

# Local Z-Buffering Rendering in Large Complex Scenes

Xiaoxu Han

Department of Mathematics  
Eastern Michigan University  
Ypsilanti MI, USA

**Abstract** - A novel output sensitive real-time rendering algorithm local Z-buffering for the large complex scenes is presented in this paper. A naïve algorithm is proposed at first based on decomposing a large complex scene into a low-depth complexity scene and a large depth complexity by computing a partition plane in the view space. The low and high depth complexity scenes are visualized by Z-buffering and ray casting respectively. This naïve algorithm works well for densely occluded scenes. The naïve algorithm is refined by a series of acceleration algorithms from different point of views. The acceleration algorithms include the selectively lazy ray casting, object-oriented ray casting and a coherence based octree traverse algorithm. The final local Z-buffering algorithm requires the least preprocessing time and can achieve satisfactory speed-up averagely in the real-time rendering of large complex. Its performance can grow up with the improvements of the CPU and GPU capacities.

**Keywords:** Multi-passing rendering, z-buffering, Octree

## 1 Introduction

Recent 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 the new bioinformatics area [1,2]. A large complex scene is generally a scene with at least 3 million triangles and average depth complexity 15. These scenes are organized by objects and each object consists of a set of triangles. There are relatively large variances of triangle sizes in a large complex scene and at least 90% primitives are hidden in rendering. For example, a virtual city with 30 million triangles is a typical large complex scene.

The interactive rendering bottleneck from such scenes is largely due to the fact that most graphics cards and APIs employ the Z-buffering algorithm or its variants in the visibility computing. The Z-buffering algorithm is an input-sensitive algorithm and its complexity grows linearly with the scene complexity; that is,  $O(|S| + |\Omega|)$ . Here  $|S|$  is the scene size and  $|\Omega|$  is the image space size, which is the total number of projected pixels generated by projecting primitives into the image space  $\Omega$ . The lower bound of the

image space size is  $|\Omega| \geq \tau \times v$ , where  $\tau$  is the average scene depth complexity and  $v$  is the screen window size in the rendering. 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 occlusion culling algorithms, these hardware techniques can improve the graphics pipeline throughputs but still have difficulty in interactive visualizations of large complex scenes [1,2].

In addition to the hardware acceleration techniques, two categories of real-time rendering algorithms can be found in the large complex scene rendering: visibility culling algorithms and scene approximation based rendering algorithms. The first type of algorithms focus on removing invisible objects to decrease number of triangles sent to the graphics pipeline. According to the different spaces where the occlusion culling computing conducted, visibility culling algorithms can be further classified as image/object space algorithms, including the cell based potential visible set (PVS) 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]. The second type of algorithms focus on approximating the input scenes by a corresponding reduced/simplified geometric data set, which is less expensive to render than the original scene, to reduce the pixel numbers projected into  $\Omega$ . 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]. Although these proposed interactive rendering approaches did great improvements on the real-time rendering of large complex scenes, they still share some of following weak points, such that 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 get 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 these 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 / a near-view scene  $S_{near}$ , and a high depth complexity / a far-view scene  $S_{far}$ , 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 the 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 [5]. However, the graphics hardware support Z-buffering, lighting and texturing. Moreover, the image  $I_{near}$  obtained from the local rendering is a partially correct image rather than the finest occlusion map in the HOM. The image  $I_{far}$  of the scene  $S_{far}$  is computed by rendering a potential visible list (PVL) through the Z-buffering. The potential visible list can be quickly computed by the selectively-lazy ray casting and object-oriented ray-casting accelerated by the coherence based ray-octree traverse algorithm. The LZB algorithm performs well for the general large complex scene rendering. In the next sections, we present the naïve LZB algorithm and its optimization: the final local Z-buffering rendering algorithm.

## 2 A naive local z-buffering algorithm

The basic idea of the naïve Local Z-buffering is to employ the Z-buffering to render the low depth complexity scene and the ray-casting to render the high depth complexity scene. This idea aims at taking advantage of the “good features” of Z-buffering and ray-casting.; that is, 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.

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 triangles in the 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 [13,14]. There are following five steps in the real-time stage.

1. Conduct the hierarchical view frustum culling by traversing the octree  $T$  to collect leaves in the current view frustum.
2. Compute a partition plane  $z = z_{clip}$  in the view space to partition the input scene  $S$  into two disjoint scenes:  $S = S_{near} \cup S_{far}$ . Scene  $S_{near}$  and  $S_{far}$  are a low and high depth complexity scene respectively.
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  $S_{far}$ .

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 in the final rendering:  $|I_{near}| \gg |I_{far}|$ . There are two ways to set a partition plane in the view space: a static / ad-hoc and a dynamic approach. In the ad-hoc approach, the plane  $z = z_{clip}$  can be reasonable set at the 15% to 25% 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 get closer to the near plane with increase of the scene size [14]. 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 interior architecture models, office models and some city models [14]. In a densely occluded scene, the near-view scene image 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 and the local rendering takes the majority rendering. Figure 1 shows 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 naïve LZB rendering time is 0.37 second where partition planes are computed by the coarse ray casting with frame coherence. However, the naïve algorithm may not work well for the general large complex scenes obviously due to the large amount of overhead from ray casting and following shading.

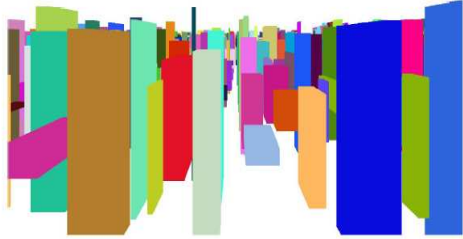


Fig. 1. A densely occluded scene with 572,412 triangles

### 3 The local z-buffering rendering

In this section, we improve the naïve local Z-buffering algorithm from four 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 and coherence based ray traverse algorithm. The final version local Z-buffering algorithm is the integration of all these improvements.

#### 3.1 The lazy ray casting

It is reasonable for us to turn to the possible hardware support to accelerate the ray casting in the naïve LZB such that it can handle general large complex scenes in real-time. For this purpose, we propose the lazy ray casting. The idea of the lazy ray casting is to decompose the classic ray casting into two parts: potential visible list finding and the potential visible list rendering. In the lazy ray casting, software is only responsible for finding the potential visible primitives list (*PVL*) for the far-view scene. 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*. Figure 2 indicates the idea of the lazy ray casting method.

In the *PVL* computing, 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 contains the nearest surface / the first-hit triangle for the ray emanated from the pixel. 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 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: there is either a found nearest surface in an octree leaf 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  $l$ . The potential visible list 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” with the CPU and GPU technology.

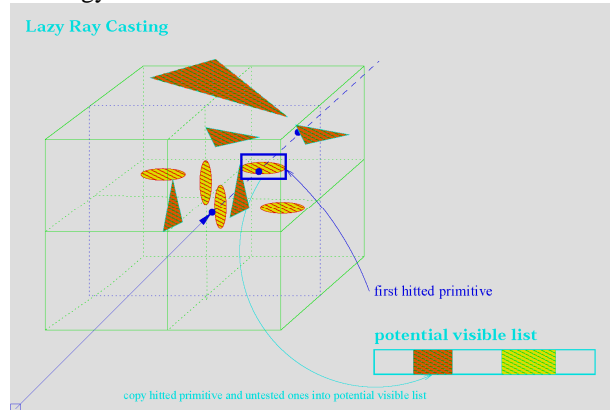


Fig. 2. The idea of the lazy ray casting

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, 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 leaf transverse and the exhaustive search case is a rare case [14]. The complexity for a ray  $r$  in the classic ray casting is  $nc_t + c_s$ . Here  $c_t$  is the average process cost to finish one triangle, which includes 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. The lazy ray casting has the average complexity  $0.5 \cdot nc_t + c_z \cdot n$ . Here  $c_z$  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]. Obviously, the average saving from the lazy ray casting is  $0.5 \cdot nc_t + c_s - c_z \cdot n$  and it is related to CPU speed and GPU capacity in the host machine.

### 3.2 Selectively lazy ray casting

In the lazy ray casting, the worst case to find the local list for each ray is to test all the primitives associative with the leaves in its ray path. How can we avoid the worst case to decrease the *PVL* finding time? In this section, two local occluder selection methods are proposed to accelerate the *PVL* finding. The first is called static local occluder selection which is to build an *Octree-o* in the preprocess. 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 idea of the local occluder selection was inspired by the general occluder selection in the object space [5,7]. 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 selective 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.

The idea of the static local occluder selection is to select the local occluders for each leaf in the octree built in the preprocessing; that is, build an *Octree-o*: octree with local occluders. In the *Octree-o* building, 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 [14].

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 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 scene 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 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

$$visibility = -v \cdot n \cdot area(p) \quad (1)$$

The  $v$  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.

The idea of the dynamic occluder selection in the lazy ray casting can be sketched as follows. 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. 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.

### 3.3 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 [14].

To decrease the overhead from these empty pixels to its minimum level in the local Z-buffering 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) 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 of the AABB is computed at first and then the projections of the other vertices of the AABB are computed by adjusting the corresponding offsets [5]. The projection of an AABB is a simple convex polygon  $P_{aabb}$  with possible 4, 5 or 6 sides, which are corresponding to 1, 2 or 3 face visible cases [14]. 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.

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 in the projection region of an object having finished ray shooting. If the pixel 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.

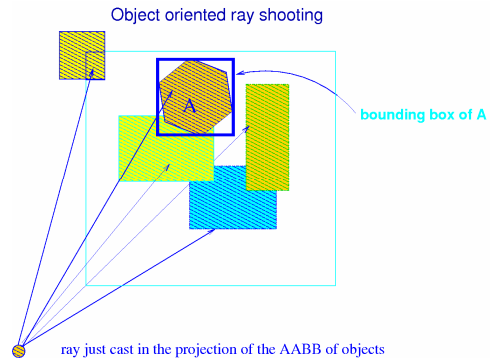


Fig. 3. The object-oriented ray casting

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.

### 3.4 Coherence ray-octree traverse

The current ray-octree traversal algorithm in the selectively lazy ray casting is a standard top-down traverse, which is by no means an efficient octree traverse algorithm. We give a coherence based ray-octree traverse algorithm to exploit the “sub-frame coherence” in the ray shooting.

If we see the “image” of each pixel is a sub-frame, there is some frame coherence [4,6] between the color from this pixel and the color from its neighbor pixels. So the images of two sub-frames are similar or same, which can also be explained by the image coherence [4]. According to such sub-frame coherence, we expect the ray paths of the rays emanated from neighbor pixels are same or similar. If we keep a cached ray path  $l_r$ , which records the ray traverse information, for a ray emanated from the previous neighbor pixel. On the other hand, for a ray emanated from the current pixel, we just test if its first intersection in the octree is in the first leaf node in the cached ray path  $l_r$ . If the answer is true, then a leave point is computed and the same query will be conducted for the next node in the cached ray path  $l_r$ ; if the answer is not true, a neighborhood search is conducted for the current ray path node to find the host leaf node or an empty leaf node for the candidate point. If the neighborhood search fails, the standard top-down search octree traverse is taken to find a host node. When the visibility status of a ray emanated from the pixel is obtained, the cached ray path  $l_r$  is updated for its next neighbor. Although there are many delicate ways to conduct the neighbor search in an octree, we do not take the popular H. Samet 's approach since it requires too much processing time and memory storages [15]. In our method, the neighbor search just finds the sibling leaves that are very easy to locate in the real time and no pre-stored information required. We call the ray-octree traverse method as the coherence based octree traverse. According to our implementation, the coherence based octree traverse algorithm achieves at least 50% optimization than the general top down octree traverse algorithm of the partition plane is set around 30% in the view frustum averagely [14].

### 3.5 The final version local Z-buffering rendering algorithm

The final local Z-buffering consists of the integrations of the acceleration techniques of naïve local Z-buffering from following aspects. 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 [14].

## 4 Rendering Experiments

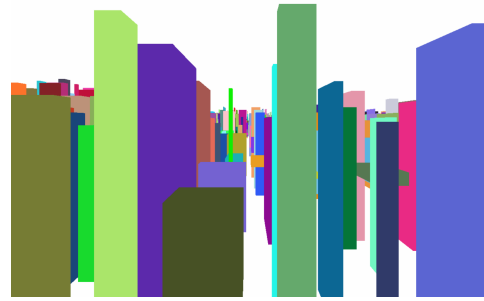


Fig. 4. A generated virtual city with 1.8 M triangles

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 (Figure 4). The general Z-buffering rendering needs 1.76 seconds averagely and the improved LZB just take 0.34 second for averagely sampled 40 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 [16]. 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 want to “prove” the output sensitivity of the LZB. A series of scenes, which are scenes scene triangle numbers range from 18,000 to 3.8 million, are taken from the UNC power plant model [16]. We check the LZB and Z-buffering performance related to the following rendering measures : image complexity, occlusion rates and scene depth complexity by sampling 40 viewpoints in a circular view path. The image complexity is the number of projected pixels in the image space  $\Omega$  ; the occlusion rate measures how many pixels are occluded in the image space and the scene depth complexity is approximated by the ratio of the image complexity and final image resolution. From the rendering results in Figure 5, we can see that our LZB is weakly dependent on the image complexity, the occlusion

rate and scene depth complexity generally. On the other hand, we can see the Z-buffering is input sensitive with these rendering measures. It is clear that the local Z-buffering rendering time increases rather weakly with the increase of the number of triangles in the input scene compared with the linear increase for the Z-buffering. This point can be seen from the relationship between the two rendering approaches with respect to scene occlusion ratio and the relationship between the scene occlusion rate and scene size. All our experiments are done under a Pentium (IV) machine running under Linux with 1.8 GHz CPU and 1.0 G main memory and Gf4 series graphics hard under Linux OS and the final image resolution is  $512 \times 512$ .

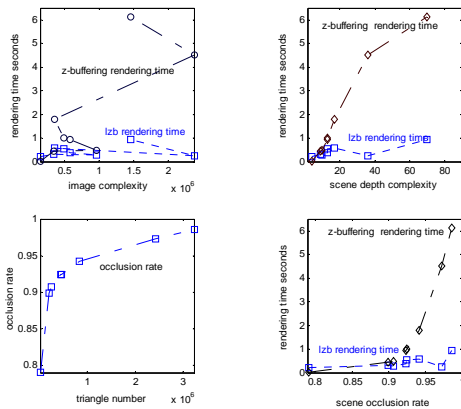


Fig. 5 Local Z-buffering rendering results

## 5 Conclusions

In this paper, we present the LZB as an output sensitive real-time rendering algorithm in large complex scenes. It is easy to implement and supported by most graphics hardware. There is no special hardware and heavy preprocessing requirement for such an algorithm. In the following work, we would like to compare the performance of the LZB and other rendering algorithms in large complex scenes like HZB (hierarchical Z-buffering), HOM (hierarchical occlusion culling) algorithm besides the polish our current version parallel LZB algorithm.

## 6 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), pp. 61-69, 1991,

[4] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility", *SIGGRAPH 93 Proceedings*, Annual Conference Series, pp. 231-238, 1993.

[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*, pp. 71-82, 2000.

[7] S. Coorg, and S. Teller, "Real-time occlusion culling for models with large occluders". *ACM Symposium on Interactive 3D Graphics*, pp. 83-90, 1997.

[8] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative visibility preprocessing using extended projections", *Proceedings of SIGGRAPH*, pp. 239-248, 2000.

[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 Graphics* 7(2), pp. 365-379, 2001.

[11] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and E. Huebner, *Level of Detail for 3d Graphics*, Morgan Kaufmann, 2002.

[12] M. Sainz, and R. Pajarola, "Point-based rendering techniques", *Computer & Graphics*, 28, pp. 869-879, 2004.

[13] T. Akenine-Miller, "Fast 3D triangle-box overlap testing", *journal of graphics tools*, Vol. 6, no.1, pp. 29-33 2001.

[14] 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.

[15] H. Samet, *Applications of Spatial Data Structures*, Addison-Wesley, 1990.

[16] The UNC power plant model: <http://www.cs.unc.edu/~geom/Powerplant/> 2004.