UltraMeshRenderer: Efficient Structure and Management of GPU Out-of-core Memory for Real-time Rendering of Gigantic 3D Meshes

HUADONG ZHANG, Rochester Institute of Technology, USA
LIZHOU CAO, Rochester Institute of Technology, USA and University of Maryland Eastern Shore, USA
CHAO PENG, Rochester Institute of Technology, USA

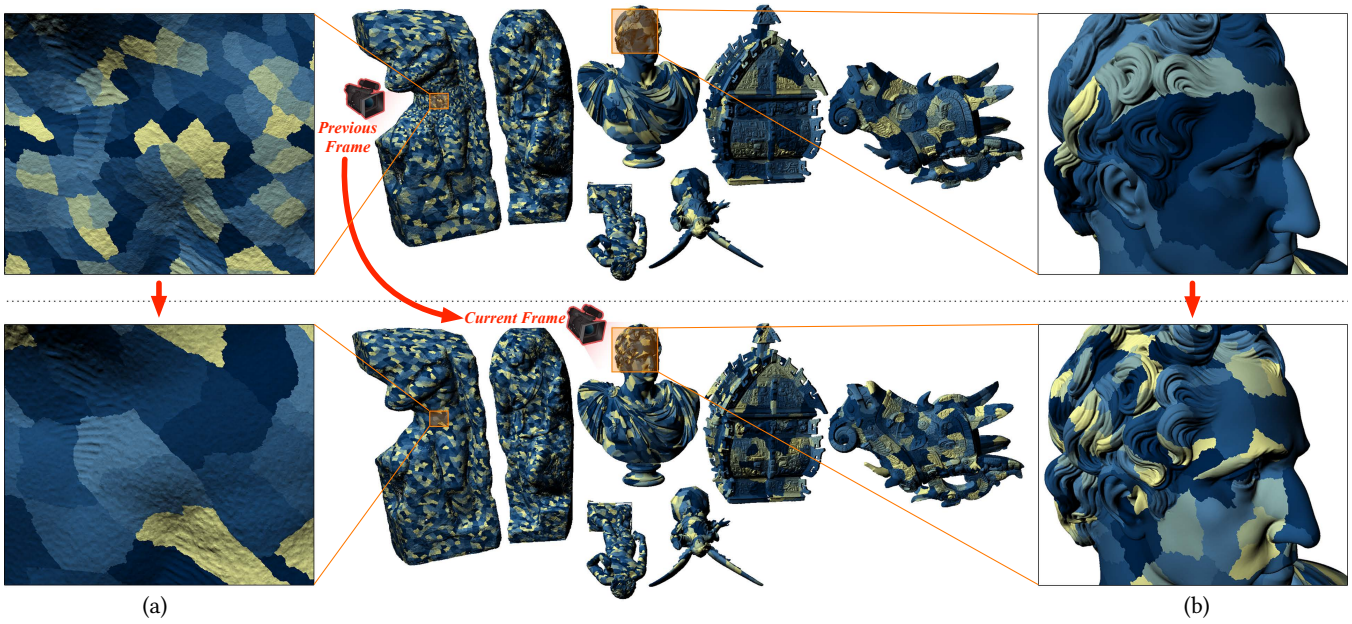

Fig. 1. Rendering of view-dependently selected patches in a complex scene composed of billions of triangles. The top row shows the previous frame, and the bottom row shows the current frame after a camera movement. Patches (nodes) that are topologically connected and near-equal in size (in terms of geometric primitive count) are dynamically selected. Coherent patches already present on the GPU (those used in the previous frame and still needed for the current frame) are reused to render the current frame. Only frame-different data is streamed from CPU to GPU at runtime. (a) shows a close-up of a surface region with reduced geometric detail compared to the previous frame, and (b) shows a region with refined detail.

GPUs can encounter memory capacity constraints, which pose challenges for achieving real-time rendering performance when processing large 3D models that exceed available memory. State-of-the-art out-of-core rendering frameworks have leveraged Level of Detail (LOD) and frame-to-frame coherence data management techniques to optimize memory usage and minimize CPU-to-GPU data transfer costs. However, the size of view-dependently selected data may still exceed GPU memory capacity, and data transfer remains the most significant bottleneck in overall performance costs. To address these, we introduce a new GPU out-of-core rendering approach that includes a LOD selection method that takes into account both memory and coherence constraints and a parallel in-place GPU memory management algorithm that efficiently assembles the data of the current frame with GPU-resident data from the previous frame and transferred data. Our approach bounds memory usage and data transfer costs, prioritizes and schedules the transfer of essential data, incrementally refining the LOD over subsequent frames to converge toward the desired visual fidelity. Our parallel memory management algorithm consolidates frame-different and reusable data, dynamically reallocating GPU memory slots for efficient in-place operations. Hierarchical LOD representations remain a core component, and we emphasize their role in supporting adaptive data transfer and coherence management, characterized by a uniform depth and near-equal patch size at all levels. Our approach adapts seamlessly to scenarios with varying levels of coherence by balancing real-time performance with visual consistency. Experimental results demonstrate that our system achieves significant performance improvements, rendering scenes with billions of triangles in real-time, outperforming existing methods while maintaining consistent visual quality during dynamic interactions.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Parallel computing methodologies**.

Additional Key Words and Phrases: GPU out-of-core, massive model rendering, frame coherence

Authors' Contact Information: Huadong Zhang, Rochester Institute of Technology, Rochester, NY, USA, hz2208@rit.edu; Lizhou Cao, Rochester Institute of Technology, Rochester, NY, USA and University of Maryland Eastern Shore, Princess Anne, MD, USA, lcao@umes.edu; Chao Peng, Rochester Institute of Technology, Rochester, NY, USA, cxpigm@rit.edu.

ACM 1557-7368/2025/8-ART139
https://doi.org/10.1145/3731186

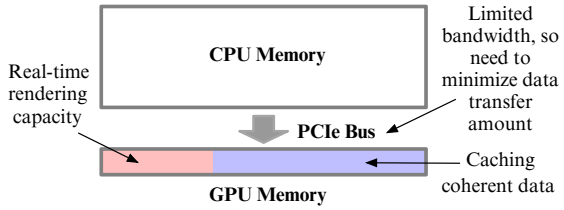

Fig. 2. A GPU-based out-of-core graphics pipeline transfers data from CPU to GPU as frequently as every frame and manages data storage in GPU memory during runtime. According to the CPU-GPU landscape (Section 1.1), a promising approach to increase overall performance is to cache coherent data between frames and transfer only the frame-different data, so that it can minimize the CPU-to-GPU data transfer cost. This is in contrast to traditional graphics pipelines that assume the entire data set fits into GPU memory or require much less frequent transfers.

ACM Reference Format:
Huadong Zhang, Lizhou Cao, and Chao Peng. 2025. UltraMeshRenderer: Efficient Structure and Management of GPU Out-of-core Memory for Real-time Rendering of Gigantic 3D Meshes. *ACM Trans. Graph.* 44, 4, Article 139 (August 2025), 19 pages. https://doi.org/10.1145/3731186

1 Introduction

The rapid growth of 3D data across diverse fields has outpaced the memory capacity of modern GPUs, which struggle to handle datasets exceeding their limits. For scenes with massive 3D meshes, data must be stored in external memory and transferred to the GPU in portions during runtime, a process known as *GPU out-of-core rendering* [Cignoni et al. 2005; Peng and Cao 2012; Sarton et al. 2019; Wang et al. 2013], as illustrated in Fig. 2. Despite advancements in hardware, rendering systems often fail to manage the immense number of vertices and triangles in large scenes. This challenge stems from differences between CPU and GPU memory capacities, limited bandwidth for CPU-GPU data transfer, and finite GPU rendering capabilities. These factors are increasingly noticeable as 3D datasets continue to grow in size and complexity, surpassing the pace of hardware advancements. Addressing this challenge requires new mesh structures and algorithms that optimize CPU-GPU communication, adapt to varying caching needs, and manage geometric data in real-time and therefore enable efficient GPU out-of-core rendering.

1.1 CPU-GPU Landscape for Out-of-Core Rendering

Memory capacity gaps between CPU and GPU.. In today's hardware landscape, large datasets that fit into CPU memory may not fit into GPU memory. In rendering applications, massive 3D meshes that exceed the GPU's memory capacity must be stored in CPU memory, with selective data transfers to the GPU and LOD mesh construction on the GPU occurring as frequently as every frame.

Memory Wall: Limited bandwidth in CPU-GPU communication. GPU memory virtualization techniques (e.g., [Garg et al. 2019; Li et al. 2019]) automate CPU-to-GPU data transfers, but the latency from heavy data streaming requests introduces significant overhead [Garg and Sakharnykh 2021; Sarton et al. 2019]. These techniques often involve transferring large amounts of vertices and triangles for each frame via the PCIe bus, whose bandwidth growth lags behind GPU rendering performance. This issue, known as the memory wall problem [Gholami et al. 2024; Wulf and McKee 1995], severely impacts frame rates. For example, transferring 24 GB of data to an RTX 3090 GPU using a PCIe 5.0 x 16 interface (64 GB/s bandwidth) takes approximately 375 ms. This far exceeds the per-frame time required for real-time rendering, which ranges from 33.3 ms to 11.9 ms (30-90 fps).

Rendering with Prefetched Data in GPU Memory. Hardware benchmarks conducted by GPU manufacturers show that modern GPUs can cache more geometric primitives than they can render into pixels in real-time [NVIDIA 2015, 2020]. This disparity often leaves some GPU memory underutilized. Analogous to a prefetch buffer or cache memory, this unused space can be repurposed to store geometric primitives that are not immediately needed by the current frame but are prefetched for future frames. This strategy minimizes the amount of data transferred during each frame operation, reducing latency and improving overall rendering efficiency.

1.2 Contributions

As illustrated in Fig. 3, this work leverages frame-to-frame coherence and uses available prefetch buffer to cache coherent data. It includes the algorithm of LOD selection that takes into account **both memory and coherence constraints**. When data requirements exceed the limits set by these constraints, the algorithm prioritizes visually important subsets to maintain smooth performance while retaining visual fidelity.

To align data access and LOD selection in both CPU and GPU memory, we introduce a hierarchical LOD structure with **uniform depth** and **near-equal node size**, which streamlines memory allocation and access patterns, facilitating efficient coordination between CPU and GPU. For runtime data management, a novel **in-place parallel algorithm** is developed to consolidate frame-different and reusable data for rendering the current frame. The algorithm is adaptive to varying levels of data coherence, efficiently handling both high-coherence and low-coherence scenarios.

2 Related Works

This section reviews technical considerations in out-of-core rendering, including the adoption of hierarchical data structures to represent massive 3D datasets and the exploitation of coherence for data management in order to speed up overall performance. These considerations play an important role in the efficacy of out-of-core rendering, as they shape data access patterns and locality, determine data transfer volumes, and optimize memory utilization.

2.1 Hierarchical Structures for Adaptive LOD Selection

Out-of-core rendering approaches have traditionally relied on spatial mesh hierarchies, such as Adaptive TetraPuzzles [Cignoni et al. 2004], Quick-VDR [Yoon et al. 2004], Far Voxels [Gobbetti and Marton 2005], KD-trees [Dietrich et al. 2007], and HLODs [Derzapf et al. 2010; Li 2023]. While spatial hierarchies facilitate view-dependent LOD selection via hierarchical traversal, they often disrupt the mesh's topological continuity. This is because spatial partitioning

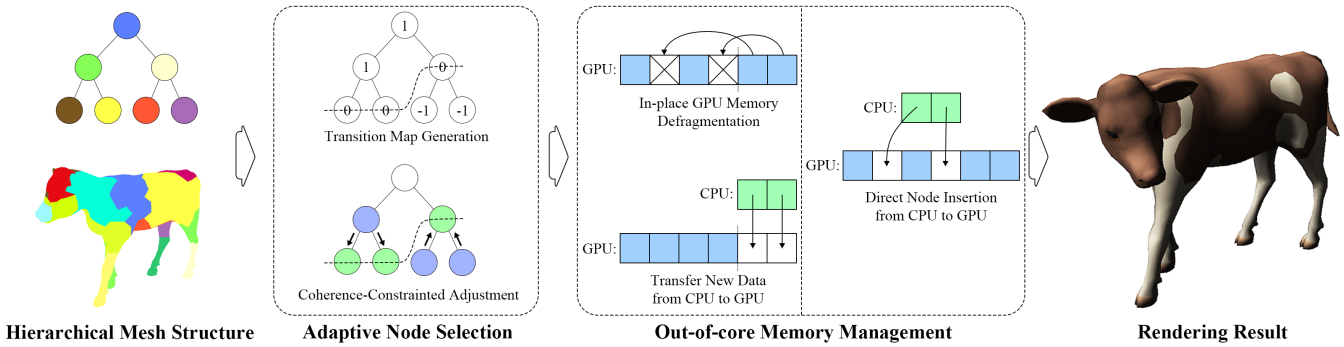

Fig. 3. A high-level overview of our approach. The original mesh will be processed into a hierarchical mesh structure. During runtime, an adaptive node selection method will be executed to select nodes based on the camera's properties. Through switching between two out-of-core memory management modes: in-place defragmentation and insertion, the nodes will be transferred and organized in the GPU memory for rendering.

may cause continuous surface regions to be simplified differently, resulting in visible gaps or inconsistencies in the rendered mesh.

Structures like halfedge [Isenburg and Gumhold 2003; Mäntylä 1987] and sparse matrices [Krüger and Westermann 2005; Wang and Chen 2023; Zayer et al. 2017] efficiently represent incident and adjacency relationships in array-like formats, making them suitable for retained-mode rendering [DiCarlo et al. 2014; Dupuy and Vanhoey 2021; Zayer et al. 2017]. While these structures can capture mesh locality and enable primitive simplification and refinement for dynamic LOD adjustments, their high memory consumption often reduces the effectiveness of caching and data transfers.

Part-based methods [Rodrigues et al. 2018] segment a mesh into meaningful parts while preserving topological connectivity, making them suitable for applications like anatomy classification. However, they are not well-suited for creating mesh structures and have limited compatibility with continuous LOD selection or parallel processing architectures. Additionally, these methods often result in significant variations in the number of primitives between parts.

Patch-based structures (e.g., [Feng et al. 2014; Mahmoud et al. 2021; Zhang and Peng 2025]) encode local topological continuity within surface regions, providing better alignment of LOD changes with mesh topology and more coherent visuals compared to spatial partitioning structures. Region growing and clustering methods are commonly used for patch creation, but uneven vertex distribution across patches often leads to imbalanced GPU thread block workloads. A GPU-based segmentation method [Zhang et al. 2023] used spherical parameterization to balance patch sizes. However, their approach is limited to genus-zero surface meshes and struggles with large, non-watertight meshes or those with high-curvature features, making it less practical for complex scenarios.

Drawing inspiration from graph partitioning, techniques like Kernighan–Lin [Karypis and Kumar 1998], PT-Scotch [Chevalier and Pellegrini 2008], and Zoltan [Boman et al. 2012] have been applied to mesh partitioning. These graph partitioning techniques leverage vertex connectivity and path lengths to carve out partitions, focusing on edge-cut and boundary smoothness criteria. However, they often require additional dependency structures, involve irregular data access behaviors, and perform global index searches at a thread level, making them unsuitable for independent node processing.

Current hierarchical structures face limitations, including imbalances, discontinuities, and inefficiencies for parallel architectures. To address these challenges, this work introduces a balanced hierarchical mesh structure comprising sub-mesh nodes with near-equal geometric primitive counts while maintaining surface locality. Using this structure, a GPU-optimized LOD selection method is proposed, designed to handle limited GPU memory and minimize CPU-GPU transfer overhead.

2.2 Frame-to-frame Coherence and GPU Out-of-core Data Management

Frame-to-frame coherence refers to the fact that a sequence of consecutive frames are likely to contain a high degree of consistency if the time difference between frames is small [Groller and Purgathofer 1995; Tost and Brunet 1990]. This concept has been used to optimize the performance by reusing information from previous frames to render or synthesize subsequent frames more efficiently, such as accelerating hidden/occluded surface removal [de Lucas et al. 2019; Hubschman and Zucker 1982] and facilitating shading reuse across adjacent frames [Mueller et al. 2021; Scherzer et al. 2010, 2012; Yang et al. 2023].

For rendering massive 3D models, the concept of frame-to-frame coherence has been employed for out-of-core data management to facilitate data reuse on the GPU and therefore to reduce the cost of data assignment and scheduling [Xu et al. 2022; Yoon et al. 2022]. One typical scenario for maximizing data reuse is to identify and transfer frame-different data to the GPU, and then consolidate them with the reusable portions of the data that are already on the GPU (e.g., [Dong and Peng 2023; Liu et al. 2012; Peng and Cao 2012; Xue et al. 2016]). We summarized the state-of-the-art approaches for conducting such out-of-core management.

Scatter transfer and insertion. Frame-different data could be composed of scattered data elements and lacking patterns in the data repository in CPU memory. A straightforward method to transfer them to the GPU involves multiple memory copy calls. This involves iterating over the scattered data elements and invoking a memory copy operation for each element. This method supports in-place

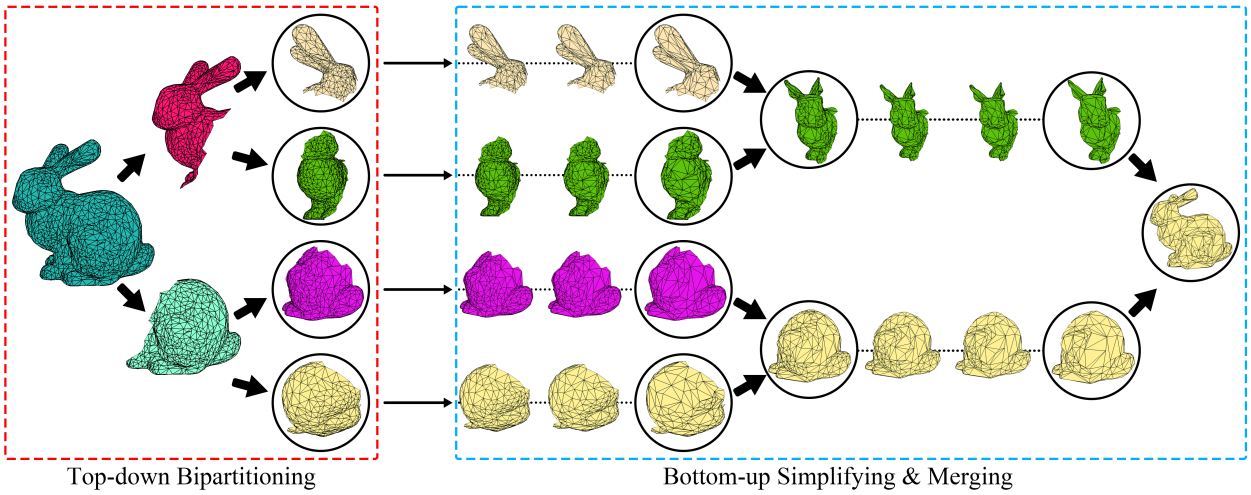

Fig. 4. An illustration depicting the construction of a hierarchical mesh structure. The original mesh is partitioned into two sub-meshes, ensuring a balance in the number of vertices. This partitioning process continues until the leaf level is reached. From the leaf level onward, sibling sub-meshes are simplified and merged, forming the sub-mesh at the parent level. This iterative process continues until reaching the root node. As a result, all nodes in the hierarchy contain a balanced number of vertices and can represent a range of detail levels.

memory management with careful page size selection [Ausavarungnirun et al. 2017] or by developing application-specific solutions [Li et al. 2019; Ponchio 2009]. The virtualized geometry and streaming implementation in Unreal Engine's Nanite [Brian Karis 2021] is similar to the scatter transfer and insertion method, where the GPU requests data whenever the rendering frame needs, and the CPU asynchronously fulfills the requests.

Sequential transfer and merging. According to the design of GPU architectures, a preferable way for transferring frame-different data is first collecting the data elements into a continuous memory space in CPU memory and then sending them as an array to the GPU with a single memory copy call. Current methods require additional memory allocation, typically involving a separate contiguous memory allocation on the GPU to receive the frame-different data and an out-of-place sorting process to merge them with the array of the reusable data into a new array of data (e.g., [Dong and Peng 2023; Maurya et al. 2023; Springer and Masuhara 2019a,b]). However, allocating additional memory leads to GPU memory wastage, as the arrays storing frame-different data and reusable data act as temporary memory spaces and are not active during rendering. More importantly, out-of-place sorting requires relocating all data elements, which can be computationally intensive.

This work presents an efficient in-place parallel data management algorithm to retain memory compactness for rendering at every frame, which leverages the GPU-friendly characteristics of our mesh structure.

3 Constructing Balanced Hierarchical Mesh Structure

We designed a balanced hierarchical mesh structure with a consistent degree of 2 for the nodes at each level and near-equal node sizes. Such structural regularity facilitates quick locating and querying, attributed to consistent indexing offsets. In addition, this structure supports adaptive view-dependent LOD selection and continuous LOD transitions within each node, optimizing GPU resource use by focusing on areas closest to the viewer and minimizing abrupt visual changes.

3.1 Top-Down Bipartitioning and Bottom-Up Merging

To create the balanced hierarchical mesh structure, we initiate a top-down bipartitioning of the mesh, as shown on the left of Fig. 4. The top-most node is the original mesh, and it is partitioned into two sub-meshes. The two sibling sub-meshes are balanced on vertex counts. This partitioning process continues iteratively until the desired depth is achieved, at which point it has reached the leaf level. We utilized METIS [Karypis and Kumar 1997] to perform this iterative partitioning process. It uses the multi-level bisection method and Kernighan-Lin algorithm to refine the partitioning result, with considerations on the relationship between vertices and faces and edge-cuts across the sub-meshes. We enabled the "-contig" option in METIS, so that each partitioning operation can produce two continuous sub-mesh surfaces.

Existing hierarchical mesh structures often contain nodes of varying sizes (e.g., [Derzapf and Guthe 2012; Gobbetti and Marton 2005; Yoon et al. 2004]). When nodes have very different triangle and vertex counts, managing them during runtime becomes challenging. For instance, if the memory slot released from a node is smaller than the size of a new node intended for this slot, additional memory must be allocated, or the current memory usage rearranged, to accommodate the new node size. Such memory management operations can lead to serious fragmentation or large memory rearrangement overhead.

Near-equal node sizes allow for allocating uniform memory slots for quick node access, removal, and insertion, while maintaining a high rate of memory utilization. In this work, starting from the leaf level, as shown on the right of Fig. 4, two sibling nodes at the

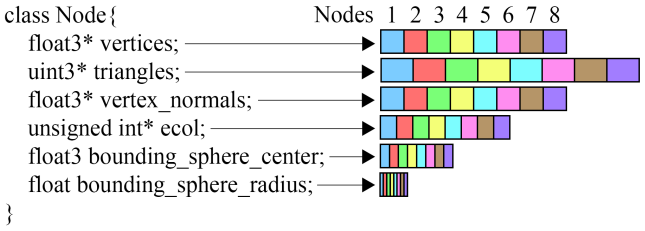

Fig. 5. The storage of nodes converted from the AoS to SoA format.

current level are simplified and then merged to form the node at the next higher level to provide a coarser surface detail for a larger area.

In particular, we employed Quadric Error Metrics (QEM) [Garland and Heckbert 1997] to simplify sub-mesh patches, with a constraint that the boundary edges of a patch are not collapsible. We aimed to reduce the vertex count in the sibling nodes by half at each level. Upon merging into a new node at the parent level, each should ideally retain the same number of vertices as a leaf node. This bottom-up simplifying and merging process continues until reaching the root. As a result, the sub-mesh patch of each node is able to represent a range of details, extending from the vertex count for the finest detail to half of it.

Ideally, we seek to ensure an equal number of edge collapses for every node. However, despite some attempts [Rahimian et al. 2013; Zhang et al. 2023], this ideal goal is not achievable due to the constraint on patch boundaries, the occurrence of cracks (Section 3.2), or poor mesh conditions such as holes on the surface. In our approach, non-significant differences on the primitive counts can still result in acceptable memory usage and efficient utilization of GPU computing resources (see Section 7.1). Each node presents its data fields and can be independently selected and transferred from CPU to GPU. Each node retains its capability for adaptive LOD sub-mesh construction on the GPU in response to view changes. This ensures locality and avoids uncoalesced mapping between local and global index space. During runtime when the GPU schedules for the executions, streaming multiprocessors assigned with slightly smaller nodes would complete their execution earlier, thereby being freed to work on other nodes.

The nodes are originally stored in an array-of-structures (AoS) format during the construction of hierarchical mesh structure. We converted them into a structure-of-arrays (SoA) format when rendering the meshes. As shown in Fig. 5, the SoA format reorganizes the nodes' vertices, triangles, and edge collapses into separate arrays with consistent per-node offsets.

3.2 Rebuilding to Address Boundary Issues

During the bottom-up reconstruction process, the quality of simplified sub-meshes could be negatively affected by the constraints on the fixed boundary vertices. Preventing collapsing boundary vertices avoids the occurrence of cracks [Yoon et al. 2004], but it would result in a dense cluster of vertices around the boundaries, which may stop the clusters from being further simplified. Our approach is crack-free by enforcing a constraint that keeps boundary

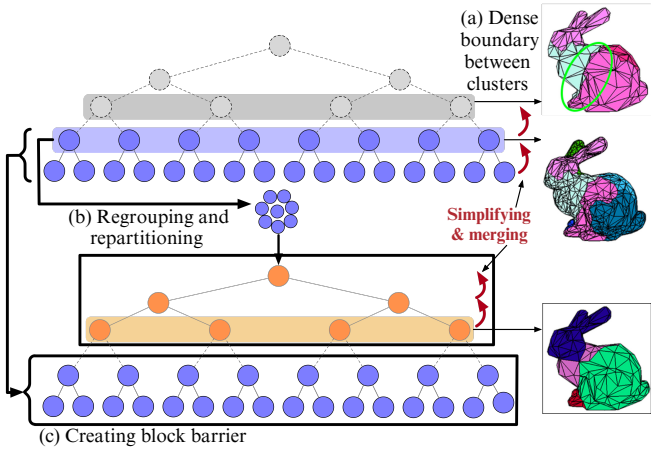

Fig. 6. An example illustrating (a) the occurrence of dense boundaries during the bottom-up simplifying and merging process, and (b-c) the regrouping and repartitioning process to resolve it.

edges non-collapsible during simplification. To address the dense boundaries issue, the affected levels are regrouped and repartitioned so that these boundaries are treated as inner edges. In particular, at a level during bottom-up merging, we designed a method to predict the occurrence of dense boundaries for the upper level. We monitor the cost of edge collapsing using QEM during mesh simplification [Garland and Heckbert 1997]. Mesh simplification terminates when the cost exceeds a predefined threshold. After simplifying all nodes in the current layer, if any nodes terminate simplification due to exceeding the cost threshold, we rebuild the level. This involves regrouping all simplified nodes into a single mesh (the root level) and repartition it into the same number of nodes. The newly generated nodes will replace the previous nodes at this level. Since METIS aims to minimize the edge-cut between meshes, the newly generated nodes won't have the dense boundaries issue because the number of boundary vertices is minimized. Thereby, the mesh structure will be cut from this level into two different blocks. Fig. 6 illustrates this process. This regrouping and repartitioning process may happen multiple times, so that the mesh structure may be divided into multiple blocks.

The existing method to handle dense boundaries issue involves rebuilding only the two nodes that have a dense inner boundary. However, once an inner dense boundary is detected, it is highly likely that the outer boundaries also contain numerous vertices and are on the verge of experiencing the same problem. These outer dense boundaries will manifest and need to be resolved at higher levels, requiring the tree to be cut at multiple levels, which is not ideal for continuous LOD algorithms. On the other hand, in our algorithm, once a dense boundary is detected, the entire layer is optimized by rebuilding, establishing a full connection between the new nodes of the current and upper layers. The number of cuts is limited to a small number of layers, which won't affect the robustness and performance of the LOD algorithm.

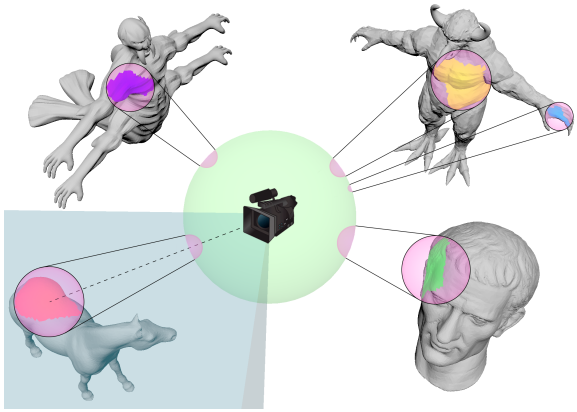

Fig. 7. A node selection example. The bounding sphere of each node is projected onto the tangent plane of the "near sphere" of the camera. The selection of the desired LOD result is determined based on the screen coverage ratio, which is the percentage of the projection size over the screen size on the near plane of the camera.

4 Adaptive Node Selection

4.1 Transition Map (*t_map*) for In-Between LOD Selection

State-of-the-art LOD selection methods for GPU out-of-core rendering incorporate view dependency by determining a desired subset of nodes per frame [Derzapf and Guthe 2012; Hu et al. 2009; Lambert et al. 2018], selected from the pool of nodes in CPU memory and transferred to the GPU. Our view-dependent selection criterion uses screen-ratio metrics [Dalei et al. 2022; Zhu et al. 2010], covering a 360-degree range, selecting both visible nodes for rendering and hidden nodes that may become visible in future frames, as shown in Fig. 7.

During camera movements, reduced visual sensitivity to details, supported by perceptual-driven theories [Bartz et al. 2008; Lin and Kuo 2011], suggests that rendering at the target LOD selection result is not always necessary. Instead, prioritizing smooth performance can maintain perceived visual fidelity [Luebke and Hallen 2001; Petrescu et al. 2023]. To achieve this, our method introduces a *transition map* (*t_map*), which guides the selection along a pathway in the node hierarchy toward the desired LOD result, particularly under the CPU-GPU bandwidth constraints (see Section 4.2). In such cases, *t_map* allows for selecting nodes on the pathway to reduce the overheads and ensure smooth rendering transitions, avoiding abrupt changes that could impact performance or cause quality inconsistencies between frames.

t_map is constructed during the process of determining the desired LOD result. As shown in the top image of Fig. 8, nodes are labeled based on the selection criteria: -1 if they need to be merged to form a coarser version at an upper level, 1 if they need to be split for a finer version at a lower level, and 0 if they meet the desired selection. Algorithm 1 details the steps. Initially, all leaf nodes are labeled as 0, and all other nodes as 1 (lines 2–4). A node's height, defined as its distance from the leaf nodes, determines its hierarchical level. Starting from the level above the leaf nodes, if both child nodes of a parent meet the selection criteria, the parent is labeled 0, and the children are relabeled as -1 (lines 6–14). This process iterates upward until reaching the root node.

Algorithm 1 Perform LOD Selection and Construct **t_map**

```
1:  procedure ViewDepSel(t_map)
2:      for each i^th node in the mesh hierarchy in parallel do
3:          t_map[i] ← node_i is in leaf level ? 0 : 1;
4:      end for
5:      Bottom-up traverse to evaluate nodes with screen-error metrics in
        parallel per-level;
6:      for d ← 1 to maxHeight − 1 do
7:          for each i^th node in the level at height d in parallel do
8:              if Screen coverage ratios of both child nodes of node_i are below
                the minimum ratios then
9:                  t_map[node_i.left_child_idx] ← −1;
10:                 t_map[node_i.right_child_idx] ← −1;
11:                 t_map[i] ← 0;
12:             end if
13:         end for
14:     end for
15:     return t_map;
16: end procedure
```

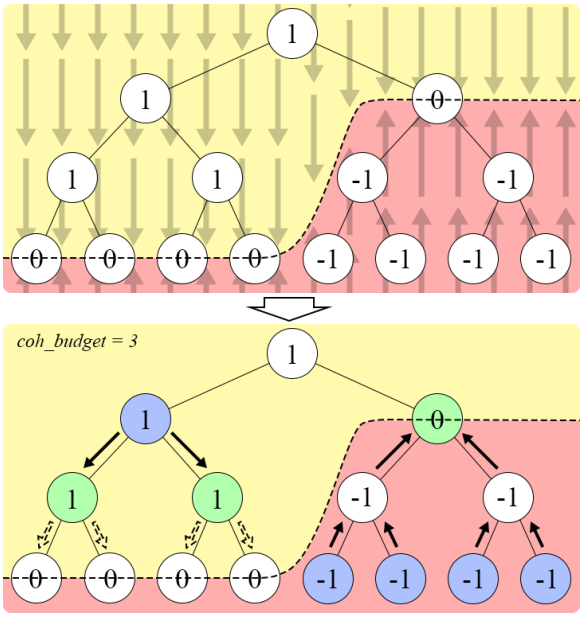

Fig. 8. A visualized example of the transition map. The nodes in the red area need to be merged, and the nodes in the yellow area need to be split. The nodes in the dashed line represent the desired LOD selection result. In the case of $coh\_budget = 3$, our method executes node merging and splitting operations towards the desired LOD as much as possible. The blue nodes represent the LOD selection result of the previous frame. The green nodes represent the LOD selection result of the current frame after node actions. The solid arrows are the node actions executed in this frame. The dashed arrows are the node actions not executed due to the limitation of the coherence budget.

Algorithm 2 Coherence-constrained adjustment to selection results

```
1: procedure SelectionAdjustment(t_map, coh_budget, node_F^c, node_F^p)
2:     copy all elements in node_F^p to node_F^c in parallel;
3:     total ← 0;
4:     while total < coh_budget do                          ▷ Perform node merging
5:     coh_adjust ← all the elements equal to 0 in parallel;
6:     for each i^th node in the mesh hierarchy in parallel do
7:         if its children in t_map = -1 & in node_F^c are selected then
8:             if its children in node_F^p are selected then
9:                 coh_adjust[i] ← 1;
10:            else if its children in node_F^p are not selected then
11:                coh_adjust[i] ← −1;
12:            else
13:                coh_adjust[i] ← 0;
14:            end if
15:        end if
16:    end for
17:    coh_adjust_sum ← the prefix-sum of coh_adjust in parallel;
18:    Find the max k where coh_adjust_sum[k] <= coh_budget − total in parallel;
19:    for each i ∈ [0, k] in parallel do
20:        if its children in t_map = -1 & in node_F^c are selected then
21:            node_F^c[node_i.left_child_idx] ← 0;          ▷ deselect
22:            node_F^c[node_i.right_child_idx] ← 0;         ▷ deselect
23:            node_F^c[i] ← 1;                              ▷ select
24:        end if
25:        total+ = coh_adjust[i];
26:    end for
27:    if no node merging is detected then
28:        break;
29:    end if
30:    end while
31:    while total < coh_budget do                          ▷ Perform node splitting
32:    coh_adjust ← all the elements equal to 0 in parallel;
33:    for each i^th node in the mesh hierarchy in parallel do
34:        if this node in t_map = 1 & in node_F^c is selected then
35:            if this node in node_F^p is selected then
36:                coh_adjust[i] ← 2;
37:            else
38:                coh_adjust[i] ← 1;
39:            end if
40:        end if
41:    end for
42:    coh_adjust_sum ← the prefix-sum of coh_adjust in parallel;
43:    Find the max k where coh_adjust_sum[k] <= coh_budget − total in parallel;
44:    for each i ∈ [0, k] in parallel do
45:        if this node in t_map = 1 & in node_F^c is selected then
46:            node_F^c[node_i.left_child_idx] ← 1;          ▷ select
47:            node_F^c[node_i.right_child_idx] ← 1;         ▷ select
48:            node_F^c[i] ← 0;                              ▷ deselect
49:        end if
50:        total+ = coh_adjust[i];
51:    end for
52:    if no node splitting is detected then
53:        break;
54:    end if
55:    end while
56:    return node_F^c;
57: end procedure
```

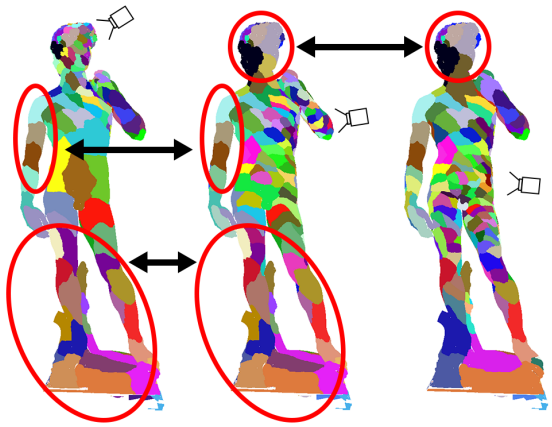

Fig. 9. The example illustrating the view-dependent node evaluation results over successive frames. Nodes at lower levels of the hierarchy are selected for the regions close to the camera. The circled regions include the coherent nodes between the frames, which are reusable.

4.2 Coherence-Constrained Adjustment

Frame-to-frame coherence tracks differences between the current and previous camera frames, transferring only the changed data to the GPU and merging it with reusable data for rendering [Dong and Peng 2023; Peng and Cao 2012]. The level of coherence varies with camera movement. As shown in Fig. 9, significant camera movement can reduce overlap between consecutive LOD selections, increasing frame-different data volume and potentially exceeding transfer bandwidth or making merging operations costly.

By leveraging *t_map*, we developed a coherence-constrained adjustment algorithm (Algorithm 2) to control coherence levels and maintain a stable frame rate while minimizing visual quality loss. The algorithm introduces a coherence budget (*coh_budget*), defined as the maximum number of nodes transferable from CPU to GPU between frames. If the node transfer load for the desired LOD result exceeds the *coh_budget*, the algorithm utilizes *t_map* to control the transfer load within the budget by performing node merging or splitting actions. A variable *total* tracks the number of new (frame-different) nodes and is compared against *coh_budget* while adjusting the selection through *t_map*.

In each frame, nodes in the mesh hierarchy have been flagged as selected or not in the previous frame, represented by the array $node\_F^p$. The algorithm begins by copying $node\_F^p$ to the current frame's flag array ($node\_F^c$), and then merges or splits nodes based on the *t_map* labels. Nodes labeled $t\_map[i] = 0$ represent the desired LOD result. The algorithm performs node merging first (lines 4-30), so that GPU memory is freed up to enable subsequent node splitting (lines 31-55). The image at the bottom of Fig. 8 illustrates an example of this coherence-constrained adjustment.

To ensure *total* remains within the *coh_budget*, an integer array *coh_adjust* tracks the number of new nodes created by merging or splitting a node. A prefix sum of *coh_adjust* determines the maximum node index (lines 17-18 and 42-43), limiting operations to nodes between indices 0 and this maximum (lines 19-26 and 44-51).

The execution of the algorithm stops when $total$ reaches the budget, or no merging or splitting is detected.

4.3 Memory Constraint and Selection Across Blocks

It happens that the amount of desired nodes exceeds the memory sector allocated for rendering. Before the coherence-constrained adjustment, a pruning process is executed on t_map to limit the number of desired nodes. All nodes are processed in parallel iteratively during pruning. Nodes with lower visual importance are prioritized for pruning to minimize the impact on rendering quality. Specifically, the desired sibling nodes are replaced with their parent nodes, merging into a coarser representation. We repeat this pruning process on t_map until the final count of the desired nodes is within the capacity of the memory sector allocated.

In addition, it is possible that the node selection result includes nodes spanning multiple blocks. As explained in Section 3.2, the blocks in the mesh structure are barriers that force the entire mesh to be retopologized. They avoid long, dense boundaries and ensure nodes at all levels can be merged and simplified to near-equal size. However, nodes across blocks do not ensure surface continuity.

If the desired nodes cross multiple blocks, a target block is determined based on which block contains the most desired nodes. On the t_map, desired nodes in upper blocks are refined to the top level of the target block, while desired nodes in lower blocks are pruned to the bottom level of the target block. After this, if the number of desired nodes exceeds the node budget, the nodes at lower levels in the target block continue to be pruned until the number is within the budget. If the number of desired nodes is still over the budget after they have all been pruned to the top level of the target block, the set of desired nodes will be replaced by the bottom-level nodes in the upper block, and we repeat the pruning process if necessary until the number of desired nodes is within the budget.

5 In-place GPU Data Management

Our approach allocates a fixed-size memory sector corresponding to the current frame. By leveraging the property that the memory allocated for every node in our mesh structure is nearly equally sized, this memory sector is sliced into an array of memory units, each corresponding to holding a node and accessing data and indices locally. As mentioned in Section 3, the vertices and triangles in each node can be accessed and rendered independently since they remain in their locality.

When the GPU receives new nodes, the memory sector can be extended using dynamic memory allocation or buffer resizing [Schäfer et al. 2013; Yang et al. 2010]. However, these techniques can result in frequent memory reallocations, particularly problematic when rendering large and complex scenes where the memory consumption accumulates and may exceed the available GPU memory.

Our data management method updates the nodes within the fixed-size memory sector. This section first describes the identification of frame-different nodes ($\Delta fnodes$). Memory units designated for nodes used in rendering the previous frame — but not selected for the current frame — will be marked as released and made available for the nodes of the current frame. Then, we present a simple and efficient algorithm to determine in a parallel fashion the target GPU memory location among these released memory slots that the nodes should be moved to. This algorithm is in-place, suitable for both moving nodes from CPU to GPU and moving them within the GPU, adaptive to both high and low coherence cases.

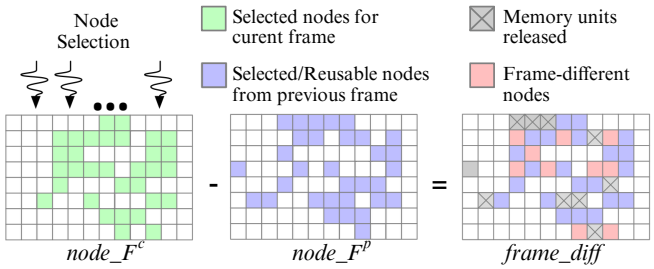

Fig. 10. Node selection example showing the frame difference by subtracting the node selection result of the previous frame ($node\_F^p$) from the result of the current frame ($node\_F^c$). The frame-different nodes will be fetched from the CPU and the memory units released by removing unnecessary nodes of the previous frame will be filled with the nodes of the current frame.

5.1 Finding Memory Utilization Changes Due to Frame Difference

The node selection result (Section 4) is in array format, with a size equal to the total number of nodes. Each element in this array corresponds to a node ID in the mesh structure and stores a binary code to represent whether the node is to be displayed or hidden. Initially, all memory units on the GPU are empty. For the first frame, all nodes labeled as "display" in the selection result are transferred to the GPU and stored into continuous memory units in increasing index order. Additionally, two indexing arrays are created to track node presence in GPU memory. The first array, denoted as $node\_to\_mem$, tracks the memory location of each node, with each element corresponding to a node ID and the value indicating the index of the memory unit it currently occupies. If the node does not exist in GPU memory, the value is set to an invalid memory index. The second array, denoted as mem_to_node, tracks which node is stored in each memory unit, with each element corresponding to a memory unit index and the value indicating the node ID. If the memory unit is empty, the value is set to an invalid node ID.

During runtime, we maintain two node selection results simultaneously: one for the current frame ($node\_F^c$) and the other inherited from the previous frame ($node\_F^p$). They are swapped at the conclusion of each frame. A parallel subtraction operation is performed over the two selection results, resulting in an array denoted as $frame\_diff$, which indicates changes in memory utilization between the current and previous frames, as shown in Fig. 10. In $frame\_diff$, a label of 1 denotes a node that needs to be added to the GPU, while -1 indicates a node that should be removed.

For example, in the array $[0, 0, 1, 0, 0, -1, -1]$, $node2$ needs to be added, $node5$ and $node6$ need to be removed from the GPU. Other nodes are either already in GPU memory and to be reused, or they are not needed and do not need to be removed or added.

Algorithm 3 Converting frame-difference to the changes in memory utilization

```
1: procedure ConvertFrameDifftoMemUtil(frame_diff, mem_util, mem_to_node, node_to_mem)
2:     for each i^th memory unit in the memory sector in parallel do
3:         if mem_util[i] == 1 then
4:             nid ← mem_to_node[i];
5:             node_to_mem[nid] ← i;
6:         end if
7:     end for
8:     for each i^th node index in frame_diff in parallel do
9:         mid ← node_to_mem[i];
10:        if frame_diff[i] == −1 then
11:            mem_util[mid] ← 0;
12:            mem_to_node[mid] ← −1;
13:        end if
14:    end for
15:    return mem_util;
16: end procedure
```

A better representation of changes in memory utilization is needed in order to provide instance access to memory units marked as available on the GPU. Algorithm 3 shows the steps to use an array, denoted as *mem_util*, to represent and compute memory utilization. The size of this array is equal to the maximum number of memory units that the GPU is capable of allocating or that the user arbitrarily defines. The value of each element in *mem_util* is either 1 or 0, indicating whether the corresponding memory unit currently stores a reusable node.

After consolidating the frame-different and reusable nodes in the memory sector using *mem_util* (Section 5.2), the arrays *node_to_mem* and *mem_to_node* are updated to reflect the updated node presence in GPU memory.

5.2 Determining Target Memory Locations for Efficient Node Movement

Before moving any node, we need to determine the target memory units for the nodes from the available memory units, which are labeled as 0 in *mem_util*. Due to the utilization of frame-to-frame coherence, the indices of available memory units are sparse in *mem_util*. Existing methods based on stream compaction [Billeter et al. 2009; Dong and Peng 2023; Springer and Masuhara 2019b] often require extra buffers or memory to store intermediate results during the scan and compact operations, consolidating renderable primitives into a new memory space [Dong and Peng 2023; Sarton et al. 2019]. This reliance on additional GPU memory can significantly impact memory efficiency, especially when dealing with large 3D meshes or when hardware has limited memory capacity.

Our algorithm enables in-place data management, eliminating the need for additional GPU memory when consolidating frame-different and reusable nodes. When coherence is high, frame-different nodes can be directly inserted from the CPU to the identified target memory units on the GPU. Since the target memory units are not continuously located, insertion must occur through multiple memory copy calls, transferring one node at a time. Even with multiple memory copy calls, data management can still be done efficiently when coherence is high due to the small number of frame-different nodes.

However, when coherence is low, transferring all the nodes in a chunk becomes preferable. To maintain the in-place management strategy, nodes already on the GPU must be moved to allow a continuous set of available memory units in the memory sector to receive the new nodes. This process is akin to defragmentation, but it aims to move reusable nodes within the memory sector so that the available memory units are rearranged into a contiguous memory chunk. This process may occur as frequently as every frame. Our algorithm accommodates both coherence cases. We describe the algorithm for handling the low-coherence case to defragment for available memory units on the GPU, and then explain its adaptation for the high-coherence case and assisting for node collection on CPU.

5.2.1 Aiming for GPU defragmentation for low-coherence case. The memory sector is divided into two memory zones: the *CycleZone* and *ForgeZone*. The CycleZone, starting at the beginning of the memory sector, will be used to hold all reusable nodes from the previous frame. Its size is determined by the number of reusable nodes counted from the '1's in *mem_util*. The ForgeZone covers the remaining section and will be used to receive the frame-different nodes. Our algorithm, as outlined in Algorithm 4, returns an *instruction* that informs parallel execution about which memory unit in the CycleZone each reusable node in the ForgeZone should move to.

Algorithm 4 Find memory locations for moving nodes

```
1: procedure FindMemLocations(mem_util, lookup range (s))
2:     for each i^th element within the range [0, s − 1] in mem_util in parallel do
3:         mem_util[i] ← mem_util[i] == 0 ? 1 : 0;
4:     end for
5:     Perform parallel prefix-sum into mem_util;
6:     for each i^th element (i > 0) in mem_util (prefix-summed) in parallel do
7:         if mem_util[i] > mem_util[i − 1] then
8:             instruction[mem_util[i] − 1] ← i;
9:         end if
10:    end for
11:    return instruction;
12: end procedure
```

After moving these reusable nodes to the CycleZone, the ForgeZone will be completely available to receive the chunk of new nodes from the CPU with a single CPU-to-GPU memory copy.

For the input of Algorithm 4, the lookup range *s* is set to the size of CycleZone. First, we invert the values of the elements within the lookup range in *mem_util* (lines 2-4), so the 1s in the CycleZone signify the locations of available memory units. The 1s in the ForgeZone, which are not altered, still correspond to the memory locations storing the reusable nodes. Then, a parallel prefix-sum operation is applied to the entire array (line 5) to signify the cumulative count of 1s at each position. This operation incurs minimal overhead due to the small size of the *mem_util* array. We identify positions where the accumulated sum values make their first appearances in a parallel fashion (lines 6-10) and record them into the

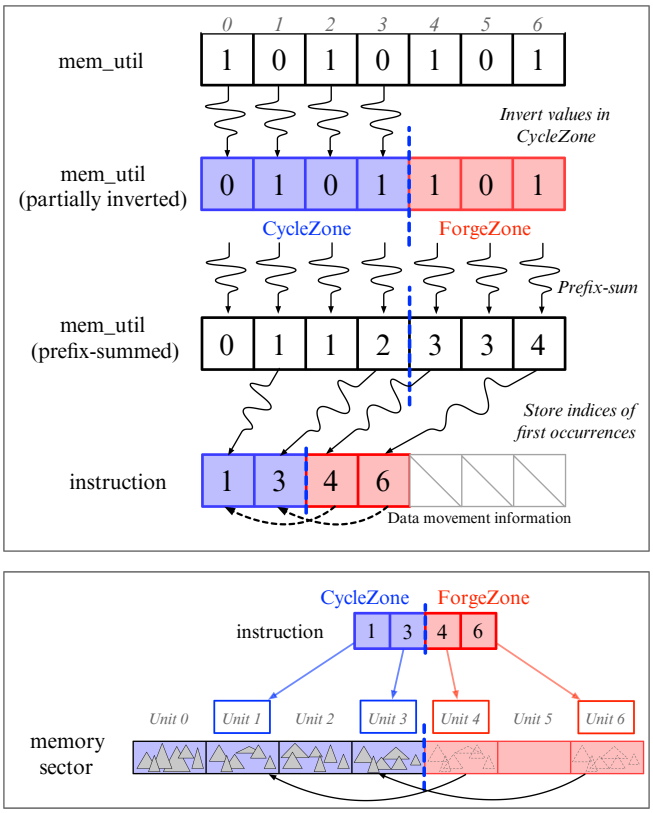

Fig. 11. An example illustrating the process of compacting reusable nodes. For this example, let's assume there are a total of 7 GPU memory units in the memory sector. The values of the elements in the CycleZone of the *mem_util* are inverted. Subsequently, a parallel computation of the prefix-sum is performed over the *mem_util* array, leading to the creation of the *instruction* array. Finally, based on the *instruction* array, the reusable nodes in the ForgeZone of the memory sector are moved to the available memory units in the CycleZone in a parallel fashion.

instruction, continuously and appearing in order. Each element in the *instruction* holds a memory unit index, representing the element holding the value 1 in the partially inverted *mem_util*.

After the inversion (lines 2-4), the counts of 1s in both zones are equal (refer to Appendix). This forms the basis for the *instruction* to establish one-to-one memory index mappings from the ForgeZone to the CycleZone. Let's denote this count as φ, which is half of the last element's value in the prefix-summed, partially-inverted *mem_util* ($\varphi = mem\_util[last]/2$). Thus, the *instruction* can be divided into two equal halves, where each element in the first half denotes an available memory unit in the CycleZone, and each element in the second half denotes the memory location storing a reusable node in the ForgeZone. With this property, we can move φ reusable nodes in the ForgeZone in a parallel fashion. Each GPU thread locates the reusable node at $instruction[\varphi + i]$ and moves it to the available memory unit at $instruction[i]$. Fig. 11 illustrates an example of using this algorithm to find the *instruction* and use it for defragmentation. After that, the chunk of frame-different nodes collected in CPU memory is transferred to the ForgeZone.

5.2.2 Aiming for node insertion for high-coherence case. The memory sector functions as a single memory zone, and the lookup range for Algorithm 4 is set to the size of *mem_util*. Consequently, the *instruction* records all available memory units in the entire memory sector in the order of their appearance. In this operation, the algorithm behaves similarly to stream compaction, but the difference lies in compacting the indices of all available memory units.

5.2.3 Aiming for finding node IDs for collecting them on CPU. Algorithm 4 can be used to find the list of frame-different node IDs (labeled as 1s in $frame\_diff$). To do this, the input *mem_util* is replaced with $frame\_diff$, and the lookup range is set to the size of *mem_util*. In the algorithm, -1s in $frame\_diff$ are treated as 0s. By looping through the list, nodes can be fetched from the mesh structure in CPU memory and inserted directly into the memory sector on the GPU without allocating additional memory.

6 LOD Mesh Construction and Rasterization

Prior to rasterizing the nodes selected for the current frame, our approach performs view-frustum culling for the nodes in parallel. The bounding sphere of each node is tested against the volume of the view-frustum. The nodes that are inside or intersected with the view-frustum will be rendered, and others will be marked as hidden and excluded from the rendering of the current frame.

During the node selection stage (Section 4), each selected node has been determined with a desired LOD, represented as the desired vertex and triangle counts in this node, which is obtained from mapping the coverage ratio interval values between the maximum and minimum vertex and triangle counts of this node. Note that even for these reusable nodes that were retained on the GPU and continue to be used by the current frame, they may receive new LOD desires. Thus, all the to-be-rendered nodes should be reconstructed to adaptive patch versions as alternatives in rasterization. As mentioned in Section 3.1, each node has been pre-processed to embed continuous edge collapses in arrays for sub-mesh simplification. During the runtime, we employed the mesh reformation method [Hu et al. 2009; Peng and Cao 2012], which looks up the pre-recorded edge collapses and constructs a valid mesh topology under the desired vertex and triangle counts. The mesh reformation method is performed at a triangle-level parallelization. When a vertex index of the triangle is larger than the number of selected vertices, the GPU thread looks up the edge collapses and replaces it with the index of the vertex it collapses to. Subsequently, this index is checked again to determine if it is now smaller than the number of selected vertices. If not, the index needs to be replaced with the index of the next vertex it collapses to, until a within-range one is found.

Our approach supports the rendering of characters with skeletal animations and textures. The implementation leverages a GPU-based texture preservation method [Peng et al. 2011], originally applied to objects, which we have adapted for use with patches. In our approach, patches are topologically connected surface regions, and because their topology remains unchanged during animation,

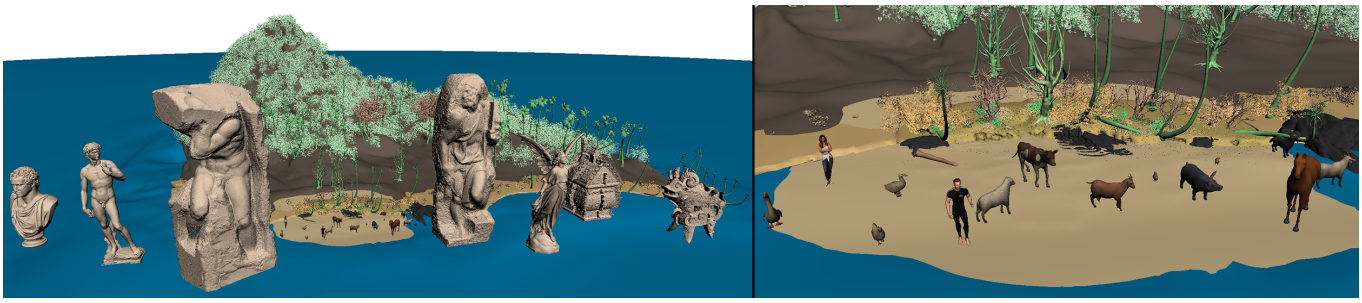

Fig. 12. The screenshots of the combined scene. The left image is a bird's eye view of the scene. The right image is a close view of the animated characters.

the texture-preserving method can treat each patch as an individual object.

We utilized the single-buffer multiple draw call technique to enhance rasterization performance. Instead of issuing a separate draw call for each node, which can incur significant CPU overhead due to buffer switching, we issued draw calls with a single buffer for rasterizing the nodes to be rendered in a single pass.

7 Evaluation

We conducted the experiments on a workstation equipped with an Intel Core i9-10980XE 3.00GHz CPU, boasting 18 cores, 256 GB RAM, a PCIe 3.0 ×16 interface, and an NVIDIA GeForce RTX 3090 GPU with 24 GB memory. The NVIDIA driver version 546.01. The operating system is 64-bit Windows 10. We used C++, CUDA 11.6, and OpenGL to implement our approach. All the parallel algorithms, including LOD selection, reformation, culling, and out-of-core data management, are implemented using CUDA. OpenGL is utilized for rasterization.

Table 1. Each model is pre-processed into the balanced hierarchical representation. The seven statue models are composed into *Statue Models* scene for this experiment.

Name	**Triangle #**	**Vertex #**
St. Matthew	372,783,137	186,868,583
Atlas	507,532,941	254,877,810
David	56,230,962	28,184,520
Fangyi	55,287,586	27,643,703
Gong	65,726,059	32,862,956
Lucy	28,055,742	14,027,870
George Washington	31,260,808	15,630,410
Statue Models	1,116,877,235	560,095,852
Moana Island	1,286,959,510	984,065,632
Total	2,403,836,745	1,544,161,484

The test scenes for our experiment incorporate a set of models from Stanford's Digital Michelangelo Project [Levoy et al. 2000], Smithsonian 3D Digitization [Smithsonian Institution 2024], and Disney's Moana Island [Disney Animation Studios 2018]. As shown in Fig. 12, our experiment demonstrates that our method is capable of rendering large surface models and complex environments mixed with a large number of individual objects and animated characters. The test scenes, totaling over 2 billion triangles, require 149.03 GB as OBJ files for meshes and FBX files for animated characters (Table 1). Each scene was rendered at 1920x1080 resolution along recorded camera paths (Fig. 13), featuring positional and rotational transitions that create varying levels of coherence between frames, commonly occurring in video games and 3D visualization applications.

The node size, which can be determined based on our desired depth of the mesh structure, is an independent variable that affects overall performance and memory usage. Another independent variable is the size of the memory sector allocated in GPU memory. This size is determined by the node budget. It affects the quality and overall performance of rendering. Additionally, the coherence budget, which specifies the maximum amount of data transferred from CPU to GPU between frames, plays a critical role in balancing visual quality and smoothness. According to the description in Section 5, the algorithms in our approach support node insertion and in-place defragmentation modes for runtime data management. We conducted a systematic analysis of these two modes and found an optimal way to combine them in terms of variation of the transferred node amount to achieve the best overall performance. We also compared our approach with other state-of-the-art methods in out-of-core rendering.

7.1 Node Size Analysis

We conducted an experiment to evaluate the impact of different node sizes on overall performance and memory usage ratio. The experiment utilizes the *Statue Models* scene. The node size is determined by the maximum number of triangles and vertices in the nodes, which in turn depends on the depths of the hierarchical mesh structures during their construction. For example, when the Atlas model is partitioned and represented in an 18-level mesh structure, we found the maximum number of triangles in a node is 2,586 triangles, and the maximum number of vertices in a node is 1,841. As shown in Table 2, the frame time (representing the average total execution time spent on a frame) decreases as the node size is reduced, reaching the time nadir (node size: 19,214 triangles, 9,680 vertices) for both insertion and in-place defragmentation data management modes.

However, further reducing the node size beyond that at the time nadir causes the frame time to increase again. This is attributed to the mixed effects of several execution components in our approach reacting to different sizes of nodes. Fig. 14 shows the performance

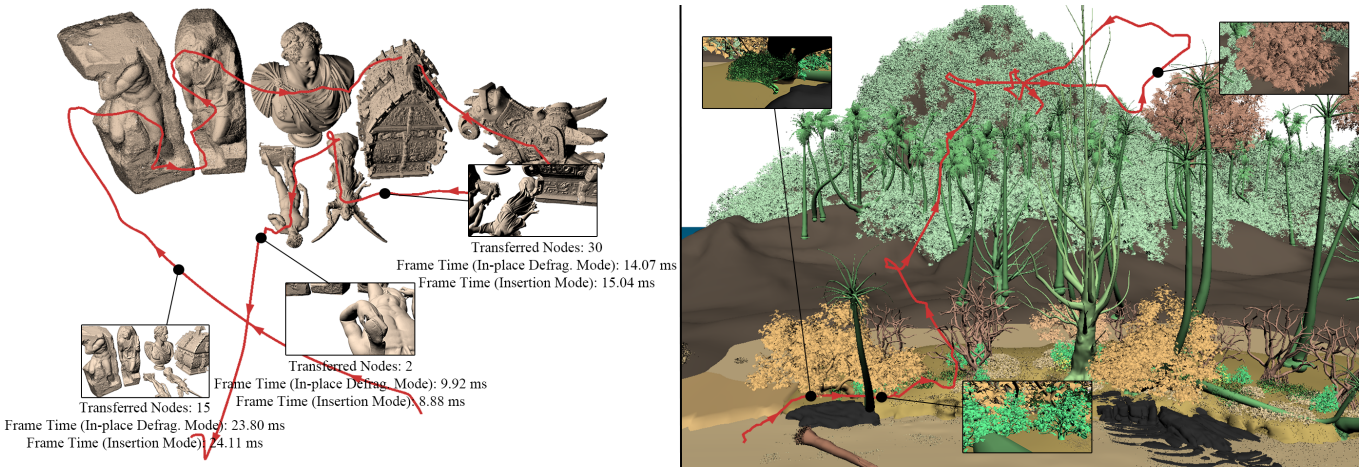

Fig. 13. The visualization of the camera paths (red lines) of *Statue Models* (left) and *Moana Island* (right) scenes. The left side of the figure shows the frame time of three picked frames. When the number of transferred nodes is larger than 15, the in-place defragmentation mode spends less time. On the contrary, the insertion mode spends less time when only transferring a few nodes.

Table 2. Performance and Memory Usage Ratio in the *Statue Models* scene under the Configuration with Different Node Sizes. For implementation, nodes have been converted from an AoS format to a SoA format to make them compatible with GPU storage (see Fig. 5). Consequently, we determined the node size in the triangle array and the node size in the vertex array, based on the highest counts of triangles and vertices, respectively, within the nodes of the mesh structure.

Node Size (Triangles/Vertices)	Frame Time		Triangles Rendered	Memory Usage Ratio
	Defrag. Mode	Insert. Mode		
4,251,178 / 2,125,752	27.86 ms	24.78 ms	50.43 million	82.88%
2,127,135 / 1,063,760	26.81 ms	23.06 ms	52.32 million	83.28%
1,070,293 / 537,062	23.37 ms	19.98 ms	47.57 million	83.30%
540,949 / 271,845	22.44 ms	18.67 ms	42.62 million	82.67%
273,454 / 137,636	20.83 ms	17.78 ms	45.08 million	81.96%
139,544 / 70,397	17.47 ms	14.82 ms	39.16 million	80.85%
71,103 / 39,630	14.04 ms	12.09 ms	36.28 million	76.70%
36,276 / 18,315	11.63 ms	10.00 ms	28.37 million	78.99%
19,214 / 9,680 (time nadir)	11.22 ms	9.75 ms	22.50 million	76.00%
9,934 / 6,204	12.75 ms	10.95 ms	19.08 million	68.41%
5,024 / 3,223	14.46 ms	12.86 ms	16.55 million	67.15%
2,586 / 1,841	18.66 ms	17.40 ms	16.42 million	63.05%

breakdown of the insertion and in-place defragmentation modes. The components from our out-of-core data management approach, including "Find Δf node IDs", "Collect Δf nodes", "Instruction (Insertion)", "Instruction (Defrag.)", and "Defrag.", do not become performance bottlenecks, nor does the "Transfer" (transfer Δf nodes from CPU to GPU) component due to leveraging frame-to-frame coherence. The "Rasterization" time emerges as a primary factor influencing overall performance, and it scales with the number of triangles to be rendered. "Reformation" time, which consistently decreases as node size decreases, is second to it. Rasterization time decreases from 16.16 ms to 3.11 ms, but beyond this point, it starts to increase again as node size continues to decrease. This is because smaller node sizes lead to more accurate culling results (where "Culling" is efficient), reducing the number of triangles for rasterization. However, if the node size becomes too small, specifically smaller than at the time nadir, it leads to a larger number of nodes to be rendered and higher overhead due to invoking more draw calls, consequently causing the rasterization time to increase again.

Looking into our data management approach and the cost of data transfer, the times spent on "Transfer" and "Collect Δ f nodes" decrease with smaller node sizes. Smaller nodes is able to reduce the amount of unnecessary vertices and triangles transferred with the nodes, thereby lowering the overall cost of data management. However, a smaller node size leads to more nodes in the mesh structure, and results in increased execution times in other components of the data management approach because they scale up with the increasing number of nodes. These execution times become more obvious and considerable when the node size falls below that at the time nadir, where the trade-off between the out-of-core data management and rasterization becomes less favorable.

Since the memory consumption of the nodes are not exactly balanced, there are unused memory portions in each memory unit when it is occupied by a node. We defined the term "Memory Usage ratio" to reflect the percentage of occupied memory size over the total size of the memory sector allocated. This ratio is influenced by the node size, varying from 82.88% to 63.05% in our experiment. When the node size is at the time nadir, it maintains an acceptable memory utilization of 76%. When the node is configured to the next smaller size, the ratio drops by 7.59%.

For other experiments, we selected the node size at the time nadir, with which the mesh structure for the entire scene requires 186.07 GB of storage (24.85% more than their original 149.03 GB), consisting of 507,785 nodes in total. Under this node size configuration, the rebuilding process was rarely invoked. For instance, in the case of the Atlas model, the rebuilding occurred only once at layer 6, producing a hierarchy with only two blocks. In contrast, for smaller models such as David, no rebuilding was required.

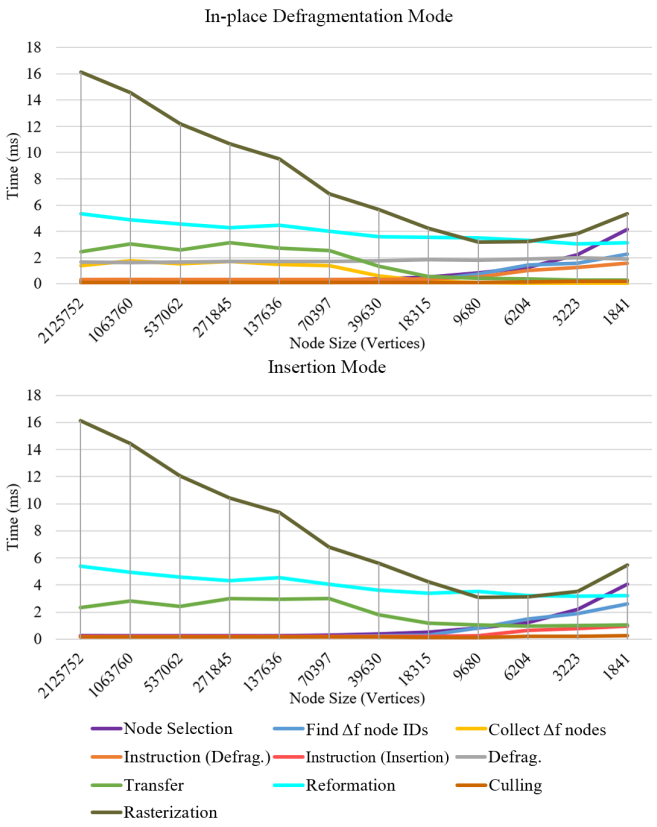

Fig. 14. Performance breakdown of different components of in-place defragmentation and insertion modes and how they change with the change of node sizes. The rasterization time first decreases and then increases, bottoming out at the node size of 9680 vertices. The costs of "Collect Δf nodes", "Reformation", and "Transfer" components decrease with the decrease of node size. The costs of "Node Selection", "Find Δf node IDs", "Instruction (Insertion)", and "Instruction (Defrag.)" components increase with the decrease in node size. The cost of "Defrag." and "Culling" components are stable over changes in node size.

7.2 Efficiency Analysis for Out-of-core Data Management

Our algorithms support the implementation of two data management modes: insertion and in-place defragmentation. The insertion mode allows identifying and transferring frame-different nodes directly to the available memory units in the memory sector, but it requires multiple memory copy calls to transfer them one by one. The in-place defragmentation mode needs to first compact the reusable nodes in an in-place manner and then transfer the frame-different nodes in a chunk to the target memory slot in the memory sector with a single memory copy call.

In addition to our two modes, we implemented a standard out-of-place defragmentation approach for comparison. This approach allocates an extra memory sector to store reusable nodes inherited from the previous frame, and uses an out-of-place sorting process to consolidate new and reusable nodes into the active memory sector for the current frame [Dong and Peng 2023]. This approach simplifies the algorithm for mitigating the risks of memory conflicts or

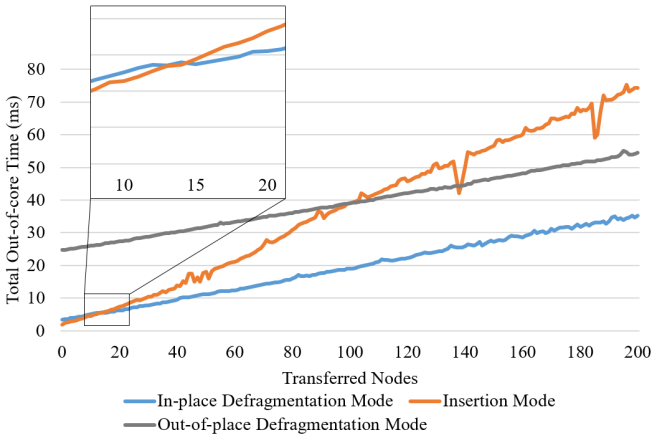

Fig. 15. Comparison of defragmentation and insertion modes in terms of total out-of-core time. The in-place defragmentation mode starts to outperform the insertion mode when the number of transferred nodes is equal to or larger than 15.

interference between read and write operations. We implemented the out-of-place defragmentation approach using the stream compaction algorithm [Bakunas-Milanowski et al. 2017] executed with CUDA Thrust.

We conducted numerical tests to evaluate and determine the optimal switching scheme between the insertion and in-place defragmentation modes, as well as to compare their performance with the out-of-place defragmentation approach. We set the budget of 16,000 nodes, and examined the out-of-core performance by streaming 0 to 200 nodes.

Fig. 15 illustrates the test results. It is evident that when the number of frame-different nodes is small, the insertion mode performs efficiently. When the number of nodes is large, the in-place defragmentation mode offers better performance. We found that the crossover point for performance occurs when transferring 15 nodes from the CPU to GPU. Fig. 16 provides more details through a performance breakdown. It shows that the impact of the "Defrag." component of in-place defragmentation mode diminishes as a significant factor with an increasing number of transferred nodes, as its contribution to overall performance remains relatively stable. Conversely, the insertion mode becomes less efficient because the cumulative overhead of transferring nodes individually progressively exceeds the cost of compacting and transferring frame-different nodes on the CPU side.

Based on the results from this test, we implemented a hybrid in-place approach that uses 15 nodes as the mode switching threshold. The rendering system automatically switches between insertion and in-place defragmentation modes to optimize performance dynamically.

In comparison, the out-of-place defragmentation mode exhibits a consistent frame time delay of approximately 20 ms compared to the in-place defragmentation mode. This additional overhead arises primarily from the need to manage an extra memory sector for reusable nodes and to perform the out-of-place sorting process. In our in-place defragmentation mode, the number of reusable nodes

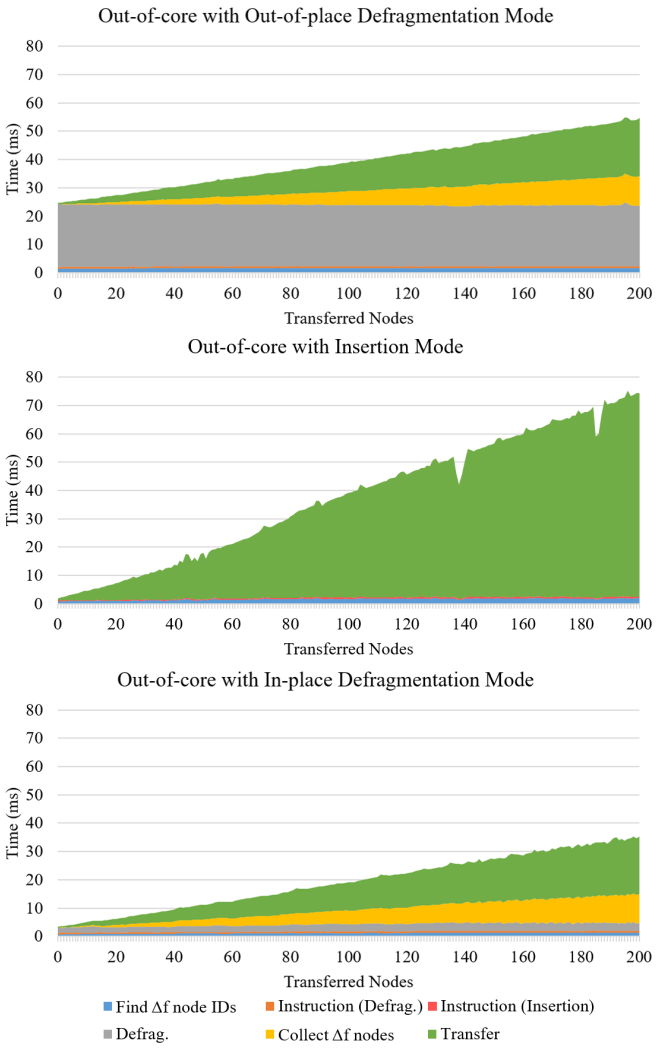

Fig. 16. Performance breakdown of the out-of-core components of insertion and defragmentation modes with the increase of transferred nodes. The cost of the "Defrag." component is very stable. The cost of "Transfer" of insertion mode is larger than the sum of the cost of "Transfer" and "Collect Δf nodes" of the in-place defragmentation mode. Thus, the in-place defragmentation mode outperforms the insertion mode when the impact of the "Defrag." component diminishes with the increase of transferred nodes. The in-place defragmentation mode consistently outperforms the out-of-place defragmentation mode by minimizing the cost of the "Defrag." component.

moving from the ForgeZone to the CycleZone is influenced by the overall count of reusable nodes and the size of the ForgeZone. As the total number of reusable nodes increases and the ForgeZone shrinks, there is a higher possibility that the number of reusable nodes in the ForgeZone decreases. We analyzed a theoretically worst-case scenario for our in-place defragmentation mode, where half of the nodes on the GPU are determined to be reusable, and they all happen to be situated in the ForgeZone. In this scenario, our approach would handle the maximum possible reusable nodes moving from

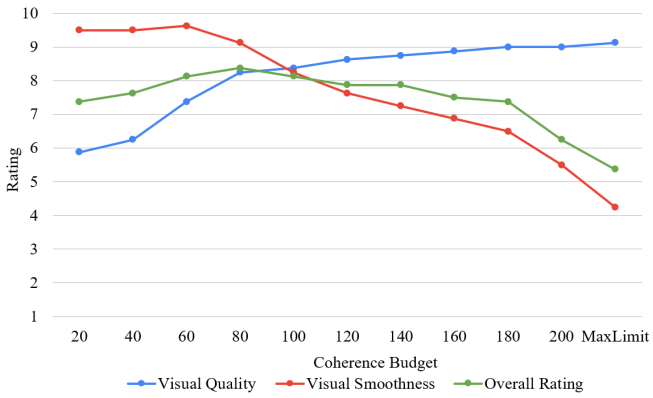

Fig. 17. User evaluations of visual quality, visual smoothness, and overall experience under different *coh_budget* settings. The overall rating peaks at a budget of 80, indicating an optimal balance between quality and smoothness.

the ForgeZone to the CycleZone. However, it would still outperform the out-of-place defragmentation mode, which has to operate all nodes residing in the GPU memory without benefiting from any frame-to-frame coherent property.

7.3 Coherence Budget Analysis

To determine an appropriate setting for *coh_budget*, we conducted a user study using the *Moana Island* scene. Eight participants were recruited to provide feedback on visual quality, visual smoothness, and overall visual experience for each *coh_budget* setting, ranging from 0 to 200. Additionally, we added a test case that *coh_budget* equals to the maximum node count that can be held by the size of GPU memory, denoted as *MaxLimit*. In this case, the node selection result will always be the desired LOD for every frame. Feedback was recorded on a scale of 1 to 10, where 1 indicates dissatisfaction and 10 indicates high satisfaction.

The user study results in Fig. 17 indicate that visual quality improves as *coh_budget* increases, as higher budgets allow selections to align more closely with the desired LOD results. Visual smoothness decreases with larger budgets due to the increased data transfer load between the CPU and GPU. At a *coh_budget* of 80, the overall visual experience receives the highest rating. This setting achieves visual quality that is only marginally lower than at higher budgets, while the reduction in visual smoothness remains minimal.

7.4 Comparison with Other Approaches

7.4.1 Comparison with Academic Methods. Our approach, featuring frame-to-frame coherence and in-place data management (FTFC-ID), was compared to three existing GPU out-of-core approaches in the research literature. These approaches were implemented based on theories and methodologies presented in prior work.

- Frame-to-frame coherence and out-of-place defragmentation (FTFC-OD): As described in Section 7.2, this method uses an additional memory sector to store reusable nodes from the previous frame. It consolidates these nodes with frame-different nodes into the active memory sector using an out-of-place sorting process to avoid read-write conflicts. But this

Table 3. Performance Breakdowns of Different Approaches in the *Statue Models* scene.

Approaches	Frame Time	Triangles Rendered	Node Selection	Out-of-core			Reformation	Culling	Rasterization
				Δf nodes Prep.	Instruction & Defrag.	Transfer			
FTFC-ID (Ours)	9.65 ms	22.50 million	0.82 ms	0.68 ms	0.41 ms	1.06 ms	3.46 ms	0.13 ms	3.10 ms
FTFC-OD	49.02 ms	(2.01% of	0.84 ms	1.82 ms	39.03 ms	0.64 ms	3.31 ms	0.15 ms	3.22 ms
FTFC-DM	1083.25 ms	total triangles)	0.99 ms	—	—	28.24 ms	965.97 ms	1.27 ms	86.78 ms
NFTFC	1932.49 ms		0.84 ms	721.07 ms	—	1203.94 ms	3.22 ms	0.20 ms	3.22 ms

method introduces additional overhead due to the need for extra memory allocation.

- Frame-to-frame coherence and dynamic memory management (FTFC-DM): This approach leverages frame-to-frame coherence but does not maintain single or double fixed-size memory buffers in GPU memory [Varadhan and Manocha 2002]. Instead, it manages nodes as individual mesh objects and is able to retain reusable ones in GPU memory between frames through buffer objects [Angel and Shreiner 2011]. The system manages memory allocation and deallocation automatically. This approach offers flexible management for nodes, but it requires multiple rendering passes.
- Streaming data without frame-to-frame coherence (NFTFC): This approach streams nodes from CPU to GPU without maintaining coherence between frames [Woo et al. 1999]. A fixed-size memory sector is allocated and used during the runtime for rendering the current frame. The nodes selected for the current frame are collected in CPU memory and transferred to GPU with a single memory copy call. This approach does not require computing data-transfer instructions or defragmenting GPU-resident data. The ways it manages the nodes and memory are the same as those in the standard GPU graphics pipeline [Kenzel et al. 2018], where the data of the entire frame is transferred to the GPU.

We used the *Statue Models* scene as the test case. Table 3 shows the performance comparison. For the GPU out-of-core approaches, the GPU memory sector is set to hold a maximum of 16,000 nodes, as this is the highest node count that FTFC-OD can support due to its requirement for extra memory allocations. During runtime, an average of 22.50 million triangles are rendered per frame, constituting 2.01% of the triangle count in the original dataset. All the GPU out-of-core approaches spend a similar amount of time on "Node Selection". FTFC-OD, NFTFC, and our approach use single-pass rendering and fixed-size memory allocation for the frame, so the execution times on "Reformation" and "Rasterization" are similar, and they are much faster than those in the FTFC-DM approach.

FTFC-DM has the worst performance. It has significant overheads in executing "Transfer", "Reformation", and "Rasterization" components. This is because each node is instantiated and destroyed on the GPU separately, and dynamic memory allocation and deallocation commands add extra cost to the process of data transfer. "Reformation" in FTFC-DM runs for each node separately, making GPU computational ability not fully utilized. Additional overhead on kernel launches makes the execution of "Reformation" very slow. In terms of "Rasterization", FTFC-DM assigns each node to a buffer object, causing extra overhead due to irregular memory access patterns.

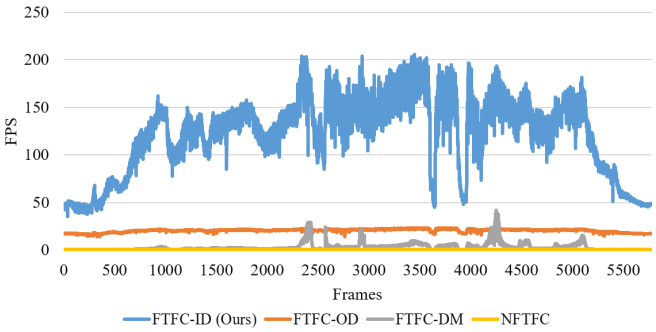

Fig. 18. The FPS over the frames of the *Statue Models* scene. The node budget is set to 16,000. Our approach significantly outperforms FTFC-OD, FTFC-DM, and NFTFC.

NFTFC, despite not requiring defragmentation, spends a significant amount of time on "Δf nodes Prep." ("Find Δf node IDs" and "Collect Δf nodes") and "Transfer" since it has to transfer all the selected nodes. The time spent on these two components constitutes 99.38% of the total time of the NFTFC approach. In comparison, our approach and FTFC-OD provide better performance, benefiting from the use of both frame-to-frame coherence and fixed-size memory allocations.

Our approach demonstrates higher efficiency in managing out-of-core data than FTFC-OD, making the GPU out-of-core component not the bottleneck in our approach. Out-of-core data management is the most time-consuming stage in FTFC-OD and NFTFC. Fig. 18 shows the FPS changes over 5,773 frames of the camera path. Our approach outperforms other approaches, with an average FPS of 122.29, compared to FTFC-OD (20.55 FPS), FTFC-DM (3.36 FPS), and NFTFC (0.52 FPS).

7.4.2 Comparison with Commercial Systems. Rendering systems in modern game engines, such as Unity and Unreal, are widely recognized for their real-time rendering applications beyond video games due to their advanced graphical capabilities.

Unity is unable to load our test scenes, as it lacks the specialized structures and data management optimizations required to handle scenes containing billions of geometric primitives. The most complex scene configuration tested with Unity (v2022.3.44f1) contains 744 million triangles and renders at a peak of 1.41 FPS with the basic shading setting.

Unreal introduces Nanite system [Brian Karis 2021], which is a commercial solution designed to support the rendering of large

Table 4. Performance Breakdowns of Our Approaches in Different Scenes

Scenes	Frame Time	Triangles Rendered	Node Selection	Out-of-core			Reformation	Culling	Rasterization
				Δf nodes Prep.	Instruction & Defrag.	Transfer			
Statue Models	9.65 ms	22.50 million (2.01%)	0.82 ms	0.68 ms	0.41 ms	1.06 ms	3.46 ms	0.13 ms	3.10 ms
Moana Island	10.99 ms	20.81 million (1.62%)	1.75 ms	1.01 ms	0.88 ms	1.23 ms	2.36 ms	0.11 ms	3.65 ms
Combined Scene	17.12 ms	41.74 million (1.74%)	2.59 ms	1.19 ms	0.96 ms	1.54 ms	2.77 ms	0.10 ms	7.97 ms

3D scenes. It uses an HLOD structure to dynamically adjust and stream geometric data based on the viewer's position and perspective. Nanite manages complex assets by delivering only the necessary details, reducing memory usage and data streaming demands.

Unlike our approach, Nanite adopts a Directed Acyclic Graph (DAG) as its hierarchical structure to handle dense boundaries between clusters. During DAG construction, clusters are grouped into fixed-size sets, and each cluster is linked to all relevant parent clusters within the DAG. This structure restricts LOD selection decisions to the group level. Nanite's structure is best suited for representing static geometries. Our approach is capable of handling dynamic scenes. As long as the original mesh topology remains unchanged, meshes in the input scene, including deformable ones or those with skeletal animations, can be preprocessed into our hierarchical mesh structure and integrated into the out-of-core GPU data management and rendering pipeline.

Nanite's LOD selection method primarily relies on screen-space error metrics. Advancing this method, our approach also accounts for CPU-GPU transfer bandwidth limits and allows specifying a coherent data budget, providing an additional layer of optimization. This is especially useful in scenarios where transferring all selected nodes would exceed the real-time rendering capability, making it impractical despite satisfying visual quality criteria.

Additionally, Nanite does not specify how GPU memory is managed after streaming. Our in-place GPU data management algorithm could serve as a potential solution to efficiently align current frame data with rasterization requirements and help mitigate data fragmentation in subsequent frames.

We tested the *Statue Models* scene in Unreal v5.5.1 using the basic shading setting and set *r.Nanite.MaxPixelsPerEdge* to its minimum value for the highest possible geometry quality; otherwise, small objects may disappear or exhibit popping artifacts. We set other settings of Nanite at their default values, as modifying them does not affect the geometry quality. As Unreal is a commercial game engine, differences in the rendering pipelines lead Nanite and our approach to not produce identical shading results. In this test, Nanite achieves an average frame time of 30.05 ms while rendering an average of 15.91 million triangles, three times slower than our approach (FTFC-ID).

7.5 Discussion

7.5.1 Scenario Adaptability. To understand the adaptability of our method across different scenarios, *Statue Models* scene, *Moana Island* scene, and a combined scene are tested and the performance results are shown in Table 4. Performance varies with scene complexity. As complexity increases, frame time grows, reaching 17.12 ms in the combined scene, primarily due to rasterization time, which rises significantly with the increased triangle count (41.74 million) and the addition of textured and animated models. In contrast, node selection and out-of-core data management times increase more moderately, demonstrating the scalability of our method. For instance, node selection time rises from 0.82 ms in the *Statue Models* scene to 2.59 ms in the combined scene, while out-of-core data management increases from 2.15 ms to 3.69 ms. These small increments highlight the approach's adaptability to larger datasets with minimal overhead.

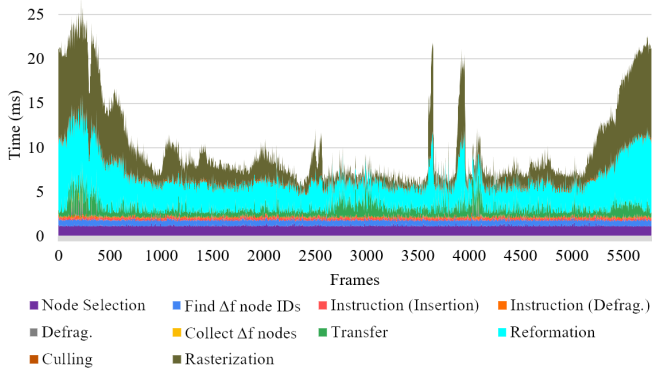

Fig. 19. The costs of components of FTFC-ID over the frames in the *Statue Models* scene. Due to high frame-to-frame coherence, the most time-consuming components are "Reformation" and "Rasterization", which made performance fluctuations based on the changes of rendered triangles.

7.5.2 Out-of-core Cost. Our approach significantly reduces the out-of-core data management cost. In the experiment with the *Statue Models* scene, the out-of-core stage of our approach comprised only 22.23% (2.15 ms) of the total time, a substantial improvement compared to FTFC-OD, where defragmentation consumes 84.64% (41.49 ms). This efficiency gain arises from our hybrid approach integrating both insertion and in-place defragmentation modes for data management.

7.5.3 Quality of Coherence. Our node selection strategy takes into account spatial coherence, ensuring superior performance compared to existing methods. In interactive visualization applications, where camera positions change more smoothly than orientations, this approach enables nodes not immediately required to be preselected and transferred to the GPU as anticipative nodes for future frames. Modern GPUs can cache more geometric primitives than they can render into pixels in real-time (Section 1). By storing anticipative nodes, memory resources for real-time rendering remain unaffected, while the buffer enhances memory utilization and contributes to smoother upcoming frames.

Table 5. Performance Breakdowns of Our Approach in the *Statue Models* scene with Different Node Budgets (# of Nodes)

# of Nodes	Frame Time	Triangles Rendered	Node Selection	Out-of-core			Reformation	Culling	Rasterization
				Δf nodes Prep.	Instruction & Defrag.	Transfer			
2,000	5.11 ms	4.28 million (0.38%)	0.81 ms	0.85 ms	0.40 ms	1.25 ms	0.80 ms	0.18 ms	0.82 ms
4,000	6.01 ms	7.53 million (0.67%)	0.79 ms	0.85 ms	0.42 ms	1.40 ms	1.16 ms	0.18 ms	1.22 ms
6,000	7.35 ms	11.06 million (0.99%)	0.82 ms	1.00 ms	0.41 ms	1.77 ms	1.56 ms	0.11 ms	1.68 ms
8,000	7.41 ms	12.60 million (1.13%)	0.81 ms	0.85 ms	0.41 ms	1.45 ms	1.89 ms	0.11 ms	1.89 ms
10,000	7.31 ms	13.30 million (1.19%)	0.80 ms	0.70 ms	0.41 ms	1.20 ms	2.15 ms	0.11 ms	1.95 ms
12,000	9.54 ms	19.33 million (1.73%)	0.81 ms	0.99 ms	0.42 ms	1.57 ms	2.84 ms	0.12 ms	2.79 ms
14,000	9.39 ms	21.81 million (1.95%)	0.81 ms	0.71 ms	0.42 ms	1.05 ms	3.30 ms	0.13 ms	2.98 ms
16,000	9.65 ms	22.50 million (2.01%)	0.82 ms	0.68 ms	0.41 ms	1.06 ms	3.46 ms	0.13 ms	3.10 ms

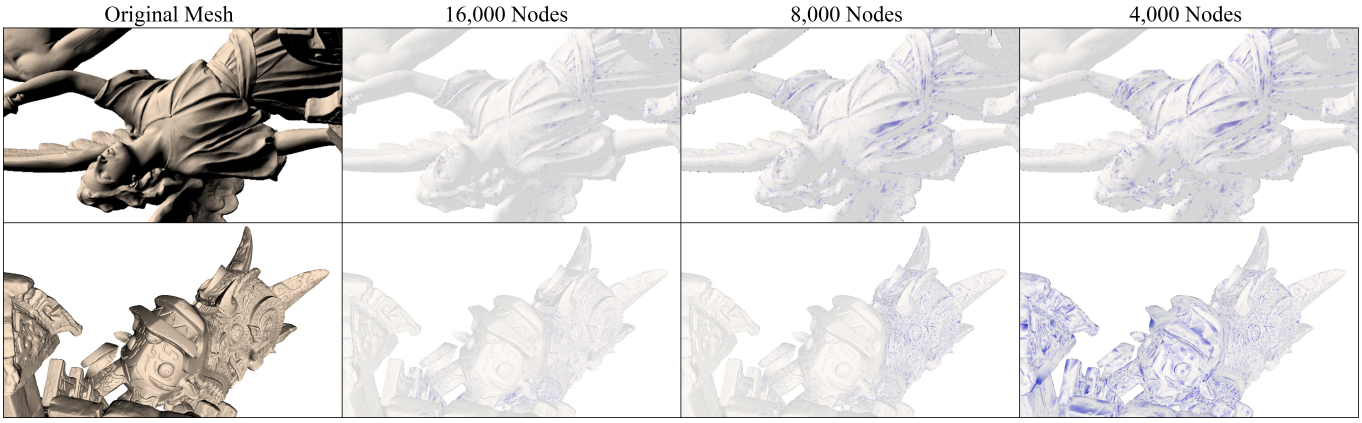

Fig. 20. Screenshots demonstrating the visual quality loss with different node budgets. The blue highlights indicate the differences between the scene with the selected nodes and the scene with the original meshes. A higher node budget better preserves visual quality. In our experiment, the budget with 16,000 nodes resulted in rendering that is very close to the groundtruth quality of the original meshes.

7.5.4 Memory Management Efficiency. A key strength of our memory management approach is that the defragmentation algorithm operates entirely in-place within a single memory buffer. This conserves GPU memory that would otherwise be consumed by FTFC-OD, enabling a higher node budget through the saved resources.

7.5.5 Performance Fluctuations. As shown in Fig. 18, most approaches exhibit performance fluctuations over time in the *Statue Models* scene. The FPS of our approach is ranged from 35.23 to 205.72. This variability is due to the dynamic changes in the number of rendered triangles, which are adjusted based on the results of LOD selection and culling. Fig. 19 shows the temporal cost variations of different components in our approach over time. The primary factors influencing performance are the reformation and rasterization components, as their costs vary with the number of rendered triangles. In contrast, the performance of other components remains stable throughout.

7.5.6 Performance and Quality with Different Node Budgets. Node budgets affect both rendering performance and quality. We evaluated the performance of rendering the *Statue Models* scene under node budgets ranging from 2,000 to 16,000 nodes, as shown in Table 5. The time spent on "Node Selection", "Out-of-core", and "Culling" components remains stable across all cases, while other components experience only modest increases in execution time as the node budget grows. As shown in Fig. 20, higher node budgets lead to improved quality, closer to the ground truth. Each node in our mesh structure supports continuous LOD, enabling smooth quality transitions to coarser parent nodes. This feature effectively reduces popping artifacts, even when nodes are selected across levels or switch back and forth.

8 Conclusion

In this paper, we presented a GPU out-of-core approach that achieves real-time performance for rendering complex scenes composed of billions of triangles. We introduced a coherence-constrained LOD selection algorithm and an in-place parallel data management approach that leverage the properties of a balanced hierarchical mesh structure and frame-to-frame coherence. This minimizes the cost on CPU-to-GPU data transfer and allows for the direct consolidation of new and reusable data within a memory sector.

We demonstrated the advancements of our approach by comparing it to the existing approaches. We also gained valuable insights into the effectiveness of our approach in supporting real-time rendering across scenarios of varying complexity, including those with textured and animated models.

For future work, we plan to incorporate advanced lighting and shading effects, such as shadow mapping and material properties, to enhance the aesthetic quality of the rendering.

Acknowledgments

This work was partially supported by NSF CNS Grants 1464323 and 2000488, and DOD Grant W911NF-16-2-0016. We thank the Digital Michelangelo Project at Stanford University, Smithsonian Institution, and Walt Disney Animation Studios for providing access to the large 3D datasets used in our experiments. We also thank the anonymous reviewers for their valuable comments and feedback, and the RIT MAGIC Center for its support in technology management and logistics.

References

Edward Angel and Dave Shreiner. 2011. Introduction to modern OpenGL programming. In *ACM SIGGRAPH 2011 Courses*. 1–136. doi:10.1145/2037636.2037643

Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 136–150. doi:10.1145/3123939.3123975

Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Yu Chansu. 2017. Efficient algorithms for stream compaction on GPUs. *International Journal of Networking and Computing* 7, 2 (2017), 208–226. doi:10.15803/ijnc.7.2_208

Dirk Bartz, Douglas W Cunningham, Jan Fischer, and Christian Wallraven. 2008. The role of perception for computer graphics. In *Eurographics (State of the Art Reports)*. 59–80. doi:10.2312/egst.20081045

Markus Billeter, Ola Olsson, and Ulf Assarsson. 2009. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the conference on high performance graphics 2009*. 159–166. doi:10.1145/1572769.1572795

Erik G Boman, Ümit V Çatalyürek, Cédric Chevalier, and Karen D Devine. 2012. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Scientific Programming* 20, 2 (2012), 129–150. doi:10.3233/SPR-2012-0342

Graham Wihlidal Brian Karis, Rune Stubbe. 2021. A deep dive into Unreal Engine's 5 Nanite. In *SIGGRAPH Course*. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6-8 (2008), 318–331. doi:10.1016/j.parco.2007.12.001

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. 2004. Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics (TOG)* 23, 3 (2004), 796–803. doi:10.1145/1015706.1015802

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. 2005. Batched multi triangulation. In *VIS 05. IEEE Visualization, 2005.* IEEE, 207–214. doi:10.1109/VISUAL.2005.1532797

Dilip Kumar Dalei, N Venkataramanan, and Narayan Panigrahi. 2022. A review of LOD based techniques for real-time terrain rendering. In *2022 IEEE 6th Conference on Information and Communication Technology (CICT)*. 1–6. doi:10.1109/CICT56698.2022.9997973

Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. 2019. Visibility rendering order: Improving energy efficiency on mobile GPUs through frame coherence. *IEEE Transactions on Parallel and Distributed Systems* 30, 2 (2019), 473–485. doi:10.1109/TPDS.2018.2866246

Evgenij Derzapf and Michael Guthe. 2012. Dependency-free parallel progressive meshes. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 2288–2302. doi:10.1111/j.1467-8659.2012.03154.x

Evgenij Derzapf, Nicolas Menzel, and Michael Guthe. 2010. Parallel view-dependent out-of-core progressive meshes. In *Vision, Modeling, and Visualization (2010)*, Reinhard Koch, Andreas Kolb, and Christof Rezk-Salama (Eds.). The Eurographics Association. doi:10.2312/PE/VMV/VMV10/025-032

Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. 2014. Linear algebraic representation for topological structures. *Computer-Aided Design* 46 (2014), 269–274. doi:10.1016/j.cad.2013.08.044

Andreas Dietrich, Enrico Gobbetti, and Sung-Eui Yoon. 2007. Massive-model rendering techniques: A tutorial. *IEEE Computer Graphics and Applications* 27, 6 (2007), 20–34. doi:10.1109/MCG.2007.154

Disney Animation Studios. 2018. Moana island scene. https://www.disneyanimation.com/resources/moana-island-scene/

Yangzi Dong and Chao Peng. 2023. Multi-GPU multi-display rendering of extremely large 3D environments. *The Visual Computer* 39, 12 (2023), 6473–6489. doi:10.1007/s00371-022-02740-7

Jonathan Dupuy and Kenneth Vanhoey. 2021. A halfedge refinement rule for parallel Catmull-Clark subdivision. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 57–70. doi:10.1111/cgf.14381

Xiaowen Feng, Hai Jin, Ran Zheng, Zhiyuan Shao, and Lei Zhu. 2014. A segment-based sparse matrix–vector multiplication on CUDA. *Concurrency and Computation: Practice and Experience* 26, 1 (2014), 271–286. doi:10.1002/cpe.2978

Anshuj Garg, Purushottam Kulkarni, Uday Kurkure, Hari Sivaraman, and Lan Vu. 2019. Empirical analysis of hardware-assisted GPU virtualization. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 395–405. doi:10.1109/HiPC.2019.00054

Chirayu Garg and Nikolay Sakharnykh. 2021. Improving GPU memory oversubscription performance. (2021).

Michael Garland and Paul S Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 209–216. doi:10.1145/258734.258849

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. Ai and memory wall. *IEEE Micro* (2024). doi:10.1109/MM.2024.3373763

Enrico Gobbetti and Fabio Marton. 2005. Far Voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph.* 24, 3 (July 2005), 878–885. doi:10.1145/1073204.1073277

Eduard Groller and Werner Purgathofer. 1995. *Coherence in computer graphics*. Technical Report TR-186-2-95-04. Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria. https://www.cg.tuwien.ac.at/research/publications/1995/Groeller-1995-CCG/ human contact: technical-report@cg.tuwien.ac.at.

Liang Hu, Pedro V Sander, and Hugues Hoppe. 2009. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics* 16, 5 (2009), 718–728. doi:10.1109/TVCG.2009.101

Harold Hubschman and Steven W Zucker. 1982. Frame-to-frame coherence and the hidden surface computation: constraints for a convex world. *ACM Transactions on Graphics (TOG)* 1, 2 (1982), 129–162. doi:10.1145/357299.357302

Martin Isenburg and Stefan Gumhold. 2003. Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics (TOG)* 22, 3 (2003), 935–942. doi:10.1145/882262.882366

George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392. doi:10.1137/S1064827595287997

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A high-performance software graphics pipeline architecture for the GPU. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–15. doi:10.1145/3197517.3201374

Jens Krüger and Rüdiger Westermann. 2005. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*. 234–es. doi:10.1145/1198555.1198795

Thibaud Lambert, Pierre Bénard, and Gael Guennebaud. 2018. A view-dependent metric for patch-based LOD generation & selection. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–21. doi:10.1145/3203195

Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, et al. 2000. The digital Michelangelo project: 3D scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 131–144. doi:10.1145/344779.344849

Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 49–63. doi:10.1145/3297858.3304044

Rui Li. 2023. View-dependent Adaptive HLOD: real-time interactive rendering of multi-resolution models. In *Proceedings of the 20th ACM SIGGRAPH European Conference on Visual Media Production*. 1–10. doi:10.1145/3626495.3626507

Weisi Lin and C-C Jay Kuo. 2011. Perceptual visual quality metrics: A survey. *Journal of visual communication and image representation* 22, 4 (2011), 297–312. doi:10.1016/j.jvcir.2011.01.005

Baoquan Liu, Gordon J Clapworthy, Feng Dong, and Edmond C Prakash. 2012. Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 19, 10 (2012), 1732–1745. doi:10.1109/TVCG.2012.151

David Luebke and Benjamin Hallen. 2001. Perceptually driven simplification for interactive rendering. In *Rendering Techniques 2001: Proceedings of the Eurographics Workshop in London, United Kingdom, June 25–27, 2001 12*. Springer, 223–234. doi:10.1007/978-3-7091-6242-2_21

Ahmed H Mahmoud, Serban D Porumbescu, and John D Owens. 2021. RXMesh: A GPU mesh data structure. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16. doi:10.1145/3450626.3459748

Martti Mäntylä. 1987. *An introduction to solid modeling*. Computer Science Press, Inc. https://dl.acm.org/doi/abs/10.5555/39278

Avinash Maurya, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2023. Towards efficient I/O pipelines using accumulated compression. In *HIPC'23: 30th IEEE International Conference on High Performance Computing, Data, and Analytics*. doi:10.1109/HiPC58850.2023.00043

Joerg H Mueller, Thomas Neff, Philip Voglreiter, Markus Steinberger, and Dieter Schmalstieg. 2021. Temporally adaptive shading reuse for real-time rendering and virtual reality. *ACM Transactions on Graphics (TOG)* 40, 2 (2021), 1–14. doi:10.1145/3446790

NVIDIA. 2015. Life of a triangle - NVIDIA's logical pipeline. https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

NVIDIA. 2020. NVIDIA Ampere GA102 GPU architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

Chao Peng and Yong Cao. 2012. A GPU-based approach for massive model rendering with frame-to-frame coherence. *Computer Graphics Forum* 31, 2pt2 (2012), 393–402. doi:10.1111/j.1467-8659.2012.03018.x

Chao Peng, Seung In Park, Yong Cao, and Jie Tian. 2011. A real-time system for crowd rendering: Parallel LOD and texture-preserving approach on GPU. In *Motion in Games*, Jan M. Allbeck and Petros Faloutsos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–38. doi:10.1007/978-3-642-25090-3_3

David Petrescu, Paul A Warren, Zahra Montazeri, and Steve Pettifer. 2023. Velocity-based lod reduction in virtual reality: A psychophysical approach. In *EUROGRAPH-ICS 2023*. Eurographics Association. doi:10.48550/arXiv.2301.09394

Federico Ponchio. 2009. *Multiresolution structures for interactive visualization of very large 3D datasets*. Univ.-Bibliothek.

Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. 2013. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 51–60. doi:10.1109/SASO.2013.13

Rui SV Rodrigues, José FM Morgado, and Abel JP Gomes. 2018. Part-based mesh segmentation: A survey. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 235–274. doi:10.1111/cgf.13323

Jonathan Sarton, Nicolas Courilleau, Yannick Rémion, and Laurent Lucas. 2019. Interactive visualization and on-demand processing of large volume data: A fully GPU-based out-of-core approach. *IEEE transactions on visualization and computer graphics* 26, 10 (2019), 3008–3021. doi:10.1109/TVCG.2019.2912752

Henry Schäfer, Benjamin Keinert, and Marc Stamminger. 2013. Real-time local displacement using dynamic GPU memory management. In *Proceedings of the 5th High-Performance Graphics Conference*. 63–72. doi:10.1145/2492045.2492052

Daniel Scherzer, Lei Yang, and Oliver Mattausch. 2010. Exploiting temporal coherence in real-time rendering. In *ACM SIGGRAPH ASIA 2010 Courses*. 1–26. doi:10.1145/1900520.1900544

Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V Sander, Michael Wimmer, and Elmar Eisemann. 2012. Temporal coherence methods in real-time rendering. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 2378–2408. doi:10.1111/j.1467-8659.2012.03075.x

Smithsonian Institution. 2024. Smithsonian 3D digitization collections. https://3d.si.edu/collections

Matthias Springer and Hidehiko Masuhara. 2019a. DynaSOAr: A parallel memory allocator for object-oriented programming on GPUs with efficient memory access. In *33rd European Conference on Object-Oriented Programming*. doi:10.4230/LIPIcs.ECOOP.2019.17

Matthias Springer and Hidehiko Masuhara. 2019b. Massively parallel GPU memory compaction. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. 14–26. doi:10.1145/3315573.3329979

Danièle Tost and Pere Brunet. 1990. A definition of frame-to-frame coherence. In *Computer Animation '90*, Nadia Magnenat-Thalmann and Daniel Thalmann (Eds.). Springer Japan, Tokyo, 207–225. doi:10.1007/978-4-431-68296-7_15

Gokul Varadhan and Dinesh Manocha. 2002. Out-of-core rendering of massive geometric environments. In *IEEE Visualization, 2002. VIS 2002*. IEEE, 69–76. doi:10.1109/VISUAL.2002.1183759

Kechun Wang and Renjie Chen. 2023. Parallel loop subdivision with sparse adjacency matrix. In *Eurographics (Short Papers)*. 49–52. doi:10.2312/egs.20231012

Rui Wang, Yuchi Huo, Yazhen Yuan, Kun Zhou, Wei Hua, and Hujun Bao. 2013. GPU-based out-of-core many-lights rendering. *ACM Transactions on Graphics (TOG)* 32, 6 (2013), 1–10. doi:10.1145/2508363.2508413

Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. 1999. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc. https://dl.acm.org/doi/abs/10.5555/554539

Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24. doi:10.1145/216585.216588

Xiang Xu, Lu Wang, Arsène Pérard-Gayot, Richard Membarth, Cuiyu Li, Chenglei Yang, and Philipp Slusallek. 2022. Temporal coherence-based distributed ray tracing of massive scenes. *IEEE Transactions on Visualization and Computer Graphics* (2022). doi:10.1109/TVCG.2022.3219982

Junjie Xue, Gang Zhao, and Wenlei Xiao. 2016. An efficient GPU out-of-core framework for interactive rendering of large-scale CAD models. *Computer Animation and Virtual Worlds* 27, 3-4 (2016), 231–240. doi:10.1002/cav.1704

Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. 2010. Real-time concurrent linked list construction on the GPU. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 1297–1304. doi:10.1111/j.1467-8659.2010.01725.x

Shuai Yang, Yifan Zhou, Ziwei Liu, and Chen Change Loy. 2023. Rerender a video: Zero-shot text-guided video-to-video translation. In *SIGGRAPH Asia 2023 Conference Papers*. 1–11. doi:10.1145/3610548.3618160

Sung-Eui Yoon, Enrico Gobbetti, David Kasik, and Dinesh Manocha. 2022. *Real-time massive model rendering*. Springer Nature. doi:10.1007/978-3-031-79531-2

Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. 2004. Quick-VDR: Interactive view-dependent rendering of massive models. In *Visualization, 2004. IEEE*. IEEE, 131–138. doi:10.1109/VISUAL.2004.86

Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. 2017. A GPU-adapted structure for unstructured grids. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 495–507. doi:10.1111/cgf.13144

Huadong Zhang, Lizhou Cao, and Chao Peng. 2023. Spherical parametric measurement for continuous and balanced mesh segmentation. (2023). doi:10.2312/hpg.20231140

Huadong Zhang and Chao Peng. 2025. Foveated VR Rendering System for Large 3D Meshes. In *2025 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. 1246–1247. doi:10.1109/VRW66409.2025.00269

Qing Zhu, Junqiao Zhao, Zhiqiang Du, and Yeting Zhang. 2010. Quantitative analysis of discrete 3D geometrical detail levels based on perceptual metric. *Computers & Graphics* 34, 1 (2010), 55–65. doi:10.1016/j.cag.2009.10.004

Appendix:

Given an array containing randomly assigned values of 0s and 1s. If the array is cut into two parts at a position where the size of the first part is equal to the number of 1s in the entire array, the following statement is true:

The count of 0s in the first part is equal to the count of 1s in the second part.

Proof: Given an array A of length n containing randomly assigned values of 0s and 1s,

(1) Let n be the length of array A and k be the total number of 1s in A.
(2) Partition A such that the first part $A[0 : k]$ has k elements, where k is the number of 1s.
(3) Let k_1 and z_1 be the counts of 1s and 0s in $A[0 : k]$, respectively. Then, $k_1 + z_1 = k$.
(4) Let k_2 and z_2 be the counts of 1s and 0s in $A[k + 1 : n]$, respectively. Then, $k_2 = k - k_1$.
(5) From $k_1 + z_1 = k$, we get $z_1 = k - k_1$. Thus, $z_1 = k_2$.

Thus, the equation holds true, confirming that the number of 0s in the first part will be equal to the number of 1s in the second part when the array is cut at the position where the size of the first part is equal to the number of 1s in the entire array.