VISUALIZATION OF LARGE 3D SCENES IN REAL-TIME
WITH AUTOMATIC MESH OPTIMIZATION

by
Haraldur Órn Haraldsson

Master’s thesis for Advanced Computer Graphics at
Linköping University, Campus Norrköping - 20.march 2009
The search for solutions to visualize large 3D virtual scenes in real-time on today's hardware is an ongoing process. The polygonal models are growing in size and complexity, and proving too large and complex to render on current consumer hardware. Mesh optimization algorithms can produce low resolution models that can be used in level-of-detail (LOD) methods to accelerate the framerate and add greater interactivity to a scene, whereas a scene with high resolution models would not be able to render at an interactive framerate.

The preprocess of simplifying models is an efficient method but it requires that, most often, many models need to be simplified into multiple levels-of-detail, where every model needs to be visually validated and tested manually. This can prove to be a bit tedious work and quite time consuming. There exist tools for mesh optimization, with graphical user interfaces, that allow for simplifying a model with a viewer to validate the model, but they only deal with single objects at a time, not allowing for producing multiple simplifications of the same model for the purpose of using in level-of-detail scenes.

This thesis presents a solution that tries to make this preprocessing stage more efficient by presenting an application that allows for creating multiple LODs, a library if you will, using a mesh optimization tool from Donya Labs. Furthermore the application has a viewer to visualize not only single models, but also a large scene with level-of-detail implemented, so to be able to validate the simplified models in an actual LOD scene, which should enhance the workflow considerably.
# Contents

1 **Introduction** .................................................. 6  
2 **Background** .................................................... 8  
   2.1. 3D graphics rendering............................................... 8  
      2.1.1. Real-time rendering.............................................. 10  
   2.2. GUI design and usability......................................... 10  
   2.3. Simplification methods............................................ 12  
   2.4. Level-Of-Detail..................................................... 13  
3 **Tools** ............................................................ 16  
   3.1. Graphical User Interface.......................................... 16  
   3.2. Viewer....................................................................... 16  
   3.3. Mesh Simplification................................................ 18  
      3.3.1. Distance bound reduction.................................... 18  
4 **Implementation** ............................................... 21  
   4.1. Application.......................................................... 21  
      4.1.1. GUI Design......................................................... 21  
      4.1.2. WxWidgets & OpenSceneGraph toolkits.................. 23  
      4.1.3. Usage.............................................................. 24  
   4.2. LOD Scene Implementation........................................ 24  
      4.2.1. Static object....................................................... 26  
      4.2.2. LOD implementation........................................... 29  
      4.2.2.1. Distance bound optimization method.................... 29  
      4.2.2.2. LOD models.................................................... 30  
      4.2.3. Scene Implementation......................................... 33  
5 **Results** .......................................................... 35  
   5.1. GUI application..................................................... 35  
   5.2. LOD scene implementation....................................... 36  
      5.2.1. Object comparison............................................. 36  
      5.2.2. Scene comparison.............................................. 37  
6 **Conclusion** .................................................... 39  
   6.1. Future work......................................................... 39  

**Bibliography** .................................................. 40  

**A Program Code** ................................................. 42  
   A.1 WxWidgets and OSG code.......................................... 42  
   A.2 Scene implementation code...................................... 43  

**B Screenshots** .................................................. 47  
   B.1 Original objects..................................................... 47  
   B.2 LOD1 scene.......................................................... 48  
   B.3 LOD scene............................................................ 49
List of Figures

1.1. A scene consisting of a mass of pipes, the whole model consists of 12.731.154 polygons and cannot be rendered at interactive frame rates. Image taken from [5]..................6
2.1. On the left a triangle (polygon) is defined with the vertices (v0,v1,v2), and on the right a mesh consisting of multiple triangles.................................................................9
2.2. The viewing frustum........................................................................................................9
2.3. Culling, three views through a window. a) Building is off-screen. b) Building is on-screen, but occluded by the wall. c) Building is on-screen, and visible through the window. Image taken from [18]................................................................................10
2.4. Mesh simplification of the Stanford Bunny. Triangle count from left to right: 69.451 (original), 2.502, 251 and 76...................................................................................13
2.5. The stanford bunny visualized at different levels-of-detail. When further away, coarser models are visualized. ............................................................................14
3.1. A scene consisting of terrain, a truck and a cow. The top node can be the root, or a group, with the three child nodes. Image taken from [3].........................................................17
3.2. OSG graphical display. Time spent in the event and update traversals is so insignificant that it doesn’t appear in the display. The cull and draw traversals, at 5.68 and 3.42 milliseconds respectively, display clearly as cyan and dark yellow bars spanning part of a single frame. The final brown bar indicates that the GPU takes 5.38 milliseconds to process the rendering commands. Taken from [3]..................................................18
3.3. The distance bound reduction feature..............................................................................20
4.1. Two sketched ideas for the application, the right GUI was ultimately chosen to be implemented..........................................................22
4.2. The implemented GUI design.........................................................................................23
4.3. An overview of the models in the scene, here the original objects are depicted (thus the low frame rate).................................................................26
4.4. Ground model.................................................................................................................28
4.5. Crane model...................................................................................................................28
4.6. Container model............................................................................................................28
4.7. Helicopter model...........................................................................................................28
4.8. Houses models...............................................................................................................28
4.9. People models................................................................................................................29
4.10. Original model, was not simplified................................................................................29
4.11. LOD1 airport model.....................................................................................................31
4.12. LOD4 airport model.....................................................................................................31
4.13. LOD1 factory model.....................................................................................................31
4.14. LOD4 factory model.....................................................................................................31
4.15. LOD1 Tower and building model..................................................................................32
4.16. LOD4 Tower and building model..................................................................................32
4.17. LOD4 factory model.....................................................................................................32
4.18. LOD1 factory model.....................................................................................................32
5.1. Object comparison in the original scene........................................................................37
5.2. Frame rate comparison of different scenes....................................................................37
B.1. A screenshot of the original objects, framerate at only 6.62........................................47
B.2. A screenshot of the original objects, different angle, framerate at 5,19..........................47
B.3. A screenshot of the original Airport model, framerate at 10,67.............................................48
B.4. A screenshot of the LOD1 scene, framerate at 19,89.................................................................48
B.5. A screenshot of the LOD scene, framerate at 31,71...............................................................49
B.6. A screenshot of the LOD scene with anti-aliasing enabled, framerate at 18,65.........................49
B.7. A screenshot of the AA LOD scene, different angle.................................................................50
Chapter 1

Introduction

The visualization of large 3D scenes in real-time is a large topic and an important application in the field of computer graphics. It applies to many different application areas, such as computer games, simulations, large-scale interactive visualizations and virtual reality. Real-time rendering allows users to interactively move around a scene and view it from all possible angles. With the possibility to view the scene in real-time, meaning at an interactive frame rate, it gives the user a good understanding of the data they are exploring.

Many methods exist to deal with real-time visualization, e.g. occlusion culling and image based algorithms. In this thesis level-of-detail (LOD) is implemented, a selection and management method, using mesh simplification.

The root of the problem is with high resolution models that are constructed using 3D software or high resolution scanners. While it works for off-line rendering they are not optimized for real-time usage and need a lot of rendering power to be visualized. Better and larger models are important for these application areas as the designer wants to produce the best looking and detailed environment possible. Specifically the designer wants only to make one high-resolution model, and not to manually model multi-resolution instances of the same object. Specific graphics systems have been constructed to accelerate the rendering process but still there are problems with rendering the scenes at an interactive frame rate, even with the modern high-end hardware available today, an example can be seen in Figure 1.1.

![Figure 1.1](image.png)

Figure 1.1. A scene consisting of a mass of pipes, the whole model consists of 12,731,154 polygons and cannot be rendered at interactive frame rates. Image taken from [5].
This requires algorithm techniques that accelerate the rendering process, by substituting the models with simplified versions in the scene during real-time and thusly rendering at an interactive frame rate. The target is basically to achieve improved interactivity without the loss of image quality.

The method of using LODs is very common and quite efficient. As mentioned before, to be able to perform this technique a lot of preprocessing is needed as the high resolution models need to be simplified to different levels-of-detail before they are implemented in the scene. These separate operations are to simplify the models individually and then to view them to see if they are visually acceptable according to the level-of-detail needed. After this preprocessing the models are ready to be implemented into the scene as LODs. But when implemented in the scene other issues can arise, the designer can spot “popping” effects (artifacts that differ in the objects) when switching or that the models are simplified too much or too little and the designer would have to start over the process. This is not very efficient considering that it can be quite time consuming to view each object individually or to render the large 3D scene.

While the length of this process can be vastly different according to what type of environment the designer is working with, it would be more efficient to be able to combine these operations into the preprocessing stage before actually having to implement the LODs into the main scene, i.e. to be able to visualize the constructed LODs in some kind of test scene before.

This thesis will discuss the techniques needed for mesh simplification and level-of-detail generation and ultimately how to optimize the process of visualizing a large 3D scene by presenting an application, using Open-Source software and a mesh optimization tool from Donya Labs, for working with this type of data. By presenting an intuitive application, then hopefully the issues regarding these preprocessing operations can be addressed in such a way that the process of visualizing a large 3D scene will become more automatic and efficient for the user.

The following chapter will present a background into graphical user interfaces, mesh simplification methods and level-of-detail techniques.

Chapter 3 will present the tools that were chosen for implementing the application. And Chapter 4 will detail the implementation of the application and a test scene, details regarding the features the application offers will be looked at and how the test scene is constructed.

Finally, in chapter 5 the results of the application and scene tests are presented.
Chapter 2

Background

To render a large 3D scene in real-time is not a trivial task. To achieve an interactive frame rate with high resolution models is quite a complex task which requires a lot of preprocessing. Many methods have been presented to solve this, and all these techniques provide some form of optimization for the scene, each having their own benefits and drawbacks. These techniques depend on the type of scene that is to be visualized but this chapter aims to introduce an overview of the topic to the reader, focusing on the methods that are relevant to this thesis.

A short overview of fundamental 3D rendering will be presented for the reader to better understand the methods presented later on, and as the focus of this thesis is on the process of visualizing a scene by the aid of an interface, the fundamentals of graphical user interface design and usability will also be discussed.

2.1. 3D graphics rendering

To get a better understanding of the techniques involved in the process of visualizing a large 3D scene it is good to have a basic understanding of the rendering process.

The process of creating 3D computer graphics can be sequentially divided into three basic phases, 3D modeling, layout/animation and 3D rendering.

Rendering, i.e. drawing, is the final process of the graphics pipeline, which generates an image from the models in the scene, it involves 3D projection that allows a three-dimensional image to be viewed in two dimensions (like on a monitor).

A scene is built with triangles that is defined by three points, referred to as vertices. 3D models are built by connecting multiple triangles, which is called a mesh.

![Figure 2.1](image.png)

*Figure 2.1.* On the left a triangle (polygon) is defined with the vertices \((v0,v1,v2)\), and on the right a mesh consisting of multiple triangles.
When the model and the rest of the content of the scene have been defined, theADING FRUSTUM is defined. The viewing frustum defines the part of the 3D scene that is rendered for the screen, i.e. the field of view for the camera. The viewing frustum is cut perpendicular to the viewing direction into planes called the near plane and the far plane. Objects in the scene that are closer to the camera than the near plane or further away than the far plane are not rendered.

The viewing frustum is then a geometric representation of the volume visible to the camera, and everything outside of this frustum is discarded, a process called culling. When objects lie on the boundary of the viewing frustum they are cut into pieces along the boundary, called clipping, and the pieces outside the boundary are discarded (culled).

Culling, is a process that defines that if objects that are invisible in the scene, do not have to be fetched, transformed or rendered at all. Which also includes objects that are situated behind another opaque object.

For more details regarding 3D rendering, the 3D pipeline and other related material, please see [7] and/or [8], which give a thorough description on general 3D computer graphics, and OpenGL, respectively.

2.1.1. Real-time rendering

The aim of real-time rendering is performance, i.e. to show as much information as possible at frame rates of approximately 20 to 120 frames per second. Which basically means that the goal is speed rather than photo-realism. So there are a lot of “hacks” involved in these methods as the final...
output is not necessarily that of the “real-world” but just close enough for the human eye to not
detect any irregularities. With that in mind an understanding of visual perception is also valuable to
the rendering of the scene as human perception has a restricted range.

Real-time rendering is a big topic and a lot of methods exist about how to optimize the rendering
but that is beyond the scope of this thesis so below is an overview of some of those methods.

- Acceleration – Software and hardware acceleration, e.g. code optimization, parallelization,
dedicated hardware.
- Fast algorithms – Fast lighting and shadows, e.g. global illumination and ray tracing are
very slow, consider other solutions, e.g. shadow texture.
- Preprocessing – Prepare and organize, e.g. using hierarchical scenes.
- Culling – Remove objects that don’t affect the scene.
- Level-of-detail – Use as low resolution as possible and select a fitting resolution at render-
time.
- Textures – Replace geometrical details with textures.

For more details regarding real-time rendering please see [9] and it’s accompanied website.

2.2. GUI design and usability

A graphical user interface (GUI) is a type of interface that allows users to interact with electronic
devices, such as computers, MP3 players and mobile phones. A GUI presents graphical icons and
visual indicators for the users to navigate with, as opposed to text based interfaces, and the user
performs action with direct manipulation of the graphical elements in the GUI.

One well known GUI to emerge was Microsoft’s Windows in 1985, developed as an add-on to
MS-DOS, which later developed into Windows 95, which no longer needed DOS to be installed
beforehand.

GUI design focuses on making the user’s interaction with the interface as simple and efficient as
possible. Graphic design focuses on making the application physically attractive, but without
compromising its usability. Good interface design focuses on the user and not the systems itself, as
it’s goal is to make the user comfortable interacting with the application.

One of the more current popular design development employ application specific touchscreen
GUIs, e.g. mobile phones and handheld gaming devices. Touchscreen devices offer very intuitive
and easy navigation for users, making e.g. buttons surplus to requirement.

Research into user interface design is a big topic and would fill many volumes, so the reader is
urged to look at further references for more details. Shneiderman [10] is a good textbook on user
interface design, and Norman [11] gives a good overview of broader design issues. Also there are
plenty of online references that can be found for an overview on the topic, e.g. the ever reliable
Wikipedia.

Usability is the term that describes the ease with which users can employ an application in order to
achieve a particular goal. It also involves the methods of usability testing and research into the
applications efficiency. Basically it describes how well a product can be used for its intended
purpose by its target users with efficiency, effectiveness and satisfaction.

There is a lot of research material to be found in this area which cannot be covered in this thesis, but according to user interface guru Jakob Nielsen [12], there are ten general principles, or heuristics, that should be considered.

- Visibility of system status
- Match between system and the real world
- User control and freedom
- Consistency and standards
- Error prevention
- Recognition rather than recall
- Flexibility and efficiency of use
- Aesthetic and minimalist design
- Help users recognize, diagnose, and recover from errors
- Help and documentation

These principles are good guidelines for the designer to reference while starting the design process, but obviously further research and adaptation is needed for each individual user interface, as the design strongly depends on various different factors, e.g. content, type of application and the target users.

For further details, refer to Jakob Nielsens [12] website, where information regarding this topic is found, and links to further references.

2.3. Simplification methods

To accelerate the rendering is the most important factor in real-time rendering and various methods exist to do that. While visibility techniques focus on culling away portions of the scene that are not visible to the user, e.g. [15] and [16], simplification methods focus on generating multi-resolution representations of polygonal models [18].

Mesh simplification, or decimation, is a class of algorithms that transform a given polygonal mesh into another with fewer faces, edges and vertices, while trying to retain a good approximation of its original shape and appearance.

There exists many algorithms for mesh simplification and is a big topic in itself, where algorithms have different properties and vary in quality, efficiency and operations.

One of the earlies algorithms was introduced by Schroeder, Zarge and Lorensen [13] in 1992 and was a algorithm that uses vertex removal. Hoppe et al. [20] also introduced in 1993 a mesh optimization algorithm by using point sampling methods, it produces good results but at a cost of being rather slow.

In 1997 Garland and Heckbert [14] introduced an algorithm that uses the quadric error distance measure and the edge collapse operator, and is a fast algorithm and produces high quality results.
Garland and Heckbert extended this algorithm in 1998 by simplifying models with surface attributes such as normals, colors, and texture coordinates[19].

Figure 2.4. Mesh simplification of the Stanford Bunny. Triangle count from left to right: 69,451 (original), 2,502, 251 and 76.

For more thorough overview of mesh simplification algorithms see [17].

The methods mentioned before depend on the CPU architecture but with recent methods of interactive visualization using the GPU to render objects, there is a possibility to accelerate the optimization process by an order of magnitude [21].

There also exists some extreme simplification approaches. Décoret et al. [22] introduces an billboard cloud technique, which represents an object as several textured rectangles in order to dramatically reduce its geometric complexity, where the models are simplified onto a set of planes with texture and transparency maps. And He et al. [23] uses a signal processing approach, the algorithm smoothes sharp edges or creases by eliminating high frequency detail of an object.

2.4. Level-Of-Detail

Level-of-detail is a method for decreasing the complexity of visualizing a 3D model in real-time. It is an important tool to maintain interactivity in a scene. By effectively switching to simplified versions of a model as it moves away from the viewer it decreases the workload on the graphics pipeline.

There exists a few other methods that complements the level-of-detail tool, e.g. parallel rendering, occlusion culling and image-based rendering.

Basically, high resolution geometric datasets can be too complex to render at interactive framerates; so the models can be simplified, as described before in Chapter 2.3., to visualize small or distant objects.
Early techniques refer to James Clark [6] in 1976, he wrote an article which presented a generic approach on level-of-detail, which is still used today without modification in most 3D graphics applications known as discrete LODs. Also Funkhouser and Séquin [24] introduced in 1993 a predictive system which targets framerates and rely on precomputed calculations of system performance.

Here is an overview of different LOD methods:

- **Discrete LOD**—Generate a handful of LODs for each object
- **Continuous LOD**—Generate data structure for each object from which a spectrum of detail can be extracted
- **View-dependent LOD**—Generate data structure from which an LOD specialized to the current view parameters can be generated on the fly. One object may span multiple levels of detail

*Discrete LOD* is the method chosen for this thesis and it involves creating multiple versions of the original object, each with different resolutions, as a preprocess. At run-time, a LOD model will be picked according to some criterion given by the system, e.g. distance from the viewer or size on the screen. The run-time rendering then basically only needs to pick a LOD according to the criterion, and since the objects that are in distance (or are small) are low resolution the rendering speed will be increased [6, 26].

While the discrete method created individual levels-of-detail in a preprocess, the *Continuous LOD* creates a data structure from which a desired level-of-detail can be extracted at run-time. A LOD is specified exactly, not chosen from a few pre-created models, which frees up polygons because the objects don’t use more polygons than necessary. This allows for the detail to be gradually and incrementally adjusted, and thus reducing the possibility of visual “popping” [25].

*View-dependent LOD* extends on the continuous LOD method by dynamically selecting the most appropriate level-of-detail for the current view, which means that a single object can span multiple levels of simplification. This enables even better overall fidelity, because the method allocates polygons where they are most needed, within objects as well as among objects. A good example of this is terrains, nearby portion of the terrain can be shown in higher resolution than the distant

---

*Figure 2.5.* The stanford bunny visualized at different levels-of-detail. When further away, coarser models are visualized.
An addition to the LOD methods mentioned before there is also *Hierarchical LOD* [4], a method that aggregates objects into assemblies with their own LODs, i.e. the method groups different objects together for higher efficiency and takes advantage of proximity considerations.

Even though that there are obvious advantages in the continuous and view-dependent methods, the discrete method remains the most common approach today. This has to do with that the discrete method is simpler and easier to implement and works well with current graphics hardware.

The continuous and view-dependent methods still impose a cost in processing and memory to be fully accepted, though the continuous has fared better, the view-dependent method is only used when absolutely necessary, such as terrain rendering or scientific visualization.
Chapter 3
Tools

Whether to use toolkits or do things from scratch is always a bit tricky as the developer always wants to have as much control as possible. For this specific application it was extremely helpful to have toolkits available as there was emphasis on the overall result, rather than specific methods. Which means that the idea was to make an application that combines different methods as means to an end, i.e. a GUI, a 3D viewer and a simplification tool in one application.

For this purpose the open source and well established API’s WxWidgets and OpenSceneGraph were chosen, because of their adaptability, good documentation and community support.

3.1. Graphical User Interface

For the GUI implementation the WxWidgets toolkit was chosen, which is a toolkit for creating graphical user interfaces that uses the C++ language and works for cross platform applications, which makes it very portable. It’s a framework, which means that it does a lot of the work and provides default application behavior. Besides being cross-platform, i.e. working on Windows, Mac and Linux, it also adapts the native “look and feel” on each respective platform. Which means that when running on, e.g. Windows, it looks like a Windows application, using their native widgets, and doing the same on other platforms.

There also exist RAD (rapid application development) tools for easier development of WxWidgets, which allows for visual development of WxWidgets interfaces, thusly allowing for code generation and plugins for extending widgets.

For more information on the WxWidgets API, please refer to [1, 2].

3.2. Viewer

For handling the 3D graphics in the application and connect to WxWidgets, the OpenSceneGraph toolkit was chosen. It is an high performance, cross-platform, 3D graphics toolkit that uses scene graph technology. It builds on top of the lower-level OpenGL hardware abstraction layer, providing higher-level rendering, input/output and organizational functionality to a 3D application.

OSG provides high performance due to support for many different accelerative methods, such as view frustum and occlusion culling, level-of-detail, multi-threading and state sorting.

The occlusion culling is automatically incorporated in the toolkit, while the level-of-detail is the method that is implemented.

The OSG is using a scene graph technology that is a collection of nodes in a graph or tree
structure. A node can have many children, but often only one parent, and if there is change to the parent it will affect all child nodes. A nice feature is the ability to group multiple objects as a new single object, which then can be manipulated easily, thusly the OSG makes it relatively easy to handle objects in a scene and a provides a basis for doing level-of-detail using these principles.

![Diagram of a scene consisting of terrain, a truck and a cow. The top node can be the root, or a group, with the three child nodes. Image taken from [3].]

OSG handles the level-of-detail, i.e. the switching between the LOD models according to specific parameters. There are two criteria available in the OSG, distance and pixel size on screen. Using these criteria work similar, the developer will set a range for each LOD model and the program will switch LODs accordingly. The difference is that with distance the developer will set the range according to how far away a object should be from the viewer before switching, while the size on screen method, the developer will set the range according to the size that the object should be when switching.

The pixel size on screen method was chosen so to complement the simplification method presented in the next section, which also depends on the size of the object.

Then the simplification and the level-of-detail method becomes very much more controlled and not only a matter of the developers perspective.

By incorporating a specific class in the code the OSG has an onscreen statistics display that gathers and displays rendering performance information. This is very useful for identifying where the rendering process is spending its time and for framerate measures.

The display mode is as follows.

- Framerate – osgviewer displays the number of frames rendered per second.
- ThreadingModel – osg has two threading models, single and multi threaded.
- Traversal time – osgviewer displays the amount of time spent in each of the event, update, cull and draw traversals, which includes a graphical display.
- GPU – time spent to process the rendering commands (in milliseconds), as measured by OpenGL.
For more information on the OpenSceneGraph API, please refer to [3, 28].

3.3. Mesh Simplification

The Simplygon tool from Donya Labs is a 3D optimization tool, which includes high-quality polygon reduction and polygon-repair features. It automatically reduces 3D models and can create highly optimized level-of-detail objects. By incorporating the Simplygon SDK in the application it gives the possibility of creating LOD objects quickly and efficiently.

Simplygon optimizes 3D models by removing, or collapsing, triangles in a model. There are basically two methods for 3D reduction available, by manually setting the triangle count wanted or by using the absolute distance feature.

The basic triangle count reduction is done manually, i.e. the artist has to manually decide the best triangle count regarding either a maximum size of the mesh or according to the visual detail needed. This often calls for considerable testing with the triangle count to get the best representation of the model. For the application discussed in this thesis the absolute distance feature is implemented.

3.3.1. Distance bound reduction

The distance bound feature generates a simplified mesh object with a guaranteed geometric distance bound, which means that the points on the simplified mesh are guaranteed to be within a user specified distance from the original surface.
The function can be used to automatically create LODs where the accuracy in absolute distance is set by the maximum allowed pixel size that the object should have on the screen. Used with the LOD switching method, pixel size on screen mentioned earlier, allows for auto creation of LODs where minimal “popping” effects occur because of the preservation of critical surface features within pixel accuracy.

Figure 3.3. The distance bound reduction feature.
Chapter 4
Implementation

In the following chapter the implementation of the application and the LOD scene are presented. As presented in the previous chapter the WxWidgets and OpenSceneGraph (OSG) toolkits were used for implementing the interface and to handle the 3D graphics.

During the implementation stage there were four critical points to consider, this included:

- Designing the Graphical User Interface.
- Implementing a combination of the OpenSceneGraph and the wxWidgets toolkits in one application, as there was not much reference material found.
- Implementing the mesh optimization algorithm in the application.
- Implementing a LOD scene to visualize in real-time.

4.1. Application

The initial idea was to construct an application that the user can interact and manipulate data within one easy to use Graphical User Interface (GUI), i.e. the user should be able to view individual models, simplify and visualize a LOD scene in the application. This should add much desired efficiency to the users workflow when working with mesh simplification and specifically LOD’s.

4.1.1. GUI Design

The main thing to consider when designing the GUI is the placement of the viewer window and the buttons. There are basically four different possibilities to consider based on standard GUI design when it comes to this type of setup, i.e. a large viewing area and a group of buttons. Basically it revolves around where to place the grouped buttons when considering the viewer area, i.e. above, below, to the right or to the left.

After some consideration the options were cut down to two distinct possibilities, above or to the left of the viewer area, as is typically seen in software applications and hyperlink setups in websites.

Ultimately the choice was made to place the buttons to the left of the viewer as it was possible to make the buttons and their respective texts bigger, and also it made it possible to add a small window that serves the purpose of a list that shows the names of the objects loaded in the scene.
Implementation

Obviously when the application is further developed and more options are added to the GUI these choices will be revised and possibly changed accordingly. But it is possible, and not difficult, to add submenus and scrollbars to further enhance the options in the interface if needed.

![Two sketched ideas for the application, the right GUI was ultimately chosen to be implemented](image1)

**Figure 4.1.** Two sketched ideas for the application, the right GUI was ultimately chosen to be implemented

The implemented GUI design

![The implemented GUI design](image2)

**Figure 4.2.** The implemented GUI design

4.1.2. WxWidgets & OpenSceneGraph toolkits

To implement the chosen design for the interface involves a basic understanding of the wxWidgets toolkit. After familiarizing with the toolkit the implementation of a simple GUI was fairly straightforward with the aid of the references and examples available, although that the basic event calls in WxWidgets, e.g. when pressing a button and moving the mouse, needed a considerable amount of testing to make it work because of the methods used in the OpenSceneGraph, but will not be described in detail as it is outside of the objective of this thesis.
The aspect of the program that needed more attention was the implementation of the OpenGL window in the application, which ultimately the OSG connects to, this is done via the class called \textit{wxGLCanvas}, which is the class for displaying OpenGL graphics in WxWidgets.

To be able to connect an OSG viewer to the \textit{wxGLCanvas} window a new rendering context is made by doing a \textit{GraphicsContext} that directs all commands from the OSG to the \textit{wxGLCanvas} window.

Once the rendering context from the OSG to the \textit{wxGLCanvas} is set, all OSG based commands will be directed there accordingly. The order of commands will then effectively be the following:

- Setup a new \textit{wxGLCanvas} in the GUI (called \textit{GLViewer})
- Initialize the rendering context in the \textit{GLViewer} constructor, called \textit{GraphicsWindowWX} (where the \textit{SetCurrent} and \textit{SwapBuffers} are called)
- Initialize the viewer supplied by OSG in the current rendering context
- Set the root of the scene graph for the viewer

Now with the rendering context initialized and the OSG viewer set to the window, the programming becomes about standard OSG methods, while inside of the \textit{GLViewer} class.

The functions for each of the buttons in the GUI are then written inside the \textit{GLViewer} class and called from the GUI class using events (\textit{wxCommandEvent}).

- **Original** – opens a dialog to choose a single model to load (\textit{wxFileDialog}), then stores the path and calls the function \textit{LoadModel} located in the \textit{GLViewer} class, which adds the model as a node and updates the scene graph (\textit{root->addChild}).
- **Remove Object** – calls the \textit{ClearModel} function in the \textit{GLViewer} class and removes the added node from the scene graph. If there are more than one it removes all nodes.
- **LOD Scene** – calls the \textit{LodScene} function in the \textit{GLViewer} class and loads the LOD scene.
- **Simplygon** – opens a dialog to choose a single model to simplify, then stores the path and calls the function \textit{LoadObjFile} in the \textit{Simplygon SDK}, which then simplifies the object, according to given parameters (distance bound) and returns the multiple LODs.
- **Exit** – calls \textit{wxWindow::Destroy} function that destroys (exits/closes) the GUI safely.

The field \textit{Object List} is intended for listing the names of the objects currently loaded into the scene, but was not implemented at the time of testing and when taking the screenshots.

Please refer to Appendix A for the code from the program.

4.1.3. Usage

One of the main advantages of the GUI is the possibility of performing simplification and then instantly viewing the 3D models within one application.

At this time the application only has the rudimentary options available. The user chooses a model and it will be simplified into different levels-of-detail and saved to the hard drive, either in the Wavefront .obj\textsuperscript{1} or the Collada .dae\textsuperscript{2} format, according to which type of model is chosen. The

\textsuperscript{1} OBJ is a simple 3D geometry data format, only representing the position of each vertex, texture coordinates associated with a vertex, the normal at each vertex and the faces that make each polygon.

\textsuperscript{2} DAE is an open standard XML schema file format, allowing to fully preserve asset data and meta data. See \textit{http://www.khronos.org/collada/} for more information on the Collada format.
Implementation

parameters (for each LOD) in the simplification process should be available in the GUI but are at this point decided beforehand in the code.

At any point, including after the simplification process, the user can add and remove individual models in the scene by the load model and remove buttons in the application.

More importantly the user can also view a full LOD scene to better evaluate the simplifications by pressing the LOD Scene button, though the scene is currently set up beforehand in the code.

The valuable part of the application is that even though traditional simplification software do have the viewer characteristics, they do lack the possibility of simplifying a model into multiple levels-of-detail (a batch if you will) to build a library of LODs. Furthermore the option to visualize a large scene in real-time, to better evaluate the multiple simplified models together with level-of-detail, gives the user a useful way of evaluating the degree of simplification needed for each model in an intuitive and dynamic way.

4.2. LOD Scene Implementation

The scene is setup using five objects, where one is a static object built in 3ds Max and includes several models of its own. The other four are the high resolution models used for the level-of-detail (LOD).

The models used in this thesis are in the Collada .dae format, which was chosen mainly because Simplygon supports simplification of Collada models while preserving textures, which allows for visualizing a richer scene, though most of the models had to be converted to Collada format from the Wavefront .obj format first.

The idea was to visualize a scene that had some aesthetic look to it, i.e. not to randomly put models into a scene or just multiple instances of the same model. So to further add to the visualization of the scene the static object was constructed. But several things contributed to making the visualization of the scene complicated, e.g. model availability, format issues and model support, so a lot of compromise was needed to make a scene that was in the end somewhat satisfying.

The high resolution models were simplified to four different levels-of-detail, each with different triangle count according to the distance bound feature as described earlier. Details regarding the LOD models is described in the LOD Objects section later on.

All the images of the individual models are saved from renderings in 3ds Max.

Here is an overview of the objects in the scene:

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static object</td>
<td>The ground object, including multiple static models.</td>
</tr>
<tr>
<td>Airport</td>
<td>The airport building in the middle of the scene.</td>
</tr>
<tr>
<td>Factory (Left)</td>
<td>The factory on the left side of the airport.</td>
</tr>
<tr>
<td>Airport Tower and building</td>
<td>The airport tower and building on the left side of the airport.</td>
</tr>
<tr>
<td>Second Factory (Right)</td>
<td>The factory on the right side of the airport.</td>
</tr>
</tbody>
</table>

Table 4.1. The objects contained in the LOD scene
4.2.1. Static object

The static object, which includes the ground, is placed by default in the middle of the scene, i.e. it’s at the coordinates (0,0,0).

Basically the static object is used as a reference to place the LOD models. It is actually made up of multiple models that were added in 3ds Max to fill up the scene, to complement the LOD objects that are added later in the code. The static models were simplified as much as possible without the loss of detail.

The models in the static object are listed below with the triangle count of the original model and the model after simplification. The images presented are of the simplified models, except where noted, as there is almost no visible difference from the original.

<table>
<thead>
<tr>
<th>Object(s)</th>
<th>Original Triangle count</th>
<th>Triangle count after simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground</td>
<td>185.785</td>
<td>26.015</td>
</tr>
<tr>
<td>Crane</td>
<td>226.890</td>
<td>99.831</td>
</tr>
<tr>
<td>Helicopter</td>
<td>88.950</td>
<td>13.342</td>
</tr>
<tr>
<td>Containers</td>
<td>280.416</td>
<td>78.504</td>
</tr>
<tr>
<td>Houses</td>
<td>1.094.500</td>
<td>737.500</td>
</tr>
<tr>
<td>People</td>
<td>232.341</td>
<td>151.506</td>
</tr>
<tr>
<td>Airplane</td>
<td>3.592</td>
<td>3.592</td>
</tr>
<tr>
<td><strong>Total count</strong></td>
<td><strong>2.112.474</strong></td>
<td><strong>1.110.290</strong></td>
</tr>
</tbody>
</table>

Table 4.2. The models contained in the static object and their triangle count

Figure 4.3. An overview of the models in the scene, here the original objects are depicted (thus the low frame rate).
Figure 4.4. Ground model

Figure 4.5. Crane model

Figure 4.7. Helicopter model.

Figure 4.6. Container model.

Figure 4.8. Houses models.
Implementation

The ground model was the first model chosen and then the others were placed on the ground to visualize the static scene, then the scene was exported as a single Collada .dae object to be loaded into the program with the LOD models.

4.2.2. LOD simplification

The four high resolution models are each simplified to four different levels-of-detail, from 1 to 4, and the first is the highest resolution (largest on screen) and the fourth is the lowest (smallest on screen). The Simplygon algorithm begins by simplifying the highest resolution first and then simplifies the next etc., and thusly the naming convention 1 to 4 is used.

The first LOD is simplified, because the original model is far too big for visualization purposes, but the optimization was done with minimal loss of visual detail when compared to the original. With the Donya labs algorithm, Simplygon, and it’s ability to keep textures intact this was possible to considerable extent.

It is possible to use more levels-of-detail than only four of course, but this was chosen to keep the program from having to load too many models into memory and was deemed suffice for smooth transitions between the LOD models.

4.2.2.1. Distance bound optimization method

To optimize the models using the distance bound function a series of calculations need to be done for each LOD before optimizing. The user sets a parameter for each LOD that defines the error bound that the simplified mesh can differ from the original mesh when optimized.

This parameter is determined by first calculating the diagonal size of the model, then dividing that with the resolution of the respective LOD that is going to be visualized on screen, i.e. how many pixels is the LOD model going to take on screen. This defines the size of the object per pixel. Then the final distance bound parameter is found by dividing the objects size per pixel with the user defined error margin, in this case only half a pixel.

Each LOD was calculated using this method but the resulting parameter was more often than not very small and thusly reducing the model very little, so the parameters were often tweaked to get

Figure 4.9. People models.
Figure 4.10. Original model, was not simplified.
the desired results.

In the following example though, using the Tower model, the resulting parameters after calculations were used without change for the optimization.

The Tower model’s size was measured in 3DStudio Max and was the following: length = 66.85, width = 210.33 and height = 55.
The diagonal calculation is found by the following formula \( \sqrt{l^2 + w^2 + h^2} \)
The diagonal of the Tower model is then -> \( \sqrt{(66.85^2 + 210.33^2 + 55^2)} = 227.45 \)

Then the size of the model per pixel is calculated. In the program we use four different levels-of-detail with the following ranges -> 0 to 60, 60 to 400, 400 to 1000 and 1000 to 500,000. The switching of the LODs happens at 1000, 400 and 60; the LOD 1 is divided by the upper range of 500,000, this done as the purpose was to also simplify the first LOD and not use the original model.

<table>
<thead>
<tr>
<th>LOD</th>
<th>Range</th>
<th>Calculation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD 1</td>
<td>1000 - 500,000</td>
<td>( \frac{227}{500.000} )</td>
<td>0.000454</td>
</tr>
<tr>
<td>LOD 2</td>
<td>400 – 1000</td>
<td>( \frac{227}{1000} )</td>
<td>0.227</td>
</tr>
<tr>
<td>LOD 3</td>
<td>60 – 400</td>
<td>( \frac{227}{400} )</td>
<td>0.5675</td>
</tr>
<tr>
<td>LOD 4</td>
<td>0 – 60</td>
<td>( \frac{227}{60} )</td>
<td>3.78</td>
</tr>
</tbody>
</table>

Finally, the error margin is defined as only half a pixel, so we divide with 2.

<table>
<thead>
<tr>
<th>LOD</th>
<th>Calculation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD 1</td>
<td>( \frac{0.000454}{2} )</td>
<td>0.000227</td>
</tr>
<tr>
<td>LOD 2</td>
<td>( \frac{0.227}{2} )</td>
<td>0.1135</td>
</tr>
<tr>
<td>LOD 3</td>
<td>( \frac{0.5675}{2} )</td>
<td>0.28375</td>
</tr>
<tr>
<td>LOD 4</td>
<td>( \frac{3.78}{2} )</td>
<td>1.9</td>
</tr>
</tbody>
</table>

These results give the final distance bound parameters to use in the Simplygon SDK, in the next section the results of the optimization and the triangle count can be seen for the Tower model.

4.2.2.2. LOD models

Below you can find detailed description of the four models and their levels-of-detail. All models were simplified into 4 different LODs according to the level-of-detail needed, using the distance bound method described earlier. Images of LOD 1 and 4 will be shown, as there is little to no visual difference in the other LODs when compared to LOD 1, while the LOD4 models are starting to show obvious decline in visual detail.

**Airport:**
The Airport LOD is placed in the middle of the scene, and the original model is 1,303,572 triangles.
### Implementation

<table>
<thead>
<tr>
<th>LOD</th>
<th>Triangle Count</th>
<th>Distance Bound Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>334.868</td>
<td>0.05</td>
</tr>
<tr>
<td>2.</td>
<td>233.481</td>
<td>0.1</td>
</tr>
<tr>
<td>3.</td>
<td>104.086</td>
<td>0.8</td>
</tr>
<tr>
<td>4.</td>
<td>86.350</td>
<td>2.0</td>
</tr>
<tr>
<td><strong>Total:</strong> 758.785</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.3.** Airport LODs

Factory on the left:
The Factory LOD is placed to the left of the airport, and the original model is 856.990 triangles.

<table>
<thead>
<tr>
<th>LOD</th>
<th>Triangle Count</th>
<th>Distance Bound Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>303.400</td>
<td>0.01</td>
</tr>
<tr>
<td>2.</td>
<td>150.769</td>
<td>0.1</td>
</tr>
<tr>
<td>3.</td>
<td>92.623</td>
<td>0.5</td>
</tr>
<tr>
<td>4.</td>
<td>73.068</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>Total:</strong> 619.860</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.4.** Factory on the left LODs
Implementation

Airport Tower and building:
The original Tower and building model is 490,596 triangles.

<table>
<thead>
<tr>
<th>LOD</th>
<th>Triangle Count</th>
<th>Distance Bound Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>451.944</td>
<td>0.000227</td>
</tr>
<tr>
<td>2.</td>
<td>91.943</td>
<td>0.1135</td>
</tr>
<tr>
<td>3.</td>
<td>56.886</td>
<td>0.28375</td>
</tr>
<tr>
<td>4.</td>
<td>33.767</td>
<td>1.9</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>634.540</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5. Airport and building LODs

Factory on the right:
The second Factory LOD is placed to the right of the airport, and the original model is 1,216,507 triangles.

<table>
<thead>
<tr>
<th>LOD</th>
<th>Triangle Count</th>
<th>Distance Bound Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>584.423</td>
<td>0.01</td>
</tr>
<tr>
<td>2.</td>
<td>279.942</td>
<td>0.06</td>
</tr>
<tr>
<td>3.</td>
<td>170.711</td>
<td>0.15</td>
</tr>
<tr>
<td>4.</td>
<td>67.513</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>1,102,589</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6. Factory on the right LODs
Using the distance bound feature keeps the visual appearance of the optimized models very close to the original, and during the scene testing there were no visible “popping” effects. There is though room for the models to be optimized considerably more for better efficiency and acceleration, but at a cost of visual detail.

4.2.3. Scene Implementation

The LOD scene is implemented in the code beforehand and the function is called when the user clicks the LOD Scene button in the GUI which promptly loads the scene. As presented in Chapter 2.4, the discrete LOD method is implemented for the scene and culling is automatically implemented within the OSG, i.e. models that are not in view are removed entirely.

The order of operations when the button is clicked is the following:

- First the static object is loaded and stored as a node and added to the scene graph (root) directly.
- The LOD objects are loaded, each into their own group consisting of four models.
- The LODs are created (and the switching parameters) using the range mode PIXEL_SIZE_ON_SCREEN, i.e. each LOD model is set to be visible within a given range according to the diagonal size on screen. When the size of the model surpasses it’s given range the switching occurs.
- The LODs are added to the scene graph using positioning in (x,y,z) coordinates, except for the airport LODs which are by default set to the center of the scene. The factory on the right are also rotated 90 degrees around the y-axis for better visualization results.
- The scene graph is optimized using the OSG utility Optimizer, which traverses the scene graph to improve efficiency.

For the code please see Appendix A.

The switching, as mentioned before, is based on the pixel size of the models on screen, and is controlled by the range specified in the code. The LOD’s in the code are ordered from the lowest resolution model first, as it ranges from zero size on screen, to the highest resolution model.

```cpp
// LOD 1 – Airport
osg::ref_ptr<osg::LOD> lod1 = new osg::LOD;
lod1->addChild(lod1one, 0.f, 60.0f);
lod1->addChild(lod1two, 60.0f, 400.0f);
lod1->addChild(lod1three, 400.0f, 1000.0f);
lod1->addChild(lod1four, 1000.0f, 900000.0f);
lod1->setRangeMode(osg::LOD::PIXEL_SIZE_ON_SCREEN);
```

Listing 4.1. The switching code for LOD 1, the airport model.

As seen in Listing 4.1. there are four LODs of the airport model, each with their own range, that depicts when to be visible on screen and when to switch. Note the convention in the code to name the lowest resolution LOD (range 0-60) lod1one, as opposed to the naming convention of the simplification tool, where LOD 4 is the lowest resolution model; i.e. lod1one = LOD4.
• **lod1one** – this is the lowest resolution model (LOD 4) that is visible only when it’s diagonal size on screen is between 0 and 60 pixels.

• **lod1two** – this is the second lowest resolution model (LOD 3) that is visible only when it’s diagonal size on screen is between 60 and 400 pixels.

• **lod1three** – this is the second highest resolution model (LOD 2) that is visible only when it’s diagonal size on screen is between 400 and 1000 pixels.

• **lod1four** – this is the highest resolution model (LOD 1) that is visible only when it’s diagonal size on screen is between 1000 and 900,000 pixels. The upper bound parameter of 900,000 was increased from earlier (was originally 500,000) and is an estimated number so that it is possible to zoom in on the model as it’s size on screen becomes very large when the camera is up close. And if this parameter is too low the object will effectively just disappear when zooming up close because the size will surpass the given parameter, which it did when using 500,000.
Chapter 5
Results

In this chapter, two separate solutions will be explored, first the GUI application will be evaluated and then the scene implementation will be validated.

The tests were performed on a laptop computer with 2 GB RAM, an Intel Core 2 Duo T5250 (1.5 GHz CPU), a NVIDIA 8600M GT 512 RAM graphics card and running Windows XP Professional SP3.

5.1. GUI application

There are four key areas that define the applications functionality at this point:

i. Automated batch simplification for LOD’s
ii. The basic viewer for simple and quick viewing of models
iii. Visualizing a large scene in real-time to validate the LOD’s
iv. GUI usability

Firstly, the batch simplification is very important as the preprocess of making the LOD’s can be very time consuming if it is done one model at a time. This simplification preprocess is now more automated for the user so that it becomes far easier to build a library of LOD’s.

Second, the viewer is absolutely indispensable for the user in the process of creating LOD’s, as it is necessary to be able to inspect the models easily before and after simplification. This optimizes the workflow when building a large scene, as there involves a lot of models and everything can be done inside of one application. The user does not want to render a large scene or change the code and recompile just for the sake of just viewing a single model.

Thirdly, the whole idea of being able to visualize a large LOD scene is also very important for the application, because the user can test the different level-of-details in real-time and more conveniently evaluate the simplified models in a large 3D environment as they are intended for.

And finally, when it comes to usability for a GUI application there are a number of factors that need to be taken into account as mentioned in Chapter 2. As this is not the focus of this thesis it’s suffice to say that usability issues were factored into the design of the application, focusing on a clean and easy to use application. There were no official usability tests made but quite a few users have tried and tested the application, and even though the application has limited functionality at this point, there were only positive responses. Obviously with expanding the functionality of the application further in the future will then give need for a controlled usability testing.

In the table below you can view some results, mostly regarding preprocessing times, just to give
an idea of how much time it takes the application to visualize a scene (in seconds). Note that these results can change somewhat when replicated, e.g. when using different hardware or even just having multiple software applications open at the same time that uses up memory.

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup the application</td>
<td>1.12</td>
</tr>
<tr>
<td>Batch simplification of the Airport LOD</td>
<td>170.7</td>
</tr>
<tr>
<td>Batch simplification of the Factory to the left</td>
<td>123.6</td>
</tr>
<tr>
<td>Batch simplification of the Tower LOD</td>
<td>81.5</td>
</tr>
<tr>
<td>Batch simplification of the Factory to the right</td>
<td>152.2</td>
</tr>
<tr>
<td>Loading the LOD scene</td>
<td>62</td>
</tr>
<tr>
<td><strong>Total time of visualizing a LOD scene</strong></td>
<td><strong>591.12</strong></td>
</tr>
</tbody>
</table>

Table 5.1. An example of the time it takes to visualize a scene (in seconds).

The final estimation is that it takes “around” 10 minutes to visualize a LOD scene, including the preprocessing, in the application. This is not considered a long time for a user to be able to simplify multiple LOD models and visualize the LODs in a large real-time scene.

### 5.2. LOD scene implementation

Before looking at the results it has to be noted that there are no visible “popping” artifacts when switching between different simplification levels, i.e. the simplification of the models were performed in such a way that a visible difference between the models is very minimal, which means that there is room for further optimization, especially for the lowest resolution LODs that are only visible at 0 to 60 pixels.

For performance testing purposes, and recording the frame rates, a walkthrough was performed in the scene. The walkthrough was structured in the way that it started with the entire scene in view, i.e. the scene is only a small portion of the screen. Then the viewer zooms in on the scene and views it from different angles in perspective so to have the LODs at different levels in the scene. Finally the viewer zooms in on individual LOD objects before zooming out again.

The frame rates were recorded at the different stages of the tests and then the average of those frame rates were calculated for an overall result. The frame rate results are presented as integers for the sake of simplicity.

Below are the statistical results for validating the LOD scene. Please see Appendix B for snapshots from the scene tests, presented with onscreen frame rates.

#### 5.2.1. Object comparison

We begin with looking at statistics for the performance difference of the models when they individually take up the whole area of the viewing screen. This test was done for the scene consisting of the original models and used as a reference for the main results presented in the next section. The whole scene is loaded and then the camera is zoomed in on the model, which includes then a small part of the static object which is the ground. When viewing the static object
individually it was only viewed partly as the whole could not be viewed at once without removing all the other models and that was deemed unnecessary for the results wanted.

As mentioned before the tests were done so that an average frame rate was calculated for the scene when viewing each model individually.

Here, we look at the scene consisting of the original models, i.e. the full resolution models.

- **Static** – the static object
- **Tower** – the tower and building model
- **Factory Left** – the factory to the left of the airport
- **Factory Right** – the factory to the right of the airport
- **Airport** – the airport in the center of the scene
- **All** – all models in view

As can be seen in Figure 5.1, the frame rates decrease sequentially with the size of the models. The models are presented from left to right, starting with the lowest triangle count to the highest, while the static object is a part of the test scene but used purely for visual purposes.

5.2.2. Scene comparison

Here are the main results of comparing the performance of the LOD scene with three other scenes; a scene consisting of the original models, a scene consisting of the LOD 1 models and, for aesthetic purposes, the LOD scene with anti-aliasing turned on.

- **Original** - The scene consisting of the original objects
- **LOD1** - The scene consisting of the first simplified models (LOD1)
- **LOD** - The LOD scene
- **LOD w/AA** - The LOD scene with Anti-Aliasing turned on

As can be seen in Figure 5.1, the frame rates decrease sequentially with the size of the models. The models are presented from left to right, starting with the lowest triangle count to the highest, while the static object is a part of the test scene but used purely for visual purposes.

Looking at Figure 5.2, the results from the test are as expected and the system has accelerated the
rendering in real-time with very little loss of visual quality. In simple terms, the frame rate increases linearly when the triangle count decreases on screen.

With the original objects the frame rate only stays around 6 frames per second (fps) as the system has to render a lot of triangles simultaneously. With the LOD1 models the frame rate increases by an average of 12fps, a 3 times speedup, which is expected as the models are simplified and are less than half the triangle count of the original models.

In the scene consisting of the LODs the frame rate increases to a respectable average of 30fps, which is a 5 times speedup from the original models, and a 1.67 times speedup from the LOD1 models. In this test it is more practical to look at the difference in the frame rates of the LOD1 scene and the LOD scene as the LOD1 models are considered the high resolution models for the LOD scene. The 1.67 times speedup is not large, but still remarkable enough to make a large difference in a real-time application, e.g. computer games. Even with the anti-aliasing turned on in the LOD scene it still is able to maintain an average frame rate higher than the LOD1 scene (20fps).

Worth noting though is that the frame rate does drop as soon as the anti-aliasing is turned on, which was expected as well, as usually the frame rates are affected by the amount of pixels per second the graphics card can render, providing the CPU and other components can keep up.

What the anti-aliasing does is that it improves the appearance of graphics by smoothing the lines between shapes or areas of different color so that the lines do not appear jagged. And when using anti-aliasing the amount of pixels that the graphics card can render will lower by a degree and therefore cut the frame rates down, in this case of about 10fps.
Chapter 6

Conclusion

In this thesis an application has been presented that accelerates the process of simplifying models for a level-of-detail scene, allowing for fast visualization of the simplified models as single objects or in a fully functional LOD scene. This is an important issue in real-time graphics where computers do not have enough memory to store a whole scene, while LOD methods can visualize large scenes by loading LOD models from the hard drive.

The application serves the basic purpose that it was set out to do, even though there could have been more advanced options available in the GUI. With that in mind, the user can do the following:

- Visualize models in the GUI by adding or removing models in the applications viewer window.
- Simplify models into different levels-of-detail efficiently.
- Visualize the models that have been simplified into LODs in a fully functional scene with multiple objects and discrete LOD switching.
- It executes quite quickly.

When considering the application and it’s tools, there are a number of things that could be improved on, these include:

- Real-time rendering techniques – acceleration methods, both software and hardware, could include code optimization and parallel computing.
- The discrete LOD method has some disadvantages, e.g. it is not suited for drastic simplification. Large objects must be subdivided, and small objects must be combined. View-dependent LOD methods solve the first issue, while HLODs solve the second, respectively. Both methods are discussed in Chapter 2.4.

6.1. Future work

- More features and options in the GUI for better interaction.
- Implement more advanced methods for level-of-detail, e.g. view-dependent or HLOD.
- A more polished tool for optimizing 3D models, i.e. to be able to choose a model to optimize with various options and dynamically be able to implement it in a LOD scene, available inside the GUI, without any extra coding.
- Implement a larger environment to visualize, including a terrain and more details, e.g. trees, shadows etc...
- Implement the Donya Labs tool XPRSS, which is a tool that does extreme optimization with a remeshing method.
Bibliography


Bibliography


[28] OpenSceneGraph 2.8.0 Documentation. Doxygen 1.5.8., February, 2009. URL: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/
Appendix A  Program Code

A.1 WxWidgets and OSG code

Listing A.1. GLViewer constructor that extends the wxGLCanvas

```cpp
// GLViewer constructor that extends the wxGLCanvas
GLViewer::GLViewer(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size,
                    long style, const wxString& name, int *attributes)
    : wxGLCanvas(parent, id, pos, size, style|wxFULL_REPAINT_ON_RESIZE , name, attributes)
{
    // default cursor to standard
    _oldCursor = *wxSTANDARD_CURSOR;
    _viewer = 0;
    Init();
}
```

Listing A.2. Set a new WxGLCanvas in the GUI

```cpp
// set a new wxGLCanvas in the GUI
mGLViewer = new GLViewer(this, wxID_ANY, wxDefaultPosition, wxDefaultSize,
                          wxSTATIC_BORDER, wxT("Donya Viewer"), attributes);
```

Listing A.3. Initialize the rendering context for the wxGLCanvas (GLViewer)

```cpp
// initialize the rendering context for the wxGLCanvas (GLViewer)
void GLViewer::Init()
{
    wxSize sz = GetSize();
    this->gw = new GraphicsWindowWX(this);
    this->SetGraphicsWindow(gw);

    _viewer = new osgViewer::Viewer;
    _viewer->getCamera()->setGraphicsContext(gw);
    _viewer->getCamera()->setViewport(0,0, sz.x, sz.y);
    _viewer->addEventHandler(new osgViewer::StatsHandler);
    _viewer->setThreadingModel(osgViewer::Viewer::SingleThreaded);
    if (!_viewer->areThreadsRunning())
        _viewer->startThreading();
    osg::setNotifyLevel(osg::WARN);
    initOSG();
}
```
Appendix A  Program Code

```cpp
bool GraphicsWindowWX::makeCurrentImplementation()
{
    _canvas->SetCurrent();
    return true;
}

// --------------------------------------------------------------------------------------------------
void GraphicsWindowWX::swapBuffersImplementation()
{
    _canvas->SwapBuffers();
}
```

**Listing A.4.** SetCurrent and SwapBuffers for the window

```cpp
// initialize the scenegraph for the viewer
bool GLViewer::initOSG()
{
    // create the root of our scene graph
    root = new osg::Group;
    _viewer->setSceneData(root.get());
    _viewer->setCameraManipulator(new osgGA::TrackballManipulator);

    return true;
}
```

**Listing A.5.** Initialize the scenegraph for the viewer

### A.2 Scene implementation code
// static object...
osg::Node* one =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/scene.dae");
root->addChild(one);

//*********************************LOD Objects*************************************

// airport in the middle
osg::Group* lod1root = new osg::Group;
osg::Node* lod1one =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/19red4b.dae");
osg::Node* lod1two =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/19red3b.dae");
osg::Node* lod1three =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/19red2.dae");
osg::Node* lod1four =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/19red1.dae");

// factory on the left
osg::Group* lod2root = new osg::Group;
osg::Node* lod2one =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/17red6.dae");
osg::Node* lod2two =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/17red3.dae");
osg::Node* lod2three =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/17red2b.dae");
osg::Node* lod2four =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/17red1.dae");

// airport tower + building
osg::Group* lod3root = new osg::Group;
osg::Node* lod3one =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/02red4.dae");
osg::Node* lod3two =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/02red3.dae");
osg::Node* lod3three =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/02red2.dae");
osg::Node* lod3four =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/02red1.dae");

// factory on the right
osg::Group* lod4root = new osg::Group;
osg::Node* lod4one =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/22red4.dae");
osg::Node* lod4two =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/22red3.dae");
osg::Node* lod4three =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/22red2.dae");
osg::Node* lod4four =
    osgDB::readNodeFile("C:/halli/donyalabs/gui/project/Viewer/models/sceneReadyModels/LOD/22red1.dae");

Listing A.6. Loading the LOD models
Appendix A  Program Code

Listing A.7. LOD settings
//********************add transforms and then add the lods to the root**************
// main airport...
root->addChild(lod1.get());

// factory on the left...
    osg::PositionAttitudeTransform* lod2Xform = new osg::PositionAttitudeTransform();
    root->addChild(lod2Xform);
    lod2Xform->addChild(lod2.get());
    lod2Posit(-272,0,95); // x, y, z - er vinstri | - er upp | - er nær
    lod2Xform->setPosition(lod2Posit);

// airport tower + house.....
    osg::PositionAttitudeTransform* lod3Xform = new osg::PositionAttitudeTransform();
    root->addChild(lod3Xform);
    lod3Posit(-176,0,-250); // x, y, z - er vinstri | - er upp | - er nær
    lod3Xform->setPosition(lod3Posit);

// factory on the right....
    osg::MatrixTransform* rotate4 = new osg::MatrixTransform;
    rotate4->setMatrix(osg::Matrix::rotate(osg::inDegrees(90.0f),0.0f,1.0f,0.0f));
    rotate4->addChild(lod4.get());
    lod4Xform->addChild(rotate4);
    lod4Posit(150,0,50); // x, y, z - er vinstri | - er upp | - er nær
    lod4Xform->setPosition(lod4Posit);

// optimize the lods
    osgUtil::Optimizer optimizer;
    optimizer.optimize(lod1.get());
    optimizer.optimize(lod2.get());
    optimizer.optimize(lod3.get());
    optimizer.optimize(lod4.get());
Appendix B  Screenshots

B.1 Original objects

Figure B.1. A screenshot of the original objects, framerate at only 6.62.

Figure B.2. A screenshot of the original objects, different angle, framerate at 5.19.
B.2 LOD1 scene
B.3 LOD scene

Figure B.5. A screenshot of the LOD scene, framerate at 31.71.

Figure B.6. A screenshot of the LOD scene with anti-aliasing enabled, framerate at 18.65.
Figure B.7. A screenshot of the AA LOD scene, different angle.