Real-time rendering of large 3D scenes using hierarchical mesh simplification

Daniel Jönsson

2009-10-02
Real-time rendering of large 3D scenes using hierarchical mesh simplification

Examensarbete utfört i vetenskaplig visualisering vid Tekniska Högskolan vid Linköpings universitet

Daniel Jönsson

Handledare Mahiar Hamedi
Examinator Karljohan Lundin Palmerius

Norrköping 2009-10-02
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Daniel Jönsson
Real-time rendering of large 3D scenes using hierarchical mesh simplification

Daniel Jönsson

October 6, 2009
3.7 Avoiding gaps ................................................. 22
3.7.1 Skirts ..................................................... 22
3.7.2 Ribbons .................................................. 23

4 Implementation ............................................. 26
4.1 Visibility selection and culling ................................. 26
4.2 Asynchronous streaming ...................................... 26
  4.2.1 Rendering and input/output ............................ 27
  4.2.2 Priority calculations .................................. 27
4.3 Rendering .................................................. 28
  4.3.1 Rendering a section .................................. 28
  4.3.2 Atmosphere .......................................... 28
4.4 Optimizations .............................................. 29
  4.4.1 Texture compression .................................. 29
  4.4.2 Normal map compression ................................ 29

5 Results ......................................................... 30
5.1 Performance ............................................... 30
  5.1.1 Laptop computer test ................................ 30
  5.1.2 Stationary computer test .............................. 31
  5.1.3 Visual results and artifacts ........................... 31

6 Discussion .................................................. 34
6.1 Conclusion ................................................ 34
6.2 Future work .............................................. 34
  6.2.1 Support for general 3D scenes ......................... 34
  6.2.2 Improve file structure and handling .................. 35
  6.2.3 Compression ........................................ 35
  6.2.4 Optimize the prefetch algorithm ..................... 35

References ..................................................... 36
List of Figures

2.1 The original sphere to the right and the decimated to the left. Blue lines represent the distance to the original object. ................................................. 4
2.2 The distance between the original object and the decimated object is called geometric error. .... 4
2.3 A scene graph where the circles represent the nodes (objects). The green node is called the root node and is a parent to the white and yellow node. The red nodes are called leaf nodes and are children to the yellow node. ............................................. 5
2.4 a) Sphere bounding volume b) AABB. ......................................................... 6
2.5 A quad tree structure. .................................................................................. 6
2.6 The gray area represents a gap between two sections. .......................................... 7
2.7 Two neighboring geometries with locked edge vertices. ........................................ 7
2.8 Skirts at one of the edges. ............................................................................. 8
2.9 The front (in red) is defined as the visible set of objects in a scene graph. .................. 8
2.10 The task schedule for the two parallel processes. ............................................. 9
2.11 An expanded frustum $F_I$ is used when calculating which objects to prefetch. ........... 10
2.12 The interval $[0, \delta]$ is divided into $B$ buckets. Buckets close to $\epsilon$ has higher priority since they are more likely to switch LOD representation. ......................................................... 10
2.13 Objects to be replaced are selected from the head of the $\text{InMemoryList}$. The prefetched objects and the objects in the front are thus less likely to be removed from memory. ......................................... 11
2.14 Integration along the camera direction. ................................................................ 12
3.1 Application work flow. .................................................................................... 14
3.2 Blue marble, next generation: a) Height values b) Color values c) Longitude and latitude for the data. .................................................................................. 15
3.3 Definition of coordinate system. ................................................................. 15
3.4 Semi-major (a) and semi-minor (b) axis of an ellipse. ........................................ 16
3.5 World coordinates are transformed to avoid precision errors. .......................... 17
3.6 The optimal frustum in red where $d_n$ and $d_f$ are the near and far plane distances. ....... 17
3.7 How to subdivide a height map. ....................................................................... 18
3.8 The top down approach starts at the root node and decimates all sections from the original height map data. ................................................................. 18
3.9 The bottom up approach starts at the leaf nodes which are triangulated from the height map data. Parent nodes merges the already decimated child data and further decimates it. .... 19
3.10 Four decimated children (separated for explanation purposes) are merged and further decimated to create the parent mesh which later can be used by its' parent. The edges must be locked in order to be able to merge the meshes into one without creating holes. .......................... 19
3.11 Lower level of detail textures are generated by combining four children textures and resizing it to half the size. ................................................................. 21
3.12 Definition of vectors for calculating a normal from neighboring points. ................. 22
3.13 Corners cannot be decimated. The edges must stay intact. .................................. 22
3.14 Projected coordinates with skirts seen from above. .......................................... 23
3.15 A visual description of how to create a ribbon. Keep in mind that vertices at the same height in the picture are actually at the exact same position. .................................................. 23
3.16 Left: Two adjacent edges. Middle: The first ribbon cover a part of the gap but also cover a part where no gap exist. Right: Last ribbon covers the rest of the gap. ................ 24
4.1 Threads must share the OpenGL context ....................................................... 27
4.2 The new task schedule. .................................................................................. 27
5.1 a) Sunset over the Caribbean b) Earth from space c) Artifacts from atmosphere.
5.2 a) Without ribbons a gap appear b) Ribbons cover the gaps.
5.3 Artifacts between different levels of detail.
5.4 Grand Canyon: a) Without using normal maps b) Using normal maps c) Normal maps only.
List of Tables

5.1 Laptop computer performance analysis ............................................... 31
5.2 Time to load compressed textures. ...................................................... 31
5.3 Stationary computer performance analysis ......................................... 31
Abstract

Captured and generated 3D data can be so large that it creates a problem for today’s computers since they do not fit into the main or graphics card memory. Therefore methods for handling and rendering the data must be developed.

This thesis presents a way to pre-process and render out-of-core height map data for real time use. The pre-processing uses a mesh decimation API called Simplygon developed by Donya Labs to optimize the geometry. From the height map a normal map can also be created and used at render time to increase the visual quality. In addition to the 3D data textures are also supported. To decrease the time to load an object the normal and texture maps can be compressed on the graphics card prior to rendering. Three different methods for covering gaps are explored of which one turns out to be insufficient for rendering cylindrical equidistant projected data.

At render time two threads work in parallel. One thread is used to page the data from the hard drive to the main and graphics card memory. The other thread is responsible for rendering all data. To handle precision errors caused by spatial difference in the data each object receives a local origin and is then rendered relative to the camera. An atmosphere which handles views from both space and ground is computed on the graphics card.

The result is an application adapted to current graphics card technology which can page out-of-core data and render a dataset covering the entire earth at 500 meters spatial resolution with a realistic atmosphere.
Acknowledgements

I would like to thank my supervisor, Dr. Mahiar Hamedi, and the other employees at Donya Labs for their advices and encouragements. I would also like to thank my academic supervisor Dr. Karljohan Lundin at the Institute of Science and Technology, Linköping University, for valuable comments and help.
Chapter 1

Introduction

1.1 Introduction

In many computer graphics applications it is necessary to render large and relatively flat 3D scenes in real-time. Such scenes could comprise of terrains, forest, or urban structures. The scene data can be created in computer graphics tools with a large amount of 3D objects containing large meshes and textures or generated by for example satellites equipped with instruments to capture image and height data. To be able to visualize the large data it is often necessary to first reduce it into a simplified form. Donya Labs is a company which specializes in reducing meshes using various decimation techniques. This report describes the work done at Donya Labs and uses their mesh decimation library Simplygon for processing the data.

1.2 Problem description

When facing datasets which does not fit into main memory and can not be rendered directly it is often not sufficient or possible to reduce the data without processing the data before. If the processed data still does not fit into main memory techniques for streaming the data from the hard drive to the main and graphics card memory must be used. To be able to render the entire scene and stream the data a data structure must be created which supports both textures and 3D data.

1.3 Purpose and aim

The aim of this thesis work project is to develop algorithms that make pre-calculations on large 3D scenes, in order to create hierarchical level of detail (HLOD) data structures of the 3D scene. A runtime system should be created for the real-time rendering of the HLOD data. To decimate the 3D data mesh simplification algorithms already developed by Donya Labs must be used. Although not planned for at the beginning of the project an atmosphere was implemented to increase the visual appearance of the final application.

1.4 Outline

Chapter 2 presents some theoretical background and concepts that are used throughout the paper. In chapter 3 the methods for pre-processing the data and building a HLOD structure are explained. Details about the implementation and how to optimize for real time rendering are given in chapter 4. In the last chapter results are discussed and suggestions to future work are given.
Chapter 2

Background and previous work

First some important concepts used in this thesis are explained and then, in the end of this chapter, a brief overview of methods used and considered is given.

2.1 Level of detail

Objects of high detail take longer time to render and consume more memory than objects of low detail. Since objects near the viewpoint need more detail than objects far away it makes sense to divide an object into a different representations which range from low to high detail, in other words level of detail. Depending on error metrics such as distance or projected screen space error a suitable representation is chosen and rendered [1]. There are two fundamental approaches to LOD, discrete LOD and continuous LOD.

Discrete LOD pre-calculates a number of representations and chooses which one to render during runtime. This introduces popping artifacts when switching between representations. Popping artifacts can be alleviated by interpolating between representations, so called geomorphing [2].

Continuous LOD exactly determines the amount of detail needed instead of using a fixed number of representations. The drawback of using continuous level of detail is that it cannot be pre-calculated and thus must use precious CPU resources to calculate the representation at runtime.

Hierarchical level of detail is the concept of merging a set of objects into one [3]. For instance consider the case of having a forest. From far only the shape of the whole forest is important. When approaching the forest individual trees start to be recognizable. Instead of having a LOD for each tree many trees are merged into a single object which can save many calculations both when rendering and culling.

2.2 Decimating geometry

Decimating geometry is the process of turning a high detail model into a lower detailed representation while preserving the shape and appearance. Decimating geometry by hand is a time consuming task but still used in industry today. Automatic algorithms need to take important features into consideration and remove detail where it is not necessary which may or may not produce the desired result. Garland and Heckbert [4] introduced an algorithm which uses quadric error metrics for deciding what to remove. The quadric error is the sum of squared distances from a vertex to its’ neighboring faces on the geometry. The algorithm continually removes the edge formed by two connected vertices with lowest combined quadric error until a desired number of vertices or triangles are reached. They use a greedy approach that moves the original vertices which is unwanted when for instance animating a character. The algorithms used in this thesis are developed by Donya Labs and are based on the same error metric but does not move vertices. In their product decimation API called Simplygon they provide two different methods for deciding how much to decimate geometry: by triangle count or distance bound.

By specifying the triangle count the resulting geometry is reduced until the given triangle count is reached. This method is very fast and does not consume much memory. Setting the distance bound guarantees that the decimated geometry does not deviate more than the specified distance from the original geometry. An example of a sphere decimated using the distance bound, also called geometric error, can be seen in figure 2.1. This method generally produces better results but is slower and consumes more memory. Other features in the Simplygon API are to lock vertices so that they cannot be removed, setting a penalty for borders so that the geometry preserves the shape and weld vertices which are close together.
2.2.1 Cascaded decimation

There is currently no way to get the geometric error caused by decimating a geometry using Simplygon when setting the triangle count. However, a method called cascaded decimation can be used. First, decimate the geometry using an initial distance bound then check if the desired triangle count is reached. If not, then increase the distance bound and continue to decimate the object. Repeat the process until the desired polygon count is reached. The final distance bound is the geometrical error caused by the decimation. The initial distance bound and how much the distance should be increased is dependent on the size and content of the geometry itself. Care must be taken not to set the initial distance to high so that too much is decimated. Increasing the distance by too little every iteration will slow down the process considerably.

2.3 Error metrics

Error metrics can be used for deciding how much to decimate an object or what LOD representation to render. When decimating an object, it is allowed to be reduced more if the error is increased at the cost of visual appearance. If the error is saved, it can also be used at render time to calculate if an object needs to switch LOD representation. Details about the geometric and screen space error are given here.

2.3.1 Geometric error

When decimating an object, it will differ from the original object. The distance between the original and the decimated object is called geometric error. Specifying a geometric error can be difficult since it is dependent on the spatial extent and scale of the object. It can therefore be useful to specify a screen space error instead.

Figure 2.1: The original sphere to the right and the decimated to the left. Blue lines represent the distance to the original object.

2.3.2 Screen space error

The screen space error determines how much the error is projected on the screen. For example, a screen space error of one allows an object to deviate by one pixel on the screen. Instead of switching LOD representation based on the distance from the object to the camera, a screen space error can be used. The user specifies an allowed screen space error, $\epsilon$, which is tolerated for objects in the scene. If the object’s screen space error, $\epsilon_O$, exceeds $\epsilon$, the LOD

Figure 2.2: The distance between the original object and the decimated object is called geometric error.

Figure 2.2: The distance between the original object and the decimated object is called geometric error.
representation need to change into a higher detail level. In [1] Ulrik Thatcher calculates the screen space error of an object using:

\[ \epsilon_O = \frac{\delta}{D} K \]  

(2.1)

where \( \delta \) is the maximum geometric error created when decimating the object, \( D \) is the distance from the viewpoint to the bounding volume and \( K \) is a perspective scaling factor. The perspective scaling factor is calculated from the viewport width and field of view (FOV) using:

\[ K = \frac{\text{viewport width}}{2 \tan \frac{\text{horizontal FOV}}{2}} \]  

(2.2)

The perspective scaling factor only needs to be evaluated when the viewport width or FOV changes. Specifying a screen space error is also useful when letting a user decide how much an object is allowed to be reduced since it is easier to relate to than an arbitrary geometric error. Given that the perspective scaling, allowed screen space error and distance is specified the allowed geometric error can be calculated using:

\[ \delta = \frac{K}{\epsilon_O \cdot D} \]  

(2.3)

2.4 Current hardware

Modern graphics hardware is capable of rendering more triangles than there are pixels on a screen [5]. It is no longer that important to have an exact representation with low triangle count such as the one used in continuous LOD. The bottleneck for rendering terrain is currently the graphics bus [6] which is used to transfer the data to the graphics card. Several ways to reduce the transfer to the graphics card has been introduced. For instance the cached geometry approach introduced by Pajarola and Tirado [7] can be used. The method in [7] remembers what is currently loaded on the graphics card and only removes it when more space is needed. Algorithms designed for rendering must keep this in mind. Throughout the report the graphics card will be referred to as the GPU (graphics processing unit). The acronym GPU will also be used instead of typing graphics card memory.

2.5 Scene graphs

Data which do not fit into main memory are called out-of-core. To render such large data a description is required which contains the extent and spatial location of the data, not the data itself. Such a description is in computer graphics called a scene graph. The scene graph is often built up by nodes which represent objects. The nodes build a tree structure describing the connectivity between the different objects. Scene graph objects can have different roles such as translating child nodes however in this thesis they are all 3D objects. The parent node is a courser description of its child nodes and also bounds all children, a HLOD.

![Scene graph](image.png)

Figure 2.3: A scene graph where the circles represent the nodes (objects). The green node is called the root node and is a parent to the white and yellow node. The red nodes are called leaf nodes and are children to the yellow node.

The extent of a node is described by a bounding volume which is a simpler object than the object it bounds, for example a sphere or an axis aligned bounding box (AABB) as in figure 2.4. An AABB is a box where the faces are oriented such that the face normals are parallel with the axis of the coordinate system used.
2.5.1 Quad tree

A specialization of the scene graph structure is the quad tree. In a quad tree data structure each node has four or none children and one or no parent. The node that has no parent is called the root node and is at the top of the hierarchy. The nodes with no children are called leaf nodes and are at the bottom of the hierarchy. A visual representation can be seen in figure 2.5.

![Quad Tree Structure](image)

Figure 2.5: A quad tree structure.

2.6 Chunked Level of Detail Control

Ulrich [1] introduced in 2002 a method for rendering massive terrains adapted to modern hardware. He uses a quad tree where each node represents a part of the terrain. The root node covers the entire terrain and each child contains a quarter of the parent terrain. Each node, called a chunk, must be a static mesh and is generated in a pre-processing step from a higher detail representation. Ulrich also assigns a geometric error $\delta$ which represents the maximum deviation the chunk has compared to the highest detail mesh. At runtime he traverses the quad tree and selects the chunks which have an error less than a specified screen-space error, $\tau$:

\begin{algorithm}
\textbf{Algorithm 2.1 render\_lod}(node)
\begin{algorithmic}
\If{$\rho$ (node, viewpoint) $\leq \tau$}
\State draw(node.mesh)
\Else
\ForAll{$c \in$ node.children}
\State render\_lod(c)
\EndFor
\EndIf
\end{algorithmic}
\end{algorithm}

To support out-of-core data he suggests that a separate thread should load objects according to algorithm 2.2. When a node has not been used for some time the loader thread should free the geometry data.
Algorithm 2.2 render_lod(node)

node.last_used_frame = current_frame
if not all_child_data_resident(node) Or rho(node, viewpoint) ≤ τ then
  draw(node.mesh)
  for all (c ∈ node.children) do
    request_residency(c)
  end for
else
  for all (c ∈ node.children) do
    render_lod(c)
  end for
end if

2.7 Avoiding gaps

After decimating a section there is no guarantee that the neighboring sections will have the same vertices left on an edge, it is even highly unlikely. In figure 2.6 two neighboring sections have different sets of vertices on their sharing edge which creates a gap where you can see through.

![Figure 2.6: The gray area represents a gap between two sections.](image)

To avoid the gaps there are a number of techniques you can use. Here follows a subset based on previous work.

2.7.1 Lock edges

In [8] Hoppe simply locks all vertices at the edges of all sections. This is possibly the easiest way to remove gaps since no processing needs to be done after the decimation and no extra information needs to be stored. However it means that the sections with lower detail will end up wasting a lot of vertices on the edges, see figure 2.7, which is not desirable.

![Figure 2.7: Two neighboring geometries with locked edge vertices.](image)
2.7.2 Skirts

This method, developed by Ulrich [1], creates vertical polygons which match the edge at the top but nothing particular at the bottom as illustrated in figure 2.8. The bottom of the skirt must stretch below any possible simplification of the other sections. The skirts can use the same texture coordinates as their connecting edge vertices which does create a texture stretch but it is only as large as the error from decimating the section. They can be created before rendering and is also independent of other sections which simplify both implementation and rendering. A disadvantage is that additional vertices needs to be created, which can be quite a few.

![Figure 2.8: Skirts at one of the edges.](image)

2.7.3 Ribbons

A ribbon is defined as triangles which join the edges of two adjacent sections, thereby adding a minimum amount of vertices to a section. The texture coordinates can be assigned in the same way as when using skirts and will also cause a small texture stretch proportional to the error caused by decimation. Information about adjacent section edges is needed to create the ribbons. One combination for each neighboring section in every LOD is required if no constraints are made during rendering.

2.8 Out-of-Core Rendering of Massive Geometric Environments

In [9] Varadhan and Manocha presents a method for rendering a 3D scene so large that it does not fit into the main memory. Their goal is to fetch objects into main memory before they are needed, so called prefetching, for rendering. The objects can have multiple discrete LOD which switch representation based on their screen space error. Loading and rendering is done in two separate threads. The order in which objects and their LOD representation are prefetched is based on the angle between the object and the view direction as well as how far away it is from the view point. The requirement for the algorithm to work is that the entire scene graph representation, which is typically small [9], fits into main memory. All objects can be stored on the hard drive and loaded into memory as they appear. Thus the overhead of the algorithm is about equal to the size of the scene graph representation. First an explanation of how objects are selected for rendering or prefetching is given. Then details about the how the rendering and input/output threads operate and a strategy for removing objects when the memory limit is reached is explained.

The visible set of objects within a scene graph is defined as the front, see figure 2.9.

![Figure 2.9: The front (in red) is defined as the visible set of objects in a scene graph.](image)
The front can be determined by using any subset the following culling\(^1\) techniques:

- **View-frustum culling**: Whether the bounding box of a node is inside or outside the view frustum.
- **LOD culling**: Whether the object satisfies the error bound set by the user.
- **Occlusion culling**: Whether the object is occluded by other objects.

As the viewpoint changes the front also changes based on two events:

- **LOD switch event**: Happens when the LOD of an object is switched because the camera moves closer or farther away.
- **Visibility event**: Occurs when an object that previously was visible/invisible becomes invisible/visible and is caused by rotating the camera or occlusion from objects.

Both threads in the algorithm compute the front, even though it is for different purposes. The purposes for computing the front are explained in the next section.

### 2.8.1 Rendering and input/output

The out-of-core algorithm uses two threads, one for rendering (\(T_R\)) and one for I/O (\(T_I\)). The two threads work in parallel according to the scheme in figure 2.10. First the render thread computes the front as explained in the previous section. When the front has been computed a fetch command together with a list of objects in the front is sent to \(T_I\). \(T_I\) then starts to fetch all objects in the received front. As soon as an object has been fetched it is available for rendering which means that the rendering is performed parallel to the fetching. If objects can not be loaded faster than they are rendered \(T_R\) must wait for the objects to load. When all objects in the front has been fetched a larger frustum is used to calculate a new front called the expanded front. The objects in the expanded front is first prioritized, \(T_I\) then continues to fetch objects in the expanded front until the next fetch command is received. In essence the \(T_I\) has two states. Either it fetches objects which are needed for rendering or it prefetches objects which it believes will soon be visible. The prioritizing of the objects in the expanded front is explained in the next section.

![Figure 2.10: The task schedule for the two parallel processes.](image)

### 2.8.2 Prefetching

To take advantage of the coherency between frames a prefetching technique is used. The goal is to load as many different objects and LOD representations as possible before they are needed for rendering. To be able to know which currently invisible object that should be prefetched first a priority must be calculated. The priority is based on whether the object soon will be visible due to a LOD switch event or due to a visibility event.

---

\(^1\)Culling is the process of determining if an object is visible to the camera or not.
**LOD switch event**

As the camera moves closer to or farther away from an object a different LOD representation may be needed. A higher priority is given to objects which are more likely to switch LOD representation. The LOD switch priority for an object \( O \) is calculated according to equation 2.4.

\[
\text{LODPr}(O) = \frac{1}{|\epsilon_O - \epsilon|} \tag{2.4}
\]

where \( \epsilon \) is the allowed screen space error and \( \epsilon_O \) is the screen space error for object \( O \). Multiple LOD representations can be fetched for an object.

**Visibility switch event**

Objects which will become visible due to rotation of the camera are found by calculating a new front using an expanded frustum \((F_I)\) as illustrated in figure 2.11. The closer the object is to \( F_R \) the higher the priority becomes. If \( \theta_0 \) and \( \theta_1 \) are the two angles defining the field of view for the two view frustums the priority for an object \( O \) is calculated as:

\[
\text{AngPr}(O) = \frac{1}{1 + \frac{\theta}{\theta_0}} \tag{2.5}
\]

where \( \theta < \theta_0 \) and \( \theta > \theta_0 \) otherwise.

The result of equation 2.5 is that objects inside \( F_R \) have a priority equal to one. As objects become farther away from \( F_R \) their priority decreases linearly to a minimum reached when an object is at angle \( \theta_1 \) away from the view direction.

**Sorting objects**

In a large scene it may not be possible to fetch all LOD representations in the front each frame. Therefore the objects are sorted based on a priority. The priority is higher for objects which will is believed to soon switch LOD representation. The objects switch LOD representation when their screen space error exceeds a user specified screen space value \( \epsilon \).

![Figure 2.11: An expanded frustum \( F_I \) is used when calculating which objects to prefetch.](image)

![Figure 2.12: The interval \([0, \delta]\) is divided into \( B \) buckets. Buckets close to \( \epsilon \) have higher priority since they are more likely to switch LOD representation.](image)
example in figure 2.12 the priority order would be $B_3, B_2, B_4, B_0$. Each bucket contains an unsorted queue into which new items are inserted last.

During each frame objects are selected from the bin with highest priority that is not empty. The parent and child of the object is loaded and then classified into the bins which they belong, see 2.3. According to the pseudo-code in 2.4 new objects are fetched until a fetch command is received by the render thread.

**Algorithm 2.3 Classify(X).**

1. Calculate projected screen-space error $\epsilon_O$ of object X
2. Place X in bin $B_i$ whose interval $[i, i+1]$ contains $\epsilon_O$

**Algorithm 2.4 Algorithm for prefetching objects.**

1. while (not recievched fetch command from $P_R$) do
   1. Pick object O from highest priority bin that is non-empty
   2. for all (object $X \in$ parent(O) $\cup$ children(O) $\cup$ O) do
      1. Load X into main memory
      2. Classify(X)
   end for
2. end while

Finally a priority, for object $O$, which combines the screen space error with the angle priority is calculated using:

$$
\text{Priority}(O) = \begin{cases} 
\epsilon_O \cdot \text{AngPr}(O) & \epsilon_O < \epsilon \\
\epsilon_O / \text{AngPr}(O) & \epsilon_O > \epsilon 
\end{cases}
$$

(2.6)

The combined priority is highest for object which are inside the view frustum and close to the viewpoint. As objects move away from the view frustum and view point the priority decreases. The classification step in 2.3 is based on the combined priority.

**2.8.3 Replacement strategy**

Until a specified upper bound memory size has been reached the algorithm keeps loading objects into memory without removing any objects. To know which object to remove when the memory limit has been reached a least recently used (LRU) strategy is adopted. A doubly linked list called $\text{InMemoryList}$ (figure 2.13) keeps track of objects which were recently in the front and loaded into memory. The $\text{InMemoryList}$ uses the following procedures:

![Figure 2.13: Objects to be replaced are selected from the head of the $\text{InMemoryList}$](image)

1. Before starting to prefetch objects a pointer called $\text{StartTail}$ is assigned to the tail of $\text{InMemoryList}$.
2. Subsequent fetching first removes the object from the $\text{InMemoryList}$ if it is in memory and then inserts it at the position directly after $\text{StartTail}$.
3. When the upper memory bound is reached objects at the head of $\text{InMemoryList}$ are removed until enough space is available for the new object. If the object selected for removal is being rendered it skips it and takes the next object.

The method ensures that the objects in the front are removed last. Moreover, since prefetched objects are loaded in an order based on their priority, they are accessed for removal in an increasing order of priority.
2.9 Atmosphere

Although not originally planned it was decided that an atmosphere was to be implemented to increase the visual appearance of the final application. Therefore an in depth explanation of how atmosphere rendering works is not given here. Instead the reader is referred to the two articles [10] and [11] which contains the methods considered for implementation. The requirements of the atmosphere method was that it could handle views from both outside and inside the atmosphere and be calculated in real-time. A brief overview is necessary to understand the concepts of atmosphere rendering and is thus given here.

As the light from the sun travels through the atmosphere it is scattered and absorbed before it reaches the eye or camera. It is common to consider two forms of scattering: Rayleigh and Mie. Rayleigh scattering is what makes the sky blue and is caused by small molecules. Mie scattering spreads the light more uniformly and is caused by aerosols which are larger particles such as dust. The equations involving scattering and absorption can be difficult to solve analytically. Therefore numerical methods are used instead. A common such method is to send rays from the camera and take samples along that ray inside the atmosphere. An example of such a ray can be seen in figure 2.14. To calculate the color which reaches the camera in figure 2.14 the amount of light absorbed, scattered in and out are evaluated at each sample point \( P_i \) and then combined.

\[
I_c(\lambda) = I_s(\lambda) \cdot K(\lambda) \cdot F(\theta, g) \cdot \int_{P_a}^{P_b} \left( e^{\frac{t}{h_0}} \cdot e^{-t(PP_c, \lambda)} - t(PP_a, \lambda) \right) ds
\]

(2.8)

where \( I_c(\lambda) \) determines how much light that is scattered in and \( t(PP_b, \lambda) \) how much that is scattered out at the given wavelength \( \lambda \). \( I_s(\lambda) \) is the intensity of the sun, \( K(\lambda) \) is a scattering constant which is different for Mie and Rayleigh scattering. \( H_0 \) is the height at which the average density occur and \( h \) is the altitude above sea level. A visual explanation of \( P_a \) and \( P_b \) is given in figure 2.14. These equations are computationally demanding and to solve them numerically in a reasonable time the GPU can be used. A very brief overview of two methods which both use the GPU to solve the equations in real-time are given in the next section. Both methods can render the atmosphere from space and ground.

2.9.1 Accurate Atmospheric Scattering

In O’Neil [10] solves the light equations by approximating the out scattering integral with other equations that are less expensive to compute. He performs numerical integration entirely on the GPU which allows the atmosphere to be computed in real-time. The method does not take multiple scattering into account and is dependent on a scale function which means that it only works when the thickness of the atmosphere is 2.5 percent of the planet’s radius and \( H_0 \) is 25 percent of that thickness. For high dynamic range rendering he uses a picture buffer which the scene is rendered to. Then he uses equation 2.9 as a global tone mapping function where the variable \( \text{exposure} \) simulates the dilation of the pupils.

\[
\text{color}_{\text{HDR}} = 1.0 - e^{\text{exposure} \cdot \text{color}}
\]

(2.9)

The source code is available which should make it easier to implement.
2.9.2 Precomputed Atmospheric Scattering

Bruneton and Neyret [11] precomputes the light from all view points, view directions and sun directions which allows them to produce effects such as light shafts and twilight color. Since the equations are precomputed the values can be accessed in constant time. Additionally they take multiple scattering into account. A comparison made between O’Neil’s [10] method using 10 samples per ray and their own without light shafts shows that their method is 75 frames per second faster on a NVIDIA 8800 GTS graphics card. The source code for this method is also available.
Chapter 3

Pre-processing

To be able to render large amounts of data a pre-processing step is performed which turns height map data into 3D objects, optimizes them for rendering and builds a hierarchical data structure of the scene. In this chapter an overview of the entire work flow is first given. Then each pre-processing operation is described in detail.

3.1 Application work flow

The pre-processing is performed in several steps. Figure 3.1 shows the general work flow for generating a hierarchical 3D-scene from a large height map. In the first step the large height map and its accompanying texture is subdivided into smaller, more manageable, sections. A quad tree is used as data structure which means that the number of sections in width and height must be a number raised to the power. Otherwise the quad tree data structure can not be formed. Normal maps can optionally be created from the large height map, if not normals are calculated from the mesh after decimation using an built in function in Simplygon. Details about how to create normal maps from the height map is given in section 3.6.

When the large height map and texture has been subdivided the 3D data can be created. The first LOD is created from each section by triangulating the height map and then decimating the result, details are given in section 3.4. To create the next LOD four geometries from the previous LOD are combined and decimated. How the geometries are combined is explained in section 3.4.2. The texture- and normal maps are also combined and then resized to the same size as one texture at the previous LOD. The LOD generation process continues until there only is one top section remaining.

3.2 The data

Researchers ([8], [13]) \(^1\) have previously used data sets such as Grand Canyon or Puget Sound. The Grand Canyon dataset has 4097 x 2049 height samples and Puget Sound has 16,385 x 16,385 samples. Both datasets

\(^1\)Losasso and Hoppe [5] use a height map with 216,000x93,600 samples but they compress it from 40,4 GB to 355 MB.
take less than 0.6 GB of storage space uncompressed and therefore fit into the main memory of modern computers. Unfortunately they are too small since they fit into main memory of modern computers. Instead the blue marble next generation (referred to as blue marble) dataset provided by NASA is used. The blue marble dataset has 86,400 x 43,200 height samples covering the entire earth with a spatial resolution of 500 meters. The data is stored as 16-bit signed integers and take about 7 GB of storage space uncompressed. NASA also supply a texture file with 86,400 x 43,200 color samples stored in RGB (Red Green Blue) order as 8-bit unsigned integers which consume about 11 GB of storage size uncompressed.

![Figure 3.2: Blue marble, next generation: a) Height values b) Color values c) Longitude and latitude for the data.](image)

This dataset requires special treatment. The data is projected using cylindrical equidistant (geographic, Platte-Carre) projection, use the WGS-84 datum, and has a spacing of 0.004166667 degrees per pixel. To project the data into world coordinates the elliptic projection in equation 3.1 is used.

\[
\begin{align*}
x &= (N(\phi) + h)\cos(\phi)\cos(\lambda) \\
y &= (N(\phi) + h)\cos(\phi)\sin(\lambda) \\
z &= (N(\phi)(1 - e^2) + h)\sin(\phi)
\end{align*}
\]

(3.1)

Here \(\phi\), \(\lambda\) and \(h\) are the latitude, longitude and height above the earth surface, as illustrated in figure 3.3.

![Figure 3.3: Definition of coordinate system.](image)

\(N\) is calculated using

\[
N(\phi) = \frac{a}{\sqrt{1 - e^2 \cdot \sin^2(\phi)}},
\]

where \(a = 6378.137\) km is the semi-major earth axis, \(b = 6356.752314245\) km is the semi-minor earth axis, see figure 3.4, and \(e\) is the eccentricity of earth defined as

\[
e^2 = 2f - f^2, \\
f = \frac{a - b}{a}
\]

(3.3)

To get the real world coordinates the geoid must be taken into consideration which corrects the fact that very few points on the earth lie on a perfect ellipsoid. The geoid approximates the height relative to the ellipsoid and can, for WGS-84, vary between values of -100 m in the Sri Lanka to 60 m in the North Atlantic. For simplicity the WGS-84 datum will not be taken into consideration when calculating world coordinates.
Even reading the large files can be problematic. Using the default 32 bit a file size of $2^{31} \approx 2$ GiB can be read which clearly is a problem when reading the blue marble data. Instead 64 bits must be used which enables $2^{63} \approx 9$ million TB large files to be read, more than enough. A binary image reader was written which has the ability to cache certain areas of the data.

### 3.2.1 Precision errors

Due to the large spatial difference of the earth and the fact that current graphics cards only can use 32-bits float values for representing values errors are introduced when rendering. A 32-bit float can accurately represent values in the range from $10^{-3}$ (millimeters) to about $10^9$ (1,000 km). The earth has a radius of 6,378 km which in itself does not represent a problem if not millimeter accuracy is necessary. If the camera is located at the surface of the earth the resulting values in the view matrix will be large. The large values in the view matrix will be multiplied with the coordinates of the earth which also have large values. The resulting multiplication of the values will be so large that accuracy is lost. The precision artifacts therefore start to show when rendering the model.

#### Camera jitter

A problem referred to as camera jitter occurs when large coordinates needs to be multiplied with the OpenGL transformation matrices. The scene start to shake as soon as the camera rotates or moves. This occurred in my application when the camera was close to the ground. To solve this problem (0,0,0) is used as the camera position when generating the view matrix. The view matrix can be created according to:

$$
M_{\text{view}} =
\begin{pmatrix}
  r_x & r_y & r_z & r \cdot p_{\text{camera}} \\
  u_x & u_y & u_z & u \cdot p_{\text{camera}} \\
  v_x & v_y & v_z & v \cdot p_{\text{camera}} \\
  0 & 0 & 0 & 1
\end{pmatrix}
$$

where $r$ and $u$ are the vectors pointing to the right and up relative the camera. $v$ is the view direction and $p_{\text{camera}}$ is the camera position. When an object is created the world position, $p_{\text{world}}$, is translated relative to a local origin, $o_{\text{origin}}$, according to:

$$
p_{\text{local}} = p_{\text{world}} - o_{\text{origin}}
$$

When rendering the relative position, $p_{\text{relative}}$, is calculated using

$$
p_{\text{relative}} = o_{\text{origin}} - p_{\text{camera}}
$$

and then pushed onto the matrix stack which forms a matrix with small numbers. If floats are used for the camera position and object origin the positions would still get inaccurate if they go beyond the 1,000 km range. Therefore double precision is used for those values which ensures that accuracy is not lost until 1 trillion km is reached. An illustration of how the camera and objects are translated can be seen in 3.5.

#### Z-buffer

For optimal performance the near and far clipping plane should be set such that the scene tightly fits within the two intervals. Only a part of the scene will be rendered if the far clipping plane is to close and setting the far clipping plane to far away makes the Z-buffer loose precision. In [14] O’Neil solves the problem by exponentially scaling the distances of objects beyond half the distance from the far clipping plane. He then scales size of the objects by the same factor. Since only one object need to be considered another approach than O’Neil uses for
rendering impostors\(^2\) is used. Noting that the atmosphere is the outer boundary of the scene and that it is spherical the frustum is resized so that it bounds the entire visible scene. O’Neil does not take the curvature of the planet into consideration, he sets the far plane at the back side of the planet. Instead the far plane is set at the center of the planet, according to figure 3.6, since the back side it will be occluded anyway. The near plane distance is simply \(\max(1, d_f - r)\).

![Figure 3.6: The optimal frustum in red where \(d_n\) and \(d_f\) are the near and far plane distances.](image)

3.3 Hierarchical data structure

The core data structure is based on a quad tree as described in section 2.5.1. A height map can be partitioned according to figure 3.7 if the width and height are of size \(n^2 + 1\) to create a quad tree structure. This is not the case with the blue marble data which is used in this project. To solve this problem the constraint is loosened by, as outlined in 3.1, instead subdividing the height map into \(n^2\) sections where each section does not have any size constraints. This made it possible to build a quad tree structure of the data even though the actual height map did not meet the \(n^2 + 1\) size criteria. The subdivision of the height map follows the pattern of the leaf nodes in figure 3.7, which means that they have the same height data at the edges.

\(^2\)An impostor is a precomputed image-based representation of an object.
3.4 Decimating geometry using Simplygon

Simplygon takes position, triangle indices and texture coordinates as input. The height map data must therefore first be turned into a 3D mesh which is simply a matter of connecting the height data in the right order. No further explanation will be given here as it can easily be looked up on the internet. Since the data is out-of-core a strategy for partitioning the height map into sections must be evaluated. The top down and bottom up approaches are described here along with their disadvantages and advantages. A strategy for storing the processed data is also given. For each partitioned section the data is turned into a 3D mesh and processed through Simplygon. Besides the two methods, triangle count and distance bound, described in section 2.2 a combination of the two was tested to get optimal performance. The following sections first describes how to partition the height map into sections to form a quad tree. Then the file structure for the processed data and the experience from decimating the sections using the three different methods follows.

3.4.1 Top down

In this approach the entire height map is first triangulated and decimated. Then the height map is partitioned into four quads according to figure 3.7 which are decimated and further partitioned until the desired number of levels is reached. The top down approach was tested early in the development but was found to be insufficient when
it came to large height maps. Since the entire height map must be triangulated it can create so much data that it
does not fit into the main or the virtual memory. When the virtual memory limit is reached the application crashes.
Another issue is that no advantage of the previously decimated sections can be taken since every quad is decimated
from the original triangulation. The consequence is that it is both slower and consumes more memory compared
to the bottom up approach. Therefore the top down approach was abandoned rather early.

3.4.2 Bottom up

![Diagram of the bottom up approach](image)

Figure 3.9: The bottom up approach starts at the leaf nodes which are triangulated from the height map data.
Parent nodes merges the already decimated child data and further decimates it.

When decimating from bottom up all sections in the highest level of detail are first decimated. Two versions are
created for each section. The first version is the one which will be used by the application for rendering. The
second version is used when decimating the next LOD. All edges of the second version are locked and it is not
reduced as much as the first version. Additionally it saves, per vertex, the corresponding height map index and
height value. To create the next LOD the four child meshes with locked edges are merged and the duplicate
vertices are welded together as illustrated in figure 3.10. If not the edges would have been locked it would be
difficult to know how to triangulate the merged mesh and holes would appear. The height map indices and height
value stored per vertex are used to recalculate the position relative to the section local origin before merging the
four children. This also ensures that no precision is lost.

![Diagram of decimation process](image)

Figure 3.10: Four decimated children (separated for explanation purposes) are merged and further decimated to
create the parent mesh which later can be used by its’ parent. The edges must be locked in order to be able to
merge the meshes into one without creating holes.

3.4.3 File structure

To organize all data a folder hierarchy similar to the quad tree structure is created. For each level, L, in the
hierarchy a folder is created called “Level_L”. Then for each section in this level a folder is created named after its
row, Y, and column, X, in the height map: “Row_Y_Column_X”. In each section folder the data is stored together
with a text file describing its child relationships and the path of its texture and normal map. For example the folder of the first column and row in the first level of detail would be named “Level_0\Row_0\Column_0”.

3.4.4 Decimating using triangle count

In many situations it is good to have control over the triangle count of the decimated result. Having too much triangles makes it impossible to render the scene in real time but having too few can result in a poor visual experience. As mentioned in 2.2 there is no way to find out the geometric error when using a fixed triangle count in Simplygon. Therefore cascaded reduction can optionally be used which allows the geometric error to be approximately decided. However using cascaded reduction introduces difficulties when deciding what initial distance bound to use and how much to increase the error. Both depend on the data being decimated. When performing the cascaded reduction the error is increased by multiplying it with a fixed number such as 1.1 or 2. In the blue marble dataset the sections vary in size from small near the North and South Pole to large at the equator. Various methods were discussed for choosing the initial error such as using a fixed percentage of the length of the diagonal of a section. Using only triangle count for decimation does not allow flat areas such as oceans to be reduced as much as they could without loosing details. Since cascaded reduction uses the distance bound method oceans will be decimated allot but at the cost of a speed, data dependent variables and the same issues as using only distance bound.

3.4.5 Decimating using distance bound

The approach was here to calculate the allowed geometric error based on a given pixel error at a certain distance with a fixed horizontal screen resolution. The idea is that errors, which are less than one pixel wide on a screen, will not be noticed. In this approach you do not care as much about the given triangle count but instead imply that you want the best possible quality given a pixel error. A distance bound is set for the highest detailed sections. The lower detail sections get their distance bound by doubling the distance bound at the previous level. This gives the effect that the detail is halved for each LOD. Choosing the initial error can be difficult since again you do not want to many triangles in the final scene and also the extent of the sections vary as described in the previous section. When using the distance bound algorithm memory consumptions could go up to as much as 2 GiB when reducing approximately 1.8 million triangles. As soon as the virtual memory is needed the process slows down considerably. Furthermore it is difficult to take advantage of multiple cores by decimating multiple sections at the same time since so much memory is consumed by one decimation.

3.4.6 Decimating using the best of two worlds

It was found that the best way to decimate a section in terms of both speed and quality was to first decimate the mesh using the faster triangle count to a fixed number of triangles which represent an upper bound for the final result. After the mesh has been reduced with the faster method the distance bound is allowed to continue to decimate until the given distance bound is met. Using this approach allows for oceans to be heavily decimated in a reasonable time and without as much memory overhead.

3.5 Textures

The large texture needs to be partitioned to be able to fit into the GPU memory. Therefore a strategy similar to the bottom up approach is used. The first level of detail textures are created by subdividing the large texture map in the same size as the height map sections. There is also an option to resize them to the nearest power of two in width and height since graphics cards handle those sizes better. If textures are going to be compressed the size must be a power of two. When the next level of detail is created the four accompanying textures are merged and resized to the same size as one texture map previously had. A texture hierarchy with the same quad tree structure as the terrain data is thus created.

3.6 Creating normal maps

It was found that the massive decimation of the terrain data removed many of the fine details when only using per vertex normals. To bring back the fine details it was decided that normal maps should be used as they are not dependent on how many vertices a mesh has. Instead the detail is dependent on the resolution of the normal map. Another advantage is that they can be compressed using either color compression techniques or algorithms
specifically designed for normal maps. The highest level of detail normal maps are created from the height map data. The other level of details are merged in the same way as textures, see 3.5, with the exception that the normal map data is normalized after resizing. There are two possibilities when creating and using normal maps: object space or tangent space normals. A short description of the two methods follows here. Then an explanation of how to create the highest level of detail normal map is given.

### 3.6.1 Object space normals

Object space normals are stored in world coordinates which means that they point in their actual direction. However if the object rotates all normals will have to be rotated by the same rotation. It makes them simple to compute and use but expensive if the object is rotated.

### 3.6.2 Tangent space normals

Tangent space normals are stored in another coordinate system which is aligned to the plane of a face, called tangent space. Therefore only two coordinates needs to be stored as the third always point in the same direction as the face normal. Using tangent space normals are more complicated since the basis vectors needs to be calculated and later sent to the shader where the lights, camera etcetera can be transformed into tangent space. However since the tangent space will remain aligned to the face normal you do not need to transform the normals which are many more than the lights, camera and vertices when rotating the object.

### 3.6.3 Calculating object space normal from height map

The object space normal was chosen because it is the simplest and fastest to implement. Also the terrain is not expected to rotate or even move which means that there will be no problems with the disadvantages of the method. The four neighboring values in horizontal and vertical direction are taken into consideration when creating a normal map. Vectors according to figure 3.12 are used to calculate the normal at a point in the height map.

\[
\mathbf{n} = \text{normalize}(e_1 \times -e_3 + -e_2 \times e_3 + e_4 \times -e_2 + e_1 \times -e_4)
\]  

(3.7)

After calculating the normal for each vertex the result is saved as an image with PNG\(^3\) compression where the RGB values correspond to the XYZ coordinates. To be able to store it as an ordinary texture with 8-bits for each component the values needs to be scaled to the range [0 255] using \((\mathbf{n} + 1.0) \cdot 127.5\). When accessing the normals in the GLSL shader the values are scaled back into the [-1 1] range using \(\mathbf{n} \cdot 2 - 1\) (texture values in the shader are already scaled to the [0 1] range). Note that if latitude is less than zero the order in 3.7 must be reversed.

---

\(^3\)Portable Network Graphics (PNG) is an image format for lossless compressed data. See [http://www.libpng.org/pub/png/](http://www.libpng.org/pub/png/) for more information.
3.7 Avoiding gaps

To avoid holes between the sections a method for covering them must be used. The three methods described in 2.7 were all implemented and tested. The locked edges method was used early in the project since the ability to lock vertices is implemented in Simplygon, thus making it trivial to implement. From the start the skirt method was intended to be used but it was found to be inadequate, as explained in the following section. Therefore the more complicated ribbon method was implemented. Independent of which technique that is used the decimations in figure 3.13 must be prevented. Therefore the corner vertices are locked and to preserve the edges of a section a penalty for removing borders are set. The penalty allows vertices lying on an edge to be removed while still preserving the edge. The advantages and disadvantages of the two primary methods will be described in the following sections.

3.7.1 Skirts

Rather late in the project it was noticed that skirts works poorly when using projected coordinates which is the case with the blue marble data. When projected coordinates are used the bottom part of the skirt is lowered toward the center of earth. The reason why it works poorly is illustrated in figure 3.14. Since the edges are not straight,
rather spherical, there will be a visible gap when viewing from above. Even though the curved edge has skirts it cannot compensate for the fact that the opposite edge is approximated by a straight line. There are ways to remove this artifact, for instance change the direction where the bottom part of the skirt is lowered such that it intersects the opposite skirt. Changing the direction introduces dependencies on the data since information about how large the angle needs to be in order to cover the cracks but not intersect the opposite section is required. Therefore skirts were abandoned for projected coordinates but can still be used for unprojected data.

3.7.2 Ribbons

This method creates a ribbon which joins the edges of two adjacent sections. Since this solution uses the actual positions of the vertices on the edges there will be no problem when projecting the coordinates. The number of possible combinations of ribbons between neighboring sections is constrained by limiting them to not differ more than one level of detail. Further limitation of combinations can be achieved by realizing that only two ribbons are needed for same level of detail. This limits the number of ribbons for each section to six, four for children edges and two for same LOD edges.

![Figure 3.14: Projected coordinates with skirts seen from above.](image)

Since knowledge about neighboring sections is required the creation of ribbons must be done as a post processing step after the hierarchical structure has been formed and the sections have been decimated. The algorithm requires that both edges are sorted from their minimum to their maximum value. No triangles are needed where two following coordinates are the same, therefore it skips those and searches for the closest point on either edge. When the closest point is found a triangle is formed from the two known points to the next closest point. This process continues until the last edge coordinate is reached. See figure 3.15 and pseudo code in 3.1. To get rid of precision errors and not worry about local coordinates the corresponding height map index is used instead of their actual coordinate. Since data cannot be accessed from other buffers on the GPU the section data need to be
independent of other sections. Therefore vertices which are missing from the edge are copied to the section that will render the edge. If the edge is in the south or west direction the triangle index order is reversed. Artifacts can be created for some cases. In figure 3.16 a case is illustrated where one ribbon covers an area which should not be covered. The result is that the ribbon blocks the view to the other section.

![Diagram of gaps, ribbons, and artifacts](image)

Figure 3.16: Left: Two adjacent edges. Middle: The first ribbon cover a part of the gap but also cover a part where no gap exist. Right: Last ribbon covers the rest of the gap.
Algorithm 3.1 Calculate ribbon.

Require: Edge vertices from two adjacent edges sorted from minimum to maximum value.

atPos1 ← 0
atPos2 ← 0
ribbonTriangles ← empty

while atPos1 < (nrOfEdgeVertices1) AND atPos2 < (nrOfEdgeVertices2) do
  coord1 ← edgeVertex1[atPos1]
  coord2 ← edgeVertex2[atPos2]
  nextCoord1 ← edgeVertex1[atPos1+1]
  nextCoord2 ← edgeVertex2[atPos2+1]
  {Search for the closest vertex}
  while coord1==coord2 do
    if nextCoord1 < nextCoord2 then
      atPos1++
    else if nextCoord1 > nextCoord2 then
      atPos2++
    else
      {Next coordinates are the same, no need to create ribbon}
      atPos1++
      atPos2++
    end if
  end while
  triangle ← empty
  triangle[0] ← atPos1
  triangle[1] ← nrOfEdgeVertices1+atPos2
  if edge is in SOUTH or WEST direction then
    {Switch triangle order}
    swap(triangle[0], triangle[1])
  end if
  if nextCoord1 < nextCoord2 then
    triangle[2] ← ++atPos1
  else if nextCoord1 > nextCoord2 then
    triangle[2] ← nrOfEdgeVertices1+(++atPos2)
  else
    atPos1++
    atPos2++
    if worldCoord1 < worldCoord2 then
      triangle[2] ← atPos1
    else
      triangle[2] ← nrOfEdgeVertices1+atPos2
    end if
  end if
end while
ribbonTriangles ← push(triangle)
Chapter 4

Implementation

The program is written in C++ using OpenGL for graphics. Shaders are written in the OpenGL shading language (GLSL). WxWidgets is used for handling images, graphical user interface and threading.

4.1 Visibility selection and culling

To select which sections to render the algorithm start at the root node of the scene graph and traverses down. There are two criterions for stopping the traversal. Either the object is culled or the screen space error is smaller than a user specified error. This allows objects of high detail to be rendered near the camera and objects of lower detail which cover a greater extent to be rendered far away. Pseudo code for computing the front is given in 4.1.

Algorithm 4.1 Compute front.

```
ComputeFront(front, node)
if Cull(node) then
    if (node.HasChildren and getScreenSpaceError(node, viewPoint) > ϵ) then
        for all ( node X ∈ Children(node) ) do
            ComputeFront(front, node)
        end for
    else
        front ← push(node)
    end if
end if
```

AABBs are used as bounding volumes and the distance is calculated by choosing distance to the the nearest corner. If the geometric error is so small that it results in a change in LOD when the distance is smaller than half the diagonal of the bounding box it can make the LOD switch when the viewpoint is inside the bounding box (for example in the middle). This problem can be alleviated by also checking the distance to the six planes which the AABB comprise of or solved by forcing the geometric error to be so large that the LOD shifts when being the half the diagonal distance away from the the bounding box.

4.2 Asynchronous streaming

Even though Ulrik [1] proposed a simple method for asynchronous streaming the method developed by Varadhan and Manocha [9] was chosen, see section 2.8. This is mainly due to two reasons: The cache system in [1] only works for LOD switch events and it is specific to the terrain method. The advantage of [9] is that it is designed for general 3D scenes which means that this method could be used if other objects were to be added later. In this thesis view frustum culling and LOD culling is considered when calculating the front. Although the basic principle is the same as in [9] a few modifications have been made. Why and which will be described in the following sections. Since the replacement strategy is exactly the same it will not be described again.
4.2.1 Rendering and input/output

The out-of-core algorithm uses two threads, one for rendering ($T_R$) and one for I/O ($T_I$). Both threads work in parallel and use mutexes to share information safely. The task schedule scheme that Varadhan and Manocha [9] suggest only considers loading objects into main memory, not to the graphics card memory. Since multiple threads cannot access the OpenGL context at the same time, see figure 4.1, it is necessary to take this into consideration when designing a scheme for sending objects to the GPU memory from a separate thread. In this project it was desirable to have objects available for rendering without needing to first transfer the objects from the main memory to the GPU memory in the render thread. This could possibly save time in the render thread. Therefore a modified scheme for the two processes in figure 4.2 is presented. The scheme starts the same by letting $T_R$ compute the front depending on the current view point. Meanwhile $T_I$ fetches objects, which were calculated in the previous frame, to the GPU instead of the main memory. When the front has been computed a fetch command together with a list of objects in the front is sent to $T_I$. In the new scheme tasks which do not require the OpenGL context, such as selecting which ribbons to render, can be performed in $T_R$ while the front is fetched to the GPU. When the front has been fetched an expanded front is calculated and then a command is sent to $T_R$ which now can start render the objects. During rendering no objects can be fetched to the GPU. Instead they can only be sent to the main memory. To not stall $T_R$ when prefetching objects a time limit based on the amount of time to render the previous frame is sent to $T_I$. If the time taken to load an object exceeds the time limit $T_I$ waits for a fetch command instead of prefetching more objects. Furthermore the prefetching of an object is partitioned into smaller parts. $T_I$ checks for a fetch command between each part which consist of loading 3D data, texture and normal map. If a fetch command is received while prefetching an object the partially loaded data is deleted and the rest of the loading process is aborted.

![Diagram of Threads and OpenGL Context](image)

Figure 4.1: Threads must share the OpenGL context

![Diagram of New Task Schedule](image)

Figure 4.2: The new task schedule.

4.2.2 Priority calculations

In many aspects the priority implementation is done in the same way as in [9]. A few changes has been made which will be explained here. Instead of sorting objects into buckets a simpler approach is used. A queue where all objects are sorted based on their combined priority was chosen since it was faster to implement. The drawback is that if many objects are used the sorting of one queue takes longer time than using several bins. The same
equations, as described in section 2.8, for calculating priority are used except for the combined priority which
instead is calculated as:

\[
\text{Priority}(O) = \text{AngPr}(O) + \frac{1}{1 + |e_O - e|}
\]  

(4.1)

The new combined priority has a maximum value of two and decreases as objects move away from the view frustum or is less likely to change LOD. The methods for classifying and prefetching objects have also been changed in order to be able to load objects to the GPU. In the beginning of each frame the prioritized queue is emptied and all objects in the expanded frustum are classified according to the pseudo-code in 4.2. As can be seen in 4.3 \(P_I\) loads objects from the prioritized queue either to main memory or GPU while waiting for a fetch command. After an object has been loaded it is also classified according to 4.2.

Algorithm 4.2 Modified Classify(X).

\[
\text{for all } (\text{object } X \in \text{parent}(O) \cup \text{children}(O)) \text{ do}
\begin{align*}
\text{if } & X \text{ not in memory } \text{ then} \\
& \text{Calculate priority} \\
& \text{Insert object into prioritized queue}
\end{align*}
\text{end if}
\text{end for}
\]

Algorithm 4.3 Modified algorithm for prefetching objects.

\[
\text{while } (\text{not received fetch command from } P_R) \text{ do}
\begin{align*}
& \text{Get object } O \text{ with highest priority} \\
& \text{if } (\text{OpenGL context is available}) \text{ then} \\
& \quad \text{Load } O \text{ into GPU memory} \\
& \text{else} \\
& \quad \text{Load } O \text{ into main memory} \\
& \text{end if} \\
& \text{Classify}(O)
\end{align*}
\text{end while}
\]

4.3 Rendering

Before all sections are rendered they are sorted from front to back based on the distance to the camera. Also the OpenGL states which are the same for all sections are set before rendering them and deactivated after. First the terrain is rendered and then the atmosphere.

4.3.1 Rendering a section

Each section which is loaded to the graphics card has two framebuffers. One of the framebuffers contains vertex data, normals (if used) and texture coordinates (if used) aligned after each other. The second framebuffer contains the triangle indices for the terrain and ribbons (if used) aligned after each other. At render time the entire section is rendered first using indexed triangles. Then each ribbon selected for rendering is rendered using the corresponding offset into the triangle index framebuffer.

4.3.2 Atmosphere

When it came to implementing a method for rendering atmosphere there was not much time left. Therefore the method developed by O’Neil [10] was chosen since it is easier to implement but can still render atmosphere from both space and ground. Instead of performing the high dynamic range calculations by first rendering to a texture it is done directly in the fragment shader. The ground is shaded using the altitude above sea level but since local coordinates is used the vertex position in the shader cannot be accessed directly. To get the altitude the local origin is sent to the shader, the vertex position is translated by the origin and finally the length of the vector is calculated. The outer boundary of the atmosphere is represented by a sphere with 2.5 percent larger radius than the earth radius.
and is stored in a display list to decrease render time. Before rendering the sphere the appropriate shader is set, blending is activated and the order in which triangles are culled is set to clockwise thus removing the triangles which cover the earth. The main work is done in the shaders of which the source code can be found in [10].

4.4 Optimizations

Before performing the optimizations explained here the application had trouble rendering in real-time. The main bottlenecks were identified as the loading of textures and 3D data. At first the text based Wavefront obj format was used for the 3D data but it was later converted to a binary format where all data was aligned after each other. The loading of textures required more work and will be explained in the following sections.

4.4.1 Texture compression

Instead of using the CPU to decompress an image and then send uncompressed data to the GPU compressed data can be sent directly to the GPU which saves both time and bandwidth. The data must first be compressed using the compression algorithms available on the GPU such as DXT1 for RGB or DXT5 for RGBA. It is possible to let OpenGL decide the compression scheme by using the function glHint() with parameters GL_TEXTURE_COMPRESSION_HINT and GL_NICEST. After the image has been compressed on the GPU it is retrieved and stored with a header that specifies the height, width, compression format and the size of the data.

4.4.2 Normal map compression

Compressing a normal map using compression techniques for ordinary textures introduces artifacts since they are designed for color, not for general vectors. There are a number of ways to alleviate or solve this problem. DXT5 uses higher accuracy for the alpha channel and can be compressed independently which can be taken advantage of by storing one vector value in that channel. DXT5 also uses 1 more bit for the green channel. One coordinate can be stored in the green channel, one in the alpha channel. When decompressing one can use the relation

\[ z = \sqrt{1 - x \cdot x + y \cdot y} \] (4.2)

since the normal is normalized and take advantage of the higher precision. Another possibility is to compress the normals using the 3Dc format which is designed for normal map compression. This requires that you have either an ATI X800 (or newer) or a graphics card from the NVIDIA 7100 series (or newer). The 3Dc compression format was chosen to get the best result. The format only uses two components which means that equation 4.2 needs to be used in the shader to get the third. Other than that it is used in the same way as other compressed image formats in OpenGL.
Chapter 5

Results

An application capable of pre-processing and rendering out-of-core environments was successfully implemented. Two different kinds of height map data are supported: ordinary and cylindrical equidistant projected. Three different methods for covering gaps are available. An atmosphere computed entirely on the GPU using vertex and fragment shader was added for additional visual appeal.

5.1 Performance

The blue marble dataset was divided into 64x64 sections, each with 1351x676 height samples. Textures and normal maps where resized to 1024x512 for the highest detail level and 512x256 for the others so that they could be compressed. Each section of the highest LOD took about 100-120 seconds to decimate on an Intel® Core 2 Quad processor Q8400 with 3GB memory. Four processes working on a quarter of the data each ran in parallel which reduced the pre-processing time from about a week to a few days. The maximum number of triangles was set to 60,000 and the most detailed geometric error to 75. The total size of all data became about 4.1 GB. The original data would have consisted of 7.3 Billion triangles which no graphics card can render today. Finally the geometric error was changed before rendering so that the LOD switched when being half the bounding box diagonal away from each section. Two tests were done, one with an older laptop and one with a new stationary computer. The screen resolution was set to 800x600. Atmosphere was turned off during the tests to only measure the performance of the terrain rendering. The frame per second (FPS) measures how long it took to render a whole frame while render time measured the time it took to render the terrain only. Triangles per second were calculated by dividing the average amount of triangles rendered with the average total time to render a whole frame. Fetch to GPU measured how long it took to load the front to the GPU. Four different test cases were evaluated:

1. **Cached on GPU**: All data was stored on the GPU by repeating the flight two times and only perform measurements the second time.
2. **Prefetch to main memory**: Data was only prefetched to main memory.
3. **Prefetch to GPU**: Data was prefetched to main memory during rendering and GPU while not rendering.
4. **No prefetch**: Data was only fetched when needed for rendering.

There was no time left at the end of the thesis to perform tests of other methods on the same computer. A good candidate for comparing the performance would have been Open Scene Graph (OSG) [15] which has the ability to prefetch LOD terrain from height maps. A last note is that it can take a while to initialize the application the first time it starts. It seems as if the result is cached because the startup time is significantly reduced when launching the application a second time.

5.1.1 Laptop computer test

This test was done using a flight path starting at 5,000 km altitude at latitude 80, longitude -180 in the direction toward the horizon. The speed was set to 15,000 m/frame which was maintained over 722 frames (about 30 seconds at 24 fps). Measurements where done on a laptop computer with an Intel® Core Duo processor T2500, 2.5 GB RAM and ATI X1400 graphics card. The measurements in table 5.1 show that the the scheme for prefetching objects to the GPU performs better than if objects were only prefetched to main memory. Also by prefetching
objects a slightly smoother experience can be given at the cost of lower performance. The lower performance when prefetching objects is caused by the object which is prefetched when a fetch command is received in the IO thread. The IO thread must then load the object before fetching all objects in the front and thereby stalls the render thread. The FPS drop, most noticeable when no prefetching is done, occurs when new sections which have not been loaded to the GPU appear in the front. When prefetching only to main memory and to GPU memory 9\% respectively 8\% of the objects were not cached before a fetch command was sent.

![Table 5.1: Laptop computer performance analysis](image)

Furthermore the time to load the binary object format and compressed textures into the GPU was measured. It turned out that loading an object in binary format of 5,000 triangles took about the same time as loading an object with 50,000 triangles, only 0.1 ms differed. Loading compressed textures varies with the texture size as can be seen in table 5.2. It would theoretically be possible to load about 15 objects with 50,000 triangles together with texture and normal map of size $1024 \times 512$ within a frame at 24 FPS if one assumes that the time to load a compressed normal map is the same as loading a compressed texture.

![Table 5.2: Time to load compressed textures.](image)

5.1.2 Stationary computer test

The stationary computer test was done on an Intel® Core2 Quad processor Q9400 with 4 GB main memory and a Geforce 9500 GT graphics card. A flight path at 15,000 m/frame for 1,442 frames over the Himalayas was chosen since it is the most densely triangulated area. When this test was done a bug was present in the application which caused random objects to be prefetched. The computer was not available after the discovery of the bug but the performance measurements for cached performance and no prefetch are still shown here to give information about how the application performs on a better computer. The graphics card caps the FPS at 60 which means that the performance measurements of the FPS and triangles/s could have been better. Again the minimum FPS is lower when the data is not cached but the visual impression on the stationary computer was much better. It was even difficult to see the drop of the FPS when no prefetching was done.

![Table 5.3: Stationary computer performance analysis](image)

5.1.3 Visual results and artifacts

The atmosphere adds a lot of realism and manages all from ground to out of space views. However visual artifacts can arise probably caused by lack of precision in the Z-buffer since the distance between the atmosphere and the...
earth is rather small. Two different view points and an example the atmosphere artifact can be seen in figure 5.1.

Figure 5.1: a) Sunset over the Caribbean b) Earth from space c) Artifacts from atmosphere.

The ribbons added for covering gaps are barely noticeable, see figure 5.2.

Figure 5.2: a) Without ribbons a gap appear b) Ribbons cover the gaps.

Artifacts between sections can arise where sections of different detail levels meet and appear as detailed terrain on one side and blurred on the other side. The blur is caused by downsampled texture, normal map and geometry of lower detail. An example of the artifact can be seen in figure 5.3.

Figure 5.3: Artifacts between different levels of detail.

For close ups even more data is needed to be able to see for example cities and other important landmarks.
the space and a bit from the surface the data is sufficient. Finally the normal maps add a lot of detail as can be seen in figure 5.4.

Figure 5.4: Grand Canyon: a) Without using normal maps b) Using normal maps c) Normal maps only.
Chapter 6

Discussion

There was no time left at the end of the thesis to perform a test with other methods on the same computer. It is therefore difficult to compare the performance of the rendering algorithm with others since they are not done on the same hardware and further tests are needed so that the render process is not capped by the graphics card. Asirvatham and Hoppe [16] report that they render 60 M triangles/sec on a 2.4 GHz Pentium 4 with 1 GB RAM and an NVIDIA GeForce 6800 GT graphics card which could be faster than the method proposed in this thesis. Their method does not make any decimation on the terrain which should be taken into consideration since a higher quality of the terrain could be visualized with the same amount of triangles when the data has been decimated. If triangle strips would be used and reordered for better cache locality the performance of this thesis work would increase without changing the algorithm. A comparison between only prefetching to the main memory and prefetching to both main and GPU memory could be made and the result show that the performance is better when also prefetching to the GPU.

On the down side the pre-processing time is an issue and you want to be sure that everything is correctly initialized before setting computers to work for days or even weeks depending on the size of your data. It is not necessary to divide the large height map although it can be useful if you want to process the data on different computers. The prefetch frustum performs reasonable when the view does not change to quickly and too many new objects become visible.

6.1 Conclusion

This thesis has presented methods for pre-processing out-of-core data to build a hierarchical data structure suitable for rendering data of large spatial difference. Textures as well as normal maps can be included and compressed for better performance. Three different methods for dealing with gaps were implemented of which the skirt approach was found to be inadequate for rendering projected data. An improved scheme for prefetching, compared to the one in [9], was introduced which enables prefetching to both main and GPU memory. At runtime the application uses the CPU only for culling, rendering, selecting ribbons and loading data, no processing of the data is necessary. Draw calls are reduced by using the hierarchical data structure together with the visibility selection which further reduces the CPU load. An atmosphere entirely computed on the GPU was added outside the original plan. A final demonstration was created which first pre-processed 18 GB of original data which would have consisted of 7.3 Billion triangles and then rendered it in real time.

6.2 Future work

There are several areas to improve the work upon. The most important related to this thesis would be to incorporate general 3D objects in the pipeline. Other areas include the file structure, improving the prefetch strategy and further compression of the data. More detail about each area is given in the following sections.

6.2.1 Support for general 3D scenes

Since the generated data is 3D objects it would be possible to edit those to include overhangs or even merge them with other 3D objects making them part of the terrain. To avoid visible pops between different LODs a way to interpolate between the representations is needed. It would be even more interesting to handle general 3D scenes.
by partitioning the hierarchical structure based on for example the number of triangles within the scene. More work would then be needed to ensure that no gaps appear.

### 6.2.2 Improve file structure and handling

Currently there are many folders and files used by the application. The scene graph description and data should be moved to one or two files. This would increase the performance at runtime since CPU power is consumed when opening and closing files. Advantage could be taken of the spatial coherency in the data so that the hard drive improves access time. Performance could be increased and storage requirements decreased if triangle strips is used instead of indexed triangles.

### 6.2.3 Compression

Both the input data and the generated data can require much storage space. The pre-processing algorithms could be extended to work with compressed height maps which would decrease the intermediate storage requirements. The output data could either be compressed as an intermediate format and then decompressed before render time or handled compressed all the way at the expense of performance.

### 6.2.4 Optimize the prefetch algorithm

More accurate prefetches could be performed if the movement of the camera is be predicted. Performance could also be increased by taking the coherency between frames is into account. Given a camera speed it is perhaps not possible to reach certain areas of the scene which could be excluded when culling the objects.
References


