Conservative Occlusion Culling for Urban Visualization using a Slice-wise Data Structure

Türker Yılmaz, Uğur Güdükbay

e-mail:{yturker,gudukbay}@cs.bilkent.edu.tr

The authors are with the Department of Computer Engineering,

Bilkent University, 06800, Bilkent, Ankara, Turkey.

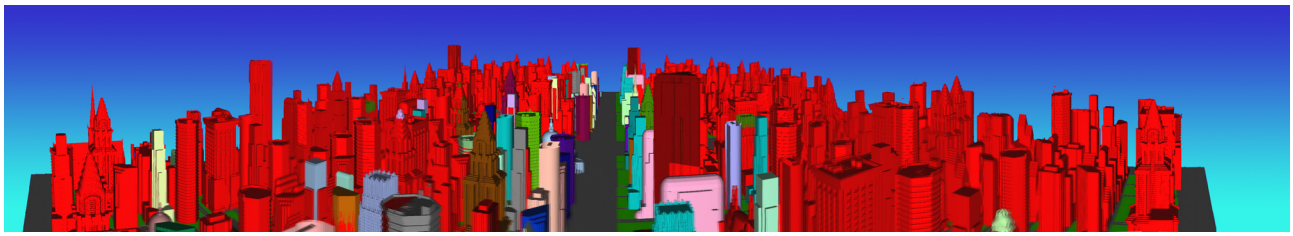


Fig. 1. An occlusion culling algorithm using our slice-wise data structure sends approximately 54 % fewer triangles to the graphics pipeline and increases frame rate by 46 %, as compared to an occlusion culling algorithm using building-level granularity of visibility in this 1.5M-triangle environment. The red colored sections show occluded regions, which are discarded from the graphics pipeline. In addition, the data structure decreases Potentially Visible Set (PVS) storage requirement drastically.

Abstract

In this paper, we propose a new shrinking process for conservative from-point occlusion culling algorithms and a data structure for the visualization of urban environments. The visible geometry in a typical urban walkthrough mainly consists of partially visible buildings. Occlusion-culling algorithms, in which the granularity is based on buildings, render these partially visible buildings completely. We observe that the visibility in urban walkthroughs shows certain characteristics. The proposed *slice-wise data structure* represents the buildings, exploiting these characteristics, in terms of slices parallel to the coordinate axes. This forms the base for occlusion culling where the occlusion granularity is at slice level. The proposed slice-wise data structure has minimal storage requirements. The visible parts can be accessed at constant time during navigation with the help of a preprocessing stage. We also propose to shrink the occluders in a scene. This is necessary for a conservative from-point occlusion culling algorithm, which can also be applied to nonconvex general 3D occluders. Empirical results show that a 54 % decrease in the number of processed polygons and 46 % speed-up in frame-rate can be achieved by using the proposed conservative occlusion-culling algorithm with rather than an occlusion-culling method where the granularity is based on individual buildings.

Index Terms

Data structures, occluder shrinking, from-point occlusion culling, urban visualization, visibility preprocessing.

I. Introduction

The efficiency of the visibility algorithm is vitally important for making an urban visualization system usable on ordinary hardware. View-frustum culling and back-face culling are ways to

speed-up the visualization, and for them, the current algorithms seem to be efficient enough. However, occlusion-culling algorithms are still very costly.

In occlusion-culling algorithms where the granularity is individual buildings, an object could be sent to the graphics pipeline even if a small portion of it becomes visible. In most cases, this would result in unnecessary overloading of the hardware, especially if the objects are very complex, containing hundreds of thousands of polygons. An efficient approach is needed to create a tight visibility set without causing further overheads. Although it is feasible to traverse the nodes in the hierarchy of an object to see which parts are visible, it is usually impractical to store the visibility lists.

The first contribution of this paper is *the slice-wise structure*. This is a simple data structure particularly suitable for urban scenes. It automatically exploits real-world occlusion characteristics in urban scenes by subdividing the objects into slices parallel to the coordinate axes. Other object hierarchies such as octrees and regular grids can well be used to partition the objects; even individual triangles can be checked. However, the storage of Potentially Visible Sets (PVSs) limits the scalability of their usage. The PVS storage requirement of the proposed slice-wise structure is very low (as low as three bytes for each viewpoint and partially visible building). An index is stored for a partially visible building, indicating the visible slices along each coordinate axis.

The second contribution of the paper is a shrinking algorithm developed on the principles of conservative from-point occlusion culling. In order to achieve conservative from-point visibility, the occluders need to be shrunk and the test must be performed on the shrunk versions of the occluders. To our knowledge, this is the first demonstrated attempt that can also be applied to general nonconvex occluders.

The organization of the paper is as follows. We give related work in Section 2. In Section 3, we describe the proposed slice-wise data structure. In Section 4, we describe using the proposed slice-wise structure for occlusion culling based on conservative from-point visibility, thereby describing our shrinking algorithm. In Section 5, we give experimental results and comparisons. Finally, we give conclusions.

II. Related Work

Scene representation has a crucial impact on the performance of the visibility algorithm in terms of memory requirement and processing time. Many data structures have been adopted for scene and object representation such as octrees [1], or scene graph hierarchy [2]. Scene graph usage that provides fast traversal algorithms is particularly popular [3]. However, these are useful mainly for the definition of object hierarchies. Their use in determining visibility may require them to be augmented with additional information, thereby increasing their storage requirements. In addition, the natural object structure is modified in some applications. In [4], the triangles that belong to many nodes of the octree are subdivided across the nodes for easy traversal. In [5], the objects could be divided into subobjects to create a balanced scene hierarchy, if necessary.

Occlusion-culling algorithms detect the parts of the scene that are occluded by other objects and do not contribute to the overall image; these parts should not be sent to the graphics pipeline. They can be classified as either *conservative* or *approximate* [6]. Conservative algorithms may classify some invisible objects as visible but never call a visible building invisible. Instead of traversing an object's internal hierarchy for fine tuned visibility, most conservative algorithms either accept the entire object as visible or reject it. Approximate occlusion-culling algorithms, such as [7], [8], [9], the visible primitives are rendered up to a specified threshold, i.e., some of them may not be sent to the graphics pipeline although they are visible. There are also approaches to occlusion culling that use parallel processing approaches, such as [10], [5], [11].

Exact visibility algorithms provide accurate images at the expense of degrading rendering performance and increasing storage requirements. An example of this class is [12], where the authors make use of a 5D subdivision of the polygons. A detailed survey on occlusion-culling algorithms can be found in [6].

Applications where the scene objects are culled and stored using octree structures are mainly suitable for scenes where there are large occluders and a large portion of the model is behind these occluders. Visibility determination by traversing a scene hierarchy requires the quick selection of occluders; this is a difficult task [10], [13], [14], [15], [16], [17], [18].

The proposed slice-wise structure can be used to create a tight visibility set of slices of objects

for any kind of occlusion-culling algorithm. The visibility set thus produced is smaller than those that measure occlusion at the building level, but larger than the exact ones that operate at the polygon level: it groups polygons by exploiting visibility characteristics in a typical urban walkthrough.

The purpose of clustering visibility is to improve scalability. There are many different approaches to compressing data, such as [19], [20], [21], [22], [23]. Our slice-wise data structure naturally decreases the amount of information that needs to be stored. Additional compression schemes may further increase its advantages.

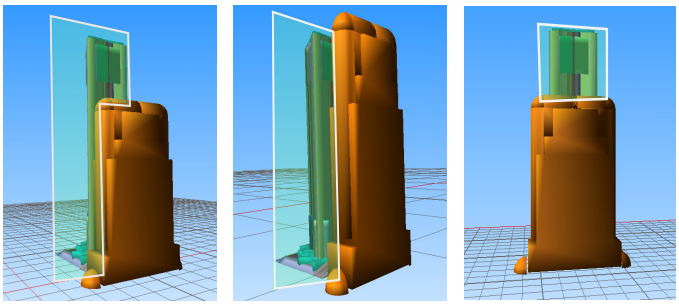
III. Slice-wise Structuring of Objects

A. Object Visibility Characteristics

Our slice-wise approach is based on the observation that while a person is navigating through a city, the visible parts of the objects usually have one of the following three forms (see Figure 2):

- The visible part looks like an *L-shaped* block in different orientations if a building is occluded in part by a smaller occluder, as in Figure 2 (a).
- The visible part looks like a *vertical rectangular* block, from the left or right of the building if the occluder is taller than the occludee (see Figure 2 (b)).
- If the occluder is a large one and appears to be shorter than the occludee, it usually hides the lower half of the building and the visible portion looks like a *horizontal rectangular* block, as in Figure 2 (c).

If the visible part of a building has a form other than the ones listed above, then we can assume that it is completely visible. Obviously, a visibility-culling algorithm could be developed without characterizing visible parts of the buildings. However, this might send unnecessarily large number of polygons to the graphics pipeline. If we could find a way to exploit these visibility characteristics with a little overhead, we could reduce both the number of polygons sent to the graphics pipeline and storage requirements for the PVSs. This is what we achieve with the proposed slice-wise structure.



(a) (b) (c)

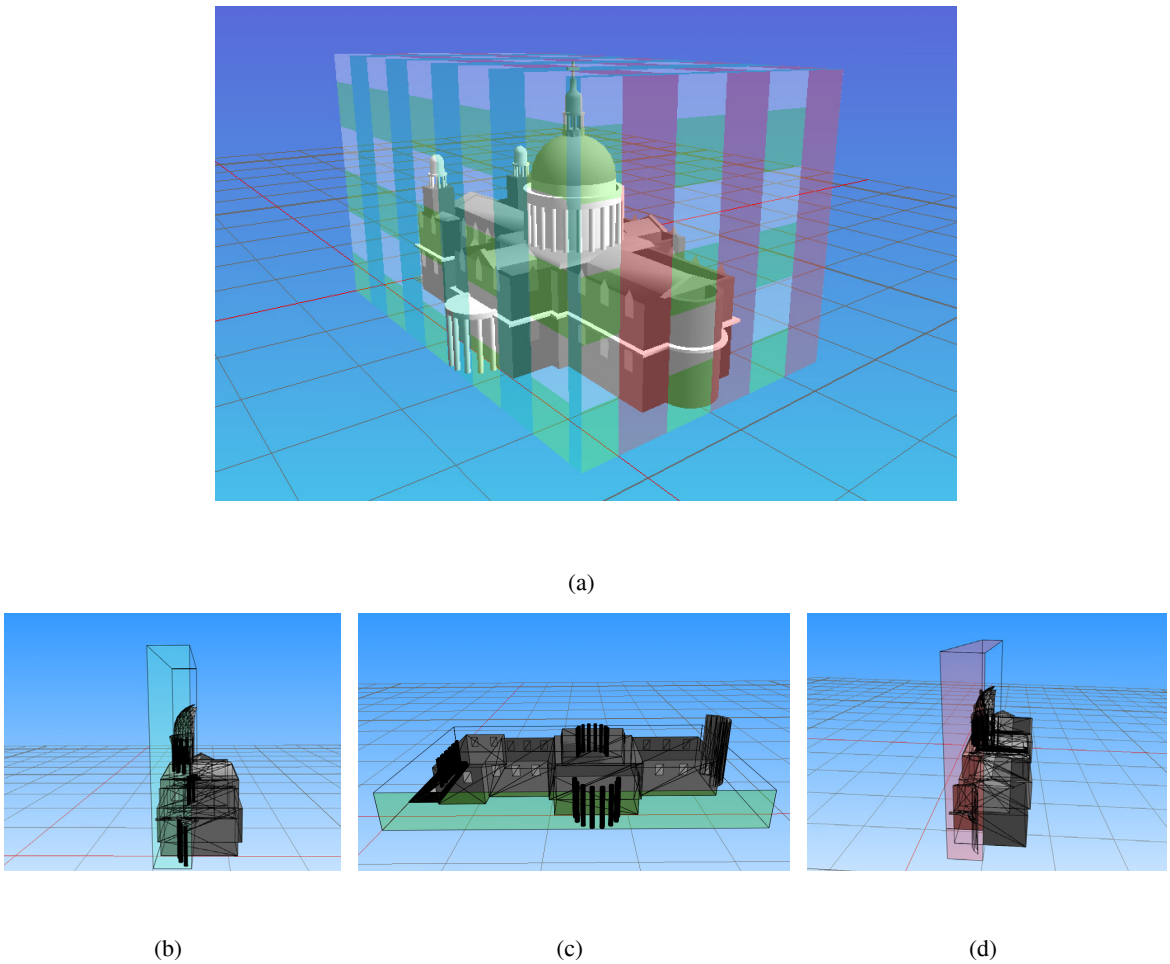
Fig. 2. Visibility forms during urban navigation: (a) *L-shaped* form; (b) *vertical rectangular* form; (c) *horizontal rectangular* form. In each part of the figure, the visible part of the occludee is the green transparent area.

B. Slicing Objects

The aim of the proposed slice-wise structure is to create tight PVSs for urban scenes. Slicing is an object representation in which an object is divided into slices parallel to the 3 coordinate axes and each slice knows which triangles belong to it. The slicing process is composed of two steps: first, each object is uniformly subdivided and the grid cells occupied by each triangle are determined. The triangles are tested against grid cells and the cell occupancies of triangles are determined. Next, the occupied cells for each object in the uniform subdivision are combined into axis-aligned slices for each coordinate axis. The process of slicing an object is shown in Figure 3. The resultant data structure is shown in Figure 4.

C. Benefits of Slicing

Here, we compare the proposed structure with octrees and regular grids. We use the same granularity for the compared subdivision schemes. In Table I, we depict subdivision depth and the number of nodes needed for each subdivision. The number of nodes for octrees refers to regularly subdivided octree. In an adaptively subdivided octree the number of nodes is below these levels. However, giving exact costs and approximations on adaptive versions is very difficult. A comprehensive study of the costs for various construction schemes of octrees is presented in [24].



(a)

(b) (c) (d)

Fig. 3. The process of slicing an object determines the triangles that belong to each slice. (a) a complete view of the object where the positions of slices are shown; (b) an x–axis slice; (c) a y–axis slice; (d) a z–axis slice.

For precomputed visibility, the size of the data stored for the view cells may become so large that the total size of the PVSs is much larger than the size of the scene. Aside from a few studies [23], [25], the problem of big PVS storage problem has not been given enough importance [6].

The slice-wise structure provides an efficient way to store PVSs. Visibility for each partially visible object is encoded to three bytes. PVS costs in bytes are depicted in Table II for various scene and object sizes, which are *visible* or *partially visible*. We assume that each node of octree and regular grids can be identified with 1 byte and we discard additional information that is stored along with them, such as corner coordinates. The pointers to polygons are not

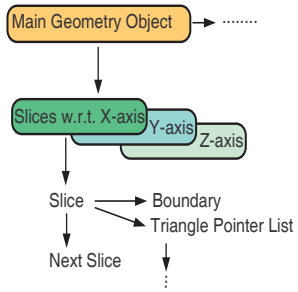


Fig. 4. The scene data structure produced by slicing operations. Boundary information that describes the two diagonal corners of the slices are needed only during the preprocessing phase.

TABLE I

THE COMPARISON OF THE NUMBER OF NODES NEEDED IN SLICE-WISE, OCTREE AND REGULAR GRIDS.

Depth	Slice-wise	Octree	Regular Grids
1	3x2=6	8	8
2	3x4=12	64+8=72	64
3	3x8=24	512+72=584	512
4	3x16=48	4,096+584=4,680	4,096
5	3x32=96	32,768+4,680=37,448	32,768

taken into account, because these are needed in all types of structures. For the worst case of the slice-wise structure, we assume that all the objects are partially visible. Since the order is implicitly determined by giving the slice index (see Figures 4 and 5), we do not have to store each slice explicitly. Additionally we provide the data needed to do occlusion culling at the polygon level, assuming that the visibility of each triangle is encoded in bits. However, since the scene size is indicated by the number of visible triangles and objects, the data needed to be stored for polygon-level occlusion culling may be much larger than those given in the table and using another data structure becomes necessary. The table shows that the slice-wise structure requires much less space to store the PVSs; this is an indispensable part of most preprocessed occlusion-culling algorithms. Other compression schemes may be used to further decrease the amount of data to be stored.

Slicing the objects provides a fast way to access visible portions of an object. Defining the visible portions requires determining the visible slices, as shown in Figure 5. We only need to

TABLE II

PVS STORAGE COMPARISON (BYTES/VIEW CELL)

Triangles	Objects	Depth	Slice-wise	Octree	Regular Grids	Triangle Level
100K	10	1	30	80	80	12.5K
		2		720	640	
		3		5,840	5,120	
		4		46,800	40,960	
		5		374,480	327,680	
1M	100	1	300	800	800	125K
		2		7,200	6,400	
		3		58,400	51,200	
		4		468,000	409,600	
		5		3,744,800	3,276,800	
10M	1000	1	3K	8K	8K	1,250K
		2		72K	64K	
		3		584K	512K	
		4		4,680K	4,096K	
		5		37,448K	32,768K	

store the last visible-slice index for each axis. Visible-slice indices are assigned to each axis after they are checked for occlusion.

In addition to facilitating the exploitation of different visibility characteristics for tight visibility processing (see Figures 1 and 2), the benefits of slicing objects are:

- Each triangle is encoded in at least 3 slices in different axes. Therefore, we can use slices on any axis during visualization. We choose the axis with maximum occlusion (see Figure 14). Choosing the maximally occluded axis allows us to tighten the visible set as much as possible, and decreases the information that needs to be stored.
- The memory required for the slice-wise approach is minimal. In order to define the visibility, three bytes, one for each axis, are used for each object and for each viewpoint. These three-byte data specify the indices of the visible slices for each axis. Defining visibility for each object by using only three bytes greatly decreases the storage requirements for PVSs (see Table II).

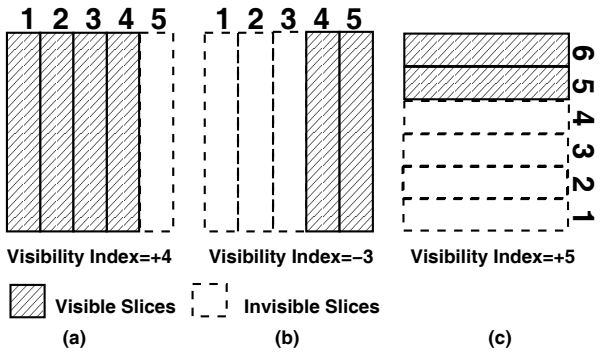


Fig. 5. Defining visibility indices for objects: the visible slice indices are determined for each axis during occlusion determination. (a) If an object is partly occluded from the right, the index of the last visible slice is stored with a "+" sign. (b) If the object is occluded from the left, then the index of the last invisible slice is stored with a "−" sign. (c) If the object is occluded from the bottom, we keep the first visible horizontal slice of the object.

- Unnecessarily traversing a tree-like data structure is prevented by directly accessing the visible slices and hence triangles of an object, thereby gaining CPU time.

Using individual triangles and testing for occlusion is a good way to create the tightest possible visibility set for any point in the scene, and at first it may sound better than the approach presented here. However, the PVS storage issue becomes a big problem, and limits the scalability. The slice-wise structure creates a good balance between PVS storage and running time.

IV. Application of the Slice-wise Structure to a Point-based Conservative Visibility Framework

Figure 6 shows the framework for urban visualization using the slice-wise representation. It mainly consists of a preprocessing phase and a navigation phase. In order to test the applicability of the slice-wise data structure, we have developed a conservative from-point visibility algorithm. In order to achieve conservative occlusion culling, we made use of the shrinking idea first proposed by Wonka et al. [10]. Since in our models we do not limit model complexity, we developed a new shrinking method, which can be applied to any kind of scene object, including nonconvex ones.

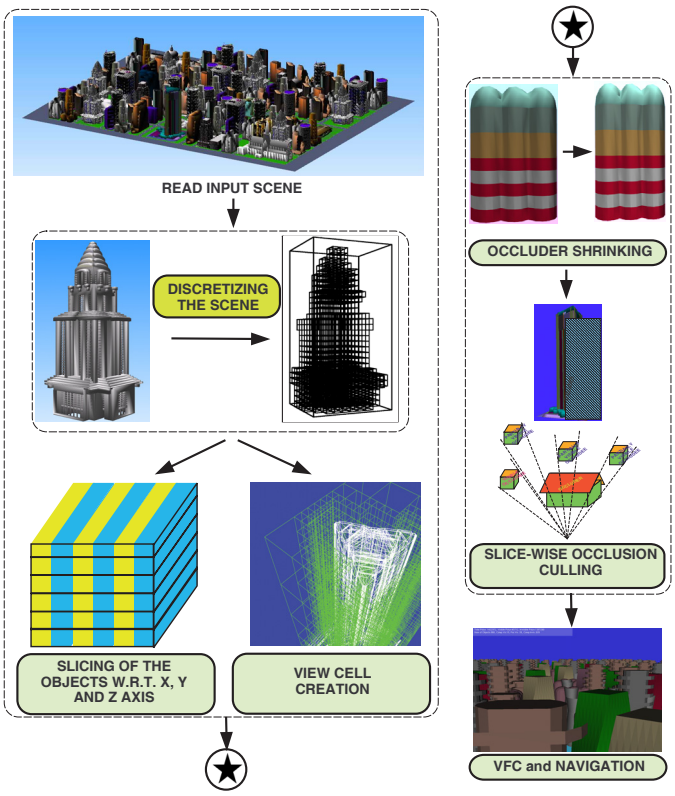


Fig. 6. The urban visualization framework: in the first phase, we read the scene and calculate the bounding boxes of objects. Next, we discretize each object by checking if each triangle intersects with a predefined threshold-sized cube. After discretizing the object, we check the cubes for fullness to create slices and create the tree of octrees of the bounding-boxed object. These are used during preprocess as viewcells. After creating the shrunk versions of the objects, these slices are checked for occlusion and a tight visibility determination is performed for each grid location. The phases in dashed blocks are performed in the preprocessing phase. The View Frustum Culling (VFC) is also done during navigation.

A. *Occluder Shrinking*

The purpose of shrinking is to achieve conservative occlusion culling by sampling from discrete locations in the scene. In this approach, it is possible to determine occlusion from a point and retain conservativeness for a limited area, because the occluders are shrunk by the maximum distance that can be traveled in the view cell. In order to achieve conservativeness, it is necessary to shrink occluders so that behind the occluder is visible even if the user moves to the farthest possible location in the view cell.

Wonka et al. shrink occluders by using a sphere constructed around 2.5D occluders. In [26],

instead of a sphere, the authors generalize the shrinking by a sphere to the erosion by a convex shape, which is the union of the "edge convex hulls" of the object, in order to create tighter visibility sets and increase the occlusion power of the objects. They compute the shrunk versions of objects using an image based algorithm at each view cell. These two approaches are valid only for 2.5D environments. Calculating shrunk versions of each object at each visibility determination location by using a voxelized representation, makes it very difficult to apply the presented image-based approach to general 3D objects.

1) Shrinking General 3D Objects: In fact, the exact shrinking can only be performed by using Minkowski sums of the viewcell and the object [27], [28] and using the volume constructed inside the object (see Figure 7).

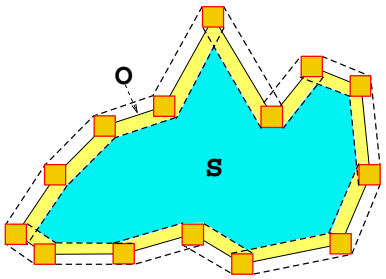


Fig. 7. Minkowski sum calculation and shrunk volume extraction of an object O and a view cell. The object O is illustrated by the outer contour of the yellow part and the shrunk version S by the blue part.

Consider a general 3D object O and set of vectors X of a view cell. The dilation of O by X, also known as the Minkowski sum of both sets is defined by the equation:

$$O \oplus X = \{M + x \mid M \in O, x \in X\}$$

Here, X is commonly called the *structuring element* [26]. Thus, the inner volume, which composes the shrunk shape S of the object, can be defined as:

$$\begin{aligned} O \ominus X &= \{S \mid \forall x \in X, S + x \in O\} \\ &= \{S \mid \{S\} \oplus X \subset O\} \\ &= \{S\} \subseteq \{O \ominus X\} \end{aligned}$$

Unfortunately, it is very hard to find the exact Minkowski sum for general 3D objects. Instead we try to find an approximation to it, which will also satisfy the conservativeness of the occlusion-culling process. Unlike previous versions of the occluder-shrinking approaches, we can apply it to any complex 3D object.

2) Shrinking Using the Minkowski Sum Concept: We shrink an object by moving the vertices in the reverse direction of their normals. Our aim is to use an approximate Minkowski sum of the viewcell and the object and still achieve conservative occlusion culling. It should be noted that we do not move the vertices with a constant distance (see Figure 8).

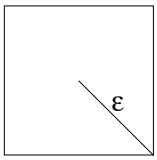


Fig. 8. A sample view cell, in which the user can move at most with ε distance.

The travel distance of a vertex affects the final position of a face and for the exact Minkowski sum calculation. The movement distances of the faces towards the inner space varies with respect to the orientation and position of the viewcell as shown in Figure 9 (a).

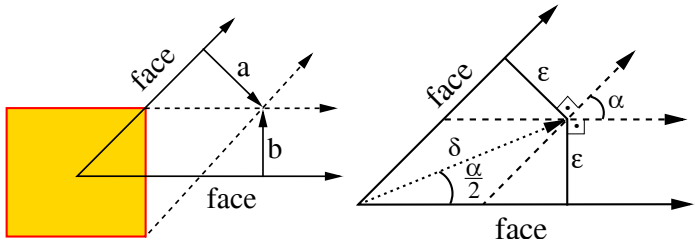


Fig. 9. Shrinking using the Minkowski sum concept: (a) In an exact Minkowski sum calculation, the movements of the faces to the inner section are different, with respect to the view cell position. The two faces move with distances a and b, (b) The face movements towards the inside of the object should be at least the distance ε of Figure 8, in order to guarantee conservativeness. In this case, the vertex movement distance to the inner part becomes δ. For easy interpretation, only an instance of the process where two faces that share a vertex is shown.

The hard part in calculating exact shrinking using the Minkowski sum concept is calculating the

distances a and b for any orientation and posture of the view cell and the faces of the object, shown in Figure 9 (a). Instead we provide an approximation for it (see Figure 9 (b)). If we assume an identical movement distance for the faces sharing a vertex, the vertex movement distance becomes a function of the maximum user movement from the center of the view cell.

Theorem 1 (Conservative Shrinking): Let O be the occluder, S be its shrunk shape, and ε be the maximum travel distance from the center of the view cell. If the minimum shrinking distance of O is greater than or equal to ε, the determined visibility from the center of the view cell by using the shrunk shape of the occluder provides a conservative estimate for the whole view cell (see Figure 10).

Proof: According to the Minkowski sum theorem the shrunk shape S is $\{S\} \subseteq \{O \ominus X\}$. Let X_d be the vectors of length d, which is smaller than ε and is the correct distance calculated using the Minkowski sum, that is $d = a$ or $d = b$ (see Figure 9 (a)). Hence, $\{S_d\} \subseteq \{O \ominus X_d\}$. Since ε is the maximum distance to be moved, then $\{a, b\} \leq \varepsilon$. Then, the volume of $\{S_d\}$ is greater than or equal to the volume of $\{S_\varepsilon\}$. Consequently, if $\{S_d\}$ is conservative, then $\{S_\varepsilon\}$, having a smaller volume, is definitely conservative.

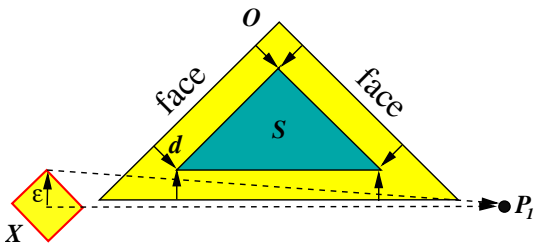


Fig. 10. If a point P_1 is visible from the center of the view cell, then it should also be visible with respect to its shrunk version S, even if the user moves to the farthest distance available in the view cell, ε. If it is defined that the inner movement distance d of the faces for the shrunk shape calculation is greater than or equal to ε, the conservativeness is guaranteed and the point becomes visible with respect to the shrunk version S. The reader is referred to [10] for the proof of the other case: if a point is occluded with respect to the shrunk version S, then it is also occluded with respect to its original version O, within an ε neighborhood.

3) Calculating Shrinking Distance for the Vertices: Using the notations of Figures 9 (b) and 11,

$$\frac{\alpha}{2} = min\left\{90 - arccos\left(\frac{\mathbf{N}_v \cdot \mathbf{N}_i}{|\mathbf{N}_v| \ |\mathbf{N}_i|}\right)\right\}, i = 1, 2, \ldots, n$$

where n is the number of faces sharing that vertex, $\mathbf{N}_v$ is the vertex normal, and $\mathbf{N}_i$ is the face normal. Then the shrinking distance of the vertex becomes $\delta = \varepsilon / sin(\frac{\alpha}{2})$. In order to calculate δ, we calculate the minimum angle between the vertex normal and all the neighboring face normals, since it guarantees conservativeness.

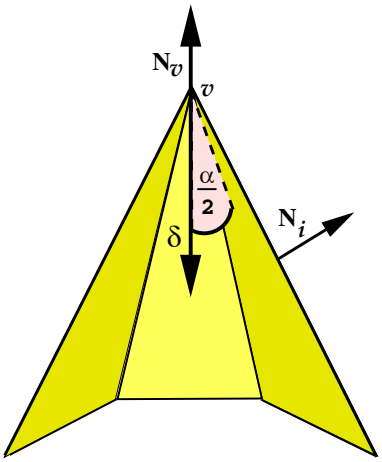


Fig. 11. The object is shrunk by moving the vertex in the opposite direction of the vertex normal. The shrinking distance δ is calculated based on the view cell parameter ε and the angle $\frac{\alpha}{2}$.

4) Shrinking Occluders: The calculation of a correct shrinking distance is not enough to create conservative shrunk versions of the occluders; some faces may go inside one another and invalidate conservativeness. To prevent such cases, we check the intersection of the volumes formed by the faces in the occluder and the corresponding faces in the shrunk version. We first check the intersection of their axis aligned bounding boxes. If the axis aligned bounding boxes intersect, we try to find a supporting plane between the volumes. If there is a supporting plane between them, the volumes do not intersect. Otherwise, we remove the faces that cause intersection from the shrunk object, as well as other bad cases such as triangles with no area. A shrinking example with increasing δ values can be seen in Figure 12.

B. Occlusion Culling

The occlusion-culling algorithm works in the preprocessing phase. It regards each scene object as a candidate occludee and performs an occlusion test with respect to all other objects in the scene.

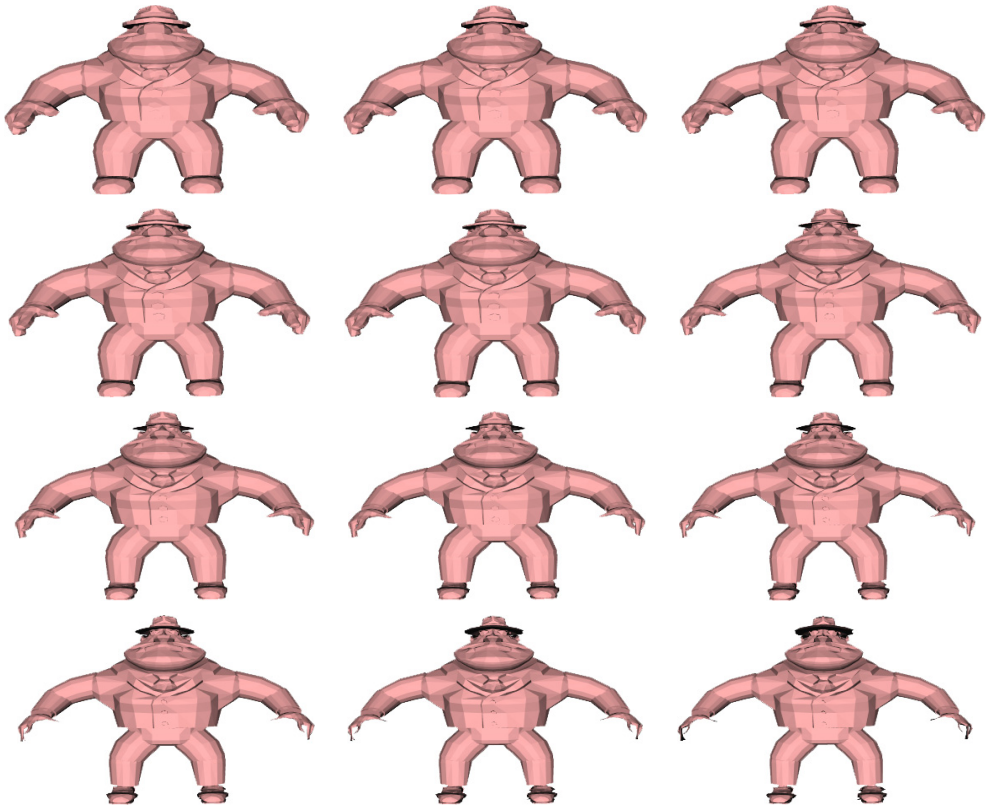


Fig. 12. Shrinking applied to a general 3D object: The δ values are increased manually to show the effect of the scheme described here (the pictures are in rowwise order). It should be noted that the vertices touching the ground are not moved vertically. Al Model is Courtesy of Viewpoint Datalabs International, Inc.

At each step, slices of the horizontal axis are checked for complete occlusion. The other two axis slices are checked for partial occlusion. The slices of an object are tested with a combination of all other objects' shrunk versions; this creates occluder fusion and determines the occlusion amounts. This process is repeated for each navigable grid location.

The pseudo-code of the occlusion culling algorithm is given in Algorithm 1. The algorithm tests buildings in a back-to-front fashion to detect a completely invisible building in an earlier stage. We start from the farthest building as a candidate occludee and construct a frustum where the viewpoint is the center of projection. First, the scene objects in the frustum other than the occludee that were not accepted as invisible are drawn in their shrunk versions and the bounding box of the occludee is tested. Most occlusion-culling algorithms stop after this step and accept an object as visible if the occludee becomes partially visible. We go through further steps and

```
foreach grid location in 3D do
    Construct frustum towards the center of the occludee;
    Draw the shrunk versions of the noninvisible objects in the frustum as occluder;
    foreach object in the back-to-front traversal do
        Test the bounding box of the occludee using NV_OCCLUSION_QUERY;
        if the bounding box of the occludee is visible then
            Mark the object as visible
        else
            Mark it as invisible
    foreach VISIBLE object do
        Test slices using Algorithm 2;
    foreach PARTIALLY_VISIBLE object in the scene do
        Optimize visible slice counts using Algorithm 3;
```

Algorithm 1: The occlusion-culling algorithm: this algorithm differentiates between visible and invisible occludees. Then the visible objects are sent to the slice-wise occlusion culling algorithm. Next the number of visible slices are optimized and the PVS for the view cell is determined.

determine a tighter visibility set for the object. If the bounding box of the occludee is visible, we submit occlusion queries using the NV_OCCLUSION_QUERY extension of NVidia graphics chip sets for the slices; we then determine the maximum occlusion height for each slice of the occludee using Algorithm 2. The visibility information for the slices is sent to Algorithm 3 in order to decrease the number of slices to be used during navigation.

The tight occlusion-culling test is given in Algorithm 2. It checks the slices of a candidate occludee. To find the exact occlusion, we first submit occlusion queries by using the NVidia OpenGL extension, in order to test the vertical slices with blocks of size Δ (see Figure 13). Horizontal slices are checked for complete occlusion. Next, we collect the query results. Finding the last invisible Δ allows us to determine the occluded height of the slice box and from this the visibility status of the slice and the object.

C. Optimizing the Visible Slice Counts

An object that is occluded by several occluders may have an irregular appearance that is not easily represented (see Figure 14). However, our aim is to decrease the amount of information

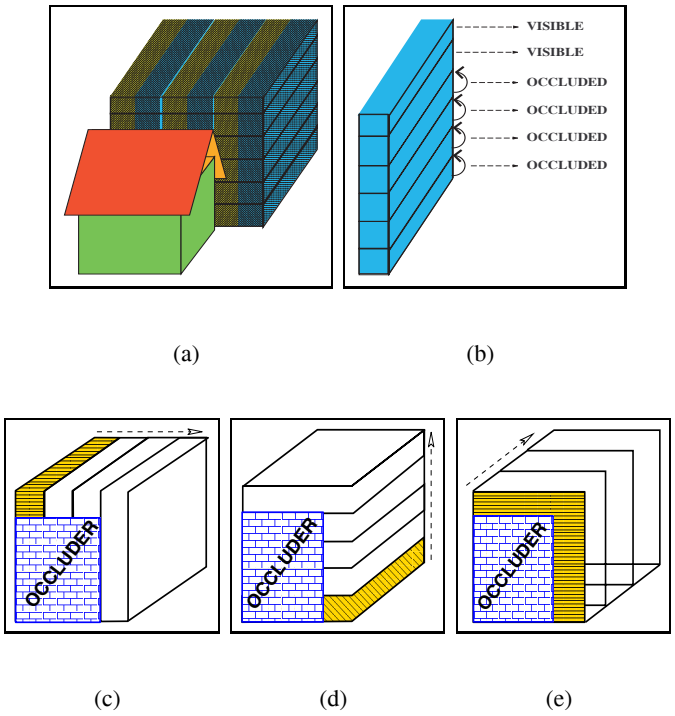


Fig. 13. In order to determine the correct occlusion height that occurs as in *(a)*, the slices are tested beginning from the lowest unoccluded height and the point of occlusion is found as in *(b)* by iteratively eliminating blocks of size Δ from the vertical slice; this creates occluder fusion (The view is drawn as if looking from the upper right). The test order for slices along the x, y, and z-axes are depicted in *(c-e)*, respectively. While the slices in the x and z-axes are tested for exact heights, the y-axis slices are tested for complete occlusion (d). Testing y-axis slices for complete occlusion may result in unnecessarily accepting the slice as visible. However, this case is handled by optimizing the slice counts.

needed to represent visibility, and therefore reduce the time to access the visible parts of the objects. In particular, the purpose of optimization is to represent the visible area by using a small number of slices. We have to sacrifice tightness of visibility somewhat to reduce the access time and memory requirement (cf. Figure 14).

The algorithm for optimizing the slice counts is given in Algorithm 3. This algorithm is used to decrease the number of slices used to represent the visible portion of an occludee. In this algorithm:

- Any triangle of the object is represented by slices from three axes. We first find the maximally occluded axis by calculating the occluded regions and the percentages of occlusion

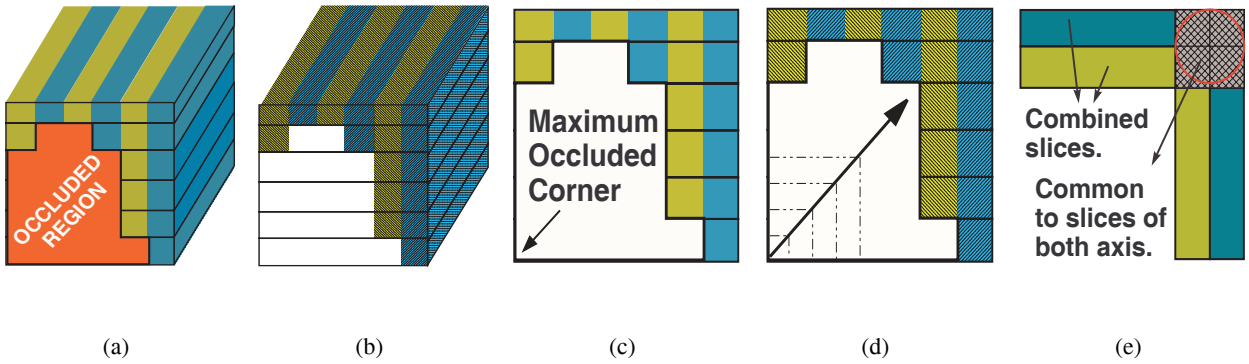


Fig. 14. The resultant shape of the occlusion may have a jaggy appearance and we need to smooth it to represent with the slice-wise structure (a and b). This is handled as described in Algorithm 3. After selecting the maximally occluded axis, the starting corner of occlusion is determined (c). The rectangle to represent the occlusion is determined (d). The vertical slices up to vertical edge of the rectangle and horizontal ones up to the horizontal edge are discarded (e).

with respect to each axis.

- The rectangle that represents the occluded area is constructed.
- For all the slices of the maximally occluded axis, we discard the vertical ones up to the vertical edge of the rectangle and the horizontal ones up to the upper edge.
- The region above the upper edge of the rectangle is represented using horizontal slices, and the region on the right- or left-hand side of the rectangle is represented using vertical slices.
- Finally, we discard the slices of the minimally occluded axis.

It should be noted that the visibility information for each partially visible object is represented using only three bytes, one byte for the visibility along each axis. If the occlusion of an object appears in the middle part, we accept the object as completely visible, because this occlusion cannot be represented by our slice-wise structure (see Figure 5).

D. Rendering

We perform visualization by creating display lists for the view cell on the fly and updating them when the user passes to another view cell. We avoid multiple renderings of the triangles at the intersections of the horizontal and vertical axes of the partially visible objects (see Figure 14 (e)). In our tests, we observed that multiple renderings of triangles brings approximately 15 %

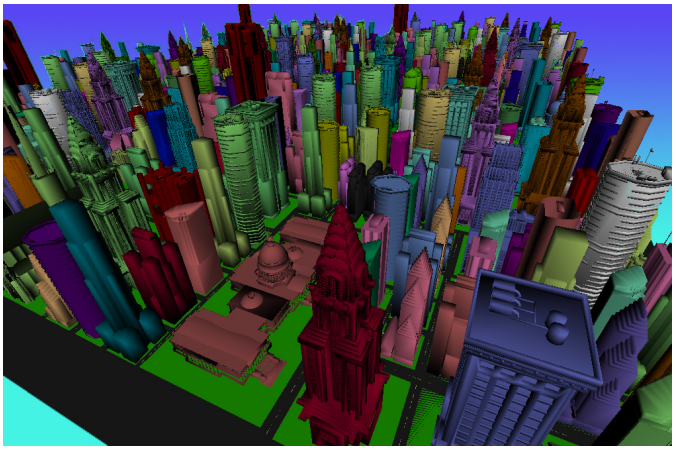


Fig. 15. 1.5M-triangle urban model used in the experiments.

overhead to the rendering process. We can compensate for this by using other rendering schemes such as Vertex Buffer Objects of modern graphics cards.

The navigation algorithm is supported by a view-frustum-culling algorithm, which eliminates the objects that are completely out of the view frustum.

V. Results and Discussion

The proposed algorithms were implemented using *C language* with *OpenGL* libraries; they were tested on two different PC platforms: an Intel Pentium IV- 3.4 GHz. computer with 4 GB of RAM and NVidia Quadro Pro FX 4400 graphics card with 512 MB of memory; and an Intel Pentium IV-2.0 Ghz. computer with 1 GB of RAM and NVidia TNT-2 graphics card with 32 MB of memory.

The navigation area is divided into 40-pixel grids. The area of the city is 18,000x7,000 pixels. There are about 83K grid points in the navigable area, from where the visibility culling is done. The city model used in experiments consists of 1,000 complex buildings with different architectures, each having from 1K to 10K polygons with a total of 1.5M polygons. The slices are 40 pixels wide, the same width as the grid cells, although they can be different to adapt to the dimensions of the buildings. On average, there are 30 slices on the x and z axes. The slices

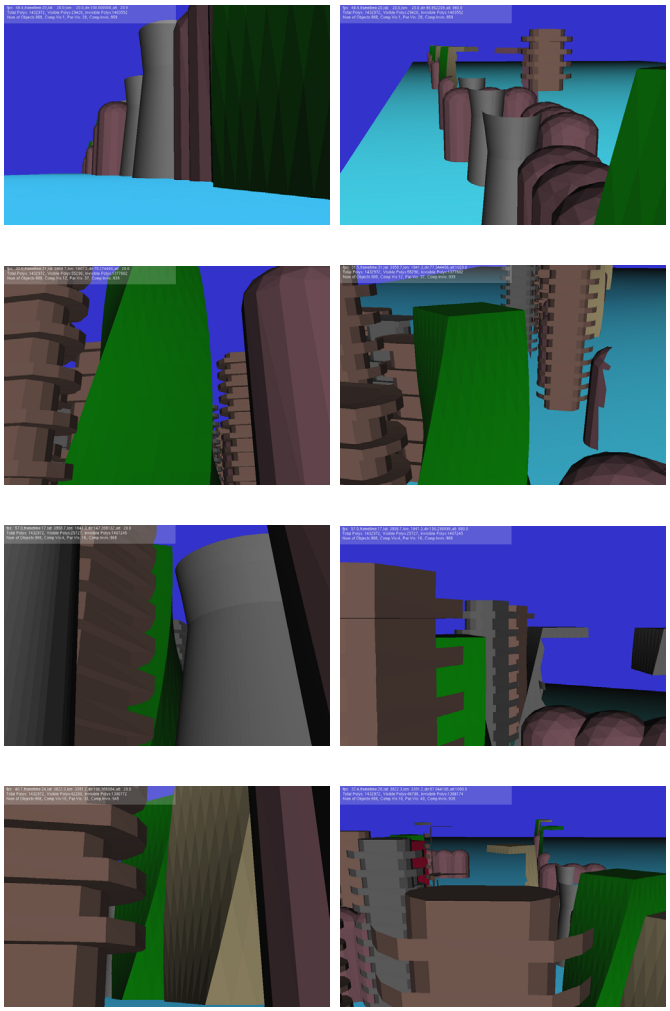


Fig. 16. Still frames from a navigation through the scene used in the experiments. On the left side, still frames from the current viewpoint are shown. On the right side of each frame, the view from above the user position shows the result of the occlusion-culling process.

of the y axis depend on the heights of the buildings, which in our case is around 40 slices. As a result each object has about 100 slices. Preprocessing took 3,060 minutes (2 days and 9 hours), the equivalent of about 2.5 seconds/viewpoint. We prepared a navigation of the scene containing about 20,000 frames (Figures 15 and 16).

The purpose of the empirical study is to test whether our slice-wise structure and the shrinking algorithm provide an advantage in occlusion culling, where an object is sent to the graphics

pipeline completely, even if it is only partially visible. *Slice-wise occlusion culling* refers to occlusion culling where the granularity is individual slices, whereas the *building-level occlusion culling* refers to the occlusion culling where the granularity is buildings.

Figure 17 shows the frame rates obtained using the proposed approach and the building-level approach in the first platform. The graphs are smoothed for easy interpretation using a regression function. The average frame rate of the building-level occlusion culling is 30.1 frames per second (fps) for the first test configuration and 11.89 fps for the second one. For our scheme, the tests gave average frame rates of 44.03 fps and 17.02 fps, 46.21 % and 43.15 % faster for the first and second test platforms, respectively. The 3 % smaller increase for the second platform is mostly due to disk swapping.

Figure 18 gives the decrease in the polygon count: the tightness of the slice-wise occlusion culling as compared to the building-level occlusion culling. For the building-based granularity, 51 buildings and 74,689 polygons are drawn on the average for each frame. In the slice-wise approach, 41 of these buildings are accepted as partially visible; this decreases the number of necessary polygons to 48,647 on the average. If the building-level granularity were used for occlusion culling, approximately 54% more polygons would be unnecessarily sent to the graphics pipeline.

We would expect a rendering performance of 46 fps for the slice-wise approach when 48,647 polygons are rendered on average. However, average frame rate was in fact 44 fps. This is because of the checks done to avoid multiple renderings of triangles at the intersection of the horizontal and the vertical slices.

The study shows that our occlusion-culling scheme incurs no noticeable overhead on the visualization algorithm, since the determination of the visible portions of the objects requires a constant time.

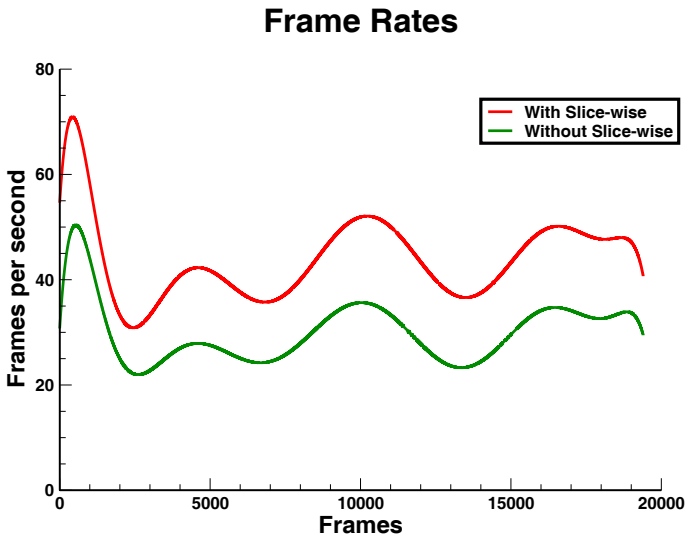


Fig. 17. Frame rate speedups of the proposed approach as compared to the building-level approach.

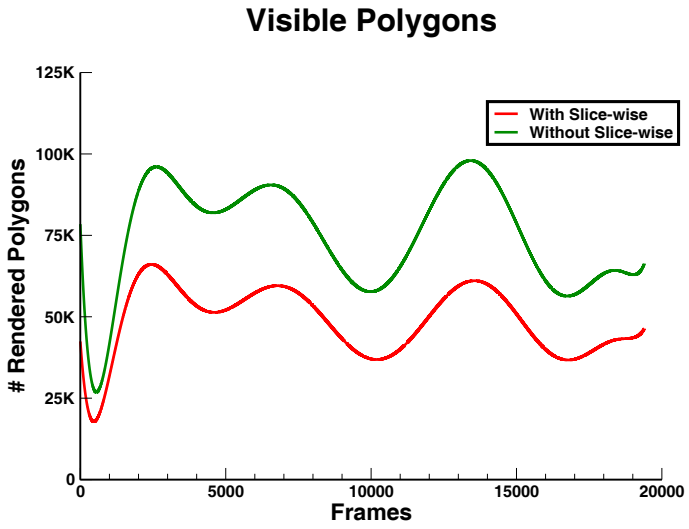


Fig. 18. The number of rendered polygons for slice-wise and building-level occlusion culling.

```
Generate and submit NV_occlusion_queries:
begin
    forall vertical slices do
        slice_increment←− 1;
        while slice_increment*Δ ≤slice_height do
            Query the slice box with height (slice_increment*Δ);
            slice_increment++;
    forall horizontal slices do
        Query the horizontal slice box;
end
Collect the results of the occlusion queries:
begin
    forall vertical slices do
        slice_increment←− 1;
        while slice_increment*Δ ≤slice_height do
            if The query returns any visible pixels then
                slice_occlusion_height ←− (slice_increment-1)*Δ);
                break;
            else
                slice_occlusion_height ←− slice_increment*Δ;
            slice_increment++;
        if slice_occlusion_height≡slice_height then
            Mark the slice as INVISIBLE;
        else
            Mark the slice as PARTIALLY_VISIBLE;
    forall horizontal slices do
        if The query returns any visible pixels then
            Mark the slice as VISIBLE;
        else
            Mark the slice as INVISIBLE;
end
if all slices are INVISIBLE then
    Mark the object as INVISIBLE;
else
    Mark the object as PARTIALLY_VISIBLE;
```

Algorithm 2: Testing the slices: Each slice is tested against the shrunk occluder geometry. The slice bounding boxes are drawn from the bottom to the top incrementing them gradually as in Figure 13(b).

$X\_occlusion \longleftarrow$ Percentage of occlusion for X-axis slices;

$Z\_occlusion \longleftarrow$ Percentage of occlusion for Z-axis slices;

$work\_axis \longleftarrow$ **max** $(X\_occlusion, Z\_occlusion)$;

Construct maximum sized rectangle of occlusion in the $work\_axis$;

forall *Slices of the* $work\_axis$ **do**

    Discard the vertical slices within the horizontal range of the rectangle;

    Discard the horizontal slices within the vertical range of the rectangle;

Discard the slices of the vertical axis other than $work\_axis$;

Algorithm 3: Optimizing the visible slice counts: The algorithm reduces the number of slices used to represent the visible portion for an occludee (see Figure 14).

VI. Conclusion

In this paper, we propose a data structure that exploits the visibility characteristics of buildings in the visualization of urban scenes. The proposed approach avoids sending a building entirely to the graphics pipeline if only a small portion of it is visible, thereby solving the partial occlusion problem. The objects are divided into slices at each axis and then the slices rather than the whole objects are checked for occlusion.

We also show how to shrink objects in a scene, including nonconvex ones, in order to use them as occluders for from-point conservative occlusion culling. Our shrinking algorithm can be used for any kind of object, not just buildings in an urban scene.

Our experiments show the proposed slice-wise occlusion culling provides a 46 % faster visualization than a building-level occlusion culling. Similarly, a visualization using our approach sends 54 % less polygons to the rendering pipeline as compared to a visualization using building-level occlusion culling.

The slice-wise structuring of objects can also be used to visualize scenes other than urban scenery, although we did not test this. Another application for our method would be for the scenes where buildings are touching (as in some European cities). In this case, a subdivision at the object level could be done to create smaller objects.

The proposed approach works for flythrough type navigations where the user is above buildings. It would be helpful for real time rendering to integrate our method with other approaches such as view dependent refinement.

Acknowledgment

We are grateful to Kirsten Ward for proofreading and suggestions. Special thanks to M. Erol Aran for suggesting many improvements. The work described in this paper is supported by the Scientific and Research Council of Turkey (TÜBİTAK) under Project Code 104E029. The first author is also supported by a scholarship from the Scientific and Technical Research Council of Turkey (TÜBİTAK).

REFERENCES

[1] H. Samet, "The quadtree and related data structures," *ACM Computing Surveys*, vol. 16, no. 2, pp. 187–260, 1984.

[2] OpenSG Forum, *OpenSG – Open Source Scene Graph*, http://www.opensg.org, 2000.

[3] D. Staneker, D. Bartz, and W. Straßer, "Occlusion culling in OpenSG PLUS," *Computers & Graphics*, vol. 28, pp. 87–92, 2004.

[4] C. Saona-Vázquez, I. Navazo, and P. Brunet, "The visibility octree: a data structure for 3D navigation," *Computers & Graphics*, vol. 23, no. 5, pp. 635–643, 1999.

[5] W. V. Baxter III, A. Sud, N. K.Govindaraju, and D. Manocha, "Gigawalk: Interactive walkthrough of complex environments," in *Proceedings of 13th Eurographics Workshop on Rendering*, pp. 203–214, 2002.

[6] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412–431, 2003.

[7] J. T. Klosowski and C. T. Silva, "Efficient conservative visibility culling using the prioritized-layered projection algorithm," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 4, pp. 365–379, 2001.

[8] ——, "Rendering on a budget: A framework for time-critical rendering," in *Proceedings of IEEE Visualization'99*, pp. 115–122, 1999.

[9] S. Nirenstein and E. Blake, "Hardware accelerated visibility preprocessing using adaptive sampling," in *Proceedings of the Eurographics Symposium on Rendering*, pp. 207–216, 2004.

[10] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility preprocessing with occluder fusion for urban walkthroughs," in *Proceedings of Rendering Techniques*, pp. 71–82, 2000.

[11] D. Davis, W. Ribarsky, T. Y. Jiang, N. Faust, and S. Ho, "Real-time visualization of scalably large collections of heterogeneous objects," in *Proceedings of IEEE Visualization'99*, pp. 437–440, 1999.

[12] S. Nirenstein, E. Blake, and J. Gain, "Exact from-region visibility culling," in *Proceedings of the 13th Eurographics Workshop on Rendering*, pp. 191–201, 2002.

[13] T. A. Funkhouser, C. H. Sequin, and S. J. Teller, "Management of large amounts of data in interactive building walkthroughs," *ACM Computer Graphics (Proceedings of ACM Symposium on Interactive 3D Graphics)*, vol. 25, no. 2, pp. 11–20, 1992.

[14] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion, "Conservative volumetric visibility with occluder fusion," in *Proceedings of SIGGRAPH'00*, pp. 229–238, 2000.

[15] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative visibility preprocessing using extended projections," in *Proceedings of SIGGRAPH'00*, pp. 239–248, 2000.

[16] J. Heo, J. Kim, and K. Wohn, "Conservative visibility preprocessing for walkthroughs of complex urban scenes," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST-00)*, pp. 115–128, 2000.

[17] F. A. Law and T. S. Tan, "Preprocessing occlusion for real-time selective refinement," in *Proceedings of the ACM Symposium on interactive 3D Graphics*, pp. 47–54, 1999.

[18] V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Virtual occluders: An efficient intermediate pvs representation," in *Proceedings of the Eurographics Workshop on Rendering Techniques*. Springer-Verlag, pp. 59–70, 2000.

[19] C. L. Bajaj, V. Pascucci, and G. Zhuang, "Progressive compression and transmission of arbitrary triangular meshes," in *Proceedings of IEEE Visualization'99*, pp. 307–316, 1999.

[20] J. Popović and H. Hoppe, "Progressive simplicial complexes," in *Proceedings of SIGGRAPH'97*, pp. 217–224, 1997.

[21] J. Rossignac, "Geometric simplification and compression in multiresolution surface modeling," in *SIGGRAPH '97 Course Notes #25*, 1997.

[22] R. Yagel and W. Ray, "Visibility computation of efficient walkthrough of complex environments," *Presence*, vol. 5, no. 1, pp. 1–16, 1996.

[23] M. V. D. Panne and A. J. Stewart, "Efficient compression techniques for precomputed visibility," in *Proceedings of Eurographics Workshop on Rendering*, pp. 306–316, 1999.

[24] B. Aronov, H. Brönnimann, A. Y. Chang, and Y. Chiang, "Cost-driven octree construction schemes: An experimental study," in *Proceedings of 19th Annual ACM Symposium on Computational Geometry*, pp. 227–236, 2003.

[25] C. Gotsman, O. Sudarsky, and J. A. Fayman, "Optimized occlusion culling using five-dimensional subdivision," *Computers & Graphics*, vol. 23, no. 5, pp. 645–654, 1999.

[26] X. Decoret, G. Debunne, and F. Sillion, "Erosion based visibility preprocessing," in *Proceedings of the 14th Eurographics Workshop on Rendering*, P. Christensen and D. Cohen-Or, Eds., pp. 281–288, 2003.

[27] P. K. Agarwal and M. Sharir, "Arrangements," in *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publishers B.V., North-Holland, Amsterdam, pp. 49–119, 1999.

[28] M. Schmitt and J. Mattioli, *Morphologie Mathématique*. Masson, Paris, 1993.