IAENG International Journal of Computer Science, 32:4, IJCS_32_4_7

The Local Z-Buffering Rendering

Henry X. Han, Michael Zeiger

***Abstract*—A local Z-buffering (LZB) rendering algorithm in the large complex scenes is presented in this paper. The local Z-buffering originates from the depth complexity based scene decomposition and following multistage rendering. In a naïve local Z-buffering, a large complex scene is decomposed into a low depth complexity scene and high depth complexity scene by computing a partition plane in the view space; the low and high depth complexity scenes are rendered by the Z-buffering and classic ray casting respectively. To achieve an output sensitive real-time rendering algorithm, a series of algorithms are developed to optimize the naïve version from aspects of general hardware support, empty pixel removal and fast potential visible primitive identification. Finally, the parallel version local Z-buffering rendering is introduced. Compared with similar types of rendering algorithms, the LZB algorithm is easy to implement, requires the least preprocessing time and achieves the satisfactory speed-up in the large complex scenes rendering.**

***Index Terms*— Visibility, real-time rendering, ray casting**

I. Introduction

The recently impressive progress in graphics hardware technologies is still facing the challenges from real-time rendering of the increasing scene complexity of the large complex scenes. The large complex scenes are large geometric data sets from the traditional compute games, virtual environments, scientific visualizations and even the new bioinformatics area [1,2]. A large complex scene is generally a scene with at least several million triangles and average depth complexity more than 10 with at least 90% invisible primitives in the rendering. The interactive rendering bottleneck from large complex scenes is mainly due to the fact that most graphics cards and APIs employ the Z-buffering algorithm or its variants in the visibility computing. This algorithm is an input-sensitive algorithm and its complexity grows linearly with the scene complexity: $O(|S|+|\Omega|)$, where $|S|$ is the scene size and $|\Omega|$ is the image space size: the total number of projected pixels generated by projecting primitives into the image space Ω. It is obvious that the Z-buffering algorithm is good at rendering low depth complexity scenes rather than large complex scenes because of its minimum z-value sorting mechanism. Although some sophisticated hardware techniques have been proposed to decrease $|\Omega|$ by implementing some occlusion culling algorithms; they are mostly customized approaches to handle specific scenes and those hardware techniques still have difficulty in interactive visualizations of a generally large complex scenes [1,2].

In additional to the hardware acceleration techniques, two categories of real-time rendering algorithms are developed for the large complex scene rendering; that is, visibility culling algorithms and scene approximation based rendering algorithms. The first type of algorithms focus on removing invisible objects to decrease the number of triangles sent to the graphics pipeline. According to where the occlusion culling computing is conducted, visibility culling algorithms can be further classified as image/object space algorithms, including the cell based potential visible set algorithm[3], hierarchical Z-buffer algorithm (HZB) [4], hierarchical occlusion map (HOM) [5], cPLP conservative prioritized layered projection (cPLP) [6,7] and occluder shrinking *et al.* [8,9]. On the other hand, the second type of algorithms focus on approximating the input scenes by a corresponding reduced/simplified geometric data set that is less expensive to render, to reduce the number of pixels projected into Ω. The level of detail method (LOD) and point based rendering belong to this class. For the more detailed information about these algorithms, we suggest the related reviews and papers on this topic [11,12,13,14,15,16]. Although these proposed interactive rendering approaches did great improvements on the real-time rendering of large complex scenes, they still share some of following potential weak points. For example, algorithms need a large amount of preprocessing or need customized hardware support to real-time visualization. Some approaches even can not guarantee the conservation in the visibility or just can apply to special scenes [1,14].

We believe an efficient real-time rendering algorithm of large complex scenes must have the following characteristics. 1) It should be an output sensitive algorithm, that is, its computational complexity is weakly dependent on scene complexity; 2) Its preprocessing stage should be light-loaded. 3) It can access commonly available graphics hardware support rather than special ones; 4) It is easy to implement and can apply to different categories of scenes. In this work, we present an output sensitive real-time rendering algorithm called local Z-buffering rendering (LZB) according to the specified criteria. The LZB rendering is based on the scene decomposition in the view space and following multi-passing rendering. In the scene decomposition, the scene in the view frustum is decomposed as a low depth complexity/ near-view scene S_{near} and a high depth complexity / far-scene S_{far}

Manuscript received March 12, 2006.

Henry X. Han was with Applied Mathematical and Computational Sciences, University of Iowa, Iowa city, IA 52240 USA. He is with Department of Mathematics, Eastern Michigan University, Ypsilanti, MI 48197 USA. (phone:734-487-5044; fax:734-487-2489; e-mail: xhan1@emich.edu).

Michael Zeiger is with Department of Computer Science, Eastern Michigan University, Ypsilanti, MI 48197 USA.

dynamically or statically. The S_{near} can be viewed as an automatically selected occluder set for S_{far} where the primitives are more likely invisible than those in S_{near}. The image I_{near} of the scene S_{near} is obtained by *local rendering*, that is, the Z-buffering is employed to render the primitives in S_{near}. The local rendering is similar to the selected occluder rendering to compute the finest hierarchical occlusion map in the hierarchical occlusion map method with the graphics hardware support, lighting and texturing. Moreover, the image I_{near} obtained in the local rendering is a partially correct image rather than the finest occlusion map. The image I_{far} for the scene S_{far} is computed by rendering a potential visible list (PVL) through the Z-buffering. The potential visible list (PVL) can be quickly computed by the selectively lazy ray casting (SLR) and object-oriented ray-casting (OOR) accelerated by coherence based ray-octree traverse algorithm. The LZB algorithm performs well in the real-time rendering of general large complex scenes. In the next sections, we present the naïve LZB algorithm and its optimization process to be the final local Z-buffering rendering algorithm.

II. THE NAÏVE LOCAL Z-BUFFERING RENDERING

The basic idea of the naïve Local Z-buffering is to employ the Z-buffering to render the low depth complexity scene and ray-casting to render the high depth complexity scene. Such an idea aims at taking advantage of the "good features" of Z-buffering and ray-casting. Z-buffering is good at rendering the low depth complexity scene and ray-casting has the built-in occlusion culling mechanism to reject the hidden objects after spatial sorting.

A. The overview of the naïve local z-buffering rendering

The naïve LZB consists of a preprocessing stage and real-time stage. In the preprocessing stage, an octree T is built to organize the input triangle set in scene S. The termination condition in the octree building is the maximum depth of the tree and the maximum primitive number in each leaf node. To accelerate the preprocess, T. Moller 's triangle-box overlap testing algorithm is employed to decide if a triangle intersects with octants in our octree building. It is faster than the default triangle-box overlap testing algorithm [17,18]. In the real-time stage, there are following five steps.

1. Conduct the hierarchical view frustum culling: traversing the octree T to collect leaves in the current view frustum.
2. Compute a partition plane $z = z_{clip}$ in the camera space to partition the input scene S into two disjoint scenes $S = S_{near} \cup S_{far}$. Scene S_{near} / S_{far} is a low/high depth complexity scene respectively after the partition.
3. Conduct the local rendering: employ the Z-buffering to render the near-view scene S_{near}.
4. Query the frame buffer to get the unfinished pixels to be shaded.
5. Cast rays from the unfinished pixels to render the far-view scene S_{far}.

B. Partition plane computing methods

A partition plane is a "good plane" if the scene S_{near} is a low depth complexity scene and its image I_{near} contributes much more pixels than the image I_{far} of the far-view scene S_{far} in the final rendering, $|I_{near}| >> |I_{far}|$. There are two ways to set a partition plane in the view space: a static (ad-hoc) and a dynamic approach (the coarse ray-casting). In the ad-hoc approach, the plane $z = z_{clip}$ can be reasonable set at $15\% - 30\%$ depth position in the view frustum because a large complex scene has at least 90% hidden primitives. The ad-hoc approach is equivalent to setting a small view frustum including all primitives in the S_{near}. We found the partition plane would be closer to the near plane with increase of the scene size [18]. In the dynamic approach, the coarse ray casting is used to decide the partition plane. A set of uniformly coarse-sampled pixels on the screen window conduct the ray casting to compute the distance, the ray length, to the nearest surface for each ray. The average distance is chosen as the location of the clipping plane and it is then transformed into the corresponding distance in the view space. This approach can get a better partition plane because the coarse ray casting probes the nearest surface locations. It is unnecessary to compute the partition plane for each frame. The frame coherence can be exploited by reusing the partition planes in the previous frames.

The naïve local-Z buffering can work well for the densely occluded scenes which are special kinds of large complex scenes. The densely occluded scenes can be found from architecture models, office models and some city models [18]. In a densely occluded scene, the near-view scene image I_{near} is very close to the final scene image due to generally available large occluder; that is, the ray-casting takes only a light workload to compute the far-view scene image I_{far} and the local rendering takes the majority rendering. Figure 1 showa a is a densely occluded scene with 572,412 triangles with the average depth complexity 15. The average Z-buffering rendering is 0.69 second and the average naive LZB rendering time is 0.37 second where partition planes are computed by the coarse ray casting with frame coherence.

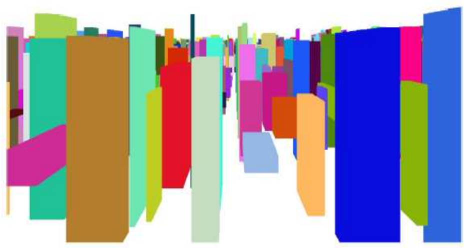
Fig. 1. A densely occluded scene with 572,412 triangles

III. Refining the Naïve Local Z-buffering Algorithm

It is clear that the naïve local Z-buffering algorithm can not work for the general large complex scenes due to the large overhead from the ray-casting stage. In this section, we improve the naïve local Z-buffering algorithm from three aspects: the general hardware support, fast potential visible list identifying, empty pixels removal and fast ray transverse. They are corresponding lazy ray casting, selectively lazy ray casting, object-oriented ray casting. The final version local Z-buffering algorithm is the integration of all these improvements.

A. "Lazy" ray casting

It is reasonable to turn to the possible hardware support to accelerate the ray casting in the naive LZB. We propose a lazy ray casting for this. The idea of the "lazy" ray casting is to decompose the classic ray casting into two parts: a potential visible list (*PVL*) finding and the potential visible list (*PVL*) rendering. In the "lazy" ray casting, software is only responsible for finding the potential visible primitives list (*PVL*) for the far-view scene S_{far}. The nearest surface identification and shading for all unfinished pixels in the ray-casting are left to the graphics hardware, that is, sending the *PVL* to the graphics pipeline and using the Z-buffering to render the *PVL*.

In the *PVL* finding, a local list l is maintained for each ray-casting pixel p to hold the identification numbers of a set of the potential visible primitives. The set of the potential visible primitives contain the nearest surface (the first-hit triangle) for the ray emanated from the pixel p. The *PVL* is the union of all the local lists where each triangle identity is only counted once. Actually, a rendering bit is set for each primitive before it is recorded in the *PVL* to remove the duplicated primitive identification numbers.

To compute the local list l for a ray r emanated from the pixel p, we just need to find the first ray-triangle hit for the ray r rather than test all triangles associative with the ray path. A ray path is a set of octree leaves traversed by a ray until the ray visibility status is resolved; that is, there is either a found nearest surface in an octree leave or no intersection occurrence between the ray and the octree. In the lazy ray casting, if there is a ray "hit" happen for a triangle in a leaf node for a ray r, the ray-triangle intersection test terminates and all the primitives associative the leaves traversed by r and current leaf are recorded in the local list l. Figure 2 indicates the idea of the lazy ray casting.

The potential visible list (*PVL*) rendering is to use the Z-buffering to render all the primitives in the *PVL*. The rendering results are just the image of the far-view scene S_{far}. It is easy to see that the computing in the lazy ray casting consists of CPU based *PVL* finding and GPU based *PVL* rendering. Thus there is general hardware support for the lazy ray casting compared with the classic ray casting and its performance can "grow up" potentially with the CPU and GPU technology.

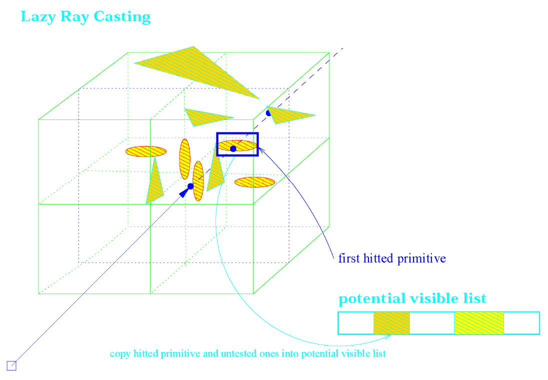
Lazy Ray Casting
first hitted primitive
potential visible list
copy hitted primitive and untested ones into potential visible list

Fig. 2. The idea of the lazy ray casting

How much "saving" can we get from the lazy ray casting compared with the classic ray casting? To answer this question, we compare the complexity between two approaches. Suppose there are total n triangles in the ray path of a ray r, there will be n ray-triangle intersection computing in the classic ray casting. However, the number of ray-triangle intersection testing in the lazy ray casting is $n/2$ averagely and n in the worst case. Actually, in many large complex scene rendering experiments, the first ray hit happens in the first several leave transverse and the exhaustive search case is a rare case [18]. Thus the complexity for a ray r in the classic ray casting is $nc_t + c_s$, where c_t is the average process cost to finish one triangle including the octree traverse and ray-triangle intersection testing time, and c_s is the average shading cost for a pixel, which is related to the shading models used in the rendering. On the other hand, the lazy ray casting has the average complexity: $0.5 \cdot nc_t + c_z n$, where c_z is the average cost to render each triangle by Z-buffering whose order is in the range 10^{-6} to 10^{-11} according to different graphics hardware [2]. The average saving from the lazy ray casting is $0.5 \cdot nc_t + c_s - c_z n$ and it is related to CPU speed and GPU capacity in the host machine.

B. Selectively lazy ray casting (SLR)

In the lazy ray casting, the worst case to find the local list l for each ray is to test all the primitives associative with the leaves in its ray path. How can we avoid such worst case? In this section, two local occluder selection methods are proposed to accelerate the *PVL* finding. The first is called a static local occluder selection, which is to build an *Octree-o* in the preprocessing stage. The second is called is a dynamic local occluder selection, which selects local occluders dynamically according to a measure called *visibility*. Two methods can also be integrated together to speed-up the *PVL* finding.

The reason why the worst case occurs in the local list finding is because that all primitives are treated uniformly regardless of their different sizes and normal directions. These factors are important measures to determinate the visibility probability of each primitive. Thus the ray-triangle intersection testing has to be conducted for all triangles associative with a ray path even if there is no intersection for the ray with any triangles. It is clear that such a uniform testing mechanism is by no means an efficient approach to resolve each pixel visibility status because these measures are not involved in the ray-triangle testing.

To resolve a pixel visibility status fast in the lazy ray casting, we introduce the local occluder selection method. The local occluder selection is a selective method, which only tests those *most likely visible primitives* (local occluders) to resolve the ray visibility status quickly. Such a selectively ray triangle intersection testing mechanism leads to the selectively lazy ray casting method (SLR). There are two SLR methods according to how to select the most likely visible primitives: the static and dynamic local occluder selection.

1) Static local occluder selection

The idea of the static local occluder selection is to select the local occluders for each leaf in the octree built in the preprocessing stage. That is, build an *Octree-o* (Octree with local occluders). In the *Octree-o* building process, one or several local occluders are selected in each leaf node by sorting triangles according to their areas. The extra memory demand for building an *Octree-o* is just several bytes to record the local occluder identification numbers. The time consuming in building an *Octree-o* is same as building a general octree [18].

In the lazy ray casting stage, the ray-triangle intersection testing starts from the first occluder, the largest triangle associative with the current leaf node. If there is an intersection occurrence for a local occluder, all the triangles associative the leaf node will be recorded into the *PVL*. Then the leaf is marked as a visible node. On the other hand, if there is no intersection ("hit") between a ray and the pre-selected local occluders in a leaf, the intersection query between the ray and other triangles in the leaf will be skipped. The identification numbers of triangles associative with the leaf will be recorded in the *PVL*. Then, the leave point is computed for the ray in the leaf node and the ray-triangle intersection query will be conducted for the local occluders in the next leaf until the visibility status of the ray is resolved.

Compared with the original lazy ray-casting, the static local occluder selection decreases the number of intersection tests and increases the size of the potential visible list (*PVL*). The approach works very well in the scenes where there are large triangles spanning many leaves in the corresponding octree. The octree built for such scenes sometimes is an ill-balanced tree. The visibility status of a ray won't be known until the all triangles tested in the ray path. Actually, the large triangles are ideal local occluders for in the *Octree-o*. Because the ray-triangle test is only conducted for the local occluders in each octree leaf, the average ray traverse time decreases largely for such "selective mechanism". According to our implementation, the number of the average ray-triangle intersection query dropped dramatically for 5 occluders selected in each *Octree-o* leaf with maximum 150 triangles. The potential visible list (*PVL*) size increases correspondingly for the lazy ray casting under static local occluder selection acceleration.

2) Dynamic local occluder selection

The static local occluder selection just considers the size of a primitive as a measure to decide the visibility probability of a primitive. It is obvious that the primitive normal also play an important measure to decide the visibility probability of a primitive. Because the distance of primitives in a same leaf node to the viewpoint is almost same. Under this case, the distance may not play an important role in determinate the visibility probability of a primitive. Thus we define a measure called *visibility* to measure the visibility probability of a primitive *p* as

$$visibility = -v \cdot n \cdot area(p) \tag{1}$$

Where *v* is the view direction and *n* is the primitive normal and *area(p)* is the area of the primitive. It is easy to see that the primitive visibility depends on the inner product of $-v$ and primitive normal and the primitive size. The primitives in a same leaf node with larger visibility values will be mostly likely to be hit by a ray for a given viewpoint.

In the dynamic occluder selection, a base *visibility* for each leaf node is set according to the average triangle area size in each leaf and a preset value of $-v\cdot n$. For example, $-v\cdot n$=0.5. In the actually ray shooting, the visibility value of each triangle is computed dynamically. If the visibility of a triangle is greater than the base visibility, the ray-triangle intersection testing will be conducted for the triangle. Otherwise, it will be skipped and the visibility of the next triangle will be computed. If there is a candidate triangle hit by a ray, the leaf will be marked as 'visible', the identification numbers of all triangles in the leaf are recorded in the final *PVL* and ray-intersection testing terminates. However, if there is no intersection for all the selected candidates, the triangles in the leaf node are still recorded in the *PVL* and ray-triangle intersection test goes to next leaf node until the ray visibility status of is resolved ; that is, there is at least one hit or no hit in the octree.

Compared with the static occluder selection, the dynamic selection can get more accurate estimate for the visibility probability of a triangle. But it will depend on the scene properties. In some scenes, if there is a large number of larger triangles existed, the performance of the dynamic occluder selection is not as good as the static occluder selection. In the implementation, a hybrid version of the two local occluder selection approaches is employed. A certain number of local occluders are pre-selected in the *Octree-o* and the visibility is computed for each occluder to filter the occluders with low visibility values.

C. Objected- oriented ray casting

Although the selectively ray casting (SLR) can find the potential visible list (*PVL*) for the far-view scene S_{far} in the local Z-buffering, it still faces the following "empty pixel problem". The image of a large complex scene, even a densely occluded scene, may just shade a relative small pixel set on the screen most pixels are just empty pixels, where no triangles in the scene are projected on these pixels for certain viewpoints. For some empty pixels in the selectively lazy ray casting, they can be "rejected" in the first several octree level traverses because there is no intersection for their rays with any triangles in the octree. However, for some empty pixels between the image sets of neighbor objects, the octree traverse may reach the deepest leaves before knowing their visibility statuses. It is obvious that these types of empty pixels bring more overhead in the ray-triangle intersection testing and increase the *PVL* size potentially. In the local Z-buffering rendering, the ray shooting overhead from these empty pixels will increase linearly with increase of the image resolution [18].

To decrease the overhead from these empty pixels to its minimum level in rendering, we introduce the *object-oriented ray casting* (OOR). The basic idea is just casting rays from the unfinished pixels, which are in the projection area of the objects in the current view frustum rather than casting rays for all unfinished pixels on the screen. In the OOR, the projection of an object on the screen can be approximated by the projection of its corresponding axis aligned bounding box (AABB) [5] or object oriented bounding box (OBB) [5] on the screen. Computing the projection of each OBB can get smaller object projection area on the screen. But it asks more preprocessing time. In our implementation, we project the AABB of each object in the view frustum on the screen. In the computing of an AABB projection, it is unnecessary to compute projections of all vertices. On the other hand, the projection of the base point, the south-west corner point in the lower surface of the AABB, is computed at first and then the projections of the other vertices of the AABB are computed by adjusting the corresponding offset vectors. The projection of an AABB (3D box) is a simple convex polygon P_{aabb} with 4, 5 or 6 sides, which are corresponding to 1, 2 or 3 face visible cases. To decide if an unfinished pixel is in the projection region, a bounding box B_p for each projection is computed on line at first. If the unfinished pixel is in the bounding box B_p, then we query if the pixel is in the P_{aabb}, a convex polygon with maximum six sides. This query can be computed in dynamically and there are many real-time algorithms to decide if a 2D point is in a convex polygon in the computer graphics literature [12].

The completeness of the ray shooting in the OOR is indicated by counting the number of objects in the current view frustum instead of counting the unfinished pixels on the screen. That is, ray shooting is *object-oriented* rather than *pixel-oriented* where the ray casting processing is considered finished if the visibility status of the last pixel is known. In the *object-oriented ray casting*, the ray casting stage is considered complete if the visibility status of the last pixel in the projection area of the last object in the current view frustum is resolved. Considering there are overlap regions on the screen for the objects in the view frustum, a two dimensional Boolean matrix maintained to record if an unfinished pixel p in the projection region of an object having finished ray shooting. If the pixel p has conducted ray shooting, its corresponding mask value in the Boolean matrix is set true. The pixel won't be involved in any ray shooting although it will be located in a projection of another object. Figure 3 indicates the idea of the object-oriented ray casting.

The object-oriented ray casting can be integrated with selectively lazy ray casting to accelerate the naïve LZB. That is, conducting the AABB projection for the objects in the current view frustum but behind the partition plane to find the interested regions on the screen to shoot rays. The *PVL* for the far-view scene is computed by the selectively lazy ray casting and sent to the graphics pipeline to be rendered by Z-buffering.

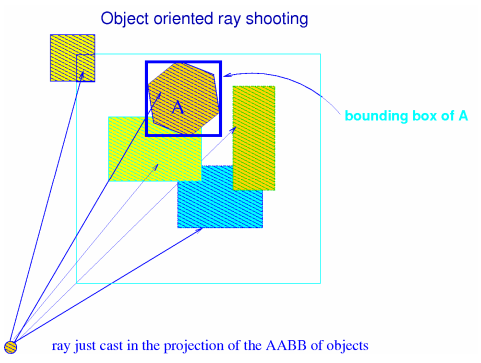
Object oriented ray shooting
A
bounding box of A
ray just cast in the projection of the AABB of objects

Fig. 3. The idea of the object-oriented ray casting

D. The final version local Z-buffering algorithm

Besides the previous three improvements, we also developed a coherence based ray-octree traverse algorithm to improve the standard top-down traverse used in the algorithm [18]. The final version local Z-buffering consists of integrations acceleration techniques of naïve local Z-buffering from these points of views: the selectively lazy ray casting provides the quick *PVL* finding and general hardware support for ray casting; the object-oriented ray casting decreases the overhead from empty pixels in the ray-shooting and the coherence based ray-octree traverse algorithm optimizes the top-down octree traverse in the ray casting stage. The LZB algorithm is an output sensitive algorithm with respect to the scene complexity and final image resolution. The detailed cost-model analysis and proof can be found in the [18].

IV. Rendering Experiments

We give the following rendering experiments to test the performance of the LZB algorithm. The first rendering example is from randomly generated densely occluded virtual city with 1,822,260 triangles. The general Z-buffering rendering needs 1.76 seconds averagely and the improved LZB just take 0.34 second for averagely sampled 80 viewpoints along a circular view path. The second rendering example is an assembling scene with 3.8 million triangles from the UNC power plant model [19] (Figure 4). We sample 120 viewpoints along a circular view path in the rendering, the LZB rendering achieves average 12 FPS compared with the average 6.2 FPS under Z-buffering rendering. In the second rendering experiment, we "prove" the output sensitivity of the LZB. A series of scenes (scenes scene triangle numbers range from 18,000 to 3.8 million) are taken from the UNC power plant model. For the 3.8 million triangle scene, the LZB rendering achieves the average 10.3 FPS which is much higher than 5.7 FPS from the Z-buffering rendering. We plot the LZB and Z-buffering rendering time with respect to the scene size, image complexity and scene depth complexity by sampling 80 viewpoints in a circular view path in the rendering. We find that the LZB is weakly dependent on the image complexity, scene size and scene depth complexity and the Z-buffering rendering time linearly increases with respect to the measures. In other words, the LZB is an output sensitive algorithm. All our experiments are conducted under an AMD machine running under Linux OS with a 2.1 GHz AMD processor with 1.0 G main memory and Gf4 series graphics hard under the Linux OS for the final image resolution with 512×512 pixels.

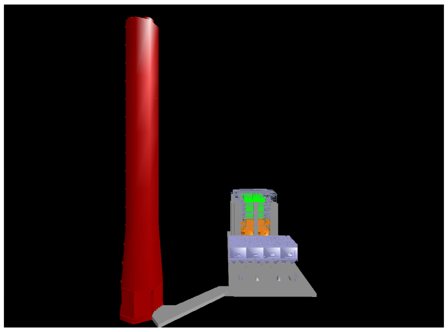

Fig.4. A large complex scene with 3.8 million triangles

We also implement the parallel version LZB. We use Message-Passing Interface (MPI) library to implement the parallel local Z-buffering algorithm and achieves the satisfactory speed-up. The Figure 5 shows the experimental results for different size of scene datasets and numbers of processors. In the parallel implementation, due to the communication overhead caused by the broadcast of depth buffer at each rendering frame, the parallel-LZB has a lower efficiency than the sequential LZB for small size (<0.5 million in our case) of data sets. However, as the size of data set increases, the parallel LZB performs much better than the sequential LZB and achieves a higher speedup (≈1.27 for 8 processors).

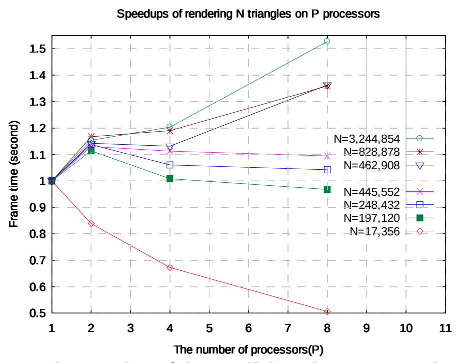

Fig. 5. The speedup of the parallel rendering n triangles on p processors

V. Conclusion and Future Work

In this paper, we present a real-time rendering algorithm LZB for the large complex scenes. Compared with the similar algorithms, the LZB is an output sensitive and easy to implement, can work for general large complex scenes and has no special hardware and heavy-loaded preprocessing requirements. In the following work, we would like to compare the performance of the LZB and other rendering algorithms: HZB (hierarchical Z-buffering), HOM (hierarchical occlusion culling) in large complex scenes besides refining our current version parallel LZB algorithm.

References

[1] D. Cohen-Or, Y. Chrysanthou, C. Silva, C. and F. Durand, "A survey of visibility for walkthrough applications", *IEEE TVCG*, 2002.
[2] R. Fernando, *GPU Gems*, Addison-Wesley, 2004.
[3] S. Teller *et al.*, "Visibility preprocessing for interactive walkthroughs", *Computer Graphics* 25(4), 1991, pp. 61-69.
[4] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility", *SIGGRAPH 93 Proceedings*, Annual Conference Series, 1993, pp. 231-238
[5] H. Zhang, D. Manocha,, T. Hudson, and K. Hoff., "Visibility culling using hierarchical occlusion maps", *SIGGRAPH'97 Proceedings*, Annual Conference Series, 1997, pp. 77-88.
[6] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility preprocessing with occluder fusion for urban walkthroughs", *Rendering Techniques 11th Eurographics workshop on rendering*, 2000, pp. 71-82.
[7] S. Coorg, and S. Teller, "Real-time occlusion culling for models with large occluders". *ACM Symposium on Interactive 3D Graphics* , 1997, pp. 83-90.
[8] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative visibility preprocessing using extended projections", *Proceedings of SIGGRAPH* , 2000, pp. 239-248.
[9] J. Klosowski and C. Silva, "The prioritized layered projection algorithm for visible set estimation", *IEEE Transactions on Visualization and Computer Graphics*, 6(2) , 2000.
[10] J. Klosowski and C. Silva, "Efficient conservative visibility culling using the prioritized layered projection algorithm", *IEEE Transactions on Visualization and Computer Graphic*s 7(2), pp. 365—379, 2001.
[11] M. Garland, "Quadric-based polygonal surface simplification", Ph.D. thesis, Technical Report CMU-CS-99-105, Carnegie Mellon University, 1999.
[12] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and E. Huebner, *Level of Detail for 3d Graphics*, Morgan Kaufmann, 2002.
[13] M. Sainz, and R. Pajarola, "Point-based rendering techniques", *Computer & Graphics*, 28, pp. 869-879, 2004.
[14] M. Wand, M. Fischer and I. Pater, " The randomized z-buffer algorithm: Interactive rendering of highly complex scenes", *SIGGRAPH 2001, Computer Graphics Proceedings*, 2001.
[15] H. Pfister, M. Zwicker, J. van Baar and M. Gross, "Surfels: Surface elements as rendering primitives", *SIGGRAPH 2000 Proceedings*, Annual Conference Series, 235-242, pp. 2000.
[16] S. Rusinkiewicz and M. Levoy, "Qsplat: A multi-resolution point rendering system for large meshes", *SIGGRAPH 2000 Proceedings*, Annual Conference Series, pp. 343-352, 2000.
[17] Akenine-Mįller,T. : Fast 3d triangle-box overlap testing, 2001.
[18] X. Han, "The Local Z-buffer Algorithm for Rendering Large Complex Scenes", Ph.D. Thesis, Department of Applied Mathematics and Computational Sciences, The University. of Iowa, 2004.
[19] UNC power plant model: http://www.cs.unc.edu/~geom/Powerplant/ 2004.