

Maxwell Pettett

Part II Computer Science Dissertation

Multiresolution Meshes:

Rendering One Billion Triangles

Cambridge University

Lucy Cavendish College

2023 – 2024

Declaration

I, Maxwell W. Pettett of Lucy Cavendish College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: *Maxwell Pettett*

Date: 2024-05-10

Proforma

Candidate Number: 2323G

Project Title: Multiresolution Meshes: Rendering One Billion Triangles

Examination: Computer Science Tripos – Part II, 2024

Word Count: 11871¹

Code Line Count: Rust: 15847, GLSL: 632, Python: 300²

Project Originator: The Candidate

Supervisor: Dr. Rafał Mantiuk

Original Aims of the Project

The aims of this project were to research, implement, and explain techniques for rendering multiresolution meshes. I wanted to construct a platform for creating and rendering multiresolutions that was both extensible and integratable into existing 3D engines. This required researching the creation of these structures, how to efficiently query them for real-time approximations of meshes, and how to render the selected approximations in real-time.

Work Completed

All aims above, and multiple extensions, have been completed. I have created a series of applications to generate, render, and evaluate multiresolution meshes, all available open-source. The renderer is designed around render graphs, commonplace in modern rendering engines, making it ideal for integration into current systems. As part of the research aims, I developed a novel and efficient rendering pipeline using mesh shading.

Special Difficulties

None.

¹wordometer:0.1.1 taken over chapters 1 to 5, inclusive.

²tokei --exclude ash_renderer/shaders/spv totals for Rust, GLSL and Python.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Goals	2
2. Preparation	3
2.1. Multiresolution Meshes	3
2.1.1. From Trees to DAGs	4
2.1.2. Selecting Clusters from a DAG	6
2.1.3. Error functions on a DAG	7
2.1.4. Limitations	8
2.2. Mesh Simplification	8
2.2.1. Half-Edge Mesh	8
2.2.2. Edge Collapse	9
2.2.3. Quadric Error Metrics	9
2.3. Procedural Geometry	11
2.3.1. Primitive Shading	11
2.3.2. Mesh Shading	11
2.4. Existing Techniques	12
2.5. Project Development	13
2.5.1. Requirement Analysis	13
2.5.2. Libraries and Tools	13
2.5.3. Starting Point	13
2.5.4. Development and Engineering	13
2.5.5. Progress	14
3. Implementation	15
3.1. Multiresolution Generation	15
3.1.1. Generation Algorithm	16
3.1.2. Mesh Simplification	16
3.1.3. Simplification within Groups	17
3.1.4. Grouping Clusters	19
3.2. Renderer Architecture	21
3.3. Cluster Selection	22
3.3.1. DAG Traversal Cluster Selection	23
3.3.2. Adaptive Cluster Selection	26
3.4. Cluster Culling	27
3.5. Cluster Rendering	27
3.5.1. Primitive Shading	27
3.5.2. Mesh Shading	29
3.5.3. Meshlet Compression	29
3.6. Render Graph	30
3.7. Repository Overview	30
4. Evaluation	32
4.1. Mesh Generation	32
4.1.1. Discussion	33

4.2. Visual Quality	33
4.3. Rasterisation Efficiency	34
4.4. Performance	36
4.4.1. Comparing Implemented Multiresolutions	36
4.4.2. Comparing Alternate Tools	37
4.4.3. Scene Complexity	38
5. Conclusion	39
5.1. Work Completed	39
5.2. Reflections	39
5.3. Future Work	40
Bibliography	41
A Minimal Mesh Pipeline	43
B Mesh Simplification Graphics	45
C Benchmarking Scenes	46
C.1 Visual Error	46
C.2 Performance	46
D Licensing	47
E Project Proposal	49

Chapter 1

Introduction

In this dissertation, I implement the framework and tools needed to render billions of triangles in real-time. I explain and implement *multiresolution meshes*, data structures which can be efficiently sampled to approximate some high-resolution source. I use these to demonstrate rendering traditionally high complexity scenes with low visual error.

1.1. Motivation

A common desire for high-fidelity scenes in modern rendering engines has brought higher and higher resolution meshes to real-time applications. Handheld photogrammetry applications have made sourcing such meshes simpler and more commonplace.

This dissertation is not about *rasterising* 1 billion triangles, the process of converting triangles to 2D images, as it would be incredibly wasteful; a screen only has a certain number of pixels. Rasterising more triangles than pixels means some will be obscured by overlap, or exist between pixels and so thrown away by the rasteriser. This dissertation is about eliminating wasted rasterisation in the latter case, enforcing a certain screen-space size of triangle.

Real-time rendering engines typically vary mesh resolution to maintain performance in complex scenes. One may vary mesh resolution with progressively coarser approximations of the mesh, a level of detail (LOD) chain (Figure 1) generated via a process of *mesh simplification*. However, LOD chains are inherently limited in flexibility, as each object can only be rendered at a single resolution. The possibility that the same object spans large depths (for example, terrain), means a single optimal resolution per level of detail cannot exist.

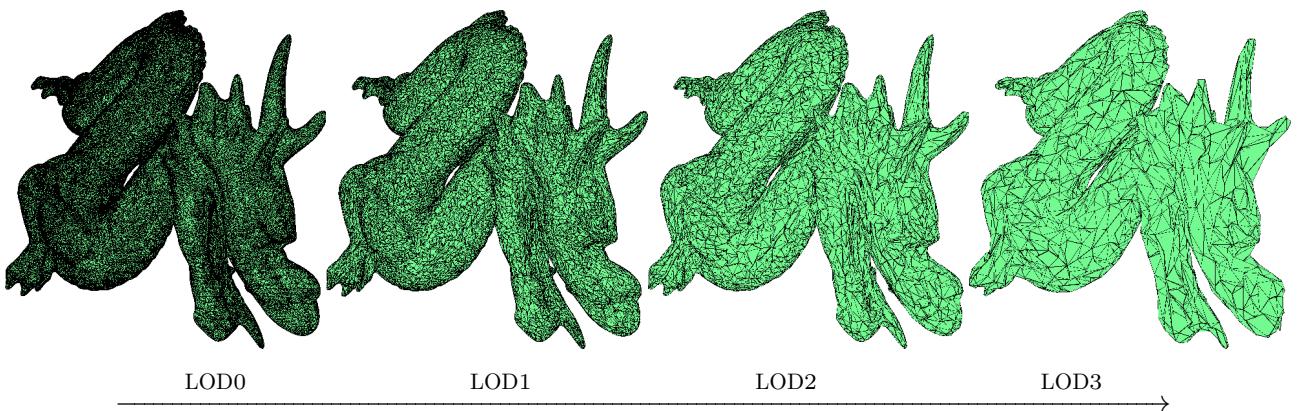


Figure 1: A LOD chain, each level with half the triangles of the last

As such, we must accept that we cannot pre-generate an approximation of a large mesh for every scenario. Instead, I will implement *view-dependant* LOD, generating these approximations at runtime, with results such as that in Figure 2. Meshes are large data structures, so, to be at all efficient, we require an acceleration structure; the multiresolution mesh that this project investigates.

A similar multiresolution technique to the one I will implement has been most prominently used in Unreal Engine 5's Nanite [1], whose source code is licensed out. It is unfortunately closely coupled with the rest of the engine so cannot be easily integrated into other tools.

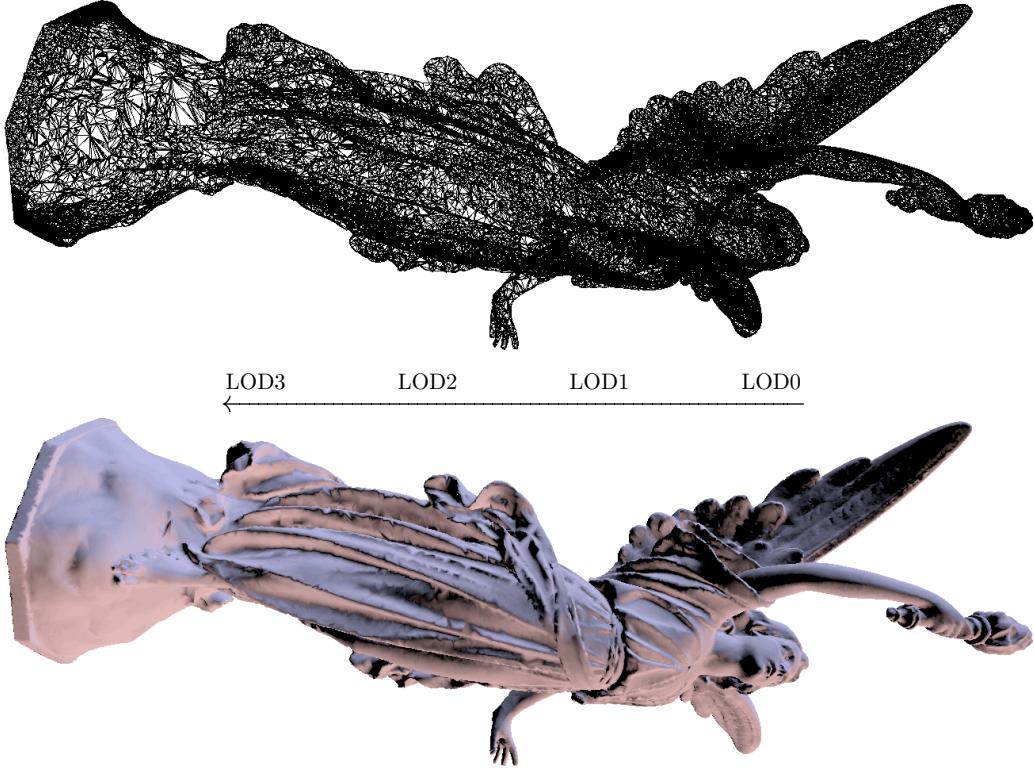


Figure 2: Fine grained multiresolution of a 28 million triangle mesh, decreasing in resolution from the viewpoint near the right. Top: generated triangulation. Bottom: lit view of the mesh.

1.2. Goals

Multiresolutions are quite difficult to implement. Challenges are split between generation, which is pre-computed, and rendering, which must be done in real-time. No open-source implementations similar to mine have yet been released, so I aim to build a foundation for future research:

Multiresolution Generation I will create a mesh simplifier that creates high-quality approximations of an original mesh and apply this to a multiresolution generation scheme.

Procedural Geometry I will investigate approaches for procedural geometry to render view-dependent LODs. The modern *mesh shading pipeline* will be a large part of this.

Selection Algorithms I will implement multiple cluster selection algorithms. Currently, Unreal Engine 5 is the primary example of multiresolutions in deployment. I wish to implement a method similar to theirs, but additionally to research alternatives that exploit the power of *mesh shading*.

Benchmarking I will evaluate the performance of my system against existing methods, and measure the quality of any algorithms I produce.

The code written for this project is available as open-source, and the novel portions of my work were awarded 3rd best paper at the CESCG student seminar.

Chapter 2

Preparation

In this chapter I will discuss 3 key topics: First, I investigate the multiresolution paradigm and methods to extract useful approximations of a mesh (§ 2.1). Next, I investigate the data structures, operations, and heuristics we use for simplifying the geometry of a mesh (§ 2.2). Finally, I examine the techniques available for real-time procedural geometry (§ 2.3).

2.1. Multiresolution Meshes

A multiresolution mesh (or simply *multiresolution*) is a data structure that encodes multiple *levels* of resolution of a mesh. They exist to optimise the rasterisation (plotting of polygons to the screen) of complex scenes by reducing the number of polygons too small to be visible. A common multiresolution is a Level of Detail (LOD) chain, a chain of progressively simpler approximations of the mesh generated by *mesh simplification*. These are not suitable for this task, as a ‘complex scene’ may contain a single complex mesh. This necessitates varying resolution across the mesh, requiring a *view-dependant* LOD. Our final scheme will be similar to Unreal Engine 5’s multiresolutions [1].

In this project, we concern ourselves with multiresolutions formed from *clusters*, each containing polygons representing a specific area of the mesh at a specific resolution. Done naïvely, mixing clusters from different levels would introduce *seams* between them, small cracks in the mesh. Mesh simplification gives no guarantees on what geometry is shared between neighbouring levels of resolution. Seamless geometry requires known shared edges between levels that can serve as a ‘bridge’ between them – the problem is deciding how and where to place these. To illustrate the difficulty of this, and the artefacts it can produce, let us define a toy multiresolution scheme, the *tree-based multiresolution*, as follows:

1. Take a set of clusters that partition the source mesh.
2. Recursively merge pairs of clusters together and simplify their contents. Edges on the boundary of pairs are **locked**, meaning they cannot be edited by the mesh simplifier.

This forms a tree containing variable resolution clusters of the mesh – each new cluster is made the parent of the pair of clusters it was formed from. Shared geometry is guaranteed by the locked edges, and clusters may be stitched together based on shared boundaries. However, this exact scheme turns out to be impractical; by only simplifying within pairs, some edges will remain locked throughout all simplification, causing the artefact visible in Figure 3. At the extreme, it bisects the mesh with a high-resolution ring of edges, visible in the last level.

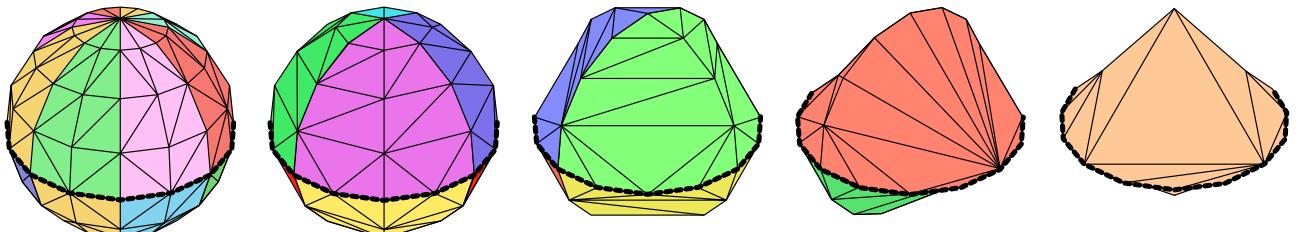


Figure 3: The tree-based multiresolution on a sphere. Coloured regions represent the working set of clusters at each level, i.e. triangle groups with *locked boundaries* for the next simplification step.

Note how it causes some edges to be locked throughout (lines marked **-----**).

This artefact can be accounted to the tree structure; it would be valid to render one half of the tree at full resolution but the other at minimum, necessitating the ring of edges to join them. To avoid these artefacts, we need a refined method that allows edges to be *unlocked* and simplified again.

2.1.1. From Trees to DAGs

As we have seen, a multiresolution can be formed of various clusters of a mesh, and the tree-based multiresolution lacks the flexibility to simplify all areas of the mesh. We can introduce this flexibility with a more general data structure. The paper “Batched multi-triangulation” [2] outlines representing multiresolutions with a directed acyclic graph (DAG), rather than a tree. The relationships are encoded similarly to the tree-based multiresolution; clusters are connected between levels if they *may overlap*.

Naturally, we can create DAGs for the two multiresolution structures we have seen thus far, as a DAG generalises both a chain and a tree. However, we have also seen that neither of these produce good results. As such, we say their DAGs are not *well-conditioned* [2].

Well-conditioned DAG

A DAG of n clusters is **well-conditioned** if:

DAG depth The length of any path connecting a root to a leaf is $O(\log n)$.

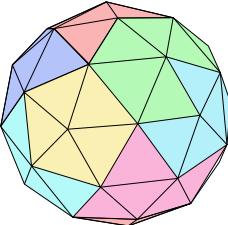
Cluster size Clusters’ diameter decreases geometrically by their depth in the DAG.

Boundary No clusters have a boundary equal to the boundary of their children.

Neither a tree nor a chain are well conditioned DAGs, as both fail the boundary check. In the LOD chain, each cluster occupies the entire surface, so the boundary is constant. In the tree-based multiresolution, the cluster formed from simplifying a pair retains the pair’s boundary.

However, the tree-based multiresolution is nearly well-conditioned, only failing the boundary check because it simplifies a pair into a *single* parent cluster. To solve this, we *split* this single parent into two parent clusters (turning our tree into a DAG). When we split the parent, we divide the area it covered between two new parent clusters, with neither new cluster sharing the boundary of the original. This allows each new cluster to pass the boundary check. However, this doesn’t reduce the total number of clusters, so fails the other two checks. The final alteration is to partition the clusters into *groups of four* instead of *pairs of two*, halving the cluster count at each level.

To reiterate, let us walk through the stages of such a scheme:

Stage	Mesh	DAG
Stage 1: Partition mesh into level 0 clusters.		

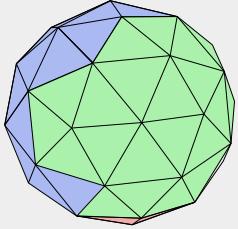
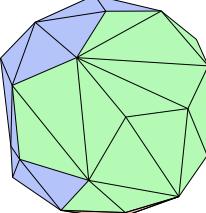
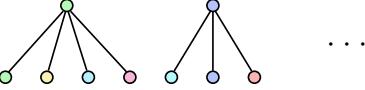
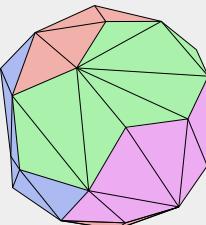
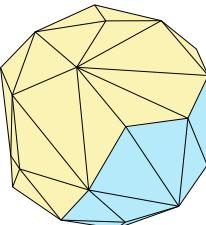
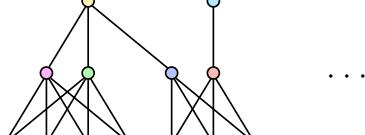
Stage	Mesh	DAG
Stage 2: Further partition clusters into groups . To end with 2 <i>parent clusters</i> , our target group size must be 4 clusters.		
Stage 3: Simplify geometry within groups, locking their boundaries.		
Stage 4: Partition groups into 2 clusters. These become the parents of the group, sharing the group's boundary on the mesh. ³		
Stage 5: Loop to stage 2, prioritising groupings across the boundaries of previous groups (and so <i>unlocking edges</i>).		

Figure 4 shows a DAG generated from this scheme. Unlike a tree, it may not be easily split into two subgraphs, mirroring the lack of long living locked edges. The DAG also shows us which clusters may be substituted for which others. For example, the top layer of the DAG contains 2 clusters sharing the same 4 children. As such, the area of the 2 parents must be exactly equal to the area of their shared children, up to and including an *identical boundary*, allowing groups to be substituted for parents. This property of pairs of parents sharing exact sets of children will allow for efficient traversal of the DAG.

This scheme satisfies the requirements for a well conditioned DAG:

DAG Depth Each iteration halves the number of clusters, so we have $O(\log n)$ depth.

Cluster Size Each group of 4 is split into 2 clusters, so their area must on average double.

Boundary A *single* cluster cannot have the same boundary as its children, as the area is split between two clusters.

³The green parent and yellow child do not actually *overlap* on the mesh, however being conservative and creating a uniformly structured DAG will allow optimisations later.

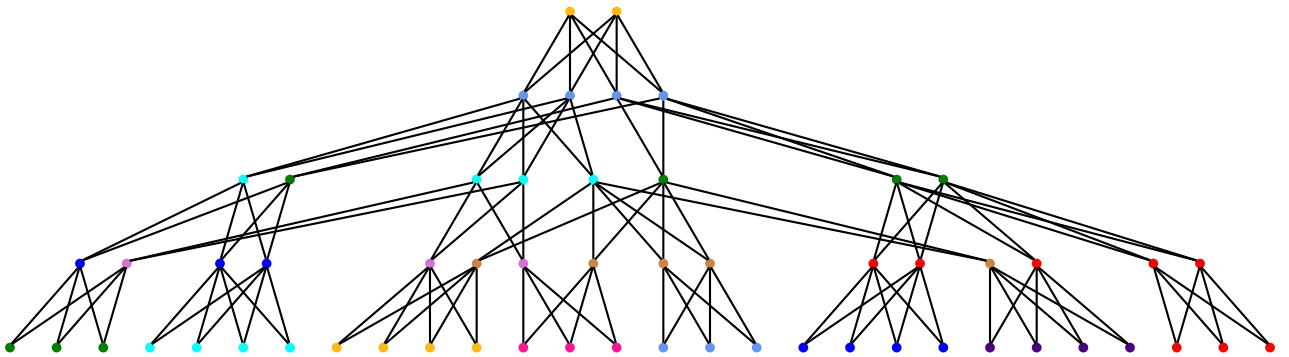


Figure 4: A DAG encoding a multiresolution mesh.

Nodes represent clusters, with nodes of the same colour sharing parents; members of the same *group*.
 Edges represent clusters that may overlap, directed top to bottom (arrows omitted).

A flexible data structure is useless if it cannot be efficiently sampled. The next two sections outline how to select clusters from a DAG that form an approximation of the original mesh, for the purpose of generating a view-dependent LOD.

2.1.2. Selecting Clusters from a DAG

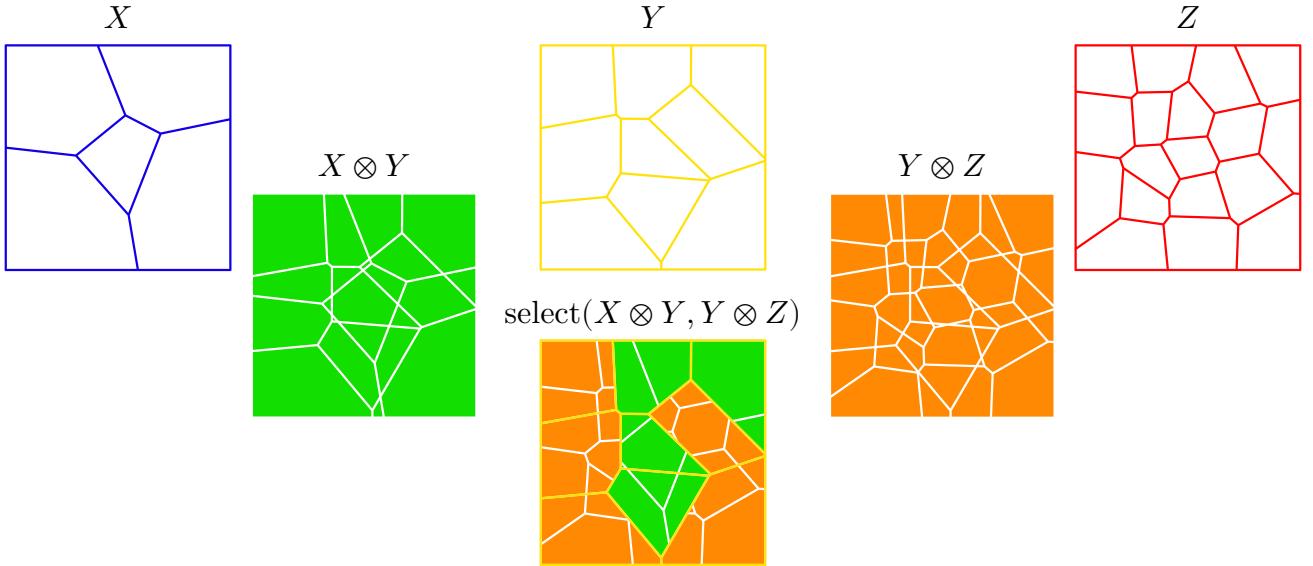


Figure 5: Sets of fixed edges (and therefore vertices), increasing resolution left to right.

Sets of clusters generated by overlaying two levels of fixed edges by an \otimes operator.

A **select** function, that we aim to quantify, forming a **valid selection** of the two sets.

To select clusters from a DAG, we cut it. Performing a *cut* will split a DAG into two disjoint subgraphs, X and Y , and we fix subgraph X to contain both root nodes. The leaf nodes of X form a *selection*. As such, a cut will produce an approximation of the mesh with a possibly varying resolution across different areas. However, this selection is not desirable without the guarantee that it is *valid*, also outlined in “batched multi-triangulation” [2]:

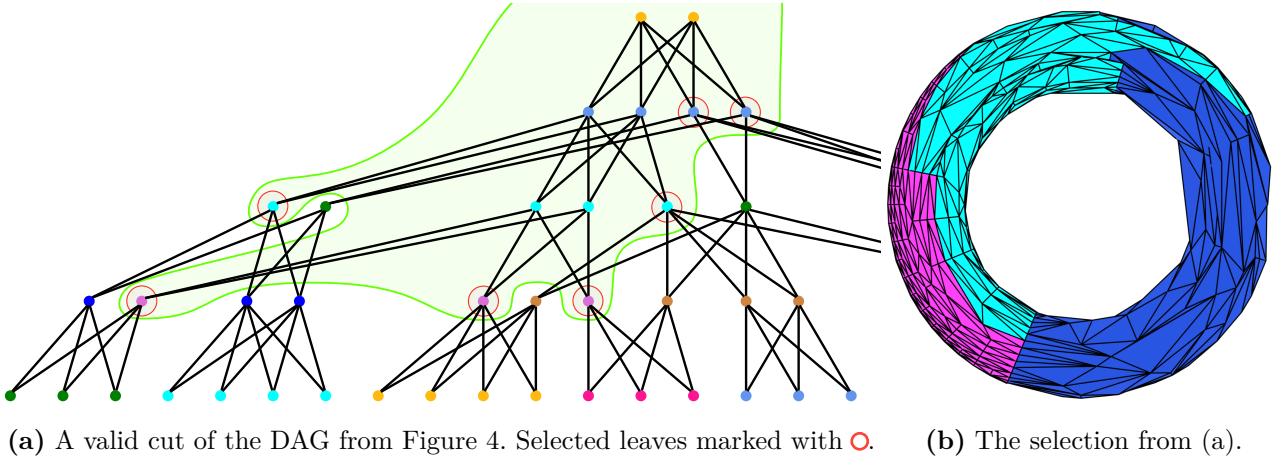
Valid Selection

A set of clusters form a **valid selection** if:

Overlap There are no clusters in the selection that overlap.

Punctures The sum of areas in selected clusters covers the mesh’s entire surface.

As such, we must add additional constraints. First, the cut must be a *dicut*, a cut of a DAG into two subgraphs X and Y such that there are no edges in the direction $Y \rightarrow X$. Second, for any valid cut, *all sibling nodes must be contained within the same subgraph*. With these two conditions, we have a valid cut, exemplified in Figure 6.



(a) A valid cut of the DAG from Figure 4. Selected leaves marked with \circ . (b) The selection from (a).

Figure 6: Cutting a DAG for a torus. Note in (b), our selection forms a seamless approximation.

Overlap Dicut – two clusters may only⁴ overlap if there is a path on the DAG between them. However, no leaf of the dicut subset X can reach another leaf; the only possible outgoing edges for a leaf are to the subset Y , from which there are no edges back to X .

Punctures Recursive Substitution – we have that, for any node, the subset must include all sibling nodes. As such, it can be generated by replacing pairs in the selection by their children, starting with the root nodes. We have seen that the area occupied by a pair of parents is identical to their shared children, and, as the two root clusters contain no punctures by definition, the final selection must also contain no punctures.

2.1.3. Error functions on a DAG

This section focuses on sampling a *good* selection of clusters. To generate a view-dependent LOD, it is useful to estimate how clusters appear to the user. These are their *screen-space errors*, the perceived incorrectness introduced by mesh simplification per unit area occupied on the screen. As we desire a uniform visual appearance, we will sample good clusters according to a user-defined threshold on screen-space error.

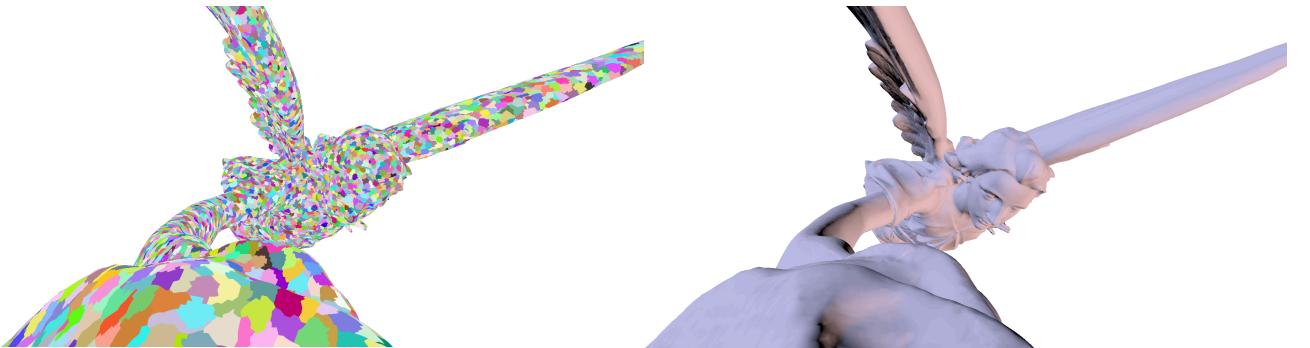


Figure 7: Viewpoint of scene in Figure 2. The error function allows clusters to be uniformly sized in screen-space, excepting those near to the camera which are already at the maximum resolution.

⁴It is possible, during generation, that parent clusters are assigned such that they do not overlap all in the original group, and so share no area. However, ignoring this loses no flexibility and makes reasoning simpler.

This screen-space error should be **projected** from a metric of the inherent visual quality of a cluster. This is its *object-space error* δ , and should be pre-computed, free from the restrictive time constraints of real-time rendering.

Having considered the goal, we must now define an error function for a cluster i , $\text{err}(i)$. To convert a cluster's error from object-space to screen-space, we must estimate its size on the screen by *projection*. We assign each cluster i a bounding sphere, with centre c_i and radius r_i , a volume in object-space that bounds the cluster. Recall each cluster also contains an object-space error δ_i . I use a method similar to *batched dynamic adaptive meshes* [3] to project the object-space error to screen-space error, $\text{err}(i)$, for eye position e :

$$\text{err}(i) = \delta_i \frac{r_i}{\|c_i - e\|} \quad (1)$$

We may fully exploit GPU parallelism if applying a threshold to the error function creates a single *unique* cut. Given a unique cut, clusters may be validly selected based on only their relations to their neighbours. This requires that our error function be *monotonically decreasing down the DAG*.⁵

To ensure the error function is monotonic, we must ensure both factors of $\text{err}(i)$, the object-space error and the projected screen-space area are monotonic. As the two factors should always be > 0 , the monotonicity will extend to our error function. Average edge length monotonically decreases down a well-conditioned DAG, as clusters double their triangle density at each level, so makes for a good object-space error representation [1]. To ensure projected screen space area is monotonic, we ensure that each cluster's bound contains all bounds of their children, creating a *nested boundings volume hierarchy* [2].

2.1.4. Limitations

DAG based multiresolutions have long-standing limitations, beyond the scope of my project:

Aggregate geometry Geometry that gains its shape from the effect of many disconnected primitives; it is greater than the sum of its parts. A common example is a mesh of a bush or leaves on a tree. Mesh simplifiers commonly have sub-par results from these, and good results require artists to manually create LOD chains; our multiresolutions are near impossible to create by hand.

Skeletal animation A technique for animation whereby bones are inserted into a mesh, and vertices are moved based on their bones. It is technically possible to implement an error function that is compatible with these transformations, but I will not attempt it.

2.2. Mesh Simplification

This section outlines choices of data structure, primitive operation, and guiding heuristic we will use for the simplification stage of generating our multiresolutions.

2.2.1. Half-Edge Mesh

To reason about mesh simplification, we need a data structure that allows efficient mesh alteration. The standard encoding of a mesh is a *triangle list*, containing 3-tuples of indices (i, j, k) defining a triangle with vertices (v_i, v_j, v_k) in anti-clockwise order. This GPU-optimised format does not encode relationships between structures, making it unsuitable

⁵Root nodes have the highest error, as they are the lowest resolution, see Figure 16.

for our task. For example, finding the triangles a vertex connects to or the neighbours of a triangle can only be done by searching the whole list. We desire such adjacency testing to have $O(1)$ time complexity.

The half-edge mesh structure is an alternative to triangle lists. As a *boundary representation*, it encodes the boundaries of faces rather than faces themselves. To encode a triangle (v_i, v_j, v_k) we store 3 directed edges $v_i \rightarrow v_j$, $v_j \rightarrow v_k$, and $v_k \rightarrow v_i$. Each of these edges contains redundant state, shown in Figure 8 [4].

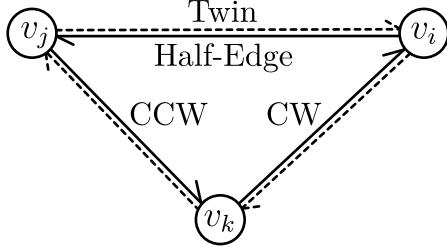


Figure 8: The references within a half-edge: the *twin*, *counter-clockwise (CCW)*, and *clockwise (CW)* edges. Dotted edges may not exist, depending on topology.

2.2.2. Edge Collapse

To simplify a mesh we need to reduce the number of triangles. Edge collapse for a half-edge $v_i \rightarrow v_j$ is an operation on a mesh that shifts and merges vertex v_i to v_j , then removing itself, its twin, and the one or two triangles attached to them (Figure 9) [5], [6].

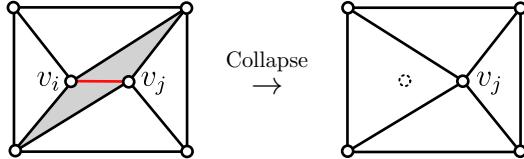


Figure 9: Edge Collapse. The two shaded triangles are made *degenerate*, and are removed.

Some edge collapses are *invalid*, and would deform the mesh in undesirable ways, such as causing triangles to overlap [7]. It is possible for an edge collapse to pull two neighbouring triangles into a single triangle, shown in Figure 10. This issue is caused by the two vertices of the collapsing edge sharing *more than one neighbour on the same side*.

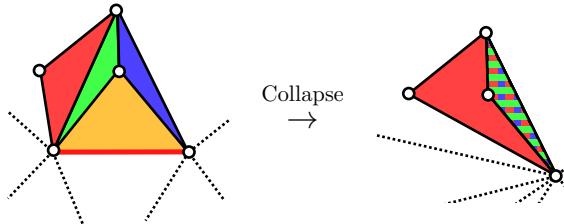


Figure 10: Invalid edge collapse creating overlapping geometry.

Additionally, we must check our collapse does not flip the normals of any attached triangles.

2.2.3. Quadric Error Metrics

We can simplify a mesh through the process of iterative edge collapse. For this purpose, we need a metric that describes the error introduced by collapsing some edge that can be used to select optimal collapses. Mesh simplification should preserve uniform tessellation (nicely shaped triangles) and feature preservation (the object's shape). We want to measure error as

a deviation from the original *shape*. As such, our metric should not be affected by collapses across a locally flat surface of coplanar vertices, as they introduce little to no visible error. Conversely, it should add more error the larger the mesh’s curvature, as this corresponds to larger changes in shape. With these goals in mind, I used the quadric error metric [6].

Edge collapse introduces small translations to vertices. A translation in vertex position can be measured with the distance from the altered vertex to the source shape. As the translations are small, this can be estimated as the distance to its original surrounding triangles. Let $\text{Planes}(v)$ be the set of planes generated from the triangles with vertex v . To estimate error as defined above efficiently, we calculate the sum of square distances of the planes formed from these triangles:

$$\Delta(v, \text{planes}) = \sum_{p \in \text{planes}} \text{sqrdist}(v, p) \quad (2)$$

However, we do not want to carry around a large set of planes for each vertex. Square distance from a plane $p = [a b c d]$ such that $ax + by + cz + d = 0$ can be encoded with a *planar distance matrix* $K_p = p p^T$. The function $\text{sqrdist}(v, p)$, for a homogeneous vertex $v = [x y z 1]$, can then be defined as

$$\text{sqrdist}(v, p) = v^T K_p v = v^T p p^T v \quad (3)$$

As matrix multiplication is distributive, these planar distance matrices can be added to form a matrix that represents multiple planes. We define an error matrix Q_i for $\text{Planes}(v_i)$ as:

$$Q_i = \sum_{p \in \text{Planes}(v_i)} K_p = \sum_{p \in \text{Planes}(v_i)} p p^T \quad (4)$$

When estimating the error of collapsing $v_i \rightarrow v_j$, we can compute the error of v_i against the union of all affected planes, representing the topology of v_i and v_j that would need to be encoded solely by v_j .

$$\Delta(v_i \rightarrow v_j) = \Delta(v_i, \text{Planes}(v_i) \cup \text{Planes}(v_j)) = v_i^T (Q_i + Q_j) v_i \quad (5)$$

We also wish to remember the shape of the mesh as we collapse edges to ensure we retain the original mesh’s features. Vertices then need some notion of history, which we can gain from simply updating $Q_j := Q_i + Q_j$ on a collapse $v_i \rightarrow v_j$. We can now store a single matrix per vertex, taking $O(1)$ time to estimate collapse error.

An additional desirable property is that boundary edges⁶ should be inclined to stick to their boundaries, as there can be a high visual error to altering the silhouette of a shape. As such, boundary vertices are given additional quadric error from the *plane of the boundary*. This plane is calculated with a vector of the boundary edge \vec{e} crossed by the normal of the edge’s face \vec{n} , $\vec{e} \times \vec{n}$, and naturally also passes through the boundary vertex. Motions away from a boundary introduce higher-than-normal visual error, so we multiply this plane’s matrix by a high scalar value.

⁶The boundary of the original shape; the locked edges in multiresolution generation will be considered later.

2.3. Procedural Geometry

This section focuses on the two methods feasible to us to draw our multiresolution meshes, outlining the available methods of computing procedural geometry on a GPU. This is a terminology heavy section, in part relying on content covered in IA Graphics and IB Introduction to Computer Architecture. Some primary definitions:

Shader General term for GPU code.

Shading Pipeline A series of shaders run in stages to render something. Each stage of execution is an abstraction that defines the inputs, the data and parallelism accessible to it, and its outputs.

Invocation A thread of GPU code running as part of a ‘work group’ with shared memory.

Dispatch Workgroups are dispatched by the CPU (normally), similarly to an `async` function call. Many workgroups are created at once, with indices assigned in a 3D grid.

Primitive A unit of geometry, most often triangles but also lines and points.

Interpolants Data passed to the GPU’s *rasteriser*. Each pixel covered by the primitive will be passed data interpolated from the interpolants, based on the pixel’s barycentric coordinates within the triangle, point on a line, etc.

2.3.1. Primitive Shading

The primitive shading pipeline is the standard shading pipeline present in all major rendering APIs. The primitive rendering pipeline works on the triangle list mesh format, accepting an index and vertex buffer. A *vertex buffer* contains per-vertex data and an *index buffer* contains a triangle list. We pass these primitives between the following *stages*:

Vertex Stage Apply transformations to individual vertices, output interpolants.

Tessellation Stage Actually three stages, subdividing primitives according [8].

Geometry Stage Transform a primitive into a strip of primitives [9].

Fragment Stage Colour pixels.

These stages are designed heavily around the concept of stream processing, processing items from various streams and pushing them into other streams. As such, there is little to no communication available between elements. This makes procedural geometry difficult to represent efficiently. The tessellation stages and geometry stage aim to support primitive geometry, but both face major limitations in usability. The tessellation stage is suited for slight increases in resolution, and is mostly used to increase triangle density in specific certain areas. The geometry stage is more suitable as it creates new primitives, but is still an uncomfortable abstraction, and faces poor performance.

A common technique for procedural geometry is to abandon this pipeline and use a *compute shader*. This general purpose shader can fill a temporary *staging* index buffer with a triangle list. This requires $O(n)$ additional memory to store the staging buffer, for n possible triangles generated, and the triangle-list format requires a *contiguous* block of triangles. We then treat the mesh as any other static mesh, passing it through the vertex and index stages.

2.3.2. Mesh Shading

Mesh shading attempts to solve shortcomings of using the primitive graphics pipeline for procedural geometry. They were introduced to modern GPU APIs in 2021 and appear as a Vulkan extension. As such, they are not universally supported. This is a new technology with many terms; we will use the Vulkan implementation and terminology.

We have seen that procedural geometry methods have widely adopted compute shaders due to their flexibility and good support. Mesh shading brings this flexibility to the graphics pipeline by stripping out everything other than the primitive pipeline’s fragment stage and creating task and mesh shaders. A mesh shader is more similar to a compute shader than a vertex shader and outputs a small collection of primitives, a *meshlet*, to be rasterised [10].

The full pipeline is simpler than primitive shading:

Task Stage⁷ A compute shader that may emit **mesh shaders**.

Mesh Stage A compute shader that may emit primitives.

Fragment Stage Identical and interoperable to the primitive fragment stage.

Task shaders communicate to their emitted mesh shaders by sharing some *payload* data with them. While not as powerful as the ability to generically launch threads on the GPU⁸, this gives us great flexibility for procedural geometry without staging buffers. As such, mesh shading is a critical tool for rendering our cluster-based view-dependent LODs.

The Nvidia recommended maximum size for a meshlet is 126 indices⁹ and 64 vertices. This memory is allocated *per workgroup*, and mesh shaders are not required to fill it all. In Appendix A I show a minimal example of task and mesh shaders using all the features listed above.

2.4. Existing Techniques

Many forms of multiresolution exist with different characteristics and drawbacks. Progressive meshes, introduced by H. Hoppe [12], are multiresolutions encoded as a low-resolution base mesh and the vertex splits required to raise resolution. Quick-VDR [13] expanded on progressive meshes with an initial coarse-grained selection (similar to our clustering) before vertex-local transformations.

Further techniques, such as BDAM [3], or Adaptive Tetrapuzzles [14], focus more on the coarse-grained selection, using spatially based partitions for 2D and 3D surfaces, respectively. Their partitions contain geometry in clusters of neighbouring triangles that can be substituted. This was generalised to a DAG of clusters by “Batched multi-triangulation”, BMT [2], used in my work. Ponchio’s thesis contains an excellent comparison of the above methods [15]. However, cluster selection as outlined in BMT operates *out of core*, as, at the time of its publishing, GPGPU compute was not feasible and offloading to other CPU cores was the premier way to extract more performance.

In industry, Unreal Engine 5’s Nanite [1] is a complete implementation of a technique based around BMT. It attempts to avoid *pop-in*, visible transitions in resolution, by selecting clusters of a high enough resolution that errors are sub-pixel, and uses GPU-driven rendering for high throughput; the same approach I take in my implementation.

⁷Known as the Amplification stage in DX12.

⁸See Work Graphs [11], a powerful extension that allows generic thread invocation.

⁹Not a typo – we allocate some space to store the meshlet’s size.

2.5. Project Development

2.5.1. Requirement Analysis

Requirements for the core of this project are included as the success criteria listed in Appendix E: improving frame time and rasterisation compare to full-resolution rendering. This required developing the following deliverables, mirroring the goals set out in § 1.2:

- A **multiresolution generator**, using what we have seen in both § 2.1 and § 2.2.
- **Procedural geometry renderers**, rendering clusters with the methods in § 2.3.
- **Cluster selection algorithms** making valid (§ 2.1.2) and good (§ 2.1.3) cluster selections.
- A **benchmarking framework** to analyse the performance of this complex system.

Once these were delivered I introduced extensions and research to iteratively expanding them.

2.5.2. Libraries and Tools

Graphics I used the Vulkan graphics API, as the requirements of this project lean heavily towards performance – Vulkan is a cutting edge API enabling me to exploit the full potential in GPUs. Vulkan is highly portable with good documentation.

Language As I use Vulkan, GLSL is the natural choice for shader code. I use Rust 1.78.0 as the CPU language, for its speed, safety, and mature package manager Cargo.

Profiling Within Rust, I used `cargo flamegraph` for profiling, giving function level breakdowns of bottlenecks. For GPU profiling, I used NVIDIA Nsight™ [16].

Libraries The two most important libraries I use are METIS 5.1 [17], for generating K-partitions of graphs, and `meshopt`, for various mesh processing tasks outside the scope of this dissertation and as a comparison mesh simplifier in our evaluation (implemented as an extension). I use Vulkan via the Rust `ash` [18] library which exposes the calls to unsafe Rust, as well as providing similar abstractions to Vulkan-Hpp [19].

Testing The Rust build and package system, `Cargo`, contains first class support for unit tests. I use `cargo llvm-cov` for code coverage metrics, guiding new tests.

Version Control Version control was performed with Git. For backup, I maintained regular incremental backups on Github, and continuous backups using `syncthing` to a NAS device and laptop.

2.5.3. Starting Point

I had no prior knowledge of mesh simplification or LOD generation. I have also never used Vulkan, although I have developed several 3D engines in OpenGL. I was made aware of the multiresolution technique from the Unreal Engine SIGGRAPH presentation [1], which I watched in summer 2023. No code from the Unreal Engine was read for the development of this project. I have experience in Rust, GLSL, and 3D applications from personal projects.

2.5.4. Development and Engineering

This project was divided into two main modules: the multiresolution generator and renderer. I followed the incremental build model, with core features implemented week by week according to the deadlines set out in my proposal. This allowed rapid prototyping and good flexibility. I developed the generator with a test-driven mindset, identifying rare edge cases in the mesh simplifier to ensure it remained robust with massive inputs. Development of the renderer required a GPU with support for modern API features, so I used a personal PC (r5 3600, GTX 1660).

2.5.5. Progress

I followed the plan in my proposal, allowing me to hit deadlines with plenty of slack. I had a working prototype within Michaelmas, allowing me to include extensions in iterative work. Through iteration, one of my extensions, mesh shading, became a core feature of my project. This caused a large delay, as it required transitioning the renderer backend to Vulkan. I had previously planned to use `wgpu` – a higher-level library, with web support as an extension. I handled this by reusing as much logic from the previous renderer as I could.

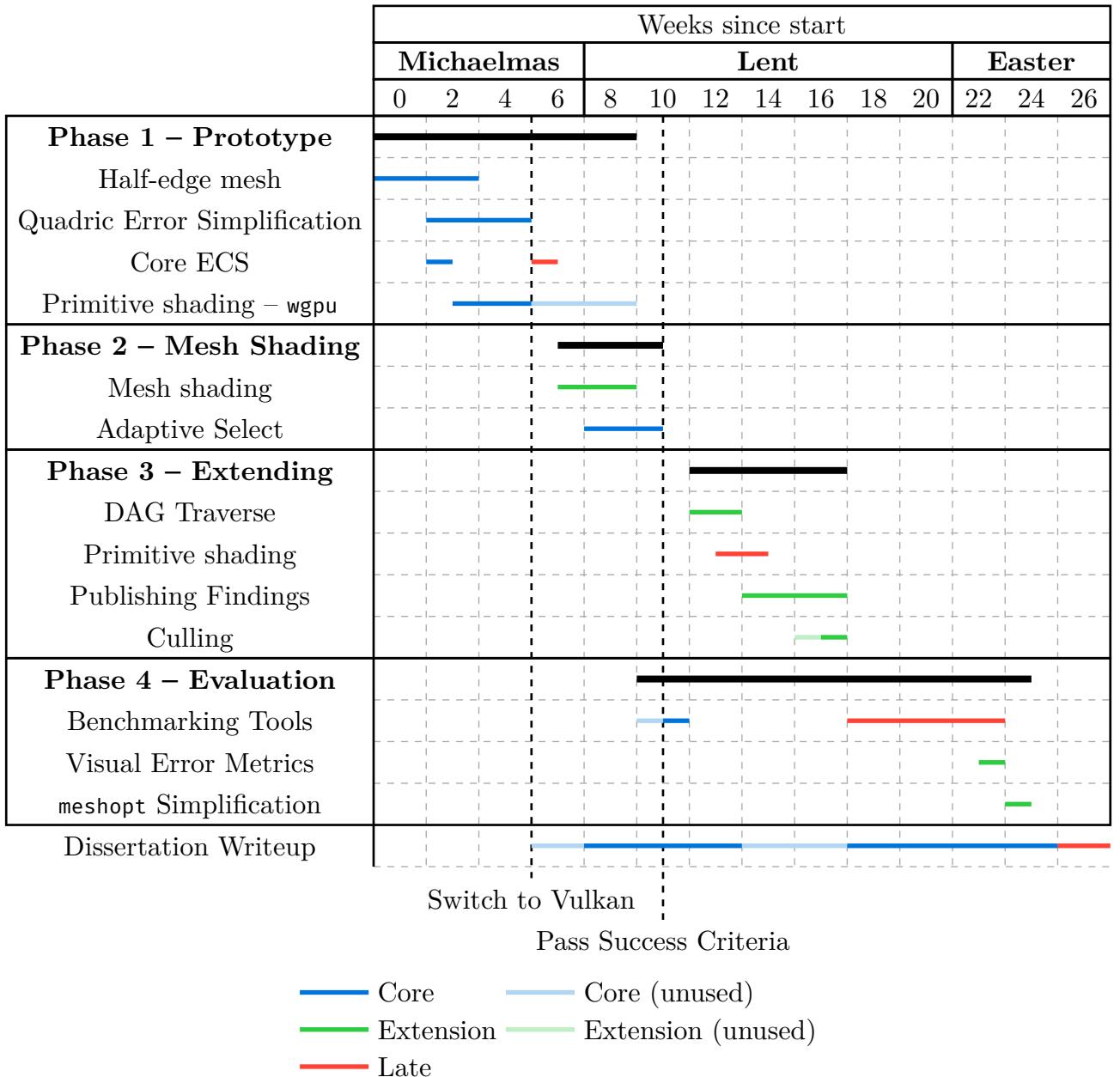


Figure 11: Gantt chart of my progress throughout 2023–2024

In Lent, I paused work on the dissertation to release a paper for my work, which was awarded 3rd best paper and 2nd best presentation at the CESCG student research seminar. This gave me an opportunity to learn and improve the quality of my writing.

Chapter 3

Implementation

I begin this chapter by describing the multiresolution generator (§ 3.1). With this foundation, I implement the renderer in stages, starting with architecture (§ 3.2), then selecting clusters (§ 3.3, § 3.4), and finally rendering as procedural geometry (§ 3.5).

3.1. Multiresolution Generation

Though this project mainly investigates rendering multiresolutions, I had to start by implementing a generator. To do this, I implement a half-edge mesh as described in (§ 2.2), followed by outlining my target, the generation algorithm (§ 3.1.1). I implement this in two parts: mesh simplification (§ 3.1.2) and efficient partitioning (§ 3.1.4).

```
1 #[derive(Debug, Clone, PartialEq)]
2 pub struct HalfEdgeMesh {
3     faces: Pidge<FaceID, Face>,
4     edges: Pidge<EdgeID, HalfEdge>,
5     verts: Pidge<VertID, Vertex>,
6     clusters: Vec<ClusterInfo>,
7     groups: Vec<GroupInfo>,
8 }
```

Rust

Listing 1: The half-edge mesh datatype

I defined a slab-allocated `Pidge` data structure with a unique index type for each primitive type; strongly-typed indices made code more readable and less error-prone. When deleting elements, I leave an empty slot at the deleted index using Rust’s algebraic data types. Deletion then occurs in $O(1)$ time, at the cost of the memory usage of the generator never decreasing from its peak. Empty slots allowed me to track and throw errors when a deleted element was referenced, a strong sign of a bug.

As real world meshes are not clean, it is possible for > 2 triangles to meet at the same edge. This is still represented by the `twin` field. I implemented this as a linked list around all triangles that share the edge.

```
1 impl HalfEdge {
2     pub fn dst(&self, mesh: &HalfEdgeMesh) → Result<VertID> {
3         Ok(mesh.try_get_edge(self.next_edge)
4             .context(MeshError::InvalidCwEdge(self.next_edge))?
5             .vert_origin)
6     }
7 }
```

Rust

Listing 2: Method to extract the destination vertex of a half-edge.

As this is a complex system, error management is crucial. As such, I annotate all operations with `context`, such as in Listing 2, allowing me to trace the cause of crashes.

3.1.1. Generation Algorithm

With a half-edge mesh in place, I created the function outline that would eventually generate multiresolutions. The generation scheme I use is described in § 2.1.1, the four-children-two-parent scheme. Throughout this implementation, I make heavy use of the open-source graph partitioner, METIS [17]. This library quickly generates K -way partitions of graphs minimising edges between partitions (PARTKWAY).

Algorithm 1: GENERATE, multiresolution generation algorithm

```

1 procedure GENERATE (triangles)
2   let clusters := PARTKWAY(triangles,  $K = \lceil |\text{triangles}| / 280 \rceil$ )            $\triangleright$  Stage 1
3   while clusters > 1
4     yield clusters
5     let groups := PARTKWAY(clusters,  $K = \lceil |\text{clusters}| / 4 \rceil$ )                  $\triangleright$  Stage 2
6     clusters := []
7     for group in groups
8       Simplify(group)                                                  $\triangleright$  Stage 3
9       let new-clusters := PARTKWAY(group,  $K = 2$ )                          $\triangleright$  Stage 4
10      for cluster in group (cluster.parents := new-clusters) end
11      clusters.push(new-clusters)
12    end
13  end
14 end

```

This algorithm contains and relies on many fine details, outlined in the following sections.

3.1.2. Mesh Simplification

This section focuses on the difficulties of applying a simplification algorithm based on quadric error metrics (§ 2.2.3) to Algorithm 1. This is composed of two parts: the individual edge collapses, and the guiding heuristic. I start by implementing edge collapse, Algorithm 2.

Algorithm 2: Edge-collapse on a half-edge mesh (§ 2.2.2).

```

1 procedure EDGECOLLAPSE ( $(v_i \rightarrow v_j)$ )
2   TRI COLLAPSE( $v_i \rightarrow v_j$ )
3   if  $v_j \rightarrow v_i$  exists                                 $\triangleright O(1)$  time complexity; this is this edge's twin
4     | TRI COLLAPSE( $v_j \rightarrow v_i$ )
5     for  $e_2$  in edges( $e \rightarrow \text{orig}$ ):
6       |  $(e_2 \rightarrow \text{source}) := (e \rightarrow \text{dest})$ 
7   end
8   procedure TRI COLLAPSE (e)                                $\triangleright$  Collapse a degenerate triangle
9      $e \rightarrow \text{ccw} \rightarrow \text{twin} := e \rightarrow \text{cw} \rightarrow \text{twin}$ 
10     $e \rightarrow \text{cw} \rightarrow \text{twin} := e \rightarrow \text{ccw} \rightarrow \text{twin}$ 
11   end

```

To execute an edge collapse on $v_i \rightarrow v_j$, we need make only local changes to our mesh. All the redundant data encoded in the half-edge mesh (§ 2.2.1) we have seen so far is used here, ensuring the algorithm is time efficient. I replace all references to the vertex v_i with references

to v_j , and remove the newly degenerate triangles. As the generation algorithm collapses edges possibly millions of times, it is crucial this primitive operation executes with no edge cases or untracked side effects. As such, I designed unit tests with pre- and post-conditions assuring mesh validity. Algorithm 2 only references values immediately retrievable from its half-edge, so takes $O(e)$ time, for e outgoing half-edges connected to the vertex. In real meshes, this has an average of six iterations. As my edge collapse moves one vertex to another, we do not introduce extra vertices to our mesh.

Following this, I implemented SIMPLIFY, Algorithm 3, for mesh simplification. In doing so, I noticed that simplification on planes quickly became messy. This was due to edge collapse occurring at random (up to floating point imprecision), as planes have 0 quadric error. For clustering, it is preferable that our mesh is uniformly tessellated. An additional per-vertex error, the square distance from the original point multiplied by some low constant, solved this. For visual testing I applied this algorithm to generate LOD chains, as shown in Figure 12.

Algorithm 3: SIMPLIFY based on quadric error metrics [20]

```

1 procedure SIMPLIFY
2   Compute the  $Q$  matrices for all the initial vertices.
3    $H :=$  All half-edges.
4   for each half-edge  $(v_i \rightarrow v_j)$  in  $H$ 
5      $\text{Err}(v_i \rightarrow v_j) := v_i^T(Q_i + Q_j)v_j$ 
6   end
7   Place all the half-edges in  $H$  in a min-heap keyed on Err.
8   while can get  $(v_i \rightarrow v_j)$  from min-heap
9     if not valid continue                                 $\triangleright$  Loop until we find a valid edge (§ 2.2.2)
10    Edge-Collapse( $v_i \rightarrow v_j$ )
11     $Q_i := Q_i + Q_j$ 
12    for  $v_p \rightarrow v_q$  in neighbours( $v_i \rightarrow v_j$ ):
13       $\text{Err}(v_p \rightarrow v_q) := v_p^T(Q_p + Q_q)v_q$ 
14    end
15  end
16 end

```

3.1.3. Simplification within Groups

The next step was extending SIMPLIFY to support the constraints of the multiresolution generation algorithm by applying it to each group separately. This creates many small heaps, which in turn improves the time complexity of simplifying the mesh as a whole. However, this optimisation is not free, as we are likely to simplify areas of the mesh at different rates, depending on the average error of each group. This must be considered when selecting which clusters to render, as depth in the DAG does not represent error nor detail level (although it is of course correlated), unlike traditional LOD chains. This is accounted for in my per-cluster error value.

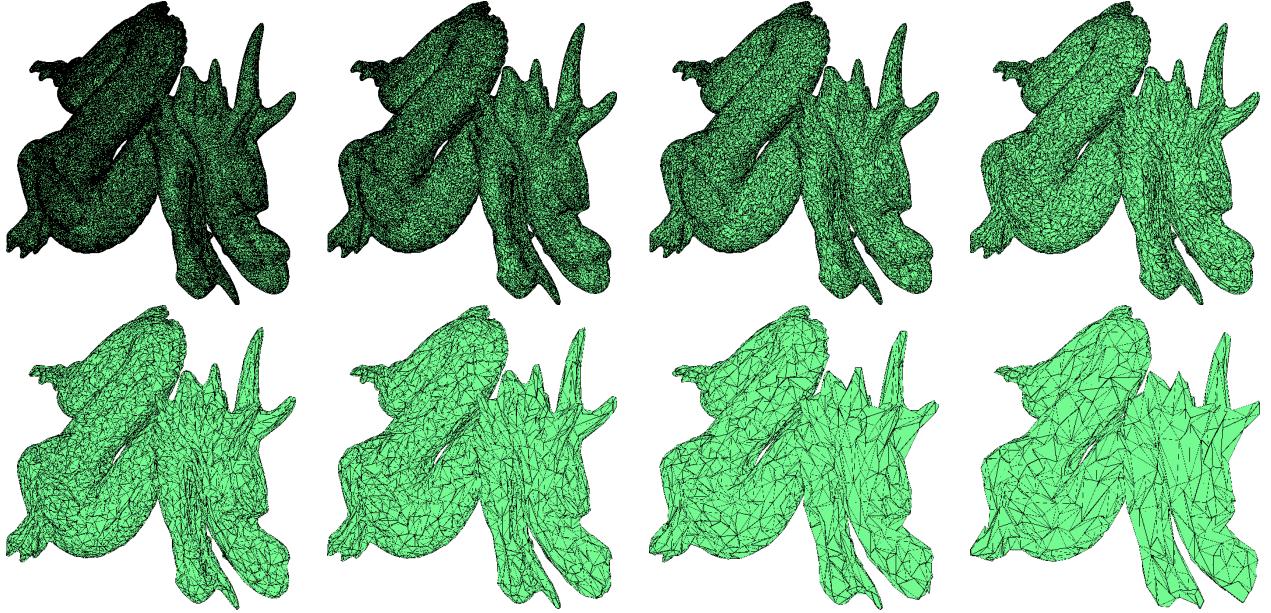


Figure 12: LOD Chain. Simplification of 1M triangle Dragon model [21], halving triangle count per step. Quadric error metrics ensure small features such as the horns are preserved.

Because GENERATE alternates between grouping and simplifying, SIMPLIFY must be interruptible. After simplification in the generation algorithm, groups are re-generated and the heaps become useless (as these are generated per-group), so the only state that needs to be carried through is the per-vertex error matrix Q .

To implement the simplification required by the generation algorithm, our valid edge check must include collapses that would introduce *seams* between neighbours in the DAG. I invalidate any operations that affect locked edges (edges on the boundaries of groups). Additionally, I invalidate *pinching*, collapsing a triangle strip to meet at a single vertex (Figure 13), as METIS requires that our triangle graphs are contiguous [17]. A boundary may be a mesh boundary or a group boundary, as follows:

Mesh boundary A vertex does not have a triangle fan surrounding it.

Group boundary Some triangles connected to a vertex are members of different groups.

With these definitions, I mark an edge $v_i \rightarrow v_j$ as invalid if:

$$\text{group-boundary}(v_i) \vee \\ (v_j \rightarrow v_i \text{ exists} \wedge \text{mesh-boundary}(v_i) \wedge (\text{mesh-boundary}(v_j) \vee \text{group-boundary}(v_j))) \quad (6)$$

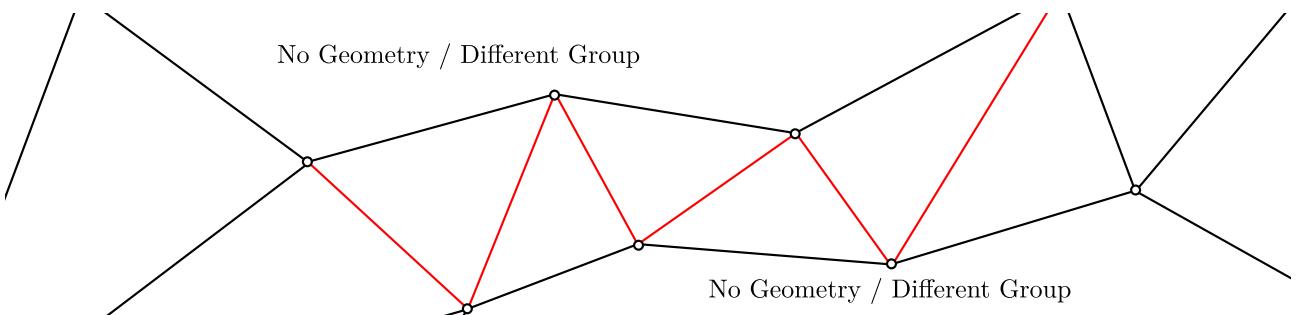


Figure 13: Pinching edge collapses, invalidated due to METIS’s contiguous graph requirement.

In §2.2.2, we looked at other forms of invalid edge collapse. In SIMPLIFY, to check all of these without comprising performance we only check if an edge is valid *after* an edge has been popped from the heap, rather than checking all edges up-front.

3.1.4. Grouping Clusters

At this point, I have connected **SIMPLIFY** to **GENERATE**. As shown in §3.1, simplification can only occur within the boundaries of a group, but we have yet to define what clusters may be grouped. Without any restrictions on this, METIS will naturally create similar groupings at each level, with no regard to how long ago two areas were last grouped. In early iterations, this caused a build-up of old edges around certain parts of the mesh, as seen in Figure 15.

To measure the success of our strategies to combat this, we need a baseline. I introduce the concept of *edge age*: the number of simplification passes an edge has survived. Our scheme has no maximum edge age, causing a possible build-up of high resolution edges, in the same style of a tree-based multiresolution, which was deemed unsuitable for that very reason; clearly I wish to measure and minimise edge age.

To resolve the issue of long-living edges, we wish to prioritise grouping clusters that have not been recently grouped. The key to this is the number of shared edges between clusters. METIS will, by design, optimise partitions by minimising the number of shared edges between them. Figure 14 shows the effect of adding additional ‘guide edges’ to a mesh – we can suggest to METIS that some nodes should be partitioned together.

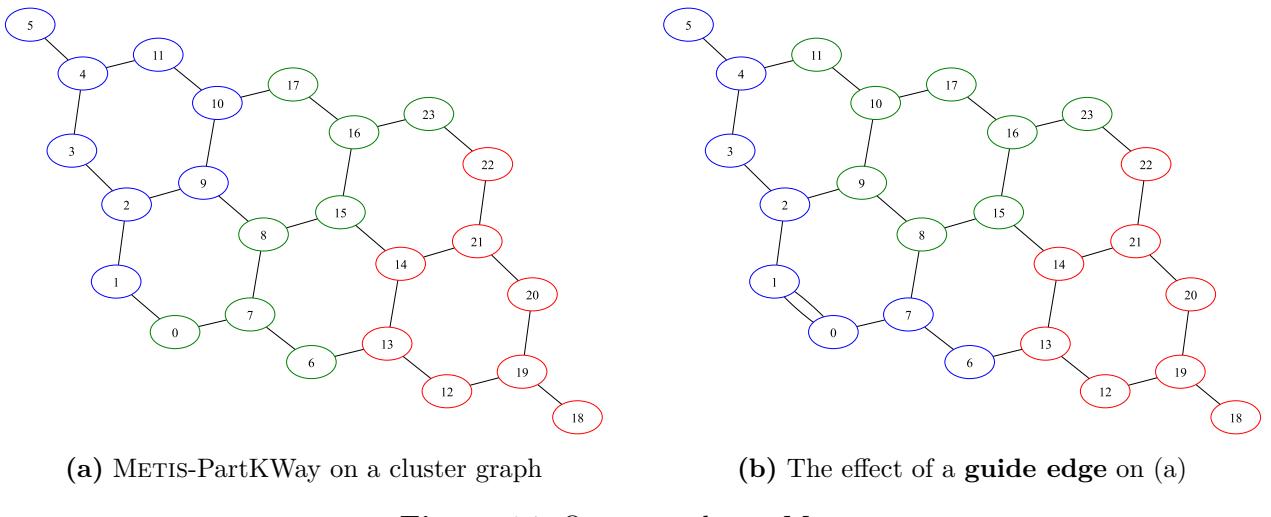
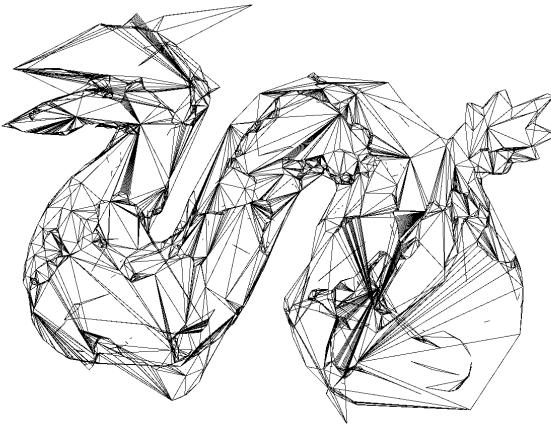


Figure 14. Our control over METIS

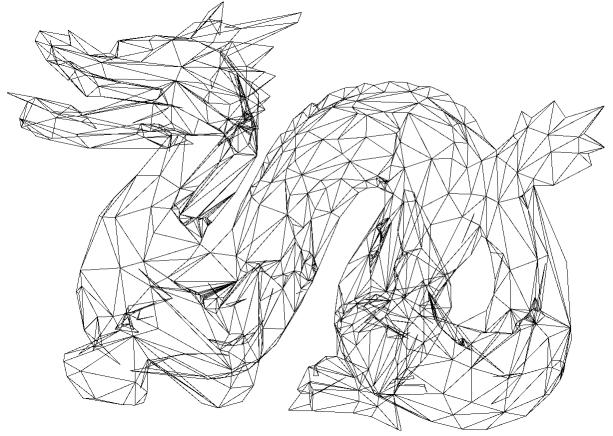
This is desirable for generating the starting set of clusters, as it results in small bounding spheres. This also means we can assume the numbers of shared edges between starting clusters are relatively uniform. However, after applying SIMPLIFY to groups, it is clear that clusters within the same group will have fewer shared edges – their shared edges have been simplified. This observation allowed me to improve groupings by applying guide edges proportionally to the sizes of boundaries between clusters – long boundaries suggest previously locked edges which should be grouped over and simplified away.

However, in applying this strategy I found that METIS failed to create groups of uniform size, instead creating groups of widely different clusters and triangle counts (but uniform guide edges between clusters, its optimisation goal). That would cost GPU performance. To counter this, I designed Algorithm 4, grouping the graph in stages, starting with a small

number of partitions and recursively subpartitioning, altering targets depending on if METIS makes a partition too large or too small.



(a) METIS-PartKWay: Build-up of old age edges after several levels of simplification.



(b) EXACTPARTITION: More uniform tessellation.

Figure 15: Effects of grouping on final result. Average edge age reduced from 2.88 to 1.05

Algorithm 4: EXACTPARTITION by recursive METIS partitionings

```

1 procedure EXACTPARTITION (graph, partition-size)
2   if graph→node-count ≤ partition-size + 1           ▷ Already small enough
3   | return graph
4   else:
5     if large(graph→node-count)                      ▷ Many nodes, recurse into few
6     | let partitions := SPLIT-FACTOR
7     else                                         ▷ Few nodes, attempt to make final partitions
8     | let partitions := graph → node-count / partition-size
9     end
10    let parts := METIS-PartKWay(graph, partitions)
11    for part in parts                         ▷ Do recursion, split graphs up if they are too large
12    | part := EXACTPARTITIONONTOGRAPH (part, partition-size)
13    end
14    return ApplyParts(graph, parts)
15  end
16 end

```

Injecting my own control between METIS operations results in a large improvement to group sizes, maintaining high quality meshes (Figure 15). It also introduced recursion into GENERATE, which caused a stack overflow for some of the larger (> 10 million triangle) meshes. I introduced a depth counter, and split recursive calls into different threads when the depth became too high. The final result is more uniform group sizes (Figure 16). Having a known maximum group size is ideal for GPU code; it allows the compiler to unroll loops over constant values.

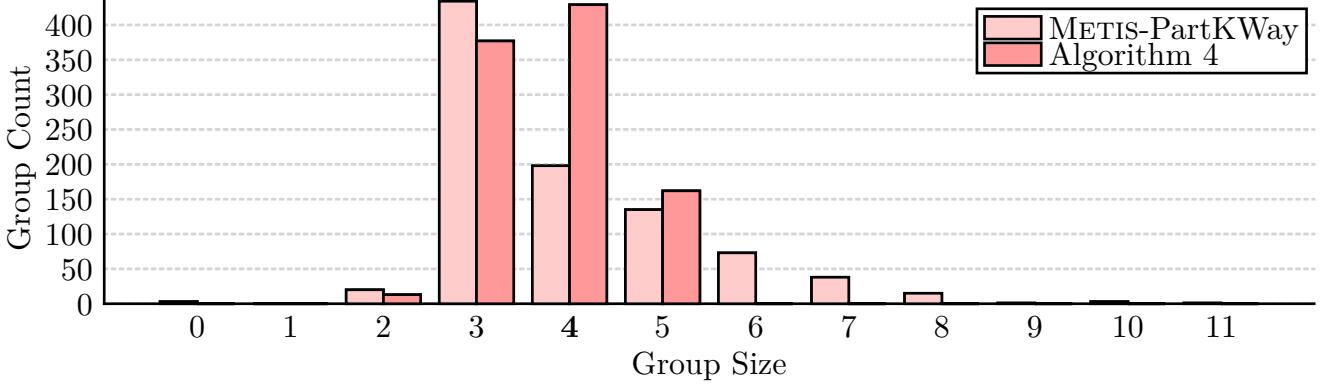


Figure 16: Group sizes on the dragon mesh. The desirable number of clusters in a group is 4.

3.2. Renderer Architecture

With a working multiresolution generator, I turned to the renderer. I began with an implementation using the WebGPU specification, though I restarted with Vulkan after reading about mesh shading (§ 2.3.2), which I use extensively.

The renderer’s complexity is managed in two parts. First, by creating a small Vulkan wrapper to manage some of its verbosity, and second by using an Entity Component System (ECS) for state management. An ECS is a system that decouples data from logic for real-time systems. We store state in *components*, structs that are attached to *entities*. Logic is handled by *systems* which query the ECS for collections of entities with desired components. It encourages object *composition* in the form of combining components rather than object-oriented inheritance and encapsulation. This reduces difficult-to-track complexities arising from polymorphism [22].

Vulkan is a low level C library and its Rust wrapper, `ash`, only adds small helper functions, not memory safety guarantees. This means I must manage Vulkan’s *handles*, opaque pointers for resources, myself. Retaining handles to unused resources is a memory leak and will nullify driver optimisations for resource reuse. Additionally, Vulkan objects are hierarchical, and may depend on some other object (for example, an item may depend on a pool¹⁰) To ensure memory safety, I wrap handles in a struct with a destructor to free the object, and a reference counted smart pointer to their parent, ensuring children outlive their parents (Listing 3).

```

1 pub struct DescriptorPool {
2     device: Arc<Device>,
3     handle: vk::DescriptorPool,
4 }
5 impl Drop for DescriptorPool {
6     fn drop(&mut self) {
7         unsafe { self.device.handle.destroy_descriptor_pool(self.handle, None); }
8     }
9 }
```

Rust

Listing 3: A `DescriptorPool` manages *descriptor set references*; groups of buffer allocations for shaders. It is owned by the `Device` so we ensure the device outlives it, and define a destructor.

¹⁰which depends on a logical device, followed by a physical device, followed by an instance.

To save boilerplate, this is implemented with a Rust macro. These are different from C preprocessor macros in that they execute with access to the abstract syntax tree of our program allowing syntax-safe operations.

3.3. Cluster Selection

In this section, I will detail the algorithms designed for selecting clusters to draw. The first is designed similarly to Unreal Engine 5’s Nanite, while the second is the product of my research.

At this point in the implementation, we have a DAG of a multiresolution that:

- Is well-conditioned.
- Contains a monotonic object-space error value per cluster (average edge length).
- Contains a bounding sphere per cluster, forming a nested bounding volumes hierarchy.

Both of these selection algorithms run on the GPU, exploiting parallelism. Having our selected clusters on the GPU ready for drawing removes some transfer and latency bottlenecks; we expect to draw several thousand clusters, and a unique set every frame, which would require large bus transfers to send from the CPU to the GPU.

As seen in § 2.1.3, these inputs applied to the cluster error function create a single unique cut on our DAG; we can determine if a cluster should be drawn given itself and its parents. This task is highly parallel, so we expect high performance on a GPU.

Each implementation of cluster selection supplied with:

- A cluster buffer, containing the DAG of cluster structures in the form of Figure 17. Clusters reference their children in the level below, and their *spouse*, the unique other cluster in the DAG with identical children.
- Instance information, the model matrix of the instance we are drawing.
- Camera information, the view-projection matrix of our camera.
- A screen-space error target, τ , the maximum screen-space error of a selected cluster.

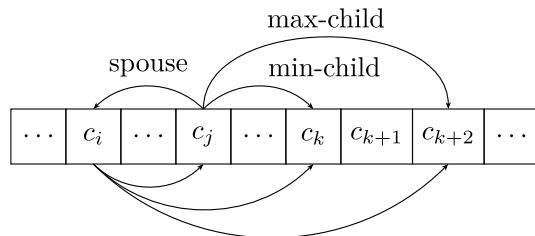


Figure 17: A diagram of a cluster structure suitable for GPU operations. Pointers within the diagram from cluster c_j correspond to a cluster $c_j = [\text{spouse} = i, \text{min-child} = k, \text{max-child} = k + 2]$.

These render pipelines will focus on drawing *instances*, ignoring the type of mesh this instance has. This is without loss of generality, as we could combine all vertex buffers and DAGs to a single *mesh atlas*, placing all the data into single buffer – this is not implemented, as it is aside from the core problem.

To start with, we need a simple way to test if a cluster has an error acceptable to be drawn. I developed Algorithm 5 to determine if a cluster should be selected locally, based on the relation of its parent’s errors to its own.

Algorithm 5: LOCALCUT

- 1 parent-err := $\min(\text{err}(c_i \rightarrow \text{parent}_0), \text{err}(c_i \rightarrow \text{parent}_1))$
- 2 this-err := $\min(\text{err}(i), \text{err}(c_i \rightarrow \text{spouse}))$
- 3 draw := $\text{this-err} \leq \tau \wedge \text{parent-err} > \tau$

Recall the conditions for a valid cut from § 2.1.3: if a cluster is included in the selection, so must all their siblings. As such, each cluster’s invocation must make the decision of what fills its group’s area; the group, or the two parents (if either). This should be agreed by each cluster in the area implicitly, with no communication. This is done by assigning each cluster the error and bounding volume of their group as a whole, similarly to the bounding volume hierarchy of BDAM [3]. If each cluster in a group shares the same input data, it will make consistent decisions with respect to its neighbours in levels above and below [1].

To decide if a cluster c_i should be drawn, we compare the error of c_i to the minimum error of the two parents $c_i \rightarrow \text{parent}_0$ and $c_i \rightarrow \text{parent}_1$. If the parents error is below τ , c_i should not be drawn. We take the minimum parent error, as the two parents are likely members of different groups. However, this leaves a hole if:

$$\text{err}(c_i \rightarrow \text{parent}_0) > \tau > \text{err}(c_i \rightarrow \text{parent}_1), \quad (7)$$

as only one of two parents could be drawn. To resolve this, Algorithm 5 takes ‘this-error’ to be the minimum error between c_i and its spouse. This fills the hole described above, as the previously missing parent will now draw based on the lower error of its spouse.

Finally, some nodes in the DAG have no children or no parents, being the leaves and the roots. In these cases, we assume the roots’ parent’s errors are ∞ , and the leaves’ children’s errors are $-\infty$. This ensures that, for any finite τ , Algorithm 5 will select a complete cut.

3.3.1. DAG Traversal Cluster Selection

Our goal is to select clusters from the leaf nodes of a dicut of the DAG. We can traverse the DAG recursively starting from the root. This method is similar to Nanite’s *persistent threads* selection, although I avoid the use of yet-undefined forward progression scheduling guarantees on the GPU by removing requirements for workgroup communication.

Traversing the DAG requires care. It is not a tree, so there are clusters that share children. To traverse efficiently, we should not traverse the same cluster twice. Our DAG is shaped similarly to a tree, as it is formed of pairs of clusters, spouses, that share identical children. As their children are shared, we can view the spanning tree of the DAG by only regarding the children of one spouse, illustrated in Figure 18.

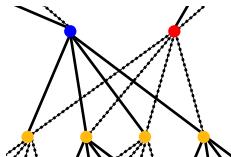


Figure 18: A snapshot of the DAG. A group of clusters whose parents are in separate groups. Solid lines represent the edges in spanning tree used to traverse the DAG as a tree.

I allocate the queue and draw buffers in high speed workgroup shared memory. This shared memory is not large; my GPU has 49KB per workgroup. We need to allocate enough space

in the queue to fit the maximum clusters selected, which is 50% of the clusters in the DAG (LOD0). To make best use of shared memory, I use 16 bit integers, space for a maximum of 24000 clusters in a multiresolution and so 3M triangles in LOD0. This is sufficient for most meshes, but would have to be extended to slower GPU memory for my largest meshes.

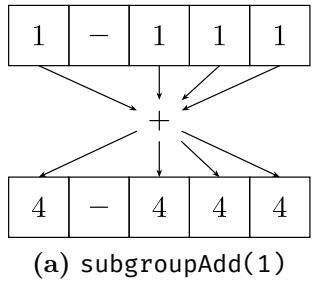
Algorithm 6: DAG-TRAVERSE cluster selection

```

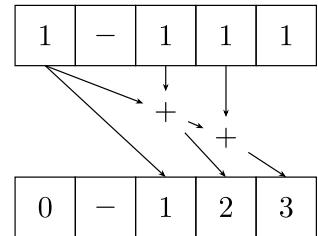
1 while queue not empty
2    $i := \text{queue}[\text{head}]$ 
3   head := head + 1
4    $\text{cluster}_i := \text{clusters}[i]$ 
5   if LOCALCUT( $i$ )                                 $\triangleright$  Cluster should be drawn
6     draw-buffer[draw-count] := cluster
7     draw-count := draw-count + 1
8   else if  $i < \text{cluster}_i \rightarrow \text{spouse}$      $\triangleright$  Only traverse 1 of the 2 parents to traverse as a tree
9     for  $c$  in  $\text{cluster}_i \rightarrow \text{children}$ 
10    queue[tail] :=  $c$ 
11    tail := tail + 1
12  end
13 end
14 end

```

Multiqueue. Algorithm 6 does not appear highly parallel, as it leverages a single shared queue. We must allow multiple invocations synchronised access to the queue to maintain parallelism. Atomic operations are too slow for this use case. The key to making this work is **subgroup arithmetic**¹¹, introduced in Vulkan 1.1 core, to synchronise queue access. A subgroup is a set of invocations operating in lock-step with each other. They will always be part of the same workgroup, but a workgroup may maintain multiple subgroups. In this algorithm, I limit the size of my workgroups to ensure they only contain one subgroup, which is generally 32 or 64 invocations. This limits the selection latency of a scene to its most complex instance.



(a) $\text{subgroupAdd}(1)$



(b) $\text{subgroupExclusiveAdd}(1)$

Figure 19: Subgroup operations on a subgroup of 4 active invocations and one inactive invocation.
Inactive invocations contribute the value 0.

Subgroup operations allow invocations to share data with reductive operations, and operate in a single function call supported by the compiler. As invocations operate in lock step, they can be made **inactive** by conditional branches and will idle while waiting for the branch to end. These are ignored by subgroup operations. An example subgroup operation is `subgroupAdd(1)`,

¹¹Subgroups are the Vulkan term, on AMD/HLSL these are waves, in NV/CUDA they are warps

which will return the number of active invocations [23] – each active invocation will contribute 1 to the total sum, shown as a invocation diagram in Figure 19. These are implemented by communication through shared registers, so are incredibly efficient [24].

This is used in Algorithm 6, where we require each invocation to pop a cluster index from the queue. When all invocations are active, we can offset the head pointer by our current subgroup index, but this leaves us with conflicting information about the queue’s true head across invocations. To solve this, I offset by invocation index while reading from the queue, but add `subgroupAdd(1)` to the pointer, ensuring the data is synchronised.

```
1 int idx = gl_LocalInvocationID.x;
2 int cluster = queue[head + idx];
3 head += subgroupAdd(1);
```

GLSL

Listing 4: Multiqueue pop routine

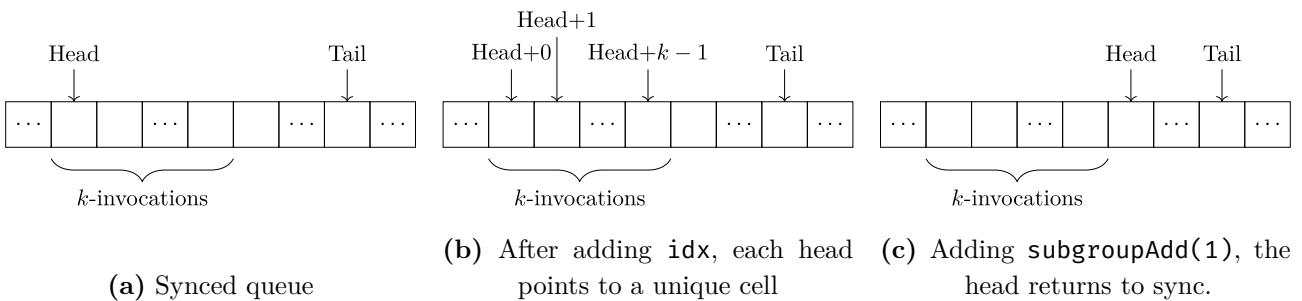


Figure 20: Pointers of the queue throughout each invocation throughout popping

Figure 20 shows this operation visually. The end result is a queue synced between invocations, with each invocation having read a different value.

The more complex operation is pushing children to the queue (lines 9-12). The number of children each cluster has is variable, resulting in some invocations adding more children than others, so we cannot offset by our subgroup index. To solve this, we use a more advanced subgroup command, `subgroupExclusiveAdd`, which performs an exclusive addition across all active invocations, also shown in Figure 19. Because we know the number of items each invocation is going to add to the queue, we can allocate offsets upfront, re-syncing the queue by taking `subgroupMax(tail)`.

```
1 int children = clusters[i].max_child_index - min_child_index + 1;
2 tail += subgroupExclusiveAdd(children);
3 for (int c = min_child_index; c <= clusters[i].max_child_index; c++) {
4     queue[tail] = c;
5     tail += 1;
6 }
7 tail = subgroupMax(tail);
```

GLSL

Listing 5: Multiqueue push routine.

These two operations make up my multiqueue. Subgroups allow for more complex operations such as swizzling and balloting, but we only require the arithmetic package.

3.3.2. Adaptive Cluster Selection

As a point of research in this project, I wanted to develop an efficient algorithm for the selection phase that would, coupled with mesh shading in later chapters, eliminate any intermediate working space. This can be achieved with a temporal optimisation, reusing a small amount of data from the previous frame.

Thinking back to LOCALCUT in § 3.3, a naïve selection algorithm would check every cluster, every frame. This saves the need for the queue while traversing the DAG. However, on average, most clusters are too high a resolution to select, making that impractical for actual use; it would take time proportional to $O(t)$ for t total triangles in the scene, far worse than LOD chains $O(i)$ for i instances.

The approach I developed to optimising this is to limit the extent of the cluster buffer we check each frame, using the fact that we can only select a cluster if a shader has been invoked for it. I control my dispatch count inside the shader by providing an indirect dispatch buffer. Additionally, the lower the object-space error of a cluster, the less likely it is to be drawn. Thus, if I order all clusters in the buffer by their object-space errors, I can cap the maximum resolution searched dynamically depending on the current view of the instance. Dispatching invocations for clusters with indices above the maximum that is selected is futile, so this is the value we wish to estimate for the *next frame*'s dispatch count.

A compute indirect dispatch command contains three fields: group counts x , y , and z . This command dispatches invocations with indices in a 3D grid, with dimensions $x \times y \times z$. As we operate on a linear buffer of clusters, we will fix group counts y and z to 1, and vary x . We must also be careful to never set group count x to 0, as this would stop the shader ever invoking again. I say the *maximum requested index* for a cluster, based on the errors calculated in LOCALCUT, is:

$$\text{max-requested-idx}(i) = \begin{cases} \text{max-parent-idx}(c_i) & \text{if parent-error} < \tau \\ \text{max-child-idx}(c_i) & \text{if this-error} > \tau \\ i & \text{else} \end{cases} \quad (8)$$

Algorithm 7: ADAPTIVESELECT cluster selection.

```

1 for each instance:
2   Lower group-count- $x$                                  $\triangleright$  Allow maximum to decrease between frames
3   parallel for each cluster  $c_i$ :
4     let draw := LOCALCUT( $i$ )
5     let max-idx $_i$  := max-requested-idx( $i$ )
6     if draw and  $c_i \rightarrow \text{spouse} < \text{group-count-}x$        $\triangleright$  Draw if our spouse is loaded
7       or max-idx $_i > \text{group-count-}x$                    $\triangleright$  Children are not loaded, must draw self
8       | draw-list := cluster::draw-list
9     end
10    group-count- $x$  := max(max-idx $_i$ , group-count- $x$ , 1)       $\triangleright$  Set dispatch count
11  end
12 end

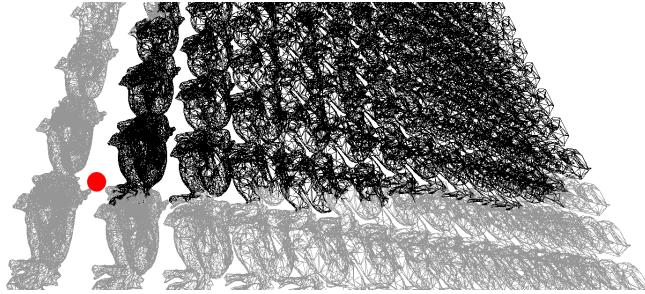
```

Algorithm 7 makes good use of our observation, eliminating the need to check a huge number of clusters for most instances. However, my algorithm feeding data to the next frame presents some challenges. Line 7 handles the additional complexity; there is a chance that clusters at the resolution we wish to draw at are not available. Because our clusters are dispatched assuming the scene will be similar to the last frame, any requested increase in resolution may take a frame of latency per level to apply: fast enough that it should never present an issue.

3.4. Cluster Culling

An engine based around instances and LOD chains may utilize *instance culling* to save time in rasterisation. Instance culling aims to test, as quickly as possible, if an object is visible from the current viewpoint. As it is based on arbitrarily sized objects, this has some of the same flaws as LOD chains. If we instead focus on culling clusters (which are generally smaller), we can achieve much finer grained culling. This technique was used in industry before cluster based multiresolutions, as cluster-based rendering has been shown to be efficient for GPU-driven rendering systems [25].

A simple culling technique we apply is frustum culling, not drawing a cluster if it is outside the camera frustum (Figure 21). The frustum can be represented by six planes, which we extract from the model-view-projection matrix [26]. These planes will be in object-space, and, from error calculations, each cluster contains a bounding sphere in object-space. We then cull clusters if their bounds are on the negative side of any plane.



**Figure 21: Frustum culling per-cluster from the viewpoint ●.
Instances may be partially culled.**

This requires testing every selected cluster, but DAG-TRAVERSE (§ 3.3.1) can be optimised further. Our error function requires the DAG to form a nested bounding volumes hierarchy, enforced for monotonicity (§ 2.1.3). As such, a bound of a cluster contains the bounds of all its children, and placing a cluster’s bound outside the frustum would imply the entire hierarchy of clusters descending from it may be culled. DAG traverse can use this fact to cull as early as possible. At the coarsest grain, a successful cull on the root cluster is equivalent to instance culling.

3.5. Cluster Rendering

In this section, I describe implementing a system capable of rendering a list of selected clusters in the context of primitive shading (§ 2.3.1) and mesh shading (§ 2.3.2).

3.5.1. Primitive Shading

My first procedural backend uses the primitive shading pipeline. As discussed in § 2.3.1, this is a limited backend that makes for complex code and suboptimal performance, but is supported on older hardware.

To draw selected clusters as a mesh, we must push all primitives of all clusters into a staging index buffer. These indices will reference a shared vertex buffer. The index buffer's length is normally included as part of the draw command, however we need to allow these to vary with the primitives pushed. Instead, I call this **indirectly**, passing three buffers instead of two: an index buffer, vertex buffer, and a *parameter buffer* (Listing 6).

```
1 pub struct DrawIndexedIndirectCommand {
2     pub index_count: u32,
3     pub instance_count: u32,
4     pub first_index: u32,
5     pub vertex_offset: i32,
6     pub first_instance: u32,
7 }
```

Rust

Listing 6: The layout of an indirect draw command buffer in Vulkan.

To draw every instance at once, I use *multi draw indirect*, a GPU feature that allows me to queue as many indirect draws as are found in a parameter buffer, reducing draw calls required from per-instance to per-scene. This is necessary as I cannot rely on instanced rendering; each instance may consist of a different set of triangles.

Algorithm 8: Draw clusters (Primitive)

```
1 procedure CLUSTERSTOPRIMITIVES (selected-clusters)
2     Allocate a shared staged-indices buffer to store all the scene's triangles.
3     total-tris := 0
4     parallel for each instance  $i$ 
5         first-index $_i$  := ATOMICADD(&total-tris $_i$ , |selected-clusters  $\rightarrow$  triangles|)
6         parallel for each  $c$  in selected-clusters $_i$ 
7             let start = ATOMICADD(&index-count $_i$ , | $c \rightarrow$  triangles|)
8             staged-indices[first-index $_i$  + start ..] := ( $c \rightarrow$  triangles)
9         end
10    end
11    DRAWINSTANCESINDIRECT()
12 end
```

The challenge faced by Algorithm 8 is filling the buffer with triangles **contiguously**. The end result will be a staging buffer tightly packed with clusters for each instance:

$$\text{staged-indices} : 0, \dots, 4, \dots, 5, \dots, 8, \dots, 9, \dots, 3, \dots, 1, \dots, 9$$

$\underbrace{0, \dots, 4}_{\text{cluster } 0_0}$	$\underbrace{5, \dots, 8}_{\text{cluster } k_0}$	$\underbrace{9, \dots, 3}_{\text{cluster } 0_n}$	$\underbrace{1, \dots, 9}_{\text{cluster } k_n}$
Instance 0		Instance n	

(9)

Each instance's triangles are marked by indirect arguments `first_index` and `index_count` (Listing 6). In a shader, communication between workgroups is limited, so I utilise atomic operations. In GLSL, the operation $\text{ATOMICADD}(x, y)$ both returns the value x_{prev} directly before the addition, and has the side effect of setting $x \leftarrow x_{\text{prev}} + y$. This provides a way to allocate contiguous chunks in our staging buffer – allocate a shared length value, atomically add the size of the data we wish to push, and place the data at result. In fact, this shared

length value is already allocated in the indirect draw command’s `index_count` field. Moreover, the atomic additions calculate the number of triangles for indirect rendering.

A notable similar implementation is AMD’s GeometryFX [27], which uses a similar staging index buffer to perform fine-grained cluster culling of a scene, with a similar goal of improving rasteriser efficiency.

3.5.2. Mesh Shading

My primitive shading algorithm has some key disadvantages:

- I allocate a result buffer large enough to store every triangle we may have selected.
- I copy indices from the original cluster data to the result buffer every frame, which will later be copied again for rasterisation – a waste of memory bandwidth.
- The use of atomic operations, while minimal, is undesirable for performance.

Mesh shading (§2.3.2) began as an extension to the project, but quickly became central due to its high performance in procedural geometry applications. Algorithm 9 uses the mesh shading pipeline to remove the need for a staging buffer. By emitting a task for each cluster, we defer some computation until we have entered the graphics pipeline. This allows all clusters from all instances to be processed in parallel, and we no longer work around a large contiguous list of indices. There is little complexity with this method, as our multiresolution has been structured ideally for mesh rendering.

Algorithm 9: Draw clusters (Mesh)

```

1 procedure DRAWCLUSTERS (selected-clusters)
2   parallel for each instance
3     parallel for each cluster in selected-clusters
4       | EMITMESHLETS(cluster → meshlets)
5     | end
6   | end
7 end
```

With knowledge of the entire render pipeline, I can optimise ADAPTIVESELECT (§ 3.3.2). Task shaders support indirect invocation, so we can effectively merge adaptive selection into Algorithm 9. This presents a pipeline for selecting and drawing with no intermediate memory, as it defers all computation until the draw call.

3.5.3. Meshlet Compression

Mesh shading pipelines forgo the vertex stage as described in § 2.3.2, and mesh shaders retrieve their own set of vertices. Our meshes are large, so we are forced to use 32-bit integers to address single vertices, but addressing a meshlet can be optimised.

Within the meshlet we will likely reference a vertex more than once within our assigned 126 triangles and 64 vertices. To exploit this, we can add a level of indirection, recording every unique index, and storing vertices as single byte offsets within that list, a format known as an indexed meshlet. This turns the size of our structure from 1512 bytes to 634 bytes. Each vertex takes $4 + n$ bytes to store compared to $4n$, for n uses, with $n > 1$ on average.

3.6. Render Graph

I have described and implemented two separate algorithms to select a view-dependent approximation of a mesh. Both can output to the primitive and mesh shading pipelines, with DAG-TRAVERSE using more intermediate memory than my own ADAPTIVESELECT.

A *render graph* is a high level directed graph representation of the operations used to render a scene [28]. I describe my methods in this way to present an easy path for integration into an existing render graph based engine. Figure 22 shows their interactions, noting that we have two choices for both the selector and renderer and so four paths for rendering.

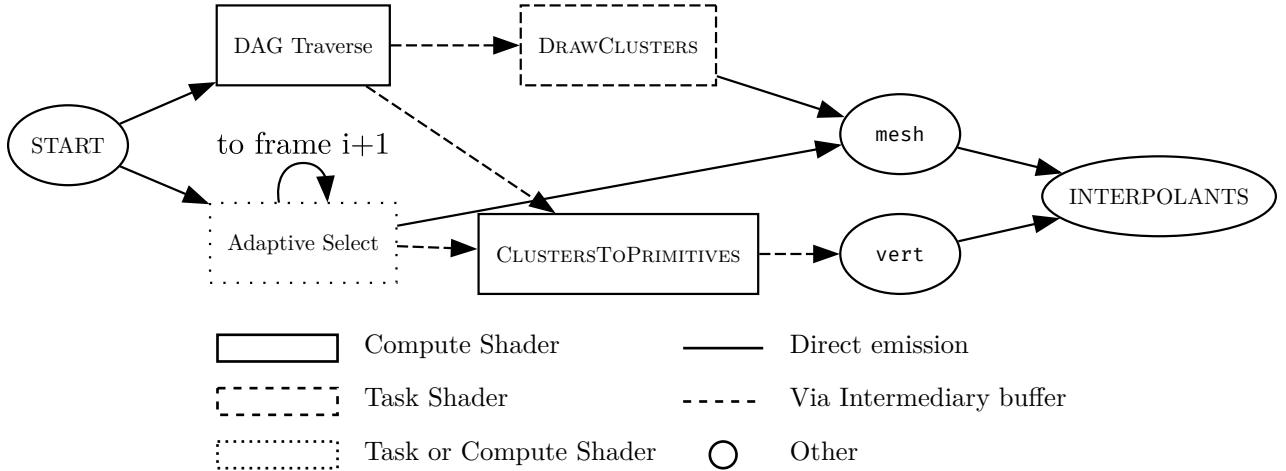


Figure 22: Graph of possible draw instructions given in various pipelines.
Cluster culling is embedded into selection.

3.7. Repository Overview

This project has multiple interoperable parts, so is divided into *crates*, Rust packages, and built with Cargo, Rust’s package manager, build system, and test runner. Their intra-dependencies are outlined in Figure 23. My own code is released under the GPLv3 license.

Path	Comments	Lines of Code
shaders/src	Source GLSL and compiled SPV shaders	632
src/app/	ECS application control and benchmarking	2255
src/draw_pipelines/	Implementations of cluster selection and drawing	2556
src/gui/	EGUI \leftrightarrow Vulkan interop	1064
src/utility/	Unsafe Vulkan handling (§ 3.2)	3446

Table 2: The crate for the renderer, /ash_renderer (9832 lines of code)
As a real-time application, unit tests were not suitable for correctness.

My project renders GUI with EGUI, a Rust native immediate mode GUI. Over the course of a frame, this builds up a series of shapes required to render the interface. Rendering must occur with a separate library for the specific graphics API – this is contained in `src/gui/`, and contains code ported (under the MIT license) from an existing Vulkan integration [29].

Path	Comments	Lines of Code	Coverage
src/mesh/	Half-edge mesh, partitioning, simplification	2324	66.3%
src/lod/	Algorithms to generate multiresolutions	456	87.4%

Table 3: The crate for the Multiresolution Generator, \baker (3034 lines of code)

Path	Comments	Lines of Code	Coverage
common/	Shared data structures	675	84.4%
common_renderer/	Backend agnostic renderer utilities	672	70.5%
metis/	METIS Rust bindings and utility	812	90.1%
evaluation/	Evaluation of mesh simplification quality	335	79.2%
vendor/	Git submodules, containing the METIS C source (omitted)	—	—

Table 4: Miscellaneous other crates.

METIS [17] is written in C and does not exist in Cargo. For interoperability, I wrapped it with an auto-generated wrapper, `metis/`, to provide a Rust API for the underlying C source via `unsafe extern` methods. I then wrapped this with further methods that allowed safe usage. The METIS source code is unmodified and the line count is omitted. METIS is distributed under the Apache 2.0 License, similarly to many of my dependencies; all licenses are recorded in Appendix D.

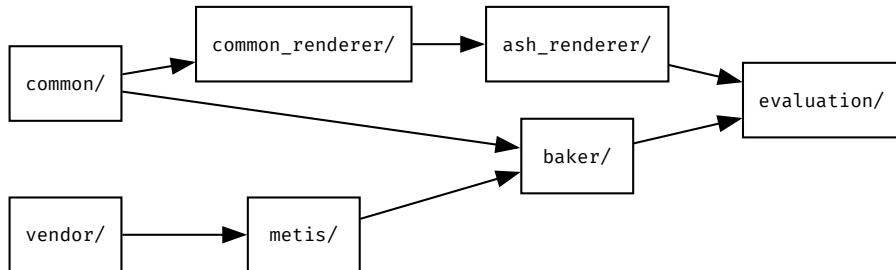


Figure 23: Dependency graph of my crates

Chapter 4

Evaluation

My evaluation covers four qualities which reiterate the core goals of the project, and show that I pass my success criteria and all requirements:

Mesh generation Testing my mesh simplifier against a high accuracy, but slow, error metric to measure the quality of simplified meshes. I will evaluate how the constraints of a multiresolution effect mesh quality.

Visual quality How well my simplifier and error function work together to produce high quality view-dependent LODs, using recordings of the software.¹² I will calibrate my system's target quality against the Unreal Engine 5's Nanite for further evaluation.

Rasterisation efficiency Measuring the number of triangles placed onto the screen, to confirm the error function's utility.

Performance Measuring the time and space efficiency of our selection and procedural geometry algorithms, with focus on viability for usage in real-time rendering.

4.1. Mesh Generation

To evaluate the quality of multiresolution generation, I compare them against LOD chains, which represent simplification without the limits imposed from locked edges (§ 2.1.1), which we have seen can have a large impact on mesh quality (§ 3.1.4). To evaluate the quality of quadric error metric guided simplification (§ 2.2.3), I compared against the simplifier included in `meshopt`, a mesh processing library with many tools that I have used throughout (§ 2.5.2).

I will estimate error of a simplified mesh M_i , for i triangles, with average square distance from its surface to the original, M_n , with n triangles [20]. I take uniform random samples of the surfaces of these to generate X_n and X_i . The final error is:

$$E_i = \frac{1}{|X_n| + |X_i|} \left(\sum_{v \in X_n} \text{sqrdist}(v, M_i) + \sum_{v \in X_i} \text{sqrdist}(v, M_n) \right) \quad (10)$$

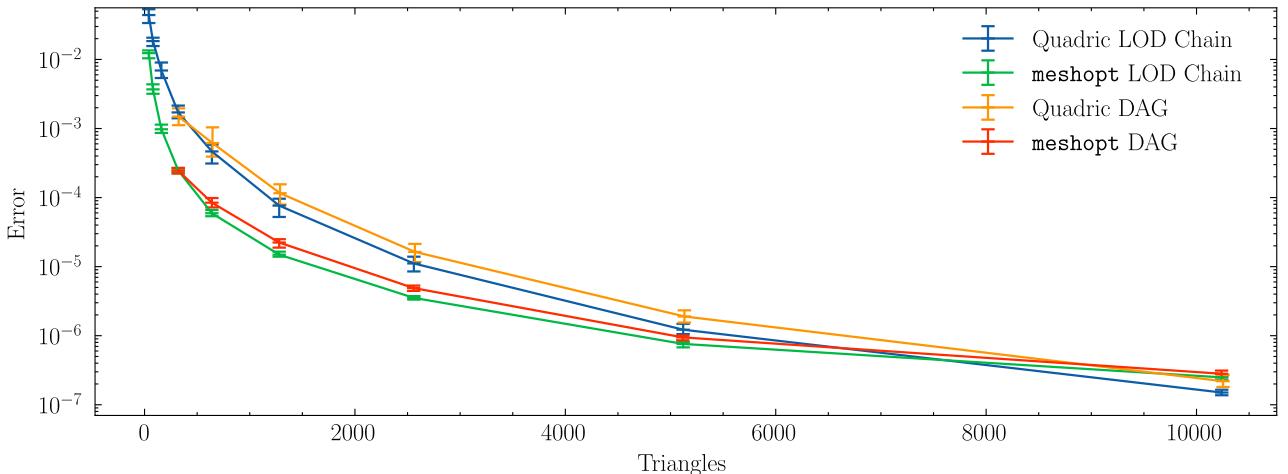


Figure 24: Mean and 10th percentiles of E_i of simplified copies of a series of $n = 20\text{K}$ triangle meshes.

¹²An extension to the project.

The small difference between the errors introduced in LOD chains and DAGs, depicted in Figure 24, shows the impressively small error incurred by the locked boundaries of multiresolution generation, almost within margin of error. As an extension for evaluation, I extended my generator to support `meshopt` simplification. The result for quadric error against `meshopt` (Figure 25) suggests that my simplifier performs worse for uniform tessellations when compared to long-standing implementations. However, feature preservation works as expected, with the object’s shape retained in both.

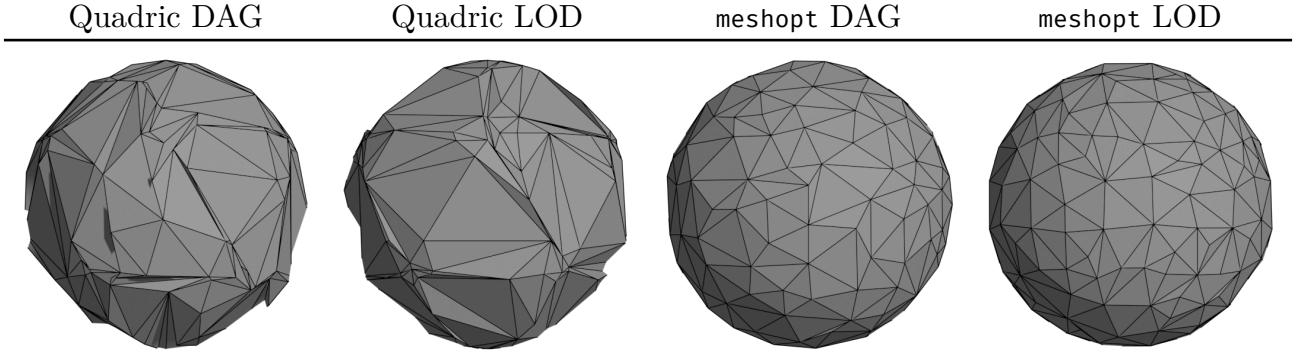


Figure 25: 20K sphere after 6 levels of simplification. Multiresolution DAGs have similar quality to LOD chains, but quadrics are far more jagged and contain more artefacts than `meshopt`.

4.1.1. Discussion

During the simplification stage of multiresolution generation, one requirement is that groups must have an appropriate *triangle to locked edge ratio* – a group with a boundary containing too many locked edges (over half its triangle count) cannot be adequately simplified. I chose to use clusters of 280 triangles, as it was the smallest count large enough to fulfil this. This results in many clusters containing partially filled meshlets, wasting some space.

Integrating an existing mesh simplifier shows superior results to my own. However, unlike my implementation, `meshopt` cannot distinguish between group boundaries and mesh boundaries, so must lock all boundaries. This makes it unsuitable for non-watertight meshes.

The final multiresolution generator operates at speeds usable for a quick iteration process. The 1M triangle dragon mesh takes 9.74s, the 28M triangle Lucy mesh takes 355.1s. Flamegraph analysis shows that generation time is dominated by METIS, followed by simplification.

4.2. Visual Quality

My system has a configurable screen space error threshold τ used to control the error function (§ 2.1.3). I expect the error function to produce view dependent LODs with screen-space quality proportional to τ . I will evaluate this at various values of τ , investigating the error function’s behaviour and the impact of my mesh simplification. This will be compared against Nanite [1], informing the results of further tests by calibrating τ for equal visual quality.

To evaluate screen-space quality, I take frame-aligned recordings within both my renderer and Unreal Engine, measuring against references of full resolution meshes. I used two metrics:

- ColorVideoVDP [30], which reports results using Just-Objectionable-Difference (JOD)¹³. This exists on a perceptually linear scale of 0–10, with 10 being best.

- Netflix’s VMAF [31] , which outputs a similar score, Visual Information Fidelity (VIF) on a 0–100 scale, with 100 being best.

VMAF is designed to detect visual differences from video compression, but produces good results and is an industry standard tool [31]. ColourVideoVDP respects colour data [30], so the results may more accurately respect quality, as mesh simplification may change the colour of a mesh in unexpected ways, e.g. altering the normal of a point, in turn altering lighting.

Metric	Simplifier	My error function					Nanite
		$\tau = 0.25$	$\tau = 0.20$	$\tau = 0.15$	$\tau = 0.1$	$\tau = 0.025$	
JOD (VDP)	Quadric Error	7.906	8.168	8.659	9.227	9.776	9.3838
	meshopt	8.332	8.595	8.924	9.286	9.849	
VIF (VMAF)	Quadric Error	75.17	79.93	88.49	97.92	99.98	98.69
	meshopt	82.21	87.27	93.28	99.20	99.98	

Table 5: Results for visual quality metrics for a test scene (Appendix C). Both tools present approximately proportional results. A JOD close to 9 is visually near indistinguishable from reference.

The difference in JOD from My own quadric error and renderer at $\tau = 0.1$ to Nanite is 0.15. ColourVideoVDP expects to see this correspond to just a 5% average increase in user preference [30]. Considering this data, I calibrated $\tau = 0.1$ as ‘high quality’ for further evaluation. Additionally, with my error function, my simplifier performs acceptably well against `meshopt`, despite the high visual error when zoomed in (Figure 25).

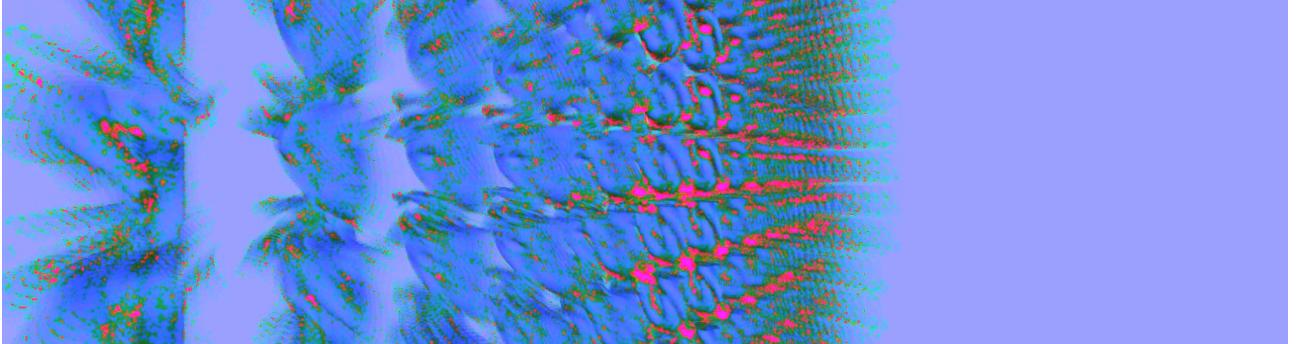


Figure 26: VDP high sensitivity heatmap for $\tau = 0.1$ from little error (Blue) to high error (Red).

It is of note in Figure 26 that error is more common around instances further away, suggesting the error function is not quite linear. Additionally, error is most pronounced on the silhouette of shapes. This aligns with intuition, as it is the region of highest contrast.

4.3. Rasterisation Efficiency

In this section, I wish to show that my renderer’s error function improves the efficiency of rasterisation over full-resolution meshes. The error function’s goal is to estimate the screen-space triangle density of clusters. As such, clusters drawn should be proportional to the area in screen-space occupied by meshes, and the error threshold τ . I measure this by testing the effect of varying both τ and the resolution of source meshes, and comparing against Nanite. I use a benchmark of a grid of 2500 instances of each source mesh (Appendix C).

¹³v0.4, 37.84 [pix/deg], Lpeak=200, Lblack=0.2, Lrefl=0.3979 [cd/m^2], standard_fhd.

Getting the data. My renderer hooks into Vulkan’s query system, allowing us to record the *clipped primitives* statistic: the number of triangles in view each frame. Due to frustum culling (§ 3.4), this represents the bulk of the cost of rasterising in my system. I obtain Nanite’s triangle readouts with the command `NaniteStat`, which draws the readout on screen and had to be recorded manually, hence the lower resolution plot.

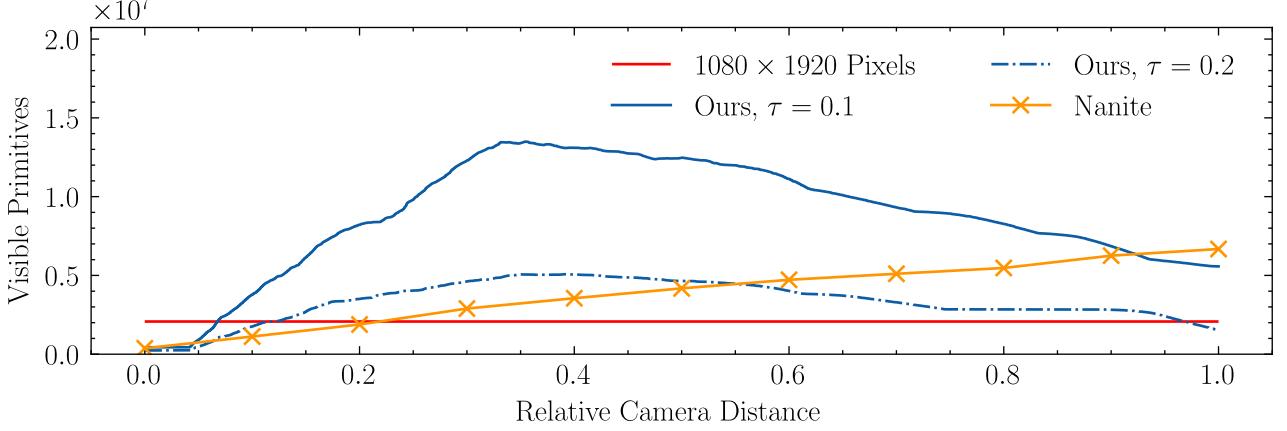


Figure 27: Triangles drawn by distance, in a scene of 2500 instances each with $T = 1\text{M}$ triangles.

We see in Figure 27 that my system peaks at 15M triangles, which is a clear improvement in rasterisation efficiency compared directly rendering the 2.5 billion triangles contained in the scene. However, we draw more than Nanite and more than the 2M pixels contained in our 1080p display (as stated, we should target a minimum 1 triangle per pixel).

Higher than expected triangle count. I believe my system outputs more triangles for a few reasons: Many clusters will exist on the far sides of meshes, which my system still rasterises as I have no occlusion culling. Additionally, because the angle clusters are viewed at is not accounted for in the error function, we may under-estimate the screen-space triangle density for clusters viewed at an extreme angle. Finally, against Nanite, the weakness in my mesh simplifier seen in § 4.2 requires more triangles to be drawn to match visual quality.

With this in mind, I set $\tau = 0.2$ to be a ‘rasterisation equivalent’ error, from which we expect roughly equivalent performance to Nanite. From the previous section, this has a higher visual error, but not so high as to be an unrealistic comparison. I will use this to compare the two renderers based on the work required to output an equal number of triangles.

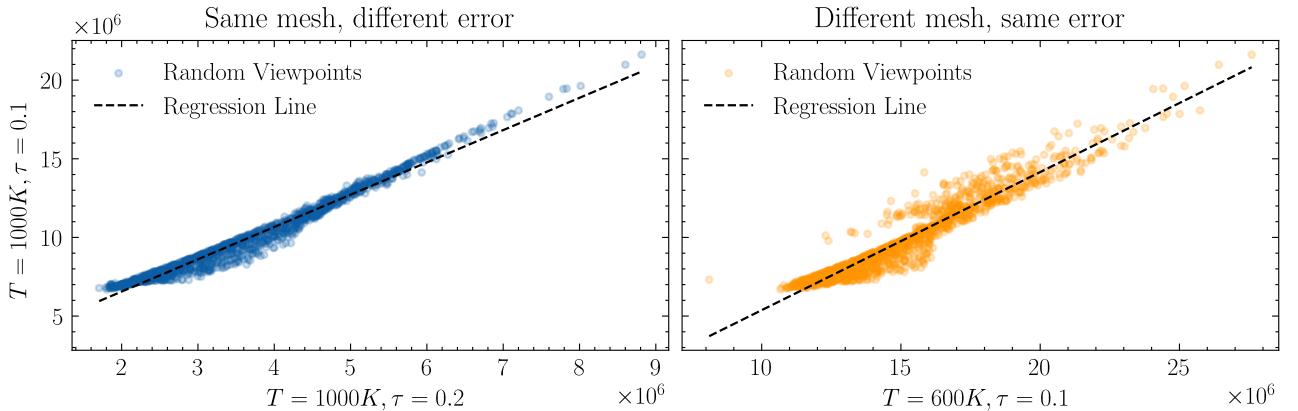


Figure 28: Relations of triangles drawn for various error thresholds τ and mesh triangle counts T , from random viewpoints in a sphere looking towards $\vec{0}$.

When measuring the response of the error function for random viewpoints, Figure 28 shows it behaves as expected:

A linear response to changing the threshold. We see a very desirable regression gradient of 2.05; We can say confidently that doubling τ doubles the triangles drawn.

A linear response to different scenes. Changing the source triangle count of a mesh should not affect final triangle count. A slope of 0.877 is not quite equal to the idea ratio of 1, however the difference in the screen-space area taken by the meshes contributes to this ratio – a mesh occupying more space on the screen requires more triangles to represent at the same uniform error in screen-space.

4.4. Performance

I benchmark each method to analyse their performance characteristics. I created two scenes, pictured in Appendix C, one optimal for traditional LOD chains, and another presenting their worst case. Results are very consistent between runs, with slight noise from background processes. As such, I take the minimum times between runs, with deviations shaded above.

GPU execution times are gathered from Vulkan timestamp queries. These have minimal probe effect within the region we measure – the main cost of measurement is transferring data back to the CPU, which occurs between rendering frames. All data in this section is captured on a GTX 1660, one of the lowest end graphics cards to support mesh shading, and so demonstrating a viable product on this card represents viability across most modern PCs.

4.4.1. Comparing Implemented Multiresolutions

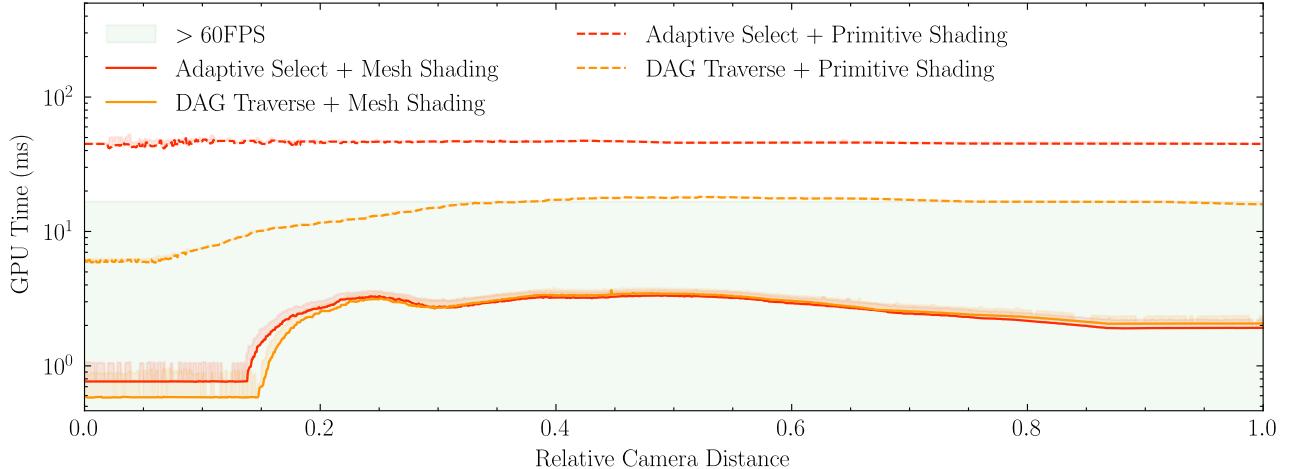


Figure 29: Performance of the 4 configurations of my renderer with 2500 instances, $\tau = 0.1$.
 $\{\text{Mesh, Primitive}\} \times \{\text{ADAPTIVESELECT, DAG-TRAVERSE}\}$

We can see in Figure 29 that mesh shading dominates primitive shading for procedural geometry in my engine. For context, the full resolution scene (2.5 billion triangles) takes a minimum of 472 ms to render in full resolution. Both mesh shaders execute in under 4 ms (250 fps), and primitive shading being an order of magnitude slower.

My algorithm, ADAPTIVESELECT (§ 3.3.2), has very similar performance across the board to DAG-TRAVERSE while saving intermediate memory (with mesh shading). At low relative camera distances, large numbers of instances in the scene are culled. In these ranges, my algorithm is partially slower. This is due to DAG-TRAVERSE’s more efficient culling; I mentioned in § 3.4 that it prunes sections of the DAG if their parents are culled.

Memory Usage. The 2500 instances require: 192 MB to store the staging indices of the primitive pipeline, 38.82 MB of GPU memory to store the intermediate cluster selection for DAG-TRAVERSE, and just 29.31 KB for ADAPTIVESELECT’s indirect dispatch buffer.

4.4.2. Comparing Alternate Tools

In this section, I aim to compare my fastest implementation (ADAPTIVESELECT with mesh shading) against most methods mentioned throughout this dissertation. Comparison against LOD chains requires drawing approximately the same number of triangles. We cannot directly apply the error function to an LOD chain. Instead, they typically have set *transition distances*, the distances where each level of detail will transition to the next. I calibrated these distances automatically by simulating the response of a multiresolution.

Measuring Unreal. Unreal measures GPU time with STARTFPSCHART. As Nanite is embedded within an engine, which may be performing numerous other tasks, constant overhead cannot be compared. To offset this, I ran the benchmark in Unreal on an empty scene, measuring a 1.4-1.7 ms constant GPU time overhead. It is unclear how much or any of this is from Nanite; the background processes of the engine makes this difficult to isolate, one of the disadvantages of Nanite for research.

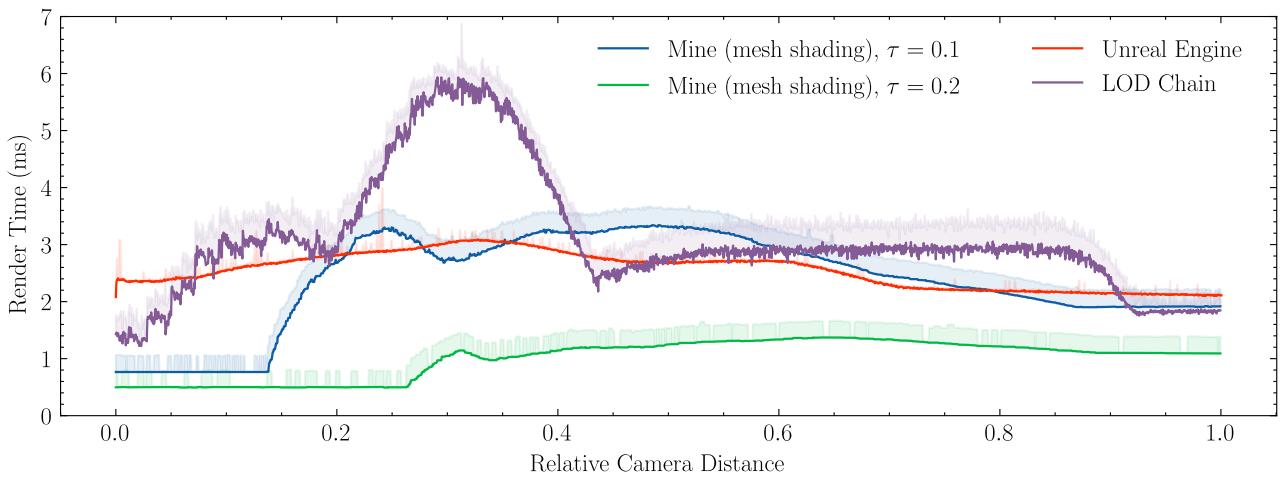


Figure 30: Performance characteristics of our system and Nanite.

Nanite. In Figure 30 we see my method can match Unreal at $\tau = 0.1$. It is difficult to draw a conclusion from this; Unreal is a complex system with many background tasks, any of which may distract from the results. From profiling with Nsight, roughly 1ms of this noise floor is background tasks, so I can say the true cost of Nanite then exists somewhere between $\tau = 0.1$ and 0.2 , a good result for my system.

LOD. Compared to CPU-driven LOD chains, we see a much larger improvement than I anticipated, showing the remarkable gains from GPU-driven rendering in general. I expect implementing the LOD chain with mesh shading would improve its performance.

Error. Comparing $\tau = 0.1$ to $\tau = 0.2$, we can see the performance characteristics are not quite linear when close to the camera, despite a linear increase in triangles. This corresponds to rendering at higher resolutions, and so the lower resolution regions of the DAG, that will not be selected, present some small overhead.

My next scene contains a single instance of the Stanford Lucy, seen in the introduction (Figure 2), from which I wish to demonstrate an architectural advantage over LOD chains that could not be solved by porting to mesh shading.

Method	GPU Time	Profiler samples		
		Task	Mesh/Vert	Frag
Adaptive Select (Mesh) $\tau=0.1$	1.28 ms	66%	29%	5%
		0.85 ms	0.37 ms	0.064 ms
LOD Chain	9.15 ms		97.5%	2.5%
			8.92 ms	0.23 ms

Table 6: Frame time analysis of a single frame rendering the Lucy model [21] in Figure 2, using NVIDIA Nsight. The fragment shaders for both methods are identical.

The LOD chain in this scene is drawing at full resolution, as is required from the position of the camera. As such, Table 6 shows a massive increase in performance from ADAPTIVESELECT over the LOD chain that results from this fundamental limitation.

4.4.3. Scene Complexity

Finally, to measure the response of my system over ranges of scene complexity, I measured the performance of the system for scenes up until the system dropped below 30 fps.

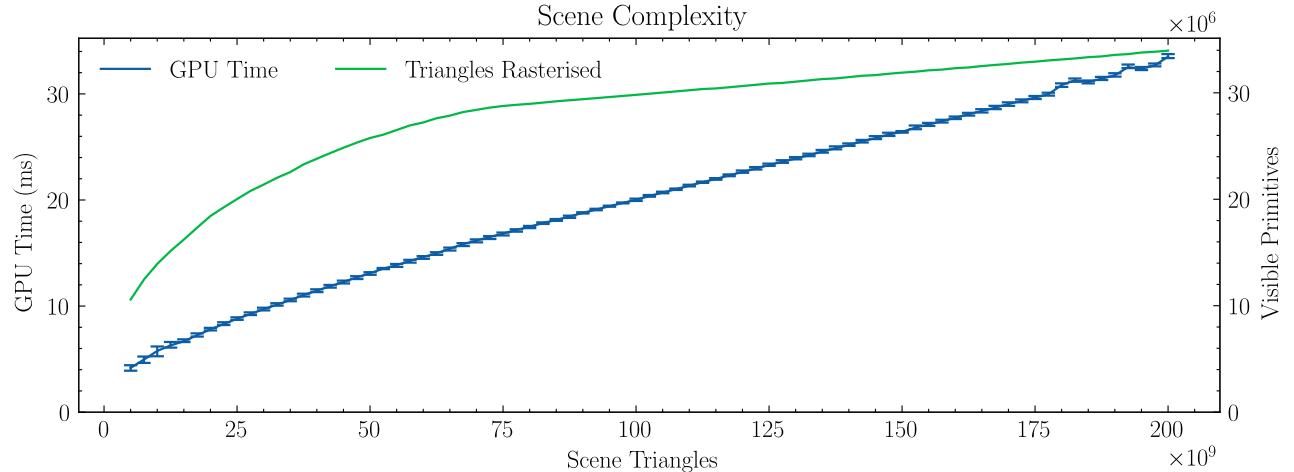


Figure 31: Primitives drawn and GPU time with 10th percentile error for massive scenes.

Visible triangles scales *sublinearly* to scene triangles, until the point where we render at the lowest quality – my system never completely removes instances. Time complexity scales well, but is still linear. I reached 200 billion triangles of source data before dropping to 29.5 fps, creating the view below. In this regard, I have completed the goal of my project 200 fold.



Figure 32: 200 billion triangles of source data. It is of note that rendering this many *unique* triangles is a different challenge, as the amount of memory required to store this would be huge.

Chapter 5

Conclusion

The project met both success criteria, improving the **frame time** and **rasterisation efficiency** compared to full-resolution rendering, and exceeded them by improving on LOD chains. As such, the dissertation's title was over-achieved, with the renderer being able to handle scenes of upwards of 100 billion triangles of source data at cinematic frame rates. Many extensions were implemented: generating LOD chains, cluster culling, temporal selection, the `meshopt` simplifier, and using perceptual metrics. The project was made feasible by the use of mesh shaders, a technology I was not even aware of while writing the project proposal, and without them my results would have been very disappointing. By the end, I had gained a deep understanding of the difficulties core to procedural geometry and multiresolutions, and created a system useful for further work, which is open-source.

5.1. Work Completed

Multiresolution Generator In order to investigate methods to render multiresolutions, I generated them from standard mesh formats. Our multiresolution scheme mirrored that of Unreal Engine's Nanite. I utilised the open-source graph partitioner, METIS, augmented with my own control flow for reliably uniform clusterings, creating *well-conditioned DAGs*. I implemented a simplifier with *quadric error metrics* that performed well enough for good visual quality.

Multiple procedural geometry backends I investigated implementing a multiresolution renderer in Vulkan on the shoulders of both classical *primitive shading* and modern *mesh shading*. I found mesh shading to benefit performance, intermediate memory usage, and code simplicity. Additionally, mesh shading allowed for memory savings with *meshlet compression*, by decompressing the meshlet within the mesh shader.

Efficient selection algorithms I described and implemented two methods to select clusters from a multiresolution to drive the procedural geometry. The first method was created in the image of Nanite's *persistent threads* selector, while the second was the product of original research on deferring cluster selection until the graphics pipeline, utilising *task shaders*. I implemented a user adjustable error function to drive creating view-dependent LODs, shown to have uniform visual quality (§ 4.2). I also extended my selection algorithms to support culling clusters that are not in view.

Benchmarking I created a series of benchmarks, in both my 3D engine and Unreal, to verify the expected properties of the system. I verified that the technique produces draws triangles scaling sublinearly to the scene's complexity. I also implemented a LOD chain renderer to aid in comparison. In the end I found a GPU-driven rendering engine is so efficient that, even with the complexity of multiresolutions, they outperform CPU-based resolution selection.

5.2. Reflections

Using Rust over C++, the typical language for graphics projects, seemed like a risk going into the project, but, looking back, it saved me both time and effort. The project resulted in

a much larger than anticipated codebase, at over 15,000 lines of code, 10,000 of which are in the renderer, with 3,000 directly accountable to the verbosity of Vulkan. I found the state of packages for graphics in Rust to be strong, and its standout feature was the ease of use of Cargo, Rust’s package manager and build system. The language’s macro system is also very powerful, allowing a large amount of boilerplate to be hidden.

The generator was the first part of the project to be implemented, so it is the part I have the most hindsight on. Starting by clustering the triangle mesh using METIS does not guarantee clusters have the ideal number of triangles for meshlets. Having later focused the project on mesh shading, I would now start by partitioning the mesh into meshlets, forcing every cluster to contain an integer number of meshlets. This effect would have carried down the DAG, making everything slightly more efficient.

Learning Vulkan for the project was a long process. I had originally planned to use the higher level Web GPU specification, as its Firefox implementation is a Rust package, `wgpu`. As such, the multiresolution generator was mostly developed with a debug viewer written in `wgpu`. However, after discovering mesh shading (not supported by `wgpu`) and the massively improved performance I could achieve, I quickly shifted everything to Vulkan. Picking a cutting-edge project also opened me to my first student research contributions, and I am happy to have created, to my knowledge, the first open-source GPU-driven renderer of this technique.

5.3. Future Work

Cluster-based rendering engines, and GPU-driven rendering as a whole, present major opportunities for optimisation with the availability of GPGPU. I investigated *cluster selection* and *cluster culling*, which form a foundation on which a large amount of further projects could stem. I present a number of these:

Foveated Rendering One extension that was sadly never implemented was foveated rendering, an optimisation that would have altered the error function to take into account *where the user was looking on the screen*.

Anisotropic selection Further extension to the error function includes *anisotropic selection*, taking into account the direction a cluster faces. This would work on the principle that error is more visually apparent on the silhouette of a shape (seen in § 4.2).

Occlusion Culling In implementing culling, we saw the relatively simple frustum based culling. Culling occluded clusters presents a further avenue for optimisation.

Data streaming This extension would have allowed the higher resolution regions of the multiresolution buffer to be loaded into GPU memory on demand, allowing memory to be saved in high complexity scenes.

Aggregate Geometry and Mesh Deformation Both of these are classical limitations of multiresolutions which were not investigated in this project. Solving either would form an interesting research problem.

Multiresolution Generation Improving my simplifier and mesh generator based on the results and discussion in § 4.1.

Finally, the ultimate goal of this project is to integrate this code into an existing Rust-based engine. Currently, most multi-platform graphics libraries do not support the concept of mesh shaders, making this integration impractical at the current time, however I expect support for this feature to mature in the coming years.

Bibliography

- [1] B. Karis, R. Stubbe, and G. Wihlida, “A Deep Dive into Nanite Virtualized Geometry,” *SIGGRAPH*, 2021.
- [2] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, “Batched Multi-Triangulation,” *IEEE Visualization*, 2005.
- [3] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, “BDAM – Batched Dynamic Adaptive Meshes for high performance terrain visualization,” in *Computer Graphics Forum*, 2003, pp. 505–514.
- [4] “Half-Edge Data Structures.” [Online]. Available: <https://jerryyin.info/geometry-processing-algorithms/half-edge/>
- [5] “Geometry Processing Algorithms.” [Online]. Available: https://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf
- [6] R. Ronfard and J. Rossignac, “Full-range approximation of triangulated polyhedra.,” in *Computer graphics forum*, 1996, pp. 67–76.
- [7] “Edge Collapse Conditions.” [Online]. Available: <https://stackoverflow.com/questions/27049163/mesh-simplification-edge-collapse-conditions>
- [8] “Tesselation.” [Online]. Available: <https://docs.vulkan.org/spec/latest/chapters/tessellation.html>
- [9] “Geometry Shading.” [Online]. Available: <https://docs.vulkan.org/spec/latest/chapters/geometry.html>
- [10] C. Kubisch, “Mesh Shading for Vulkan,” *Khronos Blog*, 2022.
- [11] A. Patel and T. Riddell, “D3D12 Work Graphs Preview,” *DirectX Developer Blog*, 2023.
- [12] H. Hoppe, “Progressive meshes,” in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, in *SIGGRAPH '96*. Association for Computing Machinery, 1996.
- [13] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, “Quick-vdr: Interactive view-dependent rendering of massive models,” *ACM SIGGRAPH 2004 Sketches*. p. 22–23, 2004.
- [14] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, “Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models,” *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 796–803, 2004.
- [15] F. Ponchio, “Multiresolution structures for interactive visualization of very large 3D datasets,” 2009.
- [16] “NVIDIA Nsight.” [Online]. Available: <https://developer.nvidia.com/nsight-graphics>
- [17] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

- [18] “Ash.” [Online]. Available: <https://github.com/ash-rs/ash>
- [19] “Vulkan-Hpp.” [Online]. Available: <https://github.com/KhronosGroup/Vulkan-Hpp>
- [20] M. Garland and P. S. Heckbert, “Surface simplification using quadric error metrics,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 209–216.
- [21] Stanford University Computer Graphics Laboratory, “The Stanford 3D Scanning Repository.” [Online]. Available: <http://graphics.stanford.edu/data/3Dscanrep/>
- [22] T. Härkönen, “Advantages and Implementation of Entity-Component-Systems,” 2019.
- [23] N. Henning, “Vulkan Subgroup Tutorial.” [Online]. Available: <https://www.khronos.org/blog/vulkan-subgroup-tutorial>
- [24] Y. Lin and V. Grover, “Using CUDA Warp-Level Primitives.” [Online]. Available: <https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/>
- [25] U. Haar and S. Aaltonen, “GPU-Driven Rendering Pipelines,” *SIGGRAPH*, 2015.
- [26] G. Gribb and K. Hartmann, “Fast extraction of viewing frustum planes from the world-view-projection matrix,” *Online document*, 2001.
- [27] “GeometryFX.” [Online]. Available: <https://gpuopen.com/geometryfx/>
- [28] R. Loggini, “Render Graphs.” [Online]. Available: <https://logins.github.io/graphics/2021/05/31/RenderGraphs.html>
- [29] MatchaChoco010, “egui-winit-ash-integration.” [Online]. Available: <https://github.com/MatchaChoco010/egui-winit-ash-integration>
- [30] R. K. Mantiuk, P. Hanji, M. Ashraf, Y. Asano, and A. Chapiro, “ColorVideoVDP: A visual difference predictor for image, video and display distortions.” 2024.
- [31] Z. Li, A. Aaron, I. Katsavounidis, A. Moorthy, and M. Manohara, “Toward A Practical Perceptual Video Quality Metric,” *Netflix Technology Blog*, [Online]. Available: <https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652>

A Minimal Mesh Pipeline

Task and mesh shaders add a number of calls that can be accessed in the mesh and task shaders, which otherwise act exactly as compute shaders [10]:

- `SetMeshOutputsEXT(uint verts, uint primitives)` to set how many primitives its workgroup will create.
- The arrays `uvec3 gl_PrimitiveTriangleIndicesEXT[]` and `vec3 gl_MeshVerticesEXT[]` form a triangle list for the meshlet.
- Additional interpolants may be outputted similarly to a vertex shader.
- The task shader calls `EmitMeshTasksEXT(uint x, uint y, uint z)` to spawn $x \times y \times z$ mesh workgroups in a grid. It may optionally push data to a *payload*, data shared with all its mesh invocations, which we use to coordinate the work between invocations.

Minimal Task Shader Demonstrates sending payload data, emitting mesh shaders.

```
1 #version 450
2 #extension GL_EXT_mesh_shader: require
3 #extension GL_ARB_separate_shader_objects: enable
4 layout (local_size_x = 2, local_size_y = 1, local_size_z = 1) in;
5 struct MeshTaskPayload {
6     uint meshlet_index[2];
7 };
8 taskPayloadSharedEXT MeshTaskPayload payload;
9 void main() {
10     // Run with meshlets.len() / 2 work groups.
11     uint meshlet_index = gl_WorkGroupID.x;
12     uint thread_id = gl_LocalInvocationID.x;
13     // We could also have a local_size_x of 1 and fill this array manually.
14     payload.meshlet_index[thread_id] = 2 * meshlet_index + thread_id;
15     EmitMeshTasksEXT(2, 1, 1);
16 }
```

GLSL

Minimal Mesh Shader Demonstrates meshlets, task group shared payload data, multiple mesh invocations per group, and setting output layout with triangle primitives.

The meshlet uses a compressed format (§3.5.3), such that each meshlet contains a list of all unique vertex indices addressed by it, and a triangle list of 8 bit indices into the unique vertex index list. This saves memory if a vertex is referenced more than once, the common case.

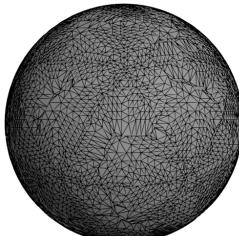
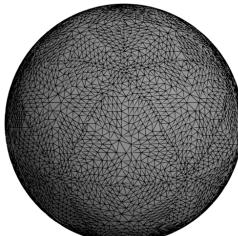
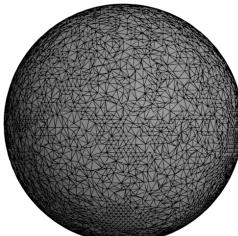
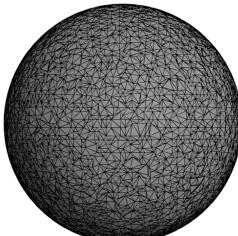
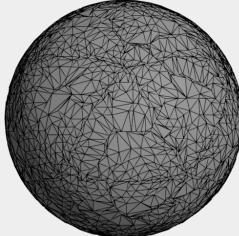
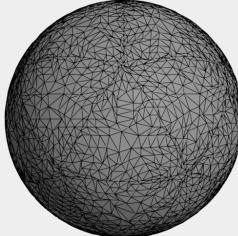
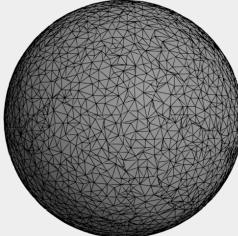
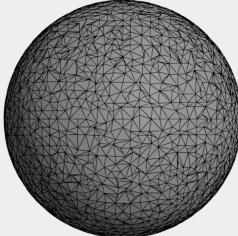
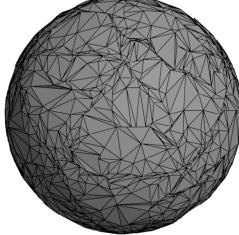
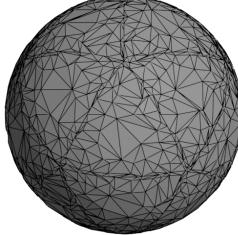
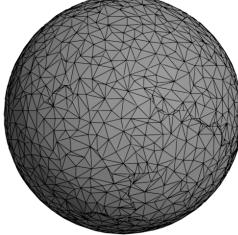
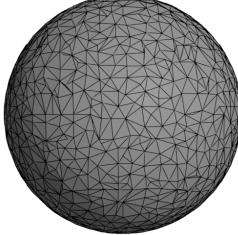
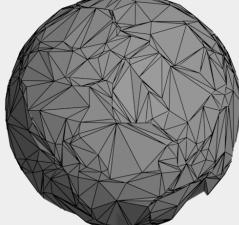
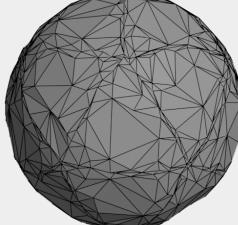
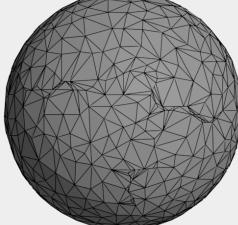
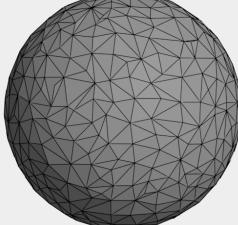
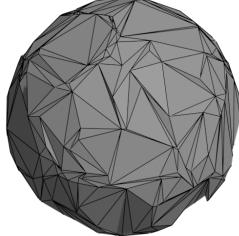
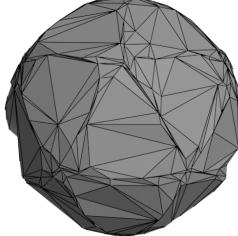
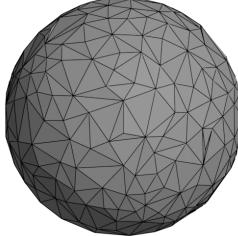
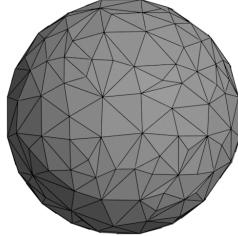
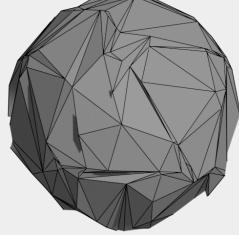
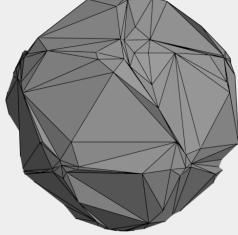
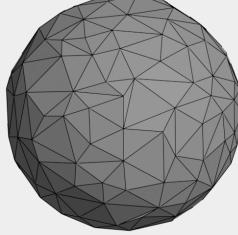
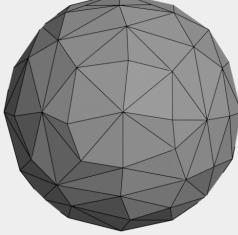
```
1 #version 450
2 #extension GL_EXT_mesh_shader: require
3 #extension GL_ARB_separate_shader_objects: enable
4 layout (binding = 0) uniform UniformBufferObject {
5     mat4 model;
6     mat4 view_proj;
7 } ubo;
8 // A meshlet is a 'unit' of a mesh sized to be drawn in complete parallel.
9 struct Meshlet {
10     uint vertices[64];
11     uint8_t indices[126 * 3]; // 126 triangles
```

GLSL

```

12     uint vertex_count;
13     uint index_count;
14 };
15 layout (std430, binding = 4) readonly buffer InputBufferV {
16     vec4 verts[];
17 };
18 layout (std430, binding = 3) readonly buffer InputBufferM {
19     Meshlet meshlets[];
20 };
21 layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
22 // (Nvidia recommended max sizes)
23 layout (triangles, max_vertices = 64, max_primitives = 126) out;
24 // We can output data other then the builtins in gl_MeshVerticesEXT
25 layout (location = 0) out Interpolants {
26     vec3 fragColor;
27 } OUT[];
28 struct MeshTaskPayload {
29     uint meshlet_index[2];
30 };
31 // Payload data from task shader telling us which meshlet to draw
32 taskPayloadSharedEXT MeshTaskPayload payload;
33 void main() {
34     uint thread_id = gl_WorkGroupID.x;
35     uint meshlet_index = payload.meshlet_index[thread_id];
36
37     s_meshlet meshlet = meshlets[meshlet_index];
38
39     SetMeshOutputsEXT(meshlet.vertex_count, meshlet.index_count / 3);
40
41     for (uint i = 0; i < meshlet.vertex_count; i++) {
42         uint vert_idx = meshlet.vertices[i];
43         gl_MeshVerticesEXT[i].gl_Position =
44             ubo.view_proj *
45             ubo.model *
46             vec4(verts[vert_idx].xyz, 1);
47         OUT[i].fragColor = abs(verts[vert_idx].xyz);
48     }
49
50     for (uint i = 0; i < meshlet.index_count / 3; i++) {
51         gl_PrimitiveTriangleIndicesEXT[i] = uvec3(
52             meshlet.indices[i * 3 + 0],
53             meshlet.indices[i * 3 + 1],
54             meshlet.indices[i * 3 + 2]
55         );
56     }
57 }
```

B Mesh Simplification Graphics

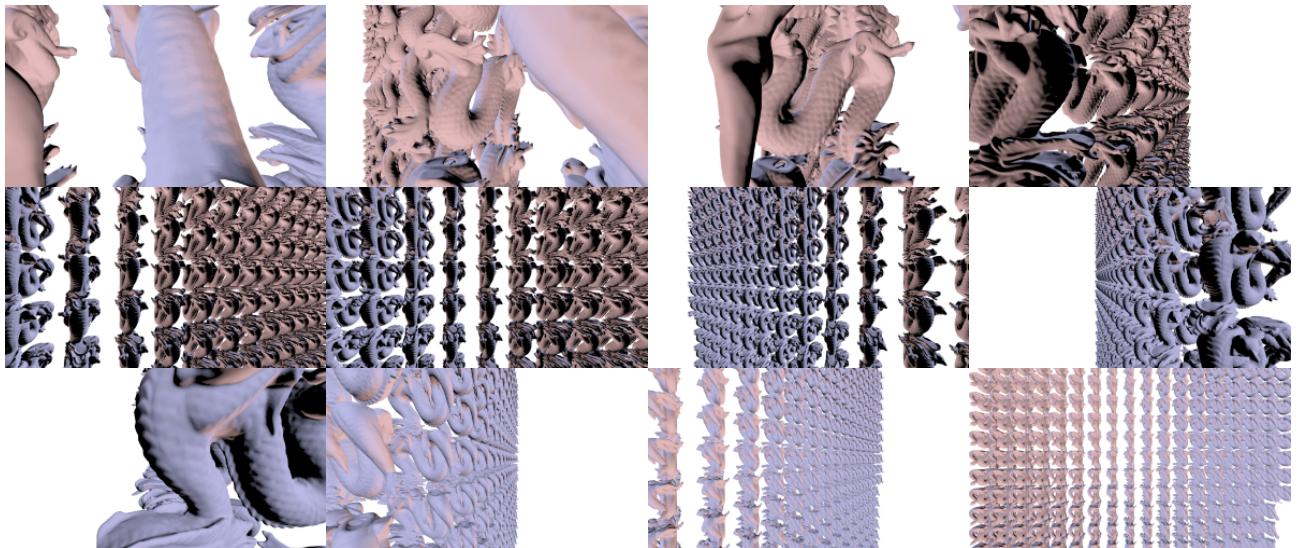
Level	Quadric DAG	Quadric LOD	<code>meshopt</code> DAG	<code>meshopt</code> LOD
1				
2				
3				
4				
5				
6				

C Benchmarking Scenes

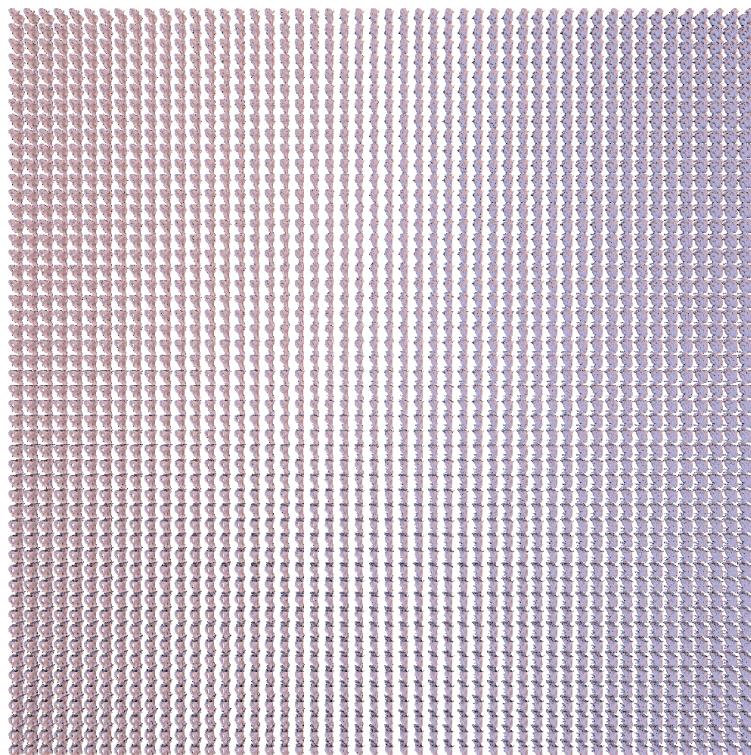
I made two benchmarking scenes, the first representing an ideal case for LOD chains for fair performance comparison, and the second representing the worst case for LOD chains to demonstrate their failings.

The best-case scene was a grid of 2500 1M triangle Stanford Dragons. For the visual error test, we rotated the camera around the center to capture various views, and so find error from all angles, while the rasterisation and performance tests simply zoomed the camera out for smooth performance curves.

C.1 Visual Error



C.2 Performance



D Licensing

```
console, dispatch, dlib, float-cmp, icrate, image, indicatif, inflate, inflections, is-docker, is-terminal, is-wsl, libmimalloc-sys, libredox, malloc_buf, memoffset, mimalloc, mint, nom, number_prefix, objc, objc-sys, objc2, objc2-encode, open, orbclient, quick-xml, redox_syscall, redox_syscall, redox_syscall, scoped_threadpool, sctk-adwaita, serde-binary, simd-adler32, slab, smithay-client-toolkit, strict-num, strsim, tiff, tracing, tracing-core, urlencoding, virtue, wayland-backend, wayland-client, wayland-csd-frame, wayland-cursor, wayland-protocols, wayland-protocols-plasma, wayland-protocols-wlr, wayland-scanner, wayland-sys, which, winnow, winnow, x11-dl, xcursor, xkbcommon-dl  
MIT OR Unlicense (7): aho-corasick, byteorder, memchr, same-file, termcolor, walkdir, winapi-util  
Zlib (1): adler32
```

E Project Proposal

Part II Computer Science Project Proposal

Multiresolution Meshes: Rendering 1 Billion Triangles

Project Originator: Maxwell Pettett

Project Supervisor: Dr Rafał Mantiuk

Director of Studies: Dr John Fawcett

Project Checkers: Dr Weiwei Sun and Prof Neil Lawrence

1. Introduction

This project will look at a graph-based multiresolution system for rendering large meshes, which uses pre-generated data for a mesh to efficiently query the most appropriate approximation of the shape to create a view-dependent triangulation that when compared with the next step of increased detail, is close to imperceptible to a user. This cut-off point is commonly set at 1 triangle per pixel in modern systems, giving a major advantage in the cost of rendering an object being only proportional in cost to the size of the screen – sublinear scaling with the complexity of the mesh.

A multiresolution data structure stores a single structure at multiple levels of resolution, where resolution is defined as accuracy to the shape we are trying to represent. Modern tools such as photogrammetry mean incredibly high resolution meshes are now commonplace, each containing upwards of 100s of millions of triangles to render, and so needs to be further approximated to display efficiently. This technique has been most prominently used in Unreal Engine 5's Nanite¹⁴, although the tech is closely coupled with the rest of the engine and cannot be easily integrated into other existing tools.

To store a multiresolution, we must divide our mesh into a system of fragments of varying resolutions, which, when a valid combination is selected, form an approximation of the original in varying resolutions in different regions. The relations between these fragments are stored in a directed acyclic graph¹⁵, a structure appearing as a tree where nodes may have multiple parents, representing which fragments can be further divided into child fragments. This structure allows an approximation of the original to be easily queried, as any cut of the DAG has fragments that piece together a mesh with the same topology as the original.

A fragment then can be simply defined as a collection of adjacent triangles, and fragments recursively generated by remeshing triangles across multiple other fragments. Special care must be taken to ensure the seams between these fragments are imperceptible, and that the data structure permits completely parallel querying for use in a GPU driven render pipeline.

Once a cut of the DAG is selected, another GPU shader can generate an index buffer for the given mesh and fill a further buffer with index information required for completely indirect

¹⁴https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

¹⁵<http://publications.crs4.it/pubdocs/2005/CGGMP05a/ieeviz2005-gpumt.pdf>

rendering, meaning no expensive sync points with the CPU for rendering - the GPU can simply execute its command buffer. This means all this project requires is simple GPU compute capabilities and no complex optional features, so the core should not require any special hardware.

This project only touches the raw triangle rendering, and not the material rendering stages after this - we may want to render each mesh individually with a different shader, but more commonly GPU driven compute combines all instances of a mesh into a single draw call with scene information and applies material/shader effects in a deferred pass.

2. Structure

The project will consist of two main modules: An executable to convert a mesh into the graph-based format, known as the ‘Baker’ and an executable to view a given multiresolution graph to test and evaluate, known as the ‘Renderer’. Glueing these together will be a common binary format, saved by the baker and loaded by the renderer.

Baker:

- Given a mesh in a standard Kronos .gltf format, recursively break mesh into clusters and remesh them, creating the multitriangulation DAG, and save to our common binary format.

Renderer:

- Given a mesh in our common binary format, select a cut of the DAG using triangle density heuristics and render it to screen with appropriate debugging/evaluation features, such as frametime indicators and triangles per frame counters.

Evaluation:

- Details in the following section.

3. Evaluation

Quantitative

- Frame image comparison with full resolution mesh rendering, and frame time indicators.
- Measure of extra memory use in the graphics memory used by the multiresolution technique to perform the speedup.
- Compare the execution time of the multiresolution generator with existing LOD chain generators, to evaluate the content creation workflow, at several levels of fixed mesh resolution.
- Image metrics in the first order comparing the final image resulting from rendering with the multiresolution compared to the original mesh and LOD chains.

4. Starting Point

I have prior experience in GPU shader programming and Rust, having developed some small programs based around GPU driven rendering, mainly with mesh culling. No prior code has been written for this project.

5. Success Criteria

For this project to have been a success, I will have achieved:

- Improved frame time: A faster frame time for multiresolution rendering compared to full resolution mesh rendering, for a suitably high resolution base mesh.
- Improved rasterisation: With low overdraw, the number of triangles drawn by the multiresolution renderer can be shown to be proportional to the resolution of the screen.

6. Possible extensions

1. Add support for foveated rendering to the renderer when deriving the cut heuristic, to only render high resolution data where the user is looking.
2. Add support to statically bake LOD chains using the multiresolution graph, to better aid comparisons.
3. Stream graph data for a mesh from main memory to the graphics card, using a fallback to lower resolution data when requested content is unavailable
4. Cull clusters based on their associated bounds after selection, for a further reduced frame time.
5. Use temporal information when selecting the cut for improved performance - when a given cluster is selected in one frame, there is a high likelihood it is selected in the subsequent frame, so use previous data as a starting point.
6. Use perceptual metrics to more precisely evaluate the differences between low resolution remeshes and the original shape, to better evaluate the baker.

7. Plan of Work Packages

Week 1 to 2: 16/10 - 29/10

- Submit this document
- Baker
 - Research into mesh cluster generation techniques or libraries.
 - Initial format for serialising and de-serialising data.
- Renderer
 - Basic mesh renderer program, capable of taking an index buffer and vertex buffer and drawing it to screen.
 - Ability to visualise abstract mesh clusters and triangles within them.

3 to 4: 30/10 - 12/11

- Baker - Successful prototype splitting a mesh into triangle clusters.
- Baker - Research into methods for remeshing within clusters for the multiresolution.
- Renderer - Implementation of a UI library.

5 to 6: 13/11 - 26/11

- Baker - Algorithm to reduce mesh complexity within a cluster.
- Slack - General slack time for previous weeks, and for increased workload due to AOS being moved to Michaelmas.

7 to 8: 27/11 - 10/12

- End of Michaelmas term.
- Baker - initial graph-based multiresolution export.
- Diss - Begin writing an introduction based on research done for previous parts.

9 to 10: 11/12 - 24/12

- Baker - Recursive clustering and reduction to make classical LOD chains, exporting back to .gltf, verifying reduction algorithm success.
- Renderer - initial viewing of different levels of the multiresolution.
- Slack - /time off for Christmas.

11 to 12: 25/12 - 07/01

- Baker: Optimise clusters to store triangle strips instead of triangle lists to reduce costs in handling index copying.
- Renderer: Tools for evaluating performance statistics.

13 to 14: 08/01 - 21/01

- Finish core project, passing or close to passing all success criteria

15 to 16: 22/01 - 04/02

- Diss - Evaluation draft.
- Begin progress report.
- Ext/Slack: Foveated rendering.

17 to 18: 05/02 - 18/02

- Ext/Slack: Investigate cluster based culling.

19 to 20: 19/02 - 03/03

- Diss - Conclusion draft
- Ext/Slack: Temporal optimisations extension.

21 to 22: 04/03 - 17/03

- Continue work on extensions (4 weeks total time).
- Work on dissertation, adding any extra details from extensions.
- End of lent term - User test given core system at full usage, including use of eye tracker if extensions allow.

23 to 24: 18/03 - 31/03

- Submit first draft dissertation to supervisor and DOS for feedback.

25 to 26: 01/04 - 14/04

- Act on any feedback, pass back to supervisor for final review.
- Code cleanup and review.

27 to 28: 15/04 - 28/04

- Submit dissertation, planning roughly 2 weeks before deadline
- Remainder is slack.

8. Resource Declaration

I will be using the following devices for development:

- Main PC: AMD R5 3600 / 16GB / GTX 1660 (Windows).
- Backup device: Steam Deck (Linux).
- Backup data: local continuous backup on NAS, regular git commit backups to Github.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.