COMPUTER-AIDED DESIGN & APPLICATIONS, 2016
VOL. 13, NO. 2, 173–183
http://dx.doi.org/10.1080/16864360.2015.1084184

Parallel LOD for CAD model rendering with effective GPU memory usage

Chao Peng[1] and Yong Cao[2]

[1]University of Alabama in Huntsville, USA; [2]Virginia Tech, USA

ABSTRACT

Lack of memory is one of reasons that makes rendering of massive CAD models challenging on a single workstation. As CPU main memory size reaches tens of gigabytes, GPU memory is far behind. To render a model with several gigabytes of vertices and triangles, Level-Of-Detail (LOD) algorithms are used to view-dependently select potions of datasets from the data repository on CPU, and then transfer them to GPU at each time a frame being rendered. But, the question of how to effectively and fully utilize GPU memory to achieve best possible rendering quality for an oversized CAD model has not been addressed. In this paper, we propose a parallel LOD approach that uses both view parameters and GPU memory size for adaptive adjustments of geometric complexity. Also, our GPU out-of-core technique minimizes the size of CPU-to-GPU data transfer by taking advantages of frame-to-frame coherence. The experimental results show that our approach effectively renders Boeing 777 airplane model, composed of over 300 million triangles, at highly interactive frame rates.

KEYWORDS

Level-of-detail; massive model rendering; GPU out-of-core; GPGPU

1. Introduction

With the support of OpenGL driver, a modern GPU is capable of rasterizing thousands of millions of triangles per second. GPU computational power is affected by the density of transistors. In the past two decades, as Chen stated in [3], the number of transistors on GPU was "*more than doubling for every one and half years, exceeding the projection of Moore's Law*". GPU evolution has achieved billions of transistors per chip. Under such performance trend, GPUs continue to support the rendering of large 3D models as they continue to advance in complexity.

The size of GPU memory is limited. While CPU main memory has been well developed to cache large datasets (feasible to allocate tens of gigabytes memory on RAMs), GPUs do not have the same storage capability. Standard GPU devices on the market are often configured with 2–4GB memory. A complex CAD model, such as Boeing 777 airplane model - all triangles, vertices, surface normals and geometry colors requiring 6GB memory on storage - cannot fit into the GPU. We can transfer only a portion of data to GPU. Then, to render it, a LOD technique is applied to construct a simplified version as the rendering alternative. In this paper, we present a GPU-accelerated parallel LOD approach to dynamically select and transfer data from CPU to GPU, and generate a simplified version of the model at each time a frame being rendered. Note that, extreme geometry models may exceed CPU memory limits, so traditional external memory algorithms were designed to fetch and access data stored in slow bulk memory (e.g., hard drives) [29]. Those external memory approaches are not our focus and excluded from the discussion of this paper.

Memory size determines the total number of primitives, which is a budget to be allocated to all objects. Conceptually, an object can be rendered at any level of detail. While preferring to determine the desired level by considering its visual importance, we also need to make sure the number of primitives for the objects is not over budget. This brings a problem known as *LOD Selection*. Existing LOD selection solutions, such as [11], [17], [21], consider only view parameters. Given a view angle and distance to the model, objects close to the camera need more geometric detail than those far away. Whenever the view angle or distance changes, the detail of objects change; accordingly, a new portion of data should be selected.

Using only view parameters may cause two problems: (1) the size of selected data portions exceeds GPU memory maximum and cannot fit into the GPU for rendering; (2) the size of selected data portions is less than the GPU memory size, so they can be rendered but GPU memory is not fully utilized. In this paper, we present a two-pass LOD selection algorithm that considers both view parameters and GPU memory size to ensure full utilization of GPU memory while preserving visual fidelity.

CONTACT Chao Peng chao.peng@uah.edu

Our rendering approach contains three major components: *LOD Selection*, *LOD Model Generation*, and *OpenGL Rendering*, as illustrated in Fig. 1. *LOD Selection* determines the geometric complexity of objects. It computes the numbers of vertices and triangles for each object. *LOD Model Generation* fetches corresponding portions of vertices and triangles from the repository of main memory to the GPU, and generates the desired simplified versions. *OpenGL Rendering* rasterizes simplified meshes.

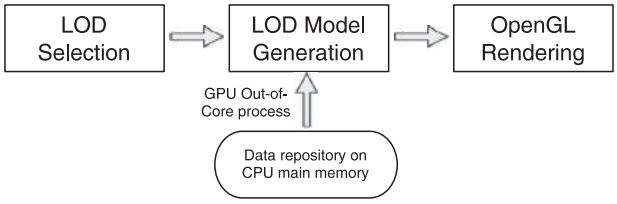


Figure 1. System overview.

The rest of this paper is organized as follows. Section 2 reviews previous work related this paper. We describe our GPU-accelerated mesh simplification approach in Section 3. We describe our contribution in LOD selection in Section 4. We show our experimental results in Section 5. Section 6 concludes our work.

2. Related work

In the past, many researchers worked on massive model rendering. Yoon et al. [33] discussed various techniques supporting interactive visualization of massive 3D models. Related to this paper, we review some existing LOD algorithms. Given a 3D object at distance to the camera, LOD algorithms can approximate a less complicated but visually faithful representation. The basic idea is discrete LOD, such as [24], that creates a set of simplified versions. Discrete LOD allows instantly accessing a LOD representation, but it may cause visual popping artifacts, where the transition of a level of detail to another is abrupt and noticeable to the viewer. Giegl et al. [14] blended neighbor LOD representations in image space to minimize the artifacts, but the smooth transitions between LOD representations are still hard to achieve. Continuous LOD algorithms have been developed by using a series of operations on geometric primitives, such as progressive modification ([12], [18]), selective refinements ([19], [32]), image-driven methods ([16], [22]) and parallel algorithms ([8], [9]). The most well-known algorithm is Progressive Meshes [18] that simplifies a triangular mesh based on a sequence of edge-collapsing operations. At each edge-collapsing operation, an edge is selected and removed by merging its two vertices, and corresponding triangles are eliminated. Operations continue until the mesh cannot be further simplified (e.g., reaching the base mesh). One of the questions is how to find the target position when merging the two vertices of the edge. Galand and Heckbert [12] and Lindstrom [23] introduced Quadric Error Metrics (QEM) to find a target position by minimizing the sum of squared distances to face planes. Later, Garland and Zhou [13] extended QEM to any dimension. Swarovsky [28] used the coordinates of one of two vertices as the target one, so that they can avoid memory allocation for new vertices.

Parallel LOD algorithms are proposed by utilizing modern GPU devices. The challenge in parallel algorithm design was how to remove data dependencies introduced in traditional CPU-based algorithms, so that fine-grained GPU architectures can be fully utilized to gain a significant performance boost. DeCoro and Tatarchuk [7] presented a vertex clustering method using the shader-based fixed GPU graphics pipeline. The representative vertex position for each cluster can be independently computed in geometry shader stage. Hu et al. [20] introduced a GPU-based approach for view-dependent Progressive Meshes, where vertex dependencies were not fully considered during a cascade of vertex splitting events. Derzapf et al. [9] encoded the dependency information into a GPU-friendly compact data structure; then later, Derzapf et al. [10] extended their previous work to out-of-core applications. Peng and Cao [26] used an array structure to support GPU parallelization. Derzapf and Guthe [8] presented a dependency-free progressive mesh algorithm on GPU using a bounding volume hierarchy over split and collapse operations.

As CAD models become more complex, they may not be stored on GPU for rendering. Instead, they are maintained in external memory (e.g., hard drives); and data portions are selected and transferred to internal memory (e.g., RAMs). Mesh simplification algorithms and out-of-core techniques are merged with hierarchical structures, where the desired simplified model can be generated by traversing the hierarchies. Aliaga et al. [1] presented an interactive rendering system of complex models. The system employed prefetching schemes for models larger than available CPU main memory. Bruderlin et al. [2] introduced a visibility guided rendering approach with KD-tree for real time visualization of large datasets. Their approach determined the polygons contributing most to the scene on a frame-by-frame basis. Popov et al. [27] introduced a high performance ray-tracing approach on GPU. KD-tree was used to partition and store triangles in the space hierarchy, and they achieve a peak performance of 16 million rays per second for reasonably complex scenes. KD-tree required a significant amount

of memory. Each GPU thread (or a ray) traverses the KD-tree to find visible triangles and eliminate invisible ones. Depending on space partitioning criteria of KD-tree, the traversal might lead to significant imbalanced workload among GPU threads. Varadhan and Manocha [30] presented a prioritized prefetching approach for efficient out-of-core data management. Their approach considered both LOD- and visibility-based coherence between successive frames. Correa et al. [6] proposed a visibility-based prefetching algorithm. They predicted the set of nodes likely seen next and sent them to memory ahead of time. Yoon et al. [34] used a cluster-based integration approach with out-of-core data management, where the input 3D model is represented with multiple progressive meshes in a clustered hierarchy (CHPM). Cignoni et al. [4] used a binary tree for mesh partitioning, and allowed the construction of multi-resolution and per-node simplification. Cignoni et al. [5] later used a geometry-based multi-resolution structure for out-of-core data management, where a hierarchy of tetrahedra is constructed by recursively partitioning the input models. Gobbetti and Marton [15] represented data with a volume hierarchy, by which their approach tightly integrated the algorithms of LOD, culling and out-of-core for massive model rendering.

3. GPU-accelerated mesh simplification

Most simplification algorithms are not naturally data parallel and do not have trivial GPU implementations. Traditional simplification algorithms rely on hierarchical data structures. The inter-dependencies introduced between levels of the hierarchy make such algorithms unable to be implemented in parallel. To address this problem, similar to [26], we present a dependency-free approach that supports continuous LOD on GPUs.

3.1. GPU-friendly preprocessing

Our approach originates from the idea of edge collapsing. The order of edge-collapsing operations indicates how details of an object are reduced. These operations are recorded in an array structure, called *ECol*. Each element in *ECol* corresponds to a vertex, and its value is the index of the target vertex that it collapses to. The storage of vertices and triangles is rearranged based on the order of the operations. In practice, the first removed vertex during collapsing is put at the last position in the vertex set; and the last removed vertex is put to the first position. The same rearrangement is applied to the triangle set. If a coarse version of the model is needed, we select a small number of continuous vertices and triangles starting from the first element in the sets. Fig. 2 gives an example of the preprocessing with a mesh composed of 7 vertices and 8 triangles. Let's define $ecol(i)$ that returns the value of theith element of *ECol*. Map is an array to store the vertex count and triangle count remaining after each collapsing operation. If j is the remaining vertex count, the value of jth element in *Map* returned by $map(j)$ is the remaining triangle count. As shown in Fig. 2(a), *Map* is initialized so that $map(7) = 8$.

To preserve visual quality, we restrict boundary edges are non-collapsible. A *boundary edge* is the edge existing only in one triangle. Two vertices of this edge are *boundary vertices*. We use Q to denote the set of boundary vertices. As a result, the lowest LOD of an object is represented with the triangles formed by boundary vertices, rather than a single triangle. Equation 1 expresses general rules to collapse an edge (v_a, v_b). N/A indicates that the edge is a boundary edge.

$$Collapse(edge) = \begin{cases} v_a \rightarrow v_b, (v_a \notin Q) \\ v_b \rightarrow v_a, (v_a \in Q, v_b \notin Q) \\ N/A, (v_a, v_b \in Q) \end{cases}, \text{where } a < b \tag{1}$$

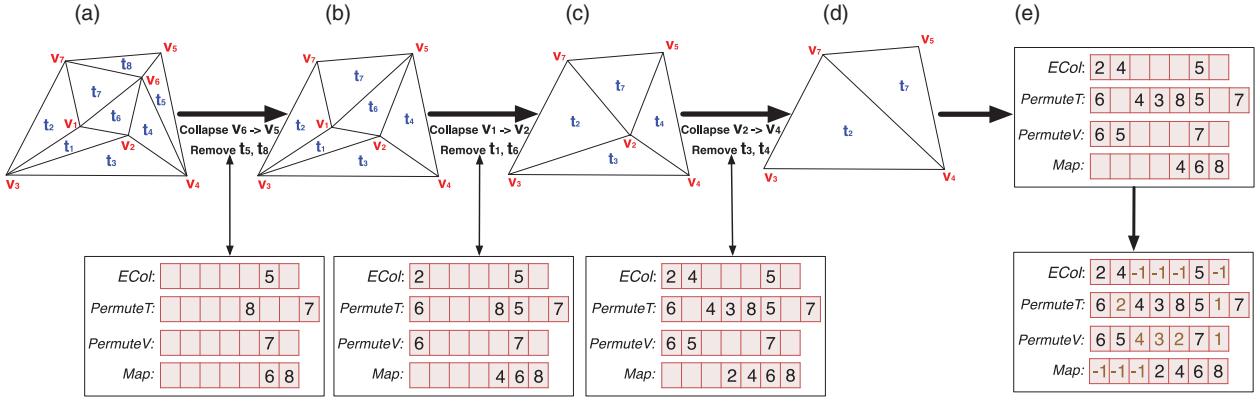


Figure 2. A preprocessing example of collapsing-based simplification. (a)-(d) show the intermediate meshes resulted by edge-collapsing operations. At each operation, the collapsing-related information is recorded into four array structures. (e) shows the final array structures after handling the boundary vertices.

There are five steps at each collapsing operation detailed as follows:

(1) **Choosing an edge to collapse.** We calculate the cost value for each edge that indicates the rate of visual changes if collapsed. Our cost function is similar to the function introduced by Melax [25] for game development, but we set the cost values of boundary edges are infinitely large. We collapse the edge evaluated with the minimal cost by merging one vertex, denoted as v_{src}, to the target one, denoted as v_{tar}, using Equation 1. This collapsing information is recorded in *ECol* such that $ecol(src) = tar$.

(2) **Permutation for vertices.** Let's say m is the number of vertices in the currently operating version of the mesh. *src* is replaced by m, which indicates v_{src}will be restored to the *mth* position in the rearranged vertex set. To maintain such information, we use array*PermuteV*, where each element represents a vertex index, and its value is the new index that it is permuted to. We denote $permuteV(src) = m$ to access the value at *src* in the array.

(3) **Vertex-triangle counts.** All triangles containing both $v_{src}$and $v_{tar}$are removed from the mesh. If $k$triangles are removed and n triangles remain, *Map*is updated such that$map(m-1) = n-k$.

(4) **Permutation for triangles.** Let's say n is the number of remaining triangles in the current operation, and two triangles, t_{r1} and t_{r2}, are removed, where $r1$ and $r2$ $(r1 < r2)$ are the indices of triangles. Similar to Step (2), we use an array called*PermuteT*, where each element represents the index of a removed triangle, and the value of the element is the new index. We denote functions of $permuteT(r1) = n$ and $permuteT(r2) = n-1$ to access the new indices, respectively.

(5) **Cleaning up.** We delete v_{src}, t_{r1} and t_{r2}, then update m and n for next iteration of collapsing operation.

Algorithm 1 :Data rearrangement

Input: V, T, *ECol*, *PermuteV*, *PermuteT*
Output: V',T',$ECol'$

1: **for** each $v_i \in V$ **do**;
2: $ECol'[permuteV(i)] \leftarrow permuteV[ecol(i)]$;
3: $V'.v_{permuteV(i)} \leftarrow V.v_i$;
4: **end for**
5: **for** each $t_i \in T$ **do**
6: $T'.t_{permuteT(i)} \leftarrow T.t_i$;
7: **end for**

The vertex and triangle sets are rearranged based on the values in *PermuteV* and *PermuteT*. In the mesh, V is the vertex set $\{v_1, v_2, \ldots, v_p\}$ and T is the triangle set $\{t_1, t_2, \ldots, t_s\}$. The rearranged object is defined as $O' = (V', T')$. our data rearrangement algorithm is described in Algorithm 1. Fig. 3(a) is an example showing how *ECols*, $V$and T are rearranged. As a result, if a simplified version of O'contains $i$vertices $(i \in [q, p])$, where q is the number of boundary vertices and p is the total number of vertices, we can retrieve the required number of triangles $j = map(i)$. As a result, the desired shape of the simplified object is generated with only a subset of V'that is $\{v_1, v_2, \ldots, v_i\}$ and a subset of T' that is $\{t_1, t_2, \ldots, t_j\}$.

3.2. Triangle reformation

A simplified object is generated by using the selected portion of vertices and triangles. With the rearranged data, the desired portions can be quickly identified as long as we know the desired number of vertices and triangles. After the desired data portions are sent to GPU, we generate a new version of the model by reshaping each triangle. We call this process as *Triangle Reformation*. Of course the renderer would be able to rasterize those GPU-residing vertices and triangles without applying the triangle reformation algorithm. But that would make the object fragmented. To preserve objects' visual appearance, shapes of selected triangles need to be reconfigured using selected vertices.

Algorithm 2 Triangle Reformation

Input: $\overline{T}$, n, $ECol$
Output: $\overline{T}'$

1: **for** each $t_k \in \overline{T}$ **in parallel do**
2: **for** $i = 1$ to 3 **do**
3: $vidx \leftarrow$ the i *th* vertex index of t_k;
4: **while** $vidx > n$ **do**
5: $vidx \leftarrow ecol(vidx)$;
6: **end while**
7: replace i *th* vertex index with $vidx$;
8: **end for**
9: **end for**

Given a trianglet_k, its three vertex indices may not be within the range of selected vertices, so they must be replaced with indices from the range. We denote $\overline{T}$ to be the set of selected triangles, n is the desired number of vertices, and $\overline{T}'$is the reformed $\overline{T}$. As shown in Algorithm 2, the reformation algorithm is executed in parallel. One GPU thread handles one triangle. For each of three vertex indices, we replace it with a new target vertex index by looking up *ECol*. Fig. 3(b) is an example demonstrating the reforming process. Fig. 4 shows five versions of Boeing 777 model generated by the triangle reformation algorithm.

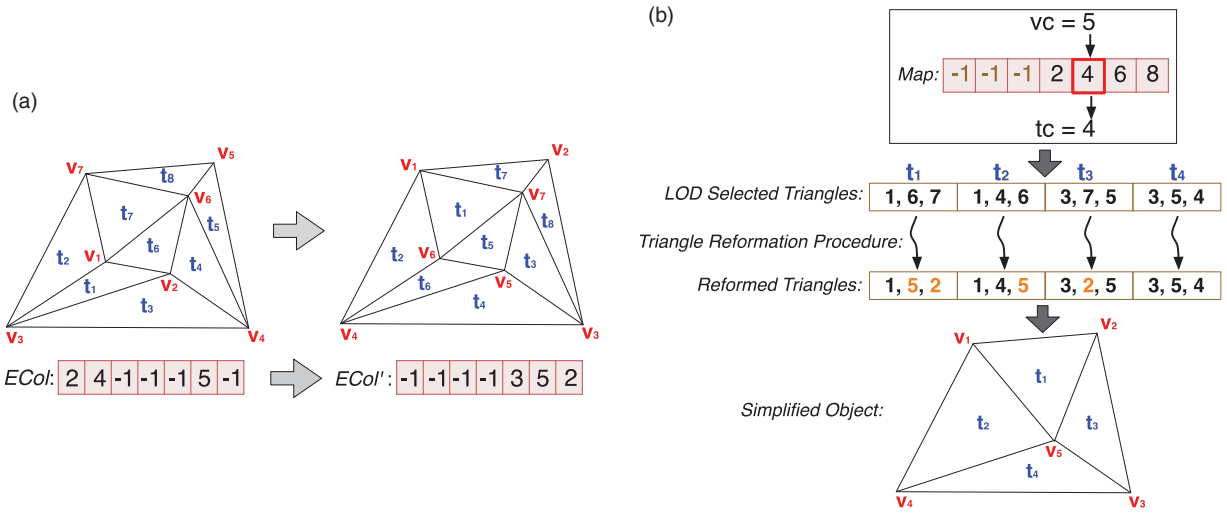


Figure 3. An example of data rearrangement. (a) shows vertices and triangles of the original object (left) with *ECol* of Fig. 2, which is rearranged to a new representation (right) with the rearranged *ECol'* after Algorithm 1. (b) shows the process of triangle reformation. Assuming the desired vertex count is $vc = 5$ for the example object of Fig. 2(a). By looking up corresponding elements in *Map*, the triangle count is $tc = 4$. Selected triangles are reformed using Algorithm 2. The result object has the same shape as Fig. 2(c).

Figure 4. Five levels of detail of Boeing 777 model. The numbers of triangles and vertices (triangle/vertex) are: (a) 38.0M/25.0M; (b) 26.6M/17.5M; (c) 15.1M/10.0M; (d) 7.4M/5.0M; (e) 3.7M/2.5M.

Our algorithm enables massive parallelization on GPU, in which each GPU thread is assigned with a single triangle rather than an object. Of course, object-level parallelization could be a trivial scheme because objects in the model are individual design parts and do not connect to each other, but that would cause GPU computation resources to be underutilized. The number of GPU threads launched would be equal to the number of objects, which is far less than the number of triangles. The reason we can have triangle-level parallelization is due to the use of *ECols*. Note that each object has an *ECol*. To reform a triangle, we first identify which object it belongs to. This can be easily done with the result of LOD selection - binary searching the array that stores the desired number of triangles of all objects (see Section 4). Then the GPU thread having this triangle looks up the corresponding *ECol* as described in Algorithm 2 (line 2–8).

3.3. GPU out-of-core

During runtime, before generating the simplified model, the selected portion of vertices and triangles are transferred from CPU to GPU, known as *GPU Out-of-Core* technique. Since the bandwidth of the PCIe bus is limited, CPU-to-GPU data transfer usually is a major performance bottleneck. For instance, let's say a GPU device is configured with 4GB memory. When transferring 4GB data to this GPU through PCIe 2.0 bus, it takes 250 ms. Let's say the target frame rate is 20 frames per second, meaning one frame must be rendered within 50 ms. Clearly, the bandwidth of PCIe bus is too small. Frame-to-frame coherence technique, such as [7, 26, 31], are used to identify data difference between frames. In a walkthrough application, there is always some data used for rendering current frame able to be used for next frame. Transferring only frame-difference (additional data) reduces the workload of PCIe bus. In this work, similar to our previous work [26], we employ a frame-to-frame coherence approach that minimizes the latency of data transfer. Because vertices and triangles have been rearranged in increasing orders of detail, the additional data is made by continuously selecting data sets from the repository on CPU main memory. For the *ith* object of the model, the set of frame-different vertices is $\Delta\overline{vc}_i^f = \overline{vc}_i^f - \overline{vc}_i^{f-1}$, where $\overline{vc}_i^f$ is the selected number of vertices, f is the current

frame, $f-1$ is the previous frame. If $\overline{vc}_i^f \leq 0$, no additional vertices are needed; otherwise, the set of vertices $\{\overline{vc}_i^{f-1}, \overline{vc}_i^{f-1}+1, \overline{vc}_i^{f-1}+2, \ldots, \overline{vc}_i^f\}$ is selected and sent to the GPU.

After the GPU receives the data, we need a reassembling procedure to combine the new data set and the old one. Between two successive frames, some objects' detail may increase, and others may decrease. If we fill new data into the blank memory blocks released by those objects whose detail decreases, the entire data buffer would be fragmented, since those blocks are small, many and unsorted. Fragmented data hurts rendering. The standard graphics pipeline supports high-performance rendering by taking the driver's hints about primitive usage patterns, for example, OpenGL's non-immediate-mode rendering method with Vertex Buffer Object (VBO). To use VBO feature for fast rendering, data has to be organized in the same appearance order as they are originally stored. Note that vertices, triangles and surface normals are stored and rasterized based on the VBO structure, and our simplification algorithm retains the VBO structure while reducing geometric complexity. Thus, the reassembling procedure must meet the VBO structure. This can be done in parallel on GPU. Each GPU thread handles one target element of the destination memory block, where the element will be filled with either an element of old data or an element of additional data. We first identify the parent object that the target element belongs to by binary-searching the prefix-summed $\overline{vc}$. Second, we convert the index of the element into a local index by subtracting the offset indicated in $\overline{vc}$. If this local index is smaller than the size of old data, the target element is filled with the element in old data; otherwise, it is filled with a new element.

Once the reassembling procedure is done, triangles and vertices stored in the destination memory block are the data for next frame. They are just a portion of original dataset selected from the CPU repository according to a few LOD criteria (see details in Section 4). Vertex indices in those triangles are original indices used to represent the full detail, so they are then replaced with new ones using Algorithm 2 to construct simplified versions of objects.

4. LOD selection

Now, the question is how to determine the desired levels of detail. Given a specific viewpoint, we need to find out how many vertices and triangles should be selected from the data repository. CAD models manage 3D data in a multi-object manner, where each object contains a small number of triangles and vertices to describe its geometric shape. To have a complete description of the whole model, hundreds of thousands of objects may be created. Conceptually, an object can be rendered at any level of detail. But, as we know, GPU memory size is insufficient to hold the entire dataset. Thus, the total number of geometry primitives must be budgeted according to the available GPU memory. Also, the criteria of allocating budget to objects must be well designed to satisfy the requirement of visual quality. In this section, we present a two-pass algorithm that effectively distribute hardware-constrained primitive budget.

4.1. First-pass algorithm

We should guide LOD selection with reference to the system of human's visual perception. As an object moves away from the viewer, less detail of the object is captured. If we can predict what detail people can perceive, we can remove imperceptible detail so that the computational resources will not be wasted for rendering unnecessary mesh details. In practice, people examine the projected area of the object on the display screen. A larger area results in a higher level of detail. We employ Equation 2 for the first-pass algorithm.

$$vc_i = N\frac{w_i^{1/\alpha}}{\sum_{i=1}^{m} w_i^{1/\alpha}}, \text{ where } w_i = \beta\frac{A_i}{D_i}P_i^{\beta}, \beta = \alpha - 1 \tag{2}$$

N represents the total number of vertices on GPU (the budget that is usually determined by hardware limitations). vc_i is the number of vertices for the *ith* object, computed out of total m objects. A_i represents the projected area on the screen. The exponent $1/\alpha$ is a factor to estimate the object's contributions for model perception. We set it to 3 in our experiments. D_i is the distance between the object and the camera viewpoint. P_i is the original number of geometry primitives. During the preprocessing step, as mentioned in Section 3.1, the remaining number of triangles after each edge-collapsing operation is recorded. Thus, during the runtime, after computing vc_i, we can retrieve the corresponding number of triangles, which would be used to generate the final simplified version of the object.

In an object, the disconnected faces separated by borders and holes are important visual features. We preserve those important features by restricting boundary edges are non-collapsible, and the simplest version of the object is made by boundary vertices. Let's define the set of boundary vertices for the *ith* object as Q_i. Then vc_i has a lower bound value, which is the size of Q_i. vc_i also has an upper bound value, which is the original number of vertices of the object.

We denote min_i as the size of Q_i and max_i as the original number of vertices. According to the result of Equation 2, if vc_i is not in $[\min_i, \max_i]$, we are not able to generate a simplified version of the object.

$$\overline{vc}_i = \begin{cases} vc_i & (vc_i \in [\min_i, \max_i]) \\ \max_i & (vc_i > \max_i) \\ \min_i & (vc_i/\min_i \in [MinT, 1]) \\ 0 & (vc_i/\min_i < MinT) \end{cases} \quad (3)$$

We employ Equation 3 to map vc_i into the range of $[\min_i, \max_i]$. Here we introduce a parameter called *MinT* ($MinT \in (0, 1)$), which is the threshold at the lower bound of vertex count. In the case of $vc_i < \min_i$, *MinT* is used to evaluate how close $vc_i$is to the min_i. If $vc_i/\min_i < MinT$, we ignore the object's visual contribution and set vc_i to zero; otherwise, we set it to $\min_i$, which is rendered with the simplest representation.

4.2. Second-pass algorithm

With Equation 3, $\overline{vc}_i$ is guaranteed to be a valid value. But if we add up all vertex counts, the sum (denoted as N') may not match the given budget N. If many objects have $vc_i > \max_i$ or $vc_i/\min_i < MinT$ ($vc_i \neq 0$), N' is smaller than N, then renderer does not obtain the expected amount of data, which means we waste computational resources. If most of objects have $vc_i/\min_i \in [MinT, 1]$, more than expected data are added, and renderer may receive data exceeding memory limit.

Our second-pass algorithm, as shown in Algorithm 3, adjusts geometry complexity of objects by redistributing remaining budget (when $N' < N$), or further reducing it (when $N' > N$). As a result, renderer is guaranteed with exact workload as specified in$N$. We describe the algorithm for the case of $N' < N$. Below are three steps to redistribute the remaining budget:

(1) **Sorting key-value pairs.** We distribute the remaining budget ($\Delta N = N - N'$) to visually important objects. In Equation 2, w_i defines the weights of visual importance of the objects. We define the set of key-value pairs $L = \{< weight, idx >\}$ ($idx \in [0, m)$), where *idx* is an object index, and *weight*is the value retrieved from w_{idx}. We sort L according to *weights* to move visually important objects to the front.

(2) **Identifying candidates.** We identify candidate objects whose details should be increased by obtaining a piece from ΔN. We first compute the number of vertices that an object can increase by. We introduce array *VDif*, where $VDif = \max_j - \overline{vc}_j$, i represents the *ith* element of sorted L, and $j = L_i.idx$. We perform prefix sum operation over *VDif*. We then employ a binary search procedure over *VDif*, so that we can find k objects obtaining pieces from ΔN, by satisfying $VDif_k \leq \Delta N < VDif_k + 1$.

(3) **Redistributing ΔN to the candidates.** We increase the number of vertices and triangles of those candidates. The final complexity of objects will match the given budget N.

Algorithm 3 Second-Pass

Input: L, $\overline{vc}$, max, N', N
Output: $\overline{vc}$

```
1:  if N' < N then
2:    sort L in decrease order of L.weights in parallel;
3:    for i th element in L in parallel do
4:      j ← L_i.idx;
5:      VDi ← max_j − vc̄_j;
6:    end for
7:    prefix sum then binary search VDif in parallel;
8:    for i th element in k selected objects in parallel do
9:      j ← L_i.idx;
10:     vc̄_j ← max_j;
11:   end for
12: end if
13: if N' > N then
14:   sort L in increase order of L.weights in parallel;
15:   for i th element in L in parallel do
16:     j ← L_i.idx;
17:     VDi ← vc̄_j;
18:   end for
19:   prefix sum then binary search VDif in parallel;
20:   for i th element in k selected objects in parallel do
21:     j ← L_i.idx;
22:     vc̄_j ← 0;
23:   end for
24: end if
```

If $N' > N$, we use a similar algorithm. But we construct *VDif* differently, where $VDif_i = \overline{vc}_j$, and $j = L_i.idx$. So *VDif* indicates the number of vertices reduced from objects. After applying the prefix sum operation and binary search, we reduce the $\overline{vc}$ of those selected objects to zero, since their *weights* indicate that they are the least important object in visual appearance. With the number of selected vertices, we find the corresponding number of selected triangles $\overline{tc}$, where $\overline{tc}_i = map(\overline{vc}_i)$.

5. Experimental results

We implemented the experimental software application on a 64-bit Windows system using C++, OpenGL and NVIDIA CUDA 6.0 SDK. We used Boeing 777 airplane

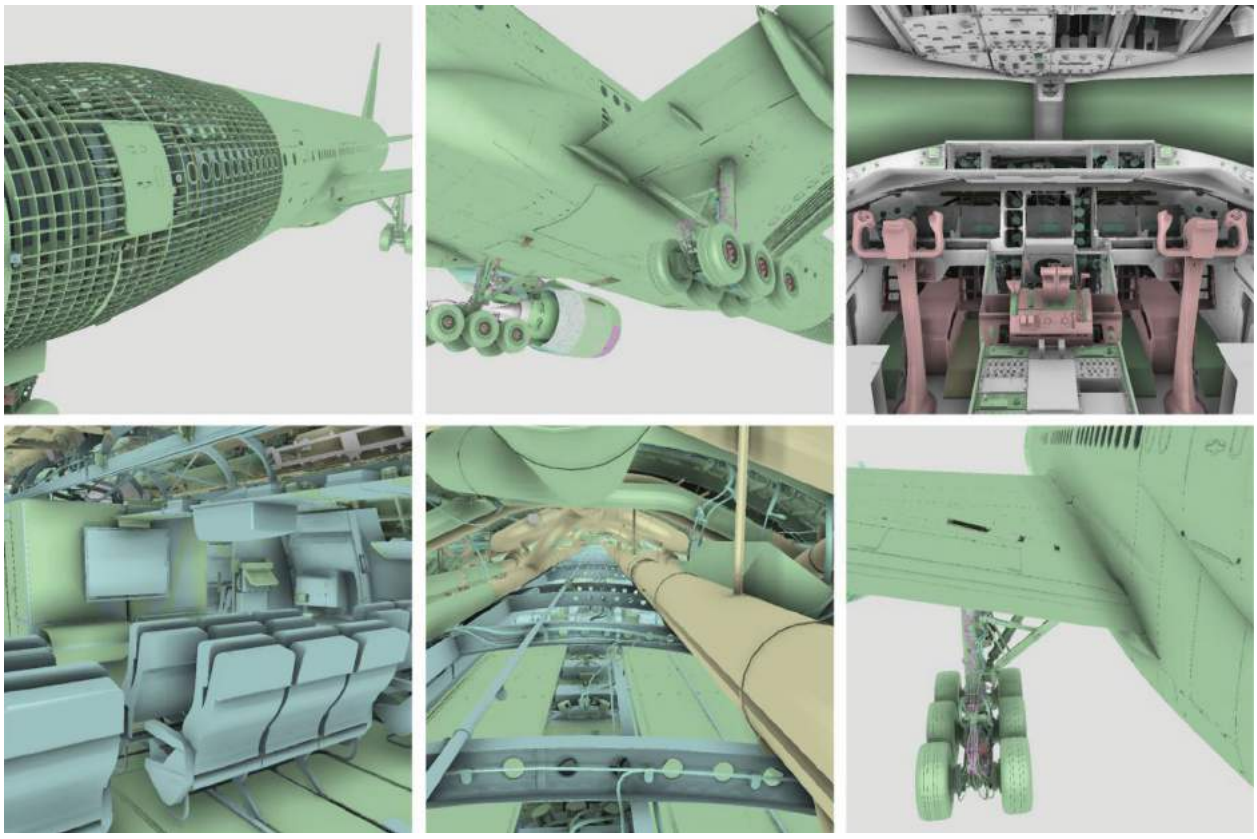


Figure 5. Boeing 777 model rendered by our approach.

model, which is an exceptionally complex CAD model. It consists of many loosely connected, badly tessellated, intertwining detailed objects with widely varying spatial ratios and complex topologies, as shown in Fig. 5. It has 718 thousand objects containing 332 million triangles and 223 million vertices. We set *MinT* to 0.65 in our experiments. We are confident that the complexity of the Boeing airplane model sufficiently represents the benchmarks of our target application domains.

5.1. Preprocessing performance

The preprocessing stage is performed to record the edge-collapsing information and rearrange data storage, where edges are collapsed one-by-one until reaching the simplest version of the object. On average, the preprocessing performance is 5.0 K triangles/sec, which is slightly slower than our previous work [26] due to the extra calculations over boundary vertices. Derzapf and Guthe [8] constructed the bounding volume hierarchy at 50 nodes/min. Yoon et al. [34] generated the CHPM structure at 3.0 K triangles/sec; Cignoni et al. [5] constructed the multiresolution-based static LODs at 30k triangles/sec on the network system with 16 CPUs; Gobbetti et al. [15] built the volume hierarchy at 1 K triangles/sec on one CPU and 20k triangles/sec on 16 CPUs. Our approach requires extra memory to store collapsing information (*ECols*). For the Boeing airplane model, 582.5MB is needed to store *ECols*, which is only 8.7% of the model size (6.7GB). *ECols* are stored on GPU memory during entire runtime, so that the overhead of sending them from CPU to GPU is avoided. Derzapf and Guthe [16] preprocessed St. Matthew model (372M triangles) requiring 2GB additional memory to store the nodes of the bounding volume hierarchy, which is 23.9% of the original size of St. Matthew model. They kept the nodes on hard drive and sent selected nodes along with the selected geometries to CPU main memory, and then to GPU memory. This was a time-consuming out-of-core data management.

5.2. Runtime performance

We evaluated runtime performance on a workstation equipped with an Intel(R) Core(TM) i7 2.67 GHz CPU (4 cores), 12GB RAM, PCIe 2.0×16 and a NVIDIA GTX 680 with 4GB GPU memory. We created a camera navigation path. With the two-pass LOD selection algorithm, we can precisely specify the desired geometry complexity of the model by giving the value of N. Fig. 6 plots the

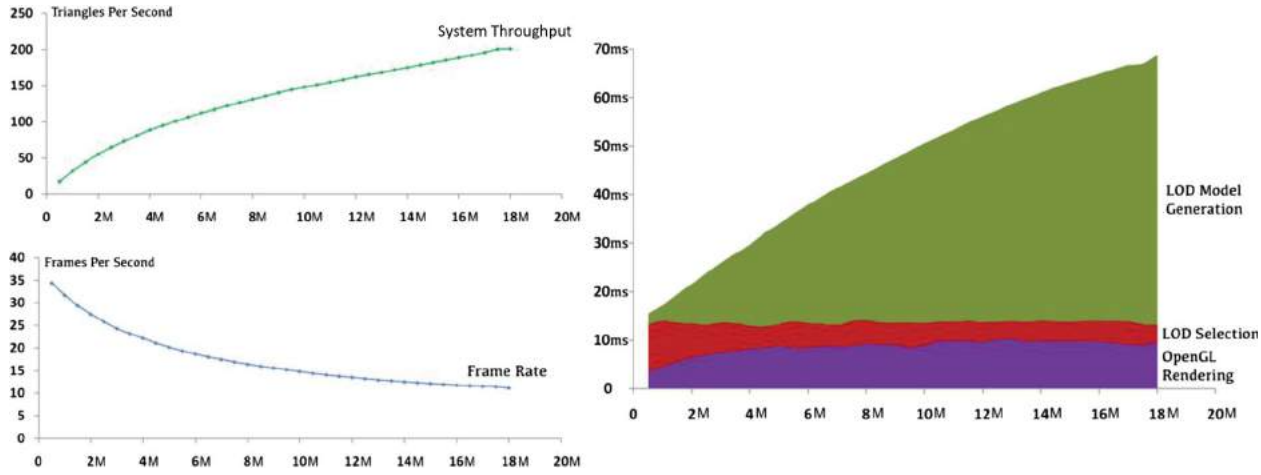


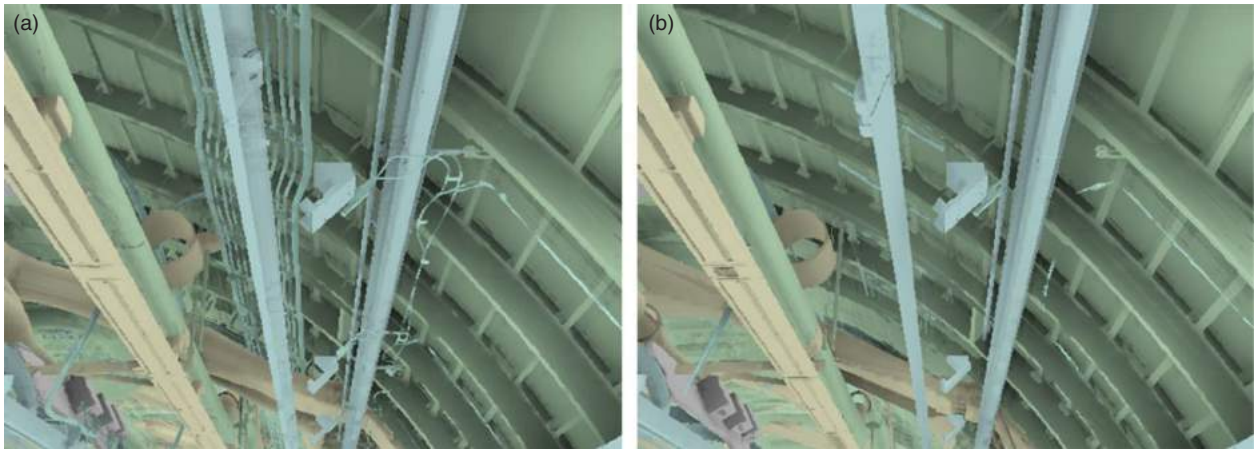
Figure 6. Performance with NVIDIA GTX 680.

relation between the overall performance and values of *N*. Each dot in graphs represents the averaged values from all rendered frames. A larger value of *N*gives a finer data representation that ensures the accuracy but decreases performance. Cignoni et al. [5] made an average of 70M triangles/sec using their TetraPuzzle approach. Gobbetti et al. [15] sustained an average of 45M primitives/sec with their far voxel approach. Derzapf and Guthe [8] achieved 180M triangles/sec for in-core implementation and 140M for out-of-core implementation. The system throughput of our implementation is a function of *N*. It is 205M triangles/sec with *N* equal to 18M.

Table 1. Performance breakdown.

N	FPS	LOD Selection	LOD Model Generation	OpenGL Rendering
18.0M	11.5	11.4 ms (13.0%)	69.5 ms (79.5%)	6.6 ms (7.5%)
10.5M	14.4	11.5 ms (16.5%)	51.4 ms (74.0%)	6.6 ms (9.5%)
3.0M	24.3	11.5 ms (27.9%)	23.8 ms (57.8%)	5.9 ms (14.3%)

Table 1 and the graph on the right in Fig. 6 show breakdowns of performance. The time on *LOD Selection* scales with the number of objects. We use *Axis-Aligned Bounding Boxes* (AABBs) of the objects for LOD selection. At each time a frame being rendered, all AABBs are used to determine desired geometry complexity. Theoretically, the execution time of *LOD Selection* is constant regardless the value of *N*. The other two components scale linearly to the values of *N*. The areas in the graph represent the averaged execution times from the entire rendered frames with different values of *N*. The time of *LOD Selection* scales with the number of objects, so it does not vary with different values of *N*. Because of the limitation of PCIe's bandwidth, transferring data from CPU to GPU is a major performance bottleneck. Note that only triangles and vertices are CPU-to-GPU transferred. Other types of data such as *ECols* and *Maps* are stored on GPU during the runtime. With the camera path in our experiments, 430 K triangles and 232 K vertices



Result with two-pass algorithm

Result with only the first-pass algorithm

Figure 7. Comparison of rendering quality. *N* is set to be 10.5M on both.

(frame-different data) per frame are transferred through the PCIe bus, but they still cause *LOD Model Generation* to be the most time-consuming component. With *N*equal to 18M, on average, data transfer time is 47.9 ms per frame, and the time for constructing the simplified model is 21.6 ms per frame. So in *LOD Model Generation* component, the processing of GPU out-of-core results in a more significant performance impact than the computations for constructing the simplified model. We employ OpenGL's VBO feature for rendering, which is never a major performance factor.

We also compare the result produced with our two-pass algorithm to the result with only the first- pass algorithm. As shown in Fig. 7, given the same value of *N*, the complexity adjustment strategy in the two-pass algorithm preserves the appearance detail, especially on those thin and small objects.

6. Conclusion

In this paper, we presented a parallel approach on GPU to interactively render complex 3D CAD models. Our LOD selection algorithm ensures full utilization of GPU memory. While achieving high performance, we maximize the details on a simplified version of the original model. Our approach also supports the tunable strategy for adaptive level-of-detail controls, which can satisfy different rendering constraints. Besides continuing to improve our implementation, we would like to explore other metrics that may deliver better performance and visual quality. Our algorithm assumes the input model contains multiple objects. We believe that, with a few changes, our approach should support the scanned surface model rendering. Also, we would like to explore different memory spaces of GPU devices. We think about using shared memory to further improve the performance. Of course, our rendering is handled by OpenGL with VBOs, which achieves best possible performance with the support of OpenGL driver for hardware acceleration; but triangle reformation and GPU out-of-core components may take advantages of shared memory. The restriction is that shared memory is visible only to threads within the block, and data there is accessible for the duration of the block. We would consider how to wisely assign triangles to GPU threads and how to handle latency caused by data transfer from global memory to shared memory at each time a frame being rendered.

Acknowledgements

This work was partially supported by the NSF grant, CNS 1464323, and the New Faculty Research Program of University of Alabama in Huntsville. We gratefully acknowledge the support of NVIDIA Corporation with the donation of GPU devices used for this research. We also thank anonymous reviewers for their suggestions and comments. We also thank David Kasik of Boeing for providing the 3D model of Boeing 777 airplane.

ORCID

Chao Peng http://orcid.org/[0000-0001-8838-2469]
Yong Cao http://orcid.org/[0000-0001-7422-8284]

References

[1] Aliaga, D.; Cohen, J.; Wilson, A.; Baker, E.; Zhang, H.; Erikson, C.; Hoff, K.; Hudson, T.; Stuerzlinger, W.; Bastos, R.; Whitton, M.; Brooks, F.; Manocha, D.: MMR: an interactive massive model rendering system using geometric and image-based acceleration, Proceedings of Symposium on Interactive 3D Graphics (I3D '99), ACM, New York, NY, USA, 1999, 199–206. http://dx.doi.org/10.1145/300523.300554

[2] Brüderlin, B.; Heyer, M.; Pfützner, S.: Visibility-guided rendering for real time visualization of extremely large data sets, Tutorial on European Conference on Computer Graphics (Eurogaphics'06), Eurographics Association and Wiley Blackwell, Vienna, Austria, 2006. http://www.sci.utah.edu/ ∼ abe/massive06/EG06-Beat.pdf

[3] Chen, J.: Gpu technology trends and future requirements, Electron Devices Meeting (IEDM), IEEE International, Baltimore, MD, USA, 2009, 1–6. http://dx.doi.org/10.1109/IEDM.2009.5424433

[4] 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. http://dx.doi.org/10.1111/1467-8659.00698

[5] Cignoni, P.; Ganovelli, F.; Gobbetti, E.; Marton, F.; Ponchio, F.; Scopign, R.: Adaptive tetra- puzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models, ACM Transactions on Graphics (TOG), 23(3), 2004, 796–803. http://dx.doi.org/10.1145/1186562.1015802

[6] Correa, W.; Klosowski, J.; Silva, C.: Visibility-based prefetching for interactive out-of-core rendering, Proceedings of Symposium on Parallel and Large-Data Visualization and Graphics (PVG'03), IEEE Computer Society, Seattle, Washington, USA, 2003, 2. http://dx.doi.org/10.1109/PVG.2003.10002

[7] DeCoro, C.; Tatarchuk, N.: Real-time mesh simplification using the GPU, Proceedings of Symposium on Interactive 3D Graphics and Games (I3D '07), ACM, New York, NY, USA, 2007, 161–166. http://doi.acm.org/10.1145/1230100.1230128

[8] Derzapf, E.; Guthe, M.: Dependency-free parallel progressive meshes, Computer Graphics Forum, 31(8), 2012, 2288–2302. http://dx.doi.org/10.1111/j.1467-8659.2012.03154.x

[9] Derzapf, E.; Menzel, N.; Guthe, M.: Parallel view-dependent refinement of compact progressive meshes, Eurographics Symposium on Parallel Graphics and Visualization (EG PGV'10), Eurographics Association,

Aire-la-Ville, Switzerland, 2010, 53–62. http://dx.doi.org/10.2312/EGPGV/EGPGV10/053-062

[10] Derzapf, E.; Menzel, N.; Guthe, M.: Parallel view-dependent out-of-core progressive meshes, Proceedings of the Vision Modeling and Visualization Workshop (VMV'10), Eurograohics Association, Siegen, Germany, 2010, 25–32. http://dx.doi.org/10.2312/PE/VMV/VMV10/025-032

[11] Funkhouser, T. A.; Séquin, C. H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments, Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93), ACM, New York, NY, USA, 1993, 247– 254. http://dx.doi.org/10.1145/166117.166149

[12] Garland, M.; Heckbert, P. S.: Surface simplification using quadric error metrics, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97), ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, 1997, 209–216. http://dx.doi.org/10.1145/258734.258849

[13] Garland, M.; Zhou, Y.: Quadric-based simplification in any dimension, ACM Transactions on Graphics (TOG), 24(2), 2005, 209–239. http://dx.doi.org/10.1145/1061347.1061350

[14] Giegl, M.; Wimmer, M.: Unpopping: Solving the image-space blend problem for smooth discrete lod transitions, Computer Graphics Forum, 26(1), 2007, 46–49. http://dx.doi.org/10.1111/j.1467-8659.2007.00943.x

[15] Gobbetti, E.; Marton, F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms, ACM Transaction on Graphics (TOG), 24(3), 878–885. http://dx.doi.org/10.1145/1073204.1073277

[16] Gu, X.; Gortler, S. J.; Hoppe, H.: Geometry images, ACM Transactions on Graphics (TOG), 21(3), 2002, 355–361. http://dx.doi.org/10.1145/566654.566589

[17] Hollander, M.; Ritschel, T.; Eisemann, E.; Boubekeur, T.: Manylods: Parallel many-view level-of- detail selection for real-time global illumination, Computer Graphics Forum, 30(4), 2011, 1233–1240. http://dx.doi.org/10.1111/j.1467-8659.2011.01982.x

[18] Hoppe, H.: Progressive meshes, Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96), ACM, New York, NY, USA, 1996, 99–108. http://dx.doi.org/10.1145/237170.237216

[19] Hoppe, H.: View-dependent refinement of progressive meshes, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997, 189–198. http://dx.doi.org/10.1145/258734.258843

[20] Hu, L.; Sander, P. V.; Hoppe, H.: Parallel view-dependent refinement of progressive meshes, Proceedings of Symposium on Interactive 3D Graphics and Games (I3D '09), ACM, New York, NY, USA, 2009, 169–176. http://dx.doi.org/10.1145/1507149.1507177

[21] Ji, J.; Wu, E.; Li, S.; Liu, X.: Dynamic LOD on GPU, in Computer Graphics International, IEEE, Stony Brook, NY, USA, 2005, 108–114. http://dx.doi.org/10.1109/CGI.2005.1500386

[22] Lindstrom, P.; Turk, G.: Image-driven simplification, ACM Transactions on Graphics, 19(3), 2000, 204–241. http://dx.doi.org/10.1145/353981.353995

[23] Lindstrom, P.: Out-of-core simplification of large polygonal models, Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000, 259–262. http://dx.doi.org/10.1145/344779.344912

[24] Luebke, D.; Watson, B.; Cohen, J. D.; Reddy, M.; Varshney, A.: Level of Detail for 3D Graphics, Elsevier Science Inc., New York, NY, USA, 2002. http://lodbook.com/

[25] Melax, S.: A simple, fast, and effective polygon reduction algorithm, Game Developer, 1998, 44–49. http://dev.gameres.com/program/Visual/3D/PolygonReduction.pdf

[26] Peng, C.; Cao, Y.: A GPU-based approach for massive model rendering with frame-to-frame coherence, Computer Graphics Forum, 31(2pt2), 2012, 393–402. http://dx.doi.org/10.1111/j.1467-8659.2012.03018.x

[27] Popov, S.; Günther, J.; Seidel, H.-P.; Slusallek, P.: Stackless KD-tree traversal for high performance GPU ray tracing, Computer Graphics Forum, 26(3), 2007, 415–424. http://dx.doi.org/10.1111/j.1467-8659.2007.01064.x

[28] Swarovsky, J.: Extreme detail graphics, Proceedings of Game Developers Conference, 1999, 899–904. http://www.svarovsky.org/ExtremeD

[29] Toledo, S.: A survey of out-of-core algorithms in numerical linear algebra, External Memory Algorithms and Visualization, American Mathematical Society, Boston, MA, USA, 50, 1999, 161–179. http://dl.acm.org/citation.cfm?id = 327789

[30] Varadhan, G.; Manocha, D.: Out-of-core rendering of massive geometric environments, Proceedings of Visualization (VIS'02), IEEE, Boston, MA, USA, 2002, 69–76. http://dx.doi.org/10.1109/VISUAL.2002.1183759

[31] Wang, R.; Huo, Y.; Yuan, Y.; Zhou, K.; Hua, W.; Bao, H.: GPU-based out-of-core many-lights rendering, ACM Transactions on Graphics (TOG), 32(6), 2013, 210:1–210:10. http://dx.doi.org/10.1145/2508363.2508413

[32] Xia, J. C.; El-Sana, J.; Varshney, A.: Adaptive real-time level-of-detail-based rendering for polygonal models, IEEE Transactions on Visualization and Computer Graphics, 3(2), 1997, 171–183. http://dx.doi.org/10.1109/2945.597799

[33] Yoon, S.; Gobbetti, E.; Kasik, D.; Manocha, D.: Real-time massive model rendering, Synthesis Lectures on Computer Graphics and Animation, 2008, 1–122. http://dx.doi.org/10.2200/S00131ED1V01Y200807CGR007

[34] Yoon, S.-E.; Salomon, B.; Gayle, R.; Manocha, D.: Quick-VDR: Interactive view-dependent rendering of massive models, Proceedings of Visualization (VIS'04), IEEE, Austin, TX, USA, 2004, 131–138. http://dx.doi.org/10.1109/VISUAL.2004.86