and Shang-Hua Teng, Journal of the ACM (JACM) Volume 47 Issue 5, 1999

17. Sliver Exudation, Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunners, Michael A. Facello and Shang-Hua Teng, ACM Proceedings of the fifteenth annual symposium on Computational geometry, 2000

18. Smoothing and Cleaning up Slivers, Herbert Edelsbrunner, Xiang-Yang Li, Gary Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör and Noel Walkington, Proceedings of the thirty-second annual ACM symposium on Theory of computing, 1999

19. Watertight tessellation using forward differencing, Henry Moreton, NVIDIA Corporation, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Aug 2001

20. Real-Time Rendering Of Realistic Landscape, Brendan Thomas Mackrill, University of Sheffield Department of Computer Science, 2003 (http://www.dcs.shef.ac.uk/teaching/eproj/ug2003/pdf/u0bm.pdf, 2004-10-26)

21. Smooth view dependent level-of-detail control and its application to terrain rendering, H Hope, IEEE Proceedings of Visualization 1998, Oct. 1998

22. Simplification of Objects Rendered by Polygonal Approximations , Michael J. DeHaemer, Jr. and Michael J. Zyda, Naval Postgraduate School Code CS, Department of Computer Science Montray, California, 1991 (http://citeseer.ist.psu.edu/cache/papers/cs/3476/http:zSzzSzwww.dcs.ed.ac.ukzSz~mxrzSzpszSzZyda.SimplifyingPolygons.pdf/dehaemer91simplification.pdf, 2004-11-03)

23. The Swing Connection - JCanyon - Grand Canyon Demo, Kenneth Russell, Sun Microsystems Inc, 2004 (http://java.sun.com/products/jfc/tsc/articles/jcanyon/, 2004-10-26)

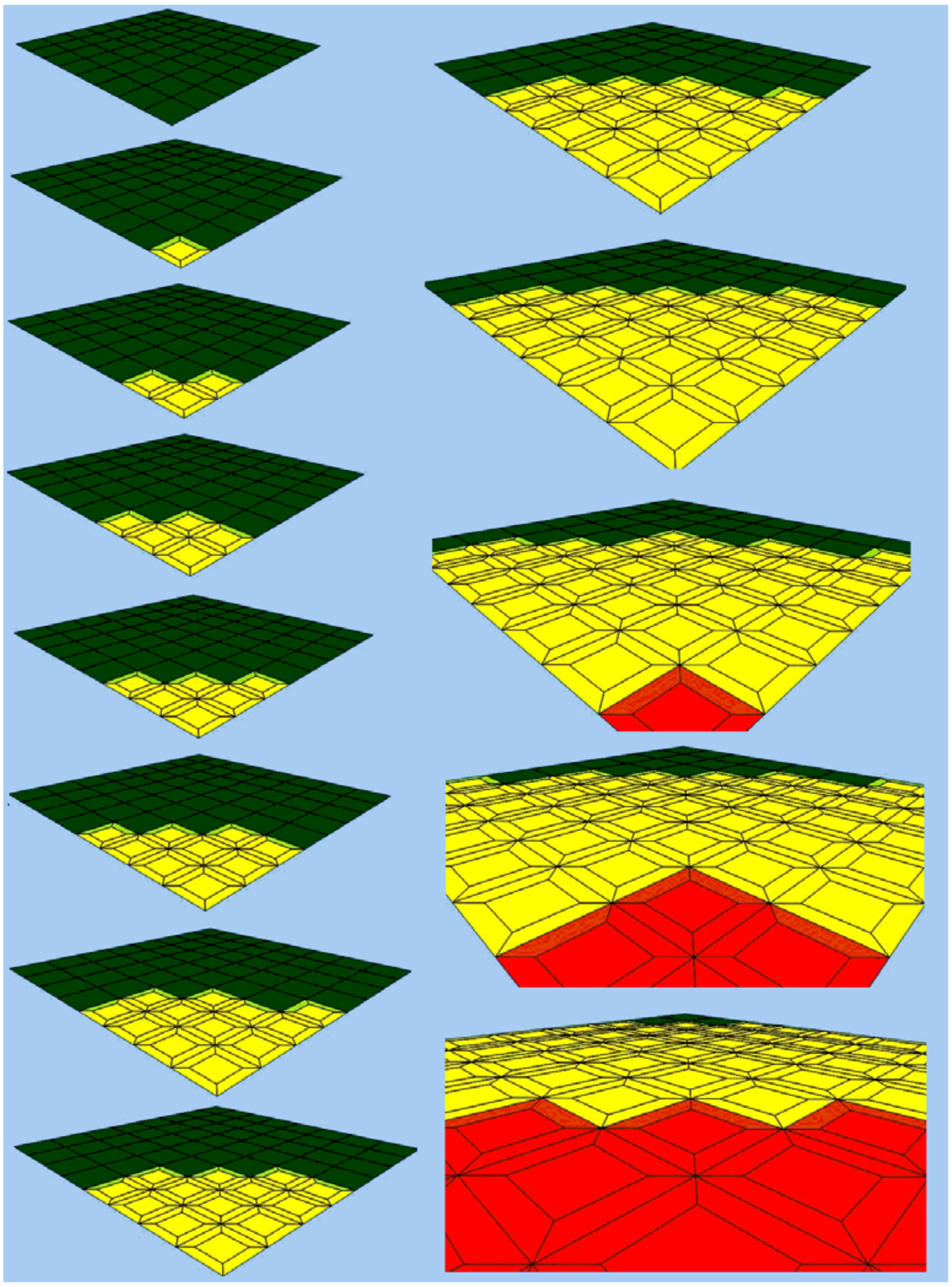
24. GeoVRML, Web3D Consortium 2000-02-02, http://www.geovrml.org/archive/msg00560.html (2004-10-26)

25. ROAMing terrain: real-time optimally adapting meshes, Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein, IEEE Proceedings of the 8th conference on Visualization 1997, oct 1997

26. Real-time Optimally Adapting Meshes, Christopher Hoult, ACM, 2004-05-10

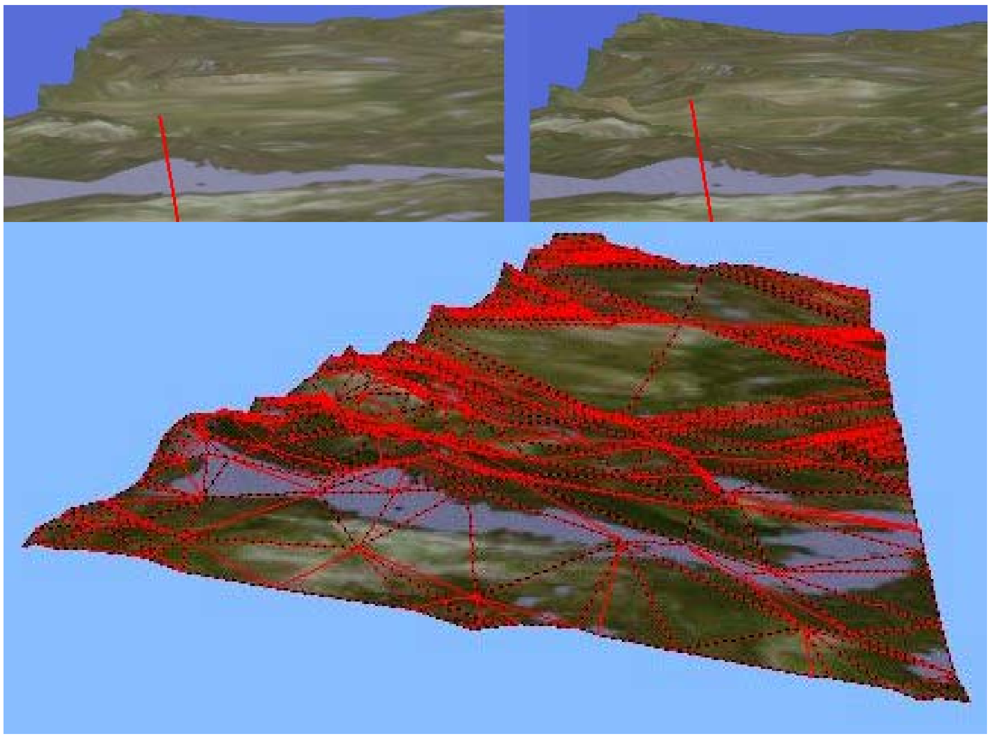
27. Decoding of Large Terrains Using a Hardware Rendering Pipeline, James E Fowler, John van der Zwaag, Shivaraj Tenginakai, Raghu Machiraju, Robert J Moorhead, Engeineering Research Center, Mississippi State University 2000, Tech Rep MSSU-COE-ERC-00-13

Appendix A – Solution working on simple terrain

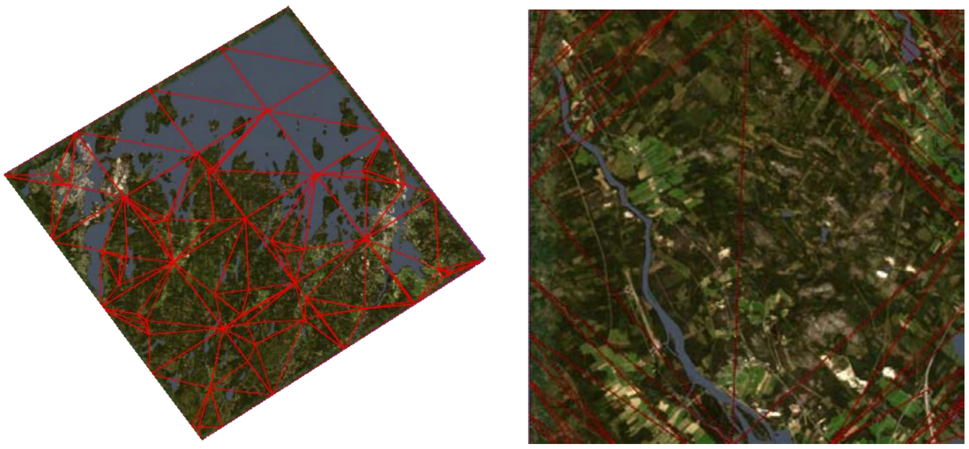


The image above shows how the method is working on simple terrain. Up in the left corner all tiles are at the least detailed level. As the viewer get closer more and more tiles switch to a more detailed LOD. The borders between two different levels of details adapt to the less detailed LOD.

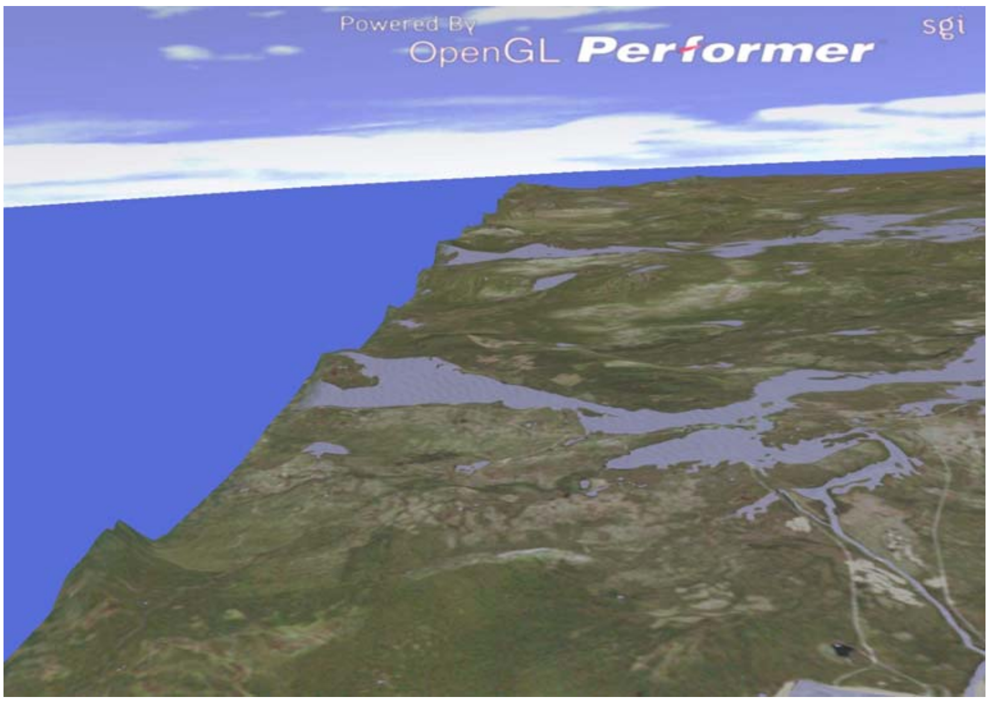
Appendix B – Result images



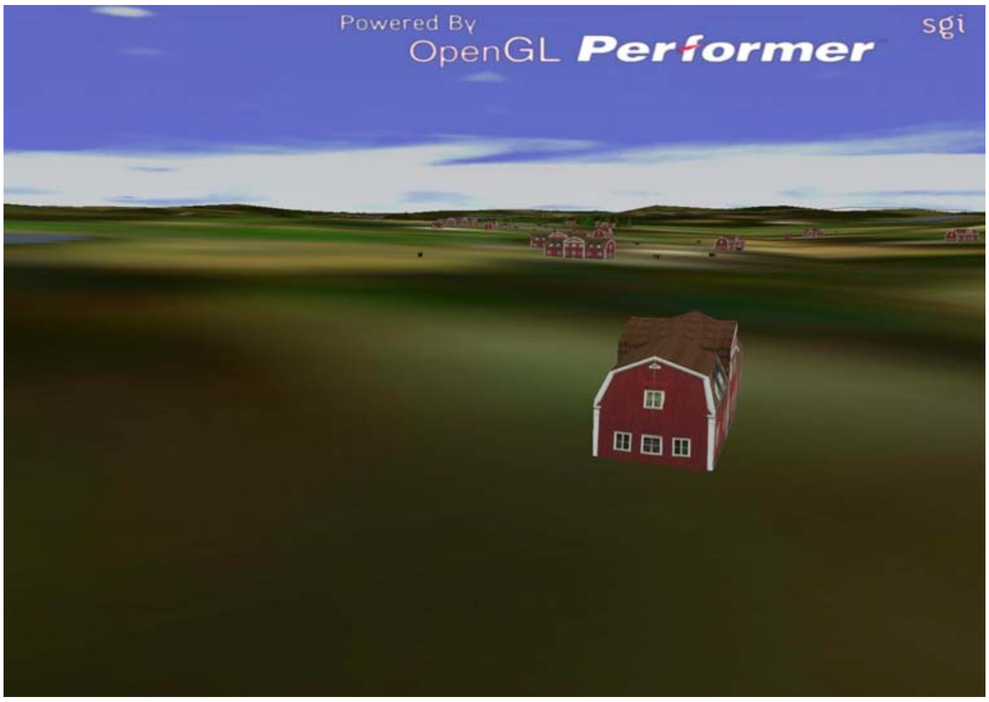
Appendix B-1: Artifacts appear with the original method when all points in the least detailed level are forced to be on the tile edges in a rough area. Flat areas appear with steep mountains around the center. In the top left image the flat area is visible and a “wall” is showing beyond it. In the bottom image the wires help to more clearly show where the flat areas are. The top right image shows that the area is not flat when the viewer comes closer.

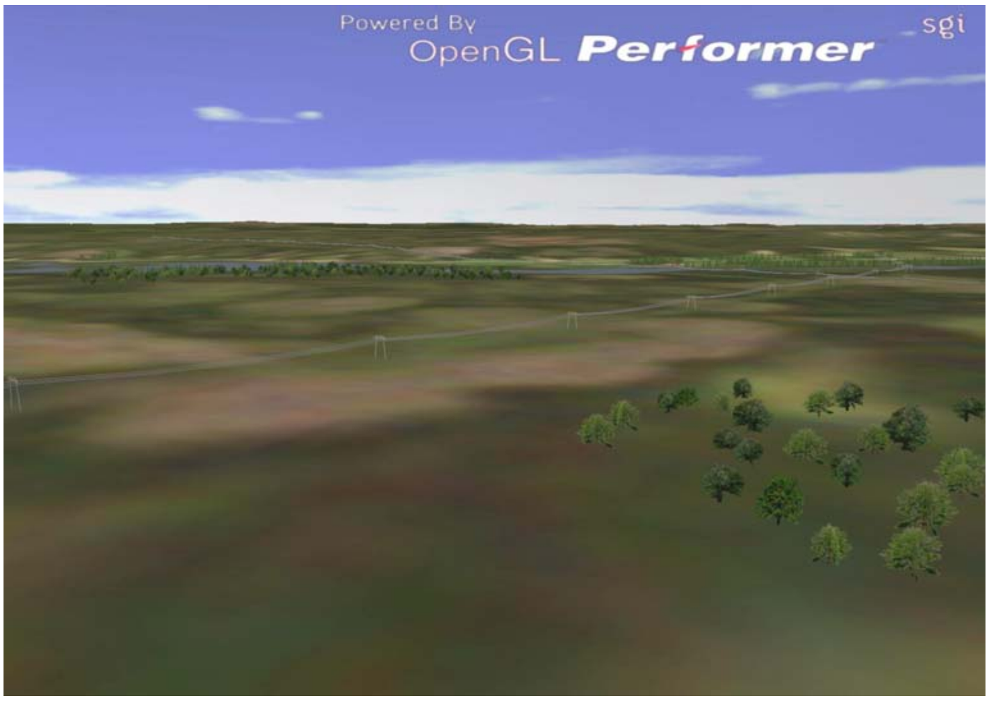


Appendix B-2: The least detailed level allows for only two polygons where it is flat. More polygons are used in the tile where it is needed. With the original method all tiles look like the one to the right; unnecessary polygons around the edges.

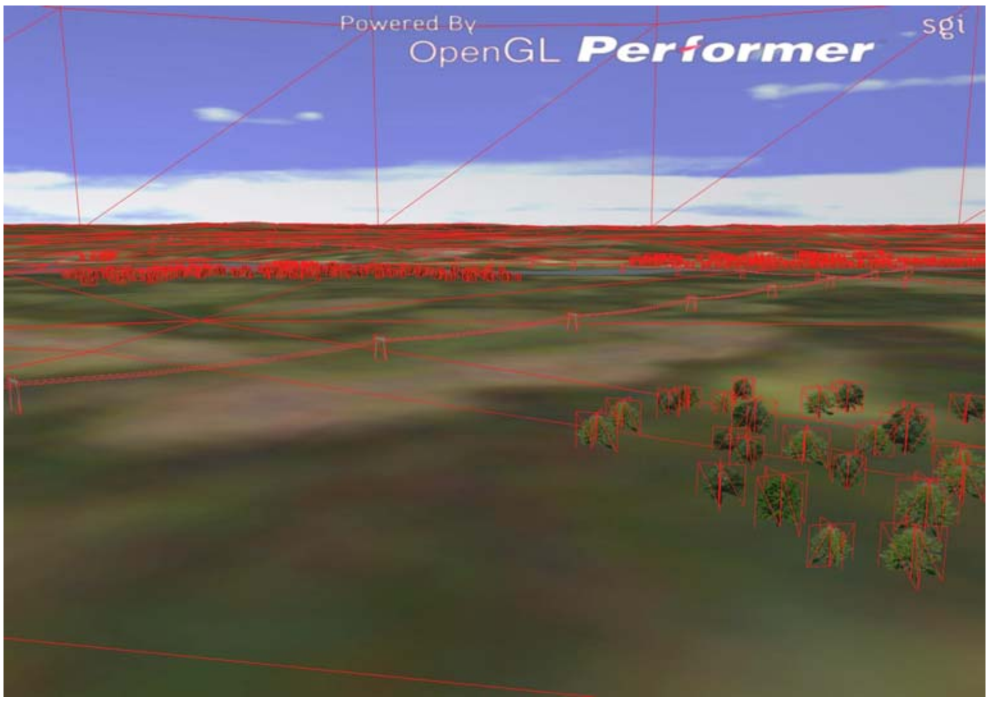


Appendix B-3: View over the rough area near the Norwegian border. The sharp edge is when the generated terrain ends.





Appendix B-4: The two images above shows the terrain with features generated in test 4.



Appendix B-5: The image above shows the terrain with features. The wires show the great number of polygon is needed for the features. If polygons can be saved in the far away part of the terrain more polygons can be used to create features without slowing the simulation down.