Multiresolution Mesh Rendering Engine Practicalities and Performance

Maxwell W. Pettett*
Supervised by: Rafał K. Mantiuk†

Department of Computer Science and Technology
University of Cambridge
United Kingdom

Abstract

A multiresolution mesh is a structure that allows multiple levels of resolution of a mesh to be sampled in different regions. They are used to accelerate the construction of view-dependent Levels of Detail (LODs) for real-time rendering, generally for complex objects that may span large depths (e.g. terrain). Nanite, introduced in Unreal Engine 5, is an example of a full multiresolution pipeline. We describe our mesh-shader based multiresolution rendering engine in Vulkan, with two implementations to extract view dependent LODs. The first implementation is based on the approach established by Nanite. Our alternative implementation has no intermediate buffers at the cost of less fine-grained control over regions of the multiresolution we explore. We finally evaluate the two methods against each other and traditional LOD chains, emphasising practicality and performance.

Keywords: Modeling and Geometry processing, Real-time Graphics, Rendering

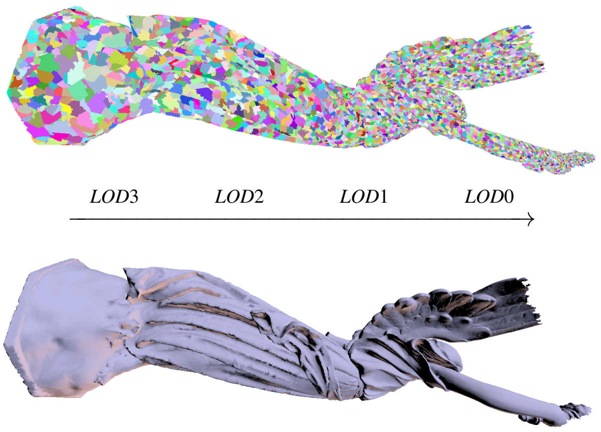

Figure 1: View-dependent LOD generated from the Stanford Lucy model (28 million source polygons) [11]. The statue's base is visibly lower quality than the top.

1 Introduction

A common desire for higher-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. Varying mesh resolution is typically used in real-time rendering engines to maintain performance in complex scenes. This is traditionally implemented with a series of coarser and coarser approximations of the mesh, a Level of Detail (LOD) chain. However, LOD chains are limited in flexibility. Each object can only be rendered at a single resolution, despite the possibility that the same object spans large depths (e.g., terrain), and, therefore, there is no single optimal LOD.

A *multiresolution mesh* is a data structure that stores geometry information at multiple levels of resolution. It is an alternative to, and generalisation of, LOD chains, with fine-grained control over rendering that can tackle their disadvantages. However, fidelity improvement may

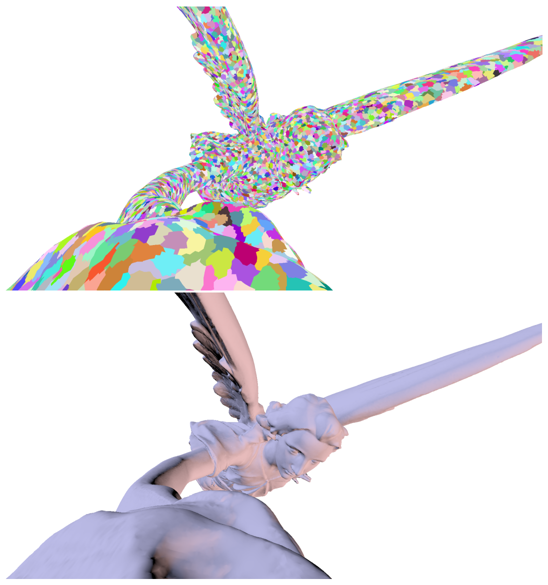

Figure 2: Figure 1's view from the camera. Note that cluster sizes are mostly uniform, excepting those close to the camera which have reached maximum resolution.

*mp2015@cam.ac.uk
†rafal.mantiuk@cl.cam.ac.uk

be deemed too expensive if the cost to calculate the view-dependent LOD is higher than that of rasterisation at a higher mesh resolution, so methods to render multiresolution meshes must be fast and scalable. Our main point of comparison as a modern multiresolution pipeline is Unreal Engine 5's Nanite, which is embedded in the engine and so difficult to study or extract.

This work leverages the introduction of mesh shaders on modern hardware, which operate by emitting small clusters of triangles rather than vertices. This paper opens with background on multiresolutions, the error functions required to generate view-dependent LODs, and mesh shaders. The implementation section then describes two methods for rendering view-dependent LODs. Our first implementation, *DAG Explore*, is based on ideas Nanite's *persistent threads*[7]. In contrast, the second implementation, *Task Select*, is developed in this work. It relies on mesh shading to insert LOD logic into our draw calls, without a compute pass or intermediate data. We then evaluate their performance and practicality.

2 Background

Many forms of multiresolution exist with different characteristics and drawbacks. Hoppe introduced **progressive meshes** [6] in 1996. A progressive mesh is a multiresolution mesh encoded as a low-resolution base mesh, and the vertex splits required to raise resolution. Quick-VDR [12] expanded on progressive meshes with an initial coarse-grained selection before vertex local transformations.

Further techniques, such as BDAM [2], or Adaptive Tetrapuzzles [3], focus more on the coarse-grained selection, using spatially based partitions for 2D and 3D surfaces, respectively. Their partitions contain geometry in patches that can be substituted, moving further from vertex transformation and decimation techniques. Batched Multi Triangulation [1] extends geometry patches to a generic framework for multiresolutions based on a Directed Acyclic Graph (DAG) of patches, the approach our renderer will be based on. Ponchio's thesis is an excellent comparison of the above methods [10].

2.1 The Multiresolution DAG

This section introduces the multiresolution mesh as a DAG of **clusters**, uniformly sized patches of triangles, described in detail in [1]. Figure 1 shows clusters selected from a multiresolution of a high-resolution mesh. A requirement of a multiresolution scheme is to ensure that clusters of neighbouring resolution levels can be interleaved without seams introduced by mesh simplification. To illustrate the difficulty of this problem, let us consider a simple scheme:

1. Start from a set of clusters that partition a mesh.
2. Recursively, merge pairs of clusters together and simplify their contents. Edges on the boundary of the pairs are *locked*, so are not moved by the simplifier.

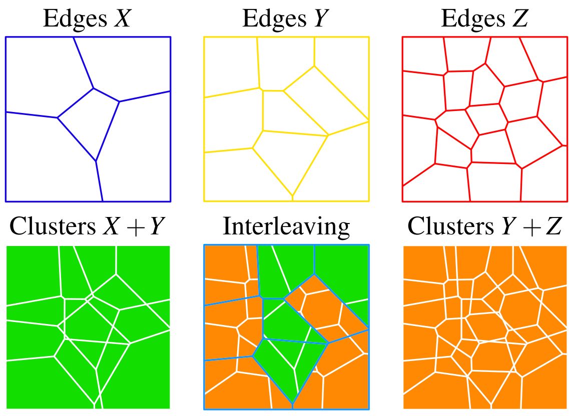

Figure 3: Example of locked edges forming clusters allowing for interleaving. 3 sets of locked edges (X,Y,Z) are merged into 2 sets of clusters $(X+Y, Y+Z)$, which can be interleaved using their shared locked edges (Y).

This scheme would form a tree of variable resolution clusters of the mesh. However, in doing this it will lock some edges from the highest resolution to the lowest, restricting the flexibility of mesh simplification. At the extreme, it will bisect the mesh with a high-resolution ring of edges, harming the quality of lower resolution clusters. To avoid this artifact, we need an alternate method that allows edges to be *unlocked*.

The multiresolution mesh scheme we use contains two sets of locked edges at each level of detail, one set for compatibility with each of the lower and upper levels. Figure 3 shows two adjacent levels and a selection of clusters from both, made possible due to their shared set of locked edges.

Such selections can only be made if we can guarantee they will approximate the original mesh, so containing no overlaps or holes. We use a DAG to encode relations between clusters to allow us to make confident selections. Nodes in the DAG represent clusters in the multiresolution, from all levels. Edges represent dependence between clusters, a relation of mutual exclusion, i.e. overlap. This property is transitive, therefore we only include dependence between adjacent levels on our DAG [1].

Our method of generating DAGs follows the Nanite *Cluster - Group - Simplify - Recluster* scheme [7]:

1. Start from a set of clusters that partition a mesh.
2. Partition clusters into **groups**, collections of around 4 adjacent clusters.
3. Simplify within groups, locking border edges.
4. Partition each group into *two* new clusters, which become the *two* parents of the group. Return to 2.

This is a generalisation of the previous method, replacing a one-parent-two-children relationship with two-parents-four-children. A DAG for a small mesh generated with our program is shown in Figure 4, in which these structures can be identified. Smaller clusters are more flex-

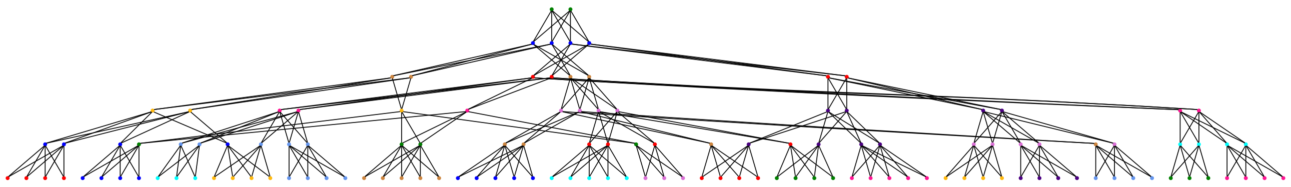
Figure 4: A DAG for a multiresolution encoding of a simple sphere mesh. Nodes represent clusters, edges represent dependencies between clusters. Nodes of the same colour share the same two parents, and so make up a group.

ible, but maintain a worse area to perimeter ratio, resulting in more locked edges at each stage and less efficient simplification; we choose to use clusters of around 300 triangles. There is a chance the new parent clusters do not overlap with all child clusters, resulting in some false dependencies in the DAG. However, parents sharing identical sets of children allows for efficient DAG traversal.

A valid selection of clusters will cover the area of the mesh, with no clusters overlapping. Ensuring the selection contains no overlapping clusters requires that no two clusters in the selection are dependent on each other. To cover the full mesh area, every path from a root to a leaf must contain a single selected node. Our DAG structure guarantees this selection will contain no seams [1].

A **dicut** is a cut into two subsets such that any cut edges connecting the two subsets share the same direction. If we select the leaf nodes of a dicut subset that includes the root, we have a valid set of clusters. This is due to two factors. First, the selected clusters will not contain overlapping geometry, as they satisfy all dependency relations. Second, we have no holes, as our selected clusters descend from the (two) roots. The root clusters cover the entire mesh area, so the sum of all descended clusters (satisfying dependency relations) area must also cover it [1].

The easiest DAG to imagine is a traditional LOD chain. Each layer's node is dependent on the next, as they overlap in area (the entire mesh), and we sample by selecting a leaf of a dicut set (any single node).

2.2 View-dependent LOD from a DAG

To generate a view-dependent LOD, it is useful to define an error function on clusters that allows us to estimate their **screen-space errors**. We then can compare this error to a user-defined threshold, , that defines the target mesh resolution. This screen-space error is projected from an object-space error δ of a cluster. The exact definition of the object-space error varies per-implementation, but we will use the average edge length of a cluster, similar to batched multi-triangulation [1]. This represents triangle density within a cluster, and is comparable between clusters as their triangle counts are roughly constant.

To convert error from object-space to screen-space, we assign each cluster a bounding sphere, with centre c and radius r, a spherical volume in object-space that bounds the cluster. We then use a method similar to [2] to project the object-space error of some cluster i to screen-space, $\mathrm{err}(i)$, for eye position e.

$$\mathrm{err}(i) = \frac{(\delta_i + r_i)^2}{||c_i - e||^2}, \tag{1}$$

An important feature of the error function is that it is monotonically decreasing down the DAG.[1] To ensure the screen-space area of clusters is monotonic, we assign each cluster's bound such that it contains all bounds of their children, turning the DAG into a nested boundings volume hierarchy [1]. Object-space error, clusters' average edge length, also monotonically decreases as clusters double their triangle density at each level.

2.3 Mesh Shaders

This paper references **mesh shaders**, a concept shared between modern graphics APIs that readers may not be familiar with. We will use the Vulkan implementation and terminology. Mesh shaders attempt to solve some shortcomings of using the traditional graphics pipeline for procedural geometry. The traditional pipeline includes tessellation, geometry, and vertex stages, that can be used for procedural geometry, but each with a limited view and control of parts of the source data.

The common way to program procedural geometry has shifted away from the graphics pipeline with the wide adoption of compute shaders, due to their flexibility and good support. Mesh shaders attempt to bring this flexibility to the graphics pipeline by stripping out everything other than the fragment stage of the pipeline, and adding a **mesh stage**. The mesh stage has all the semantics and capabilities of a compute shader, with the additional ability to emit triangles, up to a maximum primitive limit per workgroup [8]. These will output our clusters, and save writing to an intermediate index buffer.

Additionally, a similar **task stage**[2] is added, which, instead of emitting triangles, can emit mesh shaders. We will utilise this to insert LOD logic directly into the draw call. This grants us a large amount of flexibility for programming procedural geometry, although it is not as powerful as the ability to generically launch threads on the GPU.[3]

[1]Root nodes have the highest error, as they have the least polygons.
[2]Known as the Amplification stage in DX12.
[3]See **Work Graphs** [9], that will allow generic kernel invocation.

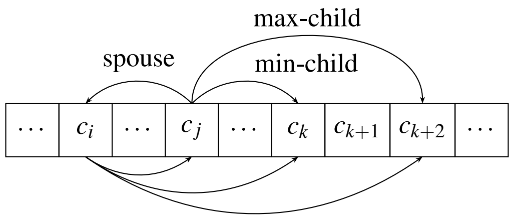

Figure 5: A diagram of our cluster structure. Pointers within the diagram from cluster c_j correspond to a cluster $c_j = [\text{spouse} = i,\ \text{min-child} = k,\ \text{max-child} = k+2]$.

3 Implementation

This section will look at the two approaches described and examined in this paper. Older view methods sample their multiresolutions on a separate CPU core, referred to as out of core [2, 1], however compute and task shaders allow us to do this work efficiently on a GPU.

Both implementations to render a view dependant LOD of a mesh are supplied with:

- A *cluster buffer*, containing the DAG of clusters that make up the multiresolution (Fig. 5).
- Instance information, containing the model matrix of the instance of a multiresolution we are drawing.
- Camera information, the view-projection matrix.
- A screen-space error target, τ, the maximum screen-space error a cluster can have to be drawn.

The first approach, *DAG Explore* (§3.1), aims to output all clusters that should be drawn into a buffer, searching the DAG for suitable clusters recursively from the root. DAG Explore is similar to Nanite's Persistent Threads implementation for generating a view dependent resolution.

The second approach, *Task Select* (§3.2), aims to use the programmable task stage of the mesh pipeline (§2.3) to eliminate the need for intermediate memory.

3.1 DAG Explore LOD Generation

A typical instance of a multiresolution in a scene will have most of its area filled with lower resolution clusters. Selecting a low resolution cluster will instantly invalidate the many higher resolution clusters that descend from it. Testing these clusters would be wasted time, so we want to avoid exploring the entire DAG. This method will traverse the DAG recursively, starting from the roots. As we are searching for leafs of a dicut subset, this must encounter all clusters that should be drawn.

Traversing the DAG requires care. It is not a tree, so there are clusters that share children, but to traverse the DAG efficiently we should not explore the same cluster twice. Our DAG is, however, shaped similarly to a tree; it is formed of pairs of clusters that share identical children; we say clusters in such a pair are **spouses**. We can then

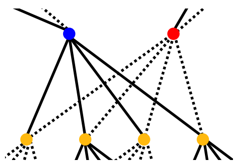

Figure 6: A group of clusters whose parents are in separate groups. Solid lines represent the edges traversed to explore the DAG as a tree.

view the DAG as a tree by only regarding the children of one cluster in each of these pairs, illustrated in Figure 6. In Alg. 1, we only explore the children of the spouse with the smaller index.

Algorithm 1 DAG Explore, breadth first search

```
queue ← root-nodes
draw-buffer ← []
draw-count, head ← 0
tail ← |root-nodes|
**while** queue not empty **do**
    $i$ ← queue[head]
    head ← head + 1
    cluster$_i$ ← clusters[$i$]
    **if** err($i$) $< \tau$ **then**                ▷ Cluster should be drawn
        draw-buffer[draw-count] ← *cluster*
        draw-count ← draw-count + 1
    **else if** $i <$ cluster$_i$ → spouse **then**      ▷ DAG as tree
        **for** $c$ **in** cluster$_i$ → children **do**
            queue[tail] ← $c$
            tail ← tail + 1
        **end for**
    **end if**
**end while**
```

3.1.1 Multiqueue

Algorithm 1 does not appear GPU-friendly, as it leverages a single shared queue. We must allow multiple invocations synchronised access to the queue to maintain parallelism. Atomic buffer operations are too slow for this use case.

Our solution relies on subgroup[4] arithmetic (introduced in Vulkan 1.1) to synchronise queue access. In Vulkan terminology, a subgroup is a set of invocations executing in lockstep on the GPU. Subgroups will always be part of the same workgroup, a set of invocations with shared memory, but a workgroup may maintain multiple subgroups. Invocations in a subgroup may be active or inactive depending on factors such as dynamic branching, and an inactive invocation will not contribute to subgroup arithmetic results.

Subgroup arithmetic allows invocations to communicate via reductive operations, with each invocation submitting data. The most straightforward subgroup operation we use is `subgroupAdd(1)`, which will return the

[4]the Vulkan term; in AMD these are waves, in Nvidia, warps.

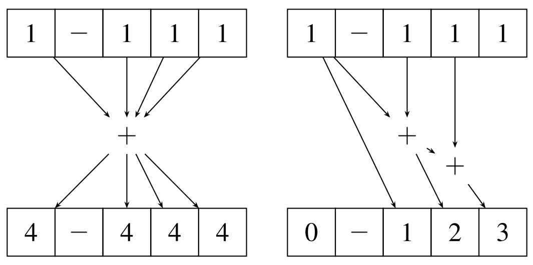

Figure 7: The subgroup operations `subgroupAdd(1)` (left) and `subgroupExclusiveAdd(1)` (right). Each block is a invocation within a single subgroup, with reduction operations linking inputs to results.

number of active invocations, as shown in Fig. 7 (left).

To implement DAG Explore, we limit the size of our workgroups to ensure they only contain one subgroup, which is generally 32 or 64 invocations, ensuring we can rely on subgroup arithmetic. This is the main source of difference from Nanite; persistent threads share work between workgroups, which relies on undefined GPU forward progression scheduling behaviour [7], while ours is limited to a single workgroup per instance. This will limit our latency of rendering a scene to processing the largest mesh, however, we assume scenes are highly populated and so parallelizable.

Algorithm 2 Synchronised Queue Pop

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

Algorithm 2 uses subgroup arithmetic to synchronize invocations each taking a cluster from the queue. To pop a unique item for each invocation, we can offset the queue head pointer by each of their **local invocation IDs**, their indices starting from 0 within the workgroup. This, however, leaves us with conflicting information about the true head of the queue across invocations. The queue may contain a number of clusters fewer than the size of our subgroup, which will result in some number of invocations being inactive. We solve this by incrementing the head pointer by the number of active invocations using subgroup arithmetic, ensuring the data is synchronized.

The more complex operation is appending children to the queue. We do not know how many children each node has, so we cannot simply offset by our local invocation ID when pushing. We know the number of items each invocation will add onto the queue, so we can allocate blocks in the queue upfront. To allocate blocks, we can use a more advanced subgroup command, `subgroupExclusiveAdd` (see Fig. 7 (right)), which will perform an exclusive addition across all active threads *in one call*. The return value for this will be an allocated index for each invocation to write to, used in Algorithm 3. We synchronize the queue afterwards by taking `subgroupMax(tail)`.

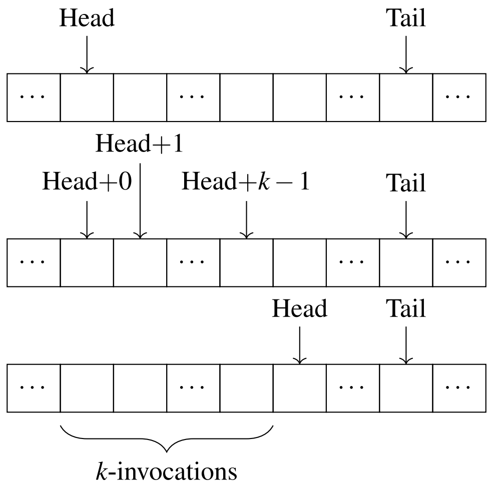

Figure 8: The positions of head and tail indices through Algorithm 2. After adding idx, each head points to a unique cell, then adding `subgroupAdd(1)` (= k), the head returns to being synced.

Algorithm 3 Synchronized Queue Push with Subgroup Arithmetic

```
cluster_t c = clusters[i];
int children = c.max_child_index
               - c.min_child_index + 1;
tail += subgroupExclusiveAdd(children);
for (int child_i = c.min_child_index;
         child_i <= c.max_child_index;
         child_i++) {
    queue[tail] = child_i;
    tail += 1;
}
tail = subgroupMax(tail);
```

3.1.2 Emitting clusters

Clusters with sufficiently low error must be emitted to be drawn. We push them to a *draw buffer* similarly to Alg. 3, but with a maximum of 1 cluster pushed. In our implementation, the draw lists for DAG Explore has enough space for the worst case (full resolution) of every instance. As future work, this size could likely be optimised, as we do not expect the worst-case full draw for every instance.

A task shader will then then emit mesh shaders for each cluster in the draw buffer. The number of clusters DAG Explore has emitted at this point is only recorded on the GPU, but must be communicated to our draw call to render, as invoking a task for each item in the buffer when many are empty would be wasteful. Indirect Dispatch is a common technique to allow some parameters of commands to be supplied by the contents of a buffer in the GPU, which mesh shaders support. This saves possible wasted bandwidth and latency sending the same counter back and forth from the GPU. After filling our draw buffer, we set the *group count* parameter of `DrawMeshTasks` in our indirect buffer to the exact number of clusters to draw.

Extending this to drawing many instances is not complex. Using atomics, we can assign each instance's emitted clusters a space in a shared draw buffer. Each instance's draw call can then be instructed to read clusters from that range. `DrawMeshTasks` also supports the **multidraw** extension, which allows us to store the indirect arguments for multiple draw calls in a single buffer. A single buffer for draw data means we can additionally invoke the compute stages of all instances in a scene in a single command.

3.2 Task Select LOD Generation

DAG Explore does a lot of work and requires shared memory in picking which clusters to draw, queuing no further clusters once the boundary of the cut has been found. We present an alternative method that does not require shared memory and is simpler to implement.

The task shader allows us to integrate computations within the graphics pipeline. Integrated computation allows us to select view-dependent LOD without intermediate buffers, a method we will call *Task Select* LOD. This method invokes a task invocation on the GPU for every cluster and emits geometry to draw if the cluster has an error below the threshold, but parents with errors too great, with additional care to ensure the result has no holes.

3.2.1 Local Cut Selection

DAG Explore explores the DAG recursively, aiming to find the cut, made up of clusters whose error is just low enough to pass the threshold value τ. However, to enable the most parallelism, it is preferable to be able to test if a cluster should be included in the cut based on only itself and local neighbours. This is possible as our DAG has a single unique cut, as the screen-space error we compare decreases monotonically through clusters.

Algorithm 4 Local Cut

```
    parent-err ← $\min(\text{err}(c_i \to \text{parent}_0),\ \text{err}(c_i \to \text{parent}_1))$
    this-err ← $\min(\text{err}(i),\ \text{err}(c_i \to \text{spouse}))$
    draw ← $(\text{this-err} \le \tau) \wedge (\text{parent-err} > \tau)$
```

Each cluster's task invocation must make the decision of what fills its group's area; the group, or the two parents (if either), and draw the cluster if appropriate. 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 the group as a whole, similarly to the bounding volume hierarchy of [2].

Algorithm 4 determines if a cluster is on the edge of the cut, and so should be drawn, based on the relation of its parent's errors to its own. The two parents are likely members of different groups, so likely have differing screen-space error. To compensate, Alg. 4 takes the minimum of the two. Comparing this against the error of this cluster would then leave a hole if $err(c_i \to \text{parent}_0) > \tau > err(c_i \to \text{parent}_1)$, as only one of the two parents would be drawn. To resolve this, Alg. 4 takes *this-error* to the minimum of the cluster's own error and that of the spouse. This fills the hole described above, as the previously missing parent will now draw based on its spouse's lower error.

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

3.2.2 Task Shader Indirect Dispatch

A major issue with this approach would be wasted work in clusters that are too high resolution to be drawn. Such high-resolution clusters will make up the majority of most instances. For example, drawing a mesh at the first level of simplification in DAG Explore will, on average, only check half of a multiresolution's clusters, as the source mesh represents 50% of total triangles.

This method uses the same indirect dispatch draw call as §3.1.2. Task shaders can write to buffers just as compute shaders can, so, if we bind our indirect arguments buffer to the task shader, we allow ourselves to alter the number of tasks we invoke on the next dispatch.

We arrange our cluster buffer such that lower indices represent clusters at lower resolutions, meaning dispatching fewer tasks than there are clusters will cap the maximum resolution view that can be selected. Because we are able to control our dispatch count inside the shader, we can then cap this resolution dynamically depending on the current view of the instance. It is clear then that dispatching tasks for indices above the maximum that is selected is futile, so this maximum index is the value we wish to estimate, and set the indirect dispatch count to.

We say the *maximum requested index* for a cluster, based on the error values calculated in Alg. 4, is:

$$\text{max-idx}(i) = \begin{cases} \text{max-parent}(i) & \text{if parent-err} < \tau \\ \text{max-child}(i) & \text{if this-err} > \tau \\ i & \text{else} \end{cases} \tag{2}$$

Intuitively, bringing an instance closer to view yields a greater error for clusters, which may require replacing a cluster with its children, so we increase max-idx and the tasks invoked for the instance. Inversely, moving an instance away from view will reduce its tasks invoked. A cluster only views local data, so does not request drawing clusters beyond the scope of its parents or children, so we set our next dispatch count to the maximum requested indices of all clusters. This brings with it a single frame of latency to apply the requested index if it increases, so we need some small additional logic when selecting clusters in case the resolution we wish to draw at is not available. Simply, if our cluster has too high an error to draw, but our children are out of range of current workgroups (and so are not being processed), we should draw ourselves anyway.

Once we have determined a cluster should be drawn, the task shader code is identical to the previous method; see §3.1.2.

3.3 Cluster Culling

An engine based around instances and LOD chains may utilize *instance culling* to save time in rasterisation. This can be improved; instance culling has some of the same flaws as LOD chains, being based on arbitrary-sized objects. If we instead focus on culling clusters, we end up doing work on much more uniformly sized items[5], which results in finer-grained culling [5]. This technique was used in industry before cluster based multiresolutions, as the GPU friendliness of clusters makes them ideal for GPU-driven rendering systems.

A simple culling technique we apply is frustum culling, not drawing a cluster if it is outside the camera frustum. The frustum can be represented by six planes, which we extract from the model-view-projection matrix as described by [4]. These planes will then exist 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.

This requires testing every cluster that may be drawn; DAG explore can be optimised further. The DAG is a nested bounding hierarchy, so a bound of a cluster contains the bounds of all children. A successful cull check on a cluster's bound would then rule out the entire hierarchy of clusters descending from it. DAG Explore can then stop exploration early if a cluster can be culled, as we then know no child may be rendered, culling as early as possible. At the coarsest grain, a successful cull on the root cluster is equivalent to instance culling.

4 Evaluation

We evaluate on two benchmarks on a GTX 1660 and r5 3600. The first has almost optimal conditions for LOD chains (*LOD efficient*), and the next demonstrates their primary limitation (*LOD deficient*).

Our LOD efficient benchmark moves a camera back from the scene origin, revealing a large 2D grid of instances. The benchmark uses the Stanford Dragon model [11], which contains 1 million source triangles, meaning our scene of 1000 instances contains 1 billion triangles. These instances will occupy a narrow slice of depth on the screen, so are suitable for traditional LOD rendering. Frame times are plotted in Figure 9.

[5]Uniformly sized in screen-space if done after LOD generation, making this take time proportional to target error.

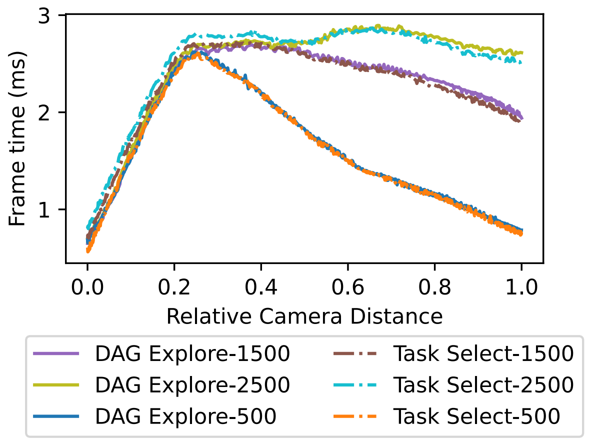

Figure 9: LOD efficient: Results combining benchmarks for 500 up to 2500 instances in the scene.

Instance Count	500	1000	1500	2000	2500
LOD Chain	2.58	3.49	3.85	4.01	4.10
Task Select	1.41	1.88	2.09	2.21	2.27
DAG Explore	1.39	1.89	2.13	2.27	2.37
% Change	1.42	-0.53	-1.93	-2.79	-4.40

Table 1: LOD efficient: Mean frame times (ms) for the different methods across instance counts. Our CPU based LOD chain uses an error function equivalent to the multiresolutions.

Task select achieves very similar performance across the board to DAG Explore while saving intermediate memory. For context, the full resolution 2000 instance scene takes a median of 472ms to render. The 2500 instance scene uses 38.82MB of GPU memory to store the draw buffer for DAG Explore, but just 29.31KB of intermediate memory for Task Select's indirect dispatch parameter buffer.

At relative camera distances of less than 0.4, the benchmark's screen is filled with instances, with varying amounts of culling. In these ranges, we can see in Figure 9 that task select is a bit slower per instance. This is due to its more fine-grained culling doing more work to cull an entire instance, while the variable-grained culler DAG Explore culls as early as possible while searching the DAG, and has an almost negligible slowdown per instance.

The cost per instance of a CPU cull check and draw

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

Table 2: LOD deficient: Frame time analysis of a single frame for the viewpoint from Fig. 2, Lucy, using NVIDIA Nsight. Fragment stages for both methods are identical.

call is clearly visible for our traditional LOD chain in Table 1, so we would expect improvements from a GPU implementation. In contrast, Table 2 shows a large efficiency gain from Task Select over the LOD chain that results from their limitations; the instance is both close and far from the camera, but the LOD chain renders at full resolution.

Our method optimises more effectively given more instances, as more instances give us more fine-grained control over the indirectly dispatched clusters. This is, however, slightly contrary to the original problem, the rendering of small numbers of massive meshes. We expect DAG explore to perform better in these cases. However, our method is still competitive due to its low reliance on memory bandwidth; massive multiresolutions would require more working space for the queue than is commonly available as workgroup shared memory.

5 Conclusions

Cluster-based rendering engines already give way to GPU-driven pipelines that excel at high-fidelity scenes. In such a pipeline, the practicality of multiresolutions is clear. They are generated automatically, sampled based on concrete metrics, and can be integrated into existing workflows.

The methods demonstrated in this paper are all limited by the rate of rasterisation, which is held roughly constant within a scene. This means multiresolutions are likely to fit into the frame budget of a high-fidelity renderer. This is a key goal of the method; a good error function should keep the screen-space triangle density roughly constant. In doing so, we grant complete flexibility on the source resolutions of any mesh in any scene, a major advantage for renderers targeting photorealism.

This renderer still relies on having enough VRAM to store a massive multiresolution, something that cannot be taken for granted within a large engine. As such, future work includes data streaming, which would load segments of the multiresolution into memory only on demand [7].

Nanite is a monolithic pipeline, making the methods used for generating and rendering multiresolutions hard to extract for general use. This paper has instead presented viable generic algorithms for rendering in this new paradigm. In future, we hope to see these help push multiresolutions as a standard tool in contemporary engines.

References

[1] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. *VIS 05. IEEE Visualization, 2005.*, 2005.

[2] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Bdam—batched dynamic adaptive meshes for high performance terrain visualization. In *Computer Graphics Forum*, volume 22, pages 505–514. Wiley Online Library, 2003.

[3] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics (TOG)*, 23(3):796–803, 2004.

[4] Gil Gribb and Klaus Hartmann. Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.

[5] Ulrich Haar and Sebastian Aaltonen. Gpu-driven rendering pipelines. *SIGGRAPH*, 2015. URL `https://advances.realtimerendering.com/s2015/index.html`.

[6] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96. Association for Computing Machinery, 1996.

[7] Brian Karis, Rune Stubbe, and Graham Wihlida. A deep dive into nanite virtualized geometry. *SIGGRAPH*, 2021. URL `https://advances.realtimerendering.com/s2021/index.html`.

[8] Christoph Kubisch. Mesh shading for vulkan. *Khronos Blog*, 2022. URL `https://www.khronos.org/blog/mesh-shading-for-vulkan`.

[9] Amar Patel and Tex Riddell. D3d12 work graphs preview. *DirectX Developer Blog*, 2023.

[10] Federico Ponchio. Multiresolution structures for interactive visualization of very large 3d datasets. 2009.

[11] The Stanford 3D Scanning Repository. URL `http://graphics.stanford.edu/data/3Dscanrep/`.

[12] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. In *ACM SIGGRAPH 2004 Sketches*, page 22. 2004.