Advances in Engineering Software 99 (2016) 73–80

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Efficient GPU out-of-core visualization of large-scale CAD models with voxel representations

Junjie Xue[a,c], Gang Zhao[a,b,c,*], Wenlei Xiao[a,b]

[a] School of Mechanical Engineering and Automation, Beihang University, Beijing, China
[b] Key Laboratory of Aeronautics Smart Manufacturing, Ministry of Industry and Information Technology, Beijing, China
[c] State Key Laboratory of Virtual Reality Technology and Systems, Beihang University, Beijing, China

ARTICLE INFO

Article history:
Received 23 December 2015
Revised 4 May 2016
Accepted 13 May 2016
Available online 24 May 2016

Keywords:
Voxel representation
Massive model rendering
GPU out-of-core
Visibility query
Geometry compression

ABSTRACT

Visualizing large-scale CAD models has been recognized as one of the most challenging tasks in engineering software development. Due to the constraints of limited GPU memory size and computation capacity, the CAD model of a complex product with hundreds of millions triangles cannot be loaded and rendered in real-time using most of modern GPUs. In this paper, an efficient voxel assisted GPU out-of-core framework is proposed for visualizing massive CAD models interactively. In order to reduce memory cost and improve efficiency of data streaming, a parallel off-line geometry attributes compression scheme is introduced to minimize the storage cost of each primitive by quantifying the LOD (levels of detail) geometries into a highly compact format. At the rendering stage, voxel representation is utilized to query visible objects by efficient ray casting algorithms, which is distinguishable from primitive or bounding box based visibility culling methods. The voxel representation is also utilized for shadow ray intersection test to generate soft shadow effect which results in enhancement of rendering realism. A prototype software system is developed to preprocess and render massive models with the proposed framework. Experimental results show that users can interactively visualize CAD models with hundreds of millions of triangles at high frame rates using our framework.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

With the development of digital design and manufacturing technology, modern engineering modeling and simulation applications have created numerous highly complex 3D models, including industrial CAD models of aircrafts, ships and production plants, etc. These models consume large amounts of storage space and are geometrically complex, which requires superior visual computing capabilities for real-time rendering. Although the hardware performance of commodity PCs has increased constantly according to Moore's law, it is still not sufficient to render these datasets at interactive frame rates using brute-force approaches.

There have been several studies conducted in the field of real-time massive model rendering. OpenRT system was able to render large models at several frames per second using a single commodity desktop PC by combining real-time ray tracing with out-of-core caching [1]. Some other research focus on utilizing parallel architecture to accelerate rendering process [2,3], such as GigaWalk and Manta system. The GigaWalk system uses two graphics pipelines with multiple processors, it was able to render CAD environments composed of tens of millions of polygons at interactive rates on an a SGI workstation [4]. Manta system employs a multi-threaded scalable parallel pipeline which takes advantage of a Itanium2 SGI supercomputer [5]. Therefore, the system was able to achieve a good rendering quality and interactive functions like transparency and clipping. More recently, voxel representations offered significant potential for massive model visualization [6–10]. The voxel based rendering method uses hierarchical voxel data to view-dependently represent the primitives in an approximate manner. Among them, Far-Voxels utilize cubical voxels to represent inner node of BSP tree [11], while R-LODs consists of a plane with material attributes [12]. More recently, VoxLOD uses a voxel with six shading attributes (colors and normals of six planes) to represent an inner Kd-tree node [13]. The voxels serve as a simplification of triangle primitives contained in a Kd-tree node. Although these studies have greatly improved rendering efficiency from many aspects, however, the rendering efficiency of large-scale CAD models on commodity PCs is still low, the refresh rates in investigated literatures are still low.

At rendering stage, GPU out-of-core algorithm can continuously stream coherent geometry data in host memory into GPU to

* Corresponding author.
E-mail addresses: bitxue@foxmail.com (J. Xue), zhaog@buaa.edu.cn (G. Zhao), xiaowenlei@buaa.edu.cn (W. Xiao).

http://dx.doi.org/10.1016/j.advengsoft.2016.05.006
0965-9978/© 2016 Elsevier Ltd. All rights reserved.

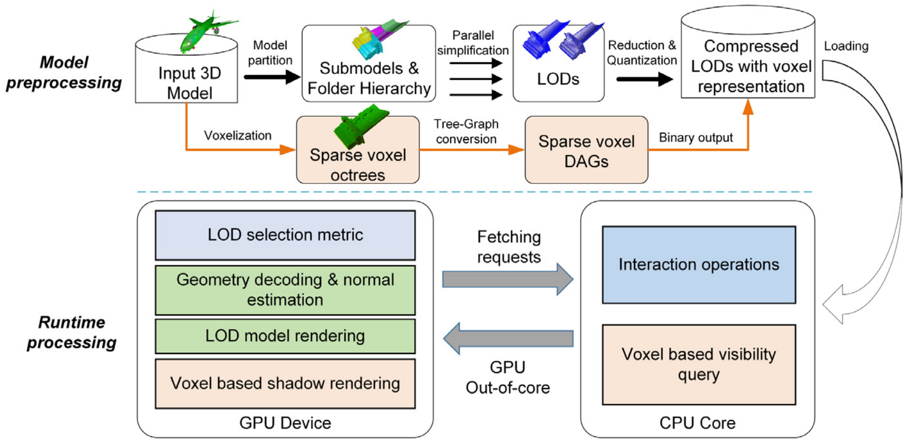

Fig. 1. The overview of our approach. The pipelines of model preprocessing and runtime processing are demonstrated.

provide necessary data for rendering each frame. This algorithm solves the issue that the massive CAD dataset cannot load into the GPU memory at once. However, due to the widening gap between data access speed and data computation speed [14], the efficiency of GPU out-of-core becomes the key factor affecting the rendering speed. Current rendering architectures using out-of-core algorithms rely on suitable data layout algorithms to reduce the memory latency and data fetch time from the disk-based secondary storage devices to the main memory. However, many factors can affect the efficiency of the out-of-core algorithms, such as the rate of memory, the storage size of individual primitives, and the primitive selection time for each rendering frame. Therefore, the out-of-core algorithms need to be optimized throughout the preprocessing and rendering pipelines.

This paper presents an efficient voxel assisted GPU out-of-core framework for real-time rendering of large-scale CAD dataset. The framework generates a sparse voxel representation of the CAD model and storages the voxel data into a compact graph structure. The LODs is bound to a bounding volume hierarchy (BVH) structure and the geometry data is compressed into a highly compact encoding format. In addition, LOD processing is integrated with voxel based visibility query using a GPU based approach to improve the runtime rendering performance. A prototype software system is implemented and tested on desktop PCs with two large CAD datasets (with 72∼337 million triangles).

2. Overview of GPU out-of-core framework

The overview of our voxel assisted GPU out-of-core framework is shown in Fig. 1. In order to accelerate the *model preprocessing*, we propose a parallel LOD generation algorithm utilizing file hierarchy of the model dataset. This algorithm treats each leaf node of the file hierarchy as an initial LOD geometry, and it uses multiple threads to simplify geometry in parallel and produce separate multi-resolution model files for each LOD geometry. Each LOD geometry is finally generated by combining each LOD level model file in order.

We also design an aggressive GPU-based geometry attributes compression scheme, which can encode the LOD geometry data with compression ratio higher than 5. In this compression scheme, the geometry vertex data is quantized from absolute coordinates to relative coordinates (relative to its AABB box), which are then encoded into 16-bit width *GLushort* data type. Moreover, all the geometry normal data is deleted to minimize the storage size. In order to fully utilize powerful computing capability of GPU, the encoded vertex data and normal data are decoded and reconstructed in the GPU graphics pipeline. The model preprocessing pipelines also generate a sparse voxel representation of the original model for efficient visibility query and shadow rendering in runtime processing. In order to reduce the storage size of high resolution voxel data, the voxel data is organized with an octree structure first and is then converted into a directed acyclic graph (DAG) structure.

The runtime processing takes advantage of both CPU and GPU computation capacity. The visibility query employs multithreaded sparse voxel ray casting algorithm with the voxel representation generated in the model preprocessing stage. A list of visible objects at current frame can be achieved after this query process. The LOD level of each object can be determined by LOD selection metric algorithm. And the model data which is needed to be transferred to or to be released from the GPU memory can be filtered by comparing the visible object list with that of last frame. The transferred LOD geometry data is then decoded in the GPU graphics pipeline. A deferred shading technique is utilized to get the final image where soft shadow effect is calculated with the voxel representation.

3. Model preprocessing

In this section, we present an algorithm to compute multi-resolution hierarchy for massive models (LODs). The algorithm proceeds in two steps: (1) the geometry attributes are compressed into a highly compact storage, (2) we build a sparse voxel representation of the original model and associate each object with it.

3.1. Geometry attributes compression

In order to reduce bandwidth cost and memory footprint for GPU out-of-core rendering, it is necessary to compress LOD geometry data into a compact size. For a comprehensive view of geometry compression, we refer the reader to [15,16]. Previous works regarding to geometry compression mainly focus on efficient encoding and decoding algorithms that compresses geometry data into a compact size with/without loss of precision. This paper introduces an *aggressive geometry compression* and *GPU based decompression* scheme which possesses higher compression ratio (Fig. 2). In this scheme, geometry normal attributes are deleted directly to minimize geometry storage size and is reconstructed in the GPU graphics pipeline, while others are quantized into a shorter bit width.

In this algorithm, an AABB box is calculated for each object, then each vertex coordinate is mapped into a relative coordinate. And each relative coordinate component is encoded from *GLdouble* or *GLfloat* data type into *GLushort* type. According to the size of vertex number, the index data is encoded into *GLubyte, GLushort* and *GLuint* type respectively. Typically normal data takes up the same amount of storage as the vertex data. While in this paper,

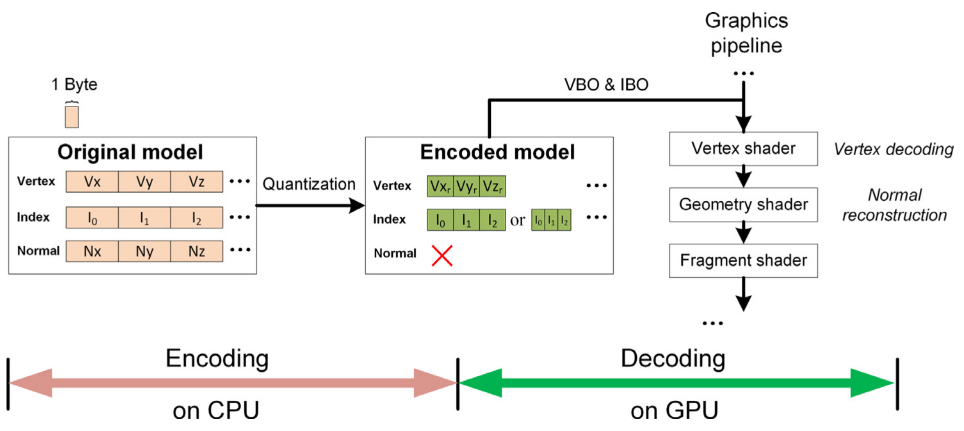

Fig. 2. Geometry attributes compression pipeline. The original absolute coordinate components of each vertex are encoded into relative coordinate components using *GLushort* data type; Index components are encoded from *GLuint* to *GLubyte* or *GLushort* type; The normal data is deleted. At the rendering stage, the vertex data is decoded in a vertex shader, then the normal data can be reconstructed using a geometry shader.

all the normal data is deleted in the preprocessing pipeline and intended to be recalculated on-the-fly in the GPU at the rendering phase. For homogeneous CAD models, generally each geometry has only one material attribute (textures are not considered when regarding to massive model). Therefore, one color information which will be used to compute the diffuse lighting results is stored for each geometry.

Geometry quantization is conducted at the end of our preprocessing pipeline. All the data is written into a binary format file to further reduce the storage size and accelerate the speed of loading LODs from extern storage to local memory at a subsequent rendering phase.

We use a vertex shader to decode the absolute coordinate of each vertex. Then normal reconstruction is implemented in a geometry shader to compute a normal vector for each vertex. Therefore, the decompression algorithm has taken advantage of the powerful GPU computing capacity. Assuming that $\mathbf{V}$ represents a vertex of the geometry, $\mathbf{M}_0$ and $\mathbf{M}_1$ are the min and max point of the axis-aligned bounding box of the geometry, then the algorithms of vertex data compression and decompression can be expressed as Eqs. 1 and 2 respectively:

$$\mathbf{V}_e = (2^n - 1)(\mathbf{V} - \mathbf{M}_0)./(\mathbf{M}_1 - \mathbf{M}_0) \quad (1)$$

$$\mathbf{V}_d = \mathbf{M}_0 + \frac{1}{(2^n - 1)}\mathbf{V}_e \circ (\mathbf{M}_1 - \mathbf{M}_0) \quad (2)$$

where $\mathbf{V}_e$ and $\mathbf{V}_d$ denote the encoded relative coordinate component and the corresponding decoded absolute coordinate component respectively. n is the bit width of the data type to store the relative coordinate components (since we use *GLushort* data type, n equals 16).

Rounding error is produced in the conversion process from absolute coordinate to relative coordinate, as shown in Eq. 1, which bring about the final compression error of vertex data. The error value is calculated by the distance between the decompressed vertex with the original vertex. The value of the compression error depends on the bounding box size of the compressed geometry as well as the data type used to store the relative coordinate components. Based on this, the compression error can be well controlled if a compatible combination of data type and geometry size is used. In order to have a small error value, large-sized geometry can be split into small ones by utilizing SAH (Surface Area Heuristic) model partition algorithm [17] while using the same bit width data type. The experimental result of the compression and decompression shows minor compression error with 16-bit vertex quantization compared with the original geometry (shown in Fig. 3).

3.2. Fast voxelization of large-scale mesh

Voxels can be generated from triangle meshes through voxelization algorithm. The algorithm firstly calculate a cubical bounding volume of the mesh, and then divide the volume into a uniform grid with a preset resolution along 3 axes. Each small cubical grid is called a voxel. With respect to each voxel, whether the voxel is empty or is occupied by the model is determined by a triangle-box intersection test. All the non-empty voxels form an approximation of the original triangle surface model, which are called sparse voxels. Typically each non-empty voxel contains position, material, and normal information, wherein the position can be given by grid resolution and index coordinates, the material and normal can be estimated by sampling the triangle mesh. In this paper, since voxels are only used for ray-voxel intersection test in visibility query and shadow rendering, these attributes do not need to be calculated.

Sparse voxel data management is mainly based on octree data structure. For sparse voxels without shading attributes, using DAG can reduce the storage size by an order of magnitude. We use a bottom-up approach to convert the sparse voxel octree into a directed acyclic graph. In order to lessen memory consumption, we employ block processing method: (1) first generates a low level of the octree; (2) each node generates a sub-tree of preset resolution, and convert the octree into a DAG; (3) after all subtrees have completed conversion, continue to convert the subgraphs upwards until we come to the root node, and finally a complete sparse voxel DAGs is generated. In order to improve the efficiency of voxel generation, the block processing phase is accelerated with multi-threading, the total voxelization time is shown in Table 1.

4. Voxel assisted GPU out-of-core rendering

In the previous section, an algorithm to generate compressed LODs is described. In this section, we present a voxel based GPU out-of-core rendering algorithm that combines the voxel based visibility query with view-dependent LOD refinement.

The core concept of the rendering algorithm is to determine a minimal scale of primitives to be transferred for each frame within the shortest time (Fig. 4). For current frame, a visible object set is determined by voxel based visibility query method, which casts rays from viewpoint towards voxels and test intersections using efficient sparse voxel DAGs ray casting algorithm; the LOD level for each object to be rendered is determined by a LOD selection metric. The resulted objects list is compared with the one in last frame: (1) the objects that were displayed in last frame but will

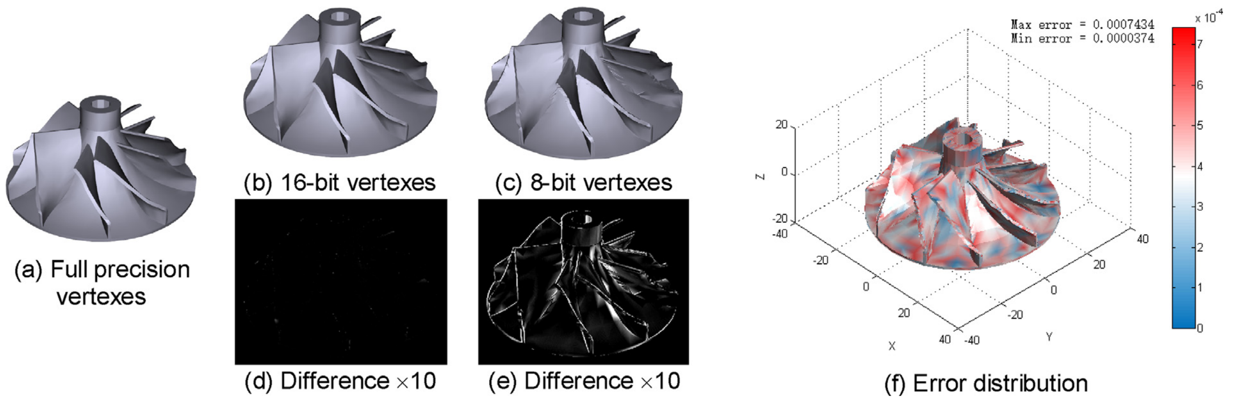

Fig. 3. Vertex quantization. Top row shows shaded result for quantized vertexes, with the difference to the ground truth below, multiplied by 10. The right figure shows compression error distribution of the tested CAD model. The model has a bounding box of [66,66,33] in size. The error value is estimated by the distance from the new vertex position to the original vertex position.

Table 1
Construction statistics for the test models. The number of triangles in the model, the size of the compressed LODs data structure and voxel data, the compression ratio, and the build time.

Models	Triangles (million)	Storage Size (GB)			LOD		Build time (minute)	
		Original	Compressed LOD	Voxels	compression ratio	Avg. Num. of LODs	LOD	Voxelization
Power Plants	72	9.4	1.8	0.3	5.24	3.7	20	12
Boeing 777	337	41.4	7.8	1.3	5.31	4.8	63	38

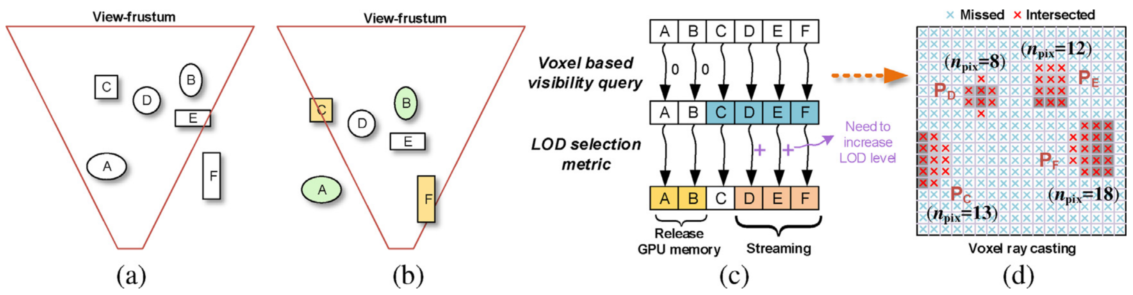

Fig. 4. The concept of primitive selection with integration of visibility query and LOD processing. (a) Camera position of last frame. (b) Camera position of current frame. (c) Primitive selection procedure. (d) Voxel based visibility query.

not display in current frame are deleted from GPU to release GPU memory, (2) the objects that are newly added to current frame or which LOD level is changed are requested to fetch from host memory.

4.1. Voxel based visibility query

Typically visibility culling contains frustum culling and occlusion culling. The objects outside of the view-frustum volume are excluded from the scene by frustum culling, and then occluded objects are removed through occlusion culling. GPU occlusion query uses the objects' bounding boxes as query objects. The bounding box is often too coarse to describe actual shape of an object. The bounding boxes of the objects may overlap or occlude with each other. This may bring about incorrect query results and the display distortion since objects which should be displayed are culled and not shown. Therefore, it is unsuitable for complex CAD model which may have perplexing occlusion relationship between parts. In this paper, we simplify frustum culling and occlusion culling as one procedure, to directly calculate visibility of current viewpoint with the sparse voxel representation. Unlike the method of culling invisible objects from the scene gradually, we tend to directly find the visible objects in the scene, as shown in Fig. 5.

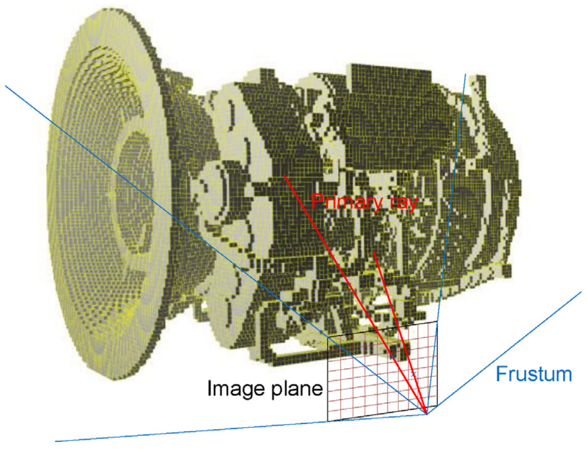

Fig. 5. Principle of voxel based visibility query.

Sparse voxels are built with uniform 3D grid, which means simpler rule can be used to storage voxels data. Therefore ray casting of sparse voxels can be implemented in parallel. According to our test result, the efficiency of ray casting is very high, about 200 million rays per second.

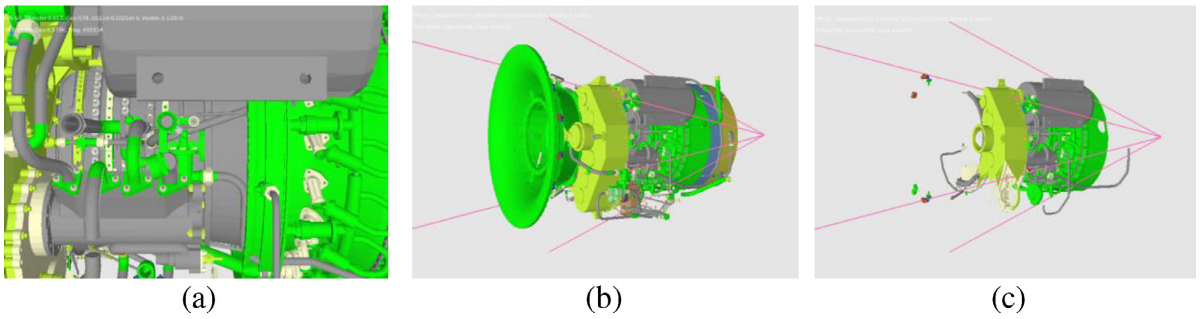
Fig. 6. The visibility query result on a gas turbine model. (a) The rendered frame. (b) A third person view from the side shows the objects rendered without visibility query. The pink lines indicate the view frustum. (c) The view shows the visible objects with visibility query.

The voxel based visibility query has taken advantage of the high efficiency of sparse voxels ray casting. As shown in Fig. 5, the image plane is divided into a uniform 2D grid, the primary rays is generated based on viewpoint and position of each sub grid. For each primary ray, we perform intersection test with the voxel representation, if the ray hit a voxel, we find the object corresponding to the voxel and add it to the visible object list, and update the count of the object been shot by rays. The accuracy of visibility query depends on the density of primary rays. To increase chance of small visible objects been shot by rays, we should increase the value of ray density. The visibility query result of a gas turbine model is shown in Fig. 6.

According to the experimental result, one ray per pixel is adequate to achieve better visibility test quality than traditional occlusion culling while costing almost equivalent time. We tested our voxel based visibility query algorithm vs. bounding boxes based occlusion culling algorithm and found that visible objects (which typically have 200 vertices or less) would be found to be occluded only 7% of the time with voxel based visibility query, vs. 38% with occlusion culling. 31% incorrect visibility test results were avoided. However, the high resolution voxels needs more memory space than coarse bounding boxes.

4.2. Integrating LOD with visibility query

After LODs are created, the main problem encountered is how to select an appropriate LOD level for each visible geometry at runtime. LOD level determines the number of triangles (or geometry complexity) to be rendered of a specific object. It not only controls the display quality of the object, but also the rendering time per frame by adjusting the total number of triangles to be rendered.

Most LOD adjustment methods use the viewpoint-object distance in object space or the projection area in image space as LOD level selection metrics. These metrics are effective for regular usage scenarios, such as terrain and urban scene rendering, etc. Nonetheless, for CAD models with extremely complex topologies and shapes, these methods may lead to improper LOD adjustment and affect the quality of display as well as the efficiency of rendering.

This paper presents a novel LOD selection metric based on voxel based visibility query. The *sampling count* (the count of an object been shot by rays) of each object can be gathered through the visibility query process (Fig. 4d). Unlike the classic occlusion queries, this algorithm uses sampling count other than the object's bounding volume to query the sampling result, thus exact sampling count can be obtained for each object. *Triangle density* is used as the selection metric of LOD levels, which is the ratio of the number of triangles to the sampling count of specific LOD geometry:

$$density = \frac{n_{\text{tri}}}{n_{\text{pix}}} \tag{3}$$

where n_{tri} and n_{pix} denote the number of triangle and sampling count of each object in current LOD resolution respectively.

With respect to a single object, *triangle density* is related to the distance and orientation between viewpoint and the object. With respect to multiple objects, it is related to the shape and size of each object. So it is a synthetical and accurate quantification of display quality for each visible object, and therefore it is able to cope with complex topologies and shapes in the massive CAD model.

The *triangle density* value is used to sort the objects in order to adaptively adjust LOD levels for the visible objects. For example, if we want to refine the scene objects, the object with lowest TPP value are firstly to be processed, and vice versa. The density metric method makes the rendering system select as fine as possible LOD models for rendering the visible objects, while ensuring the display quality of each object are basically the same. This avoids the LOD selection of some objects to be too fine or coarse.

The size of occlusion query list may be very large and querying the full list of LOD objects is time-consuming. We set up the maximum threshold of query count in a single frame. Objects which are close to the viewpoint or have large bounding volume take the precedence to be queried and added into the LOD adjustment list. The visible objects are sorted in accordance with the viewpoint-object distance and the projection area in image space. A weight metric is used, as shown in the following equation:

$$weight = \frac{\pi}{180} \arctan\left(\frac{R}{d}\right) \tag{4}$$

where R is the radius of the object's bounding sphere in object space, d is the distance from viewpoint to the center of the bounding sphere.

4.3. Voxel based shadow rendering

Shadows are essential for depth perception in physical world. In order to improve shading realism of CAD models, we implemented an efficient voxel based shadow rendering algorithm. The rendering process is shown in Fig. 7.

The primary shading is from triangle rasterization. For each frame of shading, a depth map can be generated, as shown in Fig. 7, for each pixel of which (e.g. $\boldsymbol{D}$) we can reconstruct the corresponding world space position $\boldsymbol{W}$. Then the shadow ray $\boldsymbol{S}$ can be calculated with the world space position $\boldsymbol{W}$ and light position $\boldsymbol{L}$. Then whether the pixel $\boldsymbol{D}$ is in the shadow or not is determined by an intersection test between shadow ray $\boldsymbol{S}$ and the voxel representation. Similar as visibility query, the shadow test is implemented by voxel ray casting algorithm on GPU. The final shading result is a combination of primary shading and shadow rendering.

5. Implementation and results

In this section, we describe our implementation and highlight its performance on massive CAD models.

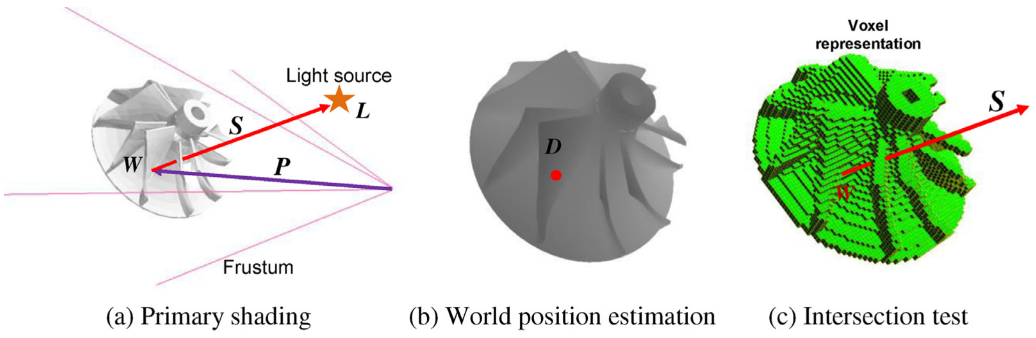

Fig. 7. Illustration of voxel based shadow rendering.

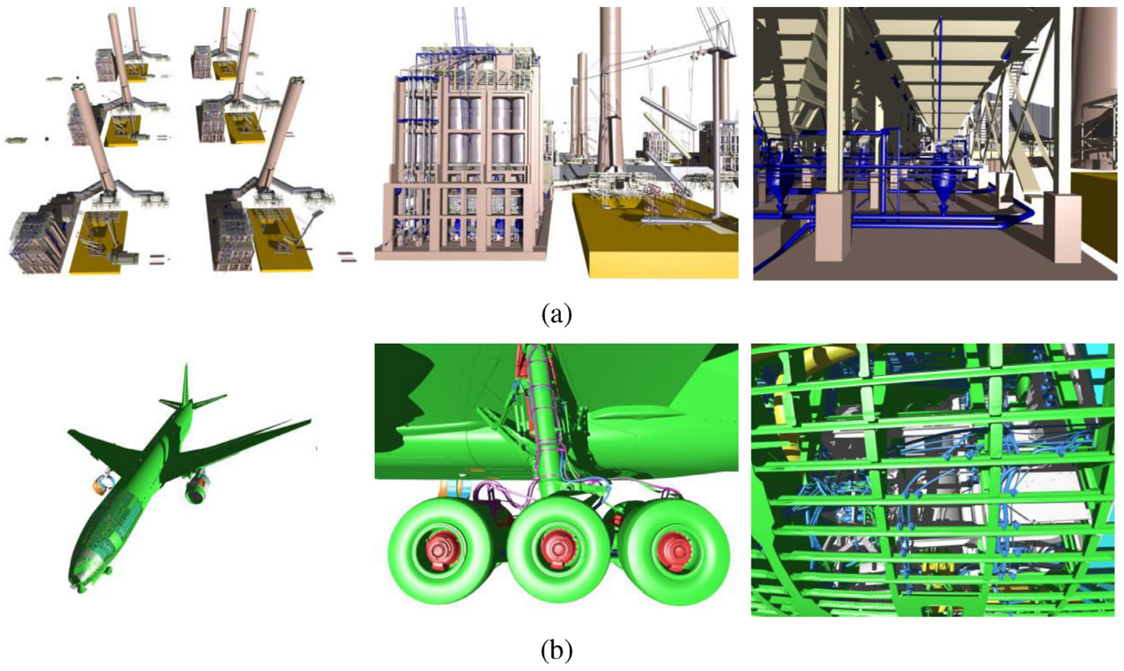

Fig. 8. Interactive rendering results of large-scale CAD models. (a) The Power Plants model (72,000,000 triangles). (b) The Boeing 777 model (337,000,000 triangles, 41.4 GB storage, one of the largest CAD datasets in manufacturing industry).

5.1. Implementation

The prototype software system is implemented on a desktop PC with an Intel Xeon E5-2620 (6 cores, 12 threads, 2.0 GHz) processor, 32 GB memory and a NVIDIA GeForce GTX 970 display card, running Windows 7 64-bit OS. The software system is developed with C++, OpenGL and GLSL, and built with x64 configuration to enable large memory address accessing. We tested the system with two massive CAD models that have different geometry and storage hierarchy complexity (Fig. 8). The two models include a combined power plants model which consists of 6 power plants (72 million triangles) and the Boeing 777 model (337 million triangles).

The system consists of two sub systems, the model preprocessing system and the real-time visualization system. Multithreading (8 threads) is utilized to accelerate processing speed during voxelization and voxel ray casting. The real-time visualization system is able to overlay a model list window on the rendering scene, as shown in Fig. 9(a). The window displays an assembly tree hierarchy which supports quick location of a specific part with its part number from millions of parts. Interaction functions like hiding or displaying, translation and rotation, displaying geometry information of a part/component, and navigation along a specific route are also integrated. The system also supports users to quickly save or switch to a viewport using its bookmark function. Users also can pick parts in the scene with mouse clicking, and display their part numbers on the scene, as shown in Fig. 9(b).

5.2. Results

5.2.1. Preprocessing performance

In order to test the efficiency of our LOD preprocessing and voxelization algorithms, we built the LODs and voxel representations of considered models with our system respectively. The construction statistics is shown in Table 1. The Power Plants model can be processed in 32 minutes, while the Boeing 777 model needs 101 minutes to generate its multi-resolution model as well as voxel representation (including the time of LOD geometry compression). The voxelization uses a preset resolution of $32K^3$ ($327{,}68^3$), which results in a uniform grid of 2 millimeter spacing on Boeing 777 model. The sparse voxel DAG structures cost only about 1/6 storage size of the LOD geometries, which are able to fit into GPU memory. With our geometry compression algorithm, the average LOD compression ratio is greater than 5, which has greatly reduced the disk storage size and runtime memory footprint.

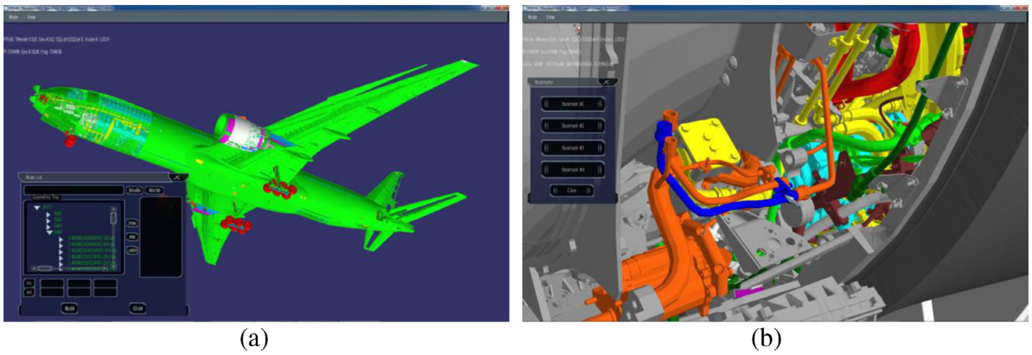

Fig. 9. Real-time visualization system for large-scale CAD models. (a) A model list window overlays on the rendering scene, which supports various interaction functions. (b) Picking a part on the engine, the picked part is highlighted in blue color and its part name is printed on the screen.

Table 2
Numerical results of real-time rendering. We show the average visibility query time, the average LOD processing time, data streaming time, rendering time (include shadow rendering), and the average frames per second statistics.

Models	Visibility query time	LOD Processing time	Streaming time	Rendering time	Avg. FPS
Power Plants	4.5 ms	1.8 ms	3.1 ms	6.0 ms	81
Boeing 777	5.3 ms	2.9 ms	8.4 ms	13.7 ms	33

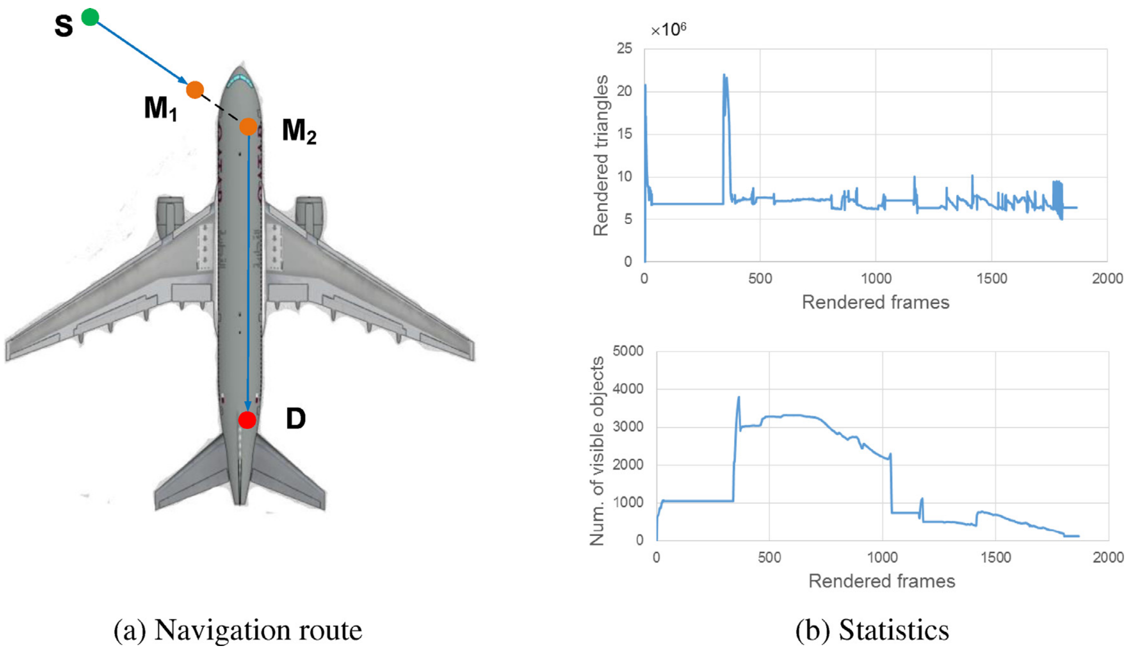

Fig. 10. The runtime performance of Boeing 777 model rendering. The number of visible objects, working set size (total rendered triangles).

5.2.2. GPU out-of-core rendering performance

The GPU out-of-core rendering algorithm is verified by different benchmarks as shown in Table 2. We computed different walkthrough paths through these models and measured the performance of our system. A resolution of 1024 × 1024 pixels is used for the interactive rendering. The results show that we can render at the interactive rates of 30~45 fps for the Boeing 777 model and 50~100 fps for the Power Plants model.

A special navigation route is designed to test the real-time rendering performance on the Boeing 777 model, as shown in Fig. 10(a): (1) the route starts from a position outside the cabin (**S**), and then the camera intermittently moves to a position close to the front fuselage (**M_1**); (2) the camera position shifts abruptly from **M_1** to a position at the interior of front fuselage (**M_2**) by triggering a viewport bookmark; (3) and then the camera intermittently moves to a position (**D**) at the rear fuselage. As the camera position moves along the route, fewer objects are inside the view frustum, the number of visible objects has been progressively diminished, as shown in Fig. 10(b). Through LOD adjustment, the total scale of the rendered triangles has fluctuated slightly between the min/max thresholds, which guarantees a fast rendering speed. The system reduces triangle number to about 8 million by removing invisible objects through visibility query and LOD processing, then the frame rate has fluctuated around 35 FPS, as shown in Fig. 11. To the best knowledge of us, this is by far the highest frame rate on the real-time rendering of Boeing 777 model on a commodity PC.

5.2.3. Comparisions

Xue. etc. employed an automatic LOD generation algorithm (without multithreading) and a coarse LOD model based occlusion culling strategy [18]. The test system was deployed on a HP

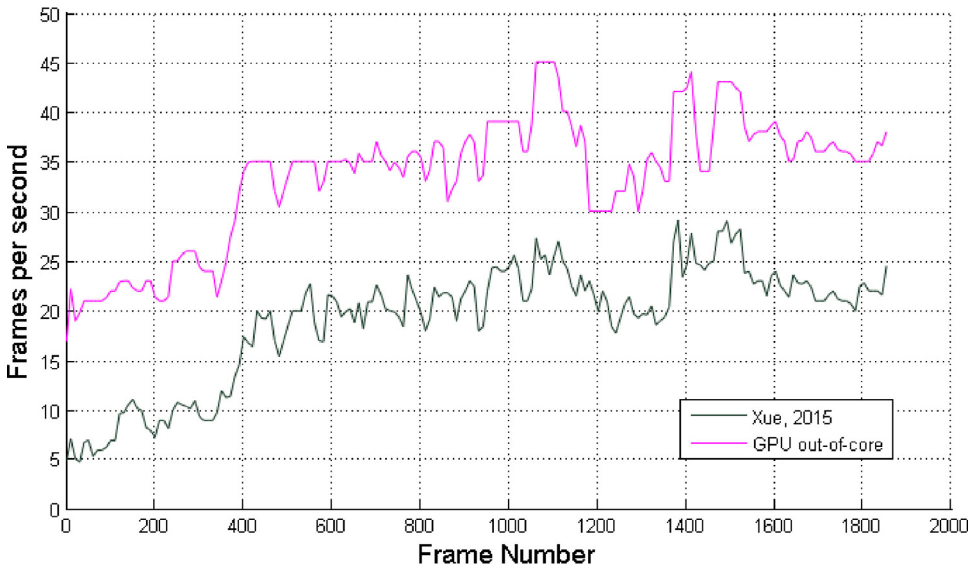

Fig. 11. Frames per second statistics of Boeing 777 model rendering (averaged each 10 frames).

Z800 workstation with two Intel Xeon X5550 processor (2.67 GHz, 8 cores, 16 threads), 16GB memory, and NVIDIA Quadro FX3800 graphics card. Their GPU out-of-core system was able to render the Boeing 777 model about 20 FPS (Fig. 11). While this paper focusses on accurate visibility test with a novel sparse voxel representation and efficient geometry compressions scheme. The compression of LOD geometry together with the integration of visibility query and LOD processing guaranteed a fast streaming speed of geometry data from main memory to GPU. The statistics in Fig. 11 show that GPU out-of-core framework in this paper has higher rendering efficiency than Xue's.

6. Conclusion and future work

An efficient voxel assisted GPU out-of-core framework has been presented for interactive rendering of large-scale CAD models. The framework has employed an aggressive geometry compression algorithm to produce compact LOD models efficiently while with only slight losses in object quality. The LOD processing has been integrated with voxel based visibility query to achieve better visibility test results and efficient LOD refinement. The framework has been tested by CAD models with tens to hundreds of millions of triangles. Users were able to explorer those models at interactive frame rates on desktop PCs with the prototype software system.

The current implementation of the rendering system includes a primary rendering and shadow rendering. Future versions will include many-light and ambient occlusion effect implementations. We are also interested in more efficient voxel data structure compression methods which help to lessen voxel storage size further.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (61170198) and the Innovation Fund of the State Key Laboratory of Virtual Reality Technology and Systems (VR-2013-ZZ-05). The Boeing 777 dataset was provided by and used with permission of The Boeing Company. The power plant model is courtesy of the University of North Carolina at Chapel Hill.

References

[1] Wald I, Dietrich A, Slusallek P. An interactive out-of-core rendering framework for visualizing massively complex models. In: Proceedings of the fifteenth eurographics conference on Rendering techniques: Eurographics association; 2004. p. 81–92.
[2] Peng C, Cao Y. A GPU-based approach for massive model rendering with frame-to-frame coherence. Wiley Online Library; 2012. p. 393–402.
[3] Peng C, Mi P, Cao Y. Load balanced parallel GPU out-of-core for continuous LOD model visualization. In: High performance computing, networking, storage and analysis (SCC), 2012 SC companion. IEEE; 2012. p. 215–23.
[4] Baxter WV III, Sud A, Govindaraju NK, Manocha D. Gigawalk: interactive walk-through of complex environments. Rendering Techniques; 2002. p. 203–14.
[5] Stephens A, Boulos S, Bigler J, Wald I, Parker S. An application of scalable massive model interaction using shared-memory systems. In: Proceedings of the 6th eurographics conference on parallel graphics and visualization: Eurographics association; 2006. p. 19–27.
[6] Crassin C, Neyret F, Lefebvre S, Eisemann E. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In: Proceedings of the 2009 symposium on interactive 3D graphics and games. ACM; 2009. p. 15–22.
[7] Laine S, Karras T. Efficient sparse voxel octrees. IEEE Trans Vis Comput Graph 2011;17:1048–59.
[8] Rusinkiewicz S, Levoy M. QSplat: a multiresolution point rendering system for large meshes. In: Proceedings of the 27th annual conference on computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co.; 2000. p. 343–52.
[9] Rusinkiewicz S, Levoy M. Streaming QSplat: a viewer for networked visualization of large, dense models. In: Proceedings of the 2001 symposium on interactive 3D graphics. ACM; 2001. p. 63–8.
[10] Tian F, Hua W, Dong Z, Bao H. Adaptive voxels: interactive rendering of massive 3D models. Vis Comput 2010;26:409–19.
[11] Gobbetti E, Marton F. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. ACM Trans Graph (TOG) 2005;24:878–85.
[12] Yoon S-E, Lauterbach C, Manocha D. R-LODs: fast LOD-based ray tracing of massive models. Vis Comput 2006;22:772–84.
[13] Áfra AT. Interactive ray tracing of large models using voxel hierarchies. Comput Graph Forum 2012;31:75–88.
[14] Yoon S-E, Gobbetti E, Kasik D, Manocha D. Real-time massive model rendering. Morgan & Claypool Publishers; 2008.
[15] Cigolle ZH, Donow S, Evangelakos D. A survey of efficient representations for independent unit vectors. J Comput Graph Tech 2014;3(2):1–30.
[16] Deering M. Geometry compression. In: Proceedings of the 22nd annual conference on computer graphics and interactive techniques. ACM; 1995. p. 13–20.
[17] MacDonald JD, Booth KS. Heuristics for ray tracing using space subdivision. Vis Comput 1990;6:153–66.
[18] Xue J, Zhao G. Interactive rendering and modification of massive aircraft CAD models in immersive environment. Comput Aided Des Appl 2015;12(4):393–402.