GPU-based Streaming for Parallel Level of Detail on Massive Model Rendering

Chao Peng*
Virginia Tech

Yong Cao†
Virginia Tech

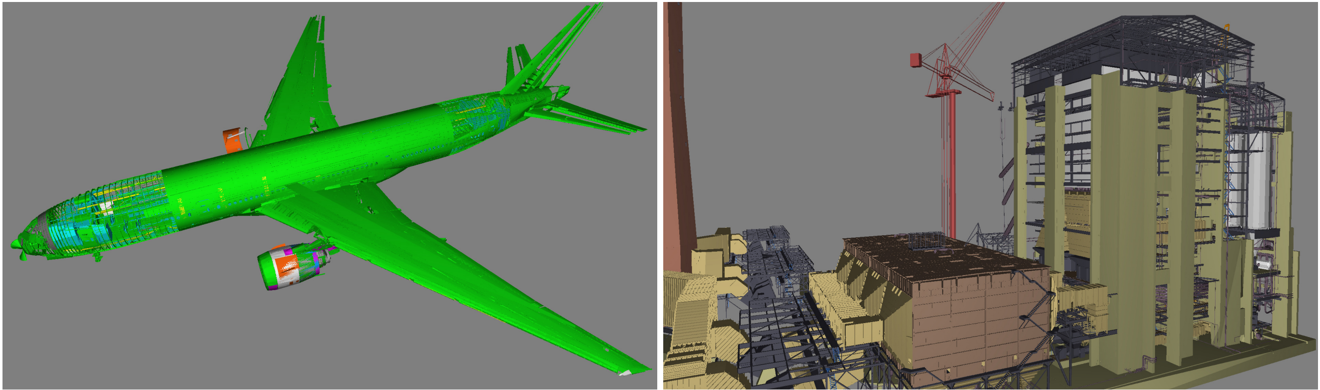

Figure 1: *Massive models rendered in our system. Left: Boeing 777 model. Right: Power Plant model.*

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, and does not scale well with the size of the models. We present a GPU-based progressive mesh simplification approach which enables the 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 develop a novel data structure to represent the progressive LOD mesh, and design a parallel mesh simplification algorithm towards GPU architecture. Second, we propose a GPU-based streaming approach which adopt a frame-to-frame coherence scheme in order to minimize the high communication cost between CPU and GPU. Our results show that the parallel mesh simplification algorithm and GPU-based streaming approach significantly improve the overall rendering performance.

Keywords: Level of detail, Streaming, Massive Model Rendering, GPGPU, Temporal Coherence.

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 models, such as industrial CAD models (e.g. airplanes, ships and architectures), composed of more than hundreds of millions of primitives. However, these complex datasets cannot be rendered interactively using brute force methods on a desktop workstation. The real challenge is how to develop a system that increases rendering performance without losing the rendering quality of datasets. One of the most commonly used techniques is mesh simplification, which can greatly reduced the amount of data stored and processed in the rendering systems.

Mesh simplification algorithms replace tessellated objects with coarser representations with less primitive count, such as *Levels of Detail* (LODs). Computing LOD models has been an very active research area in the past. For example, Hoppe [Hoppe 1996] introduced a well-known LOD mesh simplification approach, *Progressive Meshes*, where an original mesh is represented with a base mesh and a sequence of modifications (e.g. edge splits). At runtime, an appropriate LOD mesh can be rendered as the alternatives of the original mesh at a certain distance from the camera. However, given a large 3D model composed of many objects and massive number of primitives, constructing the LOD meshes for the model can be a very expensive process, which makes the online mesh simplification and rendering 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 in performance and the capability for general-purpose computation. However, most of the mesh simplification algorithms, including *Progressive Meshes*, are not naturally data parallel algorithms, and do not have trivial GPU-based parallel implementations. In addition, comparing to the processing power of GPUs, the maximum of memory on a GPU is insufficient to store a large number of primitives of really complex models. For example, the Boeing 777 model in our demonstration, shown in Figure 1 left, requires approximately 6 GByte of storage space for its vertex and triangle data, which can not fit into most modern GPUs. Another limitation of GPU-based implementations is the large overhead for transferring data from CPU to GPU, which could significantly decrease the performance if a large amount of geometry data needs to be transferred for each rendering frame.

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

To address these issues, we develop a novel data structure to represent the progressive LOD mesh, and design a parallel mesh simplification algorithm which can take advantage of the data parallel processing pipeline of GPUs. We also propose a GPU-based streaming approach which adopt a frame-to-frame coherence scheme in order to minimize the high transfer overhead between CPU and GPU.

We organize the rest of the paper as follows. In Section 2, we review some of the related works. Section 3 provides a brief overview for the data pre-process and runtime algorithm. Section 4 describes the pre-processing approach for mesh simplification and data permutation. In Section 5, we describe the parallel algorithm applied at runtime for the GPU-based streaming and LOD model generation. Section 6 describes our implementation and shows our experiment results. Finally, we conclude our work and discuss the limitations and future works in Section 7.

2 Related Work

Interactively rendering a massive and complex model is an active research area. In this section, we review some previous work that are highly related to our work. We discuss the existing approaches with respect to mesh simplification and GPU programming.

2.1 Mesh Simplification

A common representation for mesh simplification is using Progressive Meshes [Hoppe 1996] so that a LOD mesh can be recovered by applying a prefix of splitting sequence on the base mesh. One of the simplification methods is collapsing edges or vertex pairs interactively. At each iteration, an edge or vertex pair is selected and collapsed according to an energy function [Hoppe 1996], region-merging measurement [Ronfard et al. 1996] or Quadric Error Metrics [Garland and Heckbert 1997]. Garland and Zhou [2005] presented a generalized, Quadric-based simplification method that can be applied in any dimension. A different simplification method is presented in [Cohen et al. 1998], which is based on an appearance-preserving algorithm. The metric employed in the appearance-based algorithm measures the screen-space deviations in terms of the parameterized surface attributes in texture maps. [Lindstrom and Turk 2000] presented a more general image-based approach to decide which portion of a mesh to simplify.

The approaches for view-dependent rendering (VDR), as described in [Hoppe 1997] and [Xia et al. 1997], organizes the edge collapses of mesh simplification into a vertex hierarchy. An approximating mesh can be retrieved by traversing the hierarchy for possible collapsing/splitting operations. In order to increase the performance for rendering massive models , many researchers have proposed various ways in [Pajarola 2001; El-Sana and Bachmat 2002; Yoon et al. 2003; Wagner et al. 2007]. Meanwhile, several out-of-core VDR approaches for rendering massive models have proposed in [Lindstrom 2003; Yoon et al. 2004; Gobbetti and Marton 2008]. Other approaches using multi-resolution hierarchies, e.g, hierarchal level of details (HLOD). Different from traditional LOD, a HLOD is constructed by simplifying the separated portions of a scene. [Hoppe 1998] introduced a block-based algorithm for hierarchical simplification applicable to terrain rendering. [Erikson et al. 2001] employed a HLOD approach to interactively display complex static and dynamic models.

2.2 GPU Programming

Programable GPUs have allowed us to implement the traditional CPU sequential algorithms in parallel. With GPU parallelism, many approaches have been developed for efficient 3D data visualization, including: iso-surface rendering [Kruger and Westermann 2003; Buatois et al. 2006], data partitioning [Zhou et al. 2008; Lauterbach et al. 2009], and hiden surface removing[Wexler et al. 2005].

In order to efficiently generate LOD meshes, we emphasize on the techniques for GPU-based mesh simplification. [Ji et al. 2006] generated a LOD mesh on GPU based on a quad-tree structure constructed from poly-cube maps. In their techniques, an adaptive mesh is finalized in vertex shader. In [DeCoro and Tatarchuk 2007], 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 generated LOD mesh might not be visual fidelity. More recently, Hu et al. [2009] proposed a parallel algorithm for view-dependent LOD entirely implemented on GPU. The authors introduced a cascaded update method to split vertices without respecting their dependencies. However, this method is based on the assumption that the foldover artifacts cannot be observed. Moreover, their approach needs several passes to update vertex and triangle stream. When the original dataset cannot fit into GPU memory, it may lead to significant CPU-GPU data transfer overhead at runtime. In this paper, we tend to render large-scale massive model, and we contribute a GPU-based streaming approach for transferring the minimized amount of data at runtime.

As mentioned in Section 2.1, there are two major forms for mesh simplification: progressive meshes stored in a linear structure and vertex hierarchies for view-dependent rendering. As discussed in [Erikson et al. 2001], although we can selectively refine an arbitrary mesh using vertex hierarchies, the VDR algorithms may require significant memory and increase runtime cost. In order to take advantage of GPU linear memory space, we use progressive meshes instead of vertex hierarchies in our algorithm.

3 Overview

In this section, we present the basic concepts of our rendering system and give a brief overview of our approach. We also define some terminologies which are used in the rest of the paper.

Progressive algorithms are widely used for mesh simplification, where an original mesh is simplified to a base mesh by collapsing one edge at a time. The original mesh can be represented as the base mesh and a sequence of vertex splitting operations, which is the inverse transformation of edge collapsing. However, during mesh refinement, the vertex splitting operations are executed sequentially, which can be a significant performance bottleneck when processing a large mesh. We present an algorithm that performs LOD simplification in parallel at triangle level. Taking an original mesh as the input, our system applies edge collapsing operations in parallel at runtime and supports real-time rendering.

3.1 Preprocess

The input to our rendering system is a complex 3D model which consists of a large number of disconnected meshes. In preprocess stage, we calculate the edge collapsing information of each mesh and record it into an array, called $ecol$, where each element in $ecol$ corresponds to a source vertex, and its value, $ecol[i]$, is the target vertex that it collapses to. Moreover, we re-arrange the vertex and triangle data of original mesh based on the order of those collapsing operation. After the re-arrangement step, the first triangle of the mesh will the last one to be removed. Thus, at runtime, given the desired complexity of the mesh (e.g., number of triangles), we can select a set of successive triangles starting from the first, and generate the adaptive LOD mesh by reforming those triangles. We

describe how to construct $ecol$ and re-arrange the mesh data in Section 4.

3.2 System and Runtime Algorithm

In our system, each mesh in the 3D model is bounded by an Axis-aligned Bounding Box (AABB). The AABB serves for two purposes in our rendering system. First, we check the visibility of a mesh by testing its AABB against the view frustum. Second, if a mesh is visible, we use the projected area of its AABB to compute the desired complexity level of the mesh (refer to Section 5.1). Thus, the complexity of the 3D model is represented with a list of mesh complexities, we call it as *complexity list*.

In order to increase the rendering performance, we generate a simplified version of original model by utilizing GPU parallel architecture. Due to the limit of memory size on GPU, we can not fit the entire 3D model in GPU memory. However, with the re-arranged data, only a portion of triangles and a portion of vertices need to be held on GPU for LOD model generation, and we call them as *active triangles* and *active vertices*, respectively. Therefore, at a given frame, we first compute the complexity lists according to our LOD selection criteria. Then, in order to efficiently update active triangles and active vertices on GPU, we propose a *GPU-based streaming* approach that transfers much smaller amount of data from CPU memory by exploiting *frame-to-frame coherence*. In this way, we can re-use most of existing data on GPU from the previous frame such that the overhead of CPU-GPU data transfer can be minimized. We call those existing data as *existing triangles* and *existing vertices*.

Our runtime algorithm takes the following four steps to render a frame:

1. LOD selection. In this step, we need to compute the complexity list of the model. To do this, we test AABBs against view frustum to check if the meshes are visible or not. If a mesh is visible, we return its complexity with the desired number of vertices and triangles.

2. GPU-based Streaming. Based on frame-to-frame coherence, the process of streaming is to collect the triangles and vertices not existing in the previous frame, and transfer them from CPU to GPU. We called those triangles and vertices as *streamed triangles* and *streamed vertices*.

3. Triangle reformation. In this step, the active triangles and active vertices have been updated on GPU. In order to generate appropriate LOD meshes, the process of triangle reformation is to replace the vertices of each triangle with target vertices by looking up $ecol$.

4. Rendering . We use OpenGL Vertex Buffer Objects (VBOs) to render the generated LOD meshes.

We describe our runtime algorithm in detail in Section 5.The overview of the preprocess and runtime system is illustrated in Figure 2.

4 Data Preprocess

In this section, we first review the key components of the progressive algorithm for triangulated mesh simplification. Second, using an example, we describe how to construct the data array $ecol$, and re-arrange the original vertex and triangle data based on the order of edge-collapsing.

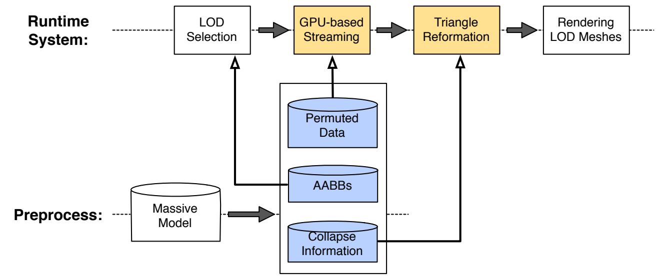

Figure 2: *The overview of the preprocess and runtime system.*

4.1 Key Algorithm Components

Progressive mesh simplification algorithm, originally proposed in [Hoppe et al. 1993], then further developed in [Hoppe 1996], [Ronfard et al. 1996] and [Garland and Heckbert 1997], is the most common simplification algorithm. The algorithm collapses edges iteratively, where an edge (v_1, v_2) is collapsed into a single vertex $\overline{v}$. Every edge is assigned a collapsing cost, which is computed based on a set of defined criteria. At each iteration, the edge with the lowest cost edge is collapsed, and one or two triangles are removed from the mesh. The following four steps are performed during eldge collapsing: (1) compute the costs of all valid edges; (2) collapse the lowest-cost edge by merging v_1 and v_2 to the position of $\overline{v}$; (3) replace v_1 and v_2 which are included in any triangle with $\overline{v}$; (4) remove v_1, v_2 and all triangles with both of the two vertices.

Position of target vertex. Obviously, the position of target vertex $\overline{v}$ can be either the endpoints of edges, 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. Normally, the new position is determined by forcing a close fit of the original mesh before edge collapsing. However, if the application tends to create adaptive representation of the simplification, using the endpoints for $\overline{v}$ is desirable [Garland and Heckbert 1998].

Error function. In order to achieve the best quality of simplification, error functions are selected to determine the order of edge collapsing operations (in other words. which edge to collapse first). [Hoppe 1996] presented an optimization scheme involving an error function with four energy terms. Although it helps to generate good results, calculating the cost for a given edge is an very costly operation, especially for large 3D models. To avoid this performance issue for real-time rendering, we choose an error function used by [Garland and Heckbert 1997]. It presents an efficient, quality-preserved method using Quadric Error Metrics (QEM). Original QEM method considers to collapse any pair of vertices (edge and non-edge). However, in our method, we only consider edge pairs are valid for collapsing.

Boundary vertices. In many 3D models, such as polygonal meshes created by CAD softwares, there are disconnected faces separated by borders and holes, which are important visual features [Garland and Heckbert 1998]. Preserving such features is crucial for an accurate simplification of the meshes. To do this, we restrict that the *Boundary Vertices* are not collapsible, which means that an edge cannot be collapsed by moving a boundary vertex to another. We define the term of *Boundary Vertex* as follows: if an edge (v_1, v_2) only existing in a single triangle, v_1 and v_2 are the boundary vertices; the edge (v_1, v_2) is a *Boundary Edge*. With the non-collapsible constraint, the simplest LOD mesh will be the one only constructed by boundary vertices, instead of a single triangle.

4.2 Data structure and processing

A 3D model visualized in our system is represented with a set of triangle meshes. A mesh M can be denoted as a tuple (V, T), where V is the vertex list. We denote $V = \{v_1, v_2, \ldots, v_m\}$, where m is the number of vertices in M. T is the triangle list defining the shape and topology of M. We denote $T = \{t_1, t_2, \ldots, t_n\}$, where n is the number of triangles. $t_i (i \in [1, n])$ is defined by a triple of vertex indices, $t_i = \{idx_1, idx_2, idx_3\}(idx_1, idx_2, idx_3 \in [1, m])$, which means that the shape of triangle t_i is formed by three vertices appearing in the order of $v_{idx_1}, v_{idx_2}, v_{idx_3}$.

We record edge collapsing information into an array, $ecol$. In order to assist the simplification process, two other data arraies are used for registering their new indices. One is for vertices, $permuteV$; the other is for triangles, $permuteT$. After each collapsing operation, we also want to know the number of vertices and triangles remaining in the mesh. We use another array , map, to record the remaining triangles. If the number of remaining vertices is i, the number of remaining triangles will be stored in $map[i]$. In order to explain our approach clearly, we will describe the process of simplification by providing a example (shown in Figure 3 and Figure 4).

4.2.1 Simplification process

Figure 3(a) shows a mesh composed of 7 vertices and 8 triangles so that we can initially set $map[7]$ to 8. The set of vertices $\{v_3, v_4, v_5, v_7\}$ are classified as boundary vertices, denoted as Q. In this example, we assume the the error function only considers the length of edge. Thus, the shorter an edge is, the smaller the cost is assigned to it. The costs on boundary edges are set to be infinitely large. Giving an $Edge = (v_a, v_b)$, where $a < b$, the edge collapsing operation can be formularized as follows:

$$Collapse(Edge) = \begin{cases} v_a \rightarrow v_b, & (v_a, v_b \notin Q) \\ v_b \rightarrow v_a, & (v_a \in Q, v_b \notin Q) \\ noncollapsible, & (v_a, v_b \in Q) \end{cases}$$

Figure 3(a-d) illustraste the steps reducing the original mesh to the simplest mesh. At each step, there are five operations:

1. Choosing and collapsing the lowest-cost edge. The index of target vertex, v_{tar}, which the source vertex, v_{src}, is collapsed to, is stored in $ecol$. The operation can be formularized as $ecol[src] = tar$;

2. Recording the information for vertex re-arrangement. The new index of v_{src} will be the number of vertices of the current mesh, denoted as vn, is recorded in $permuteV$. The operation is written as $permuteV[src] = vn$;

3. Recording the information for triangle re-arrangement. The new indices of the removed triangles depend on the number of triangles of the current mesh, tn. Assuming that two triangles, $t_{rm_1}, t_{rm_2} (rm_1 < rm_2)$, are removed, we record their new indices in $permuteT$. The operation is $permuteT[rm_1] = tn, permuteT[rm_2] = tn - 1$.

4. Recording the information for vertex-triangle number mapping. If k triangles are removed, $map[vn - 1] = tn - k$.

5. Removing v_{src}, t_{rm_1} and t_{rm_2}, then updating vn and tn.

For example, in the step of simplifying the mesh from Figure 3(b) to Figure 3(c), the current mesh shown in Figure 3(b) indicates vn is 6 and tn is 6. In Figure 3(c), after collapsing v_1 to v_2, $ecol[1]$ is equal to 2. $permuteV[1]$ is 6, the value of vn. Correspondingly, by removing t_1 and t_6, $permuteT[1]$ is 6, the value of tn; and $permuteT[6]$ is 5, the value of $tn - 1$. Since two triangles are removed, $map[5]$ is equal to 4. Figure 3(d) shows the simplest mesh compound of the minimal number of vertices and triangles.

In Figure 3(e), we deal with the boundary vertices and the remaining triangles. Since boundary vertices are non-collapsible, we assign an invalid value (e.g., -1) to $ecol[i](v_i \in Q)$ and $map[j](j \in [1, vn - 1])$. Each unassigned element of $permuteV$ corresponds to a boundary vertex. To complete $permuteV$, each of those elements is assigned with the value of vn in their increasing order. After each assigning step, we decrease the value of vn by 1. In the same way, we complete $permuteT$ for each of the remaining triangles.

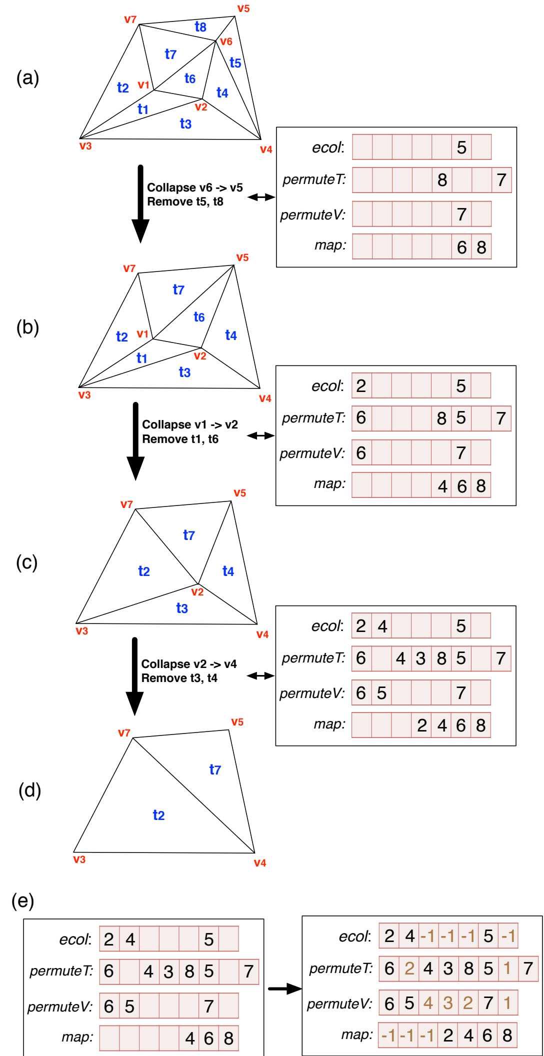

Figure 3: *An example of simplification. (a)-(d) show the intermediate meshes resulted by edge-collapsing. At each simplification step, the collapse information is recorded in the proposed array structures (on the right side). (e) shows the finalized array structures by handling the boundary vertices and the remaining triangles.*

4.2.2 Data re-arrangement

The sets of V and T in mesh M are re-arranged according to the information recorded in $permuteV$ and $permuteT$. If we define the re-arranged mesh as $M' = (V', T')$, the order of elements in V' and T' reflect the order of vertex/triangle removal. Thus, if a simplified version of M' contains k vertices ($k \in [q, m]$), where q is the number of boundary vertices, and l triangles ($l = map[k]$), it can be generated by using only a subset of V', $\{v_1, v_2, \ldots, v_k\}$, and a subset of T', $\{t_1, t_2, \ldots, t_l\}$. Data re-arrangement is required by our runtime algorithm for per-triangle reformation (see Section 5.3). We present the algorithm of data permutation in Algorithm 1. We also provide an example in Figure 4.

Algorithm 1 Re-arrange Data

Input: mesh M, $ecol$, $permuteV$, $permuteT$
Output: re-arranged mesh M', $ecol'$

for each $v_i \in M.V$ **do**
  $ecol'[permuteV[i]] \leftarrow permuteV[ecol[i]]$;
  $M'.V'.v[permuteV[i]] \leftarrow M.V.v[i]$;
end for
for each $t_i \in M.T$ **do**
  $M'.T'.t[permuteT[i]] \leftarrow M.T.t[i]$;
end for

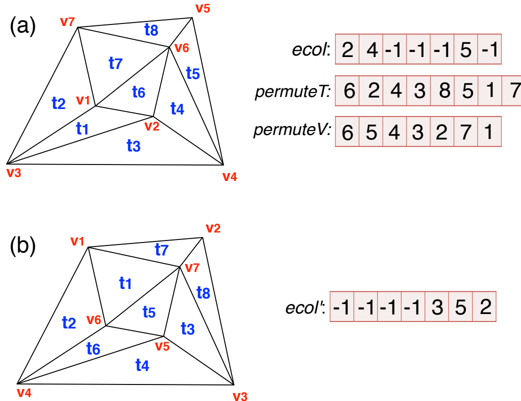

Figure 4: *An example of data re-arrangement. (a) shows the original mesh with the collapsing results from Figure 3. (b) shows the re-arranged data and the re-arranged ecol.*

5 Runtime Algorithm

In this section, we describe our runtime algorithm for rendering a complex 3D model. Our algorithm has two major contributions: (1) A novel GPU-based streaming approach minimizes the overhead of data transfer by exploiting frame-to-frame coherence (see Section 5.2); (2) A massive parallel process for generating LOD meshes by reforming the active triangles (see Section 5.3).

Algorithm Overview. The runtime algorithm takes a model with a set of triangulated meshes, AABBs, and a *ecol* array as the input. We define the model as $D = \{M_1, M_2, \ldots, M_r\}$, where r is the number of meshes; similarly, AABBs is defined as $B = \{b_1, b_2, \ldots, b_r\}$. Note that each mesh $M_i (i \in [1, r])$ has been re-arranged, and bounded by b_i in B. Also, $ecol_i$ in the $ecol$ array includes the edge collapsing information of M_i. The rendering process is shown in Algorithm 2.

Algorithm 2 Data Visualization

Input: D, B, $ecolList$

// Initialization: allocating memories to store triangles and vertices
$activeT \leftarrow$ **new** triangle list;
$existingT \leftarrow$ **new** triangle list;
$streamedT \leftarrow$ **new** triangle list;
$activeV \leftarrow$ **new** vertex list;
$existingV \leftarrow$ **new** vertex list;
$streamedV \leftarrow$ **new** vertex list;

// Initialization: allocating memories to store complexity lists
$atn \leftarrow$ **new** list; $etn \leftarrow$ **new** list; $stn \leftarrow$ **new** list;
$avn \leftarrow$ **new** list; $evn \leftarrow$ **new** list; $svn \leftarrow$ **new** list;

// LOD selection stage
for each $b_i \in B$ **in parallel do**
  Testing b_i against the view frustum;
  if M_i is visible **then**
    Compute the complexity of M_i into $avn[i]$ and $atn[i]$;
  else
    $avn[i] \leftarrow 0$; $atn[i] \leftarrow 0$;
  end if
end for
Prefix-Sum avn and atn **in parallel**;

//Updating stage: GPU-based streaming to update active lists
UpdateTriangles(D, $existingT$, $streamedT$, atn, etn, stn, $activeT$);
UpdateVertices(D, $existingV$, $streamedV$, avn, evn, svn, $activeV$);
$existingV \leftarrow activeV$; $existingT \leftarrow activeT$;
$evn \leftarrow avn$; $etn \leftarrow atn$;

// Triangle reformation stage
for each $t_i \in activeT$ **in parallel do**
  ReformTriangle(t_i, atn, avn, $ecolList$, $activeT.t_i$);
end for

// Rendering stage
for each $M_i \in D$ **do**
  Assign per-mesh attributes (e.g. color);
  Render the LOD mesh of $activeV[v_{avn[i]+1}, v_{avn[i+1]}]$ and $activeT[t_{atn[i]+1}, t_{atn[i+1]}]$;
end for

During initialization, global memory in GPU is allocated for storage. For a given frame, the active triangles, $activeT$, and the active vertices, $activeV$, are held in successive GPU memory space. Similarly, those from previous frame are stored in $existingT$ and $existingV$; and new primitives, which do not exist in the previous frame, are stored in $streamedT$ and $streamedV$. Figure 5 shows how the active primitives are organized in $activeT$ and $activeV$.

In our algorithm, The complexity of a mesh is actually represented with two numbers: the number of vertices and the number of triangles. Thus, the complexity of D is denoted as two complexity lists: one is for triangle, another is for vertex. We organize each complexity list based on the operation of prefix sum. As denoted in Algorithm 2, atn is the complexity list associating to $activeT$, and avn is the complexity list associating to $activeV$. As a result, a block of vertices in $activeT$, chosen from M_i, can be formalized as $activeV[v_{avn[i]+1}, v_{avn[i+1]}]$; and the size of this block can be recovered from $avn[i+1]-avn[i]$. Accordingly, in Algorithm 2, evn

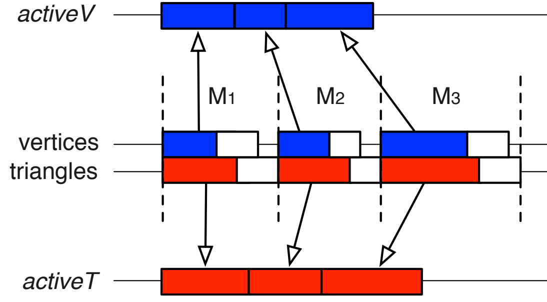

Figure 5: *This example shows how the active triangles and active vertices are organized. The colored blocks stand for the triangles and vertices chosen from the original meshes.*

and etn associate to $existingV$ and $existingT$, respectively. svn and stn associate to $streamedV$ and $streamedT$, respectively.

After initialization, the algorithm first computes the complexity lists, avn and atn. Then, the active data, $activeT$ and $activeV$, are updated on GPU. After that, in order to generate appropriate LOD meshes, each triangle in $activeT$ is reformed. Finally, we render those LOD meshes using OpenGL Vertex Buffer Objects.

5.1 LOD Selection

In our system, the task of LOD selection is to generate the adaptive complexity lists of the original model. [Funkhouser and Séquin 1993] proposed that LOD selection for a multi-mesh model is a discrete optimization problem. Their approach selects appropriate levels for all potentially visible objects so that the total number of polygons is within a given maximal count. [Wimmer and Schmalstieg 1998] re-evaluated the problem and provided a closed-form expression to solve LOD selection cheaply at each frame. Our metric of LOD selection takes advantage of this closed-form expression to generate the complexity lists, which is formularized as follows:

$$vn_i = N \frac{A_i^{\frac{1}{\alpha}}}{\sum_{i=1}^{m} A_i^{\frac{1}{\alpha}}} \qquad (1)$$

In Equation 1, the vertex count, vn_i, stands for the vertex complexity for mesh M_i. Given the maximal vertex count N and the area A_i, vn_i is computed out of m meshes. A_i denotes the projected area of the AABB of mesh M_i on image plane. The exponent, $\frac{1}{\alpha}$, aims to estimate the contributions for model perception, refer to the benefit function detailed in [Funkhouser and Séquin 1993]. Instead of calculating the accurate area of the projected region, we estimate it with a bounding circle so that A_i can be computed efficiently. N is a user-defined parameter, and can be wisely chosen in terms of the desired frame rate or rendering quality. Using data array, map, generated in the preprocess, the desired triangle count, tn_i, is defined as $map[vn_i]$.

As shown in the LOD selection step of Algorithm 2, all AABBs are first tested for view-frustum culling. If an AABB, b_i, is outside the view frustum, the complexity for mesh M_i is set to zero. If b_i is inside, the desired vertex and triangle numbers are calculated. For fast execution, We use CUDA CUDPP [Harris et al. 2007] to compute Equation 1 on GPU. If the desired vertex number is less than the number of boundary vertices, q, we set the desired vertex number to q. It can effectively avoid "popping" effects, even though the total number of vertices may exceeds N. Finally, we modify the complexity lists, avn and atn, using prefix sum by applying CUDPP on GPU.

5.2 GPU-based Streaming with Frame-to-frame Coherence

After computing the complexity lists, we stream and update the set of active data stored on GPU memory. Since the procedures used for streaming active triangles and active vertices are the same, we only elaborated the procedure for streaming active triangles, *StreamTriangles*, in Algorithm 3. This procedure takes D, $existingT$, $streamedT$ and three corresponding complexity lists as the input, and updates $activeT$ as the output. Note that $existingT$, $streamedT$ and $activeT$ are initially allocated on GPU global memory, while $streamedT$ is allocated on CPU main memory.

Let us walk through the major steps of this procedure.The first step in Algorithm 3 is collecting the triangles not existing in the previous frame and prepare $streamedT$ on CPU. To do this, we check the original meshes iteratively to find out if any triangle should be collected from them. In each iteration, if the complexity for M_i is increased, we should include more triangles in active triangle list than the previous frame. Referring to Algorithm 3, it means $n > 0$. Then, we collect those newly added triangles from $M_i.T$, and update stn and $streamedT$ accordingly. If $n \leq 0$, it indicates that the complexity for M_i is decreased or not changed; so we do not collect any triangle, and set $stn[i+1]$ equal to $stn[i]$.

Algorithm 3 Stream Active Triangles

procedure StreamTriangles(
in D, $existingT$, $streamedT$, atn, etn, stn;
out $activeT$)
  // Data collection on CPU
  $ntn[1] \leftarrow 0$;
  for each $M_i \in D$ **do**
    $n_1 \leftarrow etn[i+1] - etn[i]$;
    $n_2 \leftarrow atn[i+1] - atn[i]$;
    $n \leftarrow n_2 - n_1$;
    if $n > 0$ **then**
      $stn[i+1] \leftarrow stn[i] + n$;
      $streamedT[t_{stn[i]+1}, t_{stn[i+1]}] \leftarrow M_i.T[t_{n_1+1}, t_{n_2}]$;
    else
      $stn[i+1] \leftarrow stn[i]$;
    end if
  end for

  // CPU-GPU data transfer
  Transferring $streamedT$ from CPU memory to GPU memory;

  // Defragmentation on GPU
  for each $t_i \in activeT$ **in parallel do**
    $midx \leftarrow 0$;
    Binary searching atn to return $midx$ for t_i;
    $tidx \leftarrow i - atn[midx]$;
    $n_1 \leftarrow etn[midx+1] - etn[midx]$;
    if $tidx \leq n_1$ **then**
      $j \leftarrow tidx + etn[midx]$;
      $activeT.t[i] \leftarrow existingT.t[j]$;
    else
      $j \leftarrow tidx - n_1 + stn[midx]$;
      $activeT.t[i] \leftarrow streamedT.t[j]$;
    end if
  end for

In the second step, we transfer $streamedT$ from CPU to GPU. Because $streamedT$ is constructed based on frame-to-frame coherence, we have a minimized cost of data transfer. After transferring $streamedT$ to GPU memory, we update $activeT$ by merg-

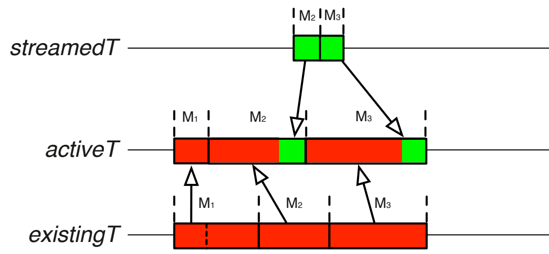

Figure 6: *Updating $activeT$ by reuniting $existingT$ and $streamedT$. In this example, the complexity of M_1 is decreased so that the block of activeT for M_1 is shrunk and filled only with the corresponding block of $existingT$. The blocks of activeT for M_2 and M_3 are extended and updated by using both $existingT$ and $streamedT$.*

ing $streamedT$ and $existingT$. Although blocks of memories are released in $activeT$ due to the decreased complexities of some meshes, we cannot simply fill $streamedT$ into empty "holes". It is because a block of triangles representing a mesh may be broken into many blocks that are not close to each other. As a result, the essential triangle-continuity of a mesh can not be preserved in $activeT$. This problem is generally known as *Data Fragmentation*.

To avoid the problem of data fragmentation, in the third step, we employ a *defragmentation* process to update $activeT$ in parallel. We first determine the original mesh that the triangle, $activeT.t_i$, belongs to. We search the triangle complexity list, atn, and return the index of the mesh, $midx$. Since the order of triangles appearing in $activeT$ is the same as the order in atn, if $atn[k] < tidx \leq atn[k+1]$, $midx$ is equal to k. In order to search atn efficiently, we use binary search algorithm. Then, $activeT.t_i$ is replaced by a triangle chosen from either $existingT$ or $streamedT$. We compute the local index of this triangle, $tidx$, in M_{midx}. If $tidx \leq n_1$, $activeT.t_i$ is replaced with a corresponding triangle in $existingT$; otherwise, it is replaced with a corresponding triangle in $streamedT$. Figure 6 shows an example of how to update the $activeT$.

5.3 Triangle Reformation

When $activeT$ and $activeV$ are ready in GPU global memory, we reform each triangle in the array, $activeT$, by collapsing its vertices to the target vertices. As shown in Algorithm 2, we perform this reforming process for all active triangles in parallel. We elaborate the procedure for per-triangle reformation, *ReformTriangle*, in Algorithm 4. As mentioned, each triangle is represented as $t_i = \{idx_1, idx_2, idx_3\}$, where $idx_j (j \in [1, 3])$ is a vertex index. The process of reformation is collapsing a vertex v_{idx_j} to a target vertex based on the vertex mapping array, *ecol*, and the vertex complexity defined in avn. The first step is to determine which original mesh the triangle, $activeT.t_i$, belongs to. Similar to the method used in the *Defragmentation* step in Algorithm 3, we conduct the binary search in atn to find the mesh index, $midx$.

In the second step, we replace a vertex index, idx_j, in t_i with a target index by looking up the collapse information stored in the $ecol_{midx}$. We elaborate this process in the procedure, $Collapsing$, which performs per-vertex collapsing. This procedure takes three inputs: src_vidx (the index of source vertex), vn (the desired number of vertices recovered from avn) and *ecol*; and it returns the index of target vertex, tar_vidx. As mentioned in Section 4.2.2, the 3D model has been re-arranged based the order of edge-collapsing in preprocess step. Thus, for a generated LOD mesh, its triangles must be formed by the set of vertices, $\{v_1, \ldots, v_{vn}\}$. According

Algorithm 4 Triangle Reformation

procedure ReformTriangle(
in $activeT.t_i$, atn, avn, $ecolList$;
out $activeT.t_i$)
  $midx \leftarrow 0$;
  Binary-searching atn to return $midx$ for $activeT.t_i$;
  $ecol \leftarrow ecolList.ecol_{midx}$;
  $vn \leftarrow avn[midx+1] - avn[midx]$;
  for $j = 1$ to 3 **do**
    $vidx \leftarrow activeTri.t_i.idx_j$;
    Collapsing($vidx$, vn, $ecol$, $activeT.t_i.idx_j$);
  end for

procedure Collapsing(
in src_vidx, vn, $ecol$;
out tar_vidx)
  $tar_vidx \leftarrow src_vidx$;
  while $tar_vidx > vn$ **do**
    $tar_vidx \leftarrow ecol[tar_vidx]$;
  end while

to this property, the procedure, $Collapsing$, maps src_vidx to a tar_vidx while satisfying $tar_vidx \leq vnum$. In Figure 7, we provide an example of how a LOD mesh is generated by using the *Collapsing* procedure.

5.4 Rendering with Vertex Buffer Objects

In the last step of our rendering system, the desired LOD meshes have been constructed in the index buffer, $activeT$, and vertex buffer, $activeV$ on GPU. In order to accelerate the rendering, we use OpenGL vertex buffer objects (VBOs) by registering them to the address space of $activeT$ and $activeV$. In order to efficiently assign per-mesh attributes (e.g., mesh color), we render LOD meshes sequentially with appropriate data offsets.

As mentioned in Section 5.1, the total number of vertices may exceed the user-defined maximal vertex count, N, due to the existence of boundary vertices. If the original data contains a large number of boundary vertices, it is possible that the active data exceeds the maximum of GPU memory. To solve this problem, we allow a second call to the runtime algorithm to render the extra data. However, this solution will significantly influence the performance. If the problem appears frequently, an optimal solution is reducing N and construct a simpler approximation of the original 3D model.

6 Experiments and Results

In this section, we discuss the evaluation of our GPU-based real-time rendering system, and highlight the performance advantages.

6.1 Implementation and Environment Models

We have implemented our rendering algorithm with parallel LOD 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 v3.2, and runs on a 64-bit Windows system. Our algorithm has been applied to two complex 3D models. One is a Boeing 777 airplane composed of more than 332 millions of triangles and 223 millions of vertices. Another is a coal fired power plant composed of more than 12 millions of triangles and 6 millions of vertices (see Figure 1).

Our system is designed for very complex models with hundreds of

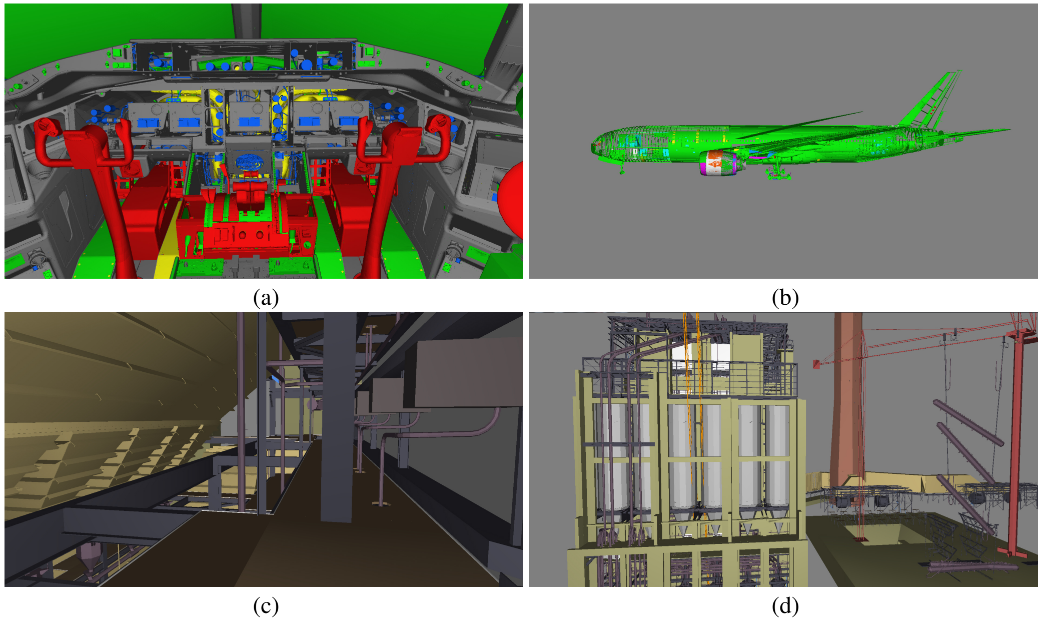

Figure 8: *Different 3D models rendered with different camera views. Boeing 777 model is used in images (a) and (b); power plant model is used in images (c) and (d).*

millions primitives. Boeing model requires approximately 6 GB memory space for storage, which exceed the maximum of memory on graphics card. As mentioned in Section 5.2, we applied a GPU-based streaming approach to update the list of active data based on frame-to-frame coherence. However, the power plant model, a relatively small model, can fit into GPU memory so that the cost of CPU-GPU data transfer can be completely eliminated.

6.2 Performance Evaluation

In our experiments, we evaluate the performance of two major contributions in our rendering system: parallel progressive simplification algorithm and GPU-based streaming with frame-to-frame coherence. We also provide overall system evaluation using two example datasets under different viewing options.

6.2.1 Parallel Progressive Simplification

We test the parallel simplification algorithm on GPU and show the result in Table 2, column *Reforming*. It shows that our implementation is very efficient, where 17 million triangles can be processed in 35 ms, and the algorithm is not the performance bottleneck of the overall rendering system.

6.2.2 GPU-based Streaming

To evaluate the efficiency the GPU-base streaming approach, especially the proposed frame-to-frame coherence method, we compare our implementation, *Streaming with Coherence (SC)* with two other approaches: *No Streaming (NS)* and *Streaming without Coherence (SnC)*, which are commonly used brute-force strategies. *No Streaming* approach sequentially copies all simplified meshes from CPU memory space to GPU one-by-one. *Streaming without Coherence* collects all simplified mesh in a continuous CPU memory block, and then streams the entire block to GPU once. Our coherence-based streaming approach only streams a small amount of geometry data to GPU, which contributes to a significant performance gain on GPU-based architecture.

We perform the performance comparison with two different camera settings using the Boeing 777 model, as shown in Figure 8 (a) and (b). From the results shown in Table 1, our approach requires much less memory transfer from CPU to GPU. With the camera setting shown in Figure 8 (a), only 0.04% of the total 5,814,786 active triangles needs to be transferred, and only 0.042% of the total 3,879,636 active vertices need to be transferred. With the camera setting shown in Figure 8 (b), only 0.038% of the total 17,373,329 active triangles needs to be transferred, and only 0.0037% of the total 14,501,553 vertices need to be transferred. Therefore, our coherence-base streaming approach achieves 1.96X to 2.39X speedup compared with *SnC* approach, and 4.82X to 19.83X speedup compared with *NC* approach.

6.2.3 Overall System Evaluation

Our results show that we can achieve an interactive rendering rate for both example 3D models used in our experiment: 26-212 fps for the Power Plant model and 6-22 fps for the Boeing 777 model. In our experiments, we have set α in Equation 1 to 3, because Wimmer and Schmalstieg [1998] claimed that when α is equal to 3, Equation 1 produces the equivalent of Funkhouser's benefit function [1993].

In Table 2, we illustrate the performance results using both example 3D models and different camera settings, as shown in Figure 8 (a) (b) (c) and (d). To provide the insight analysis of our system, we also present the breakdown of the processing time for different steps in our rendering system, including *LOD Selection*, *Streaming*, *Triangle Reforming*, and *Rendering*. The table shows that, when rendering a large number of active triangles, for Boeing 777 model and camera setting (b), a large number of triangles are streamed to GPU, making streaming step the performance bottleneck of the system (47.68% of total rendering time). In all experiments, rendering step never becomes the bottleneck, even more than 10 millions of triangles need to rendered.

Table 1: *Comparison of three different streaming approaches: Streaming with Coherence (our work), Streaming without Coherence, and No Streaming.*

Camera Setting	Approaches	FPS	CPU Collection	CPU to GPU Transfer	GPU Defrag.	# of Active Triangles	# of Active Vertices	# of Streamed Triangles	# of Streamed Vertices
(a)	*SC*	14	13.63 ms	5.13 ms	19.68 ms	5,814,786	3,879,636	2,314	1,657
	SnC	12	26.69 ms	48.82 ms	N/A	5,814,786	3,879,636	5,814,786	3,879,636
	NS	2	N/A	762.46 ms	N/A	5,814,786	3,879,636	5,814,786	3,879,636
(b)	*SC*	6	15.36 ms	12.02 ms	59.64 ms	17,373,329	14,501,553	6,649	5,476
	SnC	5	68.26 ms	139.82 ms	N/A	17,373,329	14,501,553	17,373,329	14,501,553
	NS	2	N/A	419.81 ms	N/A	17,373,329	14,501,553	17,373,329	14,501,553

Table 2: *Overall system performance.*

Model	Camera Setting	FPS	Selection	Streaming	Reforming	Rendering	Number of Active Triangles	Number of Active Vertices
Boeing 777	(a)	14	23.82 ms 31.14%	38.44 ms 50.25%	9.6 ms 12.55%	4.64 ms 6.07%	5,814,786	3,879,636
Boeing 777	(b)	6	19.14 ms 10.49%	87.02 ms 47.68%	35.05 ms 19.21%	35.05 ms 22.62%	17,377,329	14,501,553
Power Plant	(c)	110	4.79 ms 50.96%	N/A	1.75 ms 18.62%	2.86 ms 30.43%	451,355	244,699
Power Plant	(d)	39	6.46 ms 25.33%	N/A	15.23 ms 59.73%	3.81 ms 14.94%	2,790,689	1,314,593

7 Conclusion and Future Work

We presented a novel approach for visualizing complex models on GPU at interactive rates, especially focusing on LOD-based mesh simplification. In our system, the input models are re-arranged into a novel data structure in pre-process step, which enable a parallel triangle processing algorithm for real-time mesh simplification. At runtime, the mesh simplification algorithm processes hundreds of millions of triangles in parallel based on a LOD selection criterion. Since only a fraction of original data is used for final rendering, we propose a GPU-based streaming approach by employing frame-to-frame coherence. In our streaming approach, we also develop a defragmentation method to managing the data continuity on GPU, so that the data can be efficiently rendered using OpenGL.

Limitations. Our approach assumes high temporal coherence between frames. If the camera is changed dramatically from one frame to the next, the amount of the streamed data collected based on 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 the CPU main memory.

Future works. There are several future works that can strengthen our system. First, our approach can be extended to render large single-mesh models. Using a spatial partitioning structure, such as K-D tree, each partition can be preprocessed as a separated mesh, then rendered with our GPU-based runtime algorithm. Second, LOD selection metric is an important factor for managing active data and preserving visual fidelity. We would like to explore and analyze other metrics applicable for rendering massive and complex models. Third, in our runtime algorithm, we keep the data from the previous frame on GPU so that we can update the active data efficiently. However, it is not the best method for optimizing memory usage. In the future, we would like to explore some in-place algorithms for active data updating.

References

BUATOIS, L., CAUMON, G., AND LÉVY, B. 2006. Gpu accelerated isosurface extraction on tetrahedral grids. In *Advances in Visual Computing*, vol. 4291. Springer Berlin, 383–392.

COHEN, J., OLANO, M., AND MANOCHA, D. 1998. Appearance-preserving simplification. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 115–122.

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.

EL-SANA, J., AND BACHMAT, E. 2002. Optimized view-dependent rendering for large polygonal datasets. In *Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA, VIS '02, 77–84.

ERIKSON, C., MANOCHA, D., AND BAXTER, III, W. V. 2001. Hlods for faster display of large static and dynamic environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, I3D '01, 111–120.

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.

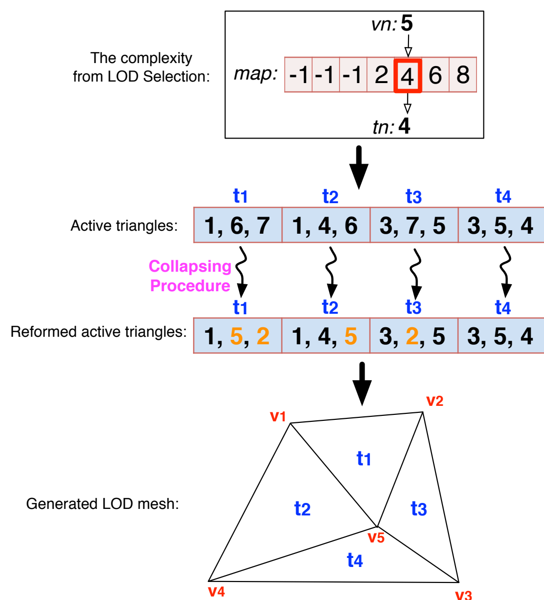

Figure 7: *In this figure, we use the mesh example shown in Figure 4(b). Assuming that the complexity of the mesh after LOD selection is $vn = 5$, $tn = 4$. After updating the active triangles of the mesh, we reform each triangle using the* Collapsing *procedure over its three vertices. The generated LOD mesh has the exactly same shape as Figure 3(c).*

GARLAND, M., AND ZHOU, Y. 2005. Quadric-based simplification in any dimension. *ACM Trans. Graph. 24* (April), 209–239.

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

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

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1993. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '93, 19–26.

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

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.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. *Visualization Conference, IEEE 0*, 35.

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.

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

KRUGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, IEEE Computer Society, Washington, DC, USA, VIS '03, 38–.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2, 375–384.

LINDSTROM, P., AND TURK, G. 2000. Image-driven simplification. In *ACM Transactions on Graphics*, vol. 19, 204–241.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, I3D '03, 93–102.

PAJAROLA, R. 2001. Fastmesh: efficient view-dependent meshing. In *Proc. of Pacific Graphics*, 22–30.

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

WAGNER, G. N., RAPOSO, A., AND GATTASS, M. 2007. An anti-aliasing technique for voxel-based massive model visualization strategies. In *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I*, Springer-Verlag, Berlin, Heidelberg, ISVC'07, 288–297.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. Gpu-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, New York, NY, USA, HWWS '05, 7–14.

WIMMER, M., AND SCHMALSTIEG, D. 1998. Load balancing for smooth levels of detail. Tech. Rep. TR-186-2-98-31, Vienna University of Technology.

XIA, J. C., EL-SANA, J., AND VARSHNEY, A. 1997. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics 3* (April), 171–183.

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.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, SIGGRAPH Asia '08, 126:1–126:11.