COMPUTER SCIENCE HONOURS THESIS

INVESTIGATING THE PERFORMANCE CHARACTERISTICS OF A HIERARCHICAL LEVEL-OF-DETAIL MESH PROCESSOR

October 29, 2014

Author:
Benjamin Francis Meier
University of Cape Town
Department of Computer Science
benmeier42@gmail.com

Supervisor:
Patrick Marais
University of Cape Town
Department of Computer Science
pmarais@cs.uct.ac.za

	Category	**Mark Allocation**
1	Requirement Analysis and Design	13
2	Theoretical Analysis	3
3	Experiment Design and Execution	5
4	System Development and Implementation	15
5	Results, Findings and Conclusion	10
6	Aim Formulation and Background Work	14
7	Quality of Report Writing and Presentation	10
8	Adherence to Project Proposal and Quality of Deliverables	10
9	Overall General Project Evaluation	0
	Total Marks	**80**

Abstract

The Zamani project, run by researchers in the UCT Geomatics Department, required a method of rendering very large architectural models generated through laser scanning. The models, that may consist of hundreds of millions to billions of triangles, are too large to view using existing software. In this report, we implemented a Preprocessor as part of a Hierarchical Level of Detail strategy. Level of Detail schemes are a commonly used method of increasing rendering speed of 3 dimensional models. Hierarchical Level of Detail utilises a hierarchical space partitioning scheme to build a hierarchy of meshes each of which represented a region of the model at a certain level of simplification. A renderer can pick the meshes to be rendered based on the position of the camera or distance of the camera from the model in order to reduce detail in regions of the model that are not as visually important as other regions.

This report focussed on testing the differences between multiple hierarchical space partitioning schemes particularly the difference between balanced and unbalanced hierarchies.

Acknowledgements

This project would not have been possibly without the guidance, support, and assistance of all those involved. In particular our project supervisor, Patrick Marais, who provided valuable input and expertise throughout the project. Patrick's experience in the field of computer graphics helped to set us on the right track when developing our project solution and splitting our project topic into manageable portions.

My project partners, Daniel Burnham-King and Justin Cossutti, put a lot of time and effort into this project and made a great team to work with. I am very grateful to have had a team that were enthusiastic and motivated to create an effective solution for the Zamani group.

I'd also like to thank my friends and family for the support they provided throughout the project. They were always there for me when I got stuck on various obstacles and needed some support and advice.

The Zamani group and its researchers from the Geomatics department proved to be great clients, and were always happy to listen to our proposed solutions and to provide example datasets.

Finally, this project would not have been possibly without a scholarship provided by the National Research Foundation.

Contents

1 | Introduction

The preservation of historical heritage sites is very important for education and future research. Historical sites, especially buildings, are constantly threatened by erosion and natural, or man-made, disasters. The Zamani Project, run by the Geomatics Department at the University of Cape Town, aims to digitally capture accurate three dimensional architectural models of African heritage sites for storage in the African Cultural Heritage and Landscape Database[RHB+12]. The project was conceptualised in 2001 by Professor Heinz Rüther in collaboration with the ITHAKA and Aluka organisations. The project utilises phase-based laser scanners along with conventional surveys and GPS recordings in order to build highly accurate three dimensional models. These models can be very detailed and can contain billions of triangles.

The researchers use existing mesh processing software, such as MeshLab and ArcGIS, to navigate and interact with the resulting model, but the models often contain too many triangles to render at an interactive frame rate[1]. The solution the researchers currently use is to reduce the density of the mesh by sampling the point cloud and thereby reducing the total number of triangles but this destroys detail such as crevices, engravings and sharp edges, which are particularly important to researchers.

1.1 Problem statement and proposed solution

The overall problem faced by the Zamani group is that they are not able to view or manipulate very large meshes at interactive frame rate. Rendering a 3D model containing around 3 million triangles in MeshLab on our test hardware achieved approximately 3.1 frames per second which is well below the 30 frames per second target. This is even more of a problem since the Zamani researchers are looking to interact with much larger models containing hundreds of millions of triangles.

To address this issue, we developed a renderer that uses Level of Detail mechanisms to reduce the detail in the mesh and to increase the rendering speed of the model. Level of detail mechanisms allow the renderer to intelligently control the number of triangles used to represent the model, based on the position of the camera

in the 3 dimensional scene, in such a way that fewer triangles are rendered but with almost no visual discrepancy to the original mesh. This concept and procedure will be described in more detail in Chapter 2.

One of the Level of Detail schemes used to render large models is Hierarchical Level of Detail. Hierarchical level of detail uses a hierarchical space-partitioning algorithm to generate a tree structure in which each node of the tree contains a different amount of detail for a specific region of the original model. This structure is generted as part of a preprocessing step which generates a larger file representing the original input model. In chapter 2 we establish that the Hierarchical Level of Detail structure is most appropriate for our problem. There are multiple space-partitioning algorithms that can be used, each one using different heuristics to partition 3 dimensional space. The remainder of this report is dedicated to comparing the performance of the different partitioning algorithms during processing as well as identifying any relationships between the algorithms and the dimensions of the input data.

1.2 Research Questions

The overall research question faced by this report, in conjunction with its companion reports, is:

- "Is it possible to process and render very large architectural models in such a manner than interactive frame rates are achieved with no noticeable loss of detail?"

Since this report specifically the details with the Preprocessor component, the research questions we want to answer in this report are:

- "Is Hierarchical Level of Detail a viable strategy when meshes are very large?"
- "Is a balanced hierarchical tree structure more appropriate than an unbalanced one in the context of Hierarchical Level of Detail"

In order to answer these questions we developed a software suite consisting of multiple components that allow the user to preprocess a model and then view the output. The software is designed to integrate into the workflow of the Zamani project researchers and is designed to process their data with no modification.

The structure of this report is as follows: a background chapter gives a detailed background to the topic of Level of Detail schemes, provides a summary of current research in the field, and shows our reasons for choosing a Hierarchical Level of Detail scheme over alternative schemes; A design chapter describes the overall structure of the Preprocessing component as well as the functions of the major subcomponents; The implementation chapter provides a detailed description of how the subcomponents were developed as well as choices made regarding algorithms and optimisations; This report then describes the methods we used to evaluate space-partitioning algorithms, as well as the results of said evaluations; Finally, we summarise the findings of this report in the concluding chapter.

[1] An interactive frame rate is a rate of frames per second that allows the user to navigate or manipulate the mesh effectively. A frame rate that is too low will cause stuttering and flickering and make the user experience unpleasant. For our project we will set 30 frames per second as our target frame rate.

2 | Background

This chapter aims to present domain-specific background of the research space and to report on research that influenced the development of this project. We focus on the purpose of level of detail schemes, what types of level of detail schemes are used, and the different types of datastructures that enable them.

Prior to discussing the purpose of level of detail schemes we briefly revisit the general structure of a 3 dimensional (3D) mesh and explain how meshes are rendered by graphics hardware. It is important to be aware of the problems that Level of Detail schemes solve in order to render large polygonal meshes.

2.1 Mesh Rendering

In 3D computer graphics, polygonal meshes are used to represent 3D objects, known as models. At the simplest level each mesh consists of a collection of vertices, edges, and faces. Each vertex is a 3D point in space and can contain additional information such as colour, texture coordinate, or a normal vector. An edge is a line between two vertices. Polygonal faces are specified as a list of vertices that form a closed loop of edges. Because faces are necessarily comprised of coplanar vertices, triangles are the most common type of face since a unique set of three points specifies a plane. Quadrilateral faces, formed by 4 vertices can be used but are split into two neighbouring triangle faces during rendering.

When a mesh is rendered by a Graphics Processing Unit (GPU), its vertex and face information is transferred to GPU memory and passed into the input of the graphics pipeline. The graphics pipeline is comprised of multiple stages each performing certain actions in order to create a 2D raster image from a 3D scene [FPI84]. Figure 2.2 shows the stages of a simple graphics pipeline. The vertex processor transforms vertices from 3D coordinate into 2D coordinates on the screen; The geometry processor creates primitives and discards those which are not visible on the screen; The fragment processor generates pixel fragments with colour and texture before passing them to the frame buffer in which depth and alpha testing are performed. The frame buffer is written to the screen to form the final image.

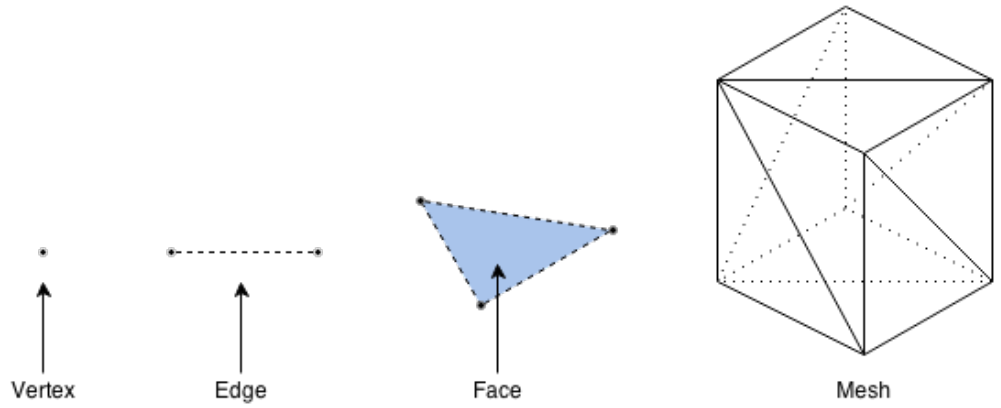


Figure 2.1: This image shows an example of a vertex, edge, face, and mesh. The mesh in the right hand image is made up of 12 triangular faces: each rectangular side of the box has been split into 2 triangles.

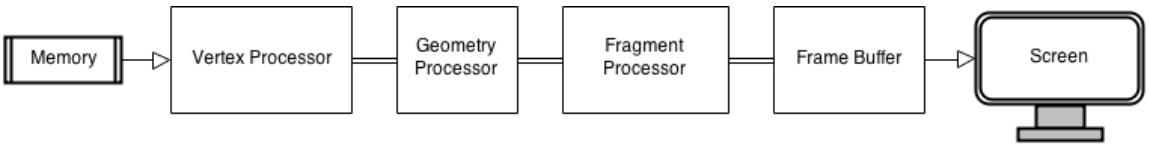


Figure 2.2: Simplified structure of a graphics pipeline. Mesh information enters the pipeline from system memory or GPU memory and proceeds through the pipeline before being written to the frame buffer which is displayed on the screen.

This pipeline contains both serial and parallel components: data describing the scene is passed through each stage individually in order but many stages can be highly parallelised internally. Because the overall stages of the graphics pipeline are serial, the speed of rendering is determined by the slowest stage in the pipeline. As such, the speed of rendering a single frame is dependant on both the number of triangles processed per second, and the size of the frame being rendered. Since the processing rate of the graphics hardware and the size of the screen can be constant, a reduction in the number of triangles directly increases the rate at which frames are rendered. This is known as the frame rate and is measured in frames per second (FPS).

Reducing the number of triangles in a mesh is an effective method of increasing rendering performance of large meshes since it reduces load on the vertex, geometry, and fragment processors in the graphics pipeline. Reducing the number of bytes required to represent the mesh can also increase performance since transferring the

mesh from memory to the graphics hardware takes less time[Fer04].

2.2 Level of Detail

Level of Detail (LOD) techniques seek to decrease mesh complexity and density according to metrics such as distance from a camera, rendering performance, or movement speed of an object [Ali10]. They increase rendering performance by rendering less geometry in a way that is intended to be unnoticeable to the human eye but with the advantage of increased rendering performance. The increase in rendering performance comes from a reduced workload in graphics pipeline stages such as vertex and fragment processing. A decrease in mesh complexity also reduces memory transfers between the GPU and main memory as well as cache usage on the GPU itself.

LOD techniques are common in computer games, where they are used to manage detail in large 3D scenes. A distant object that is further away in the scene can be represented using fewer triangles than a similar object that is closer to the camera and thus takes up a larger portion of the screen.

2.2.1 Discrete Level of Detail

Two main LOD categories exist for managing level of detail: Discrete Level of Detail and Continuous Level of Detail [RLB10]. Discrete Level of Detail is the technique most commonly used in 3D graphics and is a form of 3D mipmapping. In this approach, a small number of levels of detail for one model are stored at fixed resolutions.

These are generated as a preprocessing step using a simplification algorithm and are selected between at runtime depending on the distance to the camera. The swapping is performed at fixed intervals as the camera moves towards or away from a model in order to maintain the illusion that the model has a fixed amount of detail. This swapping causes a 'popping' effect seen when low detail models are suddenly replaced by their higher detail counterparts, this can be avoided by increasing the number of individual levels of detail which comes at the cost of increased memory usage and transfer costs. Lower levels of detail can also exhibit bad silhouettes when curved surfaces or edges are simplified. Discrete LOD meshes can also trivially contain additional mesh information such as texture maps since they are each a standalone model.

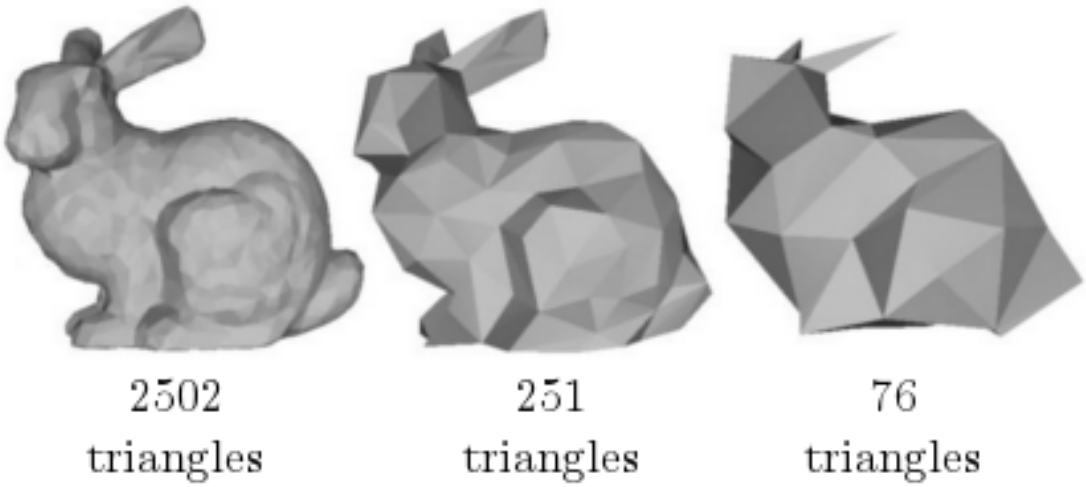


Figure 2.3: Illustration 3D Level Of Detail. Discrete degradations of the original data are used to approximate the original model using fewer triangles.

Source: Source: Stanford 3D Scanning Repository

2.2.2 Continuous Level of Detail

In order to avoid the popping effect caused by Discrete LOD one must use data structures that allow continuous variability in the complexity of the model at run time[RLB10]. These Continuous Level of Detail algorithms do not rely on fixed intervals and instead encode the original mesh in a way that allows the mesh to be reduced to an arbitrary number of triangles. This means that the detail in the mesh can be controlled as a continuous function of the distance of the model from the camera. The net result, when compared to Discrete LOD, is higher fidelity due to the continuous increase or decrease in detail as camera moves closer to or further away from the object.

Continuous LOD requires specific data structures from which the desired amount of detail can be extracted at run time. Progressive Meshes[Hop96] are a Continuous LOD data structure which encode the mesh as a coarse base mesh along with a ordered list of vertex split operations which define where new vertices are added, and how they are linked to the existing mesh at that point. If the entire list of operations is performed, the original mesh is recovered. Some Continuous LOD solutions utilise discrete meshes that are then interpolated between using smooth transitions to simulate a continuous refinement.

2.2.3 View-dependant Level of Detail

View-dependant Level of Detail is an extension of Continuous LOD that allows for continuous variability of arbitrary regions of a mesh. This is view-dependant as it can be used to refine mesh geometry closest to a camera while keeping the rest of the mesh at a low level of detail. View-dependant LOD also allows regions of the mesh that form the silhouette of the model to be shown at higher detail levels than interior regions. Many view-dependant solutions exist for point based rendering methods, such as the QSplat algorithm [RL00], but our project requires polygonal meshes containing triangular faces to be processed.

In 2000, Chris Prince developed an extension to Progressive Mesh data structure that allows View-dependant LOD. The technique, known as View Dependent Progressive Meshing [Pri00][Hop98], uses hierarchical data structures to load the vertex operations that apply to a particular area of a mesh. Prince also managed to develop significant improvements over Hoppe's original Progressive Mesh implementation[Hop96] by optimising both the simplification during preprocessing and the selection of new geometry when rendering the mesh. Prince also developed mesh splitting and stitching techniques that are applicable to our project. The main disadvantage of the Progressive mesh is the increased memory and CPU usage. Applying the vertex operations when rendering the mesh must take place on the CPU and in general cannot be performed by the graphics processor. Recent research has shown that View Dependent Progressive Meshing can be performed on the GPU very efficiently[HSH10].

2.2.4 Hierarchical Level of Detail

Hierarchical Level of Detail [Lak04] data structures combine properties of both Discrete LOD and View-dependant LOD. A Hierarchical space partitioning algorithm is used to recursively partition the mesh. Each resulting submesh in the tree is then simplified so that it contains much less geometry than its child nodes but still represents the same region in the mesh as its child nodes do. Leaf nodes contain the original mesh detail and are not simplified. The hierarchical data structure allows mesh regions to be replaced by their higher detail child nodes when at a certain distance from the camera. The data structure also allows sub-trees of objects that are not in view to be discarded. Hierarchical space-partitioning algorithms are used since the root node represents the entire 3D space, and its children represent further subdivisions of this space. Hierarchical LOD techniques still exhibit 'popping' arte-

facts, since transitions are not continuous, but these artefacts only appear in a small subset of the mesh.

Hierarchical LOD as also been combined with Continuous LOD so that the transitions between parent nodes and their child nodes are controlled by continuous interpolation[CZGZ08].

2.2.5 Summary

From the four main LOD schemes described above, two main categories arise: Discrete LOD, utilising fixed meshes; and Continuous LOD, which allows continuous refinement. Discrete LOD gains an advantage by yielding faster frame rates at the expense of more obvious 'popping' artefacts whilst Continuous LOD allows for continuous and variable refinement, but uses more CPU time and additional memory in order to support this. Continuous LOD algorithms are also more complex to implement than their Discrete counterparts. Hierarchical LOD schemes provide a good compromise between Discrete and Continuous LOD and for these reasons we further investigate the use of Hierarchical LOD schemes in section 2.3.

Table 2.1: Summary of LOD discussed techniques

LOD Technique	**Memory Usage**	**Implementation Complexity**	**Visual Fidelity**
Discrete LOD	Medium	Low	Low
Progressive Meshes	Low	High	High
View-Dependent Progressive Meshes	Medium	High	High
Hierarchical LOD	High	Medium	Medium

This table briefly compares the discussed Level-Of-Detail techniques. Visual Fidelity refers to the smoothness of the transitions between different Levels-Of-Detail.

2.3 Hierarchical Space Partitioning

Key to the implementation of Hierarchical LOD algorithms is a hierarchical spatial partitioning algorithm. Space-partitioning algorithms divide a space or N-dimensional volume into multiple non-overlapping subsets or regions along some splitting hyperplane. This spatial division is repeated recursively into the created

subregions in order to form a tree. Any point in this space can be said to lie in a particular region and thus in all of that regions ancestor regions. This process of spacial partitioning is repeated until stopping conditions are met such as the depth of the tree, or volume of the leaf regions. Spatial partitioning algorithms are commonly used to identify candidates for collision testing such as raycasting, bounding box collisions, or other spatial indexing tasks. Common space-partitioning data structures include Quadtrees, Octrees, and KD-Trees.

2.3.1 Quadtree

Quadtree data structures[FB74] divide each spatial region into exactly 4 child regions, each representing one of its quadrants. Quadtrees are commonly used to divide 2 dimensional space since the division occurs on only 2 axes. The point on which the division occurs is generally on the center of a node, but can be adjusted to optimise the weighting of the 4 subnodes.

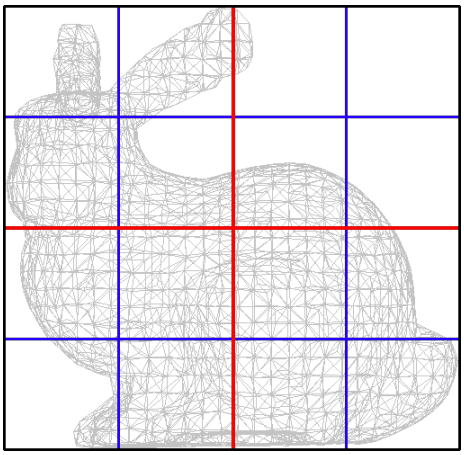


Figure 2.4: Image showing a 2 level quadtree split of an object. The red lines initially divide the object into 4 child nodes, while the blue lines further divide the 4 nodes into 16 grandchild nodes.

Each node is only divided if it meets certain criteria such as exceeding a certain area or volume, or by containing more items than a certain threshold.

2.3.2 Octree

Octrees[LM80] are a 3 dimensional extension of Quadtrees in which each node is split into exactly 8 child nodes. Each axis, X, Y, and Z, is halved in order to determine the limits of each of the 8 subnodes. Octrees were initially used for high speed generation and rendering of solid objects.

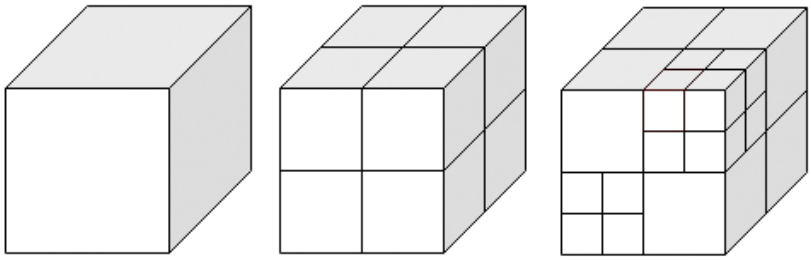


Figure 2.5: Image showing the result of splitting a cube multiple times using an octree. Notice that only 2 of the sub nodes have been further divided in this example.

2.3.3 KD-Tree

A KD-tree is a variety of binary space partitioning algorithm (BSP) in which each region is bisected into 2 child regions by splitting the longest side of the axially aligned bounding box that encloses the region.

Variable KD-Tree

The variable KD-Tree variant splits a region on the median value of the longest axis rather than simply bisecting it. This median value is determined based on the distribution of the items contained by the region.This has the effect of producing 2 equally weighted sub nodes. Variable KD-Tree's are inherently balanced trees, as long as the splitting point is calculated accurately, since geometry is distributed equally between all nodes at the same level of the tree. The Variable KD Tree is commonly known simply as a KD Tree but this report makes a distinction between the simpler, bisecting KD-Tree and the Variable KD Tree.

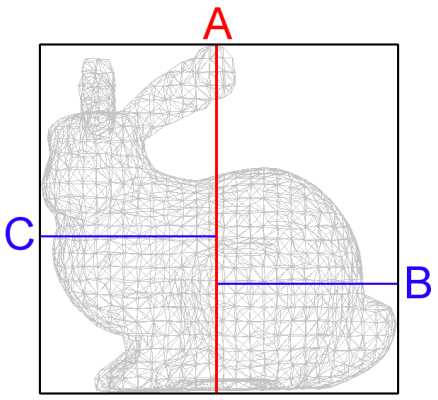


Figure 2.6: This image shows an approximate 2 level Variable KD-tree split of a 3D model. The first split is the red line marked A which divides the geometry into equally weighted halves. The second cuts, B and C, divide the halves into equally weighted quarters on the horizontal axis.

2.3.4 Multiway KD-Tree

The Multiway KD-tree is an extension of the variable KD-tree which splits each node into N subnodes. This requires N-1 splitting points along the longest axis. The splitting points are chosen as equally spaced percentiles of the axis-values. For example if N equals 4, then the splitting points will be the first, second, and third quartile values. Figure 2.7 shows an example of an order 3 variable KD-tree.

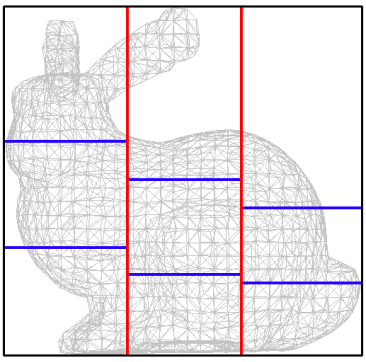


Figure 2.7: This image shows an approximate 2 level split of a model using a Multiway KD-tree of order 3. Instead of dividing on the median as the variable KD-tree does, it splits the mesh into 3 equally weighted thirds based on the node's content.

2.3.5 Summary & Comparison

Due to the fact that the Hierarchical space-partitioning data structure is used for 3D models. The quadtree would not be suitable for a Hierarchical LOD scheme since it only divides 2 of the 3 dimensions. Of the remaining 4 data structures described in the previous section, the variable KD-Tree and multiway KD-Tree are balanced tree structures. This is possibly important for a Hierarchical LOD scheme since the number of levels of detail in all parts of the mesh will be the same.

We decided to implement a Preprocessor that uses a Hierarchical Level of Detail Scheme. In the rest of this project we investigate the performance differences that exist between the different space-partitioning data structures when implemented as Hierarchical Level of Detail schemes.

Table 2.2: Summary of Hierarchical space-partitioning schemes

Tree Data Structure	**Nodes Per Split**	**Balanced**
Octree	8	No
KD-Tree	2	No
Variable KD-Tree	2	Yes
Multiway KD-Tree	>2	Yes

3 | Design

In this chapter we begin by discussing the architecture of both the overall system and of the mesh pre-processor developed in this report. We highlight the major roles and functions of each component and discuss some of the more important algorithms used for implementing a hierarchical level of detail system. As discussed in the previous chapter, we decided to choose the Hierarchical Level Of Detail structure and associated algorithms since they are applicable to very large meshes, and are more straightforward to implement since they do not require GPU programming techniques.

3.1 System Architecture

The goal of the overall system is to develop a set of components that can be easily integrated into the work flow of researchers working in the Zamani project. As mentioned in the background chapter, we wished to develop a system that uses the advantages of a hierarchical level of detail structure in order to allow researchers to view and walk-through very large three-dimensional models at interactive frame rates.

Because the building of a hierarchical level of detail data structure is a pre-processing step that only needs to be performed once per model, we separate this entirely from the viewing of the model. At this point we can treat each stage in the process as being a black box with a fixed interface of inputs and outputs which helps to reduce complexity and interdependency as well as allowing for isolated testing. The Pre-processor is responsible for consuming the 3D model generated by the Zamani project and for building the hierarchical level of detail structure. The level of detail structure is then consumed by a Viewer that presents the model to the user. Figure 3.1 illustrates separation of the Pre-processor from the Viewer.

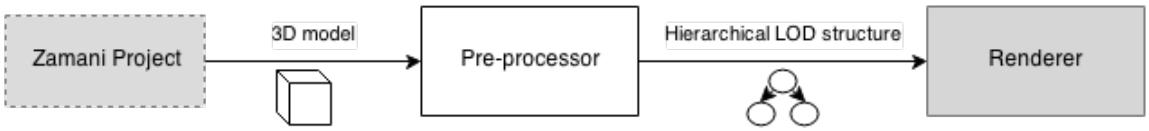


Figure 3.1: This figure shows the major components of the system: The Pre-processor and Viewer, and how they link together with the Zamani project. The Pre-processor is the subject of this report.

3.2 Pre-processor Architecture

This report is concerned with the development of the pre-processing component. In order to simplify the development of this component we can again break it down into a set of unique components each with their own functional responsibilities.

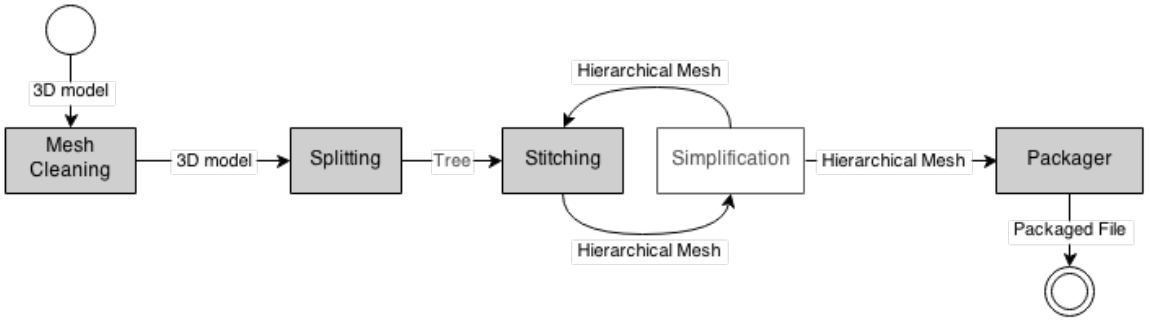


Figure 3.2: The component-wise break down of the pre-processor. The circle at the top of the image indicates the entry point and the double circle at the bottom indicates the end of the processing. The grey components will be implemented from scratch for this report.

The algorithm used by the preprocessor is designed to be recursive, each mesh node in the tree is made up of a simplified version of its processed child nodes stitched together. This will be presented in more detail in the implementation chapter.

3.2.1 Mesh Cleaning

The initial stage in the pre-processor is a mesh cleaning step. This step removes any unnecessary information from the model and applies a transform to the vertices of the mesh in order to normalise its orientation, position, and size. The orientation of the mesh may need to be modified since there are multiple conventions for the coordinate systems used by 3D programs. Firstly, a 3D environment could use either

a left-handed or a right-handed coordinate system; Secondly, some applications use the Y axis as the vertical axis, while others use the Z axis as their vertical axis. Our pre-processor must be able to translate a model from its original coordinate system into the coordinate system used by the Viewer. This is very important because without this the model will appear to be mirrored or will not match the orientation of the physical structure from which it was captured.

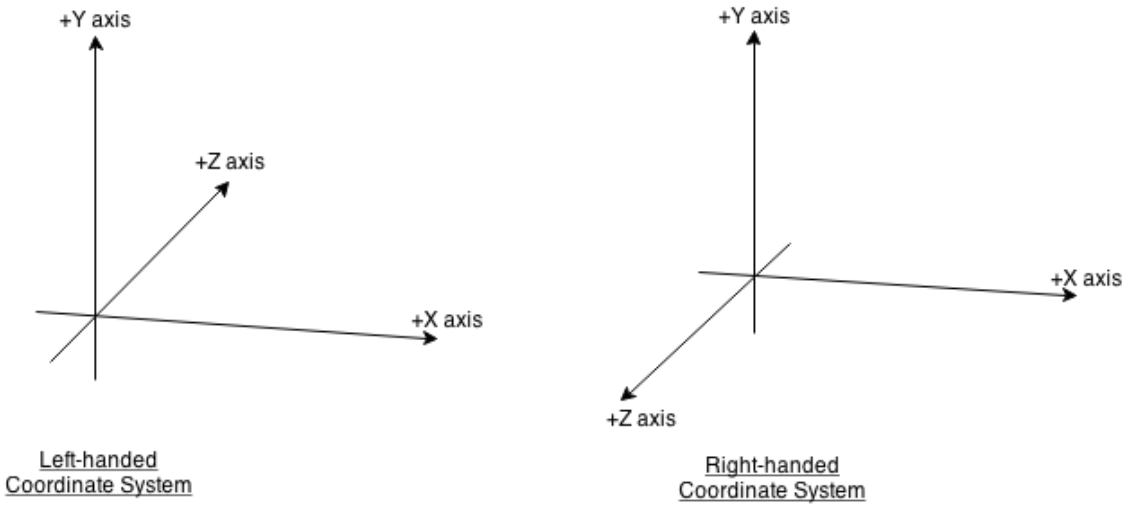


Figure 3.3: This image shows the difference between traditional left-handed coordinate systems and right-handed coordinate systems. Note the difference in the direction of the positive Z axis. Although it lies along the same line, it lies in the opposite direction.

The mesh must also be repositioned so that its center lies on the 3D origin, and rescaled so that it fits inside a predefined box. Rescaling and repositioning is required since the Viewer requires the mesh to be at a predictable location with a predictable size in order to aid camera tracking when rendering the model.

3.2.2 Mesh Splitting

The hierarchical space-partitioning algorithm is used to divide the mesh into a tree structure. This is performed in a top down fashion. A root node is first defined, which contains the entire mesh. The space-partitioning algorithm then picks a splitting plane based on either the dimensions of the bounding box, or the distribution of vertices in the mesh. Once the mesh is split into its submeshes, the submeshes are each attached to the root nodes as child nodes, and the process is repeated on them

until a certain tree depth or node size requirement is reached. For example, a KD-tree would split each mesh into 2 submeshes by halving the longest axis of its axially aligned bounding box.

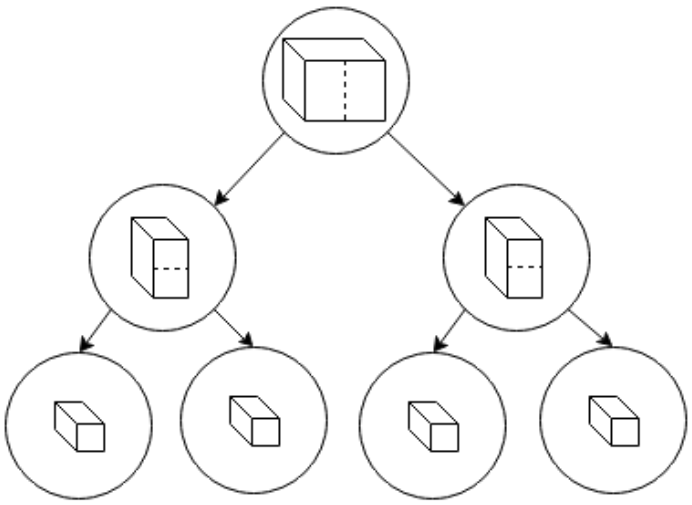


Figure 3.4: An example of a KD-tree bounding box that has been split 2 times.

There are multiple ways of handling faces that fall across the splitting plane: some algorithms split the triangle into multiple small triangles, some duplicate the triangle so that it falls on both sides, and some use metrics based on area, position, or orientation to allocate the triangle to a side (See Figure 3.5). We will be using the latter approach due to simplicity: A triangular face is determined to be inside the sub-mesh in which at least 2 of its vertices lie.

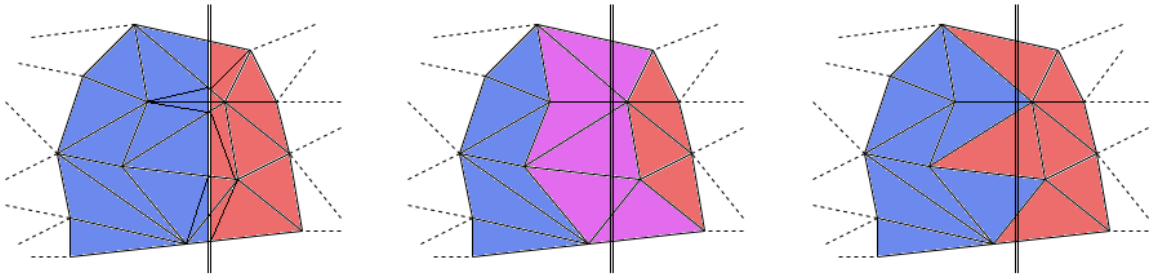


Figure 3.5: This image shows 3 different approaches to mesh splitting. The left-most technique adds additional vertices and edges in order to split triangles; The middle approach simply duplicates triangles into both sub-regions; The right-most technique allows triangles to cross the splitting plane but allocates them to the sub-region in which at least 2 of their vertices lie.

3.2.3 Simplification

Simplification of a mesh is vital for developing a level of detail scheme. The core purpose of the simplifier is to return a copy of the original mesh containing fewer faces and edges but without sacrificing too much geometric detail. The simplification process is controlled via user-defined quality control parameters.

An important constraint of the simplifier in our application is that it must be capable of preserving vertices that lie on the boundary of the mesh. This is in order to support seamless mesh stitching (Section 3.2.4).

For this report, we will be using a 3rd-party executable for simplification and will treat it as a black-box system to which we input a file and a target number of faces.

3.2.4 Mesh Stitching

Once the mesh has been fully split to its greatest depth, mesh stitching and simplification are used to create the final meshes for the interior nodes of the tree. Mesh stitching is used to join together the child meshes of each node into a single continuous mesh that can be passed to the simplifier. Mesh stitching cannot simply append the faces and vertices into a single file but must join duplicate vertices in order to form a single continuous mesh. Without proper stitching, the simplifier will cause seams to appear in the mesh as boundary edges retreat due to artefacts caused by the collapse of edges.

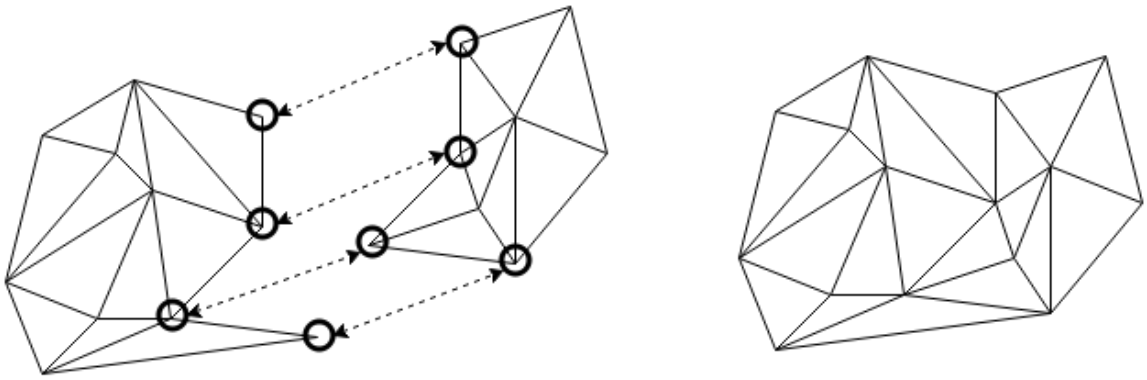


Figure 3.6: An example of 2 meshes that have been stitched together along their boundary. The circled vertices indicate boundary vertices that will stitched together and the dashed lines indicate corresponding pairs of vertices that are spatial duplicates. The meshes have been pulled apart in the left hand image to show that they are disconnected but in reality vertex pairs occupy the same point in space.

3.2.5 Packaging

The pre-processor generates a hierarchical level of detail in the form of a tree structure. Each node contains a simplified mesh, as well as information such as the bounding box, and number of vertices or faces. In order to compile this into a flat file that can be imported into the Viewer, we chose to perform a breadth first traversal of the tree and write the mesh geometry to the a destination file. By prepending a header containing the relationships between nodes and the position of their geometry in the file, the Viewer can easily retrieve a specific node in constant time.

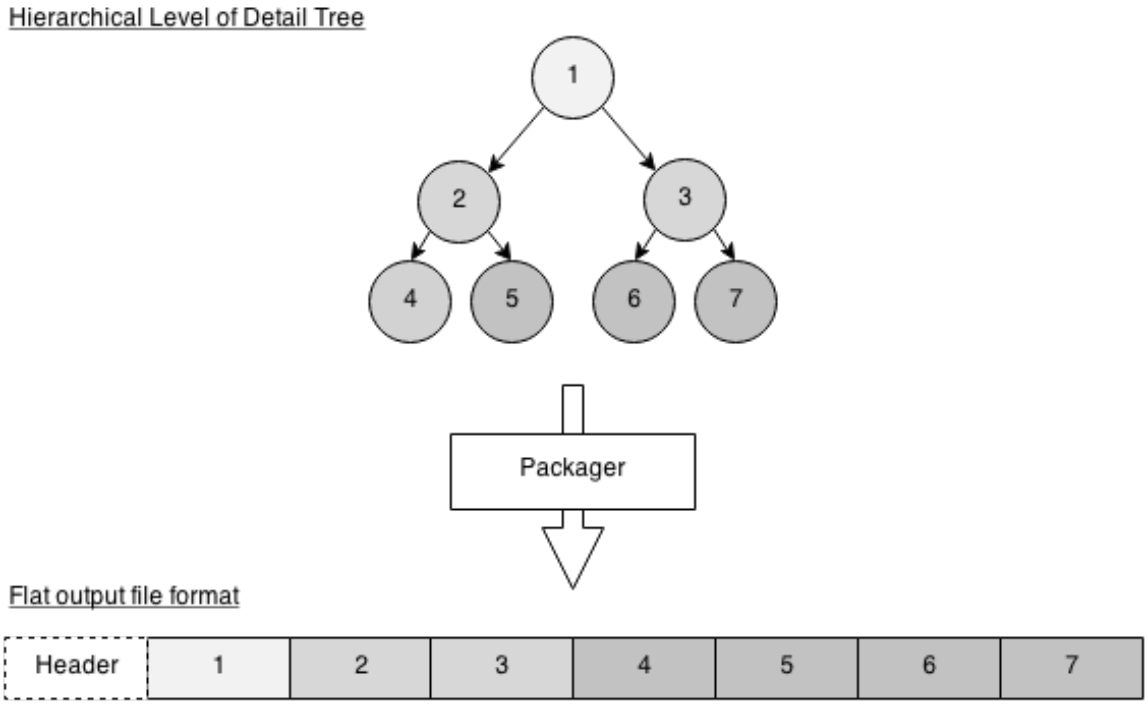


Figure 3.7: The above image shows how the hierarchical tree is transformed into a flat file by the packager. The header at the beginning of the file should allow links between the nodes to be rebuilt and the tree structure to be recreated.

An alternative approach would be to maintain the tree structure using a system of folders, sub-folders, and files. We decided against this as it is harder to distribute and can be easily corrupted if files are removed from the folder structure.

3.3 Hierarchical Space-Partitioning

In this report we test the performance of 4 hierarchical space partitioning schemes, namely: Octree, KD-Tree, Variable KD-Tree, and multiway KD-Tree. Each of these schemes uses a different technique to divide the mesh into sub-meshes.

Algorithm **??** shows the general structure of the recursive splitting operation used to divide the mesh and start the hierarchical tree construction. This algorithm can be used for the Octree, KD-Tree, Variable KD-Tree and Multiway KD-Tree.

Algorithm 1 The *.split*() function on line 6 changes dependant on the hierarchical structure being used while the *.canBeSplit*() function on line 8 determines whether the current mesh node can be split further.

```
1: procedure RecursiveSplit(mesh)
2:     q ← Queue
3:     q.add(mesh)
4:     repeat
5:         n ← q.pop()
6:         children ← n.split()
7:         for each c in children do
8:             if c.canBeSplit() then
9:                 q.add(c)
10:            end if
11:        end for
12:    until q.empty()
13: end procedure
```

3.3.1 Octree

The octree structure splits each axis-aligned rectangular region into 8 sub-regions by bisecting each axis. Because the position of the splitting plane is determined based on the dimension of the mesh's bounding box and not on the density or distribution of mesh's geometry, child nodes of a parent node do not contain equal distributions of geometry and can sometimes be completely empty. The height of the tree is either limited by some user-specified value, or nodes are split until they contain fewer vertices than some predefined limit.

3.3.2 KD-Tree

The KD-tree performs a similar splitting operation to the Octree, but bisects each region into two smaller regions. The axis to be bisected is the longest axis of the bounding box containing the geometry of the node. This ensures that the nodes do not become too thin and maintain a fairly consistent aspect ratio. The KD-tree can still contain empty nodes if geometry is not equally distributed.

3.3.3 Variable KD-Tree

The variable KD-tree takes the distribution of the geometry of the node into account. The axis to be split is chosen in the same way based on the ratio of the sides of the bounding box, but the point on which the axis is split is calculated as the median axis-value for all vertices in the mesh. This has the effect of dividing the geometry equally into the two child nodes.

3.3.4 Multiway KD-Tree

The Multiway KD-tree is an extension of the variable KD-tree which splits each node into N subnodes. This requires N-1 splitting points along the longest axis. The splitting points are chosen as equally spaced percentiles of the axis-values. For example if N equals 4, then the splitting points will be the first, second, and third quartile values.

3.4 Peformance Experiment Design

In order to evaluate the performance characteristics of the different space-partitioning structures we need to define a number of metrics for the final data structure that we can evaluate.

1. **Tree Balance:** A hierarchical tree is balanced if all of its leaf nodes exist at the same depth in the tree. This is important as the final hierarchical level-of-detail mesh should, optimally, be equally expandable in all areas. The balanced tree will also make preprocessing easier as the depth of the tree can be estimated before processing begins. We expect variants of the variable KD-tree to be well balanced while geometry-distribution independent structures such as the Octree to be very unbalanced.

2. **Geometry Distribution:** Another aspect of improving continuity and predictability is choosing a data structure that results in more distributed geometry within each level of the tree.

We can also measure resources consumed during the pre-processing stage by monitoring the computer system on which the pre-processing is run.

1. **Elapsed Time:** The elapsed time (in seconds) for processing a model is important as we want to determine whether relationships exist between the time

taken to process a model and the hierarchical tree structure used, tree-depth, or size of the initial model.

2. **Disk Usage:** Since that processing the mesh could require more system memory than is available on the test system, mesh parts will need to be stored on a persistent storage device such as a hard drive. We will measure the disk usage of the program in terms of files created and bytes written.

3. **Memory Usage:** As mentioned in the previous point, processing large models will consume large amounts of Random Access Memory (RAM). We need to measure this in order to determine what, if any, advantages there are between the hierarchical structures in terms of memory usage.

3.5 Summary

In this chapter we discussed the architecture of the overall system and of the mesh pre-processor component. We described the various subcomponents that make up the pre-processor and highlighted some of the more important algorithms.

We also defined the metrics we used to evaluate the different hierarchical structures.

4 | Implementation

In moving from our system design towards our final implementation, we need to turn our high-level algorithms into a working implementation that runs on our test hardware. This chapter focusses on detailing how the subcomponents of the system operate in order to perform their individual tasks as well as the overall algorithm and the structure of the final output file.

4.1 Algorithms

4.1.1 Recursive Preprocessing

One of the major problems of the hierarchical level of detail algorithm is the fact that any given node in the hierarchy cannot be generated before its immediate children have been generated. Thus it is tempting to perform all of the hierarchical splitting operations on the tree in order to generate all leaf nodes before beginning the process of stitching and simplification. The disadvantage of generating all leaf nodes is that each sub mesh must be stored on disk, effectively using the same amount of disk space as the original model. A more memory efficient method is to only generate the split sub-nodes when they are required. This turns the pre-processor into a completely recursive algorithm.

Algorithm 2 shows the general structure of the recursive algorithm which generates the final hierarchical tree. This algorithm is essentially a depth-first traversal of the hierarchical space-partitioning tree. The first mesh to be processed is the mesh that is returned by the mesh-cleaning component. Because the algorithm is intended to process very large models that may not fit in the memory available to the process, all mesh objects are passed as file handles rather than in memory objects. A mesh is only loaded or streamed into memory when it is actually being split, stitched, or simplified and the file is deleted as soon as possible.

Algorithm 2 Due to the recursive method call on line 13, the method call for the root node does not return until all descendant nodes have been processed. The functions Stitch, Simplify, and Split will be introduced in more detail in the following sections.

```
1: procedure PROCESS(mesh, maxDepth)
2:     PROCESS(mesh, 0, maxDepth)
3: end procedure
4:
5: (continued on next page ...)
```

```
6: procedure PROCESS(mesh, depth, maxDepth)
7:     if depth == maxDepth then
8:         return mesh
9:     else
10:        subMeshes ← SPLIT(mesh)
11:
12:        processedSubMeshes ← empty list
13:        for each m in subMeshes do
14:            temp ← PROCESS(m, depth + 1, maxDepth)
15:            processedSubMeshes.add(temp)
16:        end for
17:
18:        // stitch child nodes together
19:        stitchedMesh ← processedSubMeshes[0]
20:        for i ← 1 to processedSubMeshes.length do
21:            stitchedMesh ← STITCH(stitchedMesh, processedSubMeshes[i])
22:        end for
23:
24:        // simplify the result
25:        simplifiedMesh ← SIMPLIFY(stitchedMesh)
26:
27:        // attach child nodes to form the tree
28:        simplifiedMesh.children ← processedSubMeshes
29:        return simplifiedMesh
30:    end if
31: end procedure
```

4.1.2 Efficient Mesh Splitting

Due to the recursive nature of the pre-processor, the mesh-splitter is the only component, besides the mesh-cleaning component, that needs to perform an operation on the entirety of the original mesh. In order to keep this as memory efficient as possible, we designed and implemented a multi-pass streaming architecture to split the mesh into multiple sub-meshes.

The first step is to store which sub-node each vertex of the mesh belongs to. For this we use a bit array in order to save space. If the split creates N sub-nodes, then $\log_2 N$ bits are required to store a value from 0 to $N-1$ per vertex in order to indicate which of the sub-nodes the vertex falls in. If V is the number of vertices in the mesh, then $V\frac{\log_2 N}{8}$ bytes are required in total. For example: a KD-tree splitting mechanism (2 sub-nodes per node) for 10,000,000 vertices requires only 1.19 MB of memory to store the vertex membership array rather than 305.2 MB if 32-bit integers were used.

Once the vertex memberships have been calculated, the N sub-meshes can be generated. Each sub-mesh is generated in 2 passes: The first pass uses the membership array and creates the list of faces for the current submesh by including any faces that have at least 2 vertices in the current sub-node. It also marks any additional vertices that may be linked to the current node by faces that overlap boundaries. New vertex indices are generated and stored. The final list of triangular faces is written to a temporary file using the new vertex indices. The second pass generates the output model file in PLY format and writes the required vertices to it. It then appends the file containing face information and writes the final model file to the destination. This is repeated until all N sub-meshes have been generated.

4.1.3 Naïve Mesh Stitching

The premise of our mesh stitching algorithm is to combine multiple meshes and remove any duplicate vertices. Duplicate vertices are defined as vertices that have the same position in 3D space by having identical X, Y, and Z components. Algorithm 3 below details our approach. Given two meshes, $meshA$ and $meshB$, we want to

determine a mapping from the indices of $meshB$'s vertices to the indices of $meshA$'s vertices. We have 4 categories of vertices: 1, vertices that occur only in $meshA$; 2, vertices that occur only in $meshB$; and 3, vertices that are found in both $meshA$ and $meshB$. Category 1 and 2 vertices can be written straight to the destination file but

category 3 vertices must be written to the destination file only once. When writing the faces section, faces from $meshA$ can be written unmodified to the destination file but faces from $meshB$ require modification. Because certain vertices from $meshB$ are mapped to indices in $meshA$, this affects all other vertex indices referenced by $meshB$'s faces.

Algorithm 3

```
 1: procedure STITCH(meshA, meshB)
 2:     indexMap ← empty hash map
 3:
 4:     currentIndex ← 0
 5:     for each v in meshA.vertices do
 6:         indexMap[v] ← currentIndex
 7:         increment currentIndex
 8:     end for
 9:
10:     create temporary file vertexFile
11:     create temporary file faceFile
12:
13:     write meshA.vertices to vertexFile
14:     write meshA.faces to faceFile
15:     (continued on next page ...)
```

4.1.4 Simplifier

We use an adapted simplifier compiled from the VCGLib source code. VCGLib is an open source C++ computer graphics library developed by the Italian National Research Institute. We adapt the simplifier to allow for the specification of an explicit region in which simplification can occur. This allows the pre-processor to specify that only interior geometry of a node must be simplified and that exterior vertices must be maintained in order to aid stitching of the mesh. This is a form of boundary preservation.

The simplifier is compiled as a 64 bit binary executable for Windows and Linux and is executed from the pre-processor.

```
    meshBIndexTransform ← integer array
    for each v in meshB.vertices do
        if indexMap contains v then
            meshBIndexTransform[v] ← indexMap[v]
        else
            meshBIndexTransform[v] ← currentIndex
            increment currentIndex
            write v to vertexFile
        end if
    end for

    for each f in meshB.faces do
        f.vertex1 ← meshBIndexTransform[f.vertex1]
        f.vertex2 ← meshBIndexTransform[f.vertex2]
        f.vertex3 ← meshBIndexTransform[f.vertex3]
        write f to faceFile
    end for

    outputModel ← new mesh
    append vertices from vertexFile
    append faces from faceFile
    return outputModel
end procedure
```

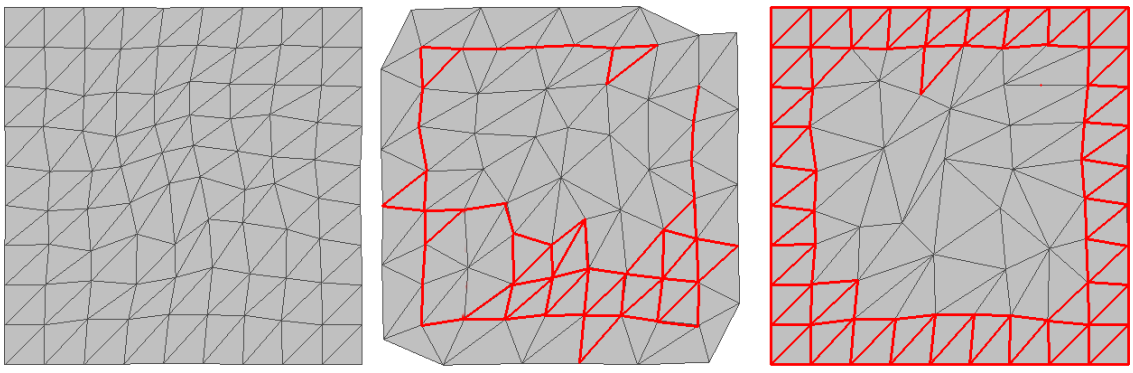


Figure 4.1: This image shows the effect of restricting simplification to interior edges of a mesh. The left-most image shows the original mesh containing 164 faces. The middle image shows the result of simplifying the mesh to 120 faces. The edges coloured red have been kept. The right-most image shows the same simplification but with our boundary simplification. As you can see, faces on the exterior of the mesh have been maintained which makes it much easier to stitch the mesh to its neighbours.

4.1.5 Calculating Tree Depth and Simplification Ratio

The depth of the hierarchical tree is critical for the performance of the system. When rendering the level of detail tree, more levels will result in less noticeable popping effects as new levels of detail are loaded, but will result in more disk read operations and take up more memory on both the system and the graphics hardware. We chose to use the number of faces as a measure of the amount of geometry, rather than the number of vertices since a vertex can be connected by many faces but a triangular face can only ever be connected to 3 vertices. For hierarchical tree structures, the tree depth can be managed directly by controlling the allowed number of faces per leaf node.

Balanced tree structures, such as the KD-Tree, have the property that their depth can be directly calculated given the number of faces in the original mesh and the number of faces allowed in the leaves. If the overall number of faces is F, the target number of faces allowed in each node is L, and each split produces k child nodes:

$$splits = \lceil \log_k \frac{F}{L} \rceil \tag{4.1}$$

where *splits* is the number of splits required to produce nodes that contain fewer than L faces. The number of splits can also be used to calculate the total number of leaf nodes in the KD-Tree:

$$leaves = k^{splits} \tag{4.2}$$

and thus the number of nodes in the entire tree:

$$nodes = k^{splits} + (k^{splits} - 1) \tag{4.3}$$

Once the tree has been built, stitching and simplification can occur. If a set of k child meshes each contain L faces, they are stitched to form a mesh with approximately Lk faces. We then simplify it by a ratio r to create a parent mesh of rLk faces. Initially we chose r such that all interior nodes in the tree also contained approximately L faces:

$$L = rLk \tag{4.4}$$

$$\therefore r = \frac{1}{k} \tag{4.5}$$

Because the leaves of the tree are representative of the original mesh, the final file size can be approximated:

$$final = L(nodes) \tag{4.6}$$
$$\therefore = L(k^{splits} + (k^{splits} - 1)) \tag{4.7}$$
$$\therefore = Lk^{splits} + Lk^{splits} - L \tag{4.8}$$
$$\therefore = original + original - L \tag{4.9}$$

Therefore, the final hierarchical mesh file size is approximately double the original mesh.

During development we found that the ratio $\frac{1}{k}$ was problematic. Although the resulting nodes all contained roughly L faces, the root node became over simplified to the point that the original geometric shape and silhouette was no longer recognisable. This was partly because boundary faces were not being simplified, so their eventual accumulation at the root node caused a high impact on the number of faces remaining in the interior of the mesh. (See section 4.1.4).

The first solution to this is to define a second constant R which sets the required amount of geometry in the root node. Since there are *splits* stages of simplification from a leaf node to the root, we can calculate an overall simplification ratio from which we can determine the simplification ratio to be used between nodes:

$$overall = \frac{R}{F} \tag{4.10}$$
$$\therefore = r^{splits} \tag{4.11}$$
$$\therefore r = \sqrt[splits]{overall} \tag{4.12}$$
$$\therefore r = \sqrt[splits]{\frac{R}{F}} \tag{4.13}$$

Unfortunately, the ratio returned by Equation 4.13 is larger than that returned by Equation 4.5 since R is larger than L. This causes less simplification in nodes below the root node and degrades performance considerably when the hierarchical model is being rendered in the viewer.

The solution to this is to allow more faces in nodes that are higher up the tree, while using more aggressive simplification in lower levels. We do this by scaling the

simplification ratio r depending on height within the tree. This scaling must still yield the same overall simplification in order to keep the final file size closer to double the file size (Equation 4.6). Since there are *splits* stages of simplification from a leaf node to the root, we define a function that returns the simplification ratio per level of the tree:

Algorithm 4

```
1: procedure GetRatio(F, R, splits, level)
2:     overallRatio ← R/F
3:     // v defines the range of variance allowed (between 0 and 1)
4:     v ← 0.5
5:     start ← 1/splits − v ∗ (1/splits)
6:     diff ← (v ∗ (1/splits) ∗ 2)/(splits − 1)
7:     f ← start + diff ∗ level
8:     return overallRatio ^ f
9: end procedure
```

For Example:
The given mesh contains 100 million faces. (F)
The root node may contain 400,000 faces. (R)
It is split using 4 levels (*splits*).
The splitting increments are: 0, 1, 2, 3, 4.

If a_l is the simplification ratio for level l then by applying the function *GetRatio*:
$a_0 \rightarrow 0.501484$
$a_1 \rightarrow 0.316540$
$a_2 \rightarrow 0.199802$
$a_3 \rightarrow 0.126116$

These have the same overall simplification ratio as $\frac{R}{F}$.

$$a_0 a_1 a_2 a_3 = \frac{R}{F} \tag{4.14}$$

$$a_0 a_1 a_2 a_3 = \frac{400000}{100000000} \tag{4.15}$$

$$0.04 = 0.04 \tag{4.16}$$

And since *GetRatio* is symmetric around $x = \frac{D-1}{2}$:

$$a_0 a_3 = a_1 a_2 \tag{4.17}$$

$$0.0632455 = 0.0632455 \tag{4.18}$$

Algorithm 4 is the final solution chosen for our project. During testing it provided good balance between conserving detail in nodes in the upper half of the tree, while reducing complexity in nodes in the lower half of the tree where there are many candidate triangles for simplification.

4.1.6 Percentile Algorithm

One of the downsides of testing variable KD-Tree variants is that they require dividing the vertices by geometric distribution into equally weighted portions using percentiles. For example the KD-Tree described in section 3.3.3 calculates the median, or p50 value, and creates 2 groups: the vertices located to the left of the median, and the vertices located on the right of the median. For larger order multiway KD-Trees, more percentile dividers are created so that the region is divided into k equally weighted regions.

The general algorithm for calculating percentiles is to sort the array of values and pick those that divide the array into equal sized portions. Unfortunately because our algorithm needs to be able to calculate a percentile of an array on the order of hundreds of millions of vertices, this naïve approach is very inefficient due to the sorting requirement as well as the need to store the entire sorted array in memory.

Our solution, Algorithm 5, trades accuracy for performance and uses an approximation algorithm based on a binary search. In this way all of the required operations can be performed as streaming algorithms which scan through the vertex collection

linearly, multiple times. This improves our time complexity from $O(n + n \log n)$ to $O(n)$ and saves memory at runtime.

Algorithm 5

```
1:  procedure GetPercentile(p, array)
2:
3:      maximumIterations ← 20
4:
5:      min ← Min(array)
6:      max ← Max(array)
7:      approximate ← (min + max)/2
8:
9:      for i ← 0 to maximumIterations do
10:         c ← count values smaller than approximate
11:         if c < (array.length * p) then
12:             max ← approximate
13:         else
14:             min ← approximate
15:         end if
16:         approximate ← (min + max)/2
17:     end for
18:
19:     return approximate
20: end procedure
```

The tradeoff is that in large vertex arrays *maximumIterations* may not be large enough to create an accurate approximation of the target percentile. We can improve the accuracy by increasing *maximumIterations* but this causes more linear traversals of the vertex array. In order to reduce this performance impact we only run through a sample of the vertices if there are more than 50 million vertices. We expect that by sampling every Nth vertex (where $N = \lfloor \frac{numVertices}{50\ 000\ 000} \rfloor + 1$) we still reach approximately the same result.

4.2 Output File Structure

The output file is stored in a custom file format we named Packaged Hierarchical File (PHF). A PHF file consists of a header followed by a number of data blocks where each data block contains the vertices and faces of a mesh. The file begins with a single 4 byte integer that stores the length of the header section. The header is a JavaScript Object Notation(JSON)[jso] formatted string that contains all of the information required to parse, and display the hierarchical mesh. JSON is a simple and standardised method of serialising simple data types and objects as a string in such a way that it is easy and straightforward to extract the same objects. See Figure A.1 in Appendix A for a detailed description of the format of the JSON string. The tree structure can be rebuilt by reading the *id* and *parent_id* fields of each node. The node with a blank *parent_id* is the root node of the tree. In order to allow direct lookup of a required data block, each node contains *data_offset* and *data_length* attributes. This means that the data block is *data_length* bytes long and begins at position *data_offset*. This means the bytes of the data block can be read without scanning linearly through the large file.

Each data block consists of 3 arrays:

1. Vertex Coordinates
2. Vertex Colour
3. Triangles

Each vertex coordinate consists of three 4 byte floating-point numbers: X, Y, Z. If the mesh contains N vertices, the array consists of $12N$ bytes where the first 12 bytes represent vertex 0, the second 12 bytes represent vertex 1, and so on. The vertex colour array is very similar. Each entry consists of 3 bytes, the first being the Red component, the second being the Green component, and the third being the Blue component. If the mesh contains N vertices, the array consists of $3N$ bytes. The vertex colour array is optional depending on the value of the *vertex_colour* field in the JSON header.

Each triangle in the list of triangles is represented as a list of 3 vertex indices. These are the 3 points of the triangle. Each vertex index is a 4 byte integer.

In total, if the mesh contains V vertices and F faces, it requires $15V + 12F$ bytes if *vertex_colour* information is included.

4.3 Graphical User Interface

We built a straightforward Graphical User Interface (GUI) in order to make the Preprocessor easier to run. The Zamani researchers use graphical display-based tools for the majority of their mesh processing, rather than command-line based tools, therefore developing a GUI to interact with our Preprocessor would be advantageous.

As Figure 4.2 shows, the GUI allows the user to pick the input and output files as well as a checkbox that allows the user to perform a Y-Z axis flip (As explained in 3.2.1). A console window in the middle of the window shows the current state of the Preprocessor and reports in detail any errors that occur. The progress bar containing the current progress and estimated remaining time is very useful.

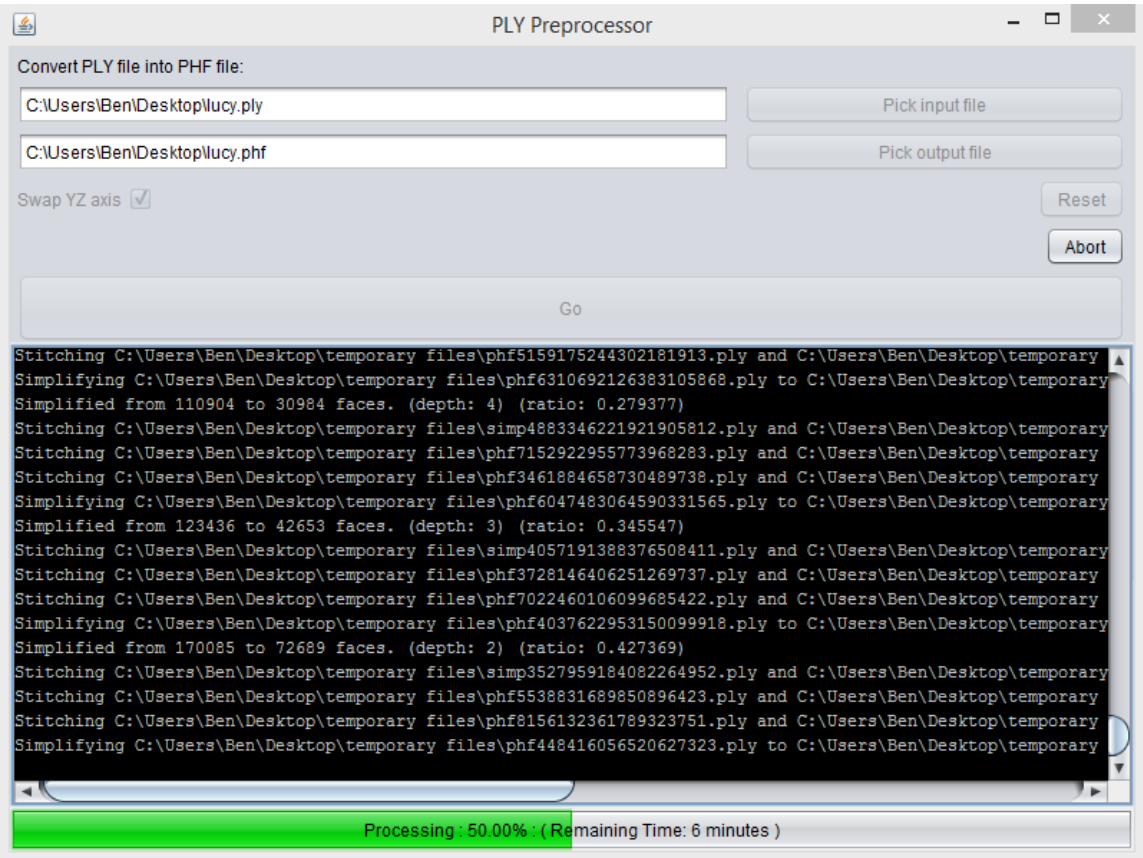


Figure 4.2

4.4 Viewing Component

Although the Viewer was not developed as part of this report, it is necessary to briefly describe its overall strategy. The viewing process begins once the user picks a PHF file to render. Once the file has been selected, the Viewer reads the header of the file in order to rebuild the tree of hierarchical nodes and to determine in which region of the file each data block lies. Once the tree has been rebuilt, the Viewer begins by loading and rendering the root node of the hierarchical mesh. This renders a coarse approximation of the mesh using approximately R faces. The root node is now an active node. While the active nodes are being rendered, the Viewer routinely checks whether the camera has been moved, if it has, new geometry may need to be loaded. The Viewer projects the bounding box of each active node onto the screen-space, and selects all of those which represent more 2 dimensional screen-space than some constant. The selected nodes are removed from the set of active nodes and replaced with their child nodes. The mesh for each new node is then loaded by a background thread and added to the geometry being rendered. A similar screen-space projection allows active nodes to be collapsed and replaced by their parent node if they take up less screen-space than some lower bound.

4.5 Tools Used

The following tools and software were used to develop our software:

Java Development Kit 8
: The software written during this project was written in the Java language and compiled by the Java Development Kit.

Linux and Microsoft Windows operating systems
: We developed and tested our software using both 64-bit Ubuntu Linux and Microsoft Windows 8.1. The software was designed to operate equally well on both operating systems.

Git Version Control
: Git was used as the version control software and all code repositories were hosted by Github.com. This made managing and distributing code between all of the developers very straightforward.

IntelliJ Community Edition
: We chose IntelliJ Community Edition as our development environment.

Trove Primitive Collections Library

We make use of high performance primitive collection objects in the Mesh Splitter and Mesh Stitcher in order to avoid allocation too many individual objects. The Trove library has a Lesser GNU Public License.

Apache Commons CLI Library

We used the Commons CLI Java library, created by the Apache Software Foundation, to build our command line interface. This library is provided under the Apache License Version 2.0.

5 | Testing

As mentioned in Section 3.4, there are a number of measurable outcomes from the Preprocessor that we can review. Each different hierarchical space-partitioning algorithm creates a different structure and uses different heuristics when building the output file. We decided to test 6 different hierarchical tree algorithms: the Octree, the traditional KD-Tree, the Variable KD-Tree, and 3 different Multiway KD-Trees. We test 3 different Multiway KD-Tree's since the splitting degree effects the height of the tree and the severity of simplification within it. Table 5.1 shows a brief summary of the structures and shows the number of child regions created per parent region as well as whether they split depending on he distribution of the content.

Table 5.1: Summary of Hierarchical Structures to be tested

#	Hierarchical Structure	Nodes per split	Geometry Distribution Dependent
1	Octree	8	No
2	KD-Tree	2	No
3	Variable KD-Tree	2	Yes
4	Multiway KD-Tree	3	Yes
5	Multiway KD-Tree	4	Yes
6	Multiway KD-Tree	5	Yes

This table shows a summary of the hierarchical structures we will be testing. Only the Variable and Multiway KD-Tree's split a mesh based on the distribution of geometry within the mesh itself.

5.1 Test Models

We used a sample of different input models in order to test our Preprocessor. All models were in a standard PLY format and spanned a large range of sizes. Two versions of the Jago model were provided with different vertex densities. See Table 5.2 for more detail.

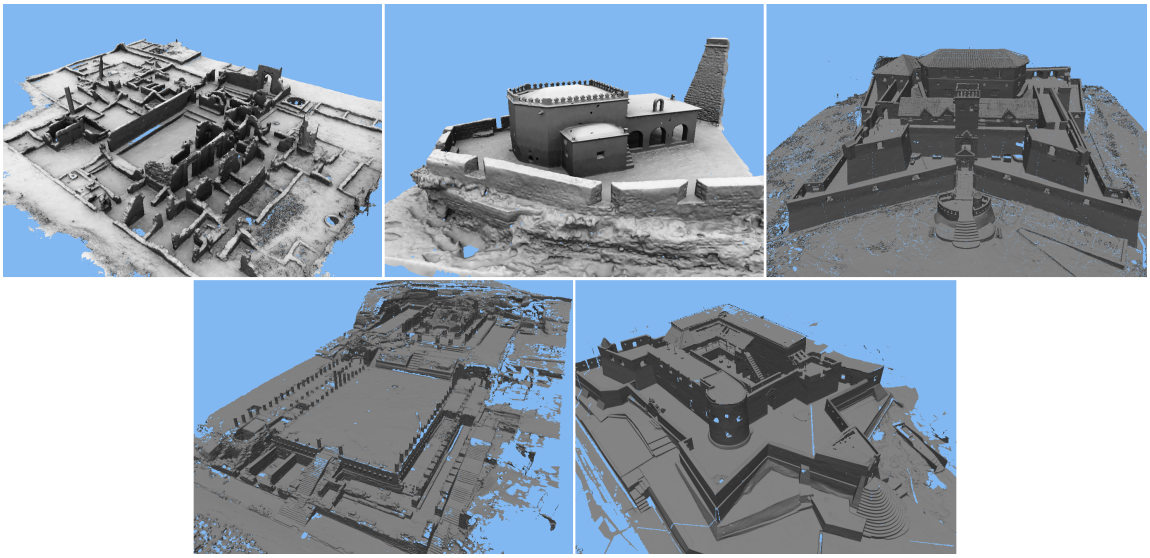


Figure 5.1: *This image shows screenshots of the test models. From top-left to bottom-right: GedePalace.ply (with colour data), ChapelOfNossa.ply (with colour data), Jago.ply, GreatTemple.ply, ShamaFort.ply*

Table 5.2: Testing Models

#	Model Name	Number of Vertices	Number of Faces	File Size (bytes)
1	GedePalace.ply	1 550 318	3 000 000	70 006 670
2	ChapelOfNossa.ply	4 727 306	9 452 066	217 423 288
3	Jago.ply	17 634 229	33 568 441	648 000 689
4	GreatTemple.ply	17 670 669	33 936 243	653 219 395
6	Jago.ply (4cm)	42 120 678	80 177 058	1 547 750 574
5	ShamaFort.ply	40 757 949	80 875 128	1 540 472 241

5.2 Testing Procedure

During testing we ran each of the 6 models in table 5.2 through each of the 6 structures in Table 5.1 yielding 36 test runs. We average the measurements across multiple test runs where possible as very large models may take a prohibitively long time to process. We will maintain the same constants throughout the tests and kept the root

node geometry set at 400 000 faces and the leaf node maximum (L) set at 100 000. For each test run, we will measure the following aspects in the following ways:

5.2.1 Tree Balance

We measure Tree Balance by recording the depths of each of the leaf nodes in the tree. We calculate a 3 number summary of the depths (min, mean, and max). For a completely balanced tree all values in the 3 number summary will be equal since all leaves are at the same depth. In an unbalanced tree, these values will be fairly distributed and will give an indication of where in the tree the leaf nodes lie.

5.2.2 Elapsed Time

Elapsed time is an important aspect of Preprocessing a mesh, although it is not a measure of quality of the output file. We would like to determine whether a relationship exists between the amount of geometry in the input file, and the elapsed time.

5.2.3 Geometry Distribution

Both balanced and unbalanced tree structures split the mesh until all leaf nodes contain fewer faces than some constant L, in this case 100 000 (See section 4.1.5). Ideally, all leaf nodes contain similar amounts of detail. For the Variable KD Tree and Multiway KD Tree, which split a mesh using percentiles, nodes will necessarily be similarly sized but the Octree and traditional KD Tree do not use this heuristic and therefore have fairly dissimilar mesh sizes. We measure the mesh distribution by measuring the mean and standard deviation of the number of faces and vertices per leaf node. We also measure the difference between the mean and the expected number of faces per node.

5.2.4 Disk Usage

While splitting and stitching meshes, temporary data must be stored on disk rather than in RAM. We need to choose a Preprocessing algorithm that uses as little disk storage as possible. Although temporary files are deleted a soon as they are no longer required, the recursive nature of the Preprocessor creates situations where many temporary files may be stored at a single moment. We measured the maximum

amount of bytes used by temporary files at any instant in the processing. This was important as it showed how much disk space the algorithm required at a minimum in order to produce the output file. We also generate a profile showing at which point the temporary files were allocated as well as how much disk space they used.

5.2.5 Memory Usage

Due to the fact that our software was written in Java, we used built in Java functionality to measure the memory usage of the Java thread. To do this, we launch a parallel thread which measures the size of the Java heap space every x milliseconds. Once the main thread exits, we calculate the 50th, 75th, and 100th percentile of the memory recordings in order to determine the trend. The reason we calculate multiple percentiles is due to the fact that the Java garbage collector is not active all the time but rather is activated only when usage hits a certain threshold or when the process is running out of heap space. With this in mind, we want to identify trends in memory usage rather than simply the maximum used. We also generate a profile of the memory usage throughout the processing in order to identify which parts of the process consume the most memory. It should be noted that simplification does not factor into memory usage as it occurs in a separate process that is not the concern of this report.

During testing it was shown that a linear relationship exists between the number of faces in an input model and the memory use incurred when simplifying it. Since the simplifier will only be run on meshes that contain below 400000 faces, this memory use will not be affected by the size of the model being processed or the hierarchical data structure being used.

5.3 Test System

All of our tests were run on two identical computers with a standard set of hardware:

- Intel Core i5-650 CPU (dual core @ 3.2GHz)
- 8GB RAM
- 500 GB Harddrive (5400 rpm)
- Ubuntu 12.10 64-bit Operating System
- Java 64-bit v1.8 (build 25.0-b66)

6 | Results & Discussion

We tested the performance of the Preprocessor by running the Preprocessor on each of the 6 models using each of the 6 hierarchical structures as described in Chapter 5. We ran 4 testing runs for each combination of model and hierarchical structure for the smaller models (GedePalace.ply, ChapelOfNossa.ply, Jago.ply, and GreatTemple.ply) and averaged the results. The data tables for all of the tests can be found in Appendix B.

6.1 Leaf node depth distribution

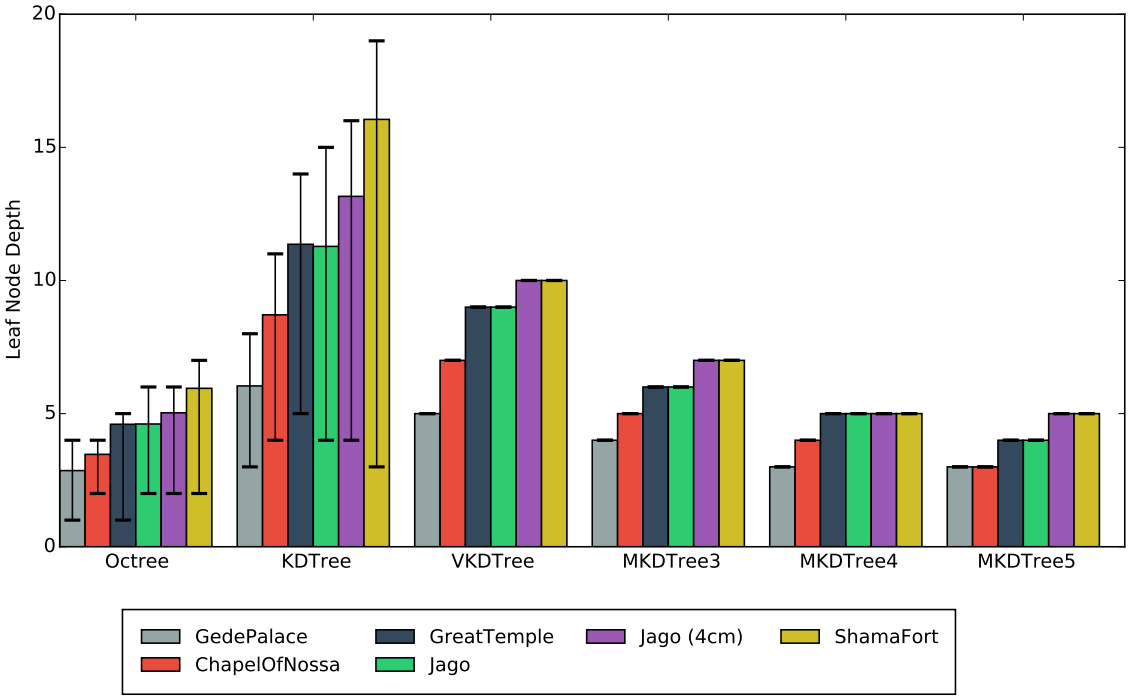
Figure 6.1 shows the range of the leaf node depths for each test. As predicted, the balanced tree structures have all of their leaf nodes at a single depth while the Octree and KD Tree have leaf nodes spread further throughout the tree.

6.2 Effect of hierarchical structure on Elapsed Time

Due to the fact that the hierarchical structure chosen influences the number of splitting, stitching, and simplification operations performed during processing, we expected the overall processing time of each model to change depending on which hierarchical structure was used. Particularly we expected differences between the geometry-distribution independent (Octree and KD-Tree) and geometry-distribution dependent hierarchies (Variable KD-Tree, and Multiway KD-Trees).

Table 6.1 show the mean elapsed times that resulted from our test runs. The Octree structure has the lowest elapsed time across all of the tests, while the remaining 5 structures take relatively similar amounts of time with no immediately obvious trends.

Figure 6.1: Graph showing range of leaf node depths per test



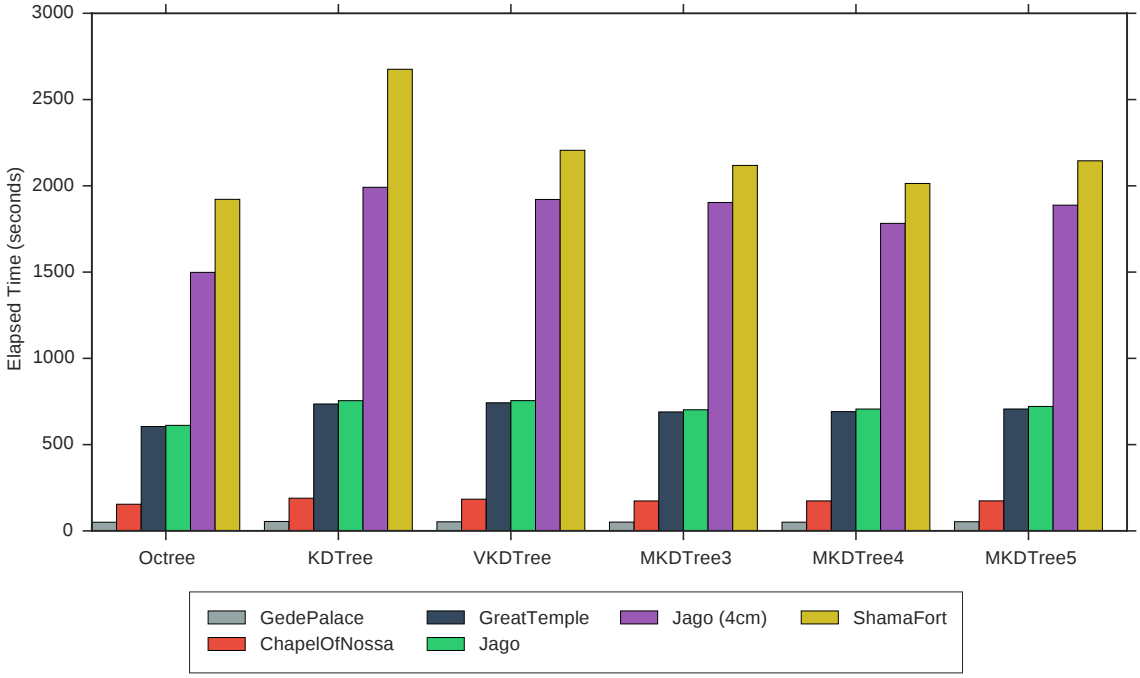
The graph above shows the mean and range of leaf node depths for each of the 6 models using the 6 hierarchical structures. The error bars on the Octree and KD Tree indicate the range of the depth of the leaf nodes.

Table 6.1: Mean Elapsed Time per model per hierarchical structure

	Elapsed Time (seconds)					
Model Name	**Octree**	**KDTree**	**VKDTree**	**MKDTree 3**	**MKDTree 4**	**MKDTree 5**
GedePalace.ply	50.28 ± 0.64	54.56 ± 0.22	52.53 ± 0.29	50.99 ± 0.34	50.51 ± 0.32	53.11 ± 0.62
ChapelOfNossa.ply	154.50 ± 0.43	189.50 ± 0.75	183.75 ± 0.41	173.50 ± 0.75	173.75 ± 0.65	174.00 ± 0.94
Jago.ply	611.75 ± 4.63	754.75 ± 5.70	755.00 ± 4.68	702.00 ± 2.47	706.25 ± 3.25	721.33 ± 1.91
GreatTemple.ply	605.00 ± 4.49	735.50 ± 3.83	742.50 ± 5.18	689.50 ± 0.25	691.25 ± 4.02	706.50 ± 1.48
ShamaFort.ply	1921.67 ± 33.78	2675.67 ± 23.10	2206.00 ± 29.63	2118.33 ± 26.80	2013.67 ± 0.27	2145.00 ± 15.28
Jago.ply (4cm)	1498.50 ± 11.67	1991.50 ± 8.84	1921.00 ± 29.70	1803.50 ± 31.47	1782.50 ± 37.12	1888.00 ± 39.60

This table shows the mean time taken to process each model using the specified hierarchical structure. Values including a standard error metric (indicated by the $\pm$) represent the mean value across 4 tests.

Figure 6.2: Graph of elapsed time per structure for each model

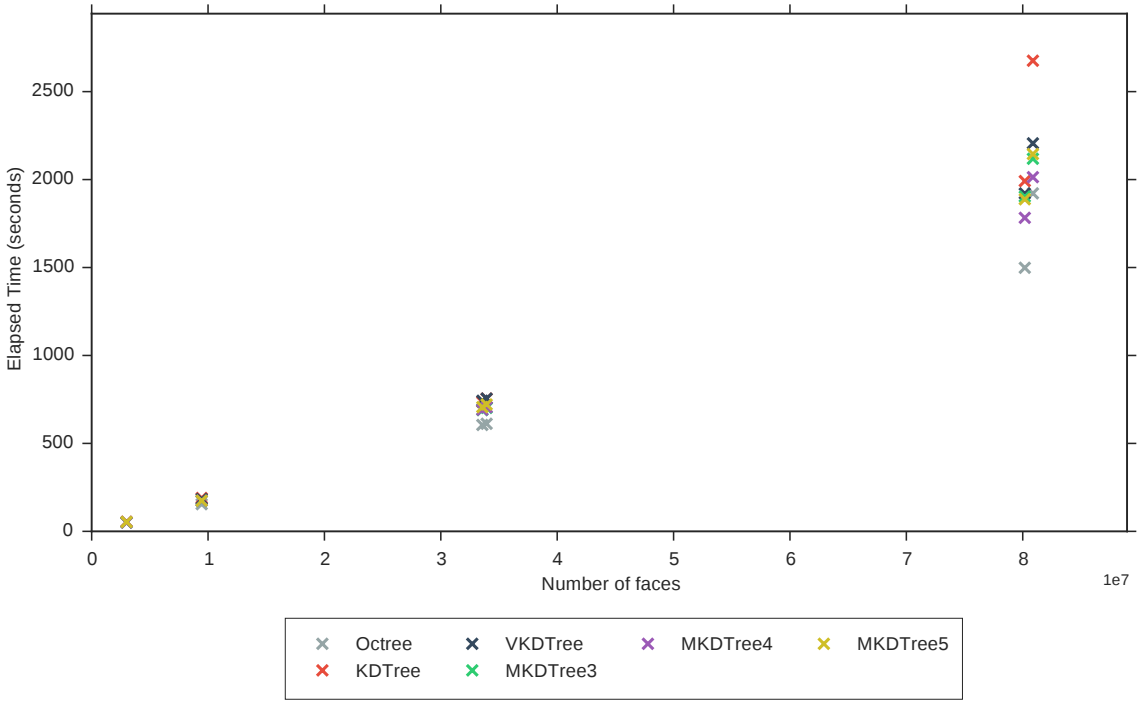


The graph above shows the time taken to process each of the 6 models using the 6 hierarchical structures. This is the same data shown in Table 6.1.

Correlation between Elapsed Time and Number of Faces

We investigated further by calculating the correlation between the number of faces with the elapsed time. By plotting the elapsed time against the number of faces for each test we arrived at the scatter plot shown in Figure 6.3. We calculated a correlation coefficient for each hierarchical structure using the Pearson product-moment correlation coefficient method in order to determine the strength of the linear relationship that exists for the structure (Table 6.2). We also performed a linear regression in order to determine the slope of the resulting best-fit lines.

Figure 6.3: Graph of elapsed time vs number of faces



The graph above shows the same data as Figure 6.2 as a scatter plot. The line of best fit, shown as a dotted line, shows the estimated linear relationship between number of faces and elapsed time. The vertical variation in elapsed time for each model is presumed to be caused by changes in the geometry distribution (See Section 6.3).

Table 6.2: Correlation coefficient and slope for elapsed time vs number of faces using each structure

Structure	**Coefficient**	**Slope** (seconds per face)	**Faces per second** $\frac{1}{Slope}$
Octree	0.9830	0.0000218 ± 0.00000204	45776.26
KDTree	0.9747	0.0000302 ± 0.00000347	33093.93
VKDTree	0.9941	0.0000264 ± 0.00000144	37906.36
MKDTree3	0.9948	0.0000258 ± 0.00000132	38740.21
MKDTree4	0.9953	0.0000242 ± 0.00000117	41320.08
MKDTree5	0.9942	0.0000258 ± 0.00000140	38703.73

This table shows the Pearson correlation coefficient as well as the slope of the line for elapsed time vs number of faces. The balanced tree structures, (shown in italic), have a coefficient closer to 1 than the unbalanced structures suggesting a stronger linear relationship between the number of faces and their performance although the Octree has highest processing speed. The $\pm$ indicates the standard error metric of the linear regression.

Our results showed that although all 6 of the hierarchical structures tested showed very strong linear relationships between input geometry and elapsed time, the Octree showed better overall processing speed. The fastest balanced hierarchy was the Multiway KD Tree of order 4 (4 child nodes per node) but the slope values for the balanced hierarchies all fell within the range of their standard errors. More data is required in order to determine which, if any, of the balanced structures have the fastest processing speed overall.

6.3 Effect of hierarchical structure on Geometry Distribution

As described in Section 5.2.3, we measure geometry distribution by measuring the mean and standard deviation of faces and vertices per leaf node. Table 6.3 shows the values for faces per leaf node while Table 6.4 shows vertices per leaf node.

Table 6.3: Mean number of faces per leaf node for each model using each structure

	Faces per leaf node (mean ± stddev)					
Model Name	**Octree**	**KDTree**	**VKDTree**	**MKDTree 3**	**MKDTree 4**	**MKDTree 5**
GedePalace.ply	26084 ± 20921	62500 ± 26422	93750 ± 1761	37037 ± 1003	46875 ± 1145	23999 ± 751
ChapelOfNossa.ply	28129 ± 21966	57634 ± 25629	73844 ± 514	38897 ± 374	36922 ± 406	75616 ± 262
Jago.ply	26007 ± 24664	57938 ± 25449	65708 ± 3067	46114 ± 2186	32821 ± 1545	53761 ± 2282
GreatTemple.ply	25892 ± 22449	58979 ± 25141	66428 ± 2802	46620 ± 2048	33181 ± 1555	54351 ± 2325
ShamaFort.ply	26111 ± 22744	57012 ± 24983	92924 ± 2561	43509 ± 1349	92924 ± 2599	30449 ± 993
Jago.ply (4cm)	29345 ± 23847	57973 ± 25899	78297 ± 4193	36660 ± 2384	78297 ± 4270	25654 ± 1821

This table shows the mean and standard deviation for the number of faces per leaf node in the output when using each structure against each model.

Table 6.4: Mean number of vertices per leaf node for each model using each structure

	Vertices per leaf node (mean ± stddev)					
Model Name	**Octree**	**KDTree**	**VKDTree**	**MKDTree 3**	**MKDTree 4**	**MKDTree 5**
GedePalace.ply	14130 ± 11212	32797 ± 13637	49091 ± 101	19662 ± 420	24758 ± 163	13022 ± 633
ChapelOfNossa.ply	14513 ± 11177	29444 ± 12998	37943 ± 360	20131 ± 225	19245 ± 296	39077 ± 776
Jago.ply	14215 ± 13257	30996 ± 13630	36230 ± 1292	25771 ± 1403	18513 ± 1147	30107 ± 2030
GreatTemple.ply	13915 ± 11978	31203 ± 13301	35647 ± 574	25204 ± 521	18147 ± 499	29495 ± 701
ShamaFort.ply	13842 ± 11847	29525 ± 12818	48166 ± 370	22786 ± 316	48182 ± 449	16045 ± 298
Jago.ply (4cm)	16019 ± 12961	31296 ± 13980	43055 ± 1578	20601 ± 1326	43376 ± 2304	14666 ± 1246

This table shows the mean and standard deviation for the number of vertices per leaf node in the output when using each structure against each model.

Tables 6.3 and 6.4 show how the unbalanced structures, the Octree and KDTree, have high deviations in the number of faces and vertices per leaf. This shows the impact of splitting the mesh without taking into account geometry distribution and simply slicing the mesh into halves. The balanced structures that use a geometry distribution dependent heuristic have much lower standard deviations. The standard deviations in vertex counts are lower than those for face counts because the mesh splitter algorithm, described in Section 4.1.6, uses the vertices of the mesh in order to determine geometry distribution. A perfect percentile algorithm would results in very low standard deviations but would cause a large performance impact and thus it is more desirable to allow a small amount of error in this calculation. We also expect larger models to have larger variance in their leaf sizes since more levels of splitting cause a larger cumulative error. This can also be seen in Table 6.3. It is

more desirable to have more similar sized leaves since this makes disk reads and mesh loading more predictable when viewing the model.

Due to the way in which we calculate the number of levels to build for balanced trees (Section 4.1.5), the sizes of the actual leaf nodes are very dependent on the number of faces and the tree structure used. We can calculate the expected amount of faces per leaf node by calculating the number of leaf nodes and dividing the total number of faces.

Let F be the total number of faces and
let L be the maximum number of faces per leaf node
K be the number of nodes created with each split:

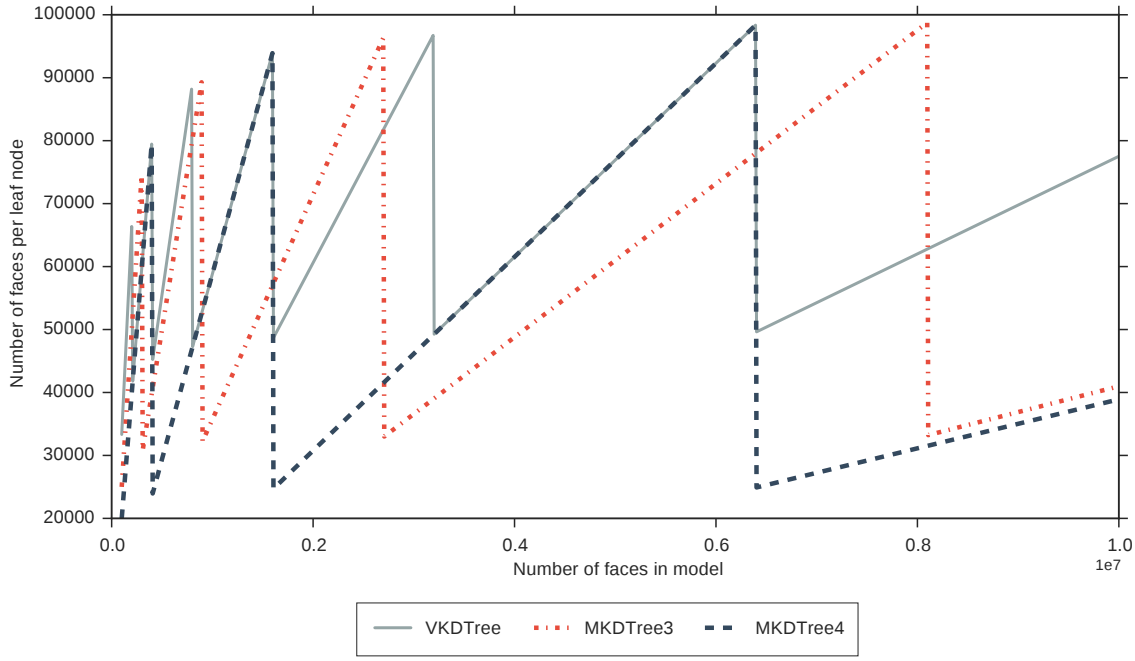
$$levels = \log_K \frac{F}{L} + 1$$

Then p is the expected faces per leaf:

$$p = \frac{F}{K^{levels} + 1}$$

Figure 6.4 shows how this function follows a sawtooth pattern for several values of K. It shows that the actual number of faces per leaf node is often much lower than the target L that the stride between peeks, where the number of leaf nodes is optimal, is larger proportional to K. If the goal is to drive p to be as close as possible to L it will be more effective to pick an optimum K value based on F rather than using a predetermined constant. This is countered by the fact that if K is too high, the tree will have a lower depth ($levels$) which will require more aggressive simplification between levels and may cause visual artefacts when viewed in the viewing component.

Figure 6.4: Graph of original model size vs expected faces per leaf node



In summary, the unbalanced hierarchical structures are unsuitable as a space-partitioning structure for Hierarchical Level of Detail implementations since the variation in the amount of geometry in the leaf nodes is too high allowing some nodes to have very few triangles while other nodes have close the maximum L faces. Balanced tree structures have much smaller ranges of variation but the variation does increase as the tree becomes deeper because of errors introduced by the percentile approximation algorithm.

6.4 Disk Usage

Disk usage was important to measure since a large amount of temporary files are created during processing of a model. Temporary files are generated during all stages of the processor and due to the recursive nature of the preprocessor, are stored until they are stitched to neighbouring nodes in the hierarchy and simplified, or until the outputfile is created.

Table 6.5: Table showing maximum disk usage for each model and hierarchical structure

	Maximum disk usage (megabytes)					
Model Name	**Octree**	**KDTree**	**VKDTree**	**MKDTree 3**	**MKDTree 4**	**MKDTree 5**
GedePalace.ply	290.67	356.49	282.17	318.84	270.62	277.42
ChapelOfNossa.ply	786.94	1 072.49	821.99	872.78	785.97	684.46
Jago.ply	2 628.85	3445.59	2 633.19	2 683.68	2 503.79	2 277.40
GreatTemple.ply	2 593.83	3 407.91	2 618.96	2 668.94	2 495.38	2 274.09
ShamaFort.ply	7 530.23	10 450.42	6 699.96	7 138.86	6 182.93	6 339.19
Jago.ply (4cm)	5 802.59	7 882.37	5 917.65	6 302.69	5 416.46	5 582.47

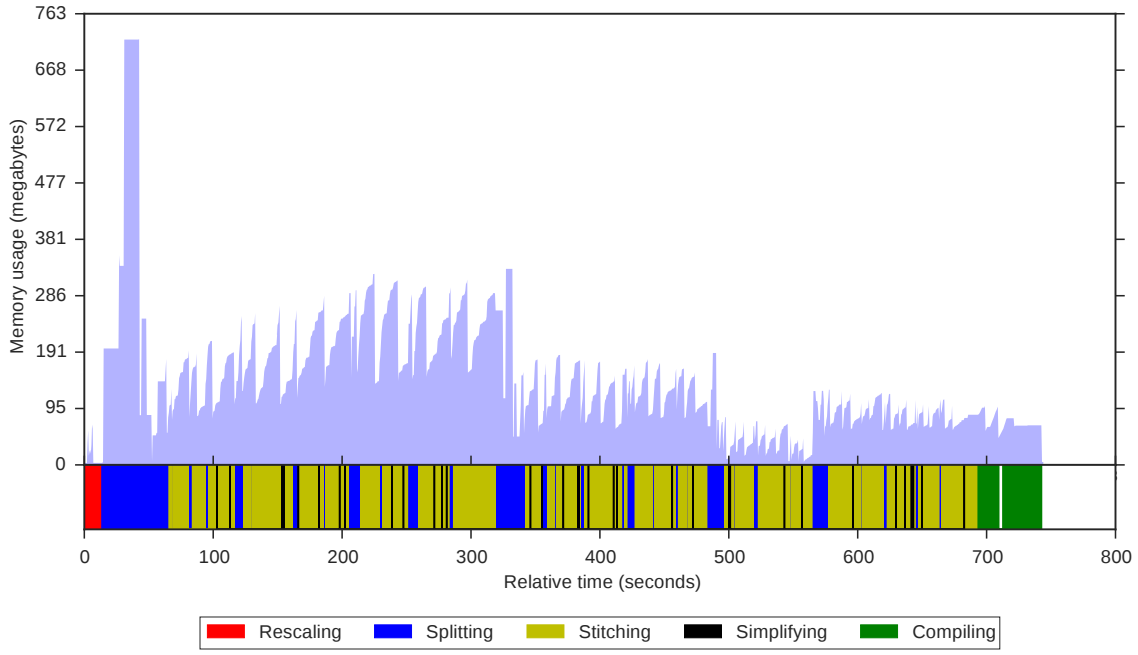
This table shows the maximum disk usage used during processing. No mean error metric or standard deviation is provided since the values are always the same when the same structure is used for the same model since the use of temporary files is completely deterministic.

6.5 Memory Usage

Once our test results were gathered and aggregated we found that memory usage was not a useful method of comparing the performance of the 6 hierarchical structures. The standard deviations for the values were too high to yield any useful relationships. We decided that this was likely due to the unpredictable actions of the Java garbage collector. The Java garbage collector uses a mark-and-sweep strategy that is triggered by multiple heap space usage heuristics. The result of this is that the measured memory usage was no indicative of the real memory usage of the data structures being used. Figure 6.5 shows a memory profile of a Preprocessing test. The large vertical drops in memory usage show where the garbage collector has been triggered. We also see that the first mesh split of the root node, uses the most memory throughout the processing.

It would have been more suitable to manually calculate the byte usage of critical data structures used during preprocessing and to seek to optimise the size of these data structures. The data usage statistics can be found in the tables in Appendix B.

Figure 6.5: Graph of memory usage when processing GreatTemple.ply with a KD Tree



The memory profile above shows the Java Heap Space usage over time while processing the GreatTemple.ply model using a KD Tree hierarchical structure. The coloured regions below the memory profile indicate the state of the Preprocessor at that point.

6.6 Summary

In summary, we found that a strong linear relationship exists between elapsed time and the size of the model for all of the hierarchical structures tested. We found that the balanced hierarchical structures are more suitable for Hierarchical Level of Detail due to the better distribution of geometry throughout their leaf nodes.

7 | Conclusions

In summary, we set out to investigate the performance differences between a variety of hierarchical space partitioning trees in the context of building a Hierarchical Level of Detail mesh. We tested two unbalanced structures: an Octree and a KD-Tree, as well as 4 balanced structures: a Variable KD-Tree, Multiway KD-Tree with 3 children per node, Multiway KD-Tree with 4 children per node and Multiway KD-Tree with 5 children per node. We identified the fact that unbalanced trees are unsuitable for our implementation of a Hierarchical Level of Detail scheme due to the uneven distribution of geometry in the leaf nodes while balanced trees have much better distribution. Our testing showed that although the balanced tree structures have a more even distribution of geometry in the leaf nodes of the tree, care must be taken to select the depth of the tree in order to arrive close to a target number of faces or vertices per leaf node. Strong linear relationships were identified between the size of the model being processed and the elapsed time, for all of the hierarchical structures tested and that the time to process a model can be approximated in advance of preprocessing.

We answered our research questions by showing that the Hierarchical Level of Detail structures were able to be generated using all of our test models with no problems due to the size of the models. We also confirmed that balanced trees are more suitable than unbalanced trees from the perspective of geometry distribution.

7.0.1 Future Work

Further work on the Preprocessing component developed in this report would focus on the following areas:

View-dependent performance analysis: Although this report tested many attributes to do with the final file structure and resources used during preprocessing, the hierarchical structures were not tested in the Renderer component itself. Further investigation could be done to determine whether the number of child nodes per node has a noticeable impact on performance when rendering the hierarchical model.

Better selection of tree structure and depth: As shown in this report, there is no single balanced tree structure that performs the best for all sizes of models.

The algorithm described in Section **??** should be improved using the function shown in Section **??** in order to determine the best tree structure and depth to be used for a model. By tuning the number of child nodes per node as well as the tree depth, the average amount of geometry per leaf node should be able to be tightly controlled.

Multi-threaded preprocessor: The recursive Preprocessor shown in the implementation chapter performs well on hardware with only a single slow storage device and uses only a single thread. The branching methodology of the Preprocessor can be easily reimplemented using multiple threads to generate the hierarchical tree. If multiple storage devices are available, this could allow the Preprocessor to run much faster.

Appendices

A | Extra Figures

```
{
    "max_depth": integer (maximum depth of the tree)
    "vertex_colour": boolean (whether or not the vertices contain colour information)
    "nodes": [
        {
            "id": integer (id of the node)
            "parent_id": integer (id of the parent node)
            "num_faces": integer (number of faces in the node)
            "num_vertices": integer (number of vertices in the node)
            "block_offset": integer (position in the file of the data block)
            "block_length": integer (length of the data block)
            "min_x": float (minimum X value in the mesh node)
            "min_y": float (minimum Y value in the mesh node)
            "min_z": float (minimum Z value in the mesh node)
            "max_x": float (maximum X value in the mesh node)
            "max_y": float (maximum Y value in the mesh node)
            "max_z": float (maximum Z value in the mesh node)
        },
        ...
    ]
}
```

Figure A.1: *This image shows the effect of restricting simplification to interior edges of a mesh. The left-most image shows the original mesh containing 164 faces. The middle image shows the result of simplifying the mesh to 120 faces. The edges coloured red have been kept. The right-most image shows the same simplification but with our boundary simplification. As you can see, faces on the exterior of the mesh have been maintained and make it much easier to stitch the mesh to its neighbours.*

B | Data Tables

This appendix contains much of the raw data used to generate the tables and graphs in this report. Due to the volume of data, they were not included inline in Chapter 6 but rather listed here. Performance data tables are averaged over several runs and values are displayed with the standard error metric indicating the standard deviation of the sample mean.

Table B.1: GedePalace.ply Performance Data

Hierarchy	Elapsed Time (seconds)	RAM p50 (bytes)	RAM p75 (bytes)	RAM p100 (bytes)	runs
Octree	50.28 ± 0.64	60623062 ± 7307225.28	66291946 ± 9489882.53	80016330 ± 13298863.39	4
KDTree	54.56 ± 0.22	40766009 ± 12560736.73	74210356 ± 13234405.73	110257576 ± 3306353.00	4
VKDTree	52.53 ± 0.29	28697654 ± 1213283.21	35146146 ± 1798312.19	48979454 ± 2805028.08	4
MKDTree3	50.99 ± 0.34	42004260 ± 9885434.29	50539452 ± 8680567.90	63503940 ± 7817098.29	4
MKDTree4	50.51 ± 0.32	24856294 ± 1079418.30	30952312 ± 2245576.29	41148492 ± 3536768.93	4
MKDTree5	53.11 ± 0.62	26050601 ± 3356993.57	34545616 ± 4503090.67	44422396 ± 6409371.05	4

Table B.2: GedePalace.ply Disk Data

Hierarchy	Output File Size (bytes)	Total Temporary Bytes Written	Maximum Temporary Bytes Stored
Octree	105088780	1270572446	304792310
KDTree	180009685	1367210125	373811656
VKDTree	142477202	1100130969	295873819
MKDTree3	121648182	1072633148	334326326
MKDTree4	100562128	966582394	283765889
MKDTree5	101314577	1081550023	290890907

Table B.3: GedePalace.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	136	115	1	2.86	4	5	26084	79662	20921	8	14130	42677	11212
KDTree	96	48	3	6.04	8	1070	62500	97376	26422	654	32797	50299	13637
VKDTree	63	32	5	5.00	5	89609	93750	97907	1761	48989	49091	49361	101
MKDTree3	121	81	4	4.00	4	34533	37037	40030	1003	19410	19662	22268	420
MKDTree4	85	64	3	3.00	3	43270	46875	50001	1145	24469	24758	25271	163
MKDTree5	156	125	3	3.00	3	19596	23999	26386	751	12559	13022	16257	633

Table B.4: ChapelOfNossa.ply Performance Data

Hierarchy	**Elapsed Time (seconds)**	**RAM p50 (bytes)**	**RAM p75 (bytes)**	**RAM p100 (bytes)**	**runs**
Octree	154.50 ± 0.43	34334236 ± 8376429.70	49692822 ± 13259267.44	$168340462 \pm 29748140.88$	4
KDTree	189.50 ± 0.75	71352790 ± 9017754.49	90583098 ± 10816303.67	172218506 ± 9437326.48	4
VKDTree	183.75 ± 0.41	90144497 ± 17443480.81	$104603474 \pm 18287513.23$	$130318586 \pm 17189860.93$	4
MKDTree3	173.50 ± 0.75	87588494 ± 11411504.48	$104564106 \pm 12660908.25$	$167959020 \pm 16198757.97$	4
MKDTree4	173.75 ± 0.65	41220847 ± 1448037.57	60785836 ± 1545193.43	84605008 ± 2379091.36	4
MKDTree5	174.00 ± 0.94	81167015 ± 2909342.18	98387858 ± 3667047.06	124100594 ± 5036143.58	4

Table B.5: ChapelOfNossa.ply Disk Data

Hierarchy	**Output File Size (bytes)**	**Total Temporary Bytes Written**	**Maximum Temporary Bytes Stored**
Octree	278707889	3845463594	825163592
KDTree	541540121	5021511831	1124585309
VKDTree	415053582	3991956635	861914357
MKDTree3	329457020	3448573703	915172827
MKDTree4	287885653	3242537719	824148518
MKDTree5	246529035	2916174929	717708658

Table B.6: ChapelOfNossa.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	395	336	2	3.47	4	6	28129	98904	21966	13	14513	50838	11177
KDTree	352	164	4	8.71	11	6	57634	99744	25629	9	29444	50886	12998
VKDTree	255	128	7	7.00	7	71029	73844	75871	514	37499	37943	39406	360
MKDTree3	364	243	5	5.00	5	37578	38897	39646	374	19746	20131	21194	225
MKDTree4	341	256	4	4.00	4	34388	36922	37933	406	18705	19245	20207	296
MKDTree5	156	125	3	3.00	3	74728	75616	76162	262	38111	39077	43204	776

Table B.7: Jago.ply Performance Data

Hierarchy	**Elapsed Time (seconds)**	**RAM p50 (bytes)**	**RAM p75 (bytes)**	**RAM p100 (bytes)**	**runs**
Octree	611.75 ± 4.63	71065700 ± 2589437.94	90803666 ± 3797049.18	$672016670 \pm 116036610.66$	4
KDTree	754.75 ± 5.70	173557416 ± 9638082.72	$237294538 \pm 12677585.12$	$753728278 \pm 51813679.26$	4
VKDTree	755.00 ± 4.68	331674064 ± 7291759.96	$428762736 \pm 17507816.44$	$588287328 \pm 36015184.88$	4
MKDTree3	702.00 ± 2.47	$271609166 \pm 62841282.98$	$322237988 \pm 71739719.78$	$551785162 \pm 48024612.29$	4
MKDTree4	706.25 ± 3.25	$192079357 \pm 55421044.95$	$215701784 \pm 57649124.70$	$315708404 \pm 63546215.57$	4
MKDTree5	721.33 ± 1.91	$217552856 \pm 97047961.77$	$252037200 \pm 99498281.88$	$463507920 \pm 62066047.03$	3

Table B.8: Jago.ply Disk Data

Hierarchy	**Output File Size (bytes)**	**Total Temporary Bytes Written**	**Maximum Temporary Bytes Stored**
Octree	998247120	13582939198	2756548469
KDTree	1835163585	19140022837	3612963885
VKDTree	1403723029	15245037286	2761101781
MKDTree3	1079627085	12097798582	2814044438
MKDTree4	977706526	11571227742	2625418184
MKDTree5	873262892	10744341119	2388029936

Table B.9: Jago.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	1542	1292	2	4.61	6	1	26007	99611	24664	3	14215	57369	13257
KDTree	1303	581	4	11.28	15	1	57938	99732	25449	3	30996	57370	13630
VKDTree	1023	512	9	9.00	9	56018	65708	76692	3067	35080	36230	41600	1292
MKDTree3	1093	729	6	6.00	6	36757	46114	50528	2186	24469	25771	36533	1403
MKDTree4	1365	1024	5	5.00	5	25341	32821	36314	1545	17465	18513	26942	1147
MKDTree5	781	625	4	4.00	4	44613	53761	57421	2282	28473	30107	39453	2030

Table B.10: GreatTemple.ply Performance Data

Hierarchy	**Elapsed Time (seconds)**	**RAM p50 (bytes)**	**RAM p75 (bytes)**	**RAM p100 (bytes)**	**runs**
Octree	605.00 ± 4.49	71220812 ± 7368731.32	93173006 ± 7986564.73	$709428666 \pm 90907879.55$	4
KDTree	735.50 ± 3.83	$204327158 \pm 45033045.47$	$279715996 \pm 40533403.74$	$686614822 \pm 50013817.06$	4
VKDTree	742.50 ± 5.18	$189349425 \pm 45347801.40$	$241414338 \pm 58596003.63$	$446216354 \pm 46861832.04$	4
MKDTree3	689.50 ± 0.25	$310425028 \pm 95144935.25$	$354447638 \pm 95895849.77$	$541148084 \pm 71039678.20$	4
MKDTree4	691.25 ± 4.02	$213024356 \pm 36520619.05$	$251979730 \pm 33245398.82$	$334878858 \pm 13910562.59$	4
MKDTree5	706.50 ± 1.48	$305976500 \pm 82778833.76$	$354373658 \pm 81638789.15$	$475627354 \pm 67720409.53$	4

Table B.11: GreatTemple.ply Disk Data

Hierarchy	**Output File Size (bytes)**	**Total Temporary Bytes Written**	**Maximum Temporary Bytes Stored**
Octree	980997605	13503084014	2719826192
KDTree	1813123378	19229950415	3573447807
VKDTree	1394254160	15304601949	2746174798
MKDTree3	1071868155	12126817541	2798582102
MKDTree4	972963811	11602770973	2616598922
MKDTree5	871004849	10780360272	2384554464

Table B.12: GreatTemple.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	1558	1312	1	4.60	5	1	25892	99874	22449	3	13915	57209	11978
KDTree	1263	577	5	11.36	14	11	58979	99659	25141	12	31203	56637	13301
VKDTree	1023	512	9	9.00	9	54685	66428	71861	2802	34766	35647	38570	574
MKDTree3	1093	729	6	6.00	6	37469	46620	50170	2048	24510	25204	29763	521
MKDTree4	1365	1024	5	5.00	5	26974	33181	36371	1555	17476	18147	22048	499
MKDTree5	781	625	4	4.00	4	44392	54351	58480	2325	28526	29495	33277	701

Table B.13: ShamaFort.ply Performance Data

Hierarchy	**Elapsed Time (seconds)**	**RAM p50 (bytes)**	**RAM p75 (bytes)**	**RAM p100 (bytes)**	**runs**
Octree	1921.67 ± 33.78	$132664419 \pm 19579374.63$	$239819061 \pm 53183221.14$	$2400612109 \pm 128005959.71$	3
KDTree	2675.67 ± 23.10	$683642397 \pm 130637181.34$	$989810693 \pm 71237311.80$	$3141190421 \pm 758794981.72$	3
VKDTree	2206.00 ± 29.63	$985964277 \pm 272076811.72$	$1144236053 \pm 279481242.47$	$1624004045 \pm 238064378.37$	3
MKDTree3	2118.33 ± 26.80	$1190191411 \pm 121275255.59$	$1462626941 \pm 74989966.31$	$1978287595 \pm 21662553.17$	3
MKDTree4	2013.67 ± 0.27	$536953701 \pm 131797020.84$	$686317781 \pm 137531113.02$	$1019260144 \pm 106538185.26$	3
MKDTree5	2145.00 ± 15.28	$1132022721 \pm 88635335.35$	$1275130776 \pm 101490487.51$	$1553068837 \pm 165781350.43$	3

Table B.14: ShamaFort.ply Disk Data

Hierarchy	**Output File Size (bytes)**	**Total Temporary Bytes Written**	**Maximum Temporary Bytes Stored**
Octree	2848899984	43828885450	7896022304
KDTree	5561284181	70697626399	10958058904
VKDTree	3565473466	44496782018	7025412888
MKDTree3	2862555062	36270203191	7485635427
MKDTree4	2403682147	30487643613	6483268334
MKDTree5	2422942931	33173091694	6647127112

Table B.15: ShamaFort.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	4319	3644	2	5.95	7	2	26111	99414	22744	4	13842	54301	11847
KDTree	3526	1669	3	16.05	19	4	57012	99981	24983	5	29525	55605	12818
VKDTree	2047	1024	10	10.00	10	79834	92924	97168	2561	47390	48166	49948	370
MKDTree3	3280	2187	7	7.00	7	36453	43509	46256	1349	22163	22786	25467	316
MKDTree4	1365	1024	5	5.00	5	79210	92924	97202	2599	47261	48182	50695	449
MKDTree5	3906	3125	5	5.00	5	24994	30449	33352	993	15504	16045	18545	298

Table B.16: Jago4cm.ply Performance Data

Hierarchy	**Elapsed Time (seconds)**	**RAM p50 (bytes)**	**RAM p75 (bytes)**	**RAM p100 (bytes)**	**runs**
Octree	1498.50 ± 11.67	170134976 ± 12913879.83	211016616 ± 12955338.92	1985460460 ± 483901.45	2
KDTree	1991.50 ± 8.84	428221000 ± 44707968.93	524606672 ± 32153271.05	2138120320 ± 145007.80	2
VKDTree	1921.00 ± 29.70	443121774 ± 1252728.76	635077424 ± 19019486.67	1155639324 ± 2345119.64	2
MKDTree3	1803.50 ± 31.47	1217394432 ± 103159785.68	1417778656 ± 81568772.26	1750778892 ± 65218938.96	2
MKDTree4	1782.50 ± 37.12	617108884 ± 5546508.82	738003672 ± 807962.84	899136712 ± 16954361.52	2
MKDTree5	1888.00 ± 39.60	783313038 ± 194517485.42	882730472 ± 205497743.57	1130716772 ± 191414210.13	2

Table B.17: Jago4cm.ply Disk Data

Hierarchy	**Output File Size (bytes)**	**Total Temporary Bytes Written**	**Maximum Temporary Bytes Stored**
Octree	2202831705	32411677080	6084459210
KDTree	4201655143	50478882203	8265265032
VKDTree	3156069725	38232834945	6205109258
MKDTree3	2534877154	31284544247	6608848377
MKDTree4	2111762620	26268262359	5679572918
MKDTree5	2140769518	28693198020	5853644282

Table B.18: Jago4cm.ply Geometry Distribution Data

			Leaf Depth			Leaf Faces				Leaf Vertices			
Hierarchy	**nodes**	**leaves**	**min**	**mean**	**max**	**min**	**mean**	**max**	**stddev**	**min**	**mean**	**max**	**stddev**
Octree	3267	2732	2	5.03	6	2	29345	99904	23847	4	16019	58767	12961
KDTree	2989	1383	4	13.16	16	5	57973	99990	25899	7	31296	61179	13980
VKDTree	2047	1024	10	10.00	10	59414	78297	88972	4193	41193	43055	58624	1578
MKDTree3	3280	2187	7	7.00	7	15805	36660	53070	2384	17993	20601	38673	1326
MKDTree4	1365	1024	5	5.00	5	61774	78297	97015	4270	40596	43376	65137	2304
MKDTree5	3906	3125	5	5.00	5	827	25654	47855	1821	1364	14666	37701	1246

References

[Ali10] Daniel Aliaga. Level of detail: A brief overview. http://www.cs.purdue.edu/homes/aliaga/cs535-10/lec-lod.pdf, 2010.

[CZGZ08] Hongmin Chen, Shouyi Zhan, Yu Gao, and Wei Zhang. View-dependent out-of-core rendering of large-scale virtual environments with continuous hierarchical levels of detail. In *Computer Science and Information Technology, 2008. ICCSIT '08. International Conference on*, pages 313–321, Aug 2008.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, pages 1–9, 1974.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FPI84] A. Fujimoto, C.G. Perrott, and K. Iwata. A 3-d graphics display system with depth buffer and pipeline processor. *Computer Graphics and Applications, IEEE*, 4(6):11–23, June 1984.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop98] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization '98. Proceedings*, pages 35–42, Oct 1998.

[HSH10] Liang Hu, P.V. Sander, and H. Hoppe. Parallel view-dependent level-of-detail control. *Visualization and Computer Graphics, IEEE Transactions on*, 16(5):718–728, Sept 2010.

[jso] Introducing json.

[Lak04] Ali Lakhia. Efficient interactive rendering of detailed models with hierarchical levels of detail. In *3DPVT*, pages 275–282. IEEE Computer Society, 2004.

[LM80] Rensselaer Polytechnic Institute. Image Processing Laboratory and D.J.R. Meagher. *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980.

[Pri00] Chris Prince. Progressive meshes for large models of arbitrary topology. Master's thesis, University of Washington, Seattle, 2000.

[RHB+12] Heinz Rüther, Christof Held, Roshan Bhurtha, Ralph Schroeder, and Stephen Wessels. From point cloud to textured model, the zamani laser scanning pipeline in heritage documentation. *South African Journal of Geomatics*, 1(1):44–59, 2012.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RLB10] José Ribelles, Angeles López, and Oscar Belmonte. An improved discrete level of detail model through an incremental representation. In John P. Collomosse and Ian J. Grimstead, editors, *TPCG*, pages 59–66. Eurographics Association, 2010.