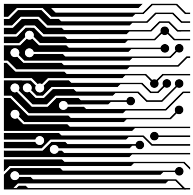
HONOURS PROJECT REPORT

HIERARCHICAL LEVEL-OF-DETAIL RENDERING OF LARGE MESHES

Author:
Justin Cossutti - CSSJUS002
University of Cape Town
Department of Computer Science
justin.cossutti@gmail.com

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

	Category	**Minimum**	**Maximum**	**Chosen**
1	Requirement Analysis and Design	0	20	15
2	Theoretical Analysis	0	15	0
3	Experiment Design and Execution	0	20	0
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	15
6	Aim Formulation and Background Work	10	15	15
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	0
	Total Marks	**80**		**80**



Department of Computer Science
University of Cape Town
2014

Abstract

The Zamani Project is an initiative started by Heinz Rüther of UCT's Geomatics Department. The project's aim is to digitally preserve architectural heritage sites around Africa using laser scanners to generate 3D models of the sites. This approach becomes problematic when the models they generate are simply too big to view using their current tool, MeshLab, achieving only 1.7 frames per second. Our goal in this project was to design and develop a viewing client for rendering large 3D models at frame rates of at least 30 frames per second, with little noticeable loss of detail.

Our solution makes use of a hierarchical level-of-detail scheme to render geometry at different qualities based on how far away it is from the camera in the viewing client. A conversion of the models from their standard 3D format into this hierarchy based format is required. The model's mesh is split into smaller chunks, simplified down to multiple qualities and saved in a hierarchy file. The viewing client, which is the focus of this report, uses this hierarchy file as input to render the model at different qualities. I make use of screen-area and distance metrics to determine which at level of detail to render each chunk. I also adopted a multi-threaded approach in an attempt to transfer as much work as possible from the hard disk to the CPU.

The viewing client was tested on 5 different models, ranging from 67 MB to 1.6 GB in size. The test results were encouraging, each passing the target benchmark of 30 FPS. While the quality of the rendered image could be improved, I consider the overall project a success.

Acknowledgments

This project would not have as successful if it was not for the guidance of our project supervisor, Patrick Marais. His extensive knowledge in the field of computer graphics proved to be an invaluable resource to guide us on the right approach to the problem this project attempts to tackle.

Credit must go to the National Research Foundation for providing me with the bursary that made my Honours year of study possible.

The team members of the Zamani Project, for providing a tough problem that has taught me a considerable deal not only in the field of computer graphics, but also in that of geomatics. Thank you for being such an enthusiastic client.

Finally, a huge thank you to my awesome group members (and friends), Benjamin Meier and Daniel Burnham-king, with whom I have had the privilege of studying alongside for 4 memorable years.

Contents

1 | Introduction

The Zamani Project is an initiative started in 2004 by the Geomatics Department at the University of Cape Town. The focus of the project is to digitally document and model African cultural heritage sites for the purposes of preservation, restoration and education. Data used in the documentation of these sites include photogrammetric imaging, land surveys, aerial and satellite images and laser scans [zam]. It is the latter provides the context for our project.

Laser scanners use a laser beam aimed at many points of an object's surface to determine the exact distance from the scanner unit. The output of these scans is a point-cloud, a cluster of thousands to billions of points which can be extrapolated to determine the surface of an object. Through a process called meshing, the points are connected together to form a continuous surface of the 3D model using triangles. The point location and inter-connectivity data forms the input data to our project.

Rendering is the conversion of a 3D model, viewed from an arbitrary location, into an image. Being able to render the 3D models captured by the Zamani team is important for obvious reasons: their work is not of much use if it cannot be viewed at a later stage. The rendering component is the final component of this project. It is this rendering component that the client will interact with to view the 3D models.

1.1 Problem statement

Improvements in laser scanning technology have resulted in incredibly large point-clouds (and hence meshes) being generated. The resolution of captured point clouds has increased greatly, with point clouds having grown from 50 million points per site to over 7 billion points per site [RHB+12]. These large data sets pose a problem to the Zamani team. The current software tools they use to clean and view their models simply cannot cope with the sheer size of their point clouds. As a brief example, I compare the frame rates achieved by MeshLab, the mesh editing tool they use, and a basic, naïve implementation of my own using no optimization techniques whereby the entire model is loaded and rendered onto the screen. I used a model of the *Chapel of Nossa* which consists of 9.4 million polygons. MeshLab achieved 1.7 frames per second, while the naïve implementation achieved 48.5 frames per second.

The 1.7 frames per second achieved by MeshLab makes the task of navigating and viewing any model frustrating, as input from the mouse and keyboard has very little effect on moving the model to the required orientation. Such a viewing client suffers heavily from stuttering and delayed input and does not produce an interactive frame rate[1].

One possible reason for the difference between a piece of open source software and a quick-and-dirty implementation is the functionality it offers. The naïve implementation only provides viewing and navigation functionality, while MeshLab offers more complex tools for manipulation meshes.

A tool providing the same functionality as MeshLab is beyond the scope of this project. Instead, we plan to exploit the lack of overhead of the complex tools to produce a product for the Zamani team that allows them to explore their models, but not manipulate them, at interactive frame rates. Such a product will enable the team to quickly view a model at a frame rate better than their current tool. This could be used during presentations of the sites they have scanned and to produce fly-through videos.

The main focus of the rendering component is therefore to efficiently render large models that are, at present, too large to open in Zamani's current software suite.

1.2 Proposed solution

Our proposed solution is to develop viewing client that renders a given model at multiple levels of detail. Each level of detail uses successively more triangles to represent the same surface of the model. By dynamically adjusting the level of detail of sections of the model based on how far away they are from the viewer, we can reduce the number of triangles that have to be rendered to represent the same surface without too much loss of detail. A detailed analysis of multiple level-of-detail (LOD) schemes and additional concept explanations is given in Chapter 2.

In order to realise this proposed solution, we require three main components: a simplification tool that reduces the number of triangles required to represent a given surface, a viewing client that renders the model in the proposed LOD format and a tool that converts a model into the correct LOD format for the viewing client by splitting the original model into successively smaller chunks. Designing, implementing and testing a suitable viewing client is the focus of this report.

1.3 Research questions

Given the context of the problem at hand and our proposed solution, our combined research question is:

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

and given the focus of my report, my individual research question is:

- *"Can a large model processed using a level-of-detail approach be rendered at interactive frame rates?"*

The difference between the questions is the emphasis placed on the actual rendering of already-processed models which is the focus of my report.

1.4 Report outline

This report is organised as follows: Chapter 2 provides explanations for essential concepts addressed in this report and takes a look at previous work done in the area of rendering large meshes. Chapter 3 takes a look at the main challenges of rendering large meshes and describes the proposed solution in greater detail. More specific implementation details and rationale is provided in Chapter 4. I present my testing plan and results in Chapter 5 and finally, a summary of the results and system's capability to answer our research questions is made in Chapter 6.

[1] An interactive frame rate is a certain number of frames per second that allows the user to navigate or manipulate the mesh effectively. A frame rate that is too low will cause the image displayed on the screen to lag behind input from a keyboard or mouse, making the user experience unpleasant. For our project, we define an interactive frame rate as at least 30 frames per second.

2 | Background

Additional context and explanations of key concepts are needed to understand the terminology used in this report. This chapter explains concepts pertaining to the graphics pipeline and common algorithms that are mentioned throughout this paper. Next, we consider previous, related work, highlighting ideas and algorithms that could potentially be used in the design of our solution. Finally, consideration is given to other widely-used algorithms.

2.1 Concepts

2.1.1 3D Geometry

3D geometry can comprise of a number of elements. Specific to this project are points and meshes. A point is an entity that has a location. Specifically, 3D models comprise of a cluster of points which are samples taken over the model's surface. A mesh is a set of points linked together. Typically in digital modeling, a mesh consists of points linked together to form a surface of triangles. Points can contain additional, optional information and attributes such as colour and (in the case of meshes) normals.3D geometry can comprise of a number of elements. Specific to this project are points and meshes. A point is an entity that has a location. Specifically, 3D models comprise of a cluster of points which are samples taken over the model's surface. A mesh is a set of points linked together. Typically in digital modeling, a mesh consists of points linked together to form a surface of triangles. Points can contain additional, optional information and attributes such as colour and (in the case of meshes) normals.

2.1.2 Graphics processing unit (GPU)

A hardware device found in most computers that can rapidly perform transformations and calculations required to display interfaces and scenes on a screen. Their limited, but parallel operations of a GPU (when compared to a CPU) allow them to perform many calculations per second. The rendering component of this project deals with efficient ways of pushing data to the GPU so as to best utilize its available resources.

2.1.3 Shaders

Shaders are programs that tell the GPU how to transform and colour vertices and pixels. The two main types of shaders are vertex shaders and fragment shaders (also known as fragment shaders). Vertex shaders are run once for each vertex sent to the GPU and typical calculate vertex position, texture coordinates and colour. Fragment shaders are run for each fragment

output from the rasterizer (described below) and typically calculates fragment colour. Other shaders also exist and more complex calculations can be performed, but they are not relevant to this project.

2.1.4 Normals

A normal is resultant vector of the cross product of another two vectors. It is commonly used in shader programs to determine the contribution of a light to a surface or point and hence, it's final screen colour. It is also used to cull away triangles and other geometry that is facing away from the viewer.

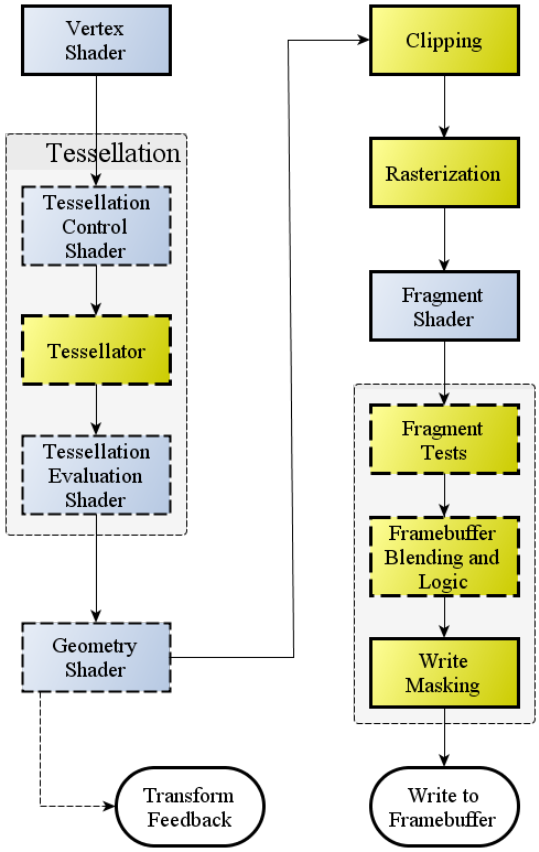


Figure 2.1: An overview of OpenGL's rendering pipeline stages. Source: [pip]

2.1.5 OpenGL

A free, open-source and cross-platform graphics library used to interface with the GPU to make use of hardware acceleration for rendering geometry. Another popular graphics library is DirectX, which is owned by Microsoft and only runs on the Windows operating system. I therefore decided to use OpenGL. OpenGL uses a sequence of steps when rendering objects

called the Rendering Pipeline (see Figure 2.1). A simple acknowledgment and understanding of the steps in the pipeline is required to weigh up the advantages and disadvantages of various rendering algorithms.

In order for OpenGL's shader programs to correctly use your vertex data stream for rendering, it needs to adhere to the vertex specification. The vertex steam consists of a series of integer or floating point values that define your vertices. For 3D graphics, 3 values are required per vertex for location information. The shader program defines a set of attributes which tell OpenGL how to interpret these values.

The main components in the rendering pipeline relevant to this report are:

Vertex shader - Programmable stage in the pipeline that operates on each vertex. Transforms vertex location and colour information that can be used in the fragment shader.

Primitive assembly - Combines vertices in the vertex stream into ordered stream of primitives, such as triangles. The order in which the vertices are submitted affects how the primitives are assembled. This stage is not apparent in Figure 1, but it occurs after the vertex shader step.

Clipping - The clipping step splits primitives that lie on the boundary of the view frustum into smaller primitives. Culling is also performed here, which discards any primitives outside the view volume or are facing away from the camera, also known as back-face culling.

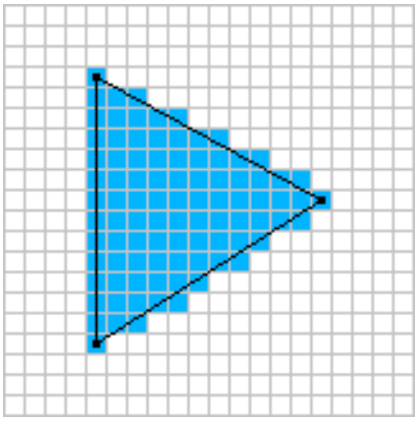


Figure 2.2: Example rasterization of a triangle into fragments. Source: [Bea]

Rasterization - The process of breaking down each remaining primitive into a collection of fragments as can be seen in Figure 2.2. Fragments are entities containing state for each pixel on the screen. The state includes information for screen location, colour and other arbitrary values that are interpolated between the primitives' vertices.

Fragment shader - Processes each fragment sent from the rasterizer; setting the final colour value for each pixel on the screen. Both the fragment and vertex shaders can be used for lighting calculations that would affect colour values.

2.1.6 Vertex buffer object (VBO)

A feature in OpenGL that acts as a storage container for vertex attributes and data (position, colour, texture coordinates) by allocating a block of memory on the GPU to make rendering faster. By placing the data on the GPU, the GPU can access it faster during rendering and it saves bandwidth as it only has to be sent to the GPU once. There are two types of rendering modes in OpenGL: immediate mode and deferred mode rendering. Immediate mode rendering pushes vertex data to the GPU immediately after a draw call is issued. This mode results in many draw calls being made to the GPU, which is not efficient. Deferred mode rendering allows the programmer to prepare and declare all the required vertex data, push it all to the GPU at once and draw multiple primitives using a single draw call. This is a far more efficient rendering mode and is used in almost all but the simplest rendering applications.

As the size and quantity of VBOs allowed on the GPU is limited by the amount of RAM available on the GPU, it is important to manage the memory allocations effectively – one of the main challenges of this project component.

2.1.7 Vertex attributes

A construct in OpenGL that allows different types of data to be stored alongside in the same VBO - this is called interleaved data. Data types can include vertex positions, colour, texture coordinates and normals. Each attributes must have its offset (position from the start of the buffer) and stride (how many bytes until the next attribute of the same type will occur) must be set before it can be used. Using multiple attributes in the same buffer has a performance advantage in OpenGL, as it can read multiple data in a single read call from a single buffer. An example layout of a buffer containing interleaved data can be seen in Figure 4.5.

2.1.8 Bounding volume hierarchy (BVH)

A tree-based data structure where each node contains a rectangular or spherical bounding volume that encompasses the data of its child nodes. Leaf node (nodes at the bottom of the tree and have no child nodes) volumes wrap around vertices or other geometry. The hierarchy enables quick rejection of an entire branch of the tree, and all its geometry, by performing a test on the parent node. If the test rejects the parent node, it would reject the child nodes and hence, they need not be tested. If the parent is not rejected, the child nodes are then each tested.

2.1.9 View frustum

A region of space or field of view of the virtual environment that is potentially visible to the viewer. It can be compared to a virtual camera navigating the virtual environment, with the direction of the camera lens determining what the viewer can see. Moving around the environment effectively shifts the location and orientation of the view frustum. Figure 2.3 shows an example view frustum where the visible region lies between the near and far clipping planes. During the clipping process of the rendering pipeline, primitives lying outside the frustum are discarded as they cannot be seen and need not be rendered. After the remaining primitives have been rasterized and the fragments have been shaded, they are projected onto the near plane using a pre-calculated projection transformation. When multiple fragments are projected onto the same location, their relative depths (distance) from the frustum's origin is used to determine which fragment should be rendered.

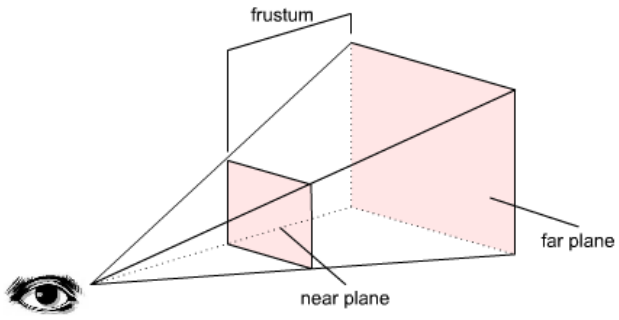


Figure 2.3: A example view frustum with the space between the near and far planes visible to the viewer positioned at the eye. Source: [Bea]

2.1.10 Level of detail (LOD)

A technique for rendering highly detailed objects and scenes whereby objects are rendered at a specific level of detail that is inversely proportional to the distance from the viewer. The purpose of this technique is improve rendering performance and reduce the workload on the GPU by decreasing the number of vertices it must operate on in the rendering pipeline. The amount of noticeable loss of detail is somewhat mitigated by the increase in distance from the viewer. This technique can also be used when moving around a scene, whereby fast moving objects can have reduced quality which can also go unnoticed at fast speeds.

2.2 Previous work

Trying to render large models is well researched problem as system resources have always been limited and the size of data sets is constantly growing. There are two distinct categories that the majority of proposed implementations fall into, namely those based on progressive meshes and those that make use of space partitioning and BVHs.

2.2.1 Progressive meshes

Progressive meshing is a scheme first introduced by Hugues Hoppe that provides a loss-less LOD mesh representation via incremental levels of mesh simplification and can be refined based on the current location of the viewer [Hop96]. It takes as input the original, high-resolution mesh and iteratively simplifies it using a technique called "edge-collapse" (the removal of an edge from a group of triangles to form a new, sparser group as shown in Figure 2.4).

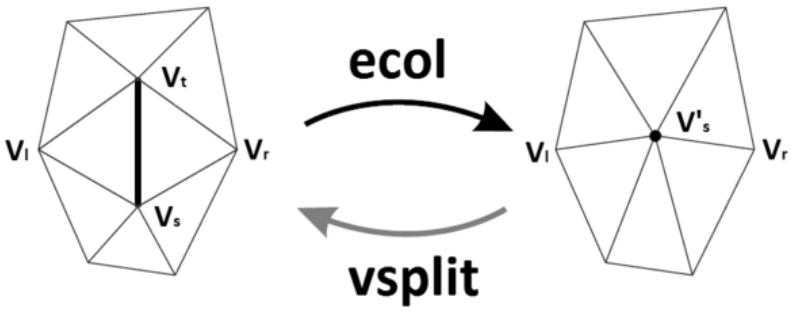


Figure 2.4: An example of an edge collapse operation and its reverse, the vertex split. Source: [Wik]

Hu et al. later proposed such a system, offering view-dependent, parallel refinement of progressive meshes [HSH09]. They improved on the data structures such that the refinement operations could be performed on the GPU. While progressive meshes appear to the perfect solution to the task at hand, they have significant disadvantages. The proposed scheme has a complex implementation that is not only would exceed this project's timeline, but would introduce many task dependencies between group members. The complex data structures required also impose a performance penalty and would slow down rendering performance. It is therefore beyond the scope of this project.

2.2.2 Hierarchical level of detail (HLOD) schemes

These schemes make use of tree-based structures such as octrees or KD-trees to partition the space into consecutively smaller, more-refined chunks. The root node of the tree contains the geometry of the full, low-detailed model, while the leaf nodes contain small, high-detailed pieces of the model. The tree structure is also well-suited for containing bounding volume information, making it a good data structure for performing frustum culling, that is, discarding the model's primitives in a volume if the volume is outside the bounds of the view frustum. As the viewer moves closer to a volume, it is replaced with the more detailed geometric content in its child nodes and vice versa.

Goswami et al. used multi-way KD-trees which split the space along the axes multiple times that allows controlling of the tree's height and using deterministic node sides,

simplifying the management of memory on the GPU [GZPG10].

Lindstrom and Lakhia both proposed solutions using octrees. Lindstrom adopts the iterative refinement approach used in progressive meshing by storing only "additional detail" in child nodes, requiring full traversal from the root to each leaf to reconstruct the model [Lin03]. Lakhia's approach ensures each node contains all required vertex data for a particular level of detail, removing the need for a complete tree traversal, but increasing data duplication [Lak04].

The independence between levels in Lakhia's HLOD scheme have the advantage of being easier to push to the GPU in VBOs as the data for each node is static. Lindstrom's solution would require a more complex solution as the vertex data changes regularly upon refinement, which provides poor performance regarding the amount of GPU transfer bandwidth required.

Another hierarchy-based algorithm is QSplat proposed by Szymon Rusinkiewicz [RL00]. QSplat also uses a BVH to recursively refine detail the closer a volume is to a viewer. The different between QSplat and the algorithms mentioned above is QSplat is a point-based renderer. This means it represents geometry using many points rather than triangle primitives. Points require less processing than triangles due to the lack of connectivity information and the primitive assembly step required in the graphics pipeline. However, the QSplat algorithm fails to exploit hardware acceleration due the traversal of the hierarchy and deciding to split or draw a splat. This requires the CPU have a direct rendering interface with the GPU rather than using VBOs in retained mode rendering.

QSplat is not well suited for architectural sites where engravings and cracks are important elements of the model that might otherwise be lost, or where the zoom level is such that it is possible to see the gaps between the measured points. A good use for a point-based renderer is for proving quick navigating around a model when high detail is not required.

A more comprehensive comparison between point-based and mesh-based renders will is discussed later on in this section.

Table 2.1 provides a brief comparison of the algorithms mentioned above. An ideal algorithm would make use of hardware acceleration provided by the GPU, have limited geometry duplication, would not have to perform a complete traversal of the required data structures and provide a method for fast culling of groups of geometry.

	Progressive mesh	**Lindstrom's incremental incremental octree**	**Lakhia's level-independent octree & QSplat**
Tree traversal	Requires parent transforms to be applied first.	Requires parent content to be rendered first. Lots of traversal required.	Only content in child node for appropriate level of detail need be rendered.
Exploitation of hardware acceleration	None. Edges regularly changing and hence, cannot be stored in VBOs.	Content of visible nodes can be stored in VBO.	Content of visible nodes can be stored in VBO (does not apply to QSplat in this instance).
Geometry duplication	None. Only stores base model and history of transforms.	Low. Successive levels in tree store finer-grained geometry that might overlap the parent's content.	High. Each node at each level of detail stores complete information required to represent its geometry.
BVH support	Can be implemented to speed up selective refinement.	Supported in the octree. Fast culling of distant geometry.	Supported in the octree. Fast culling of distant geometry.

Table 2.1: A summarised comparison between the various progressive mesh and tree-based algorithms discussed.

2.3 Additional considerations

Papers relating to the topic of rendering large amounts of geometry suggest other techniques for increasing performance that should be considered.

2.3.1 Point-based rendering (PBR)

As mentioned above, an alternative to the common mesh-based rendering is point-based rendering. Kaushik et al. [KNS] provide a good overview of using such techniques over the mesh-based counterparts. Rendering models using PBR has a few advantages, namely:

- The lack of connectivity information ensures that PBR techniques have a smaller memory footprint as there is no need to index vertex data.

- Vertex and fragment shaders are unified when using PBR, requiring fewer stages in the pipeline.
- At a zoomed out view or small models, multiple triangle primitives might project to a single pixel. This results in wasted bandwidth to the GPU due to duplication.
- Points do not have to be interpolated to form a continuous surface and also does not require many rasterization steps in the pipeline, thereby providing better rendering performance.

PBR is not without its disadvantages. In PBR techniques like QSplat, rendering large points or splats can suffer from aliasing artefacts and poor overlap when zoomed in as. Furthermore, the visible gaps between the points beyond a certain level of zoom is not optimal in some situations. Architectural models is a prime example, where important cracks and engravings require continuous surfaces, but this does not stop PBR being a viable rendering method when moving about the model.

2.3.2 View frustum culling (VFC)

VFC is the process of rejecting objects to be rendered if they are outside the view frustum to reduce rendering time. It can be used in conjunction with a BVH where each node contains multiple points or objects to be rendered. By using a bounding volume, one can quickly reject groups of objects by testing whether the bounding volume is outside the view frustum. If it is, the node's contents need not be rendered and its child nodes need not be tested.

Assarsson et al. suggest a number of improvements to speed up the process of frustum culling using axis-aligned bounding boxes. Specifically, they suggest schemes that only require testing 2 points of a bounding box, exploiting temporal coherence from the previous frame's tests and performing cheap, inaccurate tests first before moving down to more expensive tests [AM00].

2.4 Summary

Rendering large quantities of geometry is a non-trivial problem and is the main focus of this project component. An effective solution to this problem is one renders geometry in incrementally higher detail the closer it is to the viewer rather than trying to render distant geometry in high detail and which makes little difference to the current scene. It is also one that effectively manages valuable resources and requests data as it is needed. Geometry that is not visible should be discarded to save GPU resources. A number of algorithms were reviewed to find the ideal solution that will act as a reference implementation and guide my implementation. It is clear from the comparison in table 2 that a scheme based on HLOD is ideal. Progressive meshes have a complexity overhead that makes them unsuitable for this project.

3 | Design

This chapter discusses the challenges the rendering of large models poses and looks at the proposed solution specific to the viewing client in greater detail.

Figure 3.1 provides a high-level, structural overview of the project. The pre-processor is composed of a simplification tool, a mesh splitter and hierarchy builder. These are the focus of the reports of my project group members. The viewing client (or renderer) is the final component of the project and is the focus of this report. The viewing client renders models that have been processed by the previous components. It is responsible for selecting the parts of the model that are to be rendered at a level of detail, reading the appropriate data from disk and properly managing VBO and memory resources.

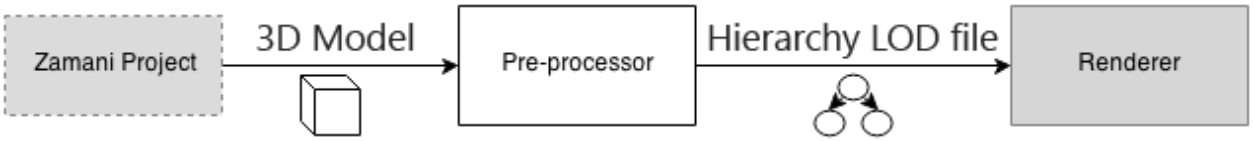


Figure 3.1: A high-level structural overview of the components that this project is composed of. The final component, the renderer, is the focus of this report.

3.1 Challenges

Unfortunately, the naïve implementation of placing the entire model into a single VBO is not a panacea. The model used in the experiment of such an approach was roughly 173MB in size – relatively small compared some models in the GB range. A model of this size is small enough to have all its vertex data placed into a single VBO for quick rendering. With limited RAM on GPUs, this is not the case for larger models and hence, a more sophisticated approach that properly manages available resources is required. This forms the crux of this component.

Another challenge faced in this component is the limited transfer bandwidth available for loading vertex data from disk onto the GPU. Large vertex streams can take a few milliseconds to be read from disk – enough to stall the graphics pipeline if it has to wait for data to be loaded.

The two main challenges to overcome can be summarised as 1) selecting the correct "chunks" of data to be loaded onto the GPU when they are needed in a manner that minimises pipeline stalls (stalling occurs when the graphics pipeline has to wait for the data it needs to be loaded, causing the screen to freeze briefly) and 2) ensuring the required data

does not exceed the available system resources. This requirement should be relatively independent of the original file size of the model being viewed. No viewing client would be useful if it did not render the models in a presentable manner (particularly architectural models where small details in the geometry are important). Thus, an additional requirement of the client is to adequately the surface and underlying mesh structure. This means correct usage of lighting and shaders is important.

3.2 Proposed solution

3.2.1 Overview

Our collective solution to rendering large models is based upon an implementation of an HLOD. The model's geometric data is recursively split along splitting planes. The data residing on each side of the splits forms the contents of a node in the hierarchy. A split of a node's data results in multiple new nodes forming the "children" of the split node. In an HLOD scheme, if one traverses the resultant tree in a bottom-up direction, the geometry contained at each level is successively more simplified. The child nodes' data occupy the least amount of volumetric space in the virtual world, but contain the original, unsimplified geometry, while the top-most node of the tree (the root node) occupies the most volumetric space at the highest level of simplification. The formed hierarchy has a tree-like branching appearance; from here on, the terms "tree" and "hierarchy" are used interchangeably. Note that the splitting of the model and the simplification of each nodes' data both occur as a pre-processing step before the model can be viewing in the viewing client.

The viewer can then use this HLOD scheme to render sections of the model at different levels of detail – sections of the model that are further away from the viewer (closer to the far clipping plane of the frustum) can be rendered using more simplified geometry found in the upper levels of the tree, while sections close to the viewer are rendered using the high detailed geometry found in the bottom levels of the tree.

Our HLOD solution is closest to the level-independent octree proposed by [Lak04]: there is geometry duplication in different levels, but it does not require rendering the contents of all nodes between the target level of detail and the root node. The contents in each node can also easily be stored in VBOs for improved performance. As previously discussed in the comparison of HLOD schemes, this tree structure has the added benefit of being able to cull away entire branches of the tree if their bounding boxes do not intersect with the view frustum. This high culling efficiency is the primary reason we adopted this approach of using an HLOD-based solution.

My approach differs from other research by trying to shift as much work as possible from the disk to the CPU by employing multi-threading techniques to reduce the amount of data having to be read. The disk is often a bottleneck in applications that require data to be loaded and thus, I try to reduce the time spent waiting on IO operations.

3.2.2 Input

The viewing client takes, as input, a single file output by the pre-processor called the hierarchy file (HF) - see Figure 3.1. The HF consists of a JSON header which contains node information such as the byte-offset to the node's data block, parent-child pointers that form the hierarchy and other information that can be seen in Figure 3.2, followed by the data blocks of the nodes.

```
{
    "max_depth": integer (maximum depth of the tree)
    "vertex_colour": boolean (vertices have colour information present)
    "nodes": [
        {
            "id": integer (node id)
            "parent_id": integer (id of parent node)
            "num_faces": integer (number of faces node contains)
            "num_vertices": integer (number of vertices node contains)
            "block_offset": integer (byte offset of start of node's data block)
            "block_length": integer (length of node's data block)
            <node bounding box coordinates>
        },
        ...
    ]
}
<node data blocks start here>
```

Figure 3.2: The hierarchy file contains this header information encoded in JSON format that describes the relationship between nodes in the hierarchy, the geometry each node contains and the offset in the file to start reading a node's data block.

The viewer uses the JSON data to construct an internal representation of the implied hierarchy. At this stage, none of the nodes' data have been loaded – only the JSON header outlined in Figure 3.2.

3.2.3 View-dependent refinement

The core algorithm to the viewer is the hierarchy traversal algorithm (outlined in Algorithm 1 and detailed in the Chapter 4) which determines the nodes that should be rendered, also called the rendered front. The rendered front is a construct that assumes that all nodes' data already reside in memory and no loading is required. This is generally not the case, however, as we have already discussed why the naïve implementation of loading all the nodes'

data at once is not feasible. We therefore need to determine the actual rendered front – a compromise between the ideal rendered front and a rendered front whose nodes' data are mostly available. This reduces the likelihood of stalling the rendering pipeline while it waits for node data to be loaded and further reduces the number of gaps in the model's geometry of non-loaded data. Geometry that is loaded to fill a gap in the rendered front will result in popping artefacts visible in Figure 3.3. This compromise comes at the cost delaying the rendering of the mode detailed geometry until it is loaded into memory.

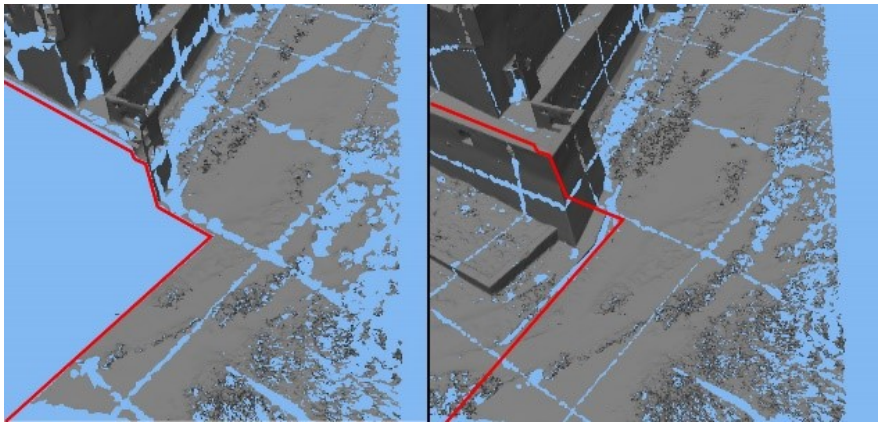


Figure 3.3: A block of data waiting to be loaded causes a gap in the rendered model.

For the remainder of this section, we assume the ideal rendered front has been determined. Additionally, "visible" nodes are exactly that – nodes that are visible given the viewer's position (their bounding box intersects the view frustum) – while "active" nodes are parents, siblings or children of visible nodes.

To illustrate how the viewer operates on the hierarchy and determines the actual rendered front, we look at a succinct classification of possible node updates Goswami et al. [GZPG10] proposed.

There are four possible updates that can occur on each node in the rendered front between frames as defined by Goswami:

1. No node $\rightarrow$ new node ($\emptyset \rightarrow$ l).
2. Parent node $\rightarrow$ child nodes (expansion; i $\rightarrow$ j, k).
3. Child nodes $\rightarrow$ parent node (collapse; e,f $\rightarrow$ d).
4. Old node $\rightarrow$ no node (a $\rightarrow \emptyset$).

Case 4 is trivial, while cases 1-3 involve introducing new nodes to rendered front, and hence, data must be loaded for them to be rendered. For case 2, the geometry of the parent node can be rendered until the data for its child nodes have been loaded. Preloading

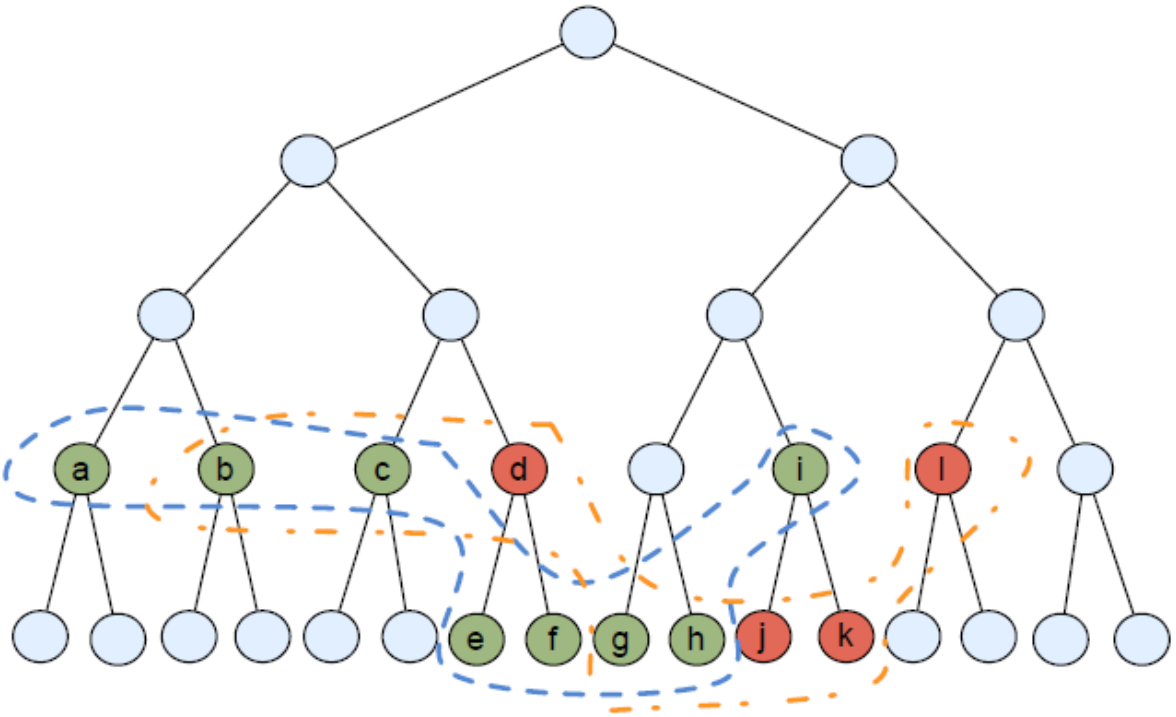


Figure 3.4: Node visibility update step between two frames. The blue and orange dotted lines correspond to the appropriate fronts in Figure 3.5. Source: [GZPG10]

data for parent and child nodes of currently visible nodes (i.e. active nodes) prevents gaps appearing for collapses in case 3. This differs to the approach adopted by Goswami in which he continues to render child nodes until the parent node is loaded – my approach makes a best effort to ensure the parent node data is always ready in memory (via preloading) so that it can reduce a set of children immediately. Case 1 is likely to produce the most gaps in the geometry, as neither parent nor child node geometry is available to be rendered while the data is loaded for the node itself.

The blue and orange striped lines in Figure 3.4 outline the rendered and ideal fronts for the previous and current frames respectively. Figure 3.5 shows the various expansions and collapses between two successive frames. It also shows the actual rendered front and the compromise with the ideally rendered front that assumes all node data has been loaded.

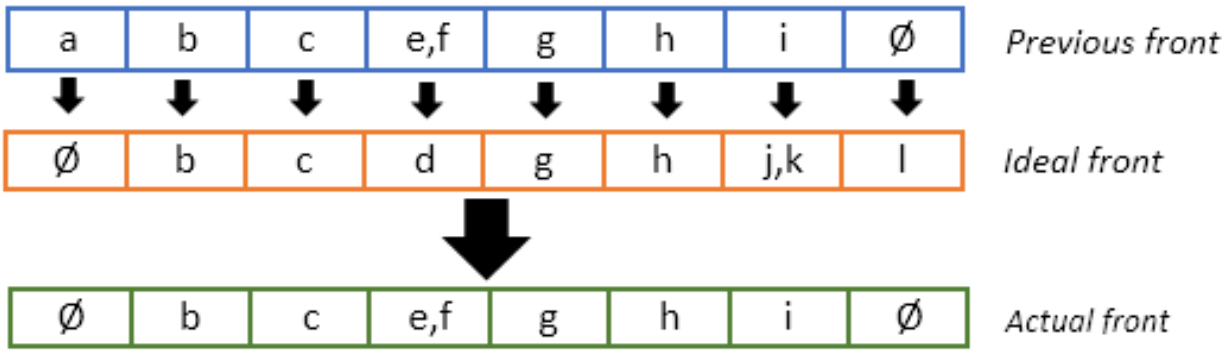


Figure 3.5: Hierarchy update step between two consecutive frames and finally, the actual rendered front determined by node data availability. Source: [GZPG10]

These figures illustrate how we can already better manage system resources by only loading and rendering data for nodes that are visible or active. For example, less-detailed geometry is used where possible by traversing up and down the hierarchy, which typically occupies less space in memory and on disk and is faster to render. Figure 3.5 also provides an example of hierarchical frustum culling can significantly reduce the amount of data sent to the GPU for rendering. In the previous frame (outlined in blue), the bounding volume of the parent node of node L would fail the test for intersection with the view frustum. If a parent node fails a particular test, all of its descendant nodes would too. Thus, we can reject a large portion of the rightmost branch of the tree for rendering (and loading) with a single test.

3.2.4 Parallelism

Loading geometric data from disk can produce gaps in the rendered model when it is only requested on-demand. Reading data from disk can easily prove to be the application's largest bottleneck. It can also cause the viewing client to stutter as it waits for data to be loaded. For these reasons the loading and processing of data is performed in separate threads from the main rendering thread. The data blocks required for the calculated rendered-front can be loaded and processed asynchronously. The advantages of utilising a separate loading thread have been previously noted by multiple authors ([Lak04], [Lin03], [Fun96]).

3.2.5 Development environment

Performant code typically suggests the requirement of using the C++ language for development, I decided to develop the viewing client in Java and use the Java OpenGL (JOGL) wrapper library to interact with OpenGL. The primary reason behind this decision is that I am more proficient in Java than in C++ and well-written Java code will provide better performance than average C++ code.

Using Java also has the added benefit of its Just In Time (JIT) compiler heavily optimising frequently called methods and loops. Java is, like C++, cross-platform and will allow the viewing client to be portable and used on both Windows and Linux machines.

4 | Implementation

In this chapter we will discuss how the viewing client implements the proposed solution outlined in Chapter 3. The viewing client is separated into three components, each responsible for a set of operations that occur in parallel on different threads. A multi-threaded approach reduces the amount of time the rendering pipeline has to wait for data to be loaded and processed, thereby reducing the amount of image stutter on the screen and providing a smooth viewing experience.

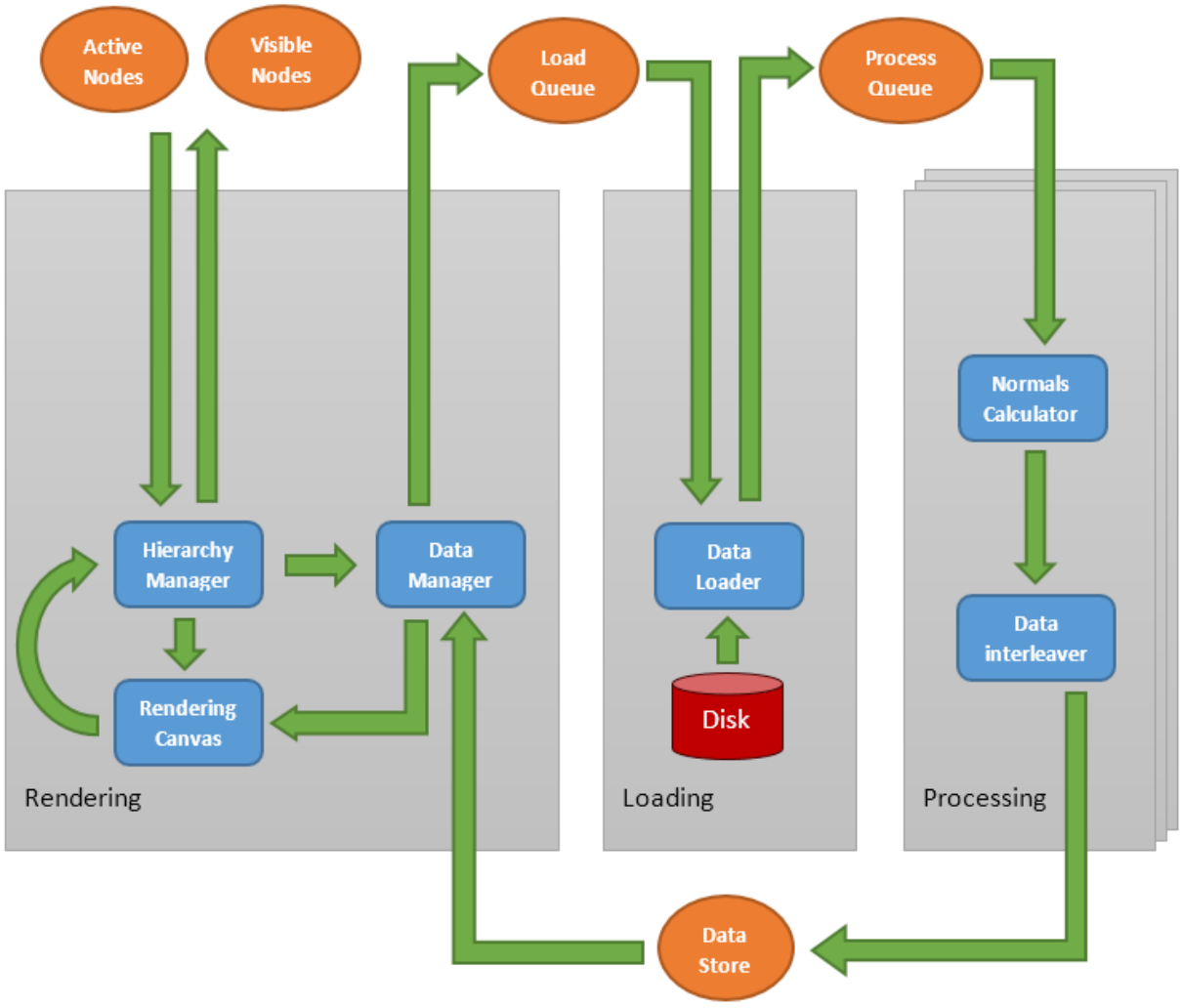


Figure 4.1: Viewing client implementation overview.

The three main sets of operations are: rendered-front determination and rendering, data

loading and data processing. Figure 4.1 provides an overview of the components, their interactions and flow of information. Specific components will be looked at in more detail below.

4.1 Rendered-front selection

To enable the rendering of large models, the correct levels of detail in the hierarchy must be selected so that we do not naïvely render all nodes at the most fine-grained detail. This process is called selecting the rendered-front.

The algorithm calculates the ideal rendered front by selecting the appropriate nodes to be rendered without taking into account whether or not a node's data already resides in memory.

Algorithm 1 Front selection algorithm for determining the ideal front to be rendered without considering data availability.

```
1: A ← list of active nodes (starting with the root node)
2:
3: procedure FrontSelect
4:     V ← set of visible nodes
5:     P ← set of potentially visible nodes from items in A
6:
7:     while P is not empty do
8:         N ← node popped from P
9:
10:        if bounding box of N intersects view frustum then
11:            if N is close to viewer and can be expanded then
12:                add N's children to P
13:            else if N is far from viewer and can be collapsed then
14:                remove N's siblings from P
15:                add N's parent to P
16:            else
17:                add N to A and V;
18:            end if
19:        end if
20:    end while
21:
22:    return V
23: end procedure
```

The front selection algorithm is outlined in Algorithm 1. A node is only considered for rendering if the corners of its bounding box are not all on the outside of a single plane

of the viewing frustum, thereby implementing a simple and effective frustum culling technique. More advanced techniques have been provided by Assarsson [AM00], however this intersection test proved to be sufficient for my usage.

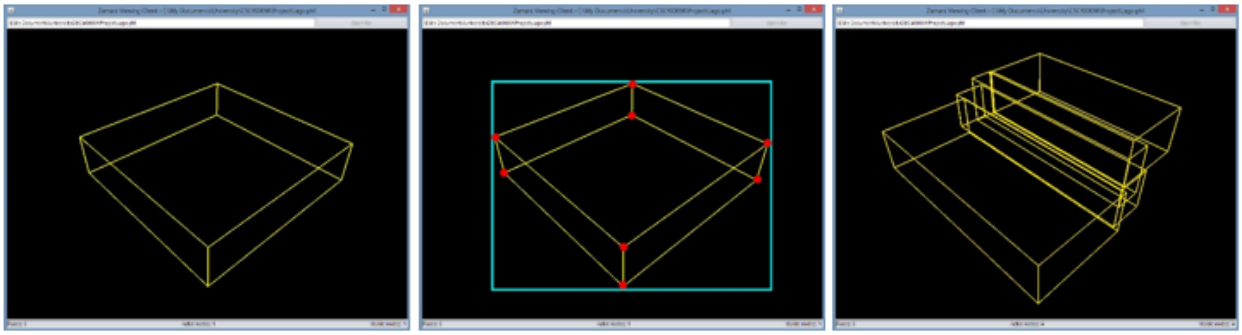


Figure 4.2: Expansion process of a node based on projected screen area.

Determining if a node should be expanded or collapsed forms a crucial part of the rendered-front determination. Expanding a node means potentially rendering its child nodes while collapsing a node means potentially rendering a node's parent. It is analogous to traversing up and down the hierarchy.

A node needs to be expanded or collapsed if it occupies too much or too little area on the screen which occurs as a user moves towards or away from a node (i.e. we want to view nodes closer to us in greater detail). The area occupancy is determined by projecting the corners of a node's bounding box onto the screen and wrapping these projected points in a 2D bounding box as can be seen in Figure 4.2. By calculating the area of the new bounding box and dividing it by the width and height of the viewing client's window, it is possible to get the proportion of the window occupied by a node. This proportion is called the screen area metric. This metric is used in a high-low threshold system. If the metric is greater than the expansion operator's high threshold value or lower than the collapse operator's low threshold, it is a candidate for being expanded or collapsed respectively. The ideal expansion and reduction thresholds were determined to be >70% and <15% of the screen area (see Chapter 5 for more details).

In order to collapse a node, all its sibling nodes must also be below the low threshold. This metric was chosen instead of a distance-from-camera metric as it is more robust against different sized node bounding boxes. The distance metric was found to expand nodes that were very small or on the edges of the view frustum that could rather be rendered in less detail.

A node that is considered renderable will not necessarily be rendered. For example: in order to reduce the number of gaps in geometry during rendering, when a node is expanded into its child nodes, those children can only be rendered if all the children's data blocks have been loaded. The expanded parent node will continue to be rendered until the data blocks

of all its visible descendant nodes have been loaded. The pseudocode for determining if a node can be rendered is provided in Algorithm 2. This addresses case 2 in the list of possible updates to the hierarchy mentioned in the Section 3.2.3 of a parent node being expanded down into its child nodes for more detail (i → j, k).

Algorithm 2 Pseudo-code of the recursive *CanBeRendered* function for determining whether a node can be rendered or not based on the load status of its data or that of its child nodes.

```
1: procedure CanBeRendered(node N)
2:     for each sibling S of N do
3:         if not S's geometric data in data store then
4:             if S has been expanded then
5:                 for each child node C of S do
6:                     if not CanBeRendered(C) then
7:                         return false
8:                     end if
9:                 end for
10:            else
11:                return false
12:            end if
13:        end if
14:    end for
15:
16:    return true
17: end procedure
```

4.2 Data loading

Once the rendered-front has been calculated, the visible and active nodes it is composed of (which is not already in the **Data Store**) are inserted into a loading queue as shown in Figure 4.1. This queue contains nodes whose data must be loaded. A separate data loading thread removes nodes from this queue and loads the appropriate data block as specified by the node's data offset and data length in the JSON header (see Figure 3.2). This multi-threaded approach allows the rendering thread to specify which nodes it needs to render, dispatch a loading request for those nodes and continue with other rendering operations. This reduces any stalling in the graphics pipeline. Once a node's data block has been loaded, the node is inserted into the processing queue.

4.2.1 Loading optimisations

To ensure the loading thread is not always playing "catch-up" for the data requested by the renderer for a previous frame, the loading queue is cleared when a new request batch arrives. If a node is still in the queue when it is cleared, it will either be in the new request batch if it is still visible or it will not be visible, in which case it need no longer be loaded anyway.

A node that is waiting in either of the queues for a long time is likely to produce gaps in the rendered geometry. The duration of visible gaps can be reduced by two methods: first, prioritized loading can be given to visible nodes and then active nodes, and second, the data blocks of visible nodes are loaded in order of the nodes' distance from the viewer such that nodes closed to the viewer are loaded first and can occlude gaps behind it.

4.2.2 Managing data

A *Data Manager* in Figure 4.1 object is responsible for caching node data until it reaches a threshold period of inactivity during which the data has not been requested by the renderer. This threshold period was heuristically determined to be around 8 seconds (further discussed in Chapter 5) – a value that provides a balance between the amount of data cached in memory at any point and the cost to reload a data block from disk. Once the inactive threshold has been reached, the node's data block is released from memory in order to make room for newly requested data blocks and must be re-loaded if it is requested again. This ensures only the relevant chunks of the model are kept in memory, allowing arbitrarily-size models to be rendered, even if its size on disk is larger than the amount of available memory.

4.2.3 Data block layout

Each node data block in the simpler format is structured as follows: vertex position data, colour data and finally, index data. A total of 28 or 24 bytes of information must be read for geometry with and without colour information respectively (3 floats per vertex + 4 bytes per colour + 3 integers per index).

Vertex indices are used to reduce the number of vertex primitives required to represent a mesh. This allows corners of triangles in a mesh can reference the same vertices as other triangles with the common vertices only having to be declared once. Figure 4.3 shows how only two vertices are required for corners A and D instead of four. A triangle now only needs three indices that point to vertices (3 integers for a total of 12 bytes) instead of three vertices (9 floats for a total of 36 bytes). The more triangles that share common vertices, the greater the amount of storage space saved.

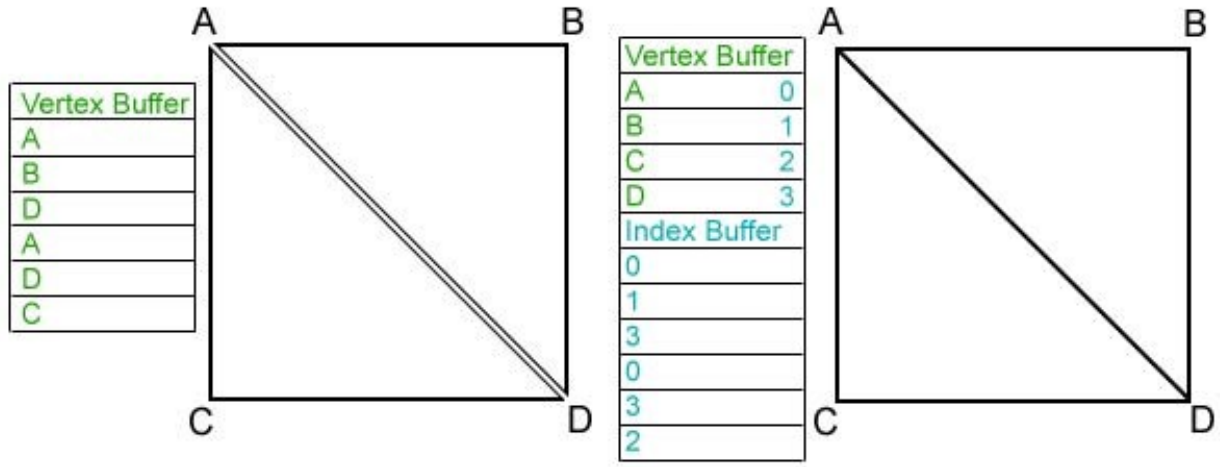


Figure 4.3: Comparison between a normal mesh and an indexed mesh. Source: [Ver]

Data for each attribute (position, colour or index) of information is read for all vertices at once. This enables all data of the same type (float, integer or byte) to be loaded using a single disk read request. Figure 4.4 shows the layout of the simple data block format and the relative disk space occupied by the various attributes. The data required for each node is stored as a single, sequential block on disk so as to reduce the number of random access reads from disk, which is slower than reading data sequentially as the HDD read head has to seek to different locations on the disk.

Figure 4.4: The ordering and relative size on disk of vertex attributes.

The separated data is not ideal for use with VBOs and so it must be interleaved during in a processing step which is detailed in Section 4.3.

4.3 Normal processing

Until this point, nodes present in the *Process Queue* in Figure 4.1 have had their data loaded, which consists of vertex position, colour and index data. To enable the use of back-face culling and proper lighting, the vertex data must also contain normals.

I decided to calculate the vertex normal on-the-fly rather than during the pre-processing stage and since this results in a smaller data block size that needs to be read from the disk. It is also an embarrassingly parallel workload since it can be trivially split into parallel calculations without dependencies. The nodes waiting in the process queue can then be

processed faster than it can be read from disk by an arbitrary number of processing threads (also called a worker-pool). Loading less data enables the disk to begin loading the next node's data while the processing stage executes alongside it, thereby providing more data throughput.

The normal is calculated for each triangle in the data block, as defined by a sequence of 3 integers from the loaded index data. A uniformly weighted per-vertex normal is then calculated by taking the average of the normals of the triangles it belongs to. This provides smoother lighting in the final image when compared to weighting each normal based on its triangle's area. This is due to the irregular sampling of the laser scanner and the mesh simplification process on rounded surfaces which causes triangle areas to greatly vary in magnitude.

4.3.1 Interleaved data

The final step in processing a data block requires interleaving the data in such a way that it provides optimal performance when loaded into a VBO. Interleaving the data transforms the layout of the data from Figure 4.4 into the format depicted in Figure 4.5.

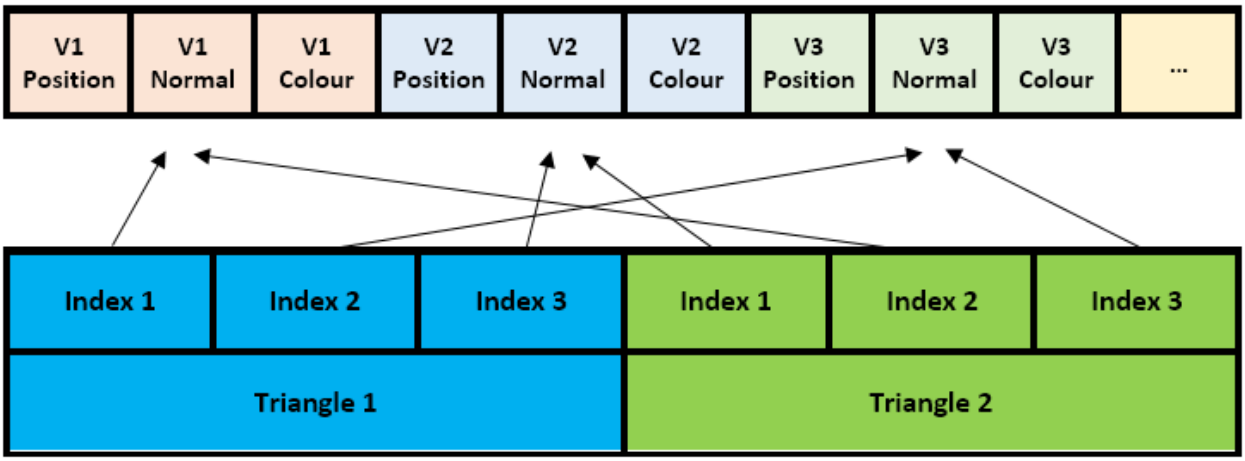


Figure 4.5: Interleaved data buffer format considering of the element buffer and the index buffer

All data that describes particular vertices is placed into the *element buffer* (top buffer in Figure 4.5) and the triangle index data is placed into the *index buffer* (bottom buffer). The element buffer is now considered interleaved as multiple vertex attributes are inserted into the same buffer at regular intervals. This layout is optimal for OpenGL because all the attributes for each vertex can be read together as opposed to having to load position, normal and colour attributes from different locations in different buffers.

After the data has been interleaved, the node's data block is inserted in the data manager's cache where it is ready to be used by the renderer.

4.4 Parallelism considerations

The use of multiple threads in any program requires use of appropriate, thread-safe data structures and methods to prevent synchronisation issues such as race-conditions (system is dependent on the sequence of events which do not occur in the order as expected) and deadlock (two or more actions are waiting for each other to release the locks on their resources).

The queues used to pass the nodes between the different threads are susceptible to such issues and thus, are implemented using Java's thread-safe *BlockingQueue* data structures. A *ConcurrentHashmap* is used to implement the *Data Store* in Figure 4.1.

4.5 Shaders and lighting

Well-rendered geometry that has is correctly lit in order to view subtle detail is just as important as achieving a good frame rate. This applies even more so to architectural models which can have smooth ripples on plastered walls and small cracks in the buildings. For restoration purposes, it is important to highlight the presence of these small details. Ambient lighting (a fixed-intensity light that affects all surfaces equally), on its own, is not sufficient for this purpose.

For this reason, I added a that follows behind the viewer as they navigate the model. To reduce the white-wash effect on the geometry due to the light being directly behind the viewer, I offset the light slightly above the viewer. This light greatly enhances the appearance of subtle detail.

Some models provided by the Zamani Project have colour information baked into them (each vertex has colour information). This further enhances detail in the model as it does not look as flat as models without colour information. The colour information comes at a cost, however: the interior of buildings appear black as there was no lighting inside the structure during the laser scan. Thus, the ability to toggle colour information on and off was added.

5 | Results

The viewing client was tested on a variety of models and the performance results are presented in this chapter.

5.1 Test setup and system

Testing was done in the form of a fly-through of the models using a macro recording program to ensure the same path was navigated in each test. The flight path is presented in Figure 5.1. The path circles around the model and then moves through the centre of the model and back to test operations that cause the hierarchy to expand to the highest LOD nodes. All tests were performed on a laptop with the following specifications:

- Intel Core i7-3630QM @ 2.4 GHz
- 8 GB RAM
- GeForce 650M 2 GB GPU
- 750 GB secondary hard drive (5400 RPM)
- Windows 8 OS

The system had a 120 GB SSD as its primary disk for the operating system, but all processed model files were read from the secondary HDD. The Java Virtual Machine (JVM) was run in 64 bit mode with a maximum heap size for all models of 5 GB. GPU Memory usage was monitored using a GPU benchmark program called GPU-Z.

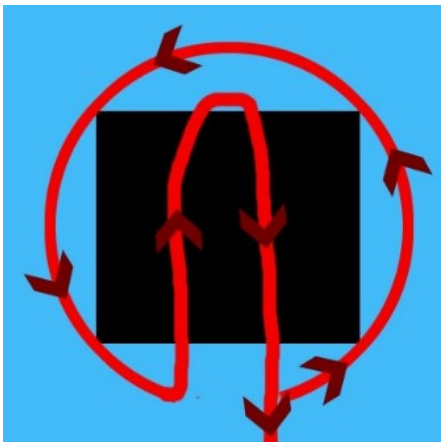


Figure 5.1: Flight path taken for each test takes a circular movement around the model before navigating down the middle. This middle path ensures metrics are tested on the most detailed nodes as the hierarchy is expanded down to the bottom-most nodes.

The results for each metric tested were averaged over 5 tests to reduce the impact any anomalies. Before each model's test, a different model was viewed to ensure the JVM and the operating system did not have any data of the current model cached anywhere. I decided to perform this cold-start method after noticing considerably reduced read times when viewing the same model more than once in a row.

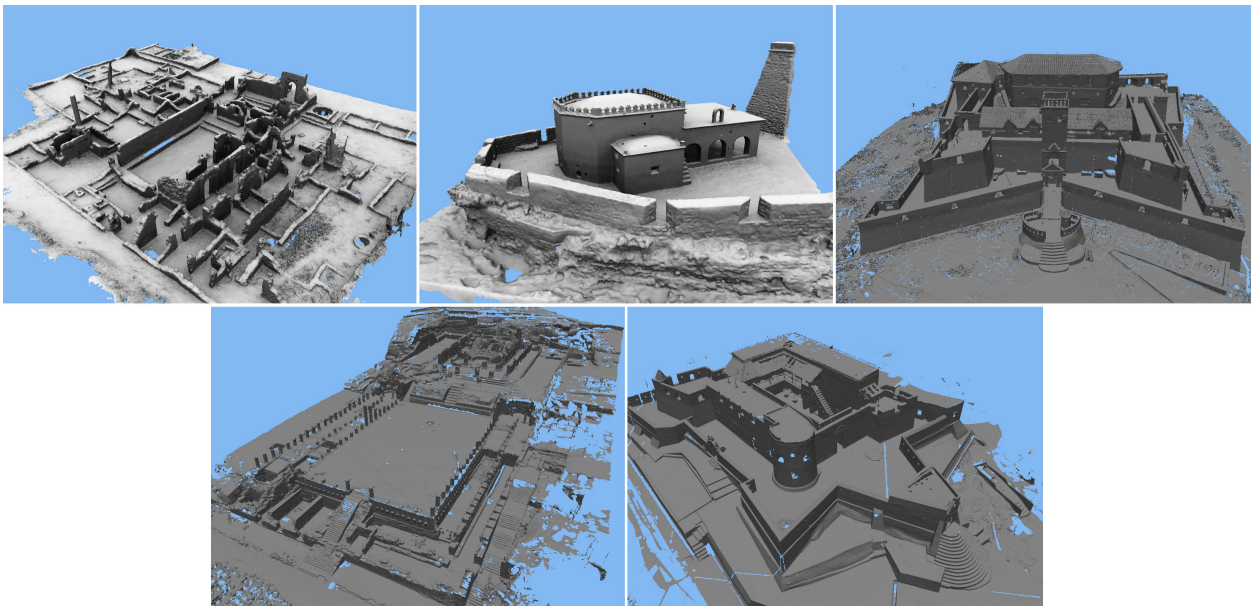


Figure 5.2: Screenshots of the models tested. From top-left to bottom-right: Gede Palace (with colour data), Chapel of Nossa (with colour data), Jago, Great Temple, Shama Fort.

5.2 Performance

Table 5.1 shows the various performance statistics for each model (see Figure 5.2) tested using the defined flight-path. At the time of testing, Shama Fort was the largest model we had been provided with by our client. The metrics recorded in Table 5.1 were chosen because they provide an indication of performance of the suspected bottlenecks in most rendering pipelines. FPS and GPU memory usage show how much load the GPU is under, while loading and processing times should have little to no impact on FPS due to my multi-threaded solution.

	Gede Palace	**Chapel of Nossa**	**Jago**	**Great Temple**	**Shama Fort**
Original size (MB)	67	207	617	622	1 635
Original no. of polygons	3 000 000	9 452 006	33 568 441	33 936 243	89 916 920
Min. FPS	59.5	57.1	57.1	56.9	46.78
Avg. FPS	59.9	59.8	59.6	59.7	58.4
Avg. visible polygons	799 798	1 376 886	1 414 874	954 929	2 679 078
Loading time (S)	2.62	1.2	18.82	14.74	33.71
Avg. processing time (S)	0.47	1.21	2.41	1.99	4.47
Max GPU Memory (MB)	162	366	626	512	948

Table 5.1: Numerical results for rendering processed models of varying size. FPS was capped at a maximum of 60 as this is usually the refresh rate of most displays.

The data in Table 5.1 shows the viewer scales well with increasing model sizes as it maintains a consistently high average frame rate when rendering models that could previously not be opened in MeshLab (e.g. Jago and Shama Fort). It appears the proposed HLOD-based solution is doing a good job at reducing the proportion of high-detailed geometry rendered each frame. The average number of polygons being rendered in each frame is significantly smaller than the number of polygons found in the original mesh. The total time the loading thread spends reading in data blocks is surprisingly low. Two possible reasons for this are that 1) the nature of the flight-path is such that only a subset of the high-detailed nodes are required and 2) it is possible that the operating system is performing its own form of caching of data that is requested multiple times. The frames-per-second (FPS) was relatively steady - well above the set target interactive frame frame of 30 FPS, and although it does deteriorate when rendering models with higher polygon densities, it seems unaffected by the disk read speed and time spent loading data. This indicates that moving the processing and loading of data to their own threads removed the feared bottlenecks and reduced pipeline stalls.

The results in Table 5.1 rely on the components before the viewer performing their tasks well and therefore are more representative of the performance of the system as a whole. The comparison between the original number of polygons and the average polygons rendered can be affected by how well the simplification and mesh splitting is performed. For this reason statistics that are solely influenced by properties of the viewing client need to be considered too.

5.3 Data caching

The Data Store object in Figure 5.3 is responsible for maintaining data blocks in memory so that they do not have to be re-read from disk. The store discards data blocks that are not in either the visible or active node lists and have not been requested by the renderer for a certain period.

It is important to determine the ideal inactivity threshold value. A low threshold value results in little memory usage, but many data blocks could have to be loaded, processed and transferred to the GPU again (resulting in a higher chance for gaps to appear), while a high threshold value means many nodes' data can be reused, but could result in the system running out of memory (due to never discarding nodes) and cause the viewer to crash.

	Total loading time (S)
2 second cut-off	20
8 second cut-off	18.82
20 second cut-off	15.23

Table 5.2: Results of the total time spent loading during the fly-throughs of the Jago model for various threshold values

Figure 5.3 illustrates this. I tested a low, medium and high inactivity threshold value and recorded the memory usage and total GPU data transfer per frame during the fly-throughs. The results are what one might expect: the increasing values are associated with increased memory usage and decreased GPU data transfer (i.e. the program is using data already loaded and transferred to the GPU). This is supported by the total loading times illustrated in Table 5.2 for the same tests. The memory usage tends to cause spikes as a result of Java's garbage collector periodically recycling the discarded data blocks and the temporary data structures required during the processing step.

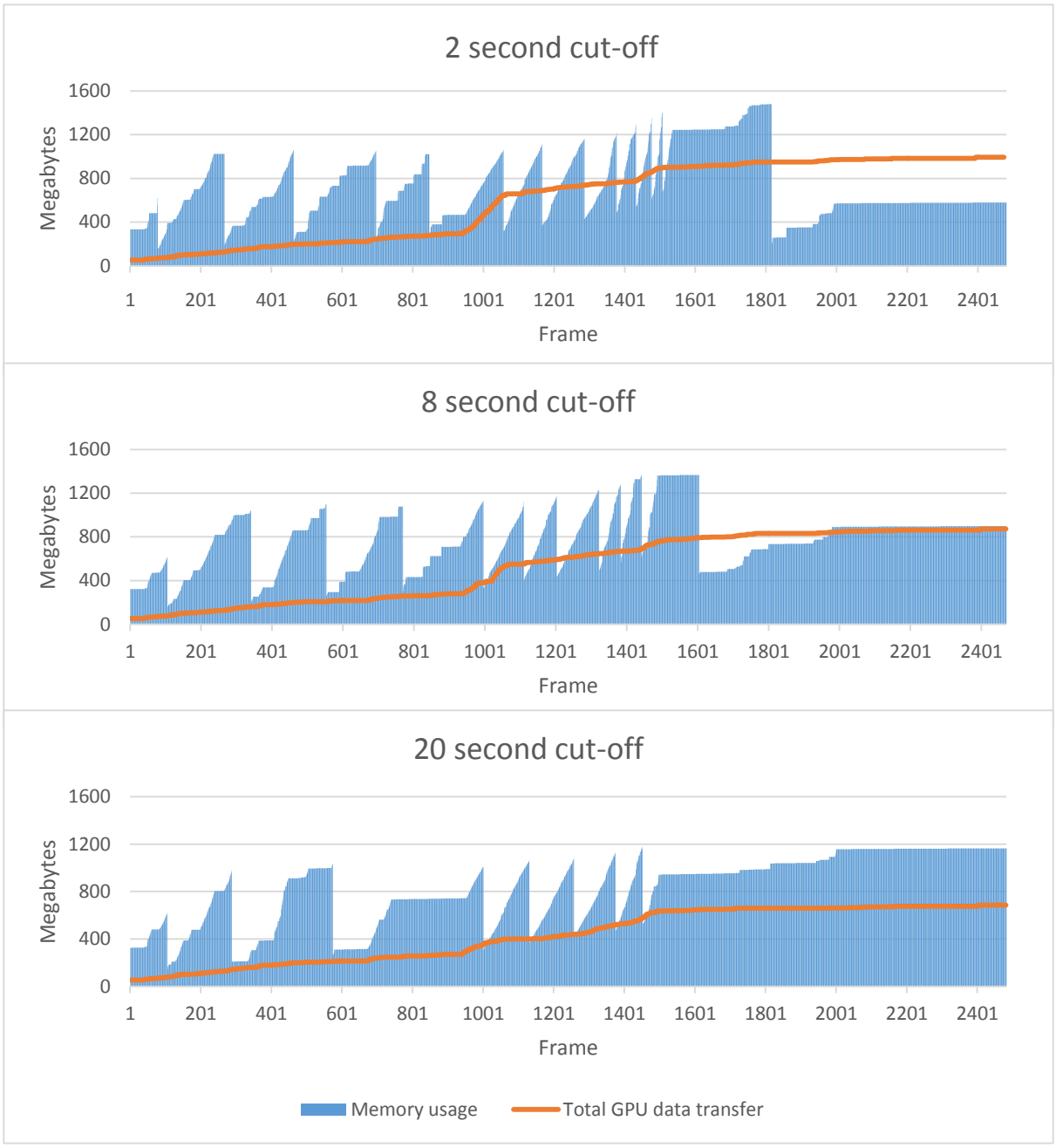


Figure 5.3: System memory usage and total GPU data transfer graphs whilst navigating the Jago model. For both sets of data, less is better.

An interesting point to note is that the spikes in memory usage when using a 2 second threshold are taller or at least equal in height to the spikes when using an 8 second threshold. While the average memory usage is less (especially further into the test), the node processing step requires additional memory temporarily. The number of nodes that require processing

is more (because they are recycled more often at the 2 second threshold) which explains the unexpected amount of memory used. I chose to make use of an 8 second threshold because it provides a balanced trade-off between memory usage and node processing.

5.4 Hierarchy operations

Another important pair of values to optimise is the threshold values for the expansion and collapsing operations performed while traversing the hierarchy. These values are expressed as a percentage of the screen area a node's projected bounding box occupies. If the expansion threshold is too high, the viewer must be very close to before the next level of detail for a node will be loaded, but if it is too low, too many nodes will expand, resulting in increased data loading, high transfer to the GPU and the GPU will struggle to render too many polygons. A similar concept applies to the collapse threshold value: too high and it will reduce detail too soon, but if it is too low, it will rarely collapse any nodes.

I also discovered a parameter tuning issue: if the two values were too close to each other, a flickering in the rendered image occurred as nodes alternated between being expanded and collapsed in successive frames. For this reason, I chose a pair of values that had sufficient range between them to avoid the flickering issue and then adjusted this range higher and lower for the tests (i.e. raised and lowered the values together, thereby maintain the same range).

Figure 5.4 shows how changing this threshold range affects the LOD the geometry is rendered at and hence, the number of polygons per frame. The values "90/35" mean a node must occupy more than 90% of the screen area to be expanded and less 35% to be collapsed. The 60/5 range renders between 20-50% more polygons than the 70/15 range. This is because this range favours expansion more than collapsing, resulting in an increased number of high-detailed level 6 nodes being rendered, reaching a peak of 9.8 million polygons. The number of polygons rendered per frame is directly associated with the number of frames per second the GPU can render. While it is important to render high-detailed geometry for architectural purposes, this should not come at the cost of such decreases in performance.

The 90/35 range does the opposite: It favours collapsing nodes more than expanding them. This can be seen from the much thinner bands of HLOD-5, -6 nodes than the other two ranges. The proportion of low level nodes is greater in this range and will render a more bland-looking image that requires one to zoom closer to a piece of geometry to view it in higher detail. I opted to use the 70/15 range which provides a balance between performance and detailed geometry. Table 5.1 shows very good FPS (almost 60 FPS for all models) that was achieved with this range

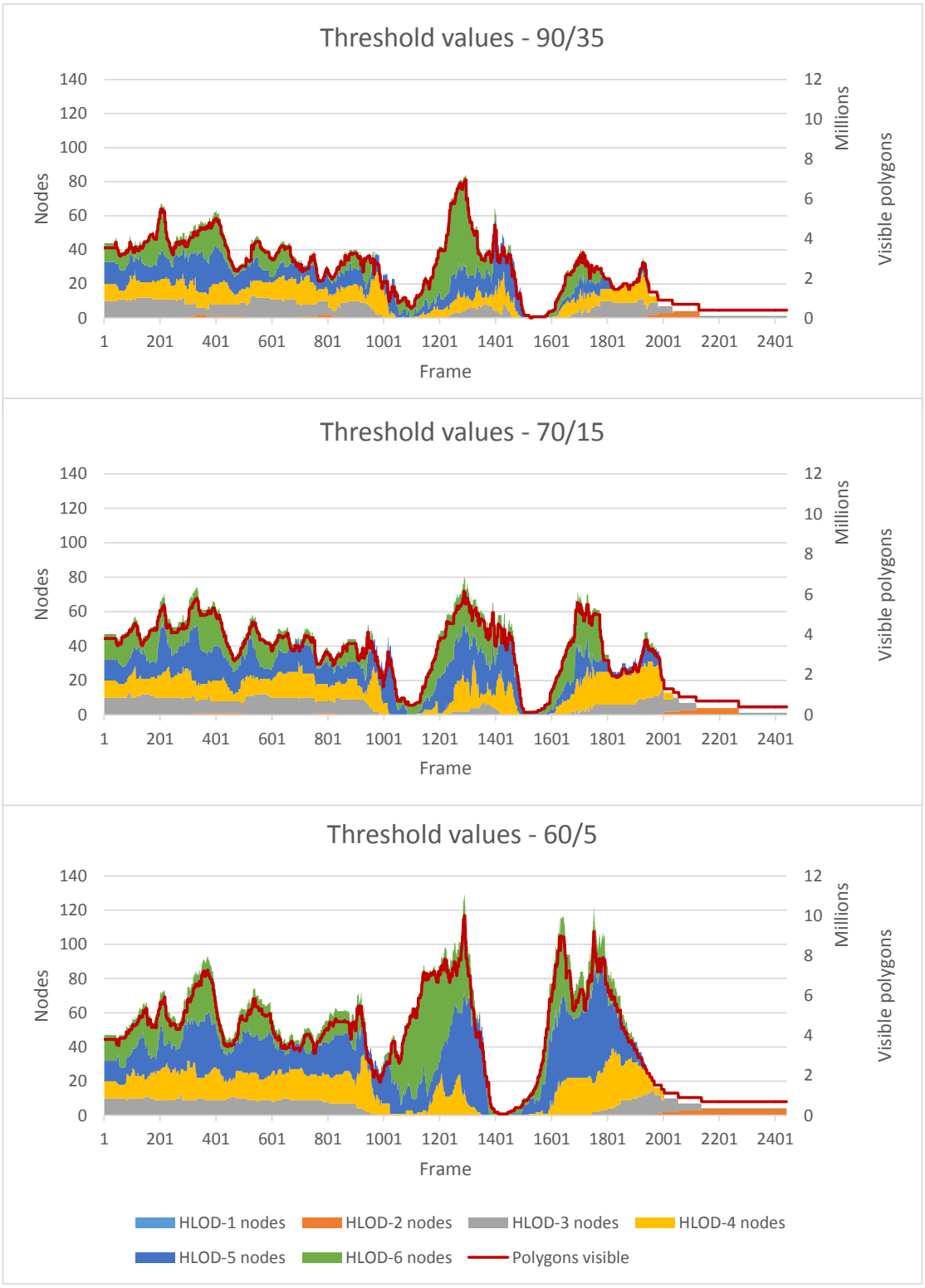


Figure 5.4: Graphs showing the number of nodes at each level of the hierarchy being rendered in each frame and how it affects the total number of polygons rendered. The large dips near frames 1080 and 1500 correspond to camera moving past the boundaries of the model before and after the first pass down the middle of the model in Figure 5.1, before it turns around for the return pass. At these points, little to none of the nodes are visible and thus, no polygons are rendered.

To show how the expansion and collapse thresholds have an effect on memory usage, Figure 5.5 shows the number of active and visible nodes per frame. Due to the nature of a tree-structure such as the hierarchy, the number of active nodes increases exponentially the further down the hierarchy is expanded. This is shown by the two spikes in the number of active nodes when the fly-through proceeds down the middle of the model and back, fully expanding branches of the hierarchy. Even though they are not visible, active nodes will still have their data loaded and will consume additional memory. Thus, the 70/15 range also finds middle-ground in memory consumption compared to the 90/35 and 60/5 ranges.

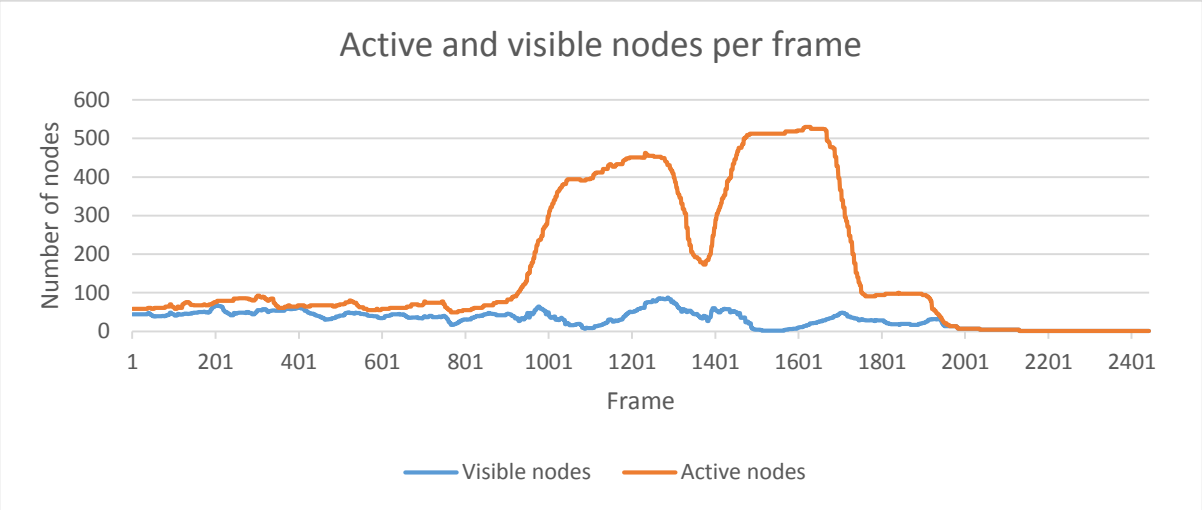


Figure 5.5: Number of active and visible nodes used per frame. The two spikes of active nodes appear during the pass down the middle of the model, resulting in many node expansions. The dip between the spikes occurs as a result of the first pass moving beyond the edge of the model before turning around for the return pass.

Most importantly, the peaks in many of the figures in this chapter show how many nodes could potentially and wastefully be loaded and rendered in the absence of an effective hierarchical culling algorithm such as the one I have implemented.

5.5 Visual appearance

The proposed solution does a good job at culling away unnecessary polygons to improve the achieved FPS, but at the cost of less-detail and a few noticeable artefacts.

One such artefact occurs when a node is expanded or a set of nodes is collapsed between frames. In the latter scenario, a parent node is rendered until its child nodes' geometry has been loaded. Once it is loaded, swapping out a parent node's geometry for the more detailed geometry in the child nodes produces a small, but noticeable, "pop" of extra detail

of geometry that is close to the viewer. Two factors affect how noticeable this artefact is, namely: it is less noticeable the further away the geometry is from the viewer and it is also less noticeable with an increased number of levels in the hierarchy. The artefact also occurs more frequently if the viewer navigates around the model faster than the queue of nodes can be loaded and processed.

Additionally, as anticipated in Section 3.2.3, the loading of data in Case 1 of No node $\rightarrow$ new node produces the most artefacts in the form of gaps appearing in the rendered geometry (see Figure 3.3). I attempted to reduce the duration and severity of the gap by loading nodes closest to the viewer first.

Figure 5.6: The effects of offsetting the light slightly above the viewer to reduce the white-wash effect. The image on the left illustrates how the white-wash effect reduces the visibility of the subtle ripples in the wall that are visible in the right image.

The additional, offset viewer-tracking light did a good job at lighting geometry the viewer is looking at and enhancing fine detail in the surfaces of walls and other geometry. Overall, the geometry still looks quite flat (see Figure 5.6) in models that do not have colour information baked into then, forcing me to use a constant, default colour in its place. The lack of any form of occlusion of shadows also makes cracks and chips difficult to notice.

6 | Conclusion

The Zamani Project faces the problem of not being able to view the architectural models their work outputs in the program they use, MeshLab. The poor viewing performance (in terms of FPS) of MeshLab makes it difficult for them to navigate the models - input from the mouse and keyboard is delayed and causes stuttering. The aim of this project was to produce a set of tools which rendered these models at interactive frame rates (defined as 30 FPS). Our result: a set of tools composed of a simplifier, a level-of-detail file generator and a viewing client. The viewing client was my responsibility and the focus of this report.

We opted for a hierarchical level-of-detail (HLOD) based scheme for two main reasons: a proven solution for rendering large scenes and quantities of geometry, based on previous work, and ease of implementation - it would fit into our time constraints.

A HLOD-based scheme is one of the more simpler solutions to the problem we faced of rendering large models, but in response to our research questions set out in Section 1.3 (whether a level-of-detail approach is a viable solution for processing and rendering large models), it is definitely a viable solution and it has proved to render large models at interactive frame rates. The viewing client is not without its flaws though. The rendered geometry of models lacking colour information look "flat", gap artefacts are still prevalent in certain cases and there is slightly noticeable loss of detail when zoomed out from a model.

While the viewer could render all the models with relative ease as the FPS results in Table 5.1 show, due the limited number of large models available for us to test on, it is not certain how performance will scale with even larger models. Nonetheless, the viewer already copes with previously un-renderable models the Zamani team has tried to view in MeshLab, our benchmark application, achieving a minimum and average FPS of 46 and 58 respectively for the largest model tested. The results also showed that the FPS and loading data from disk were successfully decoupled, with the FPS remaining largely constant when compared to the increased model sizes and loading times required.

The viewing client was a great success, both on the individual level and the integration between the various components as a group. It exceeded the key success indicator of achieving at least an interactive frame rate (defined as at least 30 FPS) for all the models that were tested.

6.1 Future work

Future work and improvements on the viewer, given more time, would likely address the shortcomings above.

Investigation is needed into more advanced shading techniques such as enhanced edge silhouettes and ambient occlusion [adv] to address the issue of the geometry looking flat and thereby making detailed geometry such as cracks less noticeable.

To eliminate or reduce the gap artifact noted in Figure 3.3, more research on the pre-loading mechanism is required. For example, one could implement a prediction algorithm based on dead-reckoning used in games [PW02] and motion-related problems to predict which nodes are likely to come into view based on the current motion of the viewer. This would reduce the delay between when a new node is request and when it has been loaded and processed.

Appendices

A | Performance test data

	Test 1	Test 2	Test 3	Test 4	Test 5
Min. FPS	57.9	59.9	59.8	59.9	59.9
Avg. FPS	59.9	59.9	59.9	59.9	59.9
Max. FPS	60	60	60	60	60
Avg. visible polygons	799798	799798	799798	799798	799798
Loading time (ms)	2760	2568	2388	2596	2498
Avg. processing time (ms)	395	503	457	550	467
Max GPU Memory (MB)	162	166	164	158	162

Table A.1: Gede Palace raw test data.

	Test 1	Test 2	Test 3	Test 4	Test 5
Min. FPS	54.5	59.8	59.9	57.1	54.4
Avg. FPS	59.7	59.8	59.9	59.8	59.6
Max. FPS	60	60	60	60	60
Avg. visible polygons	1376886	1376886	1376886	1376886	1376886
Loading time (ms)	1181	1186	1140	1139	1363
Avg. processing time (ms)	1163	1179	1184	1151	1351
Max GPU Memory (MB)	360	366	363	374	368

Table A.2: Chapel of Nossa raw test data.

	Test 1	Test 2	Test 3	Test 4	Test 5
Min. FPS	56.1	58	56.8	57	58
Avg. FPS	59.5	59.8	59.4	59.7	59.8
Max. FPS	60	60	60	60	60
Avg. visible polygons	1414874	1414874	1414874	1414874	1414874
Loading time (ms)	19185	18037	19561	18537	18790
Avg. processing time (ms)	2367	2422	2647	2396	2232
Max GPU Memory (MB)	636	622	624	632	616

Table A.3: Jago raw test data.

	Test 1	Test 2	Test 3	Test 4	Test 5
Min. FPS	56.2	57.1	57.1	56.2	57.7
Avg. FPS	59.6	59.8	59.8	59.8	59.7
Max. FPS	60	60	60	60	60
Avg. visible polygons	954929	954929	954929	954929	954929
Loading time (ms)	16792	16184	11099	15442	14207
Avg. processing time (ms)	2165	1942	1988	1922	1953
Max GPU Memory (MB)	517	502	531	515	498

Table A.4: Great Temple raw test data.

	Test 1	Test 2	Test 3	Test 4	Test 5
Min. FPS	49.5	39.5	45.6	47.4	51.9
Avg. FPS	58.2	58.1	58.6	58.6	58.7
Max. FPS	60	60	60	60	60
Avg. visible polygons	2679078	2679078	2679078	2679078	2679078
Loading time (ms)	33568	33253	33447	33835	34425
Avg. processing time (ms)	4686	4308	4498	4321	4522
Max GPU Memory (MB)	942	1002	898	926	974

Table A.5: Shama Fort raw test data.

References

[adv] Advanced Shaders. http://www.techsoft3d.com/developers/getting-started/hoops-visualize/advanced-shaders. Accessed: 28 October 2014.

[AM00] Ulf Assarsson and Tomas Moller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000.

[Bea] Josh Beam. Tutorial - Introduction to Software-based Rendering: Triangle Rasterization. http://joshbeam.com/articles/triangle_rasterization/. Accessed: 17 August 2014.

[Fun96] Thomas A Funkhouser. Database management for interactive display of large architectural models. In *Graphics Interface*, volume 96, pages 1–8, 1996.

[GZPG10] Prashant Goswami, Yanci Zhang, Renato Pajarola, and Enrico Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *Computer Graphics and Applications (PG), 2010 18th Pacific Conference on*, pages 93–100. IEEE, 2010.

[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.

[HSH09] Liang Hu, Pedro V. Sander, and Hugues Hoppe. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 169–176, New York, NY, USA, 2009. ACM.

[KNS] Mandakini Kaushik, Kapil Kumar Nagwanshi, and Lokesh Kumar Sharma. A overview of point-based rendering techniques.

[Lak04] Ali Lakhia. Efficient interactive rendering of detailed models with hierarchical levels of detail. In *3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on*, pages 275–282. IEEE, 2004.

[Lin03] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 93–102. ACM, 2003.

[pip] Rendering Pipeline Overview - OpenGL. https://www.opengl.org/wiki/Rendering_Pipeline_Overview. Accessed: 15 August 2014.

[PW02] Lothar Pantel and Lars C Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 79–84. ACM, 2002.

[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*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[Ver] Chad Vernon. Vertex and Index Buffers. `http://www.chadvernon.com/blog/resources/directx9/vertex-and-index-buffers/`. Accessed: 8 October 2014.

[Wik] Wikipedia. Progressive meshes. `http://en.wikipedia.org/wiki/Progressive_meshes`. Accessed: 15 August 2014.

[zam] About the Project - The Zamani Project. `http://www.zamaniproject.org/index.php/project.html`. Accessed: 15 August 2014.