DOI: 10.1111/j.1467-8659.2012.03018.x
EUROGRAPHICS 2012 / P. Cignoni, T. Ertl
(Guest Editors)

Volume 31 (2012), Number 2

A GPU-based Approach for Massive Model Rendering with Frame-to-Frame Coherence

Chao Peng[1] and Yong Cao[1]

[1]Department of Computer Science, Virginia Tech, USA

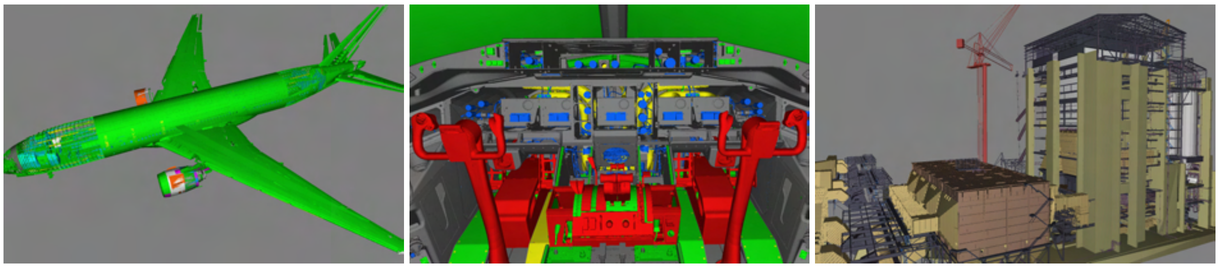
(i) **(ii)** **(iii)**

Figure 1: *Models rendered in our system:* **(i)** *Boeing 777 model;* **(ii)** *The pilot room of Boeing 777;* **(iii)** *Power Plant model.*

Abstract

Rendering massive 3D models in real-time has long been recognized as a very challenging problem because of the limited computational power and memory space available in a workstation. Most existing rendering techniques, especially level of detail (LOD) processing, have suffered from their sequential execution natures. We present a GPU-based approach which enables interactive rendering of large 3D models with hundreds of millions of triangles. Our work contributes to the massive rendering research in two ways. First, we present a simple and efficient mesh simplification algorithm towards GPU architecture. Second, we propose a novel GPU out-of-core approach that adopts a frame-to-frame coherence scheme in order to minimize the high communication cost between CPU and GPU. Our results show that the parallel algorithm of mesh simplification and the GPU out-of-core approach significantly improve the overall rendering performance.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms.

1. Introduction

Rendering large-scale massive models has become a commonly requested task for scientific simulation, visualization and computer graphics. Many research areas generate extremely complex 3D models, such as industrial CAD models (e.g. airplanes, ships and architectures), composed of more than hundreds of millions of geometric primitives. However, these complex datasets cannot be rendered efficiently using brute-force methods on a desktop workstation. Thus, the challenge is how to increase the rendering performance, so that people can visualize massive models interactively. To solve this problem, mesh simplification techniques have been commonly used to reduce the complexity of 3D models without losing visual fidelity.

The algorithms of mesh simplification replace tessellated objects with coarser representations containing less amount of primitives, such as Levels of Detail (LOD). Hoppe [Hop96] introduced a well-known LOD-based algorithm, *Progressive Meshes*, to simplify meshes using a sequence of modifications (e.g. edge-collapsing). However, given a massive 3D model, constructing its simplified representations can be a very expensive process, which makes online simplification impossible on a desktop workstation.

In recent years, graphics hardware, as a massively parallel architecture and commoditized computing platform, has been praised due to the significant improvements of performance and the capability for general-purpose computation. Since most simplification algorithms are not naturally data parallel, they do not have trivial GPU implementations. In addition, comparing to the computational power of GPU, GPU memory is insufficient to store massive datasets. For example, Boeing 777 model shown in Figure 1 requires approximately 6 GByte memory to hold its vertex and triangle data, which is not applicable for most modern GPUs. Although primitives can be directly streamed for rendering, the cost of CPU-GPU communication could decrease performance significantly, if a large number of primitives need to be transferred constantly from CPU to GPU.

To address these issues, we introduce two contributions in this paper.

1. **Parallel mesh simplification.** We present a parallel approach of mesh simplification to generate simplified representations of an input model dynamically.
2. **GPU out-of-core.** We propose a novel out-of-core approach on GPUs that minimizes the overhead of data streaming from CPU to GPU by exploiting the frame-to-frame coherence.

The rest of the paper are organized as follows. We review some related works in Section 2. Section 3 provides a brief overview of data pre-processing and run-time algorithms. In Section 4, we talk about the parallel approach of mesh simplification. In Section 5, we present the GPU out-of-core approach. Section 6 describes our implementation and shows the experimental results. Finally, Section 7 concludes our work and discusses some future works.

2. Related Work

Interactively rendering massive 3D models is an active research area. We discuss the related works with respect to mesh simplification and out-of-core techniques.

2.1. Mesh Simplification

Mesh simplification algorithms have been an active research for decades in computer graphics. Given an input 3D model, a less complicated but visually faithful representation can be approximated as an alternative for rendering. Current algorithms of mesh simplification have been designed according to a series of operations on geometric primitives, such as vertex decimation [SZL92], edge collapse transformations [Hop96, GH97], Region-Merging Measurement [RRR96] and selective refinement [Hop97]. A well-known approach of mesh simplification is Progressive Meshes [Hop96] simplifying meshes with edge-collapsing criteria, then different levels of detail of meshes can be recovered by applying a prefix of splitting sequence to the base mesh.

Related to this paper, we emphasize on the techniques of GPU-based mesh simplification. [JWLL06] generated a LOD mesh on GPU using a quad-tree structure constructed from poly-cube maps. In their techniques, an adaptive mesh is finalized in vertex shader. In [DT07], the authors used a vertex-clustering method, and designed a GPU-friendly octree structure for efficient LOD generation. Although their clustering method reduced the memory storage, the visual quality was not well preserved. More recently, Hu et al. [HSH09] proposed a parallel algorithm for view-dependent LOD on GPU. The authors introduced a cascaded update method to split vertices without respecting their dependencies. However, their approach did not demonstrate the rendering efficiency of very complex 3D models. In this paper, we tend to render large-scale 3D model that even cannot fit into GPU memory. Our system will interactively render the model with the approaches of parallel LOD generation and coherence-based GPU streaming.

2.2. Out-of-Core Techniques

Various out-of-core techniques have been proposed to solve the problem of huge amounts of static LODs, multi-resolution geometries or multi-level vertex hierarchies constructed for interactively rendering massive models, such as [CGG*03, IG03, CGG*04, YSGM04, GM08]. [CGG*03] used a binary tree for mesh partitioning, and allowed the construction of multi-resolution and per-node simplification. [CGG*04] used a geometry-based multi-resolution structure for out-of-core data management. The approach constructed a hierarchy of tetrahedra by recursively partitioning the input dataset. Each tetrahedral node contained a simplified mesh representation (or a patch) precomputed in a fine-to-coarse manner. During the run-time, the hierarchy was top-down traversed to fetch the appropriate patches from disk to CPU. [YSGM04] represented a 3D model as multiple progressive meshes in a clustered hierarchy (CHPM). At run-time, based on a list of fetched clusters, the desired model was generated by performing the refinement operations and cluster merging. However, both approaches of [CGG*04] and [YSGM04] relied on heavy pre-processing stages to build all levels of mesh details or densely clustered PMs, which enlarged the data size dramatically. Although cache-coherent layouts have been used for their out-of-core techniques, it is still a major performance overhead to fetch and access data in a slow bulk memory (e.g. hard drive). In our approach, we will not construct spatially complex structures, so that CPU main memory can hold original meshes sufficiently; and the simplified versions of meshes will be generated dynamically by taking advantage of GPU parallel com-

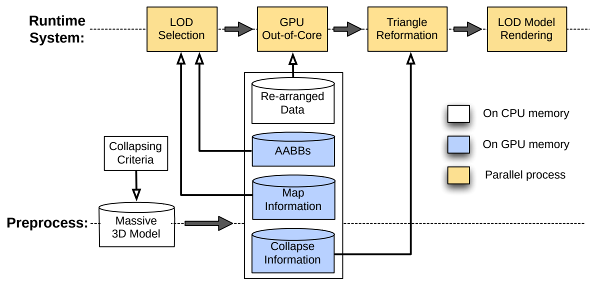

Figure 2: *The overview of our approach.*

putational architectures along with an efficient coherence-based CPU-GPU streaming.

3. Overview

The goal of our system is to simplify complex 3D models for real-time rendering. An input model consists of multiple triangulated meshes. Our approach includes a preprocess stage and a run-time stage, which are illustrated in Figure 2.

In the preprocess stage, we generate a new representation of the original model by re-arranging the vertices and triangle information based on edge-collapsing operations. We also compute other data, such as bounding boxes for the meshes, to be used in the run-time. Our run-time approach contains a series of parallel processing steps. We first determine the complexity of the model in the step of level-of-detail(LOD) selection. Second, the data is streamed from CPU main memory to GPU memory using our GPU out-of-core approach. We employ a frame-to-frame coherence scheme to minimize the size of the streamed data. Third, the meshes are simplified in parallel on GPU at the step of triangle reformation. Finally, the simplified model is rendered using OpenGL.

4. Parallel Mesh Simplification

Our simplification method is based on the idea of collapsing edges of a mesh, where the edge-collapsing operations are applied to an original mesh iteratively according to a pre-defined order. Therefore, dependencies have to be introduced between the iterations of collapsing. At an edge-collapsing iteration, an edge is removed based on a specified cost, and the topological structure of the mesh is modified. The next iteration has to rely on the resulted mesh of the previous iteration. Such dependencies make the design of a parallel algorithm very difficult. In this section, we will introduce an approach that can remove the dependencies effectively to support the parallel process of mesh simplification.

4.1. GPU Friendly Pre-processing

At each edge-collapsing iteration, an edge (v_1, v_2) is removed. There are two steps involved during the removing process: (1) collapsing the edge by merging the two vertices, v_1 and v_2, to a target vertex, $\bar{v}$; (2) removing v_1, v_2 and all triangles having both vertices.

In order to eliminate the dependencies between the iterations, we process the mesh and record the collapsing information into an array structure, called *ecol*. Similar to the data structure described in [Swa99], each element of *ecol* corresponds to a source vertex, and its value is the target vertex that it merges to. We define that the value of the *ith* element of *ecol* can be recovered from the function $ecol(i)$. Meanwhile, we also record the vertex count and triangle count remaining after each collapsing iteration. Since an edge-collapsing operation removes only one vertex but varied number of triangles, we employ a structure, named as *map*, to record the relationship between the vertex count and the triangle count. If i is the remaining vertex count after a collapsing operation, the value, recovered from the function $map(i)$, is the triangle count of currently simplified mesh.

According to the order of edge-collapsing operations, we re-arrange the vertex and triangle data of the original mesh. The order of storing re-arranged data reflects the sequential order of edge-collapsing operations. In our implementation, the vertex and triangle data of a mesh are stored as arrays; the first removed vertex by the collapsing operations is stored at the last position in the vertex array; and the last removed vertex is stored at the first position. The triangle array is also re-arranged in the same manner. With such a representation, a level of detail of the mesh can be simply determined by using a certain number of vertices and triangles starting from the beginning of the vertex and triangle array, respectively. The smaller amount of data is selected, the lower level of detail the mesh is represented in.

4.2. Key Criteria of Edge-collapsing

Position of target vertex. To collapse an edge (v_1, v_2), two vertices are merged to a target vertex $\bar{v}$. Obviously, the position of $\bar{v}$ can be either edge endpoints v_1, v_2, or a new position (e.g., $\bar{v} = (v_1 + v_2)/2$). Based on [GH98], our approach uses an endpoint for $\bar{v}$, since it requires much less storage.

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

Error function for computing the costs. An error function is used to ensure the visual quality of simplification. The value of the error function, usually defined as the cost, indicates the amount of visual changes after an edge is collapsed. Based on the description in [Mel98], we collapse the minimal-cost edge after computing the costs of all edges using Equation 1.

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

Note that, T_{v_a} is the set of triangles containing vertex v_a, and $T_{v_a v_b}$ is the set of triangles containing both vertices v_a and v_b. In Equation 1, the cost of an edge is affected by both edge length and curvature.

4.3. Level of Detail Selection

The task of LOD selection is to determine a desired complexity of a model at each rendering frame. In our system, the input 3D model is defined as a collection of the meshes, $\{M_1, M_2, \ldots, M_r\}$. For each mesh M_i, we define its complexity level as a tuple $\langle vc_i, tc_i \rangle$, where vc_i is the desired vertex count, and tc_i is the desired triangle count. In addition, we pre-calculate a tight Axis-Aligned Bounding Box (AABB) for each mesh. An AABB serves two purposes for LOD selection: (1) view-frustum culling: the visibility of a mesh is determined by testing its AABB against the view frustum; (2) the complexity level determination: if a mesh is inside the view frustum, we use the projected area (on image plane) of its AABB to compute the desired level of complexity, otherwise, vc_i and tc_i are set to be zero. Figure 3 shows an example of view-frustum culling.

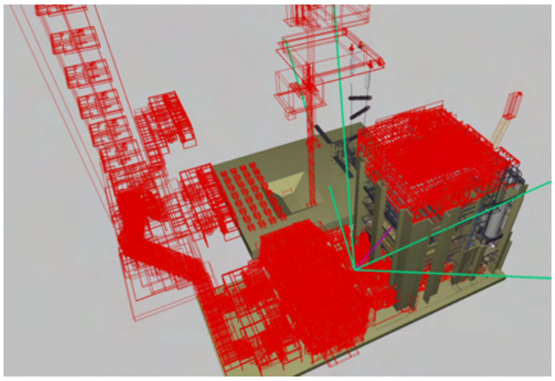

Figure 3: *An example of view-frustum culling. The green lines define the volume of camera view frustum. The red bounding boxes indicate those meshes outside the view frustum by our LOD selection method.*

To determine the complexity level of a visible mesh (after the view-frustum test), we use the projected area of its AABB. In addition, we restrict the total number of the visible vertices using a pre-defined maximum, as discussed in [FS93, WS98]. Therefore, the complexity level of mesh M_i is computed by Equation 2.

$$vc_i = N \frac{A_i^{\frac{1}{\alpha}}}{\sum_{i=1}^{m} A_i^{\frac{1}{\alpha}}}, \quad (2)$$

where N is the pre-defined maximal vertex count, which is wisely chosen based on a desired rendering frame rate or visual quality; A_i denotes the projected area of the AABB on the image plane; α is a parameter to control the perceptive contribution of the mesh to the overall model, introduced in [FS93]. To provide an efficient computation of A_i, we approximate it by using the area of the bounding rectangle of the projected region on the image plane. To have a fast execution, we use CUDA CUDPP [HSO07] to implement Equation 2 on GPU.

Given the value of vc_i calculated in Equation 2, the triangle count tc_i can be obtained based on the *map* structure for mesh M_i , described in Section 4.1, as $tc_i = map_i(vc_i)$.

4.4. Triangle Reformation

Using the computed complexity levels of the meshes, we first select the amounts of vertices (vc_i) and triangles (tc_i) from the original meshes, which will be the active data for generating the simplified version of the input model. Since the vertices and triangles of original meshes have already been re-arranged in the preprocessing step by following the edge-collapsing order, we simply select the first vc_i vertices and first tc_i triangles of mesh M_i. Then, we need to reform each of active triangles of M_i to reflect the changes of its three vertex indices during edge-collapsing by looking up the corresponding *ecol* of M_i, as mentioned in Section 4.1.

GPU parallelization. Obviously, we can parallel the reformation process at mesh-level, e.g. one GPU thread for a mesh. As we know, the design of modern graphics chips allows tens of thousands threads to be executed concurrently. If we choose the mesh-level parallelization, GPU resources will be underutilized when the number of visible meshes is less than the number of concurrent threads. In addition, using the mesh-level parallelization can lead to the load-balancing problem, since different meshes contain different number of triangles after LOD selection. To avoid these issues, we employ a triangle-level parallel approach, e,g. one triangle per thread, so that a sufficient number of GPU threads are created simultaneously, and the workload is also balanced.

Structure of GPU data storage. The natural way of organizing meshes on GPU is storing them separately into different memory blocks, then they can be rendered one-by-one after reformation. Because a 3D model can potentially have many meshes (718,727 meshes in our Boeing 777 model), the overhead of multiple rendering calls to all the meshes is very high. Therefore, we concatenate the array data for all the meshes together into a single array as an OpenGL buffer object, as illustrated in Figure 4-(i), and render the entire model with one rendering call.

Given the GPU parallelization and storage scheme, our parallel triangle reformation approach is described in detail in Algorithm 1. Since we store all the selected (or active) triangles in a single array, when reforming a triangle t_k of this array, we first need to find which mesh it belongs to, so that we can reform it by using the *ecol* of the mesh. To do this, we perform a prefix-sum on the array tc. The prefix-summed tc indicates the offsets (or ranges) of triangles of the meshes. We then use the triangle index, k, to conduct a binary search in tc to find the index of mesh that t_k belongs to. For example, if k falls into a range $(tc_i, tc_{i+1}]$, t_k belongs to mesh M_i. This is the process of line 3 in Algorithm 1.

During the reformation, each of three vertex indices of t_k is replaced with a target one by looking up the *ecol* of M_i. A vertex index may need to be updated multiple times until its value is below the total desired number of vertices, indicated in vc_i. To be consistent with tc, we also prefix-sum the array vc (also required by GPU out-of-core); thus, the desired vertex count is recovered by $vc_{i+1} - vc_i$. The process of triangle reformation is illustrated in Figure 4-(ii) and in the line 5-10 of Algorithm 1. Note that a vertex index of t_k, $vidx$, is a local index in mesh M_i.

Algorithm 1 Triangle Reformation

```
procedure ReformingTriangle(
in active triangles, array $tc$, array $vc$, the list of $ecol$s;
out reformed active triangles)
 1: for $k$th triangle $t_k$ in active triangles in parallel do
 2:     $i \leftarrow 0$;    // the mesh index that $t_k$ belongs to
 3:     binary search array $tc$ return $i$;
 4:     $ecol \leftarrow$ corresponding $ecol$ of mesh $M_i$;
 5:     $n \leftarrow vc_{i+1} - vc_i$;
 6:     for $j = 1$ to 3 do
 7:         $vidx \leftarrow jth$ vertex index of $t_k$;
 8:         while $vidx > n$ do
 9:             $vidx \leftarrow ecol(vidx)$;
10:         end while
11:     end for
12: end for
```

5. GPU Out-of-Core with Frame-to-Frame Coherence

Due to the limited memory, a GPU cannot hold the entire set of a massive 3D model. However, as described in Section 4.1, by using the re-arranged data, a GPU needs only a small portion of the original vertices and triangles to generate the LOD model. At a given frame, after the levels of complexities (the necessary amount of data) have been determined by our LOD selection approach (see Section 4.3),

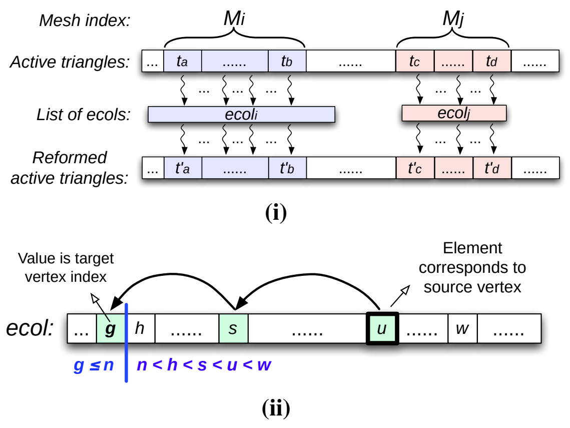

Figure 4: *An example of triangle reformation.* **(i)** *shows the parallel reformation process. The selected triangles are organized in a block of continuous memory, then each triangle is reformed by finding and using its corresponding ecol.* **(ii)** *shows how to replace a vertex with a target one by walking through ecol backwards. n is the amount of selected vertices of M_i, and we find the target index to be g by satisfying $g \leq n$.*

we access the original data stored on CPU main memory, and fetch only those active portions to the GPU memory. Since the overhead for transferring data from CPU to GPU is a significant factor impacting the overall rendering performance, we propose a *GPU out-of-core* approach that transfers much smaller amount of data by exploiting *frame-to-frame coherence*. As such, we can re-use most of existing data on the GPU, which has been devoted to the last rendered frame, so that the overhead of transferring data is minimized.

Our GPU out-of-core algorithm takes the following two steps at each rendered frame:

1. **CPU-GPU data streaming.** In this step, we first need to collect the vertices and triangles not existing on the GPU but required to render the next frame, and store them in a block of continuous CPU memory. Then, we transfer this data block to GPU memory with a single memory transfer call.
2. **GPU data defragmentation.** For the reason of efficient processes in triangle reformation and rendering, the geometry data of all the meshes are concatenated into a single continuous memory block. However, the frame-to-frame coherence approach does not preserve the continuity and the order of geometry data in the GPU memory. To solve this problem, we introduce a parallel defragmentation algorithm to re-organize GPU-ready data for efficiently rendering the 3D model.

5.1. CPU-GPU Data Streaming

In order to minimize the overhead of CPU-GPU communication, we only transfer the additional data that is required

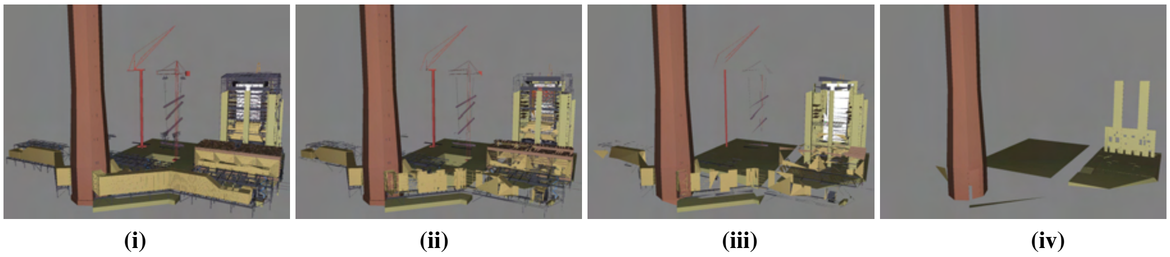

Figure 5: *A sequence of simplified versions of Power Plant model (originally 12 million triangles). The amounts of triangles and vertices in the scene (triangle/vertex) are:* **(i)** $4,193,422/2,199,478$; **(ii)** $91,325/90,466$; **(iii)** $25,091/20,950$; **(iv)** $2,369/1,799$.

in the next frame compared against the currently rendered frame. Let us denote the arrays of vertex counts and triangle counts for the current frame f as vc^f and tc^f, respectively, and for the next frame as vc^{f+1} and tc^{f+1}, respectively. The number of the additional vertices required between frame f and frame $f+1$ is defined as

$$\bar{vc}_i^f = \begin{cases} vc_i^{f+1} - vc_i^f & \text{if } vc_i^{f+1} - vc_i^f > 0, \\ 0 & \text{if } vc_i^{f+1} - vc_i^f \leq 0, \end{cases}$$

where i is the mesh index of the array. The similar definition is applied to the array of triangle counts, $\bar{tc}^f$.

In order to avoid multiple CPU-GPU memory copies, which impose a significant performance cost, we prepare the additional vertices and triangles on the CPU by assembling the data from each mesh into a block of continuous CPU memory, and only copy the block to GPU memory once per frame. To do this, we first perform a prefix sum on the count arrays of the additional vertices and triangles, $\bar{vc}^f$ and $\bar{tc}^f$, respectively, so that we can obtain the position offset for each mesh in the continuous memory block. We then copy the addition data from each mesh into this block at its corresponding position offset. According to the data rearrangement scheme used in the preprocessing step, the addition vertex and triangle data from each mesh is also stored together in a continuous memory space, as illustrated in Figure 6. As such, preparing the additional data on CPU can be efficiently implemented, because the data copy for each mesh will only require a single call of memory copy.

5.2. GPU Data Defragmentation

Between rendering frames, the desired complexity level for a mesh sometimes decreases. In this case, we do not need to transfer any data of the mesh to the GPU. Instead, we need to remove the unnecessary data for this mesh from the GPU, so that we can use the space for other meshes with additional storage requirement. Such operation will make the continuous GPU memory block fragmented. For example, many small and unusable "holes" will be created in the block. In addition, since our parallel triangle reformation approach requires that the geometry data for each mesh has to be stored

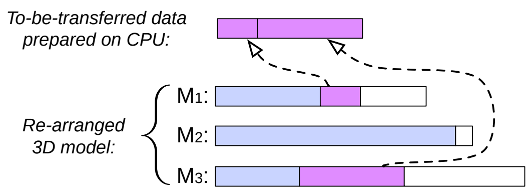

Figure 6: *An example of data preparation on CPU. The purple blocks replicated from the re-arranged meshes stand for the new data required by GPU. The blue blocks are equivalent to those data already existing on GPU.*

in an ordered and continuous fashion, we have to reshuffle the existing data on the GPU and copy the additional data into the right position in the GPU block. The goal of this *data defragmentation* process is to make sure that, (1) the active data selected for the frame is still continuously stored; (2) the vertex and triangle data for each mesh is stored in the same order as it is re-arranged in the preprocessing step; (3) the appearance of each mesh in the block is also stored in the same order as indicated in the arrays vc^{f+1} and tc^{f+1}.

At this step, on GPU memory, we have a block of existing data from the current frame, f, and a block of additional data required by the next frame, $f+1$. To assemble them into the block reserved by active data, a straightforward method is using the system calls of GPU memory copy. For mesh M_i, we copy its vertices and triangles from both existing data block and additional data block to the active data block at the position with the offsets vc_i^{f+1} and tc_i^{f+1}, respectively. However, there will be a large number of system calls of GPU memory copies, and they have to be initiated by the CPU and executed sequentially, which would be a significant cost on performance.

An alternative way of memory copy is to manipulate each element of the block in parallel. On the GPU, it has been shown that it is much more efficient than the direct memory copy when the data size is large. We design a parallel process of *data defragmentation* that each GPU thread only copies the data for one triangle into its required position. Our algorithm, as illustrated in Algorithm 2, defragments the triangle data of all the meshes with one kernel call to the GPU, instead of one call for a mesh, in order to avoid the high cost of multiple GPU calls.

As described in Algorithm 2, each GPU thread copies the data for one triangle, t_k, of the active triangles from either the existing triangles or the additional triangles, as illustrated in Figure 7. First, we identify the mesh index, i, that t_k belongs to, since the source triangle for t_k has to come from the mesh M_i. To find the mesh index quickly, we perform a binary search on the array tc^{f+1}, in the same way as we do in Algorithm 1. Second, we identify if the source triangle for t_k should be an existing one or an additional one of M_i. To do this, we convert the index of t_k in the active triangles to a local triangle index in M_i. We denote this local index as $tidx$ (see line 4 of Algorithm 2). If $tidx$ is smaller than the number of existing triangles, we copy the triangle from the existing ones (see line 6-8); otherwise, we copy from the additional triangles (see line 9-11). At the end, the block of existing triangles is replaced with the completed active triangles, so that we can use it to defragment the following frame.

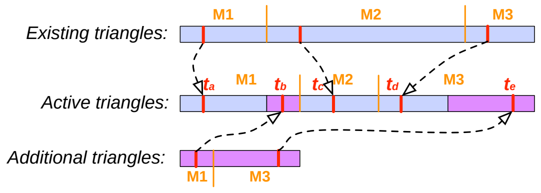

Figure 7: *An example of defragmenting triangles on GPU. t_a, t_b, t_c, t_d and t_e will be substituted with a source triangle from existing or additional block of triangles in parallel.*

Algorithm 2 Defragmenting triangles on GPU

```
procedure TriangleDefragmentation(
in array $tc^{f+1}$, array $tc^f$, array $\bar{tc}^f$, existing triangles, additional triangles;
out active triangles)
 1: for $k$th triangle $t_k$ in active triangles in parallel do
 2:     $i \leftarrow 0$;
 3:     binary search array $tc^{f+1}$ return $i$;
 4:     $tidx \leftarrow k - tc_i^{f+1}$;
 5:     $n \leftarrow tc_{i+1}^f - tc_i^f$;
 6:     if $tidx \leq n$ then
 7:         $j \leftarrow tidx + tc_i^f$;
 8:         $t_k \leftarrow$ $j$th existing triangle;
 9:     else
10:         $j \leftarrow tidx - n + \bar{tc}_i^f$;
11:         $t_k \leftarrow$ $j$th additional triangle;
12:     end if
13: end for
14: replace existing triangles with active triangles for the following frame;
```

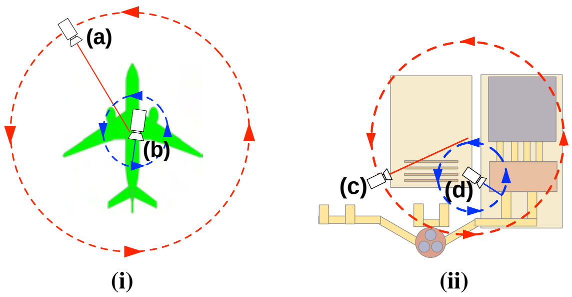

Figure 8: *The setups of Camera movements. **(i)** The path of camera for Boeing Model; **(ii)** The path of camera for Power Plant model.*

6. Experiments and Performance Results

We have implemented our approach on an Intel Core i7 2.67GHz PC with 12 GB of RAM, and a Nvidia Quadro 5000 graphics card with 2.5 GB of GDDR5 device memory. It is developed using Nvidia CUDA Toolkit v4.0, and runs on a 64-bit Windows system. Our approach has been applied to two complex 3D models. One is a Boeing 777 airplane model composed of about 332 million triangles and 223 million vertices. The other one is a coal fired power plant model composed of about 12 million triangles and 6 million vertices.

Since Boeing model requires approximately 6 GB memory space, the data is streamed for rendering based on the GPU approach explained in Section 5. But for Power Plant model, it can fit into GPU memory, so that the cost of CPU-GPU communication is completely eliminated.

We generate two 360-degree camera turning movements for each model (see Figure 8). We run 300 frames for each of four camera setups, and use them to test the performance.

6.1. Overall System Performance

The performance results show that we can achieve interactive rendering rates: 26-226 fps for Power Plant model and 6-22 fps for Boeing 777 model. Figure 9 demonstrates the live-captured images on the path of camera movements. To reach decent visual quality, we have set α of Equation 2 to 3, since [WS98] claimed that $\alpha = 3$ produces equivalent of Funkhouser's benefit function [FS93].

Table 1 shows the breakdown of timing results of the runtime steps, which are the averaged values over 300 frames. For Boeing model, since there is always a considerable number of data being transferred at each frame, GPU out-of-core becomes the most time-consuming part (44.14%(a) and 48.41%(b) out of total time). To understand its importance and efficiency, Section 6.3 provides an insight analysis of the step of GPU out-of-core. And the rendering step never becomes the bottleneck, even with more than 10 million triangles to be rendered.

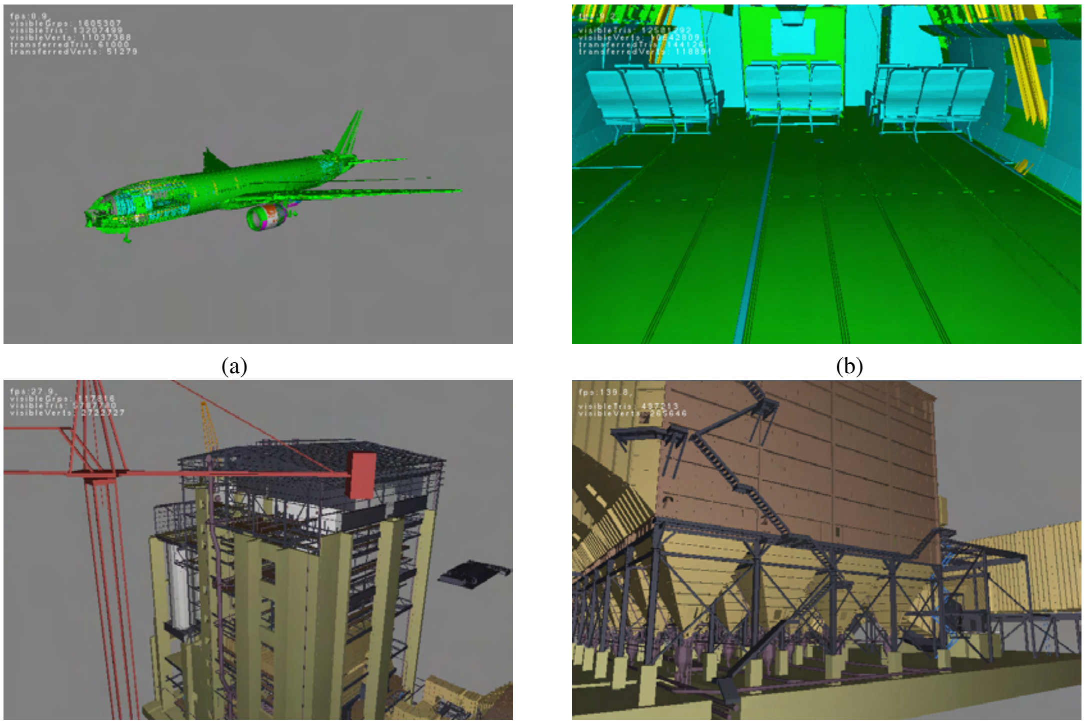

Figure 9: *The rendered 3D models in our experiments. Boeing 777 model is rendered in (a) and (b); Power Plant model is rendered in (c) and (d).*

Table 1: *Overall system performance.*

Model	**Cam.**	**Avg. FPS**	**Avg. LOD Selection**	**Avg. GPU Out-of-Core**	**Avg. Triangle Reformation**	**Avg. Rendering**
Boeing 777	(a)	9.26	11.05 ms (10.23%)	47.67 ms (44.14%)	19.66 ms (18.21%)	29.61 ms (27.42%)
	(b)	9.04	7.34 ms (6.64%)	53.55 ms (48.41%)	18.95 ms (17.13%)	30.78 ms (27.82%)
Power Plant	(c)	92.98	2.74 ms (25.46%)	N/A	4.39 ms (40.80%)	3.63 ms (33.74%)
	(d)	192.45	2.38 ms (45.77%)	N/A	0.77 ms (14.81%)	2.05 ms (39.42%)

Table 2: *Visible triangles and vertices in Power Plant model.*

Model	**Cam.**	**Avg. Visible Triangles / Vertices**
Power Plant	(c)	3.158M / 1.468M
	(d)	0.510M / 0.265M

6.2. Evaluation of Parallel LOD

Our parallel LOD approach reduces the number of triangles and vertices significantly. For example, in Table 2, Experiment (d) has only 0.510 million triangles (in average) to be rendered, which is 4.25% out of the total 12 million triangles; The computation time of LOD selection depends on the number of meshes that a model contains originally, since we have to do view-frustum culling for each mesh. But our triangle reformation method only operates the visible triangles on GPU, so that the time of reformation is scaled with the changes of the visible triangle count. In Figure 10-(i), we scatter the value pairs of reformation time and visible triangle count. Each dot corresponds to a rendered frame, and there are totally 600 dots (frames) on each graph. Usually, the overheads of thread management and data access would prevent the performance to be linear while dealing with large amount of data on the GPU. But with our implementation, Figure 10-(i) shows that the time of reformation increases linearly towards the increase of visible triangle count.

6.3. Evaluation of GPU Out-of-Core

To evaluate our coherence-based GPU out-of-core method, we compare our implementation, *Streaming with Coher-*

ence (SC), with two other approaches: *Streaming without Coherence (SnC)* and *No Streaming (NS)*, which are common brute-force strategies. *Streaming without Coherence* first collects all of them to a continuous CPU memory block, then streams the entire block to GPU with one call. *No Streaming* approach sequentially copies all selected vertices and triangles from CPU memory space to GPU one-by-one. Neither *SnC* nor *NS* approach needs the step of defragmentation. And *NS* approach even has no cost of preparing data on CPU. We show the performance comparisons of these three approaches in Figure 10-(ii). Our coherence-based streaming transfers only new-added vertices and triangles, and has a better overall performance than the other two approaches.

In average, our approach achieves about 1.66X speedup comparing to *SnC* approach, and achieves about 51.96X speedup comparing to *NS* approach. Table 3 shows the averaged timing results and the averaged data amounts of our comparison experiments. Note that *"Avg. Visible Meshes"* means the average number of meshes with non-zero complexity. *"Avg. Streamed Meshes"* means the average number of meshes with the increased complexity, so that some of their vertices and triangles will be CPU-GPU streamed. Our *SC* approach requires much less amounts of "Streamed Meshes" and "Streamed Triangles/Vertices" than the other two approaches, so that much less cost of memory transferring is required by our approach. For example, in camera (a), only 0.68% of total 12.884 million visible triangles and 0.69% of total 10.721million visible vertices are transferred.

Performance factors of CPU-GPU streaming. Based on our experiments, the time spent on CPU-GPU streaming depends on CPU side, because to-be-transferred data is prepared sequentially on CPU, which is the major cost of the streaming. Two Factors influences the time of data preparing on CPU: the size of to-be-transferred data and the number of meshes with increased complexities between frames.

Performance factors of GPU defragmentation. The defragmentation re-organizes the data on GPU. The more data is used for rendering a frame, the more time is required to defragment them. Based on our experimental results from (a) and (b), we notice that the time of GPU defragmentation is scaled with respect to the number of visible triangles and visible vertices determined by our LOD selection method. In Figure 10-(iii), we plot the relationship between defragmentation time and the number of visible data; and each dot represents a frame. It shows that defragmentation time changes linearly over different numbers of triangles and vertices.

7. Conclusion and Future Work

We presented a novel GPU approach to visualize massive 3D models at interactive rates. First, we design a parallel algorithm of mesh simplification that supports real-time generation of LOD model. Second, we propose a GPU out-of-core approach by employing frame-to-frame coherence. A parallel defragmentation algorithm is developed to maintain the data continuity in GPU memory.

Limitations. Our approach assumes the high temporal coherence between frames. If the camera is changed dramatically from one frame to the next, the amount of transferred data based on the frame difference could be increased significantly. As a result, it may lead to a noticeable performance lost. Another limitation of our system is that we require the entire 3D model can fit into CPU main memory.

Future works. There are some future works that can strengthen our approach. First, LOD selection metric is an important factor for managing the amount of selected data and preserving visual fidelities. We would like to explore other metrics applicable for massive model rendering. Second, during the phase of defragmentation, the data used for rendering the previous frame is stored at its own memory allocation. However, it is not the best method to optimize memory usage. In the future, we would like to explore some in-place algorithms to assemble GPU data.

Acknowledgements

This work is partially funded by National Science Foundation, IIS 0940723, titled "EAGER: Drummer Game: A Massive-Interactive Socially-Enabled Strategy Game". We also thank Dave Kasik of Boeing for providing the 3D model of Boeing 777 airplane.

References

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Bdam — batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (2003), 505–514.

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

[DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the gpu. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 161–166.

[FS93] FUNKHOUSER T. A., SÉQUIN C. H.: 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* (New York, NY, USA, 1993), SIGGRAPH '93, ACM, pp. 247–254.

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 209–216.

[GH98] GARLAND M., HECKBERT P.: Simplifying surfaces with color and texture using quadric error metrics. In *Ninth IEEE Visualization(VIS '98)* (1998), p. pp.264.

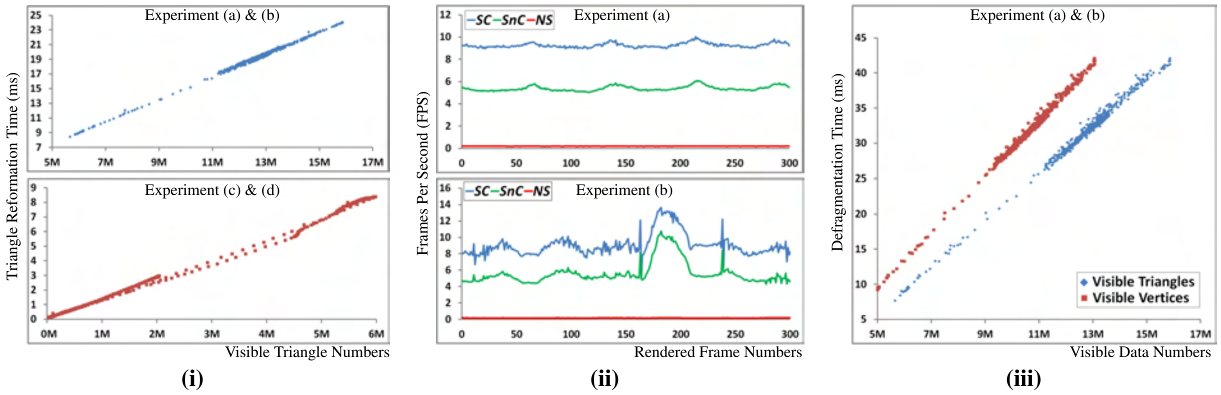

Figure 10: ***(i)*** *The scattered value pairs of reformation time and visible triangle numbers from all rendered frames;* ***(ii)*** *Comparison of rendering rates per frame in experiment (a) and (b) with three different GPU out-of-core approaches;* ***(iii)*** *The scattered value pairs of defragmentation time and visible triangle/vertex numbers from all rendered frames.*

Table 3: *Comparison of three different streaming approaches: Streaming with Coherence (our work), Streaming without Coherence, and No Streaming.*

Cam.	App.	Avg. FPS	Avg. CPU-GPU Streaming	Avg. GPU Defrag.	Avg. Visible Meshes	Avg. Streamed Meshes	Avg. Visible Triangles/Vertices	Avg. Streamed Triangles/Vertices
(a)	*SC*	9.26	15.70 ms	31.97 ms	6,056	58	12.884M / 10.721M	0.088M / 0.074M
	SnC	5.39	126.29 ms	N/A	6,056	6,056	12.884M / 10.721M	12.884M / 10.721M
	NS	0.19	5218.20 ms	N/A	6,056	6,056	12,884M / 10.721M	12.884M / 10.721M
(b)	*SC*	9.04	22.45 ms	31.10 ms	20,676	1,404	12.544M / 10.437M	0.448M / 0.370M
	SnC	5.64	127.77 ms	N/A	20,676	20,676	12.544M / 10.437M	12.544M / 10.437M
	NS	0.17	5825.70ms	N/A	20,676	20,676	12.544M / 10.437M	12.544M / 10.437M

[GM08] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 32:1–32:8.

[Hop96] HOPPE H.: Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 99–108.

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 189–198.

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 169–176.

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with cuda. In *GPU Gems 3, Chapter 39* (2007).

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 935–942.

[JWLL06] JI J., WU E., LI S., LIU X.: View-dependent refinement of multiresolution meshes using programmable graphics hardware. *The Visual Computer 22* (2006), 424–433.

[Mel98] MELAX S.: A simple, fast, and effective polygon reduction algorithm. In *Game Developer* (1998), pp. 44–49.

[RRR96] RONFARD R., ROSSIGNAC J., ROSSIGNAC J.: Full-range approximation of triangulated polyhedra. In *Proceeding of Eurographics, Computer Graphics Forum* (August 1996), Rossignac J., Sillon F., (Eds.), vol. 15(3), Eurographics, Blackwell, pp. C67–C76.

[Swa99] SWAROVSKY J.: Extreme detail graphics. In *Game Developers Conference* (1999), pp. 899–904.

[SZL92] SCHROEDER W. J., ZARGE J. A., LORENSEN W. E.: Decimation of triangle meshes. *SIGGRAPH Comput. Graph. 26* (July 1992), 65–70.

[WS98] WIMMER M., SCHMALSTIEG D.: *Load Balancing for Smooth Levels of Detail*. Tech. Rep. TR-186-2-98-31, Vienna University of Technology, 1998.

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), VIS '04, IEEE Computer Society, pp. 131–138.