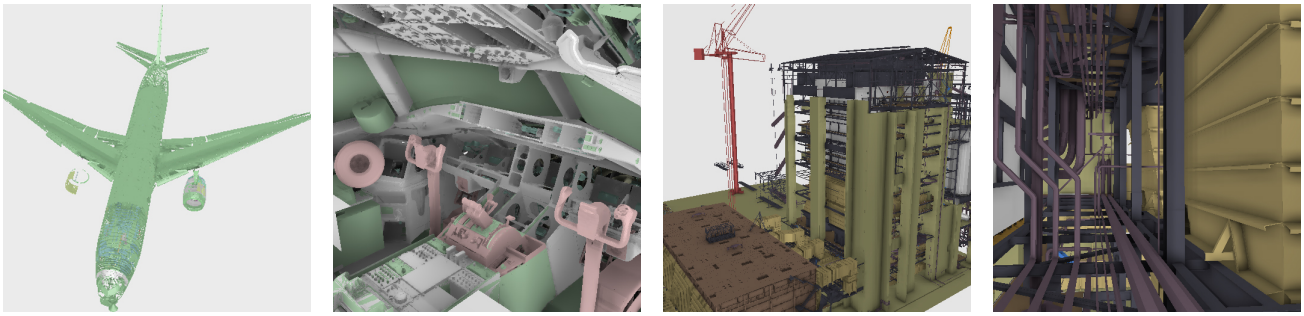
Integrating Occlusion Culling with Parallel LOD for Rendering Complex 3D Environments on GPU

Chao Peng *
Department of Computer Science
Virginia Tech, Blacksburg, VA, USA

Yong Cao †
Department of Computer Science
Virginia Tech, Blacksburg, VA, USA



(a) (b) (c) (d)

Figure 1: The images rendered by our *Parallel-GigaVis*. (a)-(b) *The Boeing 777 model;* **(c)-(d)** *The Power Plant model.*

Abstract

Real-time rendering of complex 3D models is still a very challenging task. Recently, many GPU-based level-of-detail (LOD) algorithms have been proposed to decrease the complexity of 3D models in a parallel fashion. However, LOD approaches alone are not sufficient to reduce the amount of geometry data for interactive rendering of massive scale models. Visibility-based culling, especially occlusion culling, has to be introduced to the rendering pipeline for large models. In this paper, we aim to tackle the challenge of integrated parallel processing for both mesh simplification and occlusion culling. We present a novel rendering approach that seamlessly integrates parallel LOD algorithm, parallel occlusion culling and Out-of-Core method in a unified scheme towards GPU architectures. The result shows the parallel occlusion culling significantly reduces the required complexity of the 3D model and increases the rendering performance.

Keywords: Parallel rendering, parallel LOD, parallel visibility culling, GPU out-of-core

1 Introduction

In many computer graphics applications, such as mechanical engineering, game development and virtual reality, a typical dataset from these applications is usually produced in a multi-object manner for efficient data management. Each object usually contains a relatively small number of polygonal primitives as long as they are sufficient to describe the topological properties of the object. To have a complete description of the whole model, tens of thousands of, or even millions of, such individual objects are necessarily created and loosely connected, which makes the entire dataset exceptionally complex. As data complexity continue to increase due to the fundamental advances in modeling and simulation technologies, a complex 3D model may need several gigabytes in storage. Consequently, visualizing the model becomes a computationally intensive process that impedes a real-time rendering and interaction.

Recently, massive parallelism in GPUs has become a major trend for high-performance applications. Today's GPUs can perform general purpose computation and allow researchers to solve problems by delivering fine-grained parallel processes. However, the requirement to interactively render gigabyte-scale models usually overburdens the computational power and memory capacity of the GPUs. To solve this problem, GPU-based parallel mesh simplification algorithms, such as [Hu et al. 2009; Derzapf et al. 2010; Peng et al. 2011]), have been proposed to fast simplify complex models. However, without considering view-parameters, the occluded objects are undesirably rendered in high levels of details, where processing them consumes many computational resources. The waste of GPU's computing power and memory for rendering those occluded objects will definitely hurt the overall performance and visual quality, especially when dealing with large-scale complex models.

For further improvement, more unnecessary polygonal primitives need to be excluded from the GPU processing pipeline. To do this, visibility culling techniques are commonly used to reject rendering of the invisible objects. For example, view-frustum culling is able to determine an object's visibility by testing its bounding volume against the view frustum. But performing view-frustum culling alone is not sufficient to handle the model that has a high depth complexity. To handle high-depth models, occlusion culling is used to disable rendering of the objects obscured by others. Although GPU-accelerated occlusion culling approaches have been introduced in the past, the integration with parallel LOD algorithms has not been satisfactorily addressed, which we believe is essential

*e-mail:chaopeng@vt.edu
†e-mail:yongcao@vt.edu

to speed up the processing of complex 3D models on GPU.

Main contributions. In this work, we present a parallel rendering approach for interactively visualizing gigabyte-scale complex models on a commodity desktop workstation. Recently, the capability of CPU main memory easily reaches tens of gigabytes that can hold billions of polygons. In contrast, GPU memory is still insufficient to store large size data. For example, the Boeing 777 model, shown in Figure 1(a-b), consumes more than 6 gigabytes to store only its vertices and triangles, which is over the memory limits of most modern GPUs. Our GPU out-of-core method takes the advantages of frame-to-frame coherence, so that only the necessary portions of data are selected and steamed to the GPU at each frame.

Our approach tightly integrates the LOD algorithms, occlusion culling and out-of-core towards GPU parallel architectures. To our knowledge, our work is the first attempt to integrate them for gigabyte-scale complex model rendering. Our contributions can be broken down to the following key features:

1. **Massive parallel LOD algorithm.** We generate an adaptively simplified model at run-time with an object-level parallel LOD selection and a triangle-level parallel geometric reformation.
2. **Parallel occluder selection & conservatively culling.** We perform the parallel algorithm of conservative occlusion culling with a novel metric of dynamic occluder selection. As a result, a significant number of hidden objects can be removed efficiently.

The rest of the paper is organized as follows. In Section 2, we briefly review some related works. Section 3 gives an overview of our approach. We describe the stat-of-art of LOD algorithms in Section 4. Our contribution to the LOD selection is presented in Section 5, and the contribution to the occlusion culling in Section 6. In Section 7, we describe the out-of-core methods for CPU-GPU data streaming. Our implementation and experimental results are shown in Section 8. We conclude our work in Section 9.

2 Related Works

In this section, we discuss some previous works that focus on mesh simplification, occlusion culling and their integrations.

2.1 Mesh Simplification

The techniques of mesh simplification have been studied for a few decades. Traditional simplification algorithms were based on a sequence of topological modifications, such as collapsing edges [Hoppe 1997; Garland and Heckbert 1997] and remeshing with geometry images [Gu et al. 2002; Sander et al. 2003].

Recently, GPU parallel architectures caught researchers' attention to speed up run-time computations for mesh simplification. The researchers focused on how to eliminate the data dependencies introduced in traditional algorithms. [DeCoro and Tatarchuk 2007] presented a vertex clustering method using the shader-based fixed GPU graphics pipeline. The representative vertex position for each cluster can be independently computed through geometry shader stage. [Hu et al. 2009] introduced a GPU-based approach for view-dependent Progressive Meshes, where vertex dependencies were not fully considered during a cascade of vertex splitting events. [Derzapf et al. 2010] encoded the dependency information of Progressive Meshes into a GPU-friendly compact data structure. [Peng and Cao 2012] eliminated the dependency by simply using an array structure and supported triangle-level parallelism. They also claimed that the limited GPU memory is an important issue for simplification, so that they integrated their parallel simplification method and GPU out-of-core to render large and complex models.

2.2 Occlusion Culling

Occlusion culling, one of the important topics in visibility research, aims to remove the hidden objects (the objects obscured by others) from the rendering process. Hierarchical Z-buffer (HZB) introduced in [Greene et al. 1993] is an algorithm that uses an image-space Z pyramid to quickly reject the hidden objects. [Hudson et al. 1997] used the shadow frusta to accelerate occlusion culling in object space. [Zhang et al. 1997] presented a culling approach mixed from a object-space bounding volume hierarchy and a image-space hierarchy occlusion map.

GPU-based parallel designs have also been presented. [Morein 2000] accelerated HZB with hardware fixed functions. [Klosowski and Silva 2001] designed a conservative culling algorithm based on the Prioritized-Layered Projection (PLP) with the support of OpenGL extensions. [Govindaraju et al. 2003] used a dual-GPU occlusion-switch approach to overcome the performance issues. But additional latency was introduced when exchanging data between GPUs. [Bittner et al. 2004] proposed an optimized version of hardware occlusion queries to improve the culling efficiency and performance.

One important research question of occlusion culling is how to efficiently and dynamically find the optimal occluders that can reject the hidden objects as many as possible. This problem was not satisfactorily solved by previous culling approaches. Also, although parallel solutions have been proposed in previous works, they targeted only culling-related computation.

2.3 Integration

The Integration of mesh simplification and occlusion culling is necessary for rendering complex models. [El-Sana et al. 2001] used the View-Dependence Tree to integrate them in a simple and intuitive fashion. Similarly, [Andújar et al. 2000] provided the Visibility Octree to estimate the degree of visibility of each object, which was also contributed to the LOD selection in the integration. [Yoon et al. 2003] decomposed the 3D scene into a cluster hierarchy, where the simplification algorithm was applied to the set of visible clusters. The culling accuracy was dependent on the temporal coherence. Later, [Yoon et al. 2004] improved their cluster-based integration approach for out-of-core rendering massive models. [Gobbetti and Marton 2005] represented the data with a volume hierarchy, by which their approach tightly integrated LOD, culling and out-of-core data management for massive model rendering.

These integrated approaches rely on the hierarchical data representations. The hierarchy is traversed in a top-down manner to perform the operations for both LOD and culling. But for parallel designs, the dependency between levels of the hierarchy have to be eliminated before an efficient GPU algorithm can be carried on. In this paper, we believe that the integration of simplification and culling on GPU parallel architecture is an important step for the next generation of rendering systems, especially for the systems that interactively visualize complex 3D models.

3 Overview

In a general case, a complex 3D environment model consists of many disconnected objects that are unorganized, irregularly shaped and interweaving detailed in 3D space. In order to efficiently render the model, our approach conducts two computing stages, data preprocessing and run-time processing, as illustrated in Figure 2.

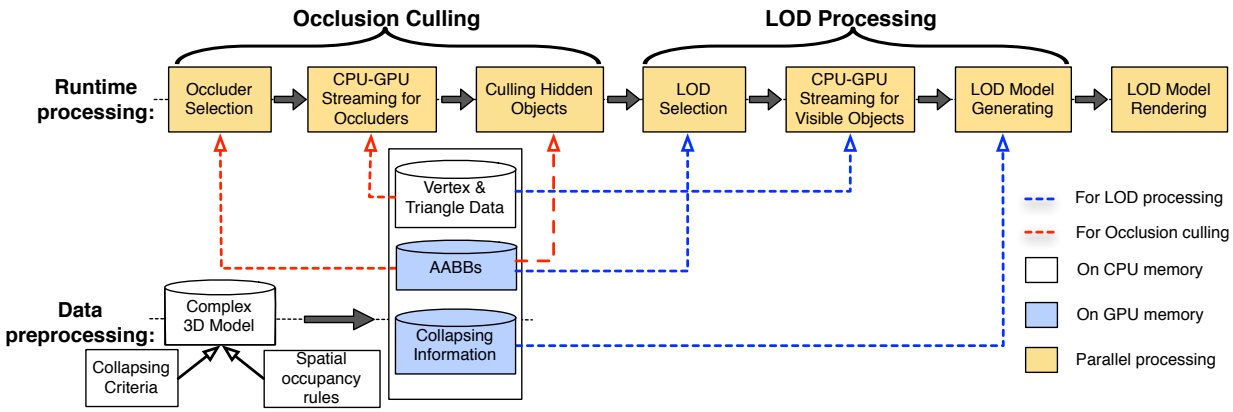


Figure 2: The overview of our approach.

In the pre-processing stage, we perform a sequence of edge-collapsing operations to simplify the input model. The order of these operations is used to re-arrange the storage of vertices and triangles. Also, to better prepare for occlusion culling, we examine the qualification of each object to be an occluder by evaluating its spatial occupancy. We also generate Axis-Aligned Bounding Boxes (AABBs) of the objects.

At run-time, two computation components, occlusion culling and LOD processing, are performed based upon a series of parallel processing steps. we select a set of adaptive occluders, transfer them to GPU memory space, and rasterize them into a Z-depth image. The objects hidden behind the occluders are then eliminated by testing them against the depth image. After that, the remaining objects are passed through the component of LOD processing, where each object's geometric complexity is determined to be used for reforming the object into a new shape. At the end, the reformed objects are rendered with OpenGL Vertex Buffer Objects (VBO).

4 State-of-Art in Parallel Mesh Simplification

Traditional approaches rely on hierarchical data representations for complex model simplification, such as [Yoon et al. 2004; Cignoni et al. 2004]. A hierarchy is usually built in a bottom-up node-merging manner. However, the major limitation of a hierarchical representation is that inter-dependency is introduced between levels of the hierarchy, which would not be suitable for data parallel computing. Most recently, [Peng and Cao 2012] built dependency-free data representations that allow triangle-level parallelism to generate LOD models on GPU. Our simplification method is extended from Peng and Cao's work, and we would like to give more details of their work in the rest of this section.

Edge-collapsing is the basic operation in the process of simplification. Edges are collapsed iteratively; at each iteration, two vertices of an edge are merged, and the corresponding triangles are eliminated, and consequently mesh topology is modified. Because each object of the model maintains its own mesh topology without connection to any others, it can be simplified independently. In order for a faithful look on a low-poly object, rules for choosing an optimal edge at an iteration have been well studied, such as [Melax 1998; Garland and Heckbert 1998], to make sure the visual changes are minimized when collapsed.

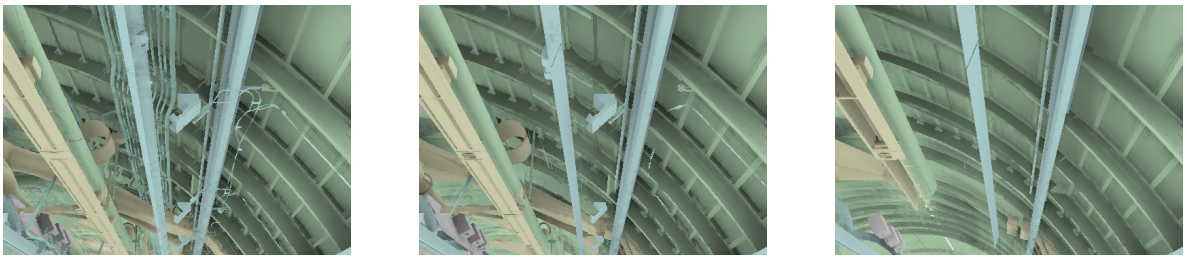
The collapsing operations indicate how the details of a 3D object are reduced. These operations are recorded in an array structure. Each element in the array corresponds to a vertex, and its value is the index of the target vertex that it merges to. According to the order of collapsing operations, Storage of vertices and triangles are re-arranged. In practice, the first removed vertex during collapsing is re-stored to the last position in the vertex data set; and the last removed vertex is re-stored to the first position. The same re-arrangement is applied to the triangle data set as well. As a result, the order of storing re-arranged data reflects the levels of details. If needing a coarse version of the model, a small number of continuous vertices and triangles are sufficient by selecting them starting from the first element in the sets.

Since the size of GPU memory is usually not large enough to hold the complex model, the run-time selected portion of vertices and triangles has to be streamed to GPU from CPU main memory. Then, each GPU thread is asked to reshape one triangle, where its three vertex indices are replaced with appropriate target ones by walking backward through the pre-recored collapsing operation array.

5 LOD Selection

Now, the question is how to determine the desired level of detail (or the desired geometric complexity) of the objects at a given viewpoint. This problem is known as *LOD Selection*. Conceptually, an object can be rendered at any level of detail. But because GPU memory is limited, the total number of polygon primitives must be budgeted based on the memory capability. Also, allocating this total number to the objects needs to be well-considered, so that both memory constraints and visual requirements can be satisfied.

Ideally, an object far away from the viewpoint deserves a lower lever of detail. In practice, people usually examine the size of the screen region occupied by an object. The larger size of the region, the higher level the object's detail should be. Based on this idea, for multi-object models, [Funkhouser and Séquin 1993] solved the *LOD Selection* as a discrete optimization problem. Nowadays, the complexity of a gigabyte-scale 3D environment has been significantly increased. The objects in the 3D environment may have dramatically different shapes with widely varying spatial ratios. It is possible that a far-away object, which deserves a lower level of detail, has a much larger projected area than a closer object. In addition, objects usually come with different number of triangles due to the nature of original design. The object that has more triangles should rationally be approximated with a higher detail level than those that have fewer, though the former one may be farther away from the viewpoint. By considering all these aspects, We re-evaluate the *LOD Selection* and provide a closed-form expression to solve it cheaply and reasonably. Below, we explain our solution in detail.



(a) (b) (c)

Figure 3: A LOD example of the Boeing 777 model (originally 332M triangles). *The number of triangles and vertices in each scene (*triangle / vertex*) is:* **(a)** $4.35M/3.18M$*;* **(b)** $1.01M/0.83M$*;* **(c)** $0.42M/0.35M$*, where PET is set to 2 pixels.*

LOD Selection Metric. In a multi-object model, the desired level of detail of the ith object is represented with a pair of vertex count and triangle count. We denote them as vc_i and tc_i, respectively. We compute vc_i using Equation 1.

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

N is the user-defined maximal count. vc_i is computed out of total m objects. A_i denotes the projected area of the object on image plane. To compute A_i efficiently, we estimate it using the area of the screen bounding rectangle of the projected axis-aligned bounding box (AABB) of the object. 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 [Funkhouser and Séquin 1993]. D_i is the shortest Z-depth distance from the corner points of the AABB. P_i is the number of available vertices of the object.

We know that, in the pre-processing step, we record the remaining number of triangles after each collapsing iteration. Thus, at run-time, when vc_i is computed, the corresponding tc_i can be easily retrieved from the pre-recored information. We use NVIDIA CUDA-based Thrust library to implement Equation 1 on GPU.

Pixel Error Threshold. As we know, no matter how large or complex an object is, the object's shape is scan-converted into pixels. At a position long-distance to the viewpoint, a very large object may be projected to a very small region of the screen (e.g. less than one pixel), so that the object might not be captured by people's visual perception. Based on this nature, we introduce a *Pixel Error Threshold* (PET) as a complementary criteria for our metric of *LOD selection*. If A_i in Equation 1 is below a given PET, vc_i is set to zero. By doing this, only the objects large enough on the screen will be allocated a cut from the overall detail budget. This constraint meets the workflow of view-dependent rendering, and removes the objects without losing visual fidelity. In Figure 3, we show the example renderings with our metric of *LOD Selection*.

6 Parallel Occlusion Culling Design

In Equation 1, N is an important factor impacting on the overall performance and visual quality. N tells how many vertices and triangles will be processed by GPU cores. A large value of N results in a heavy GPU workload, and consequently results in a low performance. We can decrease the value of N to ensure a desired performance, but a small N will result in the loss of visual quality.

One way to preserve the visual quality at a given small N is by adding visibility culling algorithms. At a specific viewpoint, many objects are invisible but they obtain the pieces of N. It would be more reasonable that these pieces should be distributed to the visible objects, so that we can increase their detail levels. Generating the simplified versions of invisible objects is a waste of GPU storage and computation resources, and definitely decrease the level of overall visual fidelity.

Hardware occlusion queries available in the 3D API (OpenGL or Direct3D) were popularly used to check whether or not objects are visible. Unfortunately, this feature is not suitable for our application because of three reasons: (1) occlusion queries use static set of occluders, rather than view-dependently selected; (2) because the previous frame depth buffer is used for occlusion testing in the current frame, "flickering" artifacts might occur; (3) occlusion query technique is an independent and fixed function module that is impossible to integrate with any LOD method.

Therefore, in this section, we introduce a novel parallel approach for occlusion culling. Our approach seamlessly integrates with the parallel LOD, and rejects the invisible objects before the LOD budget allocation step.

6.1 Pre-processing

In a model, not all objects are suitable to be occluders. Most existing systems use simple criteria to examine an object's qualification, such as the size of AABB and the number of triangles. But some complex models contain the irregularly shaped objects (e.g. casually curved long wires in the Boeing 777 airplane model), that should not be qualified as occluders at any viewpoint. In our work, we use a spatial occupancy criteria to determine the qualification of an object. Specifically, we measure its compactness in object space. We calculate a tight Object-Orientated Bounding Box(OOBB) for each object. Equation 2 returns the value of compactness that indicates how well the object fills the OOBB's space.

$$compactness = \frac{A_x\frac{P_x}{R_x} + A_y\frac{P_y}{R_y} + A_z\frac{P_z}{R_z}}{A_x + A_y + A_z}. \qquad (2)$$

We use projection-based method to compute the object's compactness. In Equation 2, we denote A to be the orthogonally projected area of OOBB along its local axises; P is the number of pixels occupied by the object on the corresponding projection; R is the total number of pixels occupied by the OOBB on the projection.

The objects are then sorted based on their compactness values. The storage of the objects are re-arranged by moving those with higher compactness to the front. The higher a value is, the better the object is deserved to be an occluder. We use a *Compactness Threshold* (CT) to find out the objects suitable to be occluders. The CT

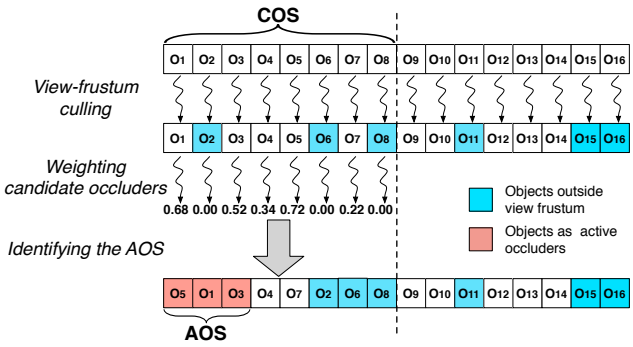


Figure 4: An example of parallel occluder selection. *Assuming that the input model are composed of 18 objects, and 8 of them are classified into the COS. The size of AOS is fixed to be 3 objects.*

defines the lower bound compactness. The objects whose compactness values are above the CT will be added into the *Candidate Occluder Set* (COS).

6.2 Occluder Selection

At run-time, for each frame, a group of occluders from the COS is view-dependently selected. We denote it as *Active Occluder Set* (AOS). Selecting the exact occluders is usually computationally intensive. Thus, we develop an efficient method to estimate the AOS, which will be sufficient and effective for occlusion culling. We perform three steps to decide the AOS: (1) view-frustum culling: we test each object's AABB against the view frustum, so that the object is invisible if its AABB outside the frustum; (2) weighting candidate occluders: we weight the objects in the COS to determine any one suitable to be in the AOS. (3) identifying the AOS: We identify the objects with higher weights and select them into the AOS. An example of these three steps is illustrate in Figure 4.

View-frustum culling. We ask one GPU thread to handle one object. By testing its AABB against the view frustum of the camera, the objects outside the frustum will not be passed into the rendering pipe. They of course will loss the opportunity to be selected into the AOS though they may have high compactness value in the COS. Each object of the model is processed by one GPU computation thread. The reason that we use AABBs instead of OOBBs is because an AABB allows a faster execution and has less memory requirements.

Weighting candidate occluders. The objects in both the COS and the view frustum still hold their candidate status to be active occluders. But we certainly do not want to select all of them to the AOS, because the number of them most-likely is large and many of them may actually be occluded. For example, in the Boeing model, the COS contains 362 thousand objects with the CT equal to 0.63 (50.4% out of totally 718 thousand objects). Hence, we need to select less but optimal candidates from the COS. Our idea is to make sure that the selected objects are spatially large and closed to the viewpoint. We develop a weighting metric to determine the AOS. we pre-assign a direction vector to each object. The vector is perpendicular to the largest face of the object's OOBB. Then the object's weight is computed based on its viewing direction, its distance to the viewpoint and the size of its bounding volume, as shown in Equation 3.

$$weight = \frac{V||\overrightarrow{N} \cdot \overrightarrow{E}||}{D^3} \tag{3}$$

V is the volume size of the object's AABB; $\overrightarrow{N}$ is the direction vector of the object; E is the direction vector of the viewpoint; D is the closest distance between the AABB and the viewpoint position. In general, an occluder will perform the best occlusion when its vector N faces to the viewpoint. In Equation 3, we use the dot product to capture this property. As shown in the second step of Figure 4, weighting the objects of the COS can be executed with a object-level parallelism. If a candidate object is inside the view frustum, its weight will be computed using Equation 3; otherwise, it will be assigned to zero and lose the opportunity to be in the AOS. Note that we always use AABBs during run-time, since they are less of memory requirement for storing the conner points comparing to OOBBs.

Identifying the AOS. In order to quickly identify the optimal candidate objects for the AOS, we sort the COS based on the descending order of the weights of its members. The higher weight the object has, the better this object is added into the AOS. We constrain the AOS to be the fixed size, so that the number of active occluders will not be changed during the entire run-time. Although it is also reasonable to use the alterable number of active occluders, such as by setting the lower bound weight threshold, the size of AOS could be very large. We consider that the objects in the AOS are significantly visible, and we will use their full details for both culling and rendering. If the size of AOS is very large, the active occluders may overburdens the GPU memory capability and the rendering workloads. For instance, the test made by [Aila and Miettinen 2004] shows 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, we use the fixed small number of active occluders as long as it is effective for occlusion culling. Because the COS is sorted, we can identify the AOS at a constant speed by selecting the subset of the COS starting from the beginning element.

6.3 Conservative Culling with Hierarchical Z-Map

Conservative culling is to determine a set of objects that are potentially visible, which is commonly known as *Potentially Visible Set* (PVS). The objects in the AOS definitely belong to the PVS. For all other objects, we determine whether or not they belong to PVS by checking whether or not they are obscured by the AOS. To do this, we build the hierarchical Z-map (HZM) from the depth image of the active occluders.

Similar to the methods used in [Greene et al. 1993; Zhang et al. 1997], the HZM is constructed by recursively down-sampling the fine-grained depth image in an octree manner. Each level of HZM is an intermediate depth image, which is down-sampled from the one-level higher image by merging its 2×2 blocks of pixels. For example, if the original image dimensions are 512×512, the number of levels is $log_2 512 + 1 = 10$. The highest level image is the original one; and the lowest level image contains only one pixel.

At each building step, all 2×2 pixel blocks are operated in parallel; and each block is merged and replaced by the pixel with the maximal depth value in the block. Note that, the memory space for HZM is allocated on GPU at initialization time. The size of HZM is computed using Equation 5, where W is the dimension of the render frame.

$$\textit{HZM size} = \frac{1-(1/4)^k}{3/4} \times W^2, \text{where } k = log_2 W + 1 \tag{4}$$

Once the HZM is constructed, we determine the visibilities of the objects. we use a bounding square of the object's projected area as

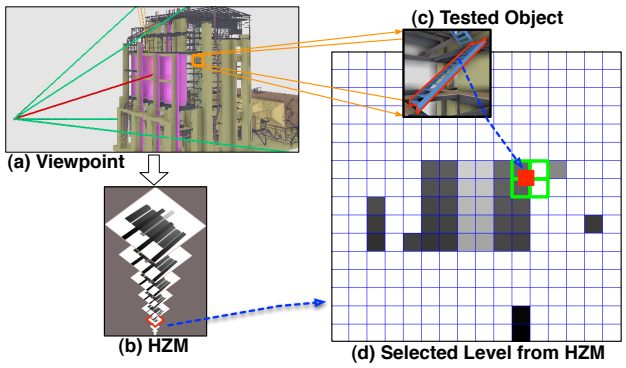


Figure 5: The concept of culling with the HZM. (a) *The viewpoint for the rendering frame. The green lines define the view frustum, and the redline defines the viewing direction from the camera position;* **(b)** *The HZM constructed based on the active occluders (the purple objects in (a)) rendered in 512 × 512 resolution;* **(c)** *The visibility of the object is being tested by using the HZM;* **(d)** *The image represents the selected level in the HZM. The red square represents the projected size of the object on the screen, and it overlaps with the four green blocks, which represent the four depth pixels.*

the approximation to test against an appropriate level of HZM. This appropriate level can be determined by Equation 5, where R represents the dimension of the bounding square in pixels; L represents the total levels of HZM.

$$level = L - log_2 \frac{W}{R}, (R \geq 1) \tag{5}$$

By doing this, the area covered by the square is guaranteed in a range of pixel size of $(1, 4)$ at the chosen level, see Figure 5(d). We unify the depth values in the square region with the AABB's minimal depth value. If the depth value is larger than all of these four pixels, the object is surely occluded; otherwise, it will be labeled as a potentially visible object into PVS.

7 GPU Out-of-Core for Data Streaming

CPU main memory has been well developed to cache large datasets. It has become feasible to allocate tens of gigabytes memory on RAMs in a PC platform. However, the increased size of GPU memory does not catch up the same storage capability. Thus, run-time CPU-GPU data streaming is unavoidable for rendering complex models.

Meanwhile, the low bandwidth of most modern GPUs is a major drawback that will significantly slow down the overall performance if a large amount of data needs to be transferred. Although the occlusion culling and LOD processing can largely reduce the amount of data required by GPU, the remaining data is usually still too large to be streamed efficiently.

Our streaming method is based on the coherence-based GPU Out-of-Core presented in [Peng and Cao 2012]. We conduct two phases of streaming: streaming the AOS and streaming the PVS. To perform occlusion culling, the AOS are streamed to GPU for the depth-buffer rendering. We do not simplify the objects in the AOS, since we consider them as the most significantly visible objects. After finishing the culling stage, we simplify the objects in the PVS so that they will not require too much GPU memory. Then, by utilizing frame-to-frame coherence, the frame-different data need to be identified on CPU for the preparation of streaming. However, identifying frame-different data requires to check the objects sequentially. To reduce the cost on the sequential execution, we utilize a CPU multithreading method to evenly distribute the workload among available CPU cores.

Table 1: Parameter Configurations.

Model	α	N	Pixel Error Threshold (PET)	Compactness Threshold (CT)	Size of the AOS
Boeing 777	3.0	12.2M	1 pixel	0.55	20
Power Plant	3.0	3.5M	1 pixel	0.88	15

Table 2: The results of preprocessing. *Two models are preprocessed on a single PC.*

Model	Data File		Collapsing Simp.		Occupancy Comp.	
	Tri/Vert Num.	Size	*ECol* Size	Time	Size	Time
Boeing 777	332M / 223M	6.7GB	582.5MB	952min	2.9MB	38min
Power Plant	12M / 6M	0.5GB	14.2 MB	40min	0.6MB	5min

8 Experimental Results

We have implemented our approach on a workstation equipped with an AMD Phenom X6 1100T 3.30GHz CPU (6 cores), 8 GBytes of RAM and a Nvidia Quadro 5000 graphics card with 2.5 GBytes of GDDR5 device memory. Our approach is developed using C++, Nvidia CUDA Toolkit v4.2 and OpenGL on a 64-bit Windows system. We have tested two complex 3D models for our experiments. One is the *Boeing 777 airplane model* composed of about 332 million triangles and 223 million vertices. Another is the *Coal Fired Power Plant model* composed of about 12 million triangles and 6 million vertices. We have maintained all AABBs and edge-collapsing information ($ECol$s) on GPU memory during run-time, which consume about 17 MBytes and 447 MBytes memory for Boeing 777 model, respectively. Since Power Plant model is relatively small and can be fit into the GPU memory, we can completely avoid the step of CPU-GPU streaming. The parameter configurations used in our system are listed in Table 1. We also used the Screen-Space Ambient Occlusion to provide realistic rendering quality.

8.1 Pre-processing Performance

In our system, the pre-processing includes two parts: simplification for recording the collapsing information and computation of the objects' spatial occupancy. We execute them on a single CPU-core for both test models. The performance results are shown in Table 2. The simplification costs more time than the computation of spatial occupancy. On average, the throughput performance of our pre-processing method is 5K triangles/sec. Comparing to other approaches, [Yoon et al. 2004] computed the CHPM structure at 3K triangles/sec; [Cignoni et al. 2004] constructed the multiresolution of the static LODs at 30k triangles/sec on a network of 16 CPUs; [Gobbetti and Marton 2005] built their volumetric structure at 1K triangles/sec on a single CPU. Our method is at least 66.7% faster in single CPU execution. In terms of memory complexity, we have generated the addition data which is only 8.7% and 3.0% of the original data sizes for the Boeing model and the Power Plant model, respectively.

8.2 Run-time Performance

We created the camera navigation paths to evaluate run-time performance. The movements of the cameras along the paths are

Table 3: Run-time Performance. *We show the average breakdown of frame time for the two benchmarks. The results are averaged over the total number of frames in the created paths of each model. The models are rendered in 512×512 resolutions.*

Model	FPS	Occlusion Culling	LOD Processing	Rendering
Boeing 777	14.5	7.9 ms (11.4%)	34.6 ms (50.1%)	26.5 ms (38.4%)
Power Plant	78.3	4.8 ms (37.5%)	2.9 ms (22.7%)	5.1 ms (39.8%)

Table 4: Effectiveness of Occlusion culling (OC) over the Boeing 777 model.

Total Object Num.	Objects culled by OC		OC Accuracy (Our Approach)	Memory Released
	Our Approach	Exact		
718K	63K (8.8%)	108K (15.0%)	58.3%	348.6MB

demonstrated in our complementary video. The results show that we can render at the interactive rates of 8-35 fps for the Boeing 777 model (Figure 1(a-b)) and 27-234 fps for the Power Plant model (Figure 1(c-d)). In Table 3, we show the breakdown of the averaged computation times of all three run-time components. For the Boeing model rendering, the computation time spent on GPU Out-of-Core has been added to components. Although the frame-to-frame coherence has been applied in LOD processing, the high cost of CPU-GPU communication still make LOD processing to be the major performance bottleneck because of the large size of the transferred data. On the other hand, because the size of the AOS is small, transferring the active occluders is not expensive, and the HZM construction on GPU is also very efficient. We use OpenGL Vertex Buffer Objects (VBOs) for rendering, which is not the performance bottleneck, even rendering more than 10 million triangles and vertices.

Comparison with previous approaches. Our run-time method can reach an average throughput of 110M triangles/sec. In contrast, [Cignoni et al. 2004] performed an average of 70M triangles/sec using their TetraPuzzle method. [Gobbetti and Marton 2005] sustained an average of 45M voxels/sec with their Fast Voxel method. Thus, we gain the advantages of using GPUs in terms of the performance of processing triangles. We also compare the run-time performance of our approach to the approach of [Peng and Cao 2012]. We set the value of N and the camera movements to be the same in both of the approaches. Figure 7 plots the frame rates over 350 frames. Although we use the CPU multithreading to reduce the cost of out-of-core in the LOD processing, we pay the extra computational cost of occlusion culling, which make the frame rates of our approach not significantly increased.

Effectiveness of occlusion culling. We evaluate the effectiveness of our occlusion culling method by comparing it to the exact occlusion culling. Table 4 shows that our heuristic method cannot deliver the result identical to the result of the exact occlusion culling, but our culling method still achieved a fairly high quality. "Memory Released" indicates the memory was occupied by the hidden objects, and now is released by our occlusion culling method for rendering the visible objects. Figure 6 demonstrates the results of occlusion culling for the Boeing model.

Performance of LOD processing. In the test with Boeing model, an average of 0.02% of the original triangles is transferred, which is 0.68% of the LOD selected triangles. We also discover that the time of the LOD processing scales linearly over the numbers of selected triangles and vertices. The triangle reformation process is very efficient, it can reform one million triangles at 1.2 milliseconds in average.

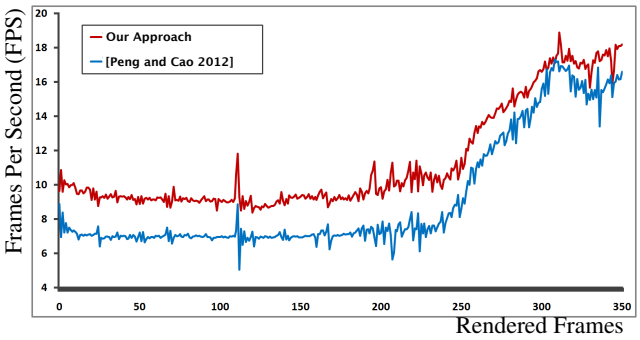


Figure 7: The run-time performance comparison.

9 Conclusions and Future Work

We have presented a GPU-based parallel approach that seamlessly integrated LOD, occlusion culling and GPU out-of-core for rendering Gigabyte-scale complex 3D models at highly interactive frame rates.

There are several aspects to strengthen our work in the future. Rendering quality is sensitive to the metrics used for LOD selection and occluder selection. We would like to explore other metrics that can deliver better performance and rendering qualities. We also would like to improve the current streaming method. We believe that, with a few changes, our approach will be able to be applied for rendering 3D-scanned surface models.

References

AILA, T., AND MIETTINEN, V. 2004. dpvs: An occlusion culling system for massive dynamic environments. *IEEE Comput. Graph. Appl. 24* (March), 86–97.

ANDÚJAR, C., SAONA-VÁZQUEZ, C., NAVAZO, I., AND BRUNET, P. 2000. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum 19*, 3, 499–506.

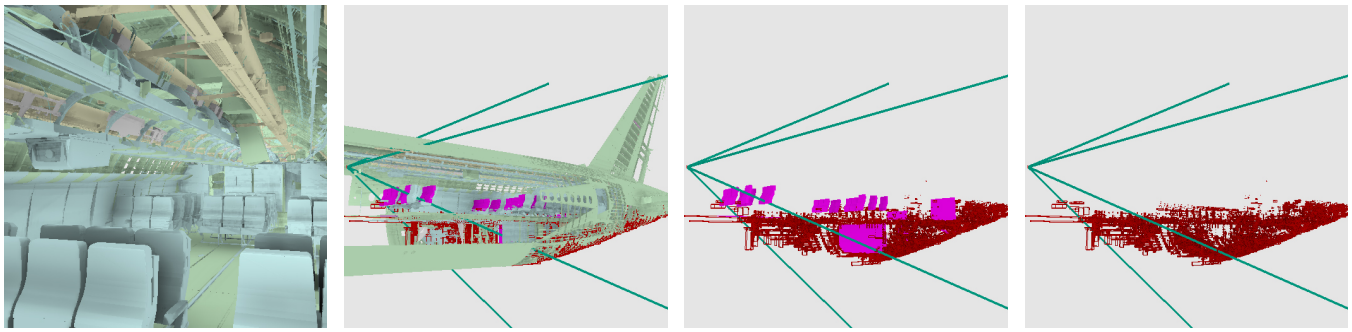
BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3, 615–624.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, SIGGRAPH '04, 796–803.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '07, 161–166.

DERZAPF, E., MENZEL, N., AND GUTHE, M. 2010. Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization*, 53–62.

EL-SANA, J., SOKOLOVSKY, N., AND SILVA, C. T. 2001. Integrating occlusion culling with view-dependent rendering. In *Proceedings of the conference on Visualization '01*, IEEE Computer Society, Washington, DC, USA, VIS '01, 371–378.



(a) (b) (c) (d)

Figure 6: The occlusion culling result on Boeing 777 model. (a) *The rendered frame.* **(b)** *The reference view. The dark green line indicates the view frustum. The objects marked purple are the active occluders. The red boxes represent the occluded objects.* **(c)** *The view showing the active occluders and the occluded objects.* **(d)** *The view showing the occluded objects only.*

FUNKHOUSER, T. A., AND SÉQUIN, C. H. 1993. 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*, ACM, New York, NY, USA, SIGGRAPH '93, 247–254.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 209–216.

GARLAND, M., AND HECKBERT, P. 1998. Simplifying surfaces with color and texture using quadric error metrics. In *Ninth IEEE Visualization(VIS '98)*, pp.264.

GOBBETTI, E., AND MARTON, F. 2005. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 878–885.

GOVINDARAJU, N. K., SUD, A., YOON, S.-E., AND MANOCHA, D. 2003. Interactive visibility culling in complex environments using occlusion-switches. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, I3D '03, 103–112.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '93, 231–238.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3 (July), 355–361.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 189–198.

HU, L., SANDER, P. V., AND HOPPE, H. 2009. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 169–176.

HUDSON, T., MANOCHA, D., COHEN, J., LIN, M., HOFF, K., AND ZHANG, H. 1997. Accelerated occlusion culling using shadow frusta. In *Proceedings of the thirteenth annual symposium on Computational geometry*, ACM, New York, NY, USA, SCG '97, 1–10.

KLOSOWSKI, J. T., AND SILVA, C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7* (October), 365–379.

MELAX, S. 1998. A simple, fast, and effective polygon reduction algorithm. In *Game Developer*, 44–49.

MOREIN, S. 2000. Ati radeon hyperz technology. In *In Graphics Hardware 2000*.

PENG, C., AND CAO, Y. 2012. A gpu-based approach for massive model rendering with frame-to-frame coherence. *Computer Graphics Forum 31*, 2.

PENG, C., PARK, S., CAO, Y., AND TIAN, J. 2011. A real-time system for crowd rendering: Parallel lod and texture-preserving approach on gpu. In *Motion in Games*, J. Allbeck and P. Faloutsos, Eds., vol. 7060 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 27–38.

SANDER, P. V., WOOD, Z. J., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SGP '03, 146–155.

YOON, S.-E., SALOMON, B., AND MANOCHA, D. 2003. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, IEEE Computer Society, Washington, DC, USA, VIS '03, 22–.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-vdr: Interactive view-dependent rendering of massive models. In *Proceedings of the conference on Visualization '04*, IEEE Computer Society, Washington, DC, USA, VIS '04, 131–138.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, III, K. E. 1997. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 77–88.