Twelfth Hungarian Conference on Computer Graphics and Geometry, Budapest, 2026

High-Performance GPU-Driven Rendering and Hierarchical Culling Architecture with Segmented Data-Oriented Entity-Component-System and Mesh Shader Support

Péter Tamás and András Fridvalszky

Department of Control Engineering and Information Technology,
Faculty of Electrical Engineering and Informatics,
Budapest University of Technology and Economics,
Műegyetem rkp. 3., H-1111 Budapest, Hungary.

Abstract

One of the most significant challenges for modern AAA game engines is the efficient real-time management of millions of entities within a hybrid CPU-GPU environment. The traditional object-oriented paradigm presents severe performance bottlenecks in this context, manifest as frequent cache misses, poor parallelizability, and cumbersome GPU synchronization. This paper details a fully data-oriented Entity-Component-System architecture integrated with a high-performance GPU-driven rendering and hierarchical culling system developed for our open-source engine, the Synapse Engine. Key elements of the system include segmented compact storage using a Static-Dynamic-Stream partition, bitflag-based change tracking, sparse-set based $\mathcal{O}(1)$ indexing, a three-stage data-oriented model pipeline (Raw $\rightarrow$ Cooked $\rightarrow$ GPU), an eight-bucket indirect draw command system, and hierarchical Model-Mesh-Meshlet level culling. Visibility is determined through a combination of frustum, Hi-Z occlusion, cone, and zero-pixel-triangle techniques, while the system provides full support for the mesh shader pipeline and bindless index-based access. This architecture ensures maximum cache locality, SIMD-friendly parallel processing, and minimal GPU upload overhead, maintaining high efficiency for dynamic scene changes and component operations. Benchmark results demonstrate that the solution achieves real-time performance even with over one million entities, significantly outperforming traditional ECS and rendering architectures.

1. Introduction

1.1. Performance Bottlenecks in Modern GPU-Driven Engines

The performance of contemporary real-time graphical applications and game engines increasingly depends on the efficiency of CPU-GPU cooperation. While modern graphics APIs enable indirect draw calls, compute-shader-based culling, and the simultaneous processing of millions of primitives, they also place immense pressure on the CPU to manage entities, components, and models. Engines must handle millions of static and dynamic objects per frame while maintaining synchronization and managing multi-frame-in-flight states (Figs. 1-2).

The object-oriented programming paradigm, which served engine development for decades through classes and polymorphism, now causes significant performance issues in GPU-driven environments. Objects that store data internally and are accessed via pointers or virtual functions lead to scattered memory layouts, resulting in high cache-miss rates, branch mispredictions, and difficult parallelization. Since GPUs operate most efficiently with compact, contiguous data arrays, the traditional CPU-side collection and upload of data creates substantial overhead. These limitations necessitate a shift toward the data-oriented paradigm, where entities are simple identifiers and data is stored in homogeneous, compact vectors.

1.2. Objective and Structure

The objective of this paper is to present a production-ready architecture that seamlessly integrates a segmented data-oriented *Entity-Component-System* with a GPU-driven ren-

dering and hierarchical culling system. The solution provides maximum cache locality and minimal CPU-GPU overhead, sustaining real-time performance for over one million entities.

The paper is structured as follows: Section 2 reviews previous work and foundational technologies. Section 3 details the data-oriented design principles and our segmented ECS architecture. Section 4 introduces the data-oriented model pipeline and management systems. Section 5 discusses the GPU-driven rendering and indirect draw system. Section 6 presents the hierarchical culling architecture, while Section 7 details the specific culling techniques applied. Section 8 explains material overrides and rendering fine-tuning. Section 9 provides performance comparisons and benchmark results. Finally, Section 10 summarizes the findings and outlines future research directions.

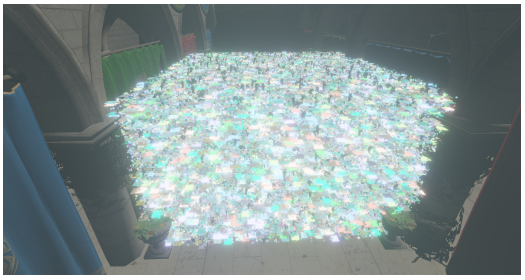

Figure 1: *Rendering of a highly complex scene in the Synapse Engine, successfully managing over one million entities in real-time.*

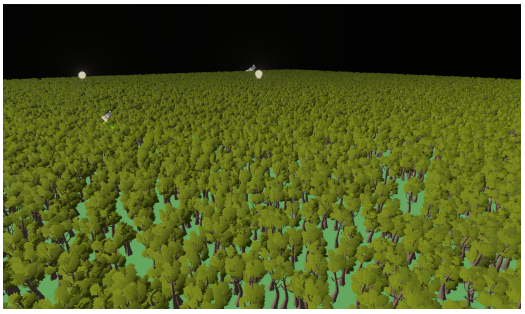

Figure 2: *An internal, high-density view of the scene, demonstrating the performance of the GPU-driven pipeline with 100k tree entities.*

1.3. Terminology and Definitions

To provide a clear framework for the following technical discussion, we define the core concepts of our architecture as follows:

Entity-Component-System (ECS): A data-oriented architectural pattern where *Entities* are simple numerical IDs, *Components* are plain data structures stored in contiguous memory, and *Systems* are logical units that process these arrays in parallel. This ensures high cache locality and SIMD efficiency.

GPU-Driven Rendering: A rendering strategy that shifts the management of draw calls from the CPU to the GPU. Instead of the CPU issuing direct commands, the GPU generates and manages its own draw commands within GPU buffers using indirect draw calls, significantly reducing CPU overhead and driver bottlenecks.

GPU-Driven Hierarchical Culling: A GPU-based visibility process that uses compute shaders to filter geometry at multiple levels (Model, Mesh, and Meshlet). By combining techniques such as frustum and Hi-Z *occlusion culling*, the GPU can autonomously update the instance counts of draw commands, ensuring that only visible primitives are processed by the rendering pipeline.

Meshlets and the Mesh Shader Pipeline: A *meshlet* is a small, localized cluster of geometry containing a strictly bounded number of vertices and triangles created by partitioning a larger complex mesh to fit optimally within the GPU's local cache (Fig. 3). These clusters serve as the foundational data structure for the Mesh Shader Pipeline, a modern compute-driven graphics architecture that replaces traditional vertex, tessellation, and geometry stages. This pipeline is typically preceded by a Task Shader, which evaluates and conditionally dispatches these meshlets based on fine-grained culling logic, enabling highly efficient, on-chip geometry filtering and vertex reuse prior to rasterization.

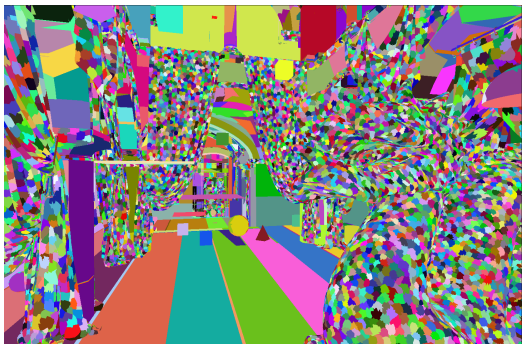

Figure 3: *Meshlet-level visualization of the scene. The geometry is partitioned into individual meshlets, each assigned a unique color to illustrate the fine-grained subdivision used during the rendering process.*

2. Previous Work

2.1. Entity-Component-System Architectures

The shift from traditional object-oriented programming to *Data-Oriented Design* has become a crucial standard in modern game engine architecture. A prominent benchmark for

this paradigm is *EnTT*, a highly performant, production-ready C++ ECS library [2]. EnTT utilizes a sparse-set data structure to map entity identifiers to component data, achieving exceptional cache locality and $\mathcal{O}(1)$ component lookups. While our architecture shares the foundational sparse-set principles popularized by EnTT, we extend this concept to better suit our specific GPU-driven needs. By implementing a custom *segmented storage model* alongside a *bitflag-based change tracking system*, we proactively eliminate redundant CPU iterations over unmodified data.

2.2. Geometry Optimization

With the advent of the modern mesh shader pipeline, the traditional vertex processing model is increasingly being replaced by the processing of small, localized geometry clusters known as meshlets. The industry standard for this complex geometry processing is the *Meshoptimizer* library, created by Arseny Kapoulkine [5]. Meshoptimizer provides robust algorithms for vertex cache optimization, overdraw reduction, and the generation of both hierarchical Levels of Detail and meshlets. Our mesh processing pipeline heavily relies on the concepts and algorithms provided by Meshoptimizer to efficiently partition arbitrary raw geometry into optimized, GPU-ready meshlet data structures that perfectly feed into our Task and Mesh shaders.

2.3. GPU-Driven Rendering

The evolution of GPU-driven rendering has been significantly shaped by recent AAA titles pushing the boundaries of hardware capabilities [1]. A major point of reference and inspiration for our rendering pipeline is the architecture detailed by Erik Jansson in his Digital Dragons presentation, "GPU driven Rendering with Mesh Shaders in Alan Wake 2" [3]. Jansson outlines a state-of-the-art approach where the CPU is largely relieved of per-object draw call generation. Instead, compute shaders handle complex hierarchical culling (including frustum, occlusion, and sub-mesh culling) and dispatch work directly to the mesh shader pipeline. Our engine adopts similar philosophies regarding compute-based hierarchical culling, bindless resource management, adapting them to work seamlessly with our custom eight-bucket indirect draw system and segmented ECS architecture.

3. Segmented Entity-Component-System Architecture

3.1. Advantages of the Data-Oriented Design Paradigm

The data-oriented design paradigm follows a fundamentally different philosophy than the traditional object-oriented approach. Entities are no longer complex objects but simple identifiers, while data is stored in homogeneous, compact vectors, providing excellent cache locality. Since components of the same type are located contiguously in memory, the CPU prefetcher operates efficiently.

This linear, fragmented data structure enables highly SIMD-friendly processing. Systems can run completely independently and in parallel, as there are no hidden dependencies or pointer-based relationships. Processing is easily vectorizable, allowing different systems to work on separate threads with minimal synchronization. This fully parallel, fragmented approach not only significantly increases CPU-side performance but also simplifies and accelerates data uploads to the GPU by providing compact, contiguous buffers. True $\mathcal{O}(1)$ indexing using a sparse-set data structure allows the system to immediately determine if an entity possesses a specific component and locate it within the compact array.

3.2. Sparse-set Based Indexing and Compact Storage

Following the core principles of the data-oriented paradigm, entities are represented as simple `uint32_t` identifiers. Component storage is implemented via a sparse-set data structure. Each component type has an independent sparse-set consisting of two main parts: a sparse array and a dense indexing array. Each element in the sparse array contains either a `UINT32_MAX` value or an index pointing into the dense array. The dense array stores only the identifiers of entities that actually possess the given component. This bidirectional indexing mechanism enables true $\mathcal{O}(1)$ lookups and efficient cross-referencing between different component types (Fig. 4).

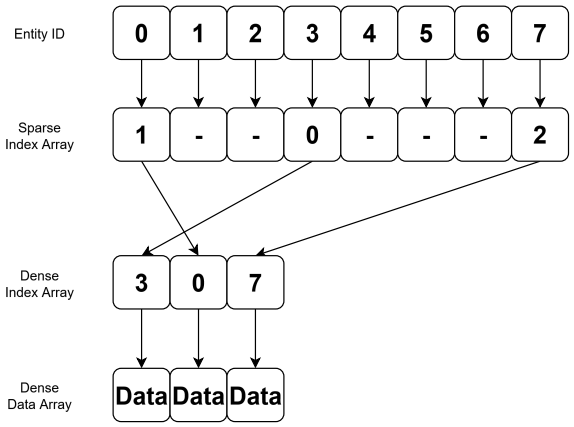

Figure 4: *Sparse-set indexing mechanism. Entity IDs are mapped through a sparse array to tightly packed dense data arrays, ensuring $\mathcal{O}(1)$ component lookups.*

Building upon this, the dense array serves as the foundation for any arbitrary auxiliary arrays kept strictly in sync with it. These synchronized arrays can store diverse data, such as a component data vector, a bitflag vector, or a dedicated compact version for the GPU. For example, the CPU-side structure of a *Transform component* contains the position, rotation, scale, and the global transformation matrix, while

the GPU-side structure only contains the `globalTransform` and `globalTransformIT` matrices required by the shaders.

3.3. Bitflag-Based Change Tracking

While the dense array of a sparse-set ensures cache-friendly linear iteration, it inherently lacks a mechanism to track temporal state changes. Consequently, systems are typically forced to iterate through the entire array and blindly recompute data for every component frame by frame, regardless of whether their actual parameters have been altered. To eliminate this redundant processing, we introduce a bitflag vector strictly synchronized with the dense array. Each component pool maintains this vector, utilizing specific bits to represent states such as **UPDATE, CHANGE, REGENERATE**, alongside other application-specific flags.

Whenever an entity's component is modified, its corresponding UPDATE flag is set to signal a required refresh. Systems evaluating these components will only execute expensive computations if this specific flag is active. Furthermore, a summary state is maintained at the pool level. If no component within the range registers an active UPDATE flag, the system can entirely bypass the iteration for that region, saving significant CPU cycles.

3.4. Segmented Component Storage

While bitflag-based tracking significantly reduces unnecessary calculations, systems still need to iterate through the entire compact array for dynamic components. The solution is the physical partitioning of the compact array into three regions with different update frequencies: *Static*, *Dynamic*, and *Stream*. Physically, the array remains a single contiguous block of memory, with two pointers defining the boundaries between regions.

- **Static region**: Contains components that change very rarely. Its management is handled by an atomic counter and an associated dirty list.
- **Dynamic region**: Contains moderately changing components, protected by the previously mentioned bitflag vectors.
- **Stream region**: Contains components that change every frame. No flag protection is applied.

This segmented system handles component addition, deletion, and category switching elegantly, requiring at most three swap operations for any of these tasks (Fig. 5).

3.5. Parallel System Processing and GPU Synchronization

The segmented architecture allows for the fully independent and parallel execution of various Systems. While Systems generally run in parallel, explicit synchronization points are utilized to manage data dependencies whenever one System's output serves as another's input. The physical partitioning into Static, Dynamic, and Stream regions enables these segments to be processed concurrently on separate threads. Furthermore, because the data within these pools is fragmented into uniform blocks, the architecture facilitates a vectorized approach, allowing for SIMD-level parallelization that ensures high-throughput processing.

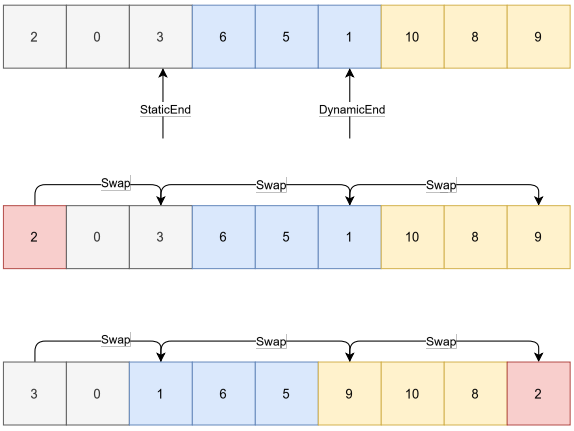

Figure 5: *Segmented component storage. A single contiguous array is partitioned into Static, Dynamic, and Stream regions. Deletions are efficiently handled via swap-and-pop operations to strictly preserve memory contiguity.*

GPU synchronization is demonstrated through the lifecycle of the Transform component. During the update phase, the System iterates through components in parallel and recalculates global transformations only when a change is detected. Following the completion of CPU-side tasks and ensuring the GPU fence has been signaled, the upload phase efficiently transfers only the modified components to mapped coherent buffers, thereby minimizing CPU-GPU synchronization overhead.

4. Data-Oriented Model Pipeline and Management Systems

4.1. Mesh Processing

The primary objective of the model processing pipeline is to efficiently prepare models of arbitrary complexity for the GPU utilizing a strict data-oriented approach. This transformation process is divided into three distinct stages: **RawModel**, **CookedModel**, and **GPUModel**.

The **RawModel** serves as the foundational stage, containing the initial data parsed by the model loader. Although the data is already stored in vectors, it remains fragmented on a per-mesh basis.

The transition to the **CookedModel** stage is governed by

a series of modular *mesh processors* (e.g., `MeshoptimizerLodProcessor`, and `ColliderProcessor`). These processors act as a sequential pipeline, yet they execute their logic in a highly parallelized manner per mesh. They are responsible for calculating and populating all necessary structural data, including Level of Detail, meshlets, and their respective bounding colliders (Fig. 6).

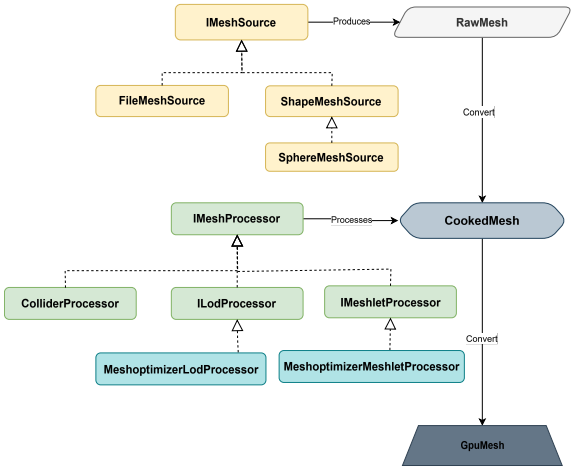

Figure 6: *The three-stage data-oriented model pipeline. Asset geometry is sequentially transformed from a raw state into an optimized intermediate format via specialized processors, before final conversion into GPU-ready structures.*

Finally, the `MeshConverter` transforms the **CookedModel** into the **GPUModel**. In this ultimate stage, the data is flattened into a completely GPU-ready representation. To maximize memory bandwidth and performance, particularly vertex positions are decoupled into a dedicated, separate buffer, while the remaining vertex attributes are stored independently. Furthermore, the index buffers for all LOD levels are batched contiguously.

Crucially, the system utilizes a flat descriptor architecture for LODs. Every mesh is accompanied by exactly four LOD descriptors containing essential vertex and index offsets. If a mesh does not generate four distinct LOD levels, the pipeline automatically duplicates the descriptor of the last valid LOD to populate the remaining slots. Because the system rigidly guarantees exactly four LOD descriptors per mesh, the GPU can trivially resolve the correct descriptor using the direct calculation $4m_i + l_i$, where m_i is the mesh index and l_i is the LOD level, entirely eliminating the need for variable-length lookups or complex branching within the shaders.

4.2. Bindless Management and ECS Integration

The `ModelManager` handles the asynchronous loading of models through the aforementioned three-stage pipeline. All successfully loaded models are placed into a compact vector, indexed sequentially starting from zero. To facilitate immediate and unified access on the GPU, the manager constructs a centralized model address buffer, which stores the specific device memory addresses for every individual model's constituent buffers. This bindless approach enables shaders to dynamically resolve and access any specific buffer of any model using merely its integer index (Fig. 7). The animation system strictly mirrors this logic: the `AnimationManager` maintains an identical bindless architecture to provide uniform shader access to animation data.

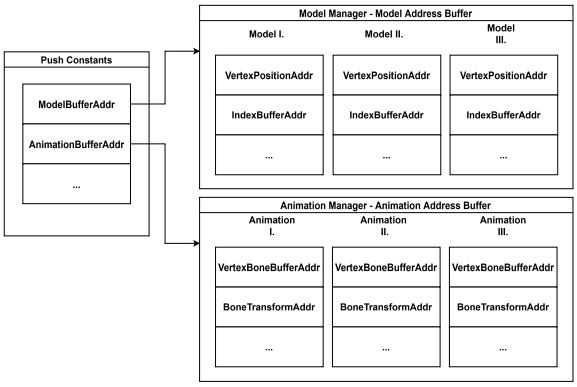

Figure 7: *Bindless buffer addressing architecture. Push constants provide global base pointers to the Model and Animation Address Buffers, allowing shaders to dynamically access per-model vertex, index buffer addresses.*

Integration with the data-oriented ECS is achieved through exceptionally lightweight components (`ModelComponent` and `AnimationComponent`), which store only the integer indices of their respective assets. Thanks to the dense array structure of the ECS, the scene can efficiently accommodate an arbitrary number of these components. During the rendering process, the GPU simply reads this integer index to immediately determine which specific model or animation data belongs to a given entity. Consequently, culling and rendering shaders can seamlessly identify and process millions of static and animated meshes uniformly, completely bypassing traditional resource binding overhead.

5. GPU-Driven Rendering and Indirect Draw System

5.1. Principles of Fully Compute-Driven Rendering

The core objective of our rendering architecture is to eliminate the bottlenecks associated with the CPU-side management of millions of objects. Once the model and animation pipelines are integrated, the system can access any asset via its unique index. The fundamental principle of this design is that the CPU is entirely oblivious to the specific details of the scene being rendered. It does not issue direct draw calls and has no immediate knowledge of which entities are currently visible.

The entire lifecycle of a frame from visibility determination to draw call generation is managed by the GPU using compute shaders. The CPU's role is strictly limited to initiating the process: it dispatches the compute shaders and then issues a few highly batched indirect draw commands. All actual rendering commands reside in GPU buffers.

A critical component of this autonomy is the indirection step. Since only a subset of the total entity count is visible at any time, the compute-based culling shaders write the IDs of visible entities into a global *Instance Index Buffer*. They also use atomic operations to update the `instanceCount` in the corresponding indirect draw commands. During the final rendering stage, the shader hardware uses the built-in `gl_InstanceIndex` to access this buffer, retrieve the correct entity identifier, and subsequently fetch the relevant transform and model data. This design ensures that the CPU remains decoupled from the per-instance complexity of the scene, providing a truly GPU-driven pipeline.

5.2. Eight-Bucket Indirect Draw Command System

To maintain maximum efficiency across varying material types and hardware capabilities, the architecture utilizes an eight-bucket indirect draw command system. Physically, all indirect draw commands for the entire scene are stored within a single, large, global indirect draw buffer. This buffer is logically partitioned into eight distinct regions (buckets) based on two primary criteria: the rendering pipeline and the material properties.

The primary level of separation distinguishes between the Traditional Vertex Shader Pipeline and the Modern Mesh Shader Pipeline. This ensures that the engine can utilize the most appropriate hardware path for each specific mesh without duplicating culling logic. The secondary level of separation handles the four fundamental material types supported by the engine: **Opaque 1-sided, Opaque 2-sided, Transparent 1-sided, Transparent 2-sided** (Fig. 8).

The combination of these two pipelines and four material types results in the eight specific buckets. It is crucial to note that while culling begins at the model level, the multi-draw indirect commands are strictly managed at the mesh level. Every mesh of every loaded model possesses its own pre-allocated indirect draw command, alongside its own specific material index and pipeline flag. This granular design allows a single model to consist of multiple meshes that arbitrarily mix traditional and mesh shader pipelines, as well as varying material types (Fig. 15).

During the GPU-side culling pass, the compute shader evaluates these parameters on a per-mesh basis to identify the exact target bucket and locate the corresponding draw command within the global buffer's range. By using atomic operations to increment the `instanceCount` of these specific pre-allocated commands, the engine can render an entire scene of immense complexity using exactly eight multi-draw indirect calls, drastically reducing CPU command overhead and driver-side bottlenecks.

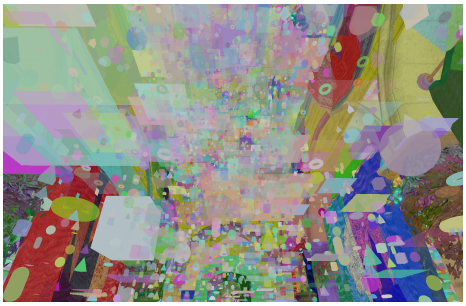

Figure 8: *Simultaneous rendering of opaque and transparent objects within the scene, demonstrating the practical result of the material bucketing system.*

5.3. Descriptor Architecture and Memory Allocation

To replace CPU-side binding with a purely GPU-driven data flow, the engine utilizes a highly structured descriptor hierarchy. This architecture relies on three core components:

Model Allocation and Mesh Indexing: Each model possesses an entry in the `ModelAllocationBuffer` (synchronized with the `ModelManager`), which stores a `meshAllocationOffset` and `meshCount`. This offset points into a single, global `MeshAllocationInfo` buffer. To support the fixed four-LOD architecture, mesh descriptors are packed sequentially ([`Mesh0_Lod0`, `Mesh0_Lod1`, ...]), allowing compute shaders to index them dynamically using the formula $4m_i + l_i$.

MeshAllocationInfo (Culling Phase): For each mesh LOD, this structure defines the pipeline type and holds pre-allocated offsets for both the indirect draw commands and the instance index buffer, strictly partitioned by the four material types. During culling, if a mesh is visible, the shader reads this info to atomically increment the target indirect draw command and writes the entity ID into its designated instance index range.

MeshDrawDescriptor (Rendering Phase): Because the global indirect draw buffer mixes various models and pipelines, rendering shaders must reverse-engineer each draw call's context. The `MeshDrawDescriptor` array is kept synchronized with the indirect command buffers. Using the hardware-provided `gl_DrawIndex`, the shader fetches its corresponding descriptor to retrieve the `modelIndex`, `meshIndex`, `lodIndex`, and `instanceOffset`. With this context, the shader can accurately access the exact vertex/index ranges and the original entity transform data.

5.4. Windowed Buffer Allocation for Dynamic Scene Management

Allocating and partitioning ranges on the CPU can introduce severe overhead if not managed correctly. It is critical to divide the global indirect draw command buffer and the global instance index buffer in a way that minimizes the need for complete reallocations whenever material types change, new models are loaded, or meshes are added to the scene.

To achieve this, the engine implements an overestimated, *windowed allocation strategy*. Instead of allocating the exact required memory footprint, ranges are allocated in discrete multiples of a predefined window size. Crucially, this window size is highly granular and customizable: it can be tailored per bucket, per material type, per model, and even down to the specific mesh level.

This identical overestimation logic governs the instance index buffer. Because entities are continuously added or removed frame-by-frame, the system calculates the maximum current occurrence of each model and proactively overestimates the required capacity using this windowed approach. The window size is dynamically adjustable based on expected volatility. For instance, highly dynamic models that frequently spawn and despawn by the thousands receive large allocation windows, whereas strictly static or rarely changing objects are assigned much smaller, tighter windows.

By strategically over-allocating these buffer ranges, the rendering architecture remains exceptionally efficient, completely avoiding the severe performance penalties of constant buffer regenerations during dynamic scene changes.

6. Gpu-Driven Hierarchical Culling Architecture

The core of the GPU-driven rendering system is a multi-stage, hierarchical culling process executed entirely within compute shaders. Visibility is evaluated at three distinct levels: Model, Mesh, and Meshlet ensuring that complex geometry is filtered with increasing precision while minimizing unnecessary vertex and task shader overhead (Fig. 14).

6.1. Model-Level Culling and the Dual-Path Strategy

The process begins at the Model level, which serves as the coarsest filter. The culling shader iterates through all active ModelComponents, testing their pre-transformed world-space colliders (AABB and Sphere) against the camera frustum and the Hi-Z depth buffer (Fig. 9). A unique feature of this architecture is the dual-path strategy triggered immediately after a model is deemed visible:

- **Fast Path:** For simple models with low mesh counts or limited geometric complexity, the system bypasses the secondary mesh culling dispatch. Instead, the model culling shader performs mesh-level visibility tests inline using an optimized thread distribution to eliminate dispatch overhead. This is particularly effective for small or distant objects where a separate compute pass would be counterproductive.
- **Slow Path:** For highly complex models, the shader writes an (entity ID, model index) pair into a specialized Visibility Buffer. This buffer acts as the input for a subsequent *Collaborative Mesh Culling* pass, where workload is distributed across multiple thread groups to handle the high mesh count efficiently.

By utilizing baked colliders, the system treats static and animated models uniformly. For animated entities, precalculated global colliders are provided per frame, ensuring that the culling logic remains identical regardless of whether the model is rigid or skinned (Fig. 10, Fig. 11).

6.2. Collaborative Mesh-Level Culling Pass

For complex models retrieved from the visibility buffer, mesh-level culling is executed in a dedicated collaborative compute pass. In this stage, the shader evaluates the previously verified visible models on a strictly per-mesh basis. To streamline the data pipeline, each mesh is bound to a single universal collider (comprising an AABB and a bounding Sphere) the architecture deliberately avoids utilizing LOD-specific colliders.

The **collaborative nature** of this pass is driven by its highly optimized thread distribution: a single task entry from the visibility buffer (representing one complex model) is explicitly mapped to 32 threads. In this paradigm, the model acts as the workload group, and these 32 threads work collaboratively to process its constituent meshes. By iterating through the mesh array using a partially unrolled loop (with a stride of 32), the shader ensures that the workload is distributed efficiently and homogeneously, even across massive models with extensive mesh counts.

During execution, the compute shader reapplies the frustum, Hi-Z occlusion, and zero-pixel-triangle tests, evaluating them at this fine-grained mesh level. If a mesh successfully passes all visibility checks, the shader utilizes atomic operations to increment the instanceCount of its target bucket within the eight-bucket indirect draw command system. Simultaneously, it securely writes the entity's identifier into the global instance index buffer at the correct pre-allocated offset.

6.3. Dynamic Level of Detail Selection

Because each mesh relies on a single, universal collider, Level of Detail selection is decoupled from the baseline visibility culling and is instead determined dynamically based on the object's projected screen-space size. During the collaborative mesh culling pass, the compute shader calculates the bounding sphere's screen-space projection. Crucially, this calculation leverages the exact same sphere-to-screen-space projection mechanism [4] utilized for the Hi-Z occlusion

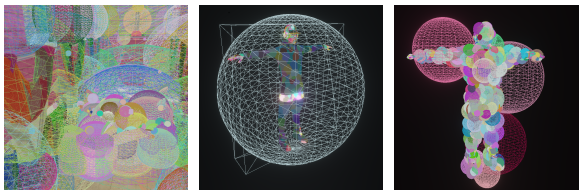

Figure 9: *External debug visualization displaying the pre-baked colliders across the visibility hierarchy. Model-level AABBs, Mesh-level bounding volumes, and Meshlet-level clusters are visualized simultaneously.*

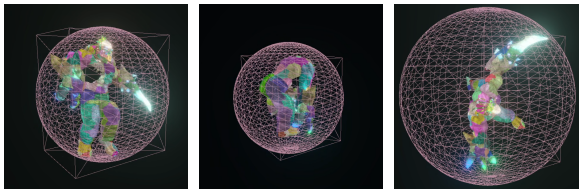

Figure 10: *Animated model-level colliders (AABB and Sphere) dynamically updating across three distinct animation frames.*

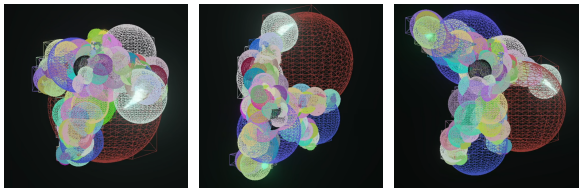

Figure 11: *Animated meshlet-level bounding spheres adapting to skeletal deformation across three frames.*

culling, effectively recycling mathematical operations and minimizing redundant compute overhead.

Based on this projected footprint, the shader directly determines the appropriate LOD level, ranging from 0 to 3. The selection relies on predefined pixel thresholds: for instance, a projected size of 512 pixels or larger triggers LOD 0 (maximum detail), a size of 256 pixels triggers LOD 1, and so forth for lower detail levels (Fig. 12). This approach is highly efficient because it circumvents the need to evaluate separate colliders for each individual LOD tier. Consequently, nearby models automatically render at the highest fidelity, while distant objects seamlessly scale down to lower LODs without incurring any additional compute overhead.

Once the dynamic LOD value is calculated, it directly governs the routing of the draw command. The shader utilizes this LOD index to fetch the corresponding mesh descriptor, ensuring that the mesh is placed into the correct indirect draw command bucket.

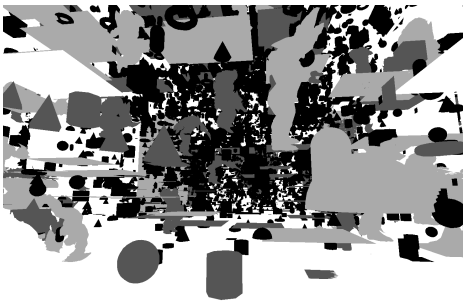

Figure 12: *Level of Detail debug visualization. The dynamically selected LOD levels (0–3) are represented by grayscale values: white indicates highest detail, light gray is LOD 1, dark gray is LOD 2, and black represents lowest detail.*

6.4. Task Shader-Based Meshlet Culling and Pipeline Support

The final and most granular stage of the hierarchy is Meshlet culling, which is executed within the Task Shader. At this level, the system performs highly precise visibility tests on individual meshlet colliders. The Task Shader acts as a dynamic scheduler, determining whether a specific meshlet should be dispatched to the Mesh Shader for rasterization. This stage incorporates specialized filtering techniques, including cone culling (for backface meshlet elimination), frustum culling, refined occlusion tests, and zero-pixel triangle checks.

Once a meshlet passes these rigorous tests, the Task Shader transmits the necessary meshlet indices to the Mesh Shader via a localized payload. The Mesh Shader then reconstructs the primitives using the `vertexIndices` and `triangleIndices` arrays, generating vertices directly on the chip. This compute-centric approach significantly reduces vertex fetch overhead and facilitates high-performance collaborative processing within the shader

Figure 13: *The final rendered scene from the perspective of the main camera.*

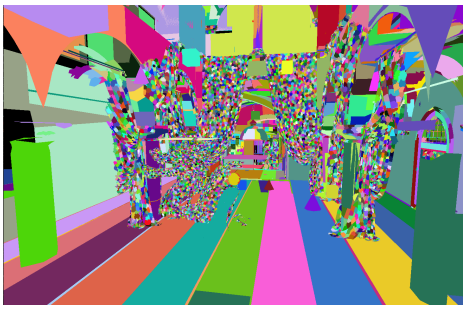

Figure 14: *External camera debug view of the hierarchical culling process, illustrating the filtering of geometry at Model, Mesh, and Meshlet levels.*

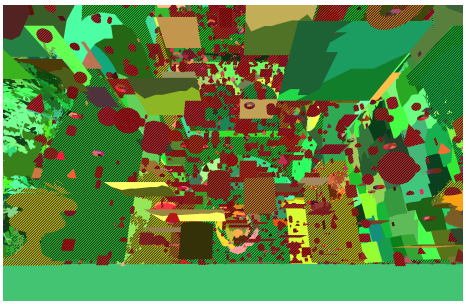

Figure 15: *Debug visualization demonstrating per-mesh pipeline and LOD flexibility. Smooth surfaces with varied colors represent the Mesh Shader pipeline with meshlet visualization, while hatched surfaces indicate the Traditional Vertex Shader pipeline. The underlying color spectrum (green, yellow, orange, red) denotes varying LOD levels (0-3), highlighting that different meshes within the same model can utilize distinct pipelines, material types, and LODs simultaneously.*

6.5. Pipeline Support and Performance Benefits

The architecture is designed to provide full, seamless support for both traditional vertex shader and modern mesh shader pipelines. The rendering flow adapts dynamically based on the assigned pipeline: for the traditional pipeline, the indirect draw command is submitted directly to the vertex shader immediately after successful mesh-level culling. In contrast, the mesh shader pipeline delegates the final, fine-grained culling to the Task Shader before rendering. The previously detailed eight-bucket system handles this routing flawlessly, as culling shaders natively categorize and write commands into the appropriate bucket based on both material and pipeline type.

Extensive profiling reveals a stark contrast in performance between the two approaches. The traditional vertex shader pipeline proves to be significantly slower when processing high-density scenes, as it lacks the ability to discard invisible geometry at the sub-mesh level. By utilizing the modern mesh shader pipeline coupled with Task Shader-driven culling, the system dramatically reduces overdraw and vertex processing, achieving up to a 1.5-2.0x performance speedup (Section 9.3).

Ultimately, this hierarchical culling architecture manages millions of entities through a single, optimized compute dispatch sequence while maintaining maximum GPU utilization (Fig. 13). The combination of dynamic fast/slow path selection, 32-thread collaborative mesh processing, and projection-based dynamic LOD management ensures that every model type achieves peak performance across the entire scene.

7. Applied Culling Techniques

7.1. Frustum culling

Frustum culling serves as the initial filtering stage at every level of the hierarchy. We utilize the six planes of the camera frustum, ordered strategically to maximize early rejection. The process involves two types of intersection tests: first, a fast but conservative Sphere-vs-Frustum test. If the bounding sphere is entirely outside the frustum, the object is immediately rejected (Fig. 16). If the sphere is entirely inside, the more complex Axis-Aligned Bounding Box (AABB) test is skipped, as the object is guaranteed to be within the view. An AABB test is only performed if the sphere intersects one of the frustum planes.

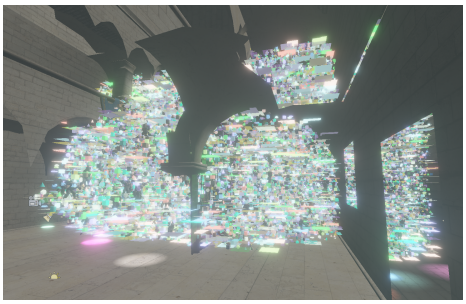

Figure 16: *External debug visualization of frustum culling. While the actual scene contains geometry spanning 360 degrees, the rendering is strictly limited to objects within the main camera's view frustum. All geometry outside this volume is successfully culled and remains empty.*

Instead of a simple boolean result, the frustum culling function distinguishes between three states: *outside*, *inside*, and *intersecting*. This tri-state logic is vital for hierarchical efficiency: if a model is determined to be entirely within the frustum, all its constituent meshes and meshlets are also guaranteed to be inside, allowing the system to bypass frustum tests at subsequent levels. To propagate this information through the pipeline, the **fully inside state** is flagged by setting the 31st bit of the entity identifier.

7.2. Occlusion Culling with Hi-Z Depth Buffering

Occlusion culling is implemented using Hi-Z technique. The process begins by generating a linearized depth texture from the previous frame's depth buffer via a dedicated copy pass. A Hi-Z mip pyramid is then constructed through successive downsampling passes (Fig. 18). This hierarchical occlusion check is applied across the model, mesh, and meshlet culling stages, utilizing a max-reduction sampler to retrieve the most distant depth value within a given footprint.

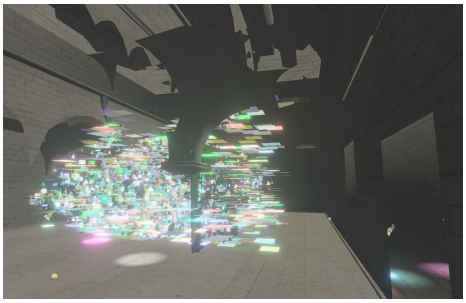

Figure 17: *External debug visualization of occlusion culling. Executed after the initial frustum pass, this stage dynamically discards geometry completely hidden by closer opaque surfaces, as clearly demonstrated by the absence of objects behind the right wall.*

For each visibility test, the object's bounding sphere is projected into screen space to determine its axis-aligned bounding box. The larger of the AABB's dimensions is used to calculate the appropriate mip level where a single texel's footprint exceeds the size of the projected AABB. In practice, the shaders always samples from the mip level with the next higher resolution to ensure a conservative and accurate test. Finally, the sampled Hi-Z depth value is compared against the linearized depth of the sphere's nearest point. If the sampled depth is smaller (closer to the camera) than the object's nearest point, the model, mesh, or meshlet is determined to be fully occluded and is discarded (Fig. 17).

7.3. Cone Culling at the Meshlet Level

Cone culling is applied exclusively at the meshlet level. Each meshlet consists of a strictly bounded number of vertices and triangles. During the asset preparation phase, the Meshoptimizer library calculates the average normal direction of these triangles alongside the maximum deviation angle. This allows the meshlet's overall orientation and angular spread to be mathematically represented as a cone.

Using this geometric property, we implement a software-based backface culling mechanism directly within the Task Shader. Meshlets whose bounding cones face entirely away from the camera are immediately discarded as backfaces (Fig. 19). We execute this filtering early in software rather than relying on the hardware's built-in rasterizer backface culling,

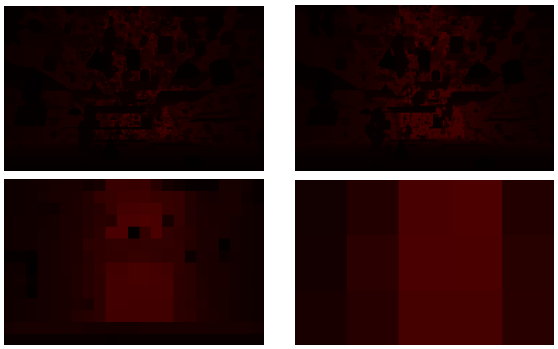

Figure 18: *Visualization of the Hi-Z depth pyramid. The four panels display progressively downsampled mip levels of the linearized depth buffer, which are used to evaluate hierarchical occlusion culling at varying screen-space scales.*

as doing so would otherwise require full vertex processing and matrix multiplications within the Mesh Shader before the primitives could be rejected.

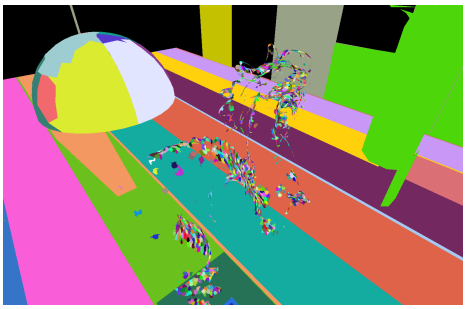

Figure 19: *External debug visualization of meshlet-level cone culling. Captured from a rear perspective relative to the main camera, this demonstrates software-based backface elimination: meshlets oriented away from the view are successfully discarded prior to rasterization.*

7.4. Zero-Pixel Triangle Culling

If a collider's screen-space projection evaluates to less than one pixel in size, the object is visually imperceptible and can be safely discarded. The necessary data for this check is obtained practically for free as a byproduct of the occlusion culling setup, which already computes the screen-space projection of the bounding sphere to determine the appropriate Hi-Z mip level.

Crucially, this check is executed immediately after the projection calculation but strictly before the actual occlusion texture fetch. This precise ordering saves valuable memory bandwidth by completely preventing unnecessary texture reads for sub-pixel objects.

7.5. Synergy and Execution Order of Culling Techniques

These four techniques operate in close synergy and build upon each other hierarchically. Frustum culling is universally executed first at every level, as it is computationally the cheapest and typically eliminates the largest volume of geometry. This is immediately followed by the zero-pixel triangle culling, and subsequently, the Hi-Z occlusion culling.

Cone culling is uniquely reserved for the meshlet level and presents a performance trade-off: in scenarios with highly aggressive and effective occlusion culling, it is often optimal to execute cone culling last or even bypass it entirely to save compute cycles. Ultimately, the robust combination of these techniques guarantees that the absolute minimum number of meshlets reaches the Mesh Shader.

8. Material Overrides and Rendering Fine-Tuning

8.1. Double Indirection-Based Material Override Mechanism

A core objective of the engine is to ensure that every material is loaded only once and stored globally in a single GPU buffer, while simultaneously allowing for per-component material overrides for any specific mesh of a model. We achieve this through a high-performance double indirection mechanism.

At the ECS level, any entity can be assigned a `MaterialOverrideComponent`. This component contains an array of indices corresponding to each mesh of the entity's assigned model. If an index is set to `UINT32_MAX`, the mesh utilizes its original material; otherwise, the specified override material is applied. The indirection operates in two distinct stages:

- **Material Offset Table**: This table is synchronized with the `ModelComponent` dense array. It stores a specific offset that identifies where an entity's material indices begin within the global lookup table. This allows multiple entities sharing the same model to point to the same material list if no overrides are present, significantly saving memory.
- **Global Material Lookup Table**: This is a centralized array containing the actual material indices. These indices provide direct, bindless access to the *Global Material Buffer*, which stores the final material parameters, such as texture descriptors, constants, and shader flags.

The GPU version of the `ModelComponent` includes this offset, granting shaders immediate access to the resolved material list. Consequently, the culling and rendering shaders can identify the correct material with a single indirect lookup, completely bypassing the need for CPU-side material gathering (Fig. 20). This mechanism remains highly efficient for dynamic scene changes: when a new override is applied, the system simply populates a new segment of the lookup table and updates the offset pointer.

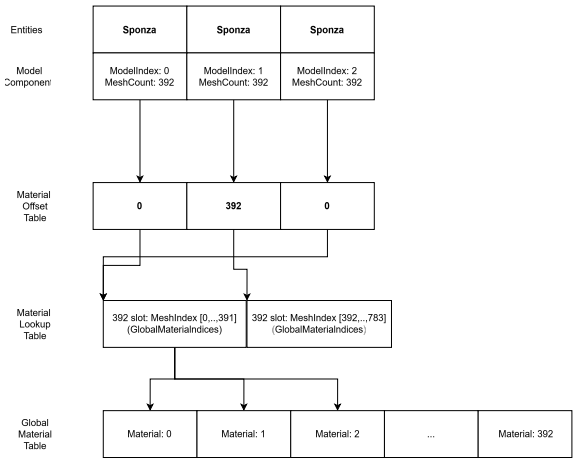

Figure 20: *Double indirection material override architecture. Geometry is mapped to the Global Material Table via an entity-level Material Offset Table and a mesh-level Material Lookup Table, enabling efficient per-instance material customization on the GPU.*

8.2. Integration with the Eight-Bucket System and Batching Potential

The material override system is designed to work seamlessly with the eight-bucket rendering architecture. Because material properties and pipeline flags are resolved at the mesh level, the same mesh can be routed to different buckets (e.g., Opaque vs. Transparent) even within the same model instance. This granular control ensures that the GPU-driven pipeline remains flexible enough to handle complex assets with diverse material requirements.

Furthermore, our vertex layout is uniquely optimized to support high-density batching. Each vertex consists of a `vec3 position` and a `uint32_t` packed value containing both a `nodeIndex` and a `meshIndex`. By embedding the mesh identifier directly at the vertex level, the architecture unlocks advanced material-type batching.

While meshes are conventionally partitioned strictly based on their individual materials, this vertex-level identification allows meshes that share the same broader material type (e.g., Opaque 1-sided) to be aggressively batched into a single draw command during the culling phase, even if they utilize entirely different actual materials. During rasterization, the shader simply extracts the `meshIndex` from the vertex and leverages the previously established double indirection mechanism to resolve the exact, specific material parameters on the fly.

9. Performance Evaluation and Benchmark Results

9.1. Test Environment and Methodology

To accurately evaluate the performance of the proposed architecture, we established a rigorous benchmark methodology designed to isolate CPU and GPU bottlenecks. The primary testing environment consisted of an Intel Core i5-13600K processor, 32 GB of DDR5 memory, and an NVIDIA GeForce RTX 4060 GPU with 8 GB of VRAM.

To ensure the purity of the measurements, tests were conducted under two highly specialized execution profiles:

- **DIST Build (CPU/ECS Isolation):** Compiled via the Visual Studio (MSVC) toolset, this maximum-release configuration strips away all defensive programming mechanisms, including runtime safety validations and assertions. This represents the theoretical maximum throughput of the engine, optimized for scenarios where deterministic logic is guaranteed.
- **Performance Mode (GPU/Culling Isolation):** To evaluate the rendering and culling pipelines without pixel-bound interference, this mode entirely disables lighting simulations and bypasses Multiple Render Targets (MRT). The fragment shader outputs a single unlit color texture, guaranteeing that frame times directly reflect raw geometry processing and culling efficiency.

9.2. ECS Performance: Custom Segmented Sparse-Set vs. EnTT

We benchmarked our custom Segmented Sparse-Set ECS against EnTT using up to 1,000,000 entities to evaluate both raw operations and a realistic engine update loop (Table 1). To simulate a production workload, entities were distributed as 70% Static, 20% Dynamic, and 10% Stream, with 5% of static entities designated as "dirty" per frame to test state tracking costs.

- **State Tracking (Realistic Update):** Our custom ECS completed the 1M entity update in 1.15 ms compared to EnTT's 1.76 ms (34% faster). While standard ECS architectures like EnTT require costly structural modifications (adding/removing tags) to track temporary states, our architecture utilizes $\mathcal{O}(1)$ bitflags and an atomic counter to completely eliminate structural overhead.
- **Mass Entity Deletion:** Our system significantly outperforms EnTT during scene unloads (26.18 ms vs. 37.01 ms at 1M entities) due to an optimized multi-level swap-and-pop algorithm, which minimizes memory disruption across segmented boundaries.
- **Parallel Scalability:** Both architectures perform exceptionally well with multi-threading (Custom ECS: 2.15 ms, EnTT Group: 2.26 ms). However, our custom approach achieves contiguous parallel iteration natively across its categories, bypassing the strict physical memory sorting and exclusivity limitations required by EnTT groups.
- **Component Insertion:** EnTT remains faster at raw insertions (35.59 ms vs. 59.66 ms). Our system incurs a deliberate upfront cost to categorize entities, a trade-off that pays massive dividends during iteration and destruction.

Table 1: *Performance comparison between the Custom Segmented ECS (Synapse) and EnTT. CPU time in ms (lower is better).*

Operation	Entities	Synapse (ms)	EnTT (ms)
Add	100k	6.08	2.99
	1M	59.66	35.59
Remove	100k	2.31	3.29
	1M	26.18	37.01
Iterate (Seq)	100k	0.24	0.20
	1M	2.46	2.25
Iterate (Par)	100k	0.25	0.25
	1M	2.15	2.26
Engine Update	100k	0.11	0.14
	1M	1.15	1.76

9.3. Rendering Pipeline Comparison: Traditional vs. Mesh Shader

To evaluate the architectural advantages of the modern GPU geometry pipeline, we compared the Traditional (Vertex/Fragment) approach against the Mesh Shader (Task/Mesh/Fragment) pipeline on the RTX 4060. This benchmark utilized the NVIDIA Bistro scene to measure performance across two distinct rendering configurations: a lightweight Performance Mode (single color output, no lighting) and a heavy DIST Mode (Deferred Shading generating a full G-buffer alongside three additional debug textures).

In the lightweight Performance Mode, the Traditional geometry pass executed in 0.611 ms, resulting in a total GPU frame time of 0.925 ms. The Mesh Shader pipeline completed the same geometry pass in just 0.411 ms (0.728 ms total GPU time), achieving a **1.49x speedup** in the geometry processing phase.

The performance gap widens significantly under heavy rasterization workloads. In the DIST Mode, writing to multiple render targets drastically increases the memory bandwidth cost per fragment. Under these conditions, the Traditional geometry pass took 1.487 ms (2.234 ms total GPU time). The Mesh Shader pipeline, however, completed the pass in a mere 0.713 ms (1.472 ms total) yielding a remarkable **2.09x speedup** in the geometry pass and a 1.52x improvement in the total GPU time.

9.4. GPU-Driven vs. CPU-Driven Culling

This section evaluates executing the primary hierarchical visibility pass on the CPU versus a GPU Compute Shader.

Both scenarios use a *Multi-Draw Indirect* (MDI) backend to render 1 million instanced entities (12 variations) in the Sponza scene.

In the CPU-driven scenario, a parallelized C++ implementation evaluates bounding volumes. Despite MDI, uploading draw buffers to VRAM creates a severe PCIe bottleneck, resulting in **12.244 ms** CPU time. Furthermore, lacking CPU-side model and mesh-level occlusion culling, a massive volume of occluded meshlets needlessly floods the Task Shader. Combined with transfer stalls, this inflates GPU execution to **3.481 ms** even under the full Mesh Shader pipeline. Testing this setup highlighted the Task Shader's mitigation capabilities: the Traditional pipeline surged to **7.687 ms**, and Mesh Shaders without occlusion took **4.504 ms**, yielding a **12.658 ms** total frame time (79 FPS) (Table 2).

In contrast, the GPU-driven scenario shifts culling entirely to Compute Shaders. The CPU's role is reduced to a single lightweight dispatch, slashing CPU time to **0.153 ms**. Generating visibility buffers and commands directly in VRAM eliminates PCIe transfers and host sync barriers. The GPU processes the full frame in **0.677 ms**, yielding a **0.744 ms** total frame time (1344 FPS). These results prove that high-density scenes require a self-contained, GPU-exclusive data flow.

Table 2: *CPU vs. GPU culling performance (1M entities).*

Scenario	CPU (ms)	GPU (ms)	FPS
CPU-Driven	**12.244**	**-**	**79**
– Full Mesh	-	3.481	-
– Mesh (no occ.)	-	4.504	-
– Traditional	-	7.687	-
GPU-Driven	**0.153**	**0.677**	**1344**

9.5. Culling Ablation Study: Impact of Individual Techniques

To quantify the performance contribution of each visibility mechanism, we conducted a cumulative ablation study using the Mesh Shader pipeline rendering 1 million instanced entities. Starting from a brute-force baseline, we incrementally enabled filtering stages to observe their impact on GPU execution time (Table 3).

- **Baseline (No Culling, LOD 0):** Processing the entire scene blindly with maximum geometric detail resulted in severe rasterization bottlenecks and a massive baseline GPU time of **124.407 ms**.
- **+ Cone Culling:** Enabling only Task Shader backface cone culling reduced the execution time to **94.123 ms**, demonstrating the raw cost of processing rear-facing triangles before any spatial filtering is applied.
- **+ Hierarchical Frustum Culling:** Activating frustum checks across all three architectural levels (Model, Mesh, Meshlet) eliminated vast amounts of off-screen geometry, drastically dropping the GPU time to **2.878 ms**.
- **+ Hierarchical Hi-Z Occlusion:** Introducing occlusion culling at all three levels successfully discarded entities hidden behind architectural structures. This refined depth-filtering reduced the GPU time to just **0.681 ms** (yielding 1270 FPS).
- **+ Dynamic LOD (Full System):** Finally, enabling projection-based LOD selection optimized distant geometry, achieving the optimal GPU execution time of **0.653** ms and pushing the overall frame rate to roughly 1398 FPS.

These results highlight that while Frustum and Occlusion culling provide the most significant macro-level optimizations, combining them with fine-grained Cone culling and Dynamic LOD is essential for reaching absolute peak performance.

Table 3: *Cumulative ablation study of culling techniques (1 million entities).*

Active Pipeline Stages	GPU Time (ms)
Baseline (No Culling)	124.407
+ Cone Culling	94.123
+ Hierarchical Frustum	2.878
+ Hierarchical Hi-Z Occlusion	0.681
+ Dynamic LOD Selection	**0.653**

9.6. Scalability Across Hardware Generations (10k to 2M Entities)

To evaluate hardware scalability under extreme geometric loads, we benchmarked the GPU-driven Mesh Shader pipeline (Performance Mode) across 10,000 to 2,000,000 instances (Fig. 21) using RTX 4060 and RTX 4070 Super GPUs (Table 5).

The results demonstrate predictable, near-linear scaling without introducing CPU bottlenecks. At 1 million entities, total GPU frame times are exceptionally low: **0.671 ms** (RTX 4060) and **0.406 ms** (RTX 4070 Super). Even under the peak load of 2 million entities, total execution remains strictly under 1 ms (0.992 ms and 0.578 ms, respectively). Furthermore, the compute-based ModelCullingPass scales linearly, processing 2 million hierarchical bounding volumes in just 0.680 ms on the RTX 4060. These metrics conclusively prove that keeping all visibility and dispatch logic exclusively within VRAM guarantees flawless, high-performance scaling across different hardware tiers. This compute-centric approach unlocks massive performance headroom, preserving the vast majority of the frame budget for advanced lighting, ray tracing, and high-fidelity materials.

Table 4: *Hardware scalability on RTX 4060.*

Entities	Model Culling (ms)	Total GPU (ms)
10,000	0.013	0.253
100,000	0.050	0.549
1,000,000	0.360	0.671
2,000,000	0.680	0.992

Table 5: *Hardware scalability on RTX 4070 Super.*

Entities	Model Culling (ms)	Total GPU (ms)
10,000	0.009	0.140
100,000	0.036	0.285
1,000,000	0.223	0.406
2,000,000	0.404	0.578

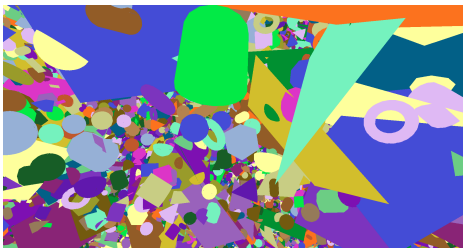

Figure 21: *High-density stress test environment. The scene is populated with 1 million instanced entities (12 geometric variations) randomly distributed within a [500,500] bounding volume across all axes. This dense configuration maximizes spatial overlap and occlusion to rigorously stress-test the hierarchical culling*

10. Conclusion and Future Work

In this paper, we presented the design and implementation of a fully data-oriented Entity-Component-System architecture, tightly integrated with a highly optimized GPU-driven rendering and hierarchical culling pipeline. The core foundation: a segmented compact storage model utilizing Static-Dynamic-Stream partitioning, combined with bitflag-based change tracking, sparse-set true $\mathcal{O}(1)$ indexing, and a zero-overhead C++ design collectively ensures maximum cache locality, SIMD-friendly parallel processing, and strictly minimized CPU-to-GPU memory transfers.

Furthermore, we detailed a robust three-stage data-oriented model pipeline. By leveraging bindless index-based access through the Model and Animation Managers, alongside an eight-bucket indirect draw command system, the architecture avoids traditional binding bottlenecks. The multi-stage, hierarchical Model-Mesh-Meshlet culling pipeline integrates frustum, Hi-Z occlusion, cone, and zero-pixel triangle filtering. Together with dynamic Fast/Slow path selection, 32-thread collaborative compute processing, projection-based dynamic LODs, and a double-indirection material override mechanism, the engine successfully manages millions of dynamically changing entities while maintaining strict real-time performance. Benchmark results clearly validate these architectural choices: our custom solution significantly outperforms traditional ECS libraries in iteration speed, cache-miss ratios, and dispatch overhead, shifting the rendering burden entirely to the GPU.

Looking ahead, our primary future objective is the full integration of a **Workgraph-based hierarchical culling system**. Utilizing explicit shader node chains will further streamline the Model-Mesh-Meshlet transition, virtually eliminating remaining compute dispatch overheads. On the ECS front, priorities include implementing atomic min-max range-tracking and hierarchical windowed change tracking within the Dynamic region. We also aim to refine GPU-side paged memory mapping and efficiently resolve complex Parent-Child spatial hierarchies.

Finally, to foster collaboration and allow the broader computer graphics community to study, utilize, and build upon our findings, the full reference implementation of the Synapse Engine has been made publicly available on GitHub at `https://github.com/TamasPetii/SynapseEngine`.

Acknowledgements

The research has been supported by OTKA K-145970.

References

1. Victor Blanco. Gpu driven rendering. vkguide (Vulkan Guide). `https://vkguide.dev/docs/gpudriven/compute_culling/`.
2. Michele Caini. Entt: Gaming meets modern c++ - a fast and reliable entity-component system. GitHub repository. `https://github.com/skypjack/entt`.
3. Erik Jansson. Gpu driven rendering with mesh shaders in alan wake 2. In *Digital Dragons Conference*, 2024.
4. Arseny Kapoulkine. Approximate bounding sphere projection. `https://zeux.io/`.
5. Arseny Kapoulkine. meshoptimizer: Mesh optimization library that makes meshes smaller and faster to render. GitHub repository. `https://github.com/zeux/meshoptimizer`.