Real-time Visualization of Massive 3D Models on GPU Parallel Architectures

Chao Peng

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

Yong Cao, Chair

Douglas A. Bowman

Ming C. Lin

Christopher L. North

Dane Webster

April 2013

Blacksburg, Virginia

Keywords: mesh simplification, LOD selection, visibility culling, GPU out-of-core, massive model rendering, GPGPU, multi-GPU

Copyright 2013, Chao Peng

Real-time Visualization of Massive 3D Models on GPU Parallel Architectures

Chao Peng

(ABSTRACT)

Real-time rendering of massive 3D models has been recognized as a challenging task due to the limited computational power and memory available in a workstation. Most existing acceleration techniques, such as mesh simplification algorithms with hierarchical data structures, suffer from the nature of sequential executions. As data complexity increases due to the fundamental advances in modeling and simulation technologies, 3D models become complex and require gigabytes in storage. Consequently, visualizing such large datasets becomes a computationally intensive process where sequential solutions are unable to satisfy the demands of real-time rendering.

Recently, the Graphics Processing Unit (GPU) has been praised as a massively parallel architecture not only for its significant improvements in performance but also because of its programmability for general-purpose computation. Today's GPUs allow researchers to solve problems by delivering fine-grained parallel implementations. In this dissertation, I concentrate on the design of parallel algorithms for real-time rendering of massive 3D polygonal models towards modern GPU architectures. As a result, the delivered rendering system supports high-performance visualization of 3D models composed of hundreds of millions of polygons on a single commodity workstation.

Acknowledgments

First and foremost, I would like to sincerely thank my advisor, Dr. Yong Cao, for his advice and direction throughout my Ph.D. study. Without his support, this project would not have become reality. I deeply appreciate his helps from the initial stage of my degree program to the completion of this dissertation. I would like to extend my thanks to my committee members Dr. Doug A. Bowman, Dr. Ming C. Lin, Dr. Chris North and Prof. Dane Webster for their suggestions on my research project. I realize the sacrifice you have made in agreeing to serve on my committee.

Finally, I express my deepest gratitude to my loving and supportive wife, Siyuan Wang, for her unceasing moral support through my study. She constantly encouraged me to follow my dreams. Thank you for being you!

Contents

1 Introduction **1**

1.1 Research Motivations on Massive Model Rendering 2

1.1.1 Hardware Limitations 4

1.1.2 Advantages of GPU Parallel Architectures 5

1.2 Contributions 7

1.3 Dissertation Organization 8

2 Parallel Mesh Simplification Algorithm **9**

2.1 Related Works 9

2.2 Research Problems and Goals 12

2.3 GPU-Friendly Preprocessing 13

2.3.1 Key Criteria of Edge-Collapsing 14

2.3.2 Data Rearrangement 15

2.4 LOD Selection 20

2.4.1 First-Pass Algorithm 20

2.4.2 Second-Pass Algorithm 23

2.5 Triangle Reformation on the GPU 26

2.5.1 Parallelism and Storage Scheme 27

2.5.2 Triangle-Level Parallel Algorithm 28

2.6 Evaluation 30

3 GPU Out-Of-Core Algorithm 33

3.1 Related Works 34

3.2 Research Problems and Goals 35

3.3 Coherence-based Streaming 36

3.4 GPU Memory Defragmentation 38

3.5 Evaluation 43

4 Massive Model Rendering System: Integrating with Occlusion Culling Algorithm 46

4.1 Related Works in Visibility Culling 47

4.2 Research Questions and Goals 49

4.3 System Overview 50

4.4 Preprocessing for Occlusion Culling 51

4.5 Parallel Occluder Selection 52

4.5.1 View-Frustum Culling 53

4.5.2 Weighting Candidate Occluders 55

4.5.3 Identifying Active Occluder Set (AOS) 56

4.6 Conservative Culling with Hierarchical-Z Map 57

4.7 Evaluation 62

4.7.1 Implementation 62

4.7.2 Evaluation of Preprocessing Component 64

4.7.3 Runtime Performance 65

4.7.4 Evaluation of Occlusion Culling Component 66

4.7.5 Evaluation of LOD Processing Component 68

5 Dual-GPU System: Distributing Data Between GPUs with a Dynamic Load Balancer 76

5.1 Related Works in Multi-GPU Applications 77

5.2 Research Questions and Goals 78

5.3 System Overview 80

5.4 Parallelism for CUDA-OpenGL Interoperability 81

5.4.1 A Single CPU Thread 81

5.4.2 Multiple CPU Threads 82

5.4.3 Multiple Processes 82

5.5 Load-Balanced Data Distribution 83

5.5.1 Fundamental of View-Frustum Partitioning for Data Distribution 83

5.5.2 Parallel Dynamic Load-Balancing Algorithm 85

5.6 Synchronization and Inter-GPU Communication 87

5.6.1 Camera Synchronization 88

5.6.2 CUDA Inter-Process Communication 89

5.7 Implementation 90

5.8 Performance Evaluation 91

6 Conclusion and Future Works 94

6.1 Conclusion 94

6.2 Future Works 95

6.2.1 Applicability to 3D-Scanned Surface Models 95

6.2.2 In-Place GPU Memory Defragmentation 96

6.2.3 Load Balancing and Inter-GPU Communication in a Multi-GPU System . . 97

List of Figures

1.1 **The rendered Boeing 777 airplane model.** This airplane model contains approximately 223 million vertices and 332 million triangles that are organized in about 718 thousand objects. 3

1.2 **Performance comparison between CPUs and GPUs, excerpted from [1]. (a)** shows the floating-point operations per second for the CPU and GPU. **(b)** shows the memory bandwidth for the CPU and GPU. 6

2.1 **An example of the edge-collapsing operations. (a)-(d)** show the sequence of simplified objects generated by the edge-collapsing operations. At each operation, the intermediate results are recorded in the array structures at the bottom. **(e)** shows the final results after handling the boundary vertices and their associated triangles. . 18

2.2 **An example of data rearrangement.** (a) shows the original mesh with the collapsing results from Figure 2.1. (b) shows the rearranged data and the rearranged *ecol*. 19

2.3 **Examples of LOD model sequences. (a)-(d)** shows the four different levels of the Power Plant model, where the pairs of triangle/vertex are: (a)5.7M/2.5M; (b)1.9M/0.8M; (c)0.9M/0.4M; (d)0.2M/0.1M. **(e)-(i)** shows the five different levels of the Boeing 777 model, where the pairs of triangle/vertex are: (e)38.0M/25.0M; (f)26.6M/17.5M; (g)15.1M/10.0M; (h)7.4M/5.0M; (i)3.7M/2.5M. 21

2.4 **The concept of triangle reformation.** Brute-force rendering the LOD selected triangles gives a fragmented object. By reforming each triangle, all triangles can be reconnected. 27

2.5 **An example of triangle reformation. (a)** shows the parallel reformation process. The selected triangles are organized in a block of continuous memory, then each triangle is reformed by finding and using its corresponding *ecol*. **(b)** shows how to replace a vertex with a target one by walking through array *ECol* backwards. n is the amount of selected vertices of an object. The target index is g, which satisfies $g \leq n$. 29

2.6 **The performance comparison of two types of parallelizations of the triangle reformation algorithm. Both implementations are executed on a NVIDIA GTX 680 device with 4GB global memory.** 31

3.1 **An example of collecting the additional data on CPU.** The purple blocks replicated from the original objects represent the additional data required by GPU. The blue blocks are equivalent to the data already existing on GPU. 38

3.2 **An example of GPU defragmentation.** e_i, e_j, e_k, e_x and e_y are substituted with a source element from either existing data block or the additional data block in parallel. 41

3.3 **The evaluation of the GPU Out-of-Core algorithm on a NVIDIA Quadro 5000 device.** The performance of coherence-based out-of-core algorithm (*PC*) is compared with the performance of the other two out-of-core approaches (*PnC* and *NP*) with the Boeing 777 model. 42

3.4 **The scattered value pairs of defragmentation time and visible triangle/vertex numbers on a NVIDIA Quadro 5000 device.** 44

4.1 **Overview of the massive model rendering system.** 50

4.2 **An example of the parallel occluder selection.** Assuming that the model consists of 18 objects. 8 of them are classified into the COS. The size of AOS is fixed to be 3 objects. 53

4.3 **An example of view-frustum culling.** The green lines define the volume of the view frustum. The red bounding boxes stand for the objects outside the view frustum. 54

4.4 **An example of HZM.** The original depth image is rendered in the dimension of 512×512. The purple objects are active occluders. The size of AOS is set to 10, and all the ten levels of the HZM are shown. Each blue cell represents a pixel at the corresponding level. 58

4.5 **The concept of culling with the HZM.** **(a)** shows the the configuration of the viewpoint and the active occluders. The green lines define the view frustum, and the redline defines the viewing vector from the camera position. The purple objects are the active occluders; **(b)** shows all levels of the HZM constructed from the active occluders. The dimension of the finest level is 512×512; **(c)** shows the object whose visibility is determined by testing its depth value against an appropriately selected level of HZM; **(d)** represents the selected level of HZM. The red square represents the projected size of the object on the screen, and it overlaps with the four green blocks associating with the depth values from the active occluders. . . . 60

4.6 **The models rendered by the system.** The first and second rows are the rendered images of the Boeing 777 airplane model. The third and forth rows are the rendered images of the Power Plant model. I apply the Screen Space Ambient Occlusion algorithm (SSAO) [2] to improve shading quality. The dimension of these rendered images is 1024×1024. 63

4.7 **The occlusion culling result with the Boeing 777 model.** **(a)** is the rendered frame. **(b)** is the reference view, where the dark green lines define the volume of the view frustum. The purple objects are the active occluders. The red boxes represent the occluded objects. **(c)** shows only the active occluders and the occluded objects. 67

4.8 **The performance of LOD processing component with the Boeing 777 model on a NVIDIA GTX 680 device.** 72

4.9 **The breakdowns of the LOD processing performance with the Boeing 777 model on a NVIDIA GTX 680 device.** 73

5.1 **The overview of the dual-GPU system.** 80

5.2 **The view-frustum partitioning.** The indices $0-5$ represent the objects bounding boxes. $a-d$ stand for the desired primitive counts from the LOD selection algorithm. 84

5.3 **The concept of the dynamic load-balancing algorithm.** The whole screen is split by balancing the number of primitives distributed between GPUs. In this example, GPU_0 transfers a potion of the image to GPU_1. 85

5.4 **The principles of synchronizations and communications between GPUs.** 88

5.5 **The performance comparison among the three systems: Dual-GPU(B), Dual-GPU(NB) and Single-GPU.** 92

5.6 **The scattered value pairs of GPU out-of-core time and the number of triangles on a GPU from 600 rendered frames.** 93

List of Tables

3.1 **The comparison of the three GPU-out-of approaches:** ***Packing with Coherence (PC)*** **(my approach),** ***Packing without Coherence (PnC)*****, and** ***No Packing (NP)*****.** 43

4.1 **The preprocessing results on a single PC.** 64

4.2 **The runtime parameter configurations.** 65

4.3 **Overall performance on a NVIDIA GTX 580 device.** The results are averaged over all the frames along the created camera paths. 65

4.4 **The effectiveness of Occlusion Culling (OC) with the Boeing 777 model.** . . . 66

4.5 **The breakdowns of runtime performance with only the LOD processing component on a NVIDIA GTX 680 device.** The results are averaged over all the frames of the created camera paths. 74

4.6 **The performance on different GPU devices.** The value of N is set to be 3.5M for all devices. The number of corresponding triangles is 4.4M. All the devices are conducted with the same camera paths. We use a part of the Boeing 777 model that contains 69.2M vertices, 100.2M triangles and 295K objects. 75

5.1 **The performance breakdowns of Dual-GPU(B), Dual-GPU(NB) and Single-GPU systems.** 91

Chapter 1

Introduction

The number, size, and scope of digital datasets are immense in the era of information technology. No matter in industrial or academic communities, we are surrounded by a mass of digital content. More often than not, details of users are constantly being documented in an ensemble of databases and servers that make up the digital universe. An IDC white paper [3] calibrated the size and growth of the digital universe. At the end of the year 2011, information stored digitally was ten times larger comparing to what it was five years prior.

Many researchers have recognized the challenges in storing, distributing and accessing massive data, but finding methods to understand the knowledge and information embedded in such data is usually more challenging and important to data analysts. Many systems and tools have been developed to assist people in understanding such massive datasets. For example, data visualization tools are developed to help people comprehend rich information with graphic illustrations.

In the past, many researchers have dedicated their efforts to massive data visualization. Yoon et. al [4] discussed various techniques that enable interactive visualization of massive 3D models, where rendering performance is one of the fundamental requirements. A high rendering frame rate is usually required to perform real-time mechanical design, user-steered interactive visualization and visual analytics. However, the complexity of geometry data has increased dramatically, requiring gigabytes in storage, which has made the rendering process computationally intensive, thus impeding real-time visualization. Thus, designing and implementing algorithms around GPU architectures has been a major trend towards improving performance. The cost of intensive computations can be significantly reduced by delivering fine-grained parallel implementations. In this chapter, I would like to present the motivations behind GPU-based massive 3D model rendering.

1.1 Research Motivations on Massive Model Rendering

Massive model rendering techniques have been used in many applications. For instance, in geographical research, the data generated by a simulation can be very complex. Visualization tools allow geographers to communicate geospatial information and provide them with real-time visual feedback while exploring data, such as research in geovisualization [5] for analyzing earthquakes and volcano activities. In web map applications, massive model rendering plays an important role. Online platforms, such as Google Earth and Microsoft Virtual Earth, visualize cities in three dimensions [6]. By dealing with complex visual context, Google Earth allows users to search and interactively explore maps at a street level in many locations. In scientific visualization, simulating

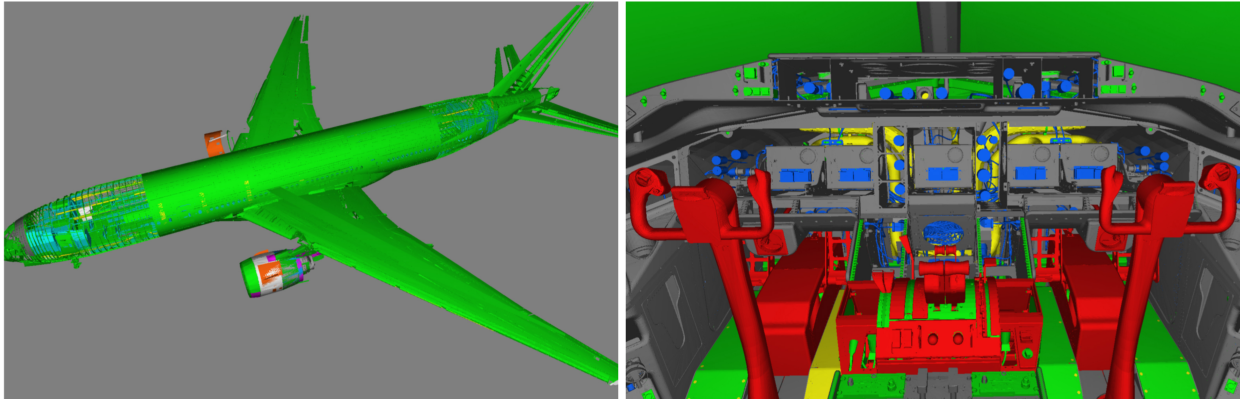


Figure 1.1: **The rendered Boeing 777 airplane model.** This airplane model contains approximately 223 million vertices and 332 million triangles that are organized in about 718 thousand objects.

and rendering three-dimensional phenomena of volumes, point clouds, and surfaces involve operations and computations over large datasets. Some research problems in scientific visualization were addressed in Johnson's paper [7]. An efficient rendering tool graphically illustrates scientific data so that a domain scientist can see the underlying structures and gain the insight knowledge of the data.

In my research, I concentrate on approaches for rendering massive 3D polygon models. A large-scale 3D model usually is the output in many research domains, such as mechanical engineering, scientific simulation, architectural design and game development. For example, CAD models from the mechanical design usually have many self-contained and fully functional parts that require hundreds of millions of triangles and vertices. The Boeing 777 airplane model, as shown in Figure 1.1, consists of more than 200 million vertices and more than 300 million triangles that consume approximately 6 gigabytes of memory.

In order to efficiently manage the model, the 3D geometric information is usually organized in a

multi-object manner. An object in the model may contain a relatively small number of polygon primitives as long as they are sufficient to describe the shape and topological properties. To have a complete description of the entire model, tens of thousands, or millions, of such individual objects are created and make it exceptionally complex.

1.1.1 Hardware Limitations

A major rendering problem is that massive models cannot be efficiently rendered via brute-force methods due to the limited computational power and memory available on a workstation. The power of modern computer hardware cannot satisfy users' performance requirements for real-time model investigation.

To overcome such hardware limitations, many approaches employ mesh simplification algorithms to speed up performance, where an input model is progressively simplified until it becomes manageable by hardware renderers. Progressive Meshes [8, 9] and Quadratic Error Metric [10, 11] are the two popular mesh simplification algorithms that support continuous Level-Of-Details (LOD). The continuities between levels of geometric details are preserved during the simplifying procedure. However, because of the sequential nature of progressive operations in those mesh simplification algorithms, classic CPU-based solutions cannot handle the massive number of triangles efficiently. Although the ideas of parallel mesh simplification have been proposed [12, 13], the dependencies introduced in those progressive algorithms have not been completely removed.

In addition to issues in execution speed, there are the challenges regarding memory usage. Histor-

ically, the models we consider are typically too large to store entirely in main memory. For each frame rendered, the renderable data is transferred from external memory (e.g., disks) to main memory. With recent hardware improvements, storing several gigabytes in main memory is no longer a problem. When rendering the model, a standard rendering pipeline, such as OpenGL, requires that all triangles and vertices reside on the GPU. However, the capacity of GPU memory is usually far less than that of CPU main memory. Triangles and vertices must be transferred from CPU to GPU through PCI Express (PCIe) buses for each frame or timestep. Although mesh simplification and visibility culling algorithms remove many unnecessary triangles and objects, the size of geometric data in need of transfer is usually too large, where sending it would be throttled by the limited bandwidth of PCIe buses, resulting in a major bottleneck in performance.

1.1.2 Advantages of GPU Parallel Architectures

Processing a massive model needs lots of computational power. In recent years, graphics hardware, as a massively parallel architecture and a commoditized computing platform, has been praised for the significant performance increase and the capability for general-purpose computation. The improvements in graphics hardware create the possibility of transplanting classic CPU algorithms to the many-core GPU architectures. Architectures of the CPU and GPU are fundamentally different. As introduced by Lee et. al [14], they are designed based on different philosophies. Chips in the CPU are built to provide fast response times to a single task with only a small number of processing cores. Graphics chips in the GPU are designed with Single-Instruction, Multiple-Data (SIMD) computing model in mind, where different data elements are processed by the same sequence of

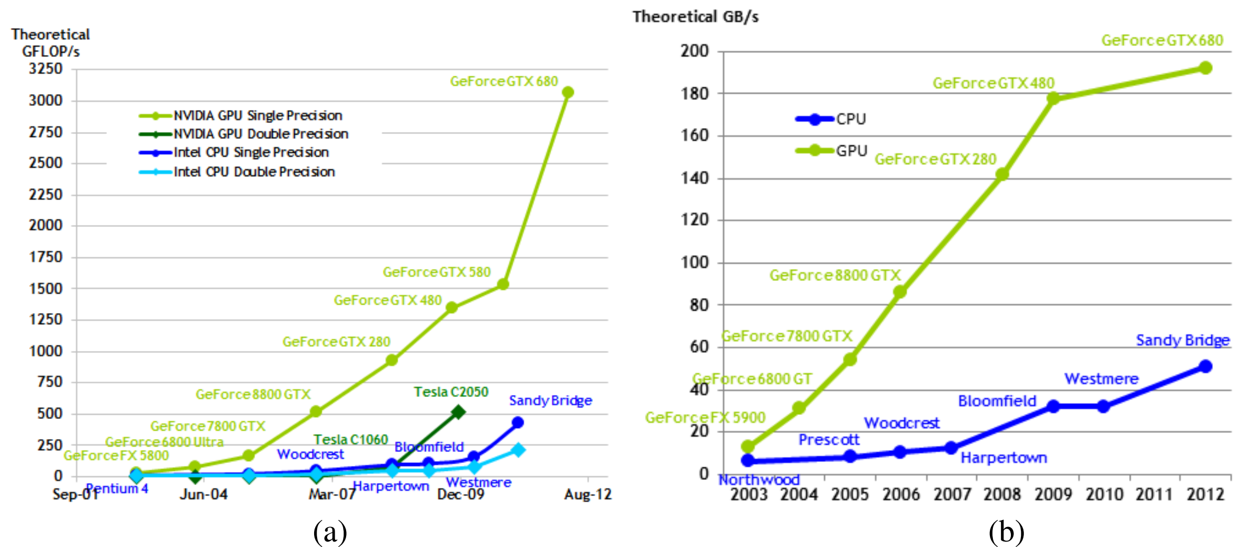


Figure 1.2: **Performance comparison between CPUs and GPUs, excerpted from [1].** **(a)** shows the floating-point operations per second for the CPU and GPU. **(b)** shows the memory bandwidth for the CPU and GPU.

arithmetic instructions. GPUs devote many processing transistors organized on arrays of computing units. This organization allows millions of threads to be launched simultaneously, while maintaining a logical structure as either one-dimensional, two-dimensional, or three-dimensional thread blocks.

When compared to CPUs, GPU architectures employ higher memory bandwidth by utilizing caches, shared memory and built-in memory controllers [1]. They have wider and faster buses and higher rates of memory clock. As illustrated in Figure 1.2, NVIDIA gives performance comparison between the CPUs and GPUs for the past few years. By the year of 2012, the NVIDIA GTX 680 is about 4x faster than the CPU in memory bandwidth.

However, designing parallel algorithms for the GPU is not trivial. The most important condition of a SIMD model is the independence of the calculations for each data element. Many traditional

algorithms have straightforward CPU implementations. But removing the dependence present in CPU sequential executions of the algorithm is not an easy task. For example, the sorting algorithms have many CPU implementations, but implementing them on the GPU is a challenging problem because of the unavoidable inter-thread or inter-block communications [15]. Other critical issues for the GPU programming come from load balancing and resource utilization. Details of the issues are addressed in Chen et. al [16]. GPU schedulers cannot efficiently handle unbalanced workloads for the threads. Also, insufficient parallelism on the GPU will underutilize computational resources. Thus, when designing parallel algorithms for the GPU, it is essential to deliver a fine-grained execution scheme to help increase parallel granularity while ensuring a balanced workload among all threads.

1.2 Contributions

In my dissertation, there are three major contributions towards the state-of-arts in massive model visualization:

1. **Parallel Mesh Simplification Algorithm.** I propose a parallel mesh simplification algorithm on the GPU. I design a GPU-friendly data structure that successfully removes the data dependency introduced in the classic LOD algorithms. As a result, the parallel algorithm supports triangle-level parallelism.

2. **GPU Out-of-Core Algorithm.** In order to efficiently transfer data from the CPU main memory to the GPU memory, I employ a frame-to-frame coherence strategy, where only

data differing across frames is transferred for each frame rendered. To support this coherence strategy, I also design a novel parallel algorithm for organizing the GPU memory.

3. **GPU-based Rendering System.** I propose a rendering system that contributes to high-performance visualization applications. The system seamlessly integrates the parallel mesh simplification algorithm with the occlusion culling algorithm in a unified parallel scheme. In order to further improve performance, I propose a dual-GPU system that distributes workloads between the GPUs with a dynamic workload balancer.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, I present a novel parallel mesh simplification algorithm. In Chapter 3, I describe a novel out-of-core algorithm that transfers the vertices and triangles from the CPU and manages the their storage on the GPU. The massive model rendering system and its experimental results are presented in Chapter 4. It seamlessly integrates the parallel mesh simplification algorithm with occlusion culling algorithm. To further improve performance, I describe a dual-GPU system in Chapter 5. By using a dynamic load balancer, the workloads distributed between GPUs are well balanced. Lastly, I conclude my work and propose future work in Chapter 6.

Chapter 2

Parallel Mesh Simplification Algorithm

The concept of mesh simplification has been proposed for several decades. Basically, if an object is far away, we can represent it with less details. In this chapter, I first review the stat-of-art approaches, then describe the active research questions and my research goals. After that, I give the detailed explanations of my GPU-based parallel simplification algorithm.

2.1 Related Works

Given an input 3D model, a less complicated but visually faithful representation can be approximated as the alternative for rendering. The basic simplification approach is known as discrete Level-Of-Detail (LOD), such as the approach presented by Luebke et. al [17], that creates a set of simplified versions of an 3D object offline. The farther is the object away from the camera, the lower level (less details) is selected to represent the object for rendering. Although the discrete

LOD approach supports the instant access to a LOD representation during the runtime, it requires too much memory for storing those offline simplified versions, especially when dealing with the massive model containing the huge amount of polygon primitives. Moreover, discrete approaches may lead to serious ”popping” artifacts when transiting from one LOD representation to another. Some methods, such as [18], blended neighbor LOD representations in image space to minimize the artifacts. But because of the nature of discrete LOD approaches, smooth transitions are still hard to achieve. To preserve high visual fidelities in the process of mesh simplification, many continuous LOD algorithms use a series of geometric primitive operations, such as progressive modifications [8, 10, 19], selective refinement [20, 21] and parallel LOD algorithms [12, 13].

The most well-known continuous LOD algorithm is *Progressive Meshes* [8, 22] that simplifies a triangular mesh based on a sequence of edge-collapsing operations, and different LOD representations can be recovered by applying a prefix of invert operations that splits the base mesh iteratively. In each edge-collapsing operation, an optimal edge is selected and removed by merging its two vertices, then the corresponding triangles are eliminated, consequently the mesh topology is modified. The operations continue until the mesh can not be simplified any more (means it reaches the base mesh). The order of these operations indicates how the details of a 3D mesh are reduced. The more operations are performed, the less details are reflected for the original mesh.

One of the questions is how to find the position of the target vertex for the two source vertices of the collapsed edge. Garland and Heckbert [10] and Lindstrom [23] used the Quadric Error Metrics (QEM) to find the coordinates of the target vertex by minimizing the sum of squared distances to face planes. Ju et. al [24] employed QEM procedure for hermit data contouring methods. Later,

Garland and Zhou [25] extended QEM to any dimension. For the purpose of memory efficiency, Garland and Heckbert [11] and Swarovsky [19] used the coordinates of a source vertex for the target one, so that they can avoid memory allocation for any new vertices.

The camera is an important parameter and must be considered during the rendering process. Researchers have dedicated their efforts to view-dependent rendering. The details of a model should always appear at the positions close to the camera. When the camera changes, the detail positions over the model should be changed accordingly. Selective refinement algorithms, such as Hoppe [9], Xia et. al [20] and Luebke and Erikson [26], compute different detail levels over different regions of the model. The important contribution of selective refinement algorithms is that they organize edge-collapsing information into hierarchical structures (e.g., vertex hierarchy [9, 20] or octree vertex clusters [26]) rather than a linear sequence. Hoppe [27] partitioned the terrain model into blocks. The mesh patch in each block is simplified using the Progressive Meshes algorithm and by locking block boundaries. The neighbor blocks are then merged to a larger block at the upper level. This process is repeated until reaching the top level. Yoon et. al [21] represented the input model with multiple progressive meshes in a clustered hierarchy (CHPM). During the runtime, by performing refinement operations and cluster merging, they selected a set of active clusters to generate the desired LOD model. Cignoni et al. [28] used a binary tree to partition the input model then represented it as the multiresolution static LODs. Similarly, Gobbetti et. al [29] represented the input model with a volume hierarchy. Shaffer et. al [30] used an external memory multiresolution approach for massive polygonal models. The central component of their method is an octree that encodes multiple levels of details. The cells at the bottom level of the octree are merged to

form coarser mesh approximations in the higher levels.

Parallel algorithms were also proposed with the GPU architectures. DeCoro and Tatarchuk [12] presented a vertex clustering method using the shader-based fixed GPU graphics pipeline. Each cluster can be independently computed through geometry shader stage, but the dependency in each cluster still remains. Hu et al. [13] introduced a parallel approach for the view-dependent Progressive Meshes, where vertex dependencies were not fully considered during a cascade of vertex splitting events. The popping artifacts occured in their results. Derzapf et al. [31] encoded the dependency information of Progressive Meshes into a compact data structure; then later, Derzapf et al. [32] extend their work for the out-of-core Progressive Meshes. Peng and Cao [33] uses an array structure to support triangle-level parallelism. Peng and Cao [34, 35] also claimed that the limited GPU memory is a problematic issue. They integrated the algorithms of the parallel simplification and the GPU out-of-core to render complex models.

2.2 Research Problems and Goals

As introduced in Section 2.1, traditional simplification algorithms are not data parallel in nature and do not have trivial GPU implementations. They rely on hierarchical data representations to support view-dependent rendering. But, the data dependency introduced between levels of the hierarchy makes the polygon primitive selection very costly, especially when the input dataset is massive. Given a specific camera configuration, a set of active nodes is selected by a top-down traversal. When dealing with massive models, the hierarchy could become very deep and complex,

and traversing it would be very time-consuming. Also, fetching the primitives in those active nodes from the disk memory would be a slow process due to the high latency of I/O communications.

Another problem comes from edge-collapsing operations. When collapsing an edge, we must pick the edge based on an intermediate model generated by the previous collapsing operation. The dependency between collapsing operations makes the simplification algorithms computationally intensive. Since a massive model may contain hundreds of millions of edges, collapsing them one-by-one would impede real-time rendering. Although the parallel implementations were proposed by some previous works, the data dependency has not been successfully removed.

Thus, the goal of designing a parallel algorithm is to remove the data dependencies introduced in both hierarchical structures and edge-collapsing operations but still preserve view-dependent features, where the model changes shape based on the direction it is viewed from. To achieve this research goal, I employ a GPU-friendly preprocessing step. I also design a dynamic LOD selection approach to preserve view-dependent features. A novel data structure is used to store edge-collapsing information and supports triangle-level parallelism for the online LOD model generation.

2.3 GPU-Friendly Preprocessing

My parallel algorithm originates from the idea of edge-collapsing. In this section, I would like to introduce an edge-collapsing approach for the step of preprocessing. As a result, the new data representation supports LOD model generation on the GPU at runtime.

Edges are collapsed iteratively until reaching the simplest version of the object. At each edge-collapsing operation, an edge, denoted as (v_1, v_2) where v_1 and v_2 are the vertices forming the edge, is removed. Two steps are involved during this process: (1) collapsing the edge by merging v_1 and v_2 to a target vertex, $\bar{v}$; (2) removing v_1, v_2 and all the triangles having both vertices.

2.3.1 Key Criteria of Edge-Collapsing

To better control the quality and behavior of collapsing operations, the following criteria are employed:

Position of the target vertex. Obviously, the position of target vertex $\overline{v}$ can be either the endpoints of the edge, v_1, v_2, or a new position (e.g., $\overline{v} = (v_1 + v_2)/2$). The choice of $\overline{v}$ depends on the intended applications. In my approach, I use the endpoints for $\overline{v}$. According to the discussion in [11], using the endpoints require less storage.

Boundary edge constraint. In many 3D models, the disconnected faces separated by borders and holes are usually important visual features. To preserve them, boundary edges are restricted to be non-collapsible. A boundary edge is defined as the edge existing in only one triangle, and the two vertices of the edge are the boundary vertices. Note that any edge containing boundary vertices cannot be collapsed by moving a boundary vertex to the other. Using this constraint, the lowest level of detail of a mesh is represented by the mesh with only boundary vertices rather than a single triangle.

Equation 2.1 shows how the constraint is applied at a collapsing operation. Note that, the edge is

(v_a, v_b), where $a < b$; Q is the set of boundary vertices. "Non-collapsible" indicates the edge is a boundary edge.

$$Collapse(Edge) = \begin{cases} v_a \rightarrow v_b, & (v_a \notin Q) \\ v_b \rightarrow v_a, & (v_a \in Q, v_b \notin Q) \\ non-collapsible, & (v_a, v_b \in Q) \end{cases} \tag{2.1}$$

Error function for optimal edge selection. At each collapsing operation, we want to select the edge that causes minimal visual changes. I use the error function introduced by Melax [36], where the selected edge has the minimal value calculated from Equation 2.2.

$$cost(v_a, v_b) = \|v_a - v_b\| \times \max_{t_i \in T_{v_a}} \{ \min_{t_j \in T_{v_a v_b}} \{1 - \frac{t_i.normal \cdot t_j.normal}{2}\}\} \tag{2.2}$$

Note that, T_{v_a} is the set of the triangles having vertex v_a, and $T_{v_a v_b}$ is the set of the triangles having both vertices v_a and v_b. By involving the face normals, the calculation balances the edge length and its curvature.

2.3.2 Data Rearrangement

The order of edge-collapsing operations indicate how the details of a 3D object are reduced. The storages of vertices and triangles are rearranged based on this order. In practice, the first removed vertex during the collapsing procedure is restored to the last position in the vertex set; and the last

removed vertex is restored to the first position. The same rearrangement is applied to the triangle set as well. As a result, the order of storing the rearranged data reflects the levels of details. If needing a coarse version of the model, it will be sufficient to use a small number of continuous vertices and triangles starting from the first element in the sets.

In order to eliminate the dependencies between those operations, the collapsing information is recorded into an array structure, called $ECol$. Similar to the data structure described in [19], each element of *ECol* corresponds to a source vertex, and its value is the index of the target vertex that it merges to. Here, I define the function $ecol(i)$ that recovers the value of the ith element of *ECol* array. Meanwhile, I record the vertex count and triangle count remaining in the object after each operation. Since an operation removes only one vertex but varied number of triangles, I employ a structure called Map, to record the relation between the vertex count and the triangle count. If saying j is the remaining vertex count after an operation, the value, recovered from the function $map(j)$, is the triangle count in the current version of the object.

There are five steps in each operation as follows:

1. **Selecting and collapsing the minimal-cost edge.** Each edge is assigned with a cost value calculated from Equation 2.2. After the minimal-cost edge is selected, its source vertex, v_{src}, is merged to the target vertex, v_{tar}. The relation between v_{src} and v_{tar} is recorded in the array $ECol$, and we know $ecol(src) = tar$.

2. **Recording the permutation information for vertex data rearrangement.** Let us say m is the number of vertices in the currently operating version of the object. The vertex index *src*

should be replaced with m. In other words, in the later data rearrangement process, the vertex v_{src} need to be restored to the mth position in the vertex set. To record this intermediate information, I use an array structure called $PermuteV$. The ith element of $PermuteV$ corresponds to a source vertex, and the value of the element is its new vertex index that it should be permuted to. I denote $permuteV(src) = m$ for accessing the elements of $PermuteV$.

3. **Mapping the vertex-triangle counts.** When merging v_{src} to v_{tar}, all the triangles having v_{src} and v_{tar} are removed. Let us say k triangles are removed and n triangles remain after this operation, I update the array Map so that $map(m-1) = n-k$.

4. **Recording the permutation information for triangle data rearrangement.** Let us say n is the number of remaining triangles, and the two triangles, t_{r_1} and t_{r_2}, are removed. r_1 and r_2 ($r_1 < r_2$) are the triangle indices. Similar to the Step 2, I use an array called $PermuteT$. The ith element corresponds to a removed triangle. The value of the element is its new triangle index. The new indices are accessed with the functions of $permuteT(r_1) = n$ and $permuteT(r_2) = n-1$.

5. **Cleaning up.** I delete v_{src}, t_{r_1} and t_{r_2} from the vertex and triangle sets respectively, then update m and n for the next operation.

Figure 2.1 shows an example of the edge-collapsing operations. Figure 2.1 (a) is an object composed of 7 vertices and 8 triangles. Initially, the last element of the Map array is set to be 8, so that $map(7) = 8$. The set of vertices $Q = \{v_3, v_4, v_5, v_7\}$ are the boundary vertices. In this example, let us assume that the error function only considers edge length. The costs for the boundary

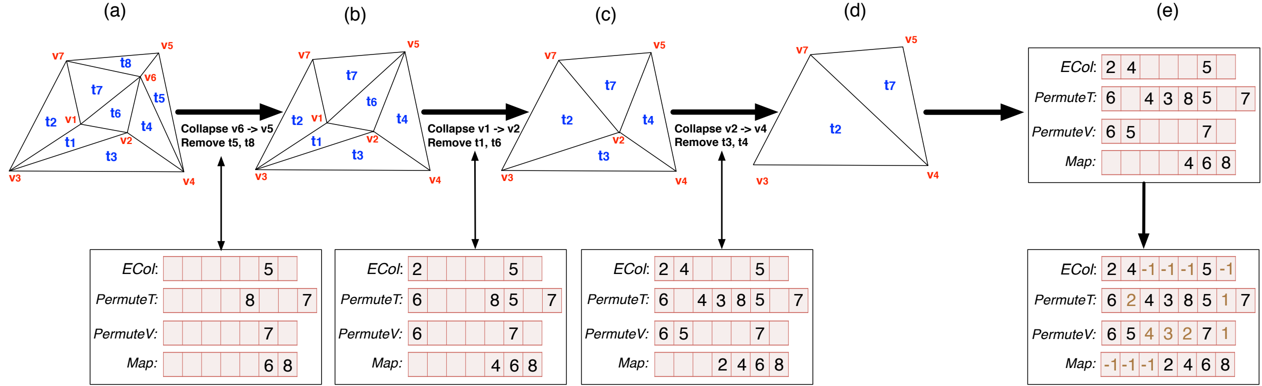


Figure 2.1: **An example of the edge-collapsing operations. (a)-(d)** show the sequence of simplified objects generated by the edge-collapsing operations. At each operation, the intermediate results are recorded in the array structures at the bottom. **(e)** shows the final results after handling the boundary vertices and their associated triangles.

edges are infinitely large. The shortest edge is collapsed at each operation using Equation 2.1. Figure 2.1 (a-d) illustrates all the operations to simplify the original object.

In Figure 2.1 (b), m is equal to 6, and n is equal to 6. In Figure 2.1 (c), after collapsing the source vertex v_1 to the target vertex v_2, $ecol(1) = 2$, and $permuteV(1) = 6$ that is equal to the value of m. Meanwhile, The triangles t_1 and t_6 are removed, and $permuteT(1) = 6$ that is the value of n; and $permuteT(6) = 5$ that is the value of n-1. Since only two triangles are removed, the Map is updated so that $map(5) = 4$, which means 5 vertices and 4 triangles remain in the object. Figure 2.1 (d) is the simplest version of the object that contains only boundary vertices.

Figure 2.1 (e) handles the boundary vertices. The invalid value (e.g., -1) associated to the boundary vertices is assigned to the blank elements of $ECol$, so that $ecol(i) = -1(i \in Q)$ and $map(j) = -1(j \in [1, m-1])$. In the $PermuteV$, From left to right, each blank element is assigned with the current value of m, where m is decreased by 1 after each assignment. The $PermuteT$ is completed

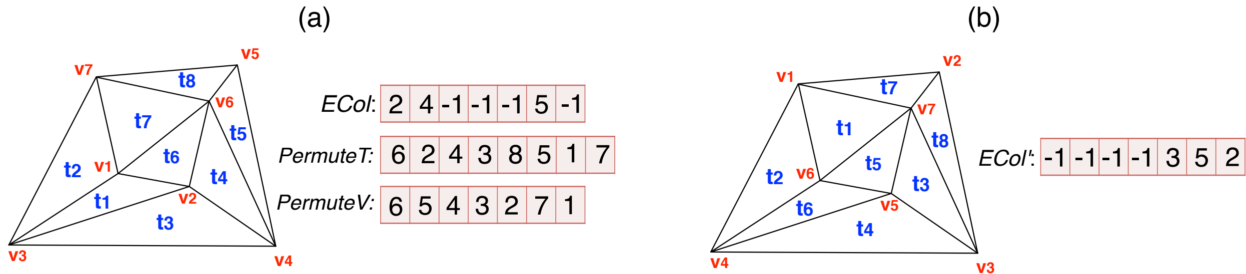


Figure 2.2: **An example of data rearrangement.** (a) shows the original mesh with the collapsing results from Figure 2.1. (b) shows the rearranged data and the rearranged *ecol*.

in the same way by decreasing the value of n.

Algorithm 1 Re-arranging the data

Input: O, $ECol$, $PermuteV$, $PermuteT$
Output: O', $ECol'$

for each $v_i \in O.V$ **do**
$ECol'[permuteV(i)] \leftarrow permuteV(ecol(i));$
$O'.V'.v_{permuteV(i)} \leftarrow O.V.v_i;$
end for
for each $t_i \in O.T$ **do**
$O'.T'.t_{permuteT(i)} \leftarrow O.T.t_i;$
end for

The storages of vertices and triangles are rearranged based on the values recorded in the $PermuteV$ and $PermuteT$. Let's define the original object as $O = (V, T)$, where V is the vertex set $\{v_1, v_2, \ldots, v_p\}$ and T is the triangle set $\{t_1, t_2, \ldots, t_s\}$. The rearranged object is defined as $O' = (V', T')$ and illustrated in Figure 2.2. Algorithm 1 describes the data rearrangement algorithm. After rearrangement, if a simplified version of O' contains i vertices ($k \in q, p$), where q is the number of boundary vertices, The coresponding number of triangles can be retrieved by $j = map(i)$. As a result, the desired LOD object is generated with the subset of V', $\{v_1, v_2, \ldots, v_i\}$, and the subset of T', $\{t_1, t_2, \ldots, t_j\}$. The algorithm for generating the new shape with the selected data is presented in Section 2.5.

2.4 LOD Selection

Now, the question is how to determine the level of detail (or the desired geometric complexity) of the objects. In other words, I want to find out the desired number of vertices and triangles for each object at a given viewpoint. This problem is known as *LOD Selection*. Conceptually, an object can be rendered at any level of detail. But due to the limited size of the GPU memory, the total number of polygon primitives must be budgeted based on the memory capacity. Also, we need to consider how to allocate the budget to the objects so that both memory constraints and visual requirements can be satisfied.

In this section, I introduce a two-pass LOD Selection algorithm that effectively distributes the hardware-constrained primitive budget. The example renderings with my LOD algorithm are shown in Figure 2.3.

2.4.1 First-Pass Algorithm

Ideally, an object farther away from the viewpoint deserves a lower detail level. In practice, people usually examine the size of the screen region occupied by an object. The larger size is the region, the higher level the objects detail should be. Based on this idea, Funkhouser et al. [37] solved the LOD selection as a discrete optimization problem, where the appropriate detail levels are determined by constraining the total number of associated primitives within a given maximal limit. Wimmer et al. [38] re-evaluated the problem and provided a closed-form expression to solve the LOD selection cheaply. Their approach took only the object's screen area into account.

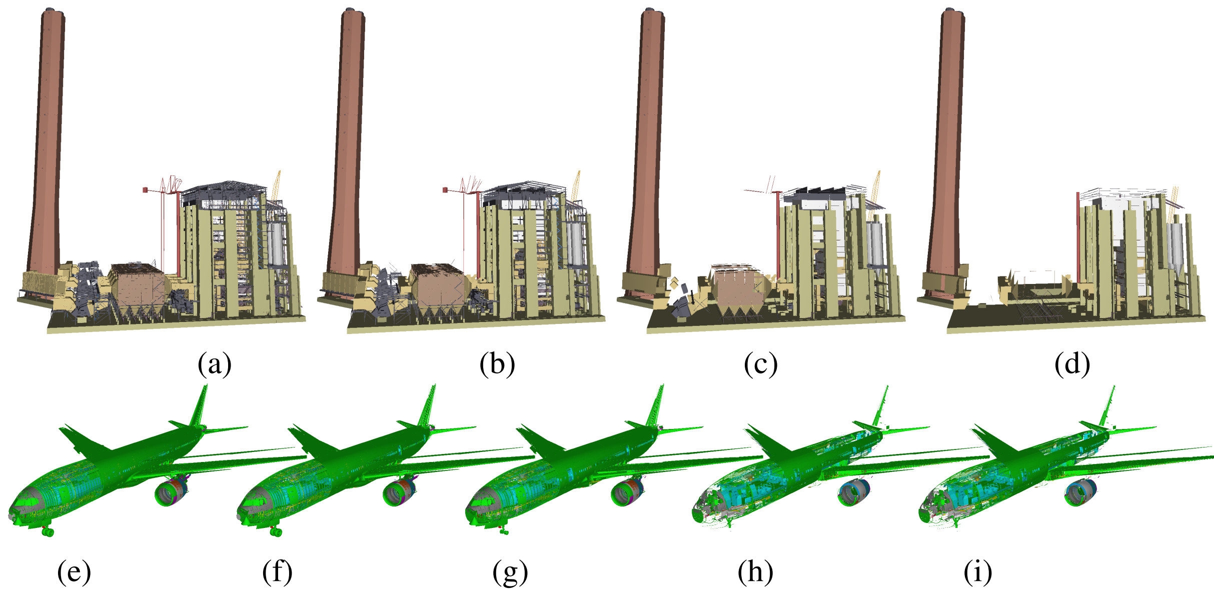


Figure 2.3: **Examples of LOD model sequences. (a)-(d)** shows the four different levels of the Power Plant model, where the pairs of triangle/vertex are: (a)5.7M/2.5M; (b)1.9M/0.8M; (c)0.9M/0.4M; (d)0.2M/0.1M. **(e)-(i)** shows the five different levels of the Boeing 777 model, where the pairs of triangle/vertex are: (e)38.0M/25.0M; (f)26.6M/17.5M; (g)15.1M/10.0M; (h)7.4M/5.0M; (i)3.7M/2.5M.

Nowadays, the complexity of 3D environments has significantly increased. The objects in a 3D environment may have dramatically different shapes with widely varying spatial ratios. It is possible that a far-away object, which should deserve a lower level of detail, has a much larger screen area than a closer one. In addition, objects are usually associated with different number of triangles due to the nature of the original design. The object with more triangles should rationally be approximated with a higher level than those with fewer triangles, though the former one may be farther away from the viewpoint. By considering all these aspects, I re-evaluate the LOD selection problem and provide a method that can be implemented in parallel on the GPU. Below, I explain my solution in detail.

LOD Selection Metric. In a multi-object model, the desired level of detail of the *i*th object is

represented with a pair of vertex count and triangle count, denoted as vc_i and tc_i, respectively. The vc_i is computed using the Equation 2.3.

$$vc_i = N \frac{w_i^{\frac{1}{\alpha}}}{\sum_{i=1}^{m} w_i^{\frac{1}{\alpha}}}, \text{where } w_i = \beta \frac{A_i}{D_i} P_i^{\beta},\ \beta = \alpha - 1 \tag{2.3}$$

N is the user-defined maximal vertex count (e.g., determining it based on the size of GPU memory). vc_i is computed out of the total of m objects. A_i denotes the projected area of the object on the image plane. To compute A_i efficiently, I estimate it using the area of the bounding rectangle of the projected Axis-Aligned Bounding Box (AABB) of the object on the image plane. The exponent, $\frac{1}{\alpha}$, is a factor aiming to estimate the object's contributions for model perception, refer to the benefit function detailed in [37]. D_i is the shortest Z-depth distance from the corner points of the AABB. P_i is the total number of vertices of the object.

In the preprocessing step, I record the remaining number of triangles into the array Map after each collapsing operation (see Section 2.3.2). At runtime, when vc_i is computed, the corresponding tc_i can be easily retrieved from the array Map by $tc_i = map_i(vc_i)$. The Equation 2.3 is implemented on the GPU by using NVIDIA CUDA Thrust library [39].

Pixel Error Threshold. As we know, no matter how complex an object is, the object is scan-converted into pixels before displaying it on the screen. At a location far away from the viewpoint, a large object may be projected to a very small region of the screen (e.g., less than one pixel) so that the shape of the object may not be captured by people's visual perception. Because of this nature, I use a *Pixel Error Threshold* (PET) as a complementary criteria for my LOD selection method.

if the value of A_i in Equation 2.3 is smaller than a given value of PET, vc_i is set to be zero. By doing this, only the objects whose screen areas are large enough are allocated with the cuts from the budget N. This constraint meets the workflow of view-dependent rendering, and removes the objects without losing the visual fidelity.

2.4.2 Second-Pass Algorithm

In many complex models, the disconnected faces separated by borders and holes are important visual features. As described in Section 2.3.1, I preserve those important features by restricting the boundary edges to be non-collapsible, and the simplest version of an object contains only boundary vertices (the vertex set Q). In other words, the lower bound of the desired vertex count vc_i is equal to the size of Q_i. Also, each object has the maximal number of vertices to form its finest representation. This number is the upper bound of vc_i. Let us define the size of Q_i as min_i and define the maximal vertex number as max_i. After obtaining the vc_i from Equation 2.3, if $vc_i < min_i$ or $vc_i > max_i$, we cannot use it to generate a LOD representation.

Here, I employ Equation 2.4 to map an out-of-bound vc_i into the value range of $[min_i, max_i]$. In Equation 2.4, the parameter $MinT$ ($MinT \in (0,1)$) is a threshold for the lower bound of vertex count. In the case $vc_i < min_i$, $MinT$ is used to evaluate how close vc_i is to the min_i. If $vc_i/min_i < MinT$, I ignore the visual contribution of this object and set its vc_i to be zero; otherwise, it is set to be min_i indicating the object will be rendered with the simplest representation.

$$\bar{vc}_i = \begin{cases} vc_i, & (vc_i \in [min_i, max_i]) \\ max_i, & (vc_i > max_i) \\ min_i, & (vc_i/min_i \in [MinT, 1]) \\ 0, & (vc_i/min_i < MinT) \end{cases} \tag{2.4}$$

$\bar{vc}_i$ is guaranteed to be valid after Equation 2.4. However, if adding up all the vertex counts, the sum may not match the given budget N. Let's denote N' to be the sum of all $\bar{vc}_i$. If many objects are in the cases of $vc_i > max_i$ or $vc_i/min_i < MinT(vc_i \neq 0)$, their $\bar{vc}_i$ values are smaller than the corresponding vc_i. Consequently, N' is smaller than N. As a result, the renderer cannot obtain the workloads as it expects so that we would underutilize the GPU memory and its computation resources. On the other hand, if most of objects are in the case of $vc_i/min_i \in [MinT, 1]$, more vertices are added to them, and N' is bigger than N. As a result, the renderer is assigned with too many workloads that would be over the capacity of GPU memory.

To avoid those problems, I design a new algorithm to redistribute the remaining budget ($N' < N$), or further reduce them ($N' > N$). Then, the renderer is guaranteed with the exact workload as the N gives. Now, let me first describe the redistribution algorithm in the situation of $N' < N$. There are three steps to redistribute the remaining budget. The details are explained as follows (also refer to Algorithm 2):

1. **Sorting key-value pairs.** At a specific viewpoint, we want to distribute the remaining budget, $\Delta N = N' - N$, to the visually significant objects. In Equation 2.3, w_i defines the visual

Algorithm 2 LOD Second-Pass Procedure

Input: L, $\bar{vc}$, max, N', N
Output: $\bar{vc}$
 if $N' < N$ **then**
  Sorting L in decrease order of $L.weights$ **in parallel**;
  for ith element in L **in parallel do**
   $j \leftarrow L_i.idx$;
   $VDif_i \leftarrow max_j - \bar{vc}_j$;
  end for
  Prefix sum then **binary search** $VDif$ **in parallel**;
  for ith object in k binary-search selected objects **in parallel do**
   $j \leftarrow L_i.idx$;
   $\bar{vc}_j \leftarrow max_j$;
  end for
 end if
 if $N' > N$ **then**
  Sorting L in increase order of $L.weights$ **in parallel**;
  for ith element in L **in parallel do**
   $j \leftarrow L_i.idx$;
   $VDif_i \leftarrow \bar{vc}_j$;
  end for
  Prefix sum then **binary search** $VDif$ **in parallel**;
  for ith object in k binary-search selected objects **in parallel do**
   $j \leftarrow L_i.idx$;
   $\bar{vc}_j \leftarrow 0$;
  end for
 end if

weights. A larger value of w_i indicates that the corresponding object should be viewed with more details. I define the set of key-value pairs $L = \{< weight, idx >\}(idx \in [0, m))$, where idx is an object index, m is the total number of objects, and the $weight$ is equal to w_{idx}. I sort the set L by the values of weights so that the visually significant objects are restored to the front in L.

2. **Identifying the number of objects.** I need to find out the number of objects having increased details. First, I identify how many vertices needs to be added to an object if rep-

resenting it at the highest level of details. I introduce the array $VDif$, where $VDif_i = max_j - \bar{vc}_j$. Here, i corresponds to the ith element in the sorted L and $j = L_i.idx$. Second, the prefix sum operation is performed over the array $VDif$. Third, I employ a binary search procedure over the prefix-summed $VDif$. Then, the first i objects will increase their details from ΔN if $VDif_i \leq \Delta N < VDif_{i+1}$.

3. **Redistributing ΔN for those selected objects.** I increase the number of vertices of those selected objects to their maximums, and the final complexity matches the given budget N.

In the situation of $N' > N$, I use a similar algorithm as above. First, the set L is sorted according to the increase order of the weights. Then, I compute the array $VDif$ that $VDif_i = \bar{vc}_j$, where $j = L_i.idx$. The the values in $VDif$ indicate the amount of vertices that the objects can reduce by. After applying the operations of prefix sum and binary search, I reduce the $\bar{vc}$ of those selected objects down to zero, since their weights indicate that they are the least important object in visual appearance.

2.5 Triangle Reformation on the GPU

A simplified object is generated by using the selected portion of vertices and triangles. with the rearranged datasets, the selected data portions can be quickly identified as long as we known the desired number of vertices and triangles. When the selected data portions are ready on the GPU, we have to reshape the triangles with the selected vertices before rendering them. Of course, the renderer would be able to render them without any reshaping operation. But the result would

be a fragmented object rather than an appropriately simplified one. As shown in Figure 2.4, the fragmented object does not maintain sufficient topological connections so that the rendered image is not acceptable. The new model representation should not have any quality loss in its view appearance. With the selected vertices, the shapes of triangles need to be reconfigured to seal those undesired holes so that they can be reconnected and construct watertight surfaces. In the following subsections, I first discuss the parallelism for reconnecting the selected triangles, then give the details of the reformation algorithm.

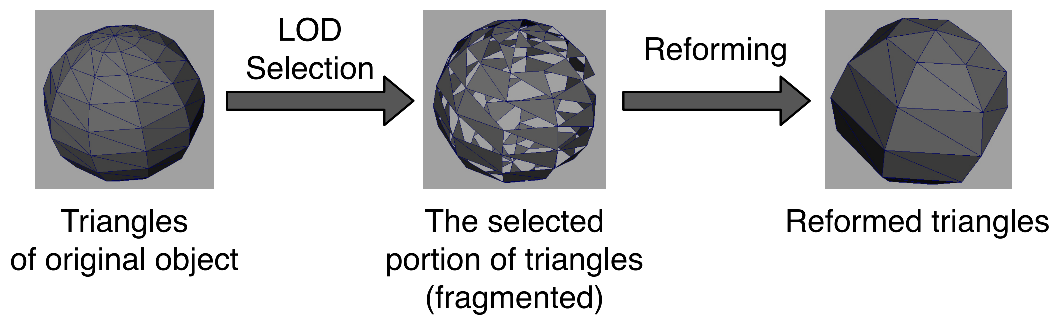


Figure 2.4: **The concept of triangle reformation.** Brute-force rendering the LOD selected triangles gives a fragmented object. By reforming each triangle, all triangles can be reconnected.

2.5.1 Parallelism and Storage Scheme

Obviously, the straight-forward parallelism is to implement the algorithm at an object level, where one GPU thread handles one object. As we know, modern graphics chips are designed to do massive parallel work and allow tens of thousands threads to be executed concurrently. If using the object-level parallelism, GPU resources would be underutilized, especially when the number of objects is less than the number of concurrent threads. In addition, the object-level parallelism

can cause load-balancing problems. objects contain different number of triangles. The threads handling the objects with many triangles could cause significant performance loss due to heavy global memory traffic and branch divergence. For example, the largest object in the Boeing 777 model contains more than 30 thousand triangles while the smallest one has only one triangle. To avoid such load-balancing issues, I employ a triangle-level parallelism that assigns each thread with one triangle, so that a sufficient number of GPU threads will be created simultaneously and the workload will be balanced.

In terms of data storage on the GPU, the natural way is to allocate separate memory blocks for the objects. Then, to render them, we can pass them through the graphics pipeline one-by-one. However, due to the large number of objects in the model (e.g., more than 700 thousand objects in the Boeing model), it would be a high cost to launch so many rendering calls. Thus, my design will concatenate the vertices and triangles selected from different objects into a single array structure as an OpenGL buffer object, as illustrated in Figure 2.5 (a), so that the objects can be rendered by using only one rendering call.

2.5.2 Triangle-Level Parallel Algorithm

Given a triangle, t_k, its three vertex indices may be out of the index range of the selected vertices. The algorithm described in Algorithm 3 is to reshape the t_k by replacing its vertex indices with the new ones from the range. The reformation task of a triangle is assigned to one GPU thread. The first step of the algorithm is to find which object the t_k belongs to, so that we can reform the t_k by

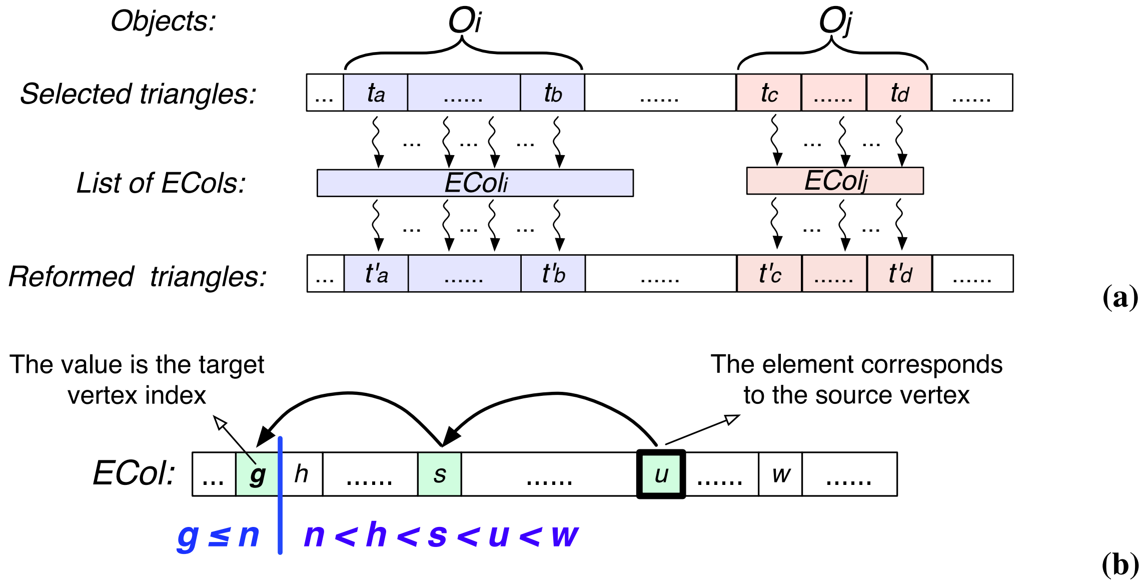


Figure 2.5: **An example of triangle reformation. (a)** shows the parallel reformation process. The selected triangles are organized in a block of continuous memory, then each triangle is reformed by finding and using its corresponding *ecol*. **(b)** shows how to replace a vertex with a target one by walking through array *ECol* backwards. n is the amount of selected vertices of an object. The target index is g, which satisfies $g \leq n$.

using the correct object's *ECol*. From Section 2.4.2, the final geometric complexities of objects are represented in the arrays of $\bar{vc}$ and $\bar{tc}$. I apply the prefix-sum algorithm over them using a parallel implementation with NVIDIA Thrust library. The prefix-summed $\bar{tc}$ indicates the offsets (or ranges) of triangles of the objects. Then, the index k is used to conduct a binary search in the $\bar{tc}$ to find its host object. For example, if k falls into a range $(\bar{tc}_i, \bar{tc}_{i+1}]$, the t_k belongs to the object O_i. This operation is at line 3 in Algorithm 3. To be consistent with $\bar{tc}$, $\bar{vc}$ is prefix-summed and the vertex count of the object O_i can be recovered by $\bar{vc}_{i+1} - \bar{vc}_i$.

The triangle is reformed to a new shape upon the LOD requirement. Each vertex index of the triangle t_k is replaced with the appropriate target one by walking backwards through the $ECol_i$. A vertex index may need to be updated several times until it reaches a value below the vertex count of

Algorithm 3 Triangle Reformation

Input: selected triangles, $\bar{tc}$, $\bar{vc}$, the list of *ECol*s
Output: reformed triangles

1: **for** kth triangle t_k **in** the selected triangles **in parallel do**
2: $i \leftarrow 0$; *// the object index that t_k belongs to*
3: **binary search** $\bar{tc}$ **return** i;
4: $n \leftarrow \bar{vc}_{i+1} - \bar{vc}_i$;
5: **for** $j = 1$ to 3 **do**
6: $vidx \leftarrow$ the jth vertex index of t_k;
7: **while** $vidx > n$ **do**
8: $vidx \leftarrow ecol_i(vidx)$;
9: **end while**
10: **end for**
11: **end for**

the host object. This process is indicated in line 4-8 of Algorithm 3 and illustrated in Figure 2.5 (b). Note that, in the algorithm, a vertex index *vidx* is the local index in the object O_i.

2.6 Evaluation

The triangle reformation algorithm reforms the shape of a triangle by replacing its vertices with the target vertices found in *ECol*s. ECol arrays are stored in the GPU global memory during entire rendering time. Since reading from the global memory is cached with a NVIDIA Fermi card or after, looking up an *ECol* to find the target vertex index is highly efficient. But when starting to write a reformed triangle to global memory, a writing operation will invalidate cache lines. The more times a GPU thread writes to global memory, the higher performance cost it makes to the algorithm implementation.

I compare my implementation of Triangle-Level Parallelism (*TLP*) to the Object-Level Paral-

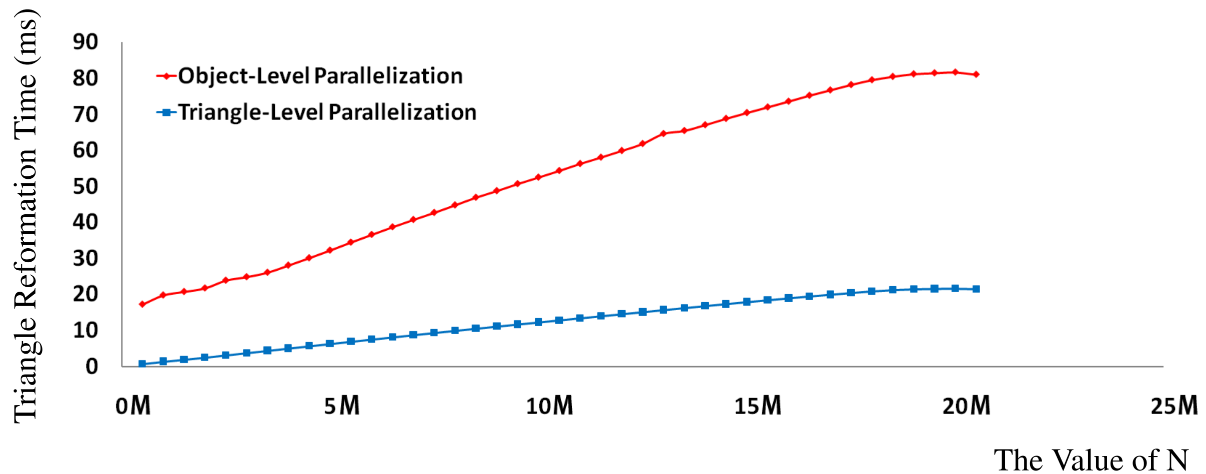


Figure 2.6: **The performance comparison of two types of parallelizations of the triangle reformation algorithm. Both implementations are executed on a NVIDIA GTX 680 device with 4GB global memory.**

lelism (*OLP*). For the implementation of *TLP*, one GPU thread reforms one triangle. It has one writing operation to GPU global memory. In contrast, the implementation of *OLP* asks one GPU thread to handle one object. The number of writing operations a thread has is determined by the number of triangles selected for the corresponding object.

Figure 2.6 shows the performance comparison of the two implementations. *TLP* is an average of 4.36X faster than the object-level parallelism. The performance of *TLP* linearly scales with the value of *N* introduced in Section 2.4.1, and thread workloads are balanced with 1-thread-per-triangle strategy. The performance of *OLP* is majorly affected by the thread associated with the largest number of triangles within a warp of GPU multiprocessor. Figure 2.6 tells us that multiple writing operations to the global memory by a thread results in a high cost to the performance.

Since the value of *N* is usually much larger the number of objects (several millions versus several hundreds of thousand), the implementation of *OLP* underutilizes the GPU processing resource. In

contrast, *TLP* have much more threads launched simultaneously. According to my evaluation, *TLP* allows more triangles to be reformed concurrently, which makes the execution time of triangle reformation algorithm significantly smaller than *OLP*.

Chapter 3

GPU Out-Of-Core Algorithm

Out-of-core algorithms are essential to organize the massive data too large to fit into the cached memory. Usually, massive models have to be stored in slow bulk memory. With a mesh simplification algorithm, a portion of data is view-dependently selected and fetched to the cached memory. Because of the high latency in I/O communications, transferring large amount of data from one memory environment to another is usually a performance bottleneck. I design a novel GPU out-of-core algorithm that efficiently transfers LOD selected data from the CPU main memory to the GPU memory.

In this chapter, I first review the related works in out-of-core data management and describe the research problems and goals. Then, I give the details of my GPU out-of-core algorithm.

3.1 Related Works

There has been lots of work on external-memory algorithms to support out-of-core rendering. Aliaga et. al [40] presented an interactive rendering system of complex models. The system employed prefetching and dat management schemes for models larger than available CPU main memory. Varadhan and Manocha [41] presented a prioritized prefetching approach for efficient out-of-core data management. Their approach considered both LOD- and visibility-based coherences between successive frames. Correa et. al [42] proposed a visibility-based prefetching algorithm to identify the geometry in the future frames, which supported interactive out-of-core rendering. They predicted a set of nodes that are likely seen next, and sent them to the memory ahead of time. When the camera views those nodes, they will be already cached in the memory so that the latency of fetching nodes from disk is reduced. Yoon et. al [43] presented an out-of-core algorithm to offline construct CHPM structure that includes data decomposition, bottom-up hierarchy generation and progressive simplification. Their approach took visibility events into account between successive frames by combining fetching and prefetching techniques. They added a frame latency at the time of fetching new visible clusters. Kontkanen et. al [44] described a coherent out-of-core approach for point-based global illumination by using properties of a space-filling Z-curve. Their out-of-core approach uses minimal I/O and stores data on the disk compactly and coherently for efficient access during shading. Cignoni et al. [45] and Gobbetti et. al [29] presented the out-of-core algorithms that preprocess input models and convert them into the hierarchical data representations. Active nodes are view-dependently selected by traversing the hierarchical representations in a top-down manner. They first check whether or not the selected nodes are already in

cached memory. If not, the nodes are then fetched from disks.

From the previous works, we can see that out-of-core algorithms were developed majorly based on the coherence between successive frames. When the camera changes from one position to another, the data contributed to the previously rendered frame can be reused for current frame rendering. Since those coherent data are already cached, we do not fetch them from disk so that the I/O latency is reduced.

3.2 Research Problems and Goals

With rapid hardware improvements, CPU main memory has been well developed to cache large datasets. It has become feasible to allocate tens of gigabytes of memory on RAMs in a PC workstation. However, the GPU does not catch up the same memory capacity of the CPU. Although the GPU allow us to perform most of computations, it cannot hold the entire dataset of massive models. Thus, the focus of out-of-core research is on how to improve the efficiency of transferring data from the CPU to the GPU.

CPU-to-GPU communication is supported through PCIe buses. A PCIe interconnects devices as a motherboard-level component and suffers a low bandwidth. The overhead of transferring data from CPU to GPU is a significant factor that impacts overall performance, especially when the size of to-be-transferred data is large. We know that mesh simplification algorithms can reduce the size of data required by renderers. As mentioned in Section 2.4, the size of LOD selected data can be determined based on the size of GPU memory. But the problem is that those selected data are

heavy workloads to the PCI Express, which makes brute-force data transfer very time-consuming. To solve this research problem, my goal is to design a GPU out-of-core algorithm by taking the advantages of frame-to-frame coherence.

As a result, I transfer only frame-different data and reuse the existing data from the previously rendered frames. The contributions of GPU out-of-core algorithm includes the two following features:

1. **Coherence-based Streaming.** I first identify the frame-different data, then collect them into a continuous memory block on CPU. This CPU memory block is then transferred to the GPU with a CPU-GPU memory copy call.

2. **GPU Memory Defragmentation.** For the sake of triangle reformation and OpenGL-based rendering, the polygon primitives on the GPU need to be concatenated into a single continuous memory block. However, the coherence-based data transferring does not preserve the data continuity. When new vertices and triangles arrive GPU memory, they are maintained at diffident locations from those for the existing data. Thus, I introduce a parallel defragmentation algorithm to reorganize the data in parallel with a GPU kernel.

3.3 Coherence-based Streaming

When users navigate a camera through 3D environments, there are always some data, which have contributed to render the previous frame, can be used for the current frame. Thus, the frame-

different data are the additional data required by the current rendering frame, which need to be fetched from the CPU data repository. Access to the CPU data repository is made by continuously selecting chunks of frame-different data. For the ith object in the model, the number of frame-different vertices $\Delta \bar{vc_i}^f = \bar{vc_i}^f - \bar{vc_i}^{f-1}$, f stands for the current frame, $f-1$ stands for the previous frame. If $\Delta \bar{vc_i}^f \leq 0$, no additional vertex is needed; otherwise, a subset of vertices, $\{v_{\bar{vc_i}^{f-1}}, v_{\bar{vc_i}^{f-1}+1}, v_{\bar{vc_i}^{f-1}+2}, \ldots, v_{\bar{vc_i}^f-1}\}$, are fetched from CPU to GPU. The same definition can be applied to the number of additional triangles, which is $\Delta \bar{tc}^f = \bar{tc_i}^f - \bar{tc_i}^{f-1}$.

Obviously, the additional data can be transferred sequentially at object level, which means that a CPU-GPU memory copy call is excited on CPU side, then it locates the target memory blocks on the GPU, and then it moves the data through PCIe. But launching multiple CPU-GPU memory copies would impose a significant performance cost. To avoid such cost, I design a more efficient way. First, I check the complexities of each object. Second,all additional data is collected into a block of continuous CPU memory. Third, I transfer this data block by one memory copy call. Collecting the additional data into a continuous block requires to check all object's geometric complexities one-by-one, which leads to an expensive cost, especially for the model containing a large number of objects. Hence, I parallelize the process of data collection in a multithreading manner, where the task is evenly distributed to the available CPU cores. A CPU core checks each object that it assigned with, and find out whether or not an object has an increased complexity. If the geometric complexity is increased from the previous frame, the increments are copied from the original data repository into the reserved continuous memory block. Here, to locate the target memory position quickly, I perform prefix-sum operations over the $\Delta \bar{vc}^f$ and $\Delta \bar{tc}^f$, so that the two

arrays give the position offsets for collecting the additional data, as illustrated in Figure 3.1.

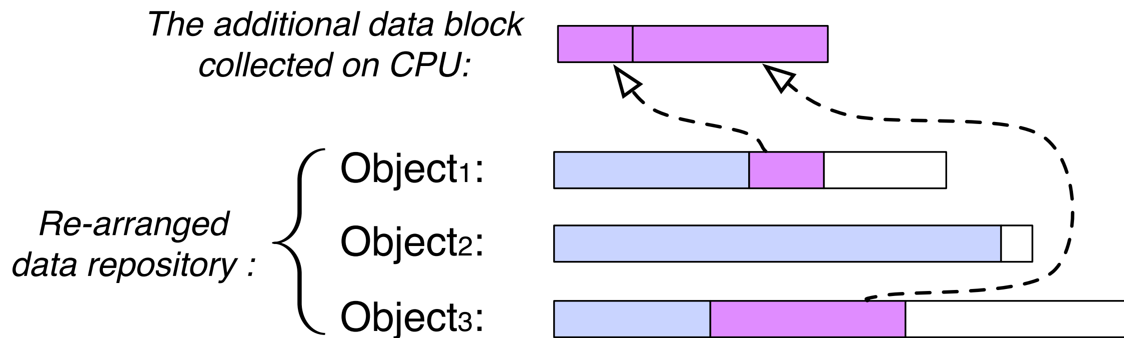


Figure 3.1: **An example of collecting the additional data on CPU.** The purple blocks replicated from the original objects represent the additional data required by GPU. The blue blocks are equivalent to the data already existing on GPU.

According to the data rearrangement scheme used in the preprocessing stage, the vertices and triangles of an object appear continuously in the data repository. As a result, only one CPU-CPU memory copy is required for each object when collecting them. After all CPU cores complete their tasks, the data block filled with the frame-different data is transferred to GPU with a CPU-GPU memory copy call.

3.4 GPU Memory Defragmentation

As we know, in graphics programming, the standard rasterization-based graphics pipeline supports high-performance rendering by taking the driver's hints about primitive usage patterns, for example, the OpenGL's non-immediate rendering method with Vertex Buffer Object (VBO). In order to enable OpenGL's VBO feature, the selected vertices and triangles have to be organized in the same

order as they are originally stored. However, with the frame-coherence-based streaming method, the additional data (new data required by current frame) and the existing data (inherited data from previous frame) are maintained in separate memory blocks on GPU. Thus, a reassembling procedure is desired so that the separate blocks can be combined into one. I call this procedure *GPU Defragmentation*, which reshuffles GPU data with respect to the storage appearance of vertices and triangles in original data repository.

Between two frames, the complexities of some objects have been increased, but others may have been decreased. If we fill the additional data into the blank memory blocks released by the objects whose complexities are decreased, the data would become fragmented because of the creation of many small and unsorted data chunks, and the result would not match the request of OpenGLs VBO.

The goal of the GPU Data Defragmentation is to make sure that,

1. The selected data for the frame is still continuously stored on the GPU.
2. Each object's vertices and triangles are organized in the order as it is rearranged in the preprocessing step.
3. The appearance of each object in the memory block is also stored in the same order as indicated in the arrays $\bar{vc}^f$ and $\bar{tc}^f$.

In the GPU memory, there are a block of existing data from the previous frame $f-1$ and a block of additional data required by the current frame f. To assemble them into the block reserved by the

frame-selected data, a straightforward method is using the system calls of GPU memory copy. For example, Nvidia CUDA SDK provides a memory management function named *cudaMemcpy()*. By specifying the copy direction to be *cudaMemcpyDeviceToDevice*, the data can be moved from one memory block to another. For each object, we could use this function to copy its vertices and triangles from both existing data block and additional data block into the selected data block at the position with the offsets $\bar{vc}_i^f$ and $\bar{tc}_i^f$, respectively. However, a massive model usually contains too many objects. Assembling the data for the objects one by one would require a large number of GPU memory copy calls. Those calls have to be executed sequentially by CPU and would lead to a significant performance cost.

Algorithm 4 Triangle Data Defragmentation on GPU

Input: array $\bar{tc}^f$, array $\bar{tc}^{f-1}$, array $\Delta\bar{tc}^f$, existing triangles, additional triangles
Output: the selected triangles

1: **for** kth triangle t_k **in** the selected triangles **in parallel do**
2: $i \leftarrow 0$;
3: **binary search** array $\bar{tc}^f$ **return** i;
4: $tidx \leftarrow k - \bar{tc}_i^f$;
5: $n \leftarrow \bar{tc}_{i+1}^f - \bar{tc}_i^{f-1}$;
6: **if** $tidx \leq n$ **then**
7: $j \leftarrow tidx + \bar{tc}_i^{f-1}$;
8: $t_k \leftarrow$ the jth existing triangle;
9: **else**
10: $j \leftarrow tidx - n + \Delta\bar{tc}_i^{f-1}$;
11: $t_k \leftarrow$ the jth additional triangle;
12: **end if**
13: **end for**
14: **replace** existing triangles **with** the selected triangles for the next frame;

An alternative way is operating each element in the memory block in parallel, whose efficiency, comparing to the above direct memory copy, has been demonstrated in many publications, such as [1, 35, 39], especially when the data size is large. Each GPU thread handles one target ele-

ment of the destination memory block, where the element will be filled with either an element of the existing data or an element of the additional data. I illustrate the defragmentation process in Figure 3.2. First, I identify the parent object that this element belongs to by binary-searching the prefix-summed complexity list (e.g., the array of vertex counts $\bar{vc}$). Second, I convert the index of the element into a local index by subtracting the offset; here, the offset is the sum of complexities of all frontal objects. Third, if the local index is smaller than the number of existing data, the target element is filled with the one from the existing data block; otherwise, it is filled with an additional one. With these three steps, I can construct the desired data block with the triangle-level parallelism. Algorithm 4 describes the defragmentation process for triangles. It reshuffles the triangles with a single kernel call rather than one call per object.

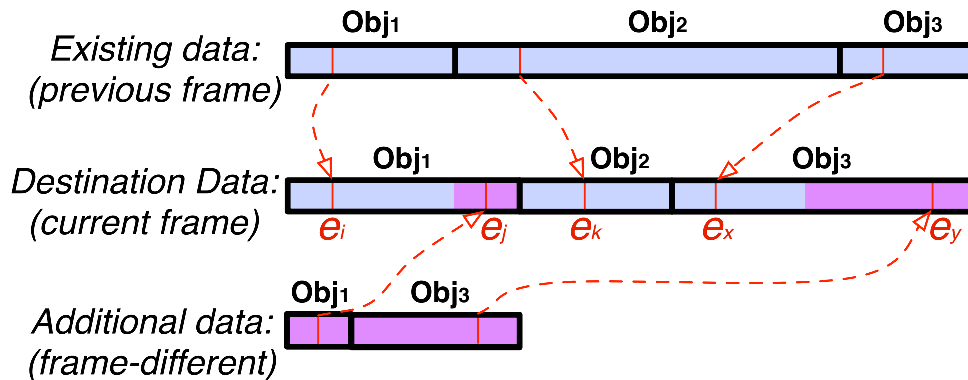


Figure 3.2: **An example of GPU defragmentation.** e_i, e_j, e_k, e_x and e_y are substituted with a source element from either existing data block or the additional data block in parallel.

In Algorithm 4, each GPU thread handles one triangle t_k of the selected triangles. t_k should come from either the existing triangles or the additional triangles, as illustrated in Figure 3.2. First, I identify the object index i that the triangle t_k belongs to because the source triangle to fill the posi-

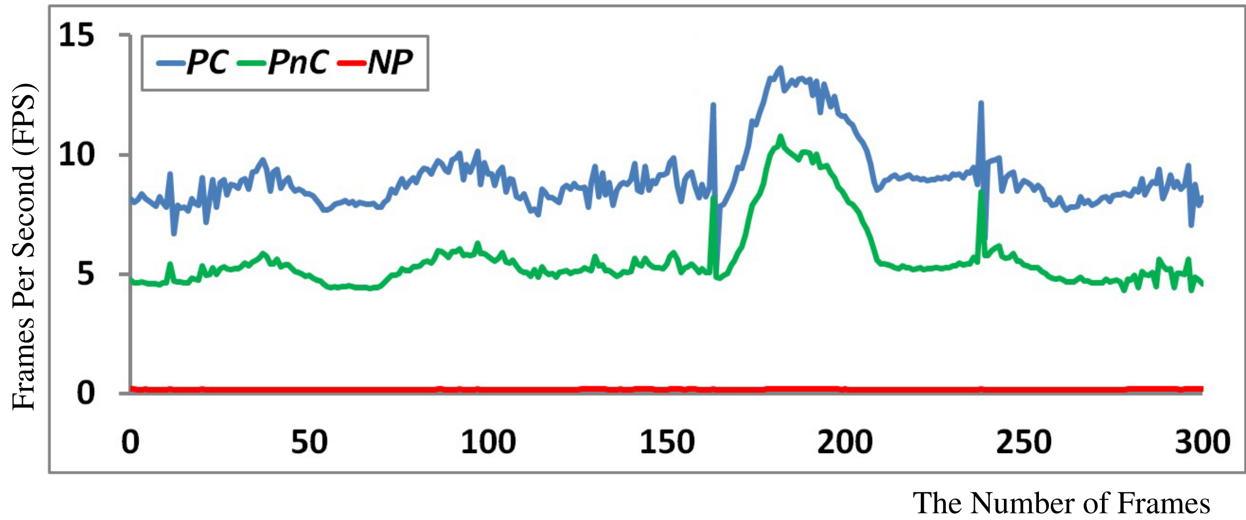


Figure 3.3: **The evaluation of the GPU Out-of-Core algorithm on a NVIDIA Quadro 5000 device.** The performance of coherence-based out-of-core algorithm (*PC*) is compared with the performance of the other two out-of-core approaches (*PnC* and *NP*) with the Boeing 777 model.

tion at t_k comes from the ith object. To find the object index quickly, a binary search is performed over the array $\bar{tc}^{f+1}$, which is the same way as done in Algorithm 3. Second, I identify if the source triangle for t_k should be an existing one or an additional one in the ith object. To do this, I convert the index of t_k in the the array of the selected triangles to a local triangle index in the ith object. This local index is denoted as $tidx$ (see line 4 of Algorithm 4). If $tidx$ is smaller than the number of existing triangles, the corresponding element from the array of the existing triangles is copied to the triangle t_k(see line 6-8); otherwise, the corresponding element in the array of the additional triangles is copied (see line 9-11). At the end, the existing triangles is replaced with the final LOD selected triangles, so that we can use the updated existing triangles to defragment the next frame.

Table 3.1: **The comparison of the three GPU-out-of approaches:** ***Packing with Coherence (PC)*** **(my approach),** ***Packing without Coherence (PnC)*****, and** ***No Packing (NP)*****.**

Approaches	**FPS**	**CPU-GPU Transferring**	**GPU Defrag.**	**Visible Triangles/Vertices**	**Transferred Triangles/Vertices**
PC	9.0	22.5 ms	31.1 ms	12.6M / 10.5M	0.5M / 0.4M
PnC	5.6	127.8 ms	—	12.6M / 10.5M	12.6M / 10.5M
NP	0.2	5825.7 ms	—	12.6M / 10.5M	12.6M / 10.5M

3.5 Evaluation

I use the Boeing 777 model to evaluate the GPU out-of-core algorithm. I compare my implementation, *Packing with Coherence (PC)*, with two other approaches: *Packing without Coherence (PnC)* and *No Packing (NP)*, which are two common brute-force strategies. Here, *Packing* means the data are collected into a continuous memory block before they are transferred. *Packing without Coherence* collects all of the LOD selected data to a continuous CPU memory block, then transfers the entire block to GPU with one memory copy call. *No Packing* approach sequentially copies all transfers vertices and triangles from CPU memory space to GPU one-by-one. Neither *PnC* nor *NP* approach needs the step of defragmentation. And *NP* approach even has no cost of preparing data on CPU. Figure 3.3 shows the performance comparisons of these three approaches with the Boeing 777 model. The coherence-based approach transfers only new-added vertices and triangles, and has a better performance than the other two approaches.

In average, the coherence-based out-of-core approach achieves about 1.66X speedup comparing to *PnC* approach, and achieves about 51.96X speedup comparing to *NP* approach. Table 3.1 shows the averaged times and data amounts of the comparison results. Note that the column of "Trans-

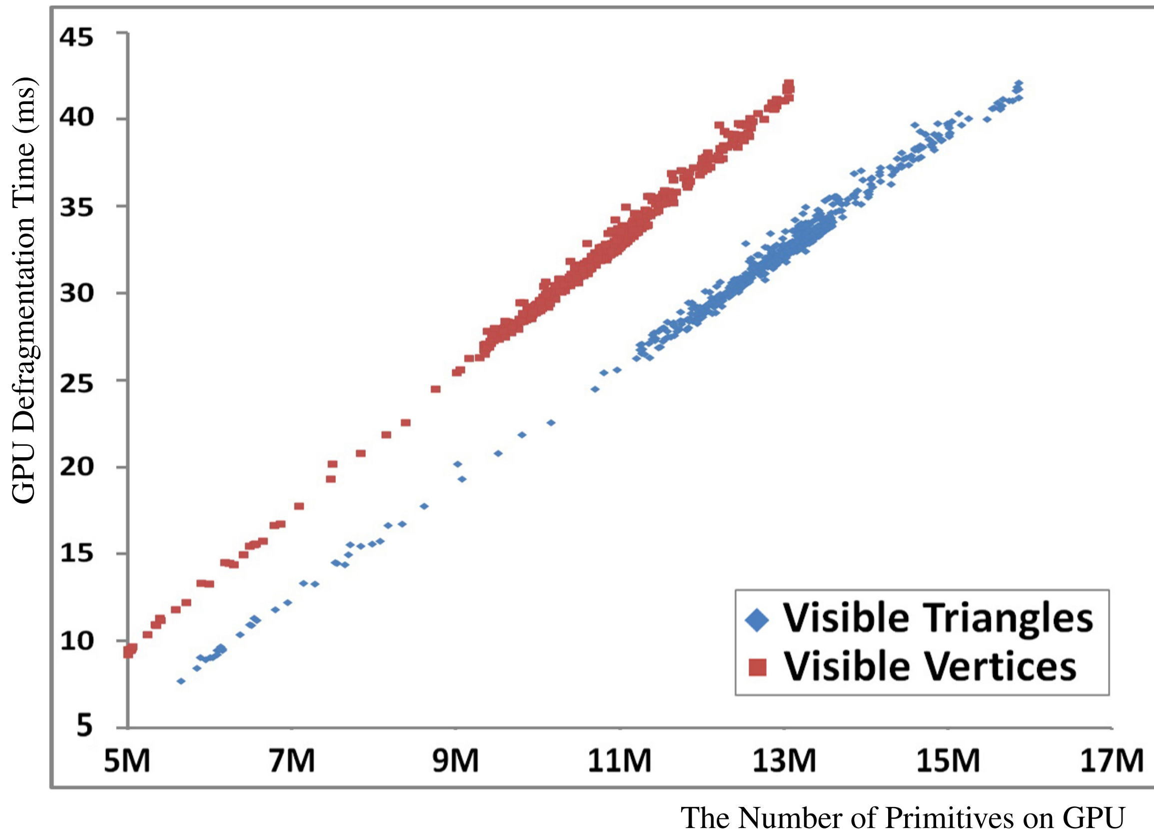


Figure 3.4: **The scattered value pairs of defragmentation time and visible triangle/vertex numbers on a NVIDIA Quadro 5000 device.**

ferred Triangles/Vertices" represents the amount of the data transferred from the CPU to the GPU. The *PC* approach requires much less amount in this column comparing to the other two approaches. It transfers only 4.0% of 12.6 million visible triangles and 3.8% of 10.5 million visible vertices.

The time on the column of "CPU-GPU Transferring" depends on the size of to-be-transferred data. With frame-to-frame coherence, its size become much smaller and is not an issue any more when passing through the PCIe bus.

The GPU defragmentation algorithm reorganizes the storages of the vertices and triangles on GPU. The more data is used in a frame, the more time is required to defragment them. The execution time

of GPU defragmentation algorithm scales with the number of visible triangles and vertices (the LOD selected data). Figure 3.4 plots the relation between the defragmentation time and the number of the visible data. It shows that defragmentation time behave linearly with the numbers of triangles and vertices.

Chapter 4

Massive Model Rendering System: Integrating with Occlusion Culling Algorithm

With simplification algorithms, we select a small portion of data to represent the original model as long as the selected data can fit into the GPU memory. However, for high-depth models, there may be many hidden objects blocked by others. They are invisible to the users, but they are processed by the GPU. Storing them on the GPU is a wast of resources. In order to further improve the memory usage, the hidden objects need to be excluded from the rendering pipeline.

In this chapter, I present a rendering system that integrates the simplification algorithm with a novel occlusion culling algorithm. I first review the related works in visibility culling. Then, I address

the research questions and goals. After giving the overview of the system, I concentrate on the details of the occlusion culling algorithm. Then, I give the details of the system evaluation.

4.1 Related Works in Visibility Culling

Computing the visibilities of 3D objects is one of the fundamentals in 3D computer graphics research. An interrelated survey [46] has reviewed many of the visibility culling approaches for walkthrough applications. Airey et. al [47] precomputed the set of potentially visible objects for a set of predefined viewpoints. Their method reduced the runtime computational complexity and produced significant speedups in virtual building simulations. Teller and Sequin [48] proposed an efficient preprocessing method for visualizing the axis-aligned architectural models.

Occlusion culling is a commonly used technique that rejects the hidden objects (blocked by other objects) before sending them to the GPU rendering pipeline. Greene et. al [49] used a hierarchical-Z buffer to help quickly rejecting the hidden objects in both object space and image space. Hudson et. al [50] used the shadow frusta to accelerate occlusion culling in object space, which was restricted to convex objects. Zhang et. al [51] used a bounding volume hierarchy in the object space and a hierarchy occlusion map in the image space for their culling algorithm. Klosowski et.al [52] presented a conservative culling algorithm based on the prioritized-layered projection (PLP) to determine the visibility status of the geometries by using OpenGL hardware extensions. Bittner et. al [53] optimized the usages of hardware occlusion queries for culling on arbitrary scenes. More recently, the hull-tree hierarchy [54] was used for occlusion culling with two phases of updating

the visible set.

To speed up the process of occlusion culling, researches have studied GPU-accelerated parallel approaches. In modern GPU devices, traditional Z-hierarchy method is accelerated with the fixed-functions [55], which are the parallel implementations. Govindaraju et. al [56] presented an approach using occlusion-switches on two GPUs. One GPU renders the depth information of occluders, and the other GPU performs culling operations using *GL_NV_occlusion_query_extension*. Two GPUs switch their roles after a frame is rendered. But this approach introduces the additional latencies of sending data between the GPUs. Other multi-GPU approaches, such as [57, 58], were proposed to speed up the culling performance, but the communication latencies and load-balancing issues are the major bottlenecks.

Integrating the approaches of mesh simplification and occlusion culling is necessary for rendering complex models. Aliaga et. al [40] presented a system for rendering complex 3D models. The model is partitioned into manageable cells, each of which contains the near geometries (after culling). Andujar et. al [59] provided the Visibility Octree to estimate the degree of visibility, which was also contributed to the LOD selection in the integration. Similarly, El-Sana et. al [60] used the View-Dependence Tree to integrate them in a simple and intuitive fashion. Yoon et. al [61] decomposed the 3D scene into a cluster hierarchy, where the simplification algorithm was applied to the set of visible clusters.

4.2 Research Questions and Goals

In Equation 2.3 of Section 2.4.1, *N* is an important factor impacting the overall performance and the visual quality. *N* determines how many vertices and triangles are processed by the GPU cores. A large value of N results in a heavy GPU workload. If we decrease the value of *N*, the performance is increased but we may result in the loss of visual quality. One way to preserve the visual quality with a small value of *N* is by adding visibility culling algorithms. At a specific viewpoint, many objects are invisible but they obtain the cuts from *N*. It would be better to give these cuts to the visible objects and increase their geometric details.

Hardware occlusion queries available in 3D APIs (e.g., OpenGL and Direct3D) were popularly used to check whether or not an object is visible. Unfortunately, this feature is not suitable for my application because of the following reasons: (1) the latency between issuing the query and the result availability may cause CPU stalls and GPU starvation; (2) flickering artifacts might occur because the previous frame depth buffer is used for occlusion testing in the current frame; (3) occlusion query technique is an independent and fixed function module, which is impossible to integrate with any LOD methods.

Therefore, I would like to present a parallel occlusion culling algorithm that seamlessly integrates with my parallel mesh simplification algorithm in a unified scheme on the GPU. It successfully removes hidden objects before the LOD selection algorithm allocates the primitive budget.

4.3 System Overview

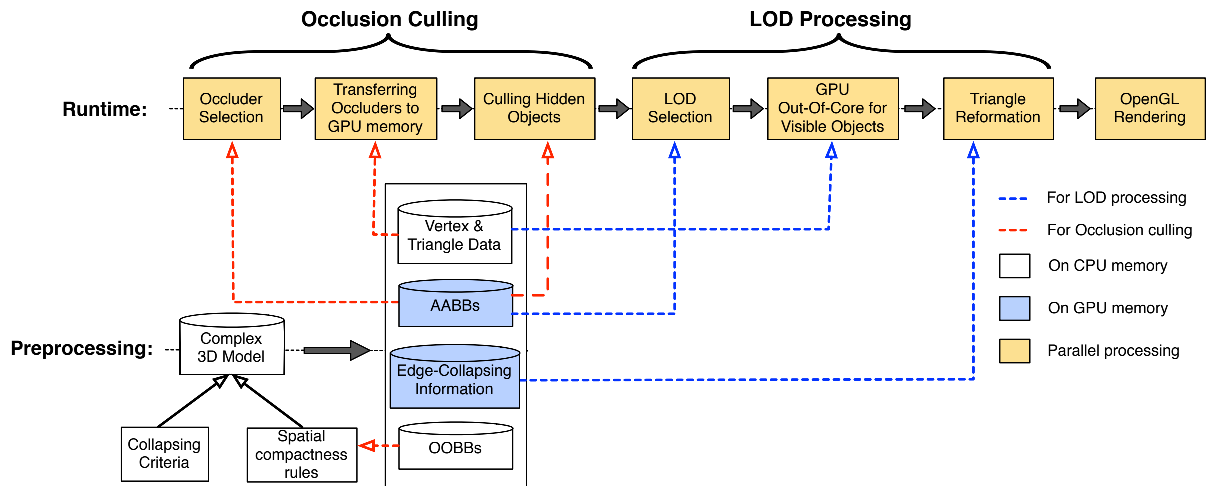


Figure 4.1: **Overview of the massive model rendering system.**

Figure 4.1 illustrates the overview of the rendering system. The input model is a complex multi-object model that consists of many disconnected objects that are unorganized, irregularly shaped and interweaving detailed in 3D space. The system conducts two computation stages, data preprocessing and runtime processing.

In preprocessing stage, the input model is converted into a new data representation based on the approach presented in Section 2.3. The generated edge-collapsing information is stored in GPU memory. In order to support occlusion culling at runtime, I exam the qualification of each object as an occluder by evaluating its spatial occupancy. I also generate the Axis-Aligned Bounding Box (AABB) and the Object Oriented Bounding Box (OOBB) for each object.

At runtime, there are two major computation components: occlusion culling and LOD processing. The total primitive budget allocated by the LOD processing component is applied to the visible

objects determined by the occlusion culling component. To exam the visibilities of the objects, a set of adaptive occluders is selected, transferred to the GPU, and rasterized into a Z-depth image. Then, if the depth of an object's AABB is deeper than the corresponding regions of the Z-depth image, it is blocked by the occluders and invisible to the camera. The remaining visible objects are passed into the LOD processing component. The LOD selection algorithm allocates the primitive budget to the objects. The GPU out-of-core algorithm transfers the selected data to the GPU. When the data is ready on the GPU, the triangle reformation algorithm generates the desired shapes for the objects. At the end, they are rasterized with OpenGL Vertex Buffer Objects (VBO).

4.4 Preprocessing for Occlusion Culling

In a model, not all objects are suitable to be occluders. Most existing systems use simple criteria to examine an objects qualification, such as the size of its AABB or the number of triangles it has. But some objects may have an irregular shape (e.g., casually curved long wires in the Boeing 777 airplane model), which should not be qualified as occluders at any viewpoint. Li et. al [62] presented a simple method that approximates an object's spatial occupancy to determine an object's qualification. In my approach, I employ a similar spatial occupancy criteria. Specifically, I calculate a tight Object-Orientated Bounding Box (OOBB) for an object, and measure its compactness in its object space. Equation 4.1 returns the compactness value indicating how well an object fills its OOBB's space.

$$compactness = \frac{A_x \frac{P_x}{R_x} + A_y \frac{P_y}{R_y} + A_z \frac{P_z}{R_z}}{A_x + A_y + A_z}. \tag{4.1}$$

In Equation 4.1, A_x, A_y and A_z represent the orthogonally projected area of an object along its OOBB's axises X, Y, Z, respectively; P_x, P_y and P_z are the numbers of pixels occupied by the object on the corresponding projection planes; R_x, R_y and R_z are the total numbers of pixels occupied by the OOBB on the corresponding projection planes.

The objects are then sorted based on their compactness values. The storage of objects is rearranged by moving the objects with higher compactness values to the front. I use a *Compactness Threshold* (CT) to find out a subset of objects as candidates for occluders. The *CT* defines the lower bound of compactnesses. The objects whose compactness values are above the value of *CT* will be added into the *Candidate Occluder Set* (COS).

4.5 Parallel Occluder Selection

During the runtime, a set of occluders from the COS is view-dependently selected. I denote this set as *Active Occluder Set* (AOS). Selecting the exact occluders is usually computationally intensive. I develop an efficient method which can sufficiently and effectively estimate the desired AOS. Three steps are performed to find out the AOS: (1) view-frustum culling: each object's AABB is tested against the view frustum so that the object whose AABB is outside the view-frustum is considered to be invisible; (2) weighting candidate occluders: the objects in the COS are weighted based on

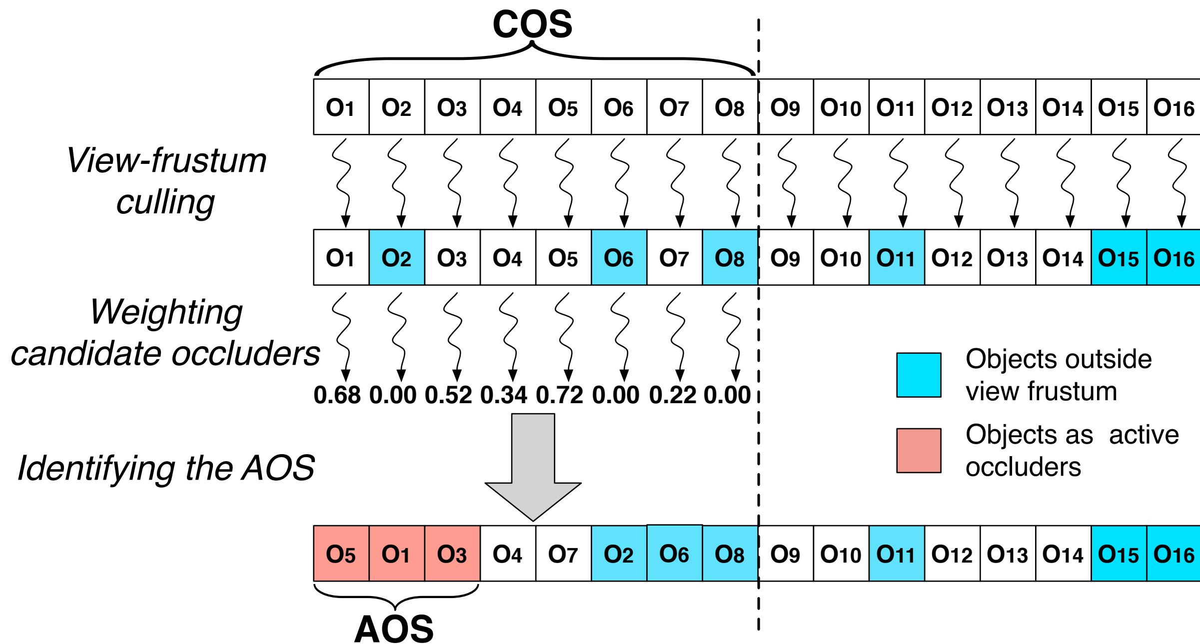


Figure 4.2: **An example of the parallel occluder selection.** Assuming that the model consists of 18 objects. 8 of them are classified into the COS. The size of AOS is fixed to be 3 objects.

the setting of a specific viewpoint. (3) identifying the AOS: I identify the objects with higher weights and put them into the AOS. An example of the three steps is illustrated in Figure 4.2.

4.5.1 View-Frustum Culling

In a standard rendering pipeline, an object is converted into pixels and displayed on the screen only if it is inside the view frustum, at least partially. The view frustum is the pyramid-shaped volume with the top chopped off. The exact shape of the volume varies depending on how the camera lens is specified. The position of the camera is at the apex of the pyramid. The near plane and far plane truncate the pyramid at the top and bottom respectively to restrict the viewing depth of the camera, and they are perpendicular to the viewing direction of the camera.

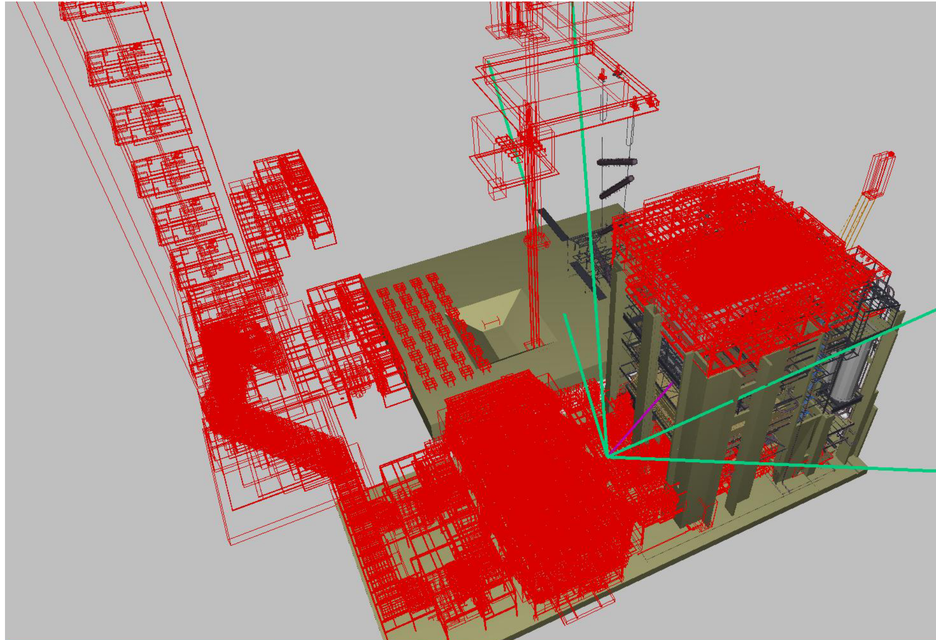


Figure 4.3: **An example of view-frustum culling.** The green lines define the volume of the view frustum. The red bounding boxes stand for the objects outside the view frustum.

To remove the objects outside the frustum from the rendering pipeline, I use their AABBs to test against the volume of the view frustum. The obvious method to do this would loop through all the objects and check the intersection between the AABBs and the frustum volume one-by-one. This sequential method would be slow because of the large number of objects present in the model, I deliver a parallel implementation on the GPU. The AABBs are stored on the GPU, and each GPU thread handles one object. If any of the eight points of an AABB is inside the frustum volume, the corresponding object is considered as an inside object. As a result, the objects outside the frustum will not be rendered, and of course they lose the opportunity to be in the AOS though they may have high compactness values. The reason of using AABBs instead of OOBBs is because an AABB supports a faster execution and has less memory requirements on the GPU. Figure 4.3 shows an example of view-frustum culling with the Power Plant model.

4.5.2 Weighting Candidate Occluders

The objects of the COS inside the view frustum still hold their candidate status as active occluders. But we certainly do not want to select all of them to the AOS, since the number of them is large and many of them may actually be occluded. For example, in the Boeing 777 model, the COS contains 362 thousand objects with the value of CT equal to 0.63 (50.4% out of the total of 718 thousand objects). Hence, we need to use less but optimal candidates.

My occluder selection criteria is to choose the objects that are spatially large and close to the viewpoint. I develop a weighting method to determine the AOS. I compute a direction vector to each object. The vector is perpendicular to the largest surface of the objects OOBB. Then, the weight of the object is calculated based on its direction vector, its distance to the viewpoint, the size of its bounding volume and the viewing vector of the camera, as showing in Equation 4.2.

$$weight_i = \frac{V_i||\overrightarrow{R_i} \cdot \overrightarrow{E}||}{D_i^3} \tag{4.2}$$

V_i is the size of the ith object's AABB; $\overrightarrow{R_i}$ is the direction vector of the object; E is the viewing vector of the camera; D is the closest distance between the AABB and the viewpoint position. In general, an occluder will give the best occlusion result when its direction vector R faces to the viewpoint. In Equation 4.2, we use the dot product to capture this property. As shown in the second step of Figure 4.2, weighting the objects are executed with an object-level parallelism. If a candidate object is inside the view frustum, the value of weight is computed using Equation 4.2; otherwise, the value is assigned to be zero and lose the opportunity to be in the AOS. Note that I

always use AABBs since they requires less memory of storing the corner points than the OOBBs.

4.5.3 Identifying Active Occluder Set (AOS)

In order to quickly identify the AOS, I sort the COS based on the descending order of the weights of objects. The higher weight the object has, the better this object is added into the AOS. I constrain the AOS with a fixed size, so that the number of active occluders will not be changed during the runtime. Although it might be reasonable to use an alternative way to decide the number of active occluders (e.g., setting the lower bound of weight), the size of AOS could vary with the changes of the camera, which would be not optimal to the memory-bound applications. I consider that the objects in the AOS are significantly visible. The full geometric details of these objects are used for both culling and rendering. If the size of AOS is too large, the active occluders would overburden the GPU memory capability and the rendering workloads. For instance, the experimental results presented by Aila and Miettinen [63] show that selecting 30% of the visible objects from a model consisted of only hundreds of millions of polygon primitives will result in up to many hundreds of megabytes of storage. Hence, I use a small size of the AOS as long as it is sufficiently effective for occlusion culling. Because the COS is sorted, I can identify the AOS at a constant speed by selecting the subset of the COS starting from the beginning element.

4.6 Conservative Culling with Hierarchical-Z Map

The goal of occlusion culling is to determine a set of objects that are potentially visible, which is commonly known as *Potentially Visible Set* (PVS). After the step of occluder selection (Section 4.5), the objects in the AOS definitely belong to the PVS. For all other objects inside the view frustum, I determine whether or not they belong to PVS by testing whether or not they are occluded by the AOS. To do this, I first build the Hierarchical-Z Map (HZM) using the active occluders. The active occluders are rendered to the depth image in the framebuffer.

Algorithm 5 Building HZM

Input: depthImg, imgDim
Output: HZM

```
1:  for ith level of HZM do
2:      size ← imgDim × imgDim;
3:      if i = 0 then
4:          HZM_i ← depthImg;
5:      else
6:          for HZM_i.pixel_(x,y) in size in parallel do
7:              n0 ← HZM_{i−1}.pixel_(2x,2y);
8:              n1 ← HZM_{i−1}.pixel_(2x+1,2y);
9:              n2 ← HZM_{i−1}.pixel_(2x,2y+1);
10:             n3 ← HZM_{i−1}.pixel_(2x+1,2y+1);
11:             HZM_i.pixel_(x,y) ← Max(n0, n1, n2, n3);
12:         end for
13:     end if
14:     ImgDim ← ImgDim/2;
15: end for
```

Similar to the approaches described in [49, 51], the HZM is constructed by recursively down-sampling the fine-grained depth image in an octree manner. The depth values can read from the texture memory. Each GPU thread computes a new depth value by reading and operating the four adjacent pixels of the upper level of HZM. Each level of HZM is an intermediate depth image,

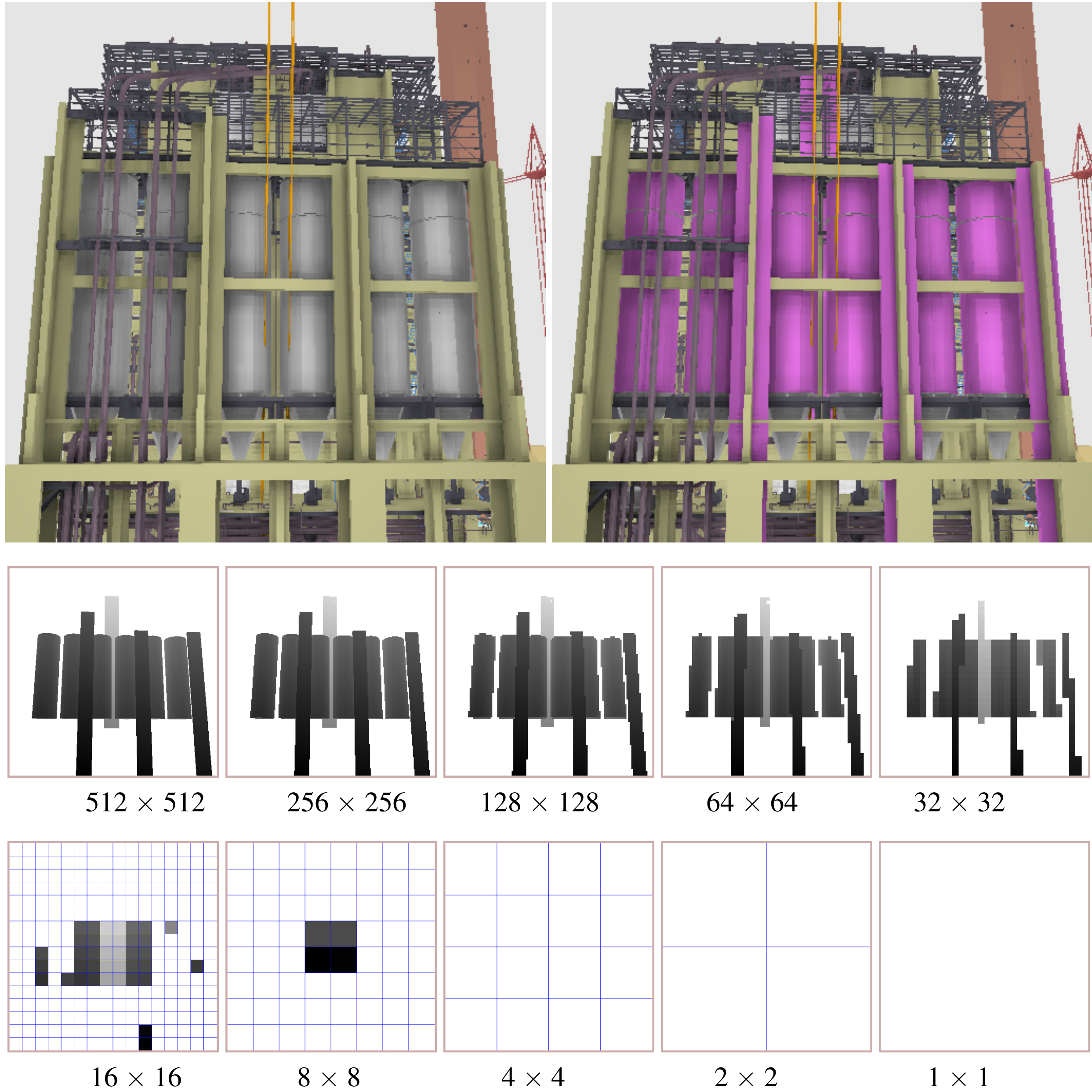


Figure 4.4: **An example of HZM.** The original depth image is rendered in the dimension of 512 × 512. The purple objects are active occluders. The size of AOS is set to 10, and all the ten levels of the HZM are shown. Each blue cell represents a pixel at the corresponding level.

which is down-sampled by merging 2×2 blocks of pixels from the parent image (one level finer). If, for example, the dimension of the finest level is 512×512, the number of levels of the HZM is $log_2 512 + 1 = 10$. The finest level is identical to the original image from the framebuffer; and the coarsest level contains only one pixel. Algorithm 5 shows the pixel-level parallel algorithm to build the HZM on GPU. The *imgDim* represents the dimension of the finest level. So, the total number of levels is $log_2 imgDim + 1$. After finishing the ith down-sampling step, the image for the $level_i$ is created, and the value of *imgDim* is reduced by half for the next step. At each step, a pixel is equal to the one whose depth value is the largest among the four adjacent pixels from the finer level. I show an example of HZM in Figure 4.4. Since the down-sampling operations are managed on the GPU, there are no interruption/delay caused by CPU controls. The HZM is stored in a continuous block of memory allocated at the initialization. The reason I can allocate the memory at the initialization is because the size of HZM is known from Equation 4.3. In Algorithm 5, the pointer to the ith level of the HZM can be identified with an appropriate offset. The offset is a certain amount of pixels that we should jump over from the beginning of the HZM. This offset is computed by Equation 4.3, where i is the index of the level.

$$Pixel\ Count = \frac{1-(1/4)^i}{3/4} \times imgDim^2 \tag{4.3}$$

I determine the visibility of an object by testing its depth against the corresponding pixels at an appropriate level of HZM. The level is selected based on the projected area of the object on the screen. I calculate the projected area by using the object's AABB. Since the shape of projected

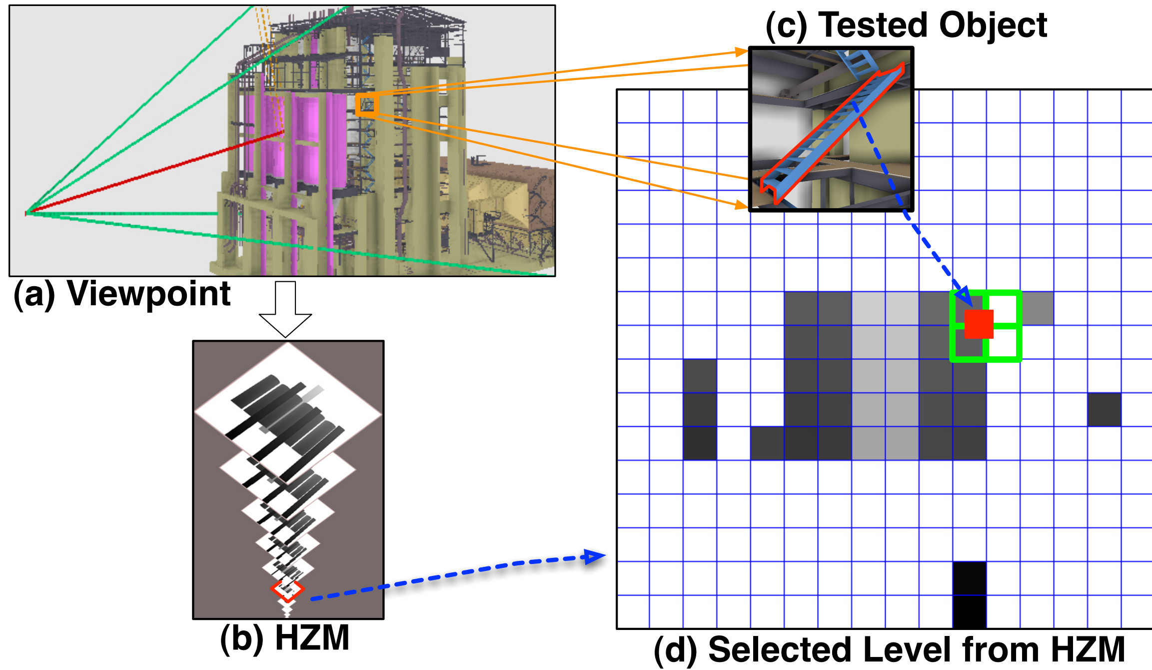


Figure 4.5: **The concept of culling with the HZM. (a)** shows the the configuration of the viewpoint and the active occluders. The green lines define the view frustum, and the redline defines the viewing vector from the camera position. The purple objects are the active occluders; **(b)** shows all levels of the HZM constructed from the active occluders. The dimension of the finest level is 512×512; **(c)** shows the object whose visibility is determined by testing its depth value against an appropriately selected level of HZM; **(d)** represents the selected level of HZM. The red square represents the projected size of the object on the screen, and it overlaps with the four green blocks associating with the depth values from the active occluders.

region is a hexagon, it would be costly to compute the exact area. Also, the shape of hexagon is not suitable for comparing to the squarish pixels. Thus, I use a bounding square of the hexagon as an approximation, where the dimension of the square is equal to the longer edge of the bounding rectangle of the projected AABB.

According to the size of bounding square, I select a level of the HZM for testing the objet's visibility. Let us denote the dimension of the square as *sqDim* (in pixel) and the total number of levels of HZM as *B*, the appropriate level can be calculated from $B - log_2 \frac{imgDim}{sqDim}, (sqDim \geq 1)$, where

imgDim represents the dimension of the finest depth image. If $sqDim = 0$, the object is considered to be invisible since its coverage is less than one pixel on the screen. By doing this, the area covered by the square is guaranteed to overlap with at most four pixels at the chosen level (see Figure 4.5 (d)).

The position of the square is generated after the projection transformation. By matching the position of the square to the grid of the pixels, I identify the indices of the four pixels overlapping with the region of the square. The depth of the square is assigned to be the depth of the AABB's conner point that is the closest to the camera. If the square depth is deeper than all of the four overlapped pixels, the object is surely occluded; otherwise, the object is labeled as a potentially visible object and put it into the PVS. I parallelize this culling process at an object-level on the GPU. Regardless of sizes of the projected objects, the time of culling each is constant time.

With this occlusion culling approach, the visibilities of the objects are the conservative estimations rather than the exact results. This is because the bounding square occupies a larger screen region than the corresponding object actually does. As a result, an actually occluded object could be labeled to be visible if it cannot stand behind with a deeper depth. But, this conservative method does not cull away any actually visible objects, so that it does not affect the correctness of rendering. After executing the occlusion culling component, I apply the LOD selection algorithm to the objects of the PVS.

4.7 Evaluation

In this section, I evaluate the system by providing the detail analysis of each processing components. I describe my implementation configurations in Section 4.7.1. The evaluation of preprocessing component is presented in Section 4.7.2. The runtime performance is discussed in Section 4.7.3. Section 4.7.4 and 4.7.5 analyze the components of occlusion culling and LOD processing respectively. Section 4.7.5 also gives performance breakdowns of the LOD processing component.

4.7.1 Implementation

The proposed rendering system has been implemented on a 64-bit Windows system using C++, OpenGL and NVIDIA CUDA 4.2 Toolkits. I use two multi-object 3D models. Both of them are exceptionally complex and consist of many loosely connected, badly tessellated, intertwining detailed objects with widely varying spatial ratios and complex topologies. I am confident that, as shown in Figure 4.6, the two models sufficiently represent the benchmarks of my target application domains: the Boeing 777 airplane model has 718 thousand objects containing more than 332 million triangles and more than 223 million vertices. The Power Plant model has about 151 thousand objects containing about 12.7 million triangles and about 6.1 million vertices. I maintain all AABBs and edge-collapsing information (*ECol*s) on the GPU memory permanently during the runtime.

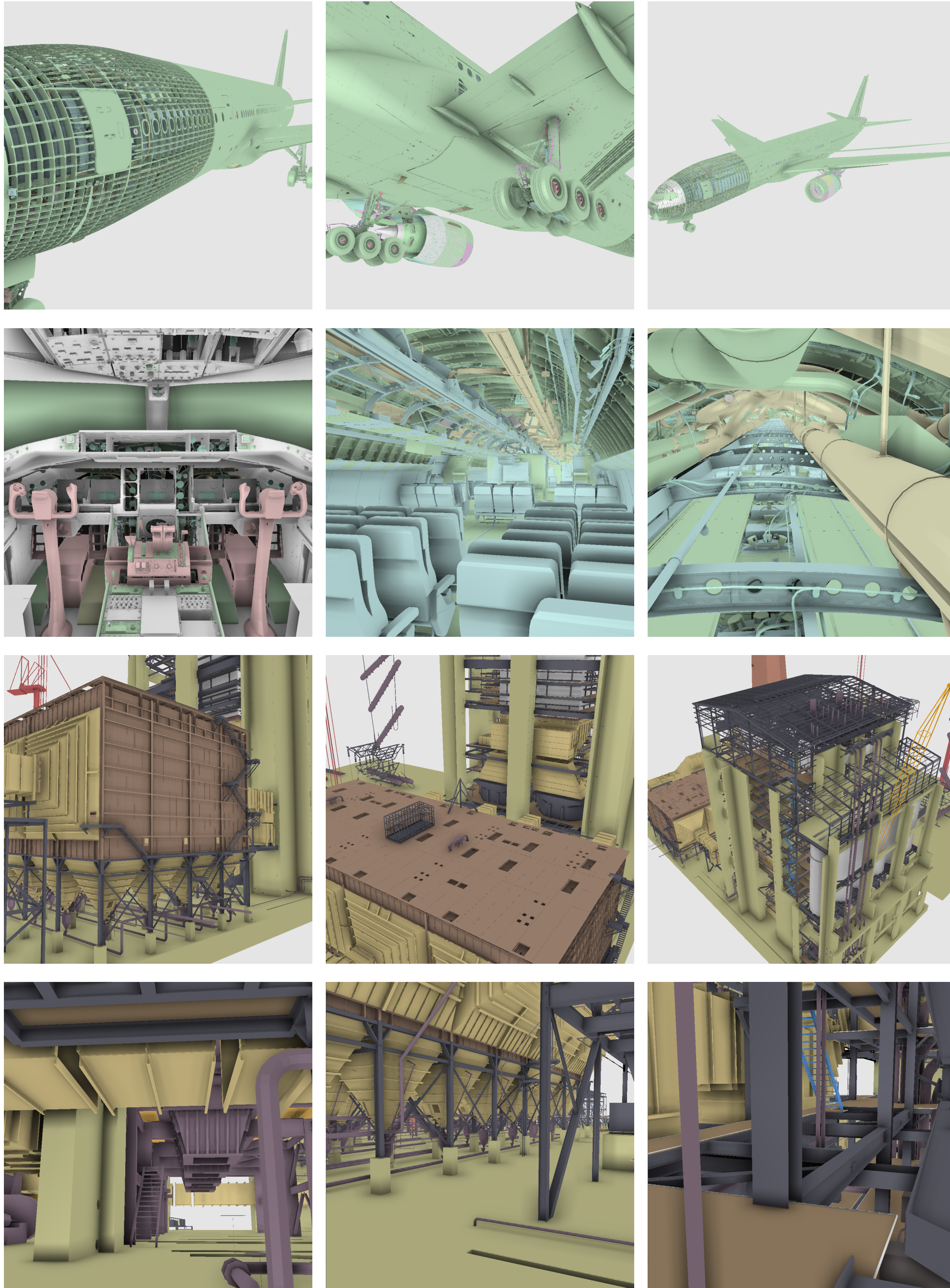


Figure 4.6: **The models rendered by the system.** The first and second rows are the rendered images of the Boeing 777 airplane model. The third and forth rows are the rendered images of the Power Plant model. I apply the Screen Space Ambient Occlusion algorithm (SSAO) [2] to improve shading quality. The dimension of these rendered images is 1024×1024.

Table 4.1: **The preprocessing results on a single PC.**

Models	**Data File**		**Edge-Collapsing**		**Spatial Compactness**	
	Tri/Vert Num.	**Size**	***ECol* Size**	**Time**	**Size**	**Time**
Boeing 777	332M / 223M	6.1GB	582.5MB	952min	2.9MB	38min
Power Plant	12M / 6M	0.5GB	14.2 MB	40min	0.6MB	5min

4.7.2 Evaluation of Preprocessing Component

I preprocessed the input models with a workstation equipped with an Intel(R) Core(TM) i7 2.67GHz CPU (4 cores) and 12 Gigabytes of RAM. The preprocessing stage includes two parts: recording the collapsing information and computing the objects' spatial compactnesses. The preprocessing stage is done on the CPU. The performance results are shown in Table 4.1.

On average, the throughput performance of the preprocessing method is 5K triangles/sec. Comparing to the state-of-art approaches, Yonn et. al [21] generated the CHPM structure at 3K triangles/sec; Cignoni et. al [45] constructed tetrahedron hierarchies at 30k triangles/sec on a network of 16 CPUs; and Gobbetti and Marton [29] built far voxel hierarchies at 1K triangles/sec on a single CPU and 20k triangles/sec on 16 CPUs. My preprocessing method is at least 66.7% faster than the previous single-CPU solutions.

In terms of memory complexity, Table 4.1 shows that the extra memory required by the columns of "Edge-Collapsing" and "Spatial Compactness" is approximately 0.6GB for the Boeing 777 model. As a result, its total memory requirement is 6.7GB. In contrast, for a 6GB-sized input model, Cignoni et. al [45] required 17.1GB memory to represent the model in tetrahedron hierarchy, and Gobbetti and Marton [29] required 14.9GB memory to represent the model in far voxel hierarchies.

Table 4.2: **The runtime parameter configurations.**

Models	α (Section 2.4.1)	*PET* (Section 2.4.1)	*MinT* (Section 2.4.2)	*CT* (Section 4.4)	AOS Size (Section 4.5)
Boeing 777	3.0	1 pixel	0.65	0.55	20
Power Plant	3.0	1 pixel	0.88	0.77	15

Table 4.3: **Overall performance on a NVIDIA GTX 580 device.** The results are averaged over all the frames along the created camera paths.

Models	**N**	**Visible Triangles**	**FPS**	**Occlusion Culling**	**LOD Processing**	**OpenGL Rendering**
Boeing 777	10.5M	15.2M	14.5	7.9 ms (11.4%)	52.6 ms (76.2%)	8.5 ms (12.3%)
Power Plant	3.5M	6.1M	67.6	4.8 ms (32.4%)	4.9 ms (33.1%)	5.1 ms (34.5%)

4.7.3 Runtime Performance

I evaluate the runtime performance on a workstation equipped with an Intel(R) Core(TM) i7 2.67GHz CPU (4 cores), 12 Gigabytes of RAM, PCIe 2.0 $\times 16$ and four GPU devices. I created the camera navigation paths for each model for real-time investigations. Table 4.2 lists the parameter configurations used for the two tested 3D models.

Table 4.3 shows the execution time of the three major runtime components. *N* is a user-specified parameter to define the total vertex budget (see Equation 2.3). Since the Boeing 777 model cannot be stored on the GPU, the GPU out-of-core algorithm has been applied to both components of occlusion culling and LOD processing. The size of the Power Planet model is small, and it can be stored on the GPU, so that it does not need the out-of-core algorithm. According to the averaged performance for the Boeing model listed in Table 4.3, the system throughput is approximately

Table 4.4: **The effectiveness of Occlusion Culling (OC) with the Boeing 777 model.**

Total Num. of Objects	Objects Culled by OC		OC Accuracy (My Approach)	Released Memory
	My Approach	Exact		
718K	63K (8.8%)	108K (15.0%)	58.3%	348.6MB

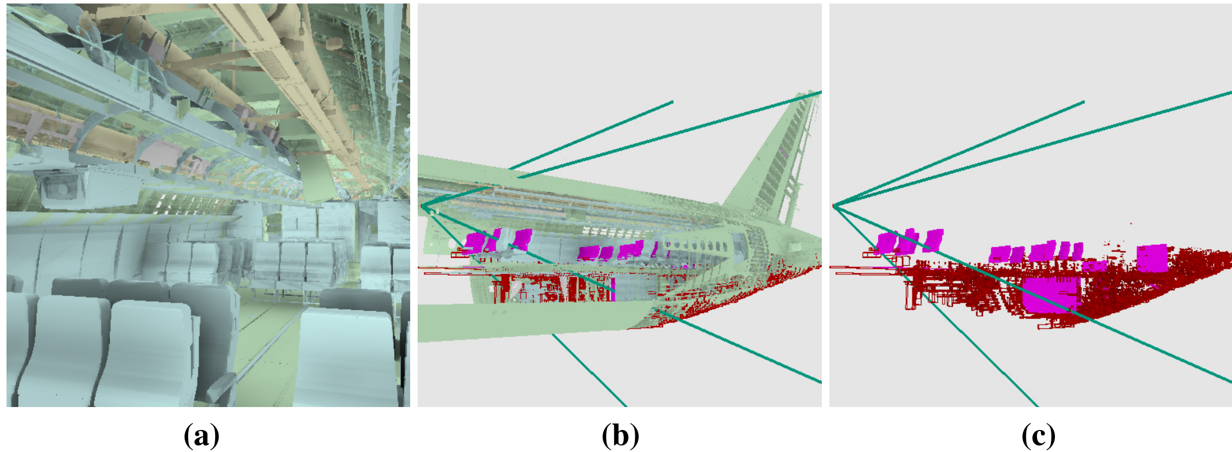
220 million triangles/second. Comparing to the state-of-art massive model rendering systems, the TetraPuzzles approach [45] achieved 70 million triangles/second, and the Far Voxel approach [29] achieved 45 million voxels/second.

In occlusion culling component, the size of the AOS is fixed. Transferring the active occluders is not expensive. Constructing HZM on the GPU is also very efficient. the HZM construction time is determined by the resolution of display image, which is constant as long as the image resolution does not change. The LOD processing component plays a dominant role in performance for the Boeing 777 model. This is because transferring 15.2 million triangles is a heavy workload for the out-of-core and triangle reformation algorithms. Rendering the model with OpenGL is fast, even with more than 10 million triangles and vertices.

In the following subsections, I would like to provide more detailed analysis for the components of occlusion culling and LOD processing.

4.7.4 Evaluation of Occlusion Culling Component

The goal of occlusion culling component is to release the memory spaces occupied by the hidden objects, so that they can be used to increase the geometric complexity of the visible objects. In



(a) (b) (c)

Figure 4.7: **The occlusion culling result with the Boeing 777 model.** **(a)** is the rendered frame. **(b)** is the reference view, where the dark green lines define the volume of the view frustum. The purple objects are the active occluders. The red boxes represent the occluded objects. **(c)** shows only the active occluders and the occluded objects.

order to release the memory, a certain performance penalty occurs because of the procedures of occluder selection, HZM construction and conservative culling. But, according to the facts in Section 4.7.3, the occlusion culling component is not the major overhead of the system.

Here, my evaluation focuses on the effectiveness of releasing memory from the hidden objects. Table 4.4 shows the comparison results between my approach and the exact occlusion culling approach. The exact approach first rendered the entire model into a depth buffer, then I culled away the hidden objects by testing the depth values of their AABBs against the buffer. This exact approach makes the overall performance less than 1 frame per second.

Table 4.4 shows that my occlusion culling algorithm cannot deliver identical results as the exact occlusion culling approach. It is possible that my algorithm treats an actually occluded object as a potential visible one due to the criteria used for the occluder selection and the heuristic way of conservative culling with the HZM. But it still achieves a fairly high effectiveness with little

sacrifice in performance. Figure 4.7 demonstrates the results of occlusion culling algorithm for the Boeing model. In general, the culling results vary depending on the behaviors of the camera. When the camera faces the environment having more objects along the depth direction, the culling results are much closer the exact results.

4.7.5 Evaluation of LOD Processing Component

The LOD processing component is important to the proposed rendering system since it operates the majority of data repository on the GPU. It is the core for achieving a good balance between the performance and rendering quality. As described in Section 4.7.4, the occlusion culling component operates objects, and its performance scales with the number of objects of the original model. The out-of-core and reformation algorithms in the LOD processing component and the OpenGL rendering execution are all bound to the number of polygon primitives. Thus, It is safe to disable the functionality of occlusion culling when evaluating the LOD processing component.

Performance Model

The performance model of the LOD processing component is essential for understanding the performance factors of the system and identifying intuitive control parameters to satisfy different rendering constraints/goals. Here, I formulate a comprehensive model to describe the time of different execution steps. I introduce several user-tunable control parameters to allow trade-off between different rendering constraints.

As illustrated in Figure 4.1, when the occlusion culling component is disabled, the runtime of the system includes the four execution steps, LOD Selection, GPU Out-of-Core, Triangle Reformation and OpenGL Rendering. Therefore the total execution time, T, of rendering a particular image frame can be formulated as follows:

$$T = T_{lod}(M) + T_{ofc}(N) + T_{tr}(N) + T_r(N) \tag{4.4}$$

where, T_{lod}, T_{ofc}, T_{tr} and T_r are the execution time for LOD Selection, GPU Out-of-Core, Triangle Reformation and OpenGL Rendering respectively; M is the total number of objects of the original model; N is the number of primitives selected for rendering the frame.

According to the details described in the previous chapters, I refine the Equation 4.4 by providing the time complexity analysis for each execution step as follows:

$$T_{lod} = w_{lod} \times M \tag{4.5}$$

$$T_{ofc} = w_{ofc} \times N \tag{4.6}$$

$$T_{tr} = w_{tr} \times N \tag{4.7}$$

$$T_r = w_r \times N \tag{4.8}$$

where w_{lod}, w_{ofc}, w_{tr} and w_r are the performance factors determined by the computational power of the hardware and the algorithmic settings. Therefore, the Equation 4.4 can be rewritten as

follows:

$$T = w_{lod} \times M + (w_{ofc} + w_{tr} + w_r) \times N \tag{4.9}$$

I also define three commonly used goals for the rendering system: frame rate (a.k.a FPS), rendering quality and system throughput. Here are the quantitive definitions of these goals:

1. **Frame rate (FPS).** The frame rate is an important rendering goal for many performance sensitive visualization applications, such as smooth animation. In these applications, users often requires the frame rate beyond certain threshold.

$$FPS = \frac{1}{T} = \frac{1}{w_{lod} \times M + (w_{ofc} + w_{tr} + w_r) \times N} \tag{4.10}$$

2. **Rendering Quality.** This goal is essential for the applications needs certain level of image quality. In our case, it is defined as the mesh simplification ratio $SR = N/$(total number of primitives).

3. **System Throughput.** The throughput of a system indicates the efficiency and effectiveness of the algorithms. It is usually defined as the final amount of effective output in a second. In my case, I define the System Throughput (TP) as the final rendered number of triangles in a second.

$$TP = N \times FPS = \frac{N}{w_{lod} \times M + (w_{ofc} + w_{tr} + w_r) \times N} \tag{4.11}$$

Among all the parameters defined in the three rendering goals, the intuitive one that can be easily tuned is the number of primitives selected for rendering a frame, N, which is also constrained by hardware limitations, such as GPU memory size. There are two scenarios for determining a desired value of N:

1. **GPU memory capability.** Since the size of an input massive model is usually far beyond GPU memory capability, the entire dataset may not be maintained on GPU. In order to fully utilize the GPU memory, we can set the value of N based on the size of GPU memory.

2. **The tunable N satisfying the rendering constraints.** In the LOD selection approach presented in Section 2.4, the value of N scales linearly with the execution time of the GPU Out-of-Core, Triangle Reformation and OpenGL Rendering. Thus, N can be used to tune the overall system performance, such as the frame rate and throughput.

Performance Breakdowns

With the LOD selection algorithm described in Section 2.4, I can precisely specify the desired geometric complexity of the model by giving the value of N in Equation 2.3. According to the performance model presented in Section 4.7.5, the overall performance should behave linearly to the values of N. In Figure 4.8, I plot the relation between the overall performance (total execution time and FPS) and the values of N. If the renderer favors interactivity, the N can be assigned with a smaller value, so that the GPU and the PCI Express bus handle less vertices and triangles; otherwise, a larger value of N gives a finer data representation that ensures the accuracy by sacrificing

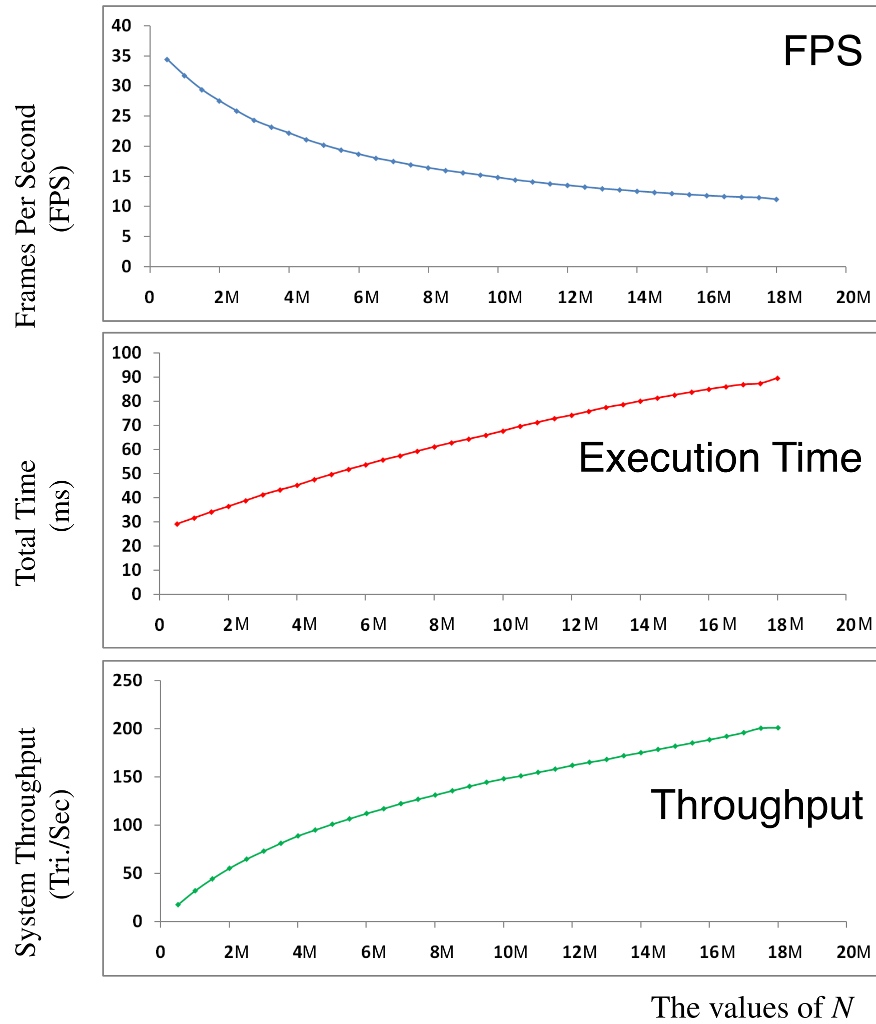


Figure 4.8: **The performance of LOD processing component with the Boeing 777 model on a NVIDIA GTX 680 device.**

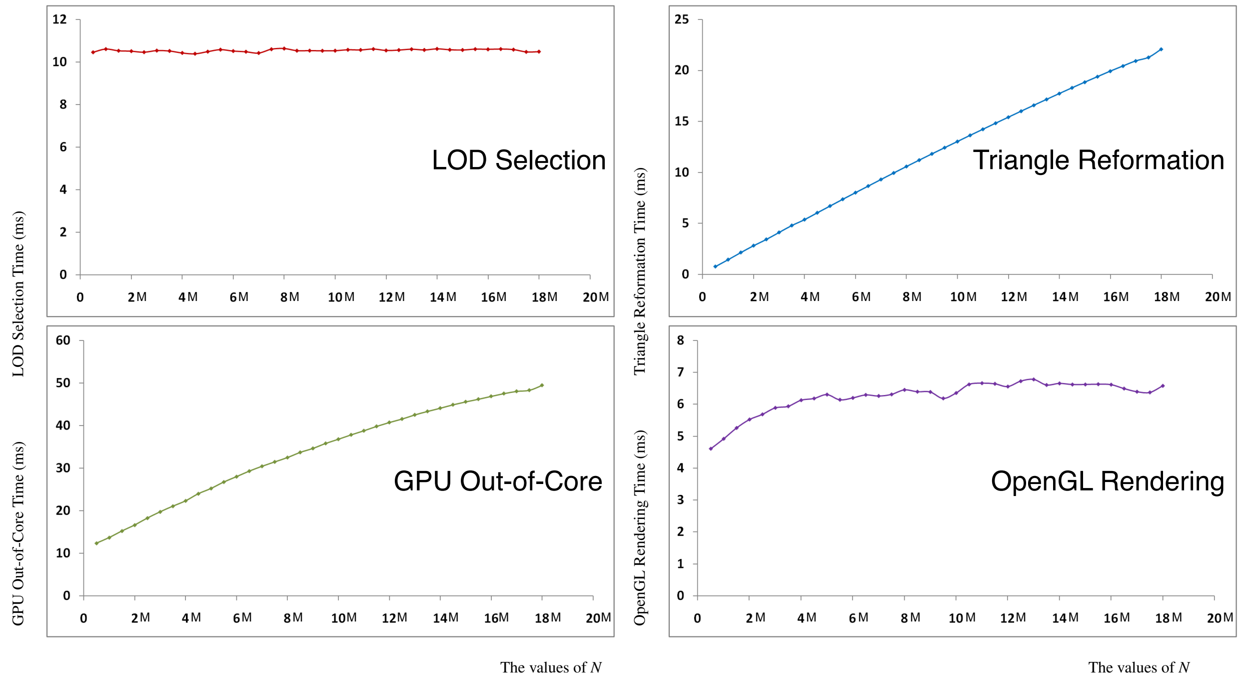


Figure 4.9: **The breakdowns of the LOD processing performance with the Boeing 777 model on a NVIDIA GTX 680 device.**

certain performance. The system throughput is a function of N (Equation 4.11). Since the maximal value of N is determined by the capacity of the GPU memory, the maximal throughput is bound to the size of GPU memory.

Table 4.5 shows the averaged breakdown times. the LOD selection algorithm uses the AABBs. Its execution time is constant regardless of values of N. Others, as explained in Section 4.7.5, scale linearly with the values of N. I scatter the pairs of their execution times and the values of N in Figure 4.9. Each dot in the graphs represents the execution time averaged over a sequence of rendered frames with a specific value of N. Transferring frame-different data and defragmenting the GPU data are heavy computations and make out-of-core component dominates in performance cost for the Boeing 777 model.

Table 4.5: **The breakdowns of runtime performance with only the LOD processing component on a NVIDIA GTX 680 device.** The results are averaged over all the frames of the created camera paths.

Models	N	Visible Triangles	FPS	Total Time	LOD Selection	GPU Out-of-Core	Triangle Reformation	OpenGL Rendering
Boeing 777	18.0M	27.2M	11.5	87.5 ms	11.4 ms (13.0%)	47.9 ms (54.7%)	21.6 ms (24.7%)	6.6 ms (7.5%)
	10.5M	15.8M	14.4	69.6 ms	11.5 ms (16.5%)	37.8 ms (54.3%)	13.6 ms (19.5%)	6.6 ms (9.5%)
	3.0M	4.4M	24.3	41.2 ms	11.5 ms (27.9%)	19.7 ms (47.8%)	4.1 ms (10.0%)	5.9 ms (14.3%)
Power Plant	5.5M	11.8M	46.5	21.5 ms	6.8 ms (31.6%)	—	8.4 ms (39.1%)	6.4 ms (29.8%)
	2.5M	5.7M	62.1	16.1 ms	6.9 ms (42.8%)	—	4.5 ms (28.0%)	4.7 ms (29.2%)
	0.5M	1.2M	108.1	9.3 ms	6.9 ms (74.2%)	—	1.3 ms (14.0%)	1.1 ms (11.8%)

Performance on Different GPU Devices

To better understand the system, I perform evaluation with the four GPU devices. Since their specifications are varied by the manufacturer, I want to understand how a specification influences the performance. The results may help the manufacturer design next generation cards. I use a part of the Boeing 777 model (storing less *ECol*s and AABBs on GPU). The data size is determined based on the GTX 280 since it has the smallest memory size, so that all devices are guaranteed to have the same workloads. The results are shown in Table 4.6. The newest device provides the best performance, because it has the highest number of compute capability. A higher number of compute capability indicates a newer version of the GPU architecture. It runs CUDA programs and OpenGL commands faster than the older ones. The columns of LOD Selection and Triangle

Table 4.6: **The performance on different GPU devices.** The value of *N* is set to be 3.5M for all devices. The number of corresponding triangles is 4.4M. All the devices are conducted with the same camera paths. We use a part of the Boeing 777 model that contains 69.2M vertices, 100.2M triangles and 295K objects.

Devices	**GPU Mem.**	**Memory Bandwidth**	**CUDA Cores**	**Compute Capability**	**FPS**	**LOD Sel.**	**GPU Out-of-Core**		**Triangle Reform.**	**OpenGL Rendering**
							Transferring	Defrag.		
GTX 280	1GB	141.7GB/s	240	1.3	12.4	28.7 ms	8.9 ms	21.9 ms	10.5 ms	10.7 ms
Quadro 5000	2.5GB	120.0GB/s	352	2.0 (Fermi)	15.6	10.9 ms	3.6 ms	10.6 ms	8.2 ms	4.1 ms
GTX 580	3GB	192.4GB/s	512	2.0 (Fermi)	38.9	8.7 ms	3.5 ms	6.0 ms	4.1 ms	3.4 ms
GTX 680	4GB	192.2GB/s	1536	3.0 (Kepler)	40.7	7.6 ms	3.7 ms	5.8 ms	4.0 ms	3.5 ms

Reformation are consistent to the execution time of their CUDA kernels. The device with more CUDA cores and higher memory bandwidth allow more threads to launch the kernel simultaneously and faster write results to global memory. I break the column of GPU Out-of-Core into the two sub-columns. The time spent on transferring the data is bound to the PCIe bandwidth and the GPU compute capability. I test the devices on the same motherboard, and find out that the device with higher compute capability transfers data faster. I also notice that the latest Kepler architecture does not have a significant improvement on host-device communication.

Chapter 5

Dual-GPU System: Distributing Data Between GPUs with a Dynamic Load Balancer

With the rapid hardware developments, a motherboard is commonly configured with multiple PCIe slots that enable the installation of two or more graphics cards. This configuration has high potentials to increase performance since it provides more computational power and more GPU memory. With additional GPU display ports, multiple display monitors are connected so that the rich information embedded in the massive data is rendered at much higher resolutions as what they should deserve. However, it is not trivial to transplant a parallel approach from a single-GPU system to a multi-GPU system. One reason is the lacks of both programming models and well-established

inter-GPU communication in the multi-GPU system. Although major GPU suppliers, such as NVIDIA and AMD, support multiple GPUs by establishing Scalable Link Interface (SLI) and Crossfire respectively, their technologies are primarily designed for gaming and lack the functionalities for both general programming and software implementations. Also, when enabling SLI or Crossfire, all GPUs behave as one hardware entity and per-GPU execution is not allowed. Another reason is the workload balancing issue between GPUs. Imbalanced workload distribution definitely hurts performance.

In this chapter, I present a device-level parallel system to real-time render massive models. I use a standard workstation configured with two GPU devices.

5.1 Related Works in Multi-GPU Applications

Nowadays, many computing systems have been built with multiple GPUs or GPU clusters for high-performance applications. Eilemann [64] summarized and analyzed existing approaches targeting on parallel rendering designs with multiple or clustered GPUs. One popular software package, *Equalizer*, has been commonly used in multi-GPU rendering society. As introduced in the paper [65], Equalizer is a scalable parallel rendering framework suitable for large data visualization and OpenGL-based applications. It utilizes a flexible compound tree structure to support its rendering and image compositing strategy. However, load balancing issues are unsolved and affecting the parallel performance.

Because of the importance of load balancing, Fogal et. al [66] discussed it by considering the data

transferring overhead and the visibility of GPU clusters in a distributed memory environment. Erol et. al [67] concentrated on the cross-segment load-balancing issues within Equalizer framework. Their work proposed a dynamic task partition strategy for the best usage of available shared graphics resources. Another approach presented by Marchesin et. al [68] was designed to dynamically balance the workloads. By utilizing frame-to-frame coherence, the approach redistributes data based on the historical frame rates. If one GPU has a higher frame rate in the pervious frames, it will be allocated with larger workloads in the coming frames; otherwise, its workloads will be reduced.

Multiple GPUs can visualize the data at a high resolution by porting the rendered image to the multiple display monitors connected to the GPUs. To achieve this, the tiled display systems have been widely used in many visualization researches. As discussed in [69], the entire picture is projected onto multiple display nodes with proportional viewports. Besides describing the system settings, the authors provided their solution for multi-display synchronization.

5.2 Research Questions and Goals

The goal of a dual-GPU system is to further improve performance. Here, I would like to review performance bottlenecks of the single-GPU system. I also address the load balancing issues in multiple-GPU programming.

Transferring the data from CPU to GPU during the runtime is a unavoidable process in large-scale data visualization. Although frame-to-frame coherence is applied to minimize the data-transferring

overhead, reorganizing the storages of vertices and triangles with the GPU defragmentation algorithm (see Section 3.4) spends significant computation time since it involves many operations of non-coalesced global memory accesses. The defragmentation time scales with the size of the data residing on the GPU. Defragmenting a large amount of GPU data greatly influences overall performance. With a multi-GPU system, I distribute data among GPUs, so that each GPU operates less data. As a result, the overhead caused by the defragmentation algorithm is reduced.

Using multiple GPUs is a trend to increase computational power and memory capacity. However, balancing the workload and resource utilizations between GPUs has not been satisfactorily addressed. Load balancing problems are centered around how to distribute the renderable data to GPUs. Imbalanced workload distribution causes underutilization of GPU resources and waste memory spaces.

Nvidia uses SLI to balance the geometry workload between two GPUs. SLI is a bridge that spans two GPUs to send data directly within a master-slave configuration. For example, the master GPU sends half of rendering work to the slave GPU. Then the slave GPU sends its output image back to the master for compositing. However, SLI is not suitable for the problems that I want to solve here. SLI does not incorporate with out-of-core techniques or CUDA developments. It requires all data to fit on the GPU. Also, This master-slave configuration is only for single-display applications, not suitable for multi-display applications, because the latter one connects each GPU to a display monitor, which breaks the master-slave concept. Therefore, it is essential to design a load balancer for the specific requirements of my application.

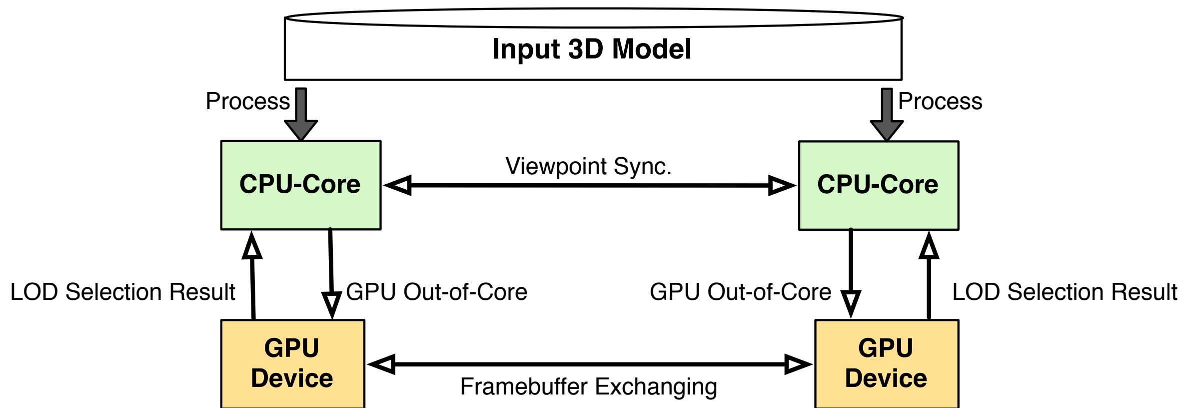


Figure 5.1: **The overview of the dual-GPU system.**

5.3 System Overview

In my dual-GPU system, One GPU is driven by one CPU core. One display monitor is connected to one GPU. The CPU core executes an instance of the program and feeds the data from the CPU main memory to the GPU that it associates with. The input 3D model and the runtime control parameters are shared among processes by employing a method of Inter-Process Communication (IPC). I illustrate the overview of the dual-GPU system in Figure 5.1. During the runtime, a dynamic load balancer calculates the optimal solution for distributing data across the GPUs. The inter-communication between GPUs is established to perform framebuffer exchanging as necessary for displaying on the screen.

5.4 Parallelism for CUDA-OpenGL Interoperability

When asking a GPU to perform both general-purpose and graphics computations, we need to the interoperation between Nvidia CUDA and OpenGL In the proposed parallel algorithms, CUDA is used for the components of occlusion culling and LOD processing, while OpenGL is used to render the simplified models. The CUDA-OpenGL interoperability in CUDA SDK can be easily applied to the single-GPU system, but the interoperation among multiple CUDA and OpenGL contexts in a multi-GPU system is an issue. In general, there are three possible solutions: (1) a single CPU thread to control multiple GPU devices; (2) multiple CPU threads, where one GPU is controlled by one CPU thread; (3) multiple processes, where one GPU is controlled by one process. In this section, I wold like to discuss them and choose the one best for my application.

5.4.1 A Single CPU Thread

Since the release of CUDA Toolkits 4.0, multi-GPU programming can be performed in a single CPU thread. CUDA kernel executions and contexts are switched between GPUs by calling *cudaSetDevice()*. But switching OpenGL contexts between GPUs with a singe CPU thread is not allowed. For example, *cudaSetGLSetDevice()* call be called only once at the start of the program. This restriction makes the single CPU thread implementation not compatible for my rendering applications.

5.4.2 Multiple CPU Threads

Creating multiple CPU threads allows one GPU thread to be bound with one CPU, so that each GPU has not only its own contexts but also its own host. A thread maintains its local storage and controls the device that it connects to. The OpenGL context issued by the CPU thread can interoperate with the CUDA context of the corresponding GPU. However, the problem is that OpenGL is not a thread-safe graphics API because of the asynchronous nature. The GL commands within different threads cannot be executed in-order as they are issued. When the driver schedules these calls, OpenGL contexts would switch frequently, which is particularly time-consuming and would decrease overall performance.

5.4.3 Multiple Processes

To eliminate the overheads caused by context switching, a multi-process strategy will be the solution to interoperate CUDA and OpenGL. One process communicates to one GPU, and maintains its own memory spaces and private runtime resources. The GL commands within a process are executed in-order without interruptions of other processes. Thus, in my implementation, I use the multi-process strategy plus the employments of Inter-Process Communication (IPC) to synchronize shared properties (e.g., camera viewpoints).

5.5 Load-Balanced Data Distribution

In the dual-GPU system, each GPU displays the half size of the image frame. It contains the data only appearing in its window. Unfortunately, this simple strategy usually results in poor performance and memory usage issues when the data distributed to the GPUs are imbalanced.

Of course, more GPUs mean more available memory, and indicate higher potentials to increase geometric complexities by adding more vertices and triangles to objects. However, imbalanced data distribution may be over the size limit of GPU memory. For example, in an extreme case, one GPU may obtain the entire size of selected triangles and vertices that exceed its memory maximum, while the other one is idling without any.

To solve this problem, I design a load balancer applied after the execution of the LOD selection algorithm. It uses a dynamic partitioning procedure to recursively split the volume of the view frustum. The balancer is executed in a parallel manner on the GPU. In the following subsections, I first introduce the fundamental method of view-frustum partitioning. Then I propose the dynamic load balancing algorithm.

5.5.1 Fundamental of View-Frustum Partitioning for Data Distribution

As shown in Figure 5.2, the view frustum is divided into two sub-frustums, each of which is associated to a GPU. For each visible object, I identify the GPU it belongs to by testing its AABB against the sub-frustums. If an object is not in a sub-frustum, the detail level of the object will be set to zero for its associated GPU. At the stage of rasterization, the GL calls of perspective

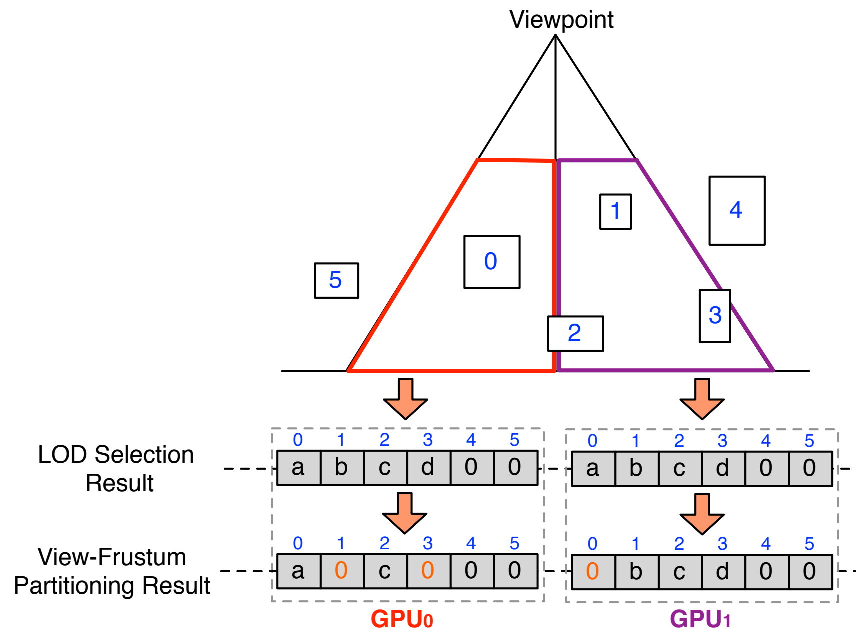


Figure 5.2: **The view-frustum partitioning.** The indices $0-5$ represent the objects bounding boxes. $a-d$ stand for the desired primitive counts from the LOD selection algorithm.

and projection transformations take the full size of the view frustum; but, in order to display the contents on the screen, the viewport only needs to be set with the half of the framebuffer containing the rendered objects.

Obviously, the fundamental approach leads to load-balancing problems since it always distributes the data by partitioning the frustum statically. The static method distributes the vertices and triangles by evenly splitting the frustum at its near plane rather than by balancing the workloads of GPUs. To achieve the optimal rendering performance, I present a dynamic load-balancing algorithm. At a given viewpoint, the view frustum is dynamically split by balancing the number of primitives between GPUs. The rendered images will be exchanged between GPUs to adjust the image projection with a inter-process communication technique. For example, as illustrated in

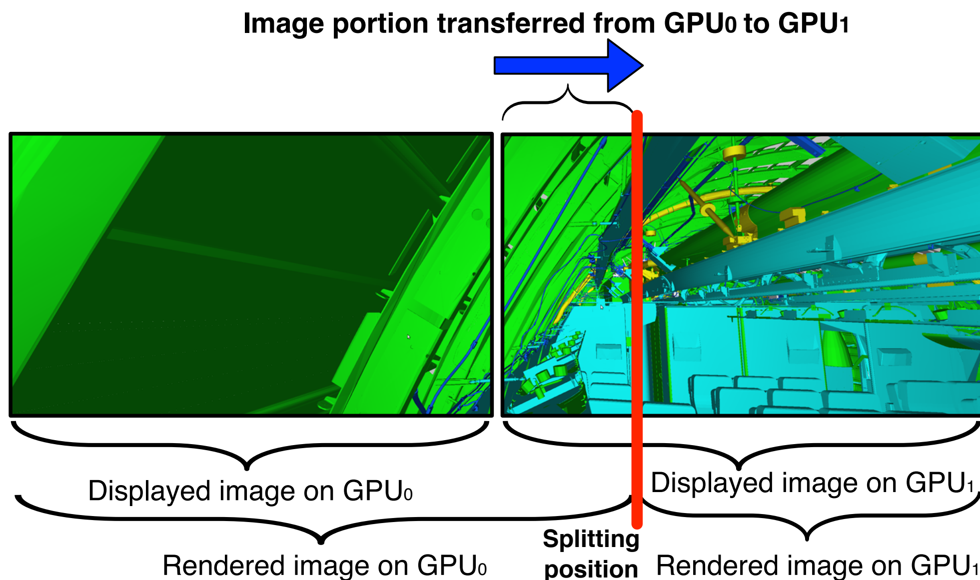


Figure 5.3: **The concept of the dynamic load-balancing algorithm.** The whole screen is split by balancing the number of primitives distributed between GPUs. In this example, GPU_0 transfers a potion of the image to GPU_1.

Figure 5.3, the numbers of vertices and triangles are balanced between GPU_0 and GPU_1. GPU_0 renders a larger part of the screen, so that it transfers an image portion to GPU_1 to ensure the correctness of viewport transformation.

5.5.2 Parallel Dynamic Load-Balancing Algorithm

The load-balancing algorithm is described in Algorithm 6. The goal of this algorithm is to calculate the position to split the view frustum on the screen space. In the algorithm, the returned value, $split$, ranges between (0, 1). It tells the splitting position on the near plane. The algorithm is executed in a per-process manner. The vf represents the view frustum defined by the camera, and the *id* is the index of the process. Here, I balance the workload by balancing the number of selected triangles

given to each GPU, denoted as tc (refer to Section 2.4). The number of selected triangles for the ith object is denoted as tc_i.

Algorithm 6 Load-Balancing Algorithm

Input: id, vf, threshold, tc
Output: split, tc

1: $split \leftarrow 0.5$;
2: $increment \leftarrow 0.5$;
3: $subF_l \leftarrow vf$;
4: **while** 1 **do**
5:    **UpdateSubFrustum**$(subF_l, split, vf)$; // updating the left sub-frustum
6:    // balancing the amount of data
7:    **for** the ith object's AABB **in parallel do**
8:       **if** $AABB_i$ inside $subF_l$ **then**
9:          $tc_i^l \leftarrow tc_i$;
10:          $tc_i^r \leftarrow 0$;
11:       **else**
12:          $tc_i^r \leftarrow tc_i$;
13:          $tc_i^l \leftarrow 0$;
14:       **end if**
15:    **end for**
16:    $sum^l \leftarrow$ the sum of the elements in tc^l **in parallel**;
17:    $sum^r \leftarrow$ the sum of the elements in tc^r **in parallel**;
18:    $ratio \leftarrow sum^l / sum^r$;
19:    $increment \leftarrow increment \times 0.5$;
20:    **if** $ratio < 1 - LBT$ **then**
21:       $split \leftarrow split + increment$;
22:    **else if** $ratio > 1 + LBT$ **then**
23:       $split \leftarrow split - increment$;
24:    **else**
25:       $tc \leftarrow tc^{id}$;
26:       **return**;
27:    **end if**
28: **end while**

In the initialization of Algorithm 6, the value of $split$ is set to be 0.5 indicating that the view frustum is divided at the middle. Then, the algorithm iteratively finds the optimal splitting value. At each

iteration, the sub-frustum is updated. The left sub-frustum is denoted as $subF_l$. The AABBs of objects are tested against the $subF_l$ to update the tc for the GPU. tc^l represents the array of triangle counts for GPU rendering the left part of the screen; and tc^r is for the GPU rendering the right part (refer to Line 8-16 in Algorithm 6). I employ a CUDA implementation to compute tc^l and tc^r in parallel. In Line 19, the *ratio* is defined to find out if the value of the *split* has reached the satisfaction by comparing it to the value of predefined *Load-Balancing Threshold (LBT)*. Ideally, setting the value of the *LBT* to be 0.0 will distribute the data evenly. However, this may need too many iterations, which would potentially slow down performance. It is better to learn an appropriate value for the *LBT* in practice that weights between the execution time of the load balancer and the distribution proportion. After that, each GPU renders its own objects classified by the splitting value. As shown in Figure 5.3, the GPU receiving a larger portion of the view frustum will send the unwanted portion of its output image to the other GPU.

5.6 Synchronization and Inter-GPU Communication

In NVIDIA CUDA programming environments, GPUs cannot interact with each other directly. The only way of the inter-GPU communication is through the controls of the inter-CPU communication, where each GPU is controlled by a CPU core. I use Massage-Passing Interface (MPI) for inter-process communication. MPI is a specification that moves the data among processes through cooperative operations on each. MPI is portable, hardware optimized and widely used in High Performance Computing (HPC). Recently, researchers and developers have demon-

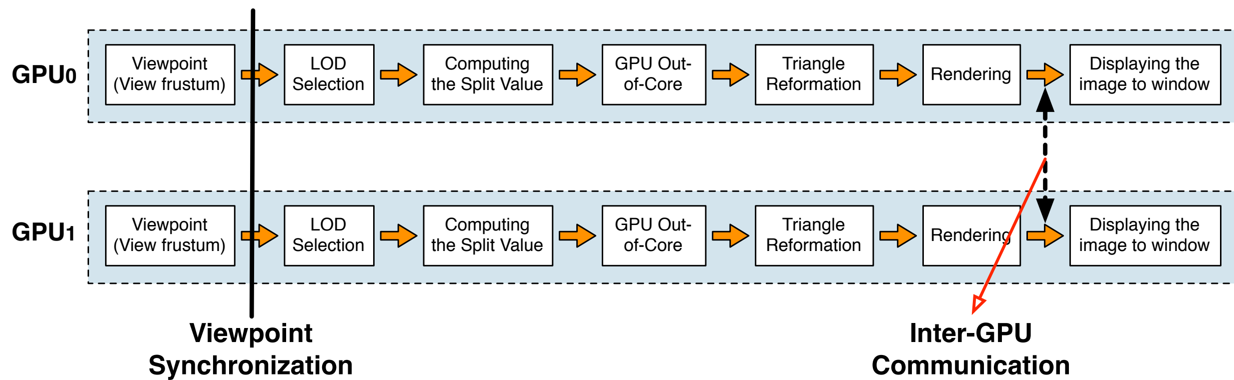


Figure 5.4: **The principles of synchronizations and communications between GPUs.**

strated the efficiency of multi-GPU applications using MPI to facilitate GPU data movements, such as [70, 71, 72]. In this section, I would like to discuss the MPI methods of synchronization and the IPC method used in the dual-GPU system.

The Synchronization and communication is to make the GPUs behave as co-workers. Figure 5.4 shows their principles in the system, which meet the requirements of: (1) controlling the camera only by the active process and sharing the camera configurations between the processes; (2) exchanging the framebuffers between the GPUs. I explain them in the following subsections.

5.6.1 Camera Synchronization

When updating the camera, only one process is active to response the mouse/keyboard callback events. I use the shared memory to synchronize the camera between the processes. It can be accessed by the processes simultaneously without any redundant copy. I select one process as the master, and others send their activation status to the master. The master identifies which process is activated and broadcasts the index of this process to all others. Then, after the active process

finishes the updates of the camera in the shared memory, it will broadcast a finish flag. When other processes received the flag, they read the updated values for their local computations. I use MPI blocking calls to suspend process executions until the values in the shared memory are safe to use. This is because non-blocking calls return immediately after the calls are initiated without waiting for their finish. In that situation, nonactive processes may execute faster than the active one and read the values that have not been updated. Then, LOD selection and GPU out-of-core would produce wrong results. Therefore, using MPI blocking calls is a safe way to guarantee the camera synchronized for all processes.

5.6.2 CUDA Inter-Process Communication

When all GPUs finish their rendering tasks, the output images are maintained in framebuffer. The GPU assigned with a larger part of the screen gives its unwanted portion of the output to the other GPU. Traditionally, if a GPU sends data to another GPU, it first sends the data to the shared memory on the CPU host. Then, this host copy of the data is sent to the target GPU. Although the CUDA driver supports pinned memory allocation to the main memory (e.g., page-locked buffer on RAM) to help reduce the CPU-to-GPU communication overhead, the meander progress of moving data through the PCIe bus still reduces performance and increases the needs of host resources.

Since the release of CUDA 4.1, NVIDIA has removed such limitations by introducing an efficient method of inter-process GPU-to-GPU communication, so-called as CUDA-IPC. It significantly reduces the overhead of GPU-to-GPU communication by directly transferring data through PCIe

bus rather than through the host. A process creates the handle mapping a buffer on the source GPU. Then, the handle is sent to the target process with the help of MPI host-based communication. The target process opens the handle and tell its GPU to require a direct transfer from the source GPU. This feature offers us an efficient way of exchanging framebuffers between the GPUs.

The roles of source and target are switched based on the value of the $split$ defined in Algorithm 6. If $split = 0.5$, the screen is partitioned evenly and no need for exchanging; if $split < 0.5$, the GPU associated with the left display monitor will be the target that receive the data from the source associated to the right monitor; If $split > 0.5$, the roles between GPUs will be switched the other way round.

5.7 Implementation

I have implemented the dual-GPU system on the workstation equipped with an AMD Phenom X6 1100T 3.30GHz CPU (6 cores), 16 GB RAM, and two Nvidia GTX 580 graphics cards, each of which has 3GB GDDR5 device memory. The system is developed using C++, Nvidia CUDA Toolkit 4.2 and OpenGL on a 64-bit Linux system. The Linux system has better programmability than Microsoft Windows system for multi-GPU programming. For example, CUDA-IPC is supported only on the Linux system. In my benchmarks, the resolution of OpenGL framebuffer is set to be 1024×1024. The value of the LBT (see Section 5.5.2) is set to be 0.3.

Table 5.1: **The performance breakdowns of Dual-GPU(B), Dual-GPU(NB) and Single-GPU systems.**

Approaches	FPS	Triangle Diff.	Visible Triangles	LOD Selection	Split	GPU Out-Of-Core	Triangle Reformation	OpenGL Rendering
Single-GPU	14.9	—	12.3M	3.4ms	—	29.6ms	3.6ms	30.2ms
Dual-GPU(NB)	17.8	7.9M	12.3M	3.4ms	—	24.5ms	2.9ms	25.3ms
Dual-GPU(B)	20.4	0.4M	12.3M	4.0ms	5.4ms	18.6ms	2.0ms	19.1ms

5.8 Performance Evaluation

To evaluate the performance of the dual-GPU system, I compare my implementation, Dual-GPU with Load Balancer (Dual-GPU(B)), to other two implementations: Dual-GPU without Load Balancer (Dual-GPU(NB)) and Single-GPU. The three implementations are tested with the same camera paths and constrained with the same amounts of vertices and triangles. The Single-GPU assigns the workloads to only one GPU; The Dual-GPU(NB) always splits the screen at the middle, so that it does not have costs from the execution of the load balancer. For both dual-GPU implementations, the runtime performance is bound to the slower-performed GPU. As a result, the Dual-GPU(B) achieves 1.14 and 1.37 times faster than the Dual-GPU(NB) and the Single-GPU, respectively.

Table 5.1 shows the performance breakdown of the three implementations. The values in the table are averaged over a total of 300 rendered frames. The timing results of Dual-GPU(NB) and Dual-GPU(B) are recorded from the slower-performed GPU at each frame. The column of ”Triangle Diff.” means the difference of the numbers of triangles between two GPUs, which is not applicable in Single-GPU implementation. The column of “ Visible Triangles” means the number

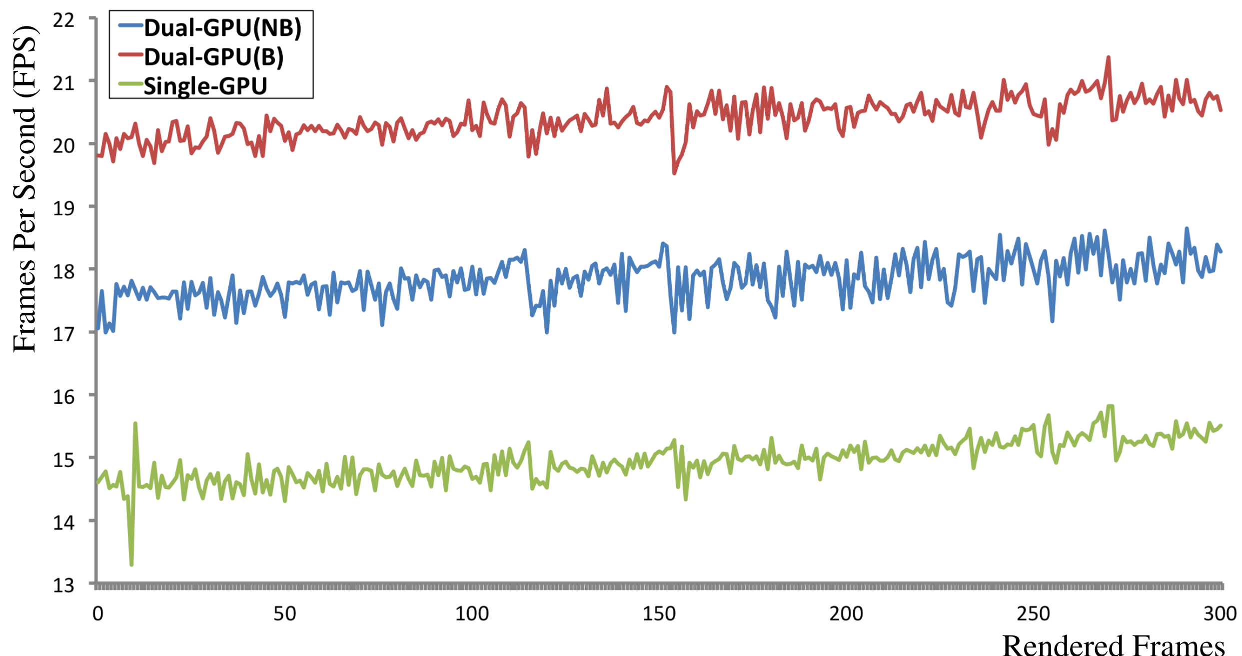


Figure 5.5: **The performance comparison among the three systems: Dual-GPU(B), Dual-GPU(NB) and Single-GPU.**

of triangles selected to generate the simplified version of the model. Since the numbers of visible triangles are the same among the three implementations, the comparison is ensured with the same rendering quality and GPU workloads. The last five columns show the timing results of computing components.

The performance of dual-GPU systems are higher than the single-GPU system. This is because the selected vertices and triangles distributed each GPU make the GPU have less workload than the one in the Single-GPU. The column of "Split" is the computation time that the load-balancer spend on finding the partitioning position of the view frustum. The Single-GPU and Dual-GPU(NB) do not have a load-balancer, so that they do not have any value in the"Split". But, they have significant costs on the "GPU Defragmentation", which make their performance slower than the

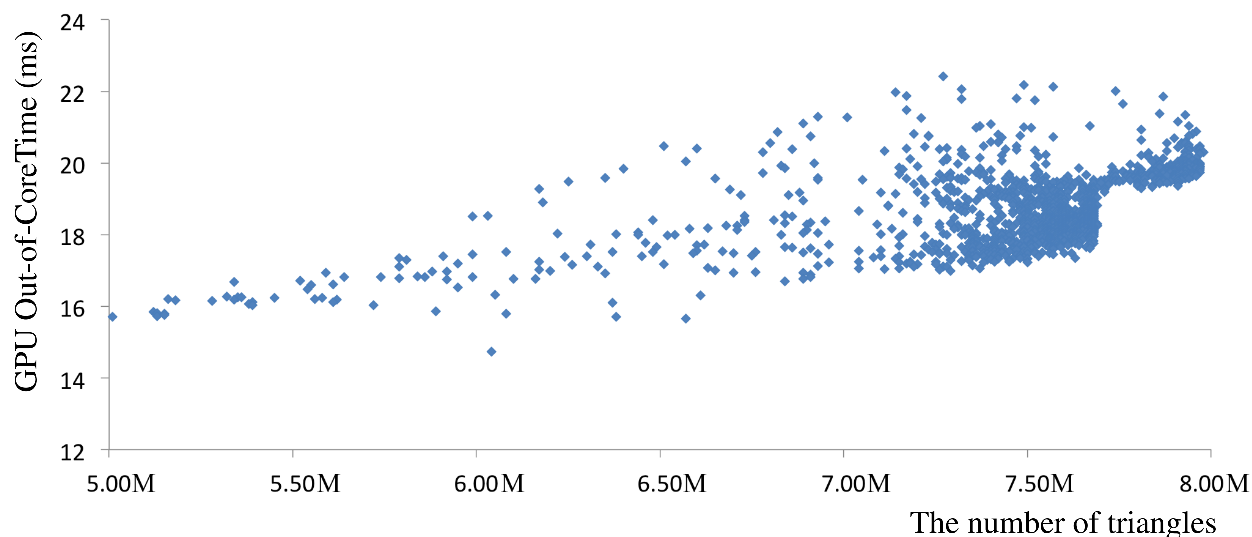


Figure 5.6: **The scattered value pairs of GPU out-of-core time and the number of triangles on a GPU from 600 rendered frames.**

Dual-GPU(B). I plot the FPS comparison of the three implementations in Figure 5.5. The Dual-GPU(B) achieves the best FPS because of the advantages of the load-balancing algorithm.

Now, I would like to give more analysis between the two dual-GPU implementations. The performance of a GPU scales with the amount of data it operates with. The larger value is in the column of the "Triangle Diff." in Table 5.1, the bigger difference in performance between the GPUs will be. Although the Dual-GPU(B) spends the additional cost to balance the workload distribution, the computation time spent on the GPU out-of-core, its costs in the columns of the "Triangle Reformation" and "GL Rendering" are lower than Dual-GPU(NB). Although the dual-GPU system gives less data for each GPU, the GPU out-of-core algorithm is still the most time-comsuming computation. In Figure 5.6, I plot the relation between GPU out-of-core time and the number of triangles. Each dot in the graph represents a rendered frame. It shows that the time spend on the out-of-core computation increases towards the increase of the number of triangles.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

In this dissertation, I presented an end-to-end GPU-based parallel solution for view-dependently rendering the multi-object massive model that contains several gigabytes worth of vertices and triangles. The proposed algorithms deliver performance suitable for visualizing and interacting with such large datasets, by leveraging the GPU's parallel architecture.

With the preprocessing method, the data dependency of traditional mesh simplification algorithms is eliminated. The LOD selection algorithm supports adaptive level-of-detail controls to satisfy different rendering constraints. The triangle reformation algorithm exploits the triangle-level parallelism on the GPU, such that the desired LOD model is generated in a view-dependent manner. The GPU out-of-core algorithm minimizes the overhead of transferring data from the CPU main

memory to the GPU memory by taking advantages of frame-to-frame coherence. The defragmentation algorithm reorganizes the data storage to maintain their continuity in the GPU memory. As a result, I have achieved highly interactive frame rates that dutifully exhibit complicated mesh topology, complex geometry and varying color attributes.

The proposed massive model rendering system seamlessly integrates the algorithms concerning mesh simplifications, occlusion culling on the GPU. The parallel occlusion culling algorithm provides an object-level parallelism. The released memory from the hidden objects is used to increase the details of visible objects. In order to further improve rendering performance, I have employed a dual-GPU system, where the workload distributed to each GPU has been balanced with the proposed dynamic load balancer.

6.2 Future Works

Aside from continuing to improve the implementation of the algorithms outlined in this dissertation, there are other avenues of research which merit further investigation. I describe them in the following subsections.

6.2.1 Applicability to 3D-Scanned Surface Models

A possible extension of my work is applying the proposed algorithms to 3D-scanned surface models. Different from multi-object models, a surface model is a single solid-appearing object that

contains many triangles for representing fine details. The proposed LOD selection and occlusion culling algorithms cannot be directly applied to the surface model since they are executed at an object-level parallelism. One idea could be integrating them with traditional multiresolution approaches. The input surface model is represented as a hierarchy by binary space partitioning the triangles of the model. Leaf nodes would contain a fixed maximal number of original triangles. Inner nodes are generated by merging from its two child nodes at the lower level of the hierarchy.

The active nodes for a specific camera can be selected by a bottom-up node merging process rather than top-down traversal. At each merging step, nodes of the same level are treated as individual objects and applied with the object-level parallel algorithms.

After the active nodes are identified, the desired LOD model can be generated using the proposed triangle reformation algorithm. To preserve the topological connection of neighbor nodes, the triangles residing on the node boundaries are not simplified.

6.2.2 In-Place GPU Memory Defragmentation

Improving the GPU out-of-core algorithm is another potential work. The defragmentation algorithm illustrated in Figure 3.2 uses three buffers to store the currently selected data, existing data and frame-different data. The size of frame-different data is small since the temporal coherence has been applied at every time of changing the camera. But the existing data takes more memory. It is a copy of the selected data at the previously rendered frame, which is the same size as the data used for the currently rendering frame.

Thus, developing a space-efficient algorithm can improve the GPU memory usage. Such demands are also growing in embedded software, which is often constrained by limited amounts of memory. For example, using an in-place algorithm to defragment the GPU memory could be a way to remove the required memory buffer used for storing the existing data. An in-place algorithm transforms the data using small amount of storage space. While executing the algorithm, the output data is overwritten by the new input data. In this way, the saved memory from the in-place defragmentation could be used for rendering more triangles.

6.2.3 Load Balancing and Inter-GPU Communication in a Multi-GPU System

Today, modern motherboards are capable of multiple GPU installation in single shared memory architecture. Using these multiple GPU computing resources give us an opportunity to program at a device level of parallelism and further scale-up performance. As a long-term goal, I would like to focus on multi-GPU (more than two GPUs) research for visualization applications, which include workload balancing among GPUs, as well as inter-GPU communication on large-scale high-resolution displays.

Keeping balanced workload per device is essential for efficiently using the GPUs. Improving the proposed view-frustum partitioning method is one promising direction. The workloads are balanced by offloading an equal number of triangles to each partition, or subview frustum. The objects containing these triangles in the subview frustum can simplified on the corresponding GPU

device.

A dynamic load balancing scheme meets the requirement of view-dependent rendering. However, such schemes typically require run-time dynamic data transfers across devices. Moving data in and out of GPUs is a serious performance bottleneck. Although NVIDIA introduced the functionality of inter-processing communication and point-to-point data transfer over PCI Express, scheduling those GPU-to-GPU communications could be a challenging task. A successful scheme should efficiently monitor the order of running communication processes, their priorities and device-level synchronization.

Bibliography

[1] NVIDIA, *NVIDIA CUDA C Programming Guide*, NVIDIA, 2013. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] T.-D. Hoang and K.-L. Low, “Multi-resolution screen-space ambient occlusion,” in *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, ser. VRST ’10. New York, NY, USA: ACM, 2010, pp. 101–102.

[3] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, “The diverse and exploding digital universe.” 2008, an update forecast of worldwide information growth through 2011. IDC white paper.

[4] S. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, “Real-time massive model rendering,” *Synthesis Lectures on Computer Graphics and Animation*, vol. 2, no. 1, pp. 1–122, 2008.

[5] A. M. MacEachren and M.-J. Kraak, “Research challenges in geovisualization,” *Cartography and Geographic Information Science*, vol. 28, no. 1, pp. 3–12, 2001.

[6] R. J. Lisle, “Google earth: a new geological resource,” *Geology Today*, vol. 22, no. 1, pp. 29–32, 2006.

[7] C. Johnson, “Top scientific visualization research problems,” *Computer Graphics and Applications, IEEE*, vol. 24, no. 4, pp. 13–17, 2004.

[8] H. Hoppe, “Progressive meshes,” in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 99–108.

[9] H. Hoppe, “View-dependent refinement of progressive meshes,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 189–198.

[10] M. Garland and P. S. Heckbert, “Surface simplification using quadric error metrics,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 209–216.

[11] M. Garland and P. Heckbert, “Simplifying surfaces with color and texture using quadric error metrics,” in *Ninth IEEE Visualization(VIS ’98)*, 1998, p. pp.264.

[12] C. DeCoro and N. Tatarchuk, “Real-time mesh simplification using the gpu,” in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ser. I3D ’07. New York, NY, USA: ACM, 2007, pp. 161–166.

[13] L. Hu, P. V. Sander, and H. Hoppe, “Parallel view-dependent refinement of progressive meshes,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ser. I3D ’09. New York, NY, USA: ACM, 2009, pp. 169–176.

[14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, June 2010.

[15] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–10.

[16] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, “Dynamic load balancing on single- and multi-gpu systems,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–12.

[17] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney, *Level of Detail for 3D Graphics*. New York, NY, USA: Elsevier Science Inc., 2002.

[18] M. Giegl and M. Wimmer, “Unpopping: Solving the image-space blend problem for smooth discrete lod transitions,” *Computer Graphics Forum*, vol. 26, no. 1, pp. 46–49, 2007.

[19] J. Swarovsky, “Extreme detail graphics,” in *Game Developers Conference*, 1999, pp. 899–904.

[20] J. C. Xia, J. El-Sana, and A. Varshney, "Adaptive real-time level-of-detail-based rendering for polygonal models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pp. 171–183, April 1997.

[21] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, "Quick-vdr: Interactive view-dependent rendering of massive models," in *Proceedings of the conference on Visualization '04*, ser. VIS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 131–138.

[22] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh optimization," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 19–26.

[23] P. Lindstrom, "Out-of-core simplification of large polygonal models," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 259–262.

[24] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '02. New York, NY, USA: ACM, 2002, pp. 339–346.

[25] M. Garland and Y. Zhou, "Quadric-based simplification in any dimension," *ACM Trans. Graph.*, vol. 24, pp. 209–239, April 2005.

[26] D. Luebke and C. Erikson, "View-dependent simplification of arbitrary polygonal environments," in *Proceedings of the 24th annual conference on Computer graphics and interactive*

techniques, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 199–208.

[27] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *Proceedings of the conference on Visualization '98*, ser. VIS '98. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 35–42.

[28] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Bdam-batched dynamic adaptive meshes for high performance terrain visualization," *Computer Graphics Forum*, vol. 22, no. 3, pp. 505–514, 2003.

[29] E. Gobbetti and F. Marton, "Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 878–885.

[30] E. Shaffer and M. Garland, "A multiresolution representation for massive meshes," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 11, no. 2, pp. 139–148, 2005.

[31] E. Derzapf, N. Menzel, and M. Guthe, "Parallel view-dependent refinement of compact progressive meshes," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2010, pp. 53–62.

[32] E. Derzapf, N. Menzel, and M. Guthe, "Parallel view-dependent out-of-core progressive meshes." in *Vision, Modeling, and Visualization*, 2010, pp. 25–32.

[33] C. Peng, S. I. Park, Y. Cao, and J. Tian, "A real-time system for crowd rendering: Parallel lod and texture-preserving approach on gpu," in *The Fourth International Conference on Motion in Games (MIG2011)*, ser. LNCS 7060. Springer, 2011, pp. 27–38.

[34] C. Peng and Y. Cao, "Gpu-based streaming for parallel level of detail on massive model rendering," Computer Science, Virginia Tech, Tech. Rep. TR-11-12, 2011.

[35] C. Peng and Y. Cao, "A gpu-based approach for massive model rendering with frame-to-frame coherence," *Comp. Graph. Forum*, vol. 31, no. 2pt2, pp. 393–402, May 2012.

[36] S. Melax, "A simple, fast, and effective polygon reduction algorithm." in *Game Developer*, 1998, pp. 44–49.

[37] T. A. Funkhouser and C. H. Séquin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 247–254.

[38] M. Wimmer and D. Schmalstieg, "Load balancing for smooth levels of detail," Vienna University of Technology, Tech. Rep. TR-186-2-98-31, 1998.

[39] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems*, pp. 359–371, 2011.

[40] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha, "Mmr: an interactive

massive model rendering system using geometric and image-based acceleration," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, ser. I3D '99. New York, NY, USA: ACM, 1999, pp. 199–206.

[41] G. Varadhan and D. Manocha, "Out-of-core rendering of massive geometric environments," in *Visualization, 2002. VIS 2002. IEEE*, 2002, pp. 69–76.

[42] W. T. Correa, J. T. Klosowski, and C. T. Silva, "Visibility-based prefetching for interactive out-of-core rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, ser. PVG '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 2–.

[43] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, "Quick-vdr: Out-of-core view-dependent rendering of gigantic models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, pp. 369–382, July 2005.

[44] J. Kontkanen, E. Tabellion, and R. S. Overbeck, "Coherent out-of-core point-based global illumination," *Computer Graphics Forum*, vol. 30, no. 4, pp. 1353–1360, 2011.

[45] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 796–803, Aug. 2004.

[46] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand, "A survey of visibility for walk-through applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412–431, July 2003.

[47] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr., "Towards image realism with interactive update rates in complex virtual building environments," in *Proceedings of the 1990 symposium on Interactive 3D graphics*, ser. I3D '90. New York, NY, USA: ACM, 1990, pp. 41–50.

[48] S. J. Teller and C. H. Séquin, "Visibility preprocessing for interactive walkthroughs," *SIGGRAPH Comput. Graph.*, vol. 25, no. 4, pp. 61–70, July 1991.

[49] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 231–238.

[50] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang, "Accelerated occlusion culling using shadow frusta," in *Proceedings of the thirteenth annual symposium on Computational geometry*, ser. SCG '97. New York, NY, USA: ACM, 1997, pp. 1–10.

[51] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff, III, "Visibility culling using hierarchical occlusion maps," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 77–88.

[52] J. T. Klosowski and C. T. Silva, "Efficient conservative visibility culling using the prioritized-layered projection algorithm," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, pp. 365–379, October 2001.

[53] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent hierarchical culling: Hardware occlusion queries made useful," *Computer Graphics Forum*, vol. 23, no. 3, pp. 615–624, 2004.

[54] T. Süß, C. Koch, C. Jähn, and M. Fischer, "Approximative occlusion culling using the hull tree," in *Proceedings of Graphics Interface 2011*, ser. GI '11. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2011, pp. 79–86.

[55] S. Morein, "Ati radeon hyperz technology," in *Presentation at Workshop on Graphics Hardware, Hot3D Proceedings, ACM SIGGRAPH/Eurographics*, 2000.

[56] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha, "Interactive visibility culling in complex environments using occlusion-switches," in *Proceedings of the 2003 symposium on Interactive 3D graphics*, ser. I3D '03. New York, NY, USA: ACM, 2003, pp. 103–112.

[57] W. V. Baxter, III, A. Sud, N. K. Govindaraju, and D. Manocha, "Gigawalk: interactive walkthrough of complex environments," in *Proceedings of the 13th Eurographics workshop on Rendering*, ser. EGRW '02. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 203–214.

[58] H. Xiong, H. Peng, A. Qin, and J. Shi, "Parallel strategies of occlusion culling on cluster of gpus: Research articles," *Comput. Animat. Virtual Worlds*, vol. 18, pp. 165–177, July 2007.

[59] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet, “Integrating occlusion culling and levels of detail through hardly-visible sets,” *Computer Graphics Forum*, vol. 19, no. 3, pp. 499–506, 2000.

[60] J. El-Sana, N. Sokolovsky, and C. T. Silva, “Integrating occlusion culling with view-dependent rendering,” in *Proceedings of the conference on Visualization ’01*, ser. VIS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 371–378.

[61] S.-E. Yoon, B. Salomon, and D. Manocha, “Interactive view-dependent rendering with conservative occlusion culling in complex environments,” in *Proceedings of the 14th IEEE Visualization 2003 (VIS’03)*, ser. VIS ’03. Washington, DC, USA: IEEE Computer Society, 2003, p. 22.

[62] B. Li, C. Wang, and L. Li, “Efficient occlusion culling with occupancy proportion,” in *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1058–1061.

[63] T. Aila and V. Miettinen, “dpvs: An occlusion culling system for massive dynamic environments,” *IEEE Comput. Graph. Appl.*, vol. 24, pp. 86–97, March 2004.

[64] S. Eilemann, “An analysis of parallel rendering systems,” 2007.

[65] S. Eilemann, M. Makhinya, and R. Pajarola, “Equalizer: A scalable parallel rendering framework,” *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 3, pp. 436–452, 2009.

[66] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-gpu clusters," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 57–66.

[67] F. Erol, S. Eilemann, and R. Pajarola, "Cross-segment load balancing in parallel rendering," in *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, ser. EG PGV'11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 41–50.

[68] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *6th Eurographics Symposium on Parallel Graphics and Visualization*, May 2006, pp. 43–50.

[69] S. Deshpande, C. Yuan, and I. Daly, Scott asnd Sezan, "A large ultra high resolution tiled display system: Architecture, technologies, applications, and tools," in *16th International Display Workshops*, 2009.

[70] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. Panda, "Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Träff, S. Benkner, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2012, vol. 7490, pp. 110–120.

[71] S. Potluri, H. Wang, D. Bureddy, A. Singh, C. Rosales, and D. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *Parallel*

and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, may 2012, pp. 1848 –1857.

[72] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and D. Panda, "Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, sept. 2011, pp. 308 –316.