Available online at www.sciencedirect.com
ScienceDirect
Graphical Models 69 (2007) 191–210
Graphical Models
www.elsevier.com/locate/gmod

Conservative occlusion culling for urban visualization using a slice-wise data structure

Türker Yılmaz, Uğur Güdükbay *

Department of Computer Engineering, Bilkent University, Bilkent, 06800 Ankara, Turkey

Received 11 May 2006; received in revised form 11 January 2007; accepted 15 January 2007
Available online 27 February 2007

Communicated by Daniel G. Aliaga

Abstract

In this paper, we propose a framework for urban visualization using a conservative from-region visibility algorithm based on occluder shrinking. The visible geometry in a typical urban walkthrough mainly consists of partially visible buildings. Occlusion-culling algorithms, in which the granularity is buildings, process these partially visible buildings as if they are completely visible. To address the problem of partial visibility, we propose a data structure, called *slice-wise data structure*, that represents buildings in terms of slices parallel to the coordinate axes. We observe that the visible parts of the objects usually have simple shapes. This observation establishes the base for occlusion-culling where the occlusion granularity is individual slices. The proposed slice-wise data structure has minimal storage requirements. We also propose to shrink general 3D occluders in a scene to find volumetric occlusion. Empirical results show that significant increase in frame rates and decrease in the number of processed polygons can be achieved using the proposed slice-wise occlusion-culling as compared to an occlusion-culling method where the granularity is individual buildings.

Keywords: Space subdivision; Octree; Occlusion-culling; Occluder shrinking; Minkowski difference; From-region visibility; Urban visualization; Visibility processing

1. Introduction

The efficiency of a 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 there exist efficient methods for them. 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. 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

* Corresponding author. Fax: +90 312 266 4047.
E-mail addresses: yturker@cs.bilkent.edu.tr (T. Yılmaz), gudukbay@cs.bilkent.edu.tr (U. Güdükbay).

1524-0703/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.gmod.2007.01.002

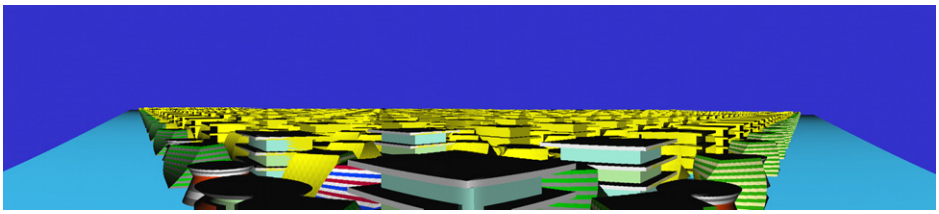

Fig. 1. Slice-wise occlusion culling sends approximately 51% fewer triangles to the graphics pipeline and increases frame rate by 81%, as compared to occlusion culling using building-level granularity for this model. The yellow colored sections show occluded regions, which are discarded from the graphics pipeline. In addition, the slice-wise representation decreases Potentially Visible Set (PVS) storage requirement drastically. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this paper.)

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, which takes advantage of the special topology of buildings within an urban scene. It automatically exploits real-world occlusion characteristics in urban scenes by subdividing the objects into slices parallel to the coordinate axes (Fig. 1). 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 requirement of Potentially Visible Sets (PVSs) limits the scalability of their usage. The PVS storage requirement of the proposed slice-wise structure is very low (3 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 an occluder-shrinking algorithm to achieve conservative from-region visibility. Conservative occlusion-culling can be performed by shrinking the occluders and performing the visibility tests using the shrunk versions of the occluders. To our knowledge, this is the first demonstrated attempt that can also be applied to general nonconvex occluders as a whole.

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 the occlusion-culling method based on conservative from-region visibility and our shrinking algorithm. In Section 5, we give experimental results and comparisons. Finally, we give conclusions.

2. Related work

Scene representation has a crucial impact on the performance of a 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 [3,4]. However, these are useful mainly for the definition of object hierarchies. Their usage 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 [5], the triangles that belong to many nodes of the octree are subdivided across the nodes for easy traversal. In [6], the objects could be divided into subobjects to create a balanced scene hierarchy, if necessary.

Occlusion-culling algorithms detect the parts of the scene occluded by other objects and do not contribute to the overall image; these parts should not be sent to the graphics pipeline. Since most of the geometry is hidden behind occluders for urban environments, it is extremely helpful to detect these occluded portions.

Occlusion-culling algorithms can be classified as *from-point* and *from-region*. From-point algorithms calculate visibility with respect to the position and viewing direction of the user, whereas from-region algorithms calculate visibility which is valid for a certain area or volume. One of the most advantageous property of the from-region algorithms is that the visibility can be precomputed and stored for later use. However, it has the disadvantage of large storage requirements, which we intend to overcome by developing the slice-wise data structure.

The occlusion-culling algorithms can also be classified as either *conservative* or *approximate* [7]. 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. These algorithms may even accept invisible buildings as visible. Approximate occlusion-culling algorithms, such as [8–10], render the visible primitives 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 methods, such as [6,11,12].

Another class of algorithms is the *exact* visibility algorithms, which provide accurate visibility lists at the expense of degrading the rendering performance and increasing storage requirements. An example of this class is [13], where the authors represent triangles and the stabbing lines in a 5D Euclidean space derived from a Plücker space and perform geometric subtractions of the occluded lines from the set of potential stabbing lines. In [14], the authors compute visibility from a region by using a hierarchical line space partitioning algorithm. They map the oriented 2D lines to points in dual line-space and test the visibility of a line segment with respect to the occluders yielding to a visibility from a region.

There are occlusion-culling algorithms developed for specific environments, such as indoor scenes [15], outdoor environments like urban walkthroughs [11,16–18], and general environments—environments having no natural object definition [13,19–21]. In all of the algorithms the navigable area is clustered in a way to provide the fastest occlusion-culling possible. For indoor scenes, the navigation area is naturally clustered into rooms and specific techniques were developed such as portal usage [15]. For the case of urban walkthroughs, the navigable area is clustered or *cellulized* so that precomputations can be performed with respect to a limited area. Most of the algorithms developed for general environments are also applicable to others with little or no modifications such as [21], but the best performance is achieved by using the algorithms in their target environments.

Some applications are only suitable for the environments where there are large occluders and a large portion of the model is behind these occluders. These algorithms strongly rely on temporal coherence. The traversal cost and other overheads increase as the occluded regions decrease, thereby limiting the scalability [5,22,23]. Visibility determination by traversing a scene hierarchy requires the quick selection of occluders or the occluders should be selected beforehand to decrease the time required for this process. Performing occluder selection is a difficult task [24–27] because it must be completed in a limited time and there are many factors affecting the occluder selection process, such as the projected area of the occluder, triangle counts, transparency factors and holes. A survey of occlusion-culling can be found in [7].

The purpose of creating visibility lists for each view-cell is to improve scalability. Time consuming operations are done beforehand. This results in a large amount of data to be stored. There are many different approaches to compressing the resultant data, such as [21,28–31]. Our slice-wise data structure significantly decreases the amount of information that needs to be stored.

The proposed slice-wise structure is able to create a tight visibility set of slices of objects for any kind of occlusion-culling algorithm. The visibility set thus produced is tighter than those that measure occlusion at the building level, but more conservative than the exact ones that operate at the polygon level: it groups polygons by exploiting visibility characteristics in a typical urban walkthrough.

Our urban visualization framework can be compared with the previous state of the art work as follows:

- It does not make any assumption on the architectures of the buildings. Unlike [17,23,32,33], our occlusion-culling algorithm handles all kind of 3D occluders as in [10,13,19,34], not just 2.5D buildings generated by extruding the city plans.
- The occlusion-culling algorithm is based on occluder shrinking performed in the object-space. It is a from-region method, as in [16,21,33]; however, our algorithm is capable of shrinking all kinds of 3D objects by calculating the Minkowski difference of the occluders and the view-cell. We can shrink the nonconvex 3D occluders as a whole.
- All of the previous approaches use some kind of data structure to speed up scene and object traversal. We also use quadtree-based scheme for culling large portions of the scene. However, we make use of our proposed slice-wise structure to determine visible parts of each building to gain more rendering time by eliminating those invisible portions. Instead of traversing and storing a large amount of data for the representation of visible portions, we store only 3 bytes for each building and access them in constant time.
- Unlike [8,9,35], which are *approximate* occlusion-culling algorithms and [13,14], which are *exact* occlusion-culling algorithms, our algorithm is *conservative*, like [10,17,19,22,23,26,32–34] and handles all types of occluder fusion.

- We use hardware occlusion queries to determine occlusions, as in [10,21,22]. We calculate the visibility with respect to the centers of the calculated view-cells [36]. Since we use occluder shrinking, the potentially visible set (PVS) calculated for the center of the view-cell is valid throughout the whole view-cell.

3. Slice-wise structuring of objects

3.1. 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 Fig. 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 Fig. 2a.
- The visible part looks like a *vertical rectangular* block, from the left or right of the building if the occluder seems taller than the occludee (see Fig. 2b).
- If the occluder is a large one and appears to be shorter than the occludee, it usually hides the lower half of the building. In this case the visible portion looks like a *horizontal rectangular* block, as in Fig. 2c.

Most of the occlusion can be represented by the proposed slice-wise structure using these characteristics. However, there are of course some other cases that the occlusion cannot be perfectly represented. These may include a configuration such as both sides of the building are occluded resulting in a middle part visibility and the top of the building is visible in addition to the middle part visibility. In any of these cases the occlusion-culling algorithm tries to capture the occlusion as much as possible in one of the three visibility forms. For example, the middle part visibility is regarded as *vertical rectangular*, the top and middle part visibility is regarded as *L-shaped* visibility.

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 the storage requirements for the PVSs.

3.2. Slicing objects

The aim of the proposed slice-wise structure is to create tight PVSs for urban scenes. Slice-wise representation is obtained by subdividing an object into axis-aligned slices and determining the triangles that belong each slice. The slicing process is composed of two steps: *subdivision* and *slice creation*. In the subdivision step, each object is uniformly subdivided and the grid cells occupied by each triangle are determined. Next, the occupied cells in the uniform subdivision are combined into axis-aligned slices for each coordinate axis. The process of slicing an object is shown in Fig. 3. The resultant data structure is shown in Fig. 4.

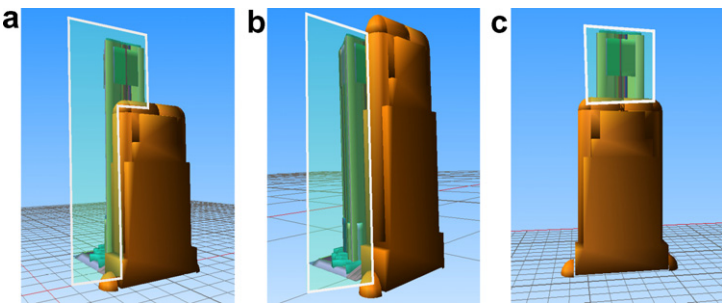

Fig. 2. Visibility forms during urban navigation: (a) *L-shaped* form; (b) *vertical rectangular* form; (c) *horizontal rectangular* form. In each part of the figure, the visible part of the occludee is the green transparent area. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this paper.)

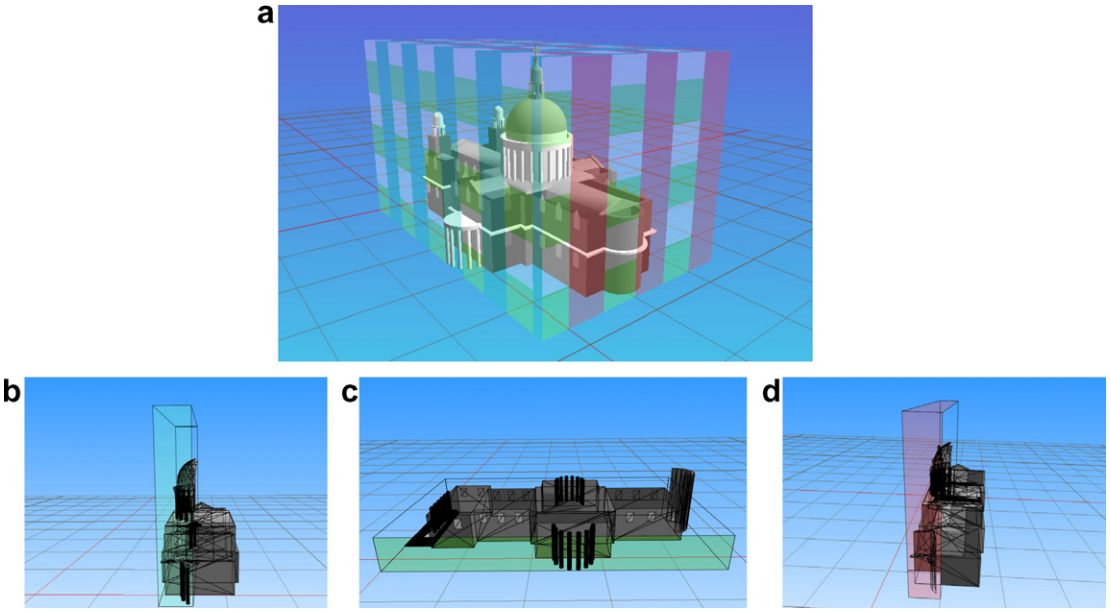

Fig. 3. The process of slicing an object determines the triangles that belong to each slice. (a) A complete view of the object where the positions of slices are shown; (b) an x-axis slice; (c) a y-axis slice; (d) a z-axis slice.

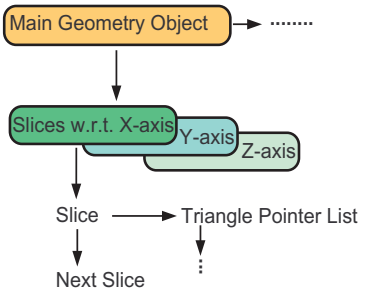

Fig. 4. The scene data structure produced by slicing operations.

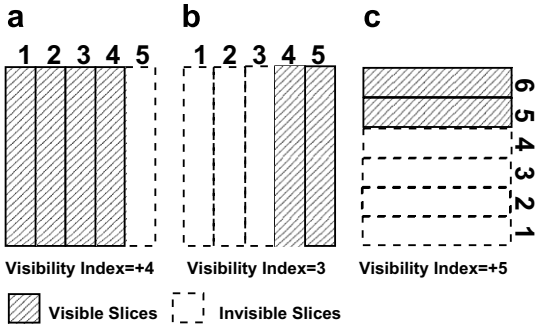

Fig. 5. Defining visibility indices for objects: the visible slice indices are determined for each axis during occlusion determination. (a) If an object is partially occluded from the right, the index of the last visible slice is stored with a "+" sign. (b) If the object is partially occluded from the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

3.3. Visibility representation using slices

Defining the visible portions requires determining the visible slices. As shown in Fig. 5, we only need to store one visible-slice index for each axis. The combination of these indices facilitates the representation of the visibility characteristics (see Fig. 2).

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

- Each triangle is encoded in at least three slices in different axes. Therefore, we can use slices on any axis during visualization. We choose the axis with maximum occlusion (see Fig. 19). Choosing the maximally occluded axis allows us to tighten the visible set as much as possible.
- The memory required for the slice-wise approach is minimal. In order to define the visibility, 3 bytes, one for each axis, are used for each object and for each view-cell. This representation greatly decreases the storage requirements for PVSs (see Fig. 7).
- Slicing the objects provides a fast way to access visible portions of an object. Unnecessarily traversing a tree-like data structure is prevented by directly accessing the visible slices and hence triangles of an object.

3.4. Comparison with other storage schemes

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 [21,31,37], the problem of big PVS storage problem has not been given enough importance [7].

We compare the proposed structure with octrees and regular grids in terms of the memory requirements. In Fig. 6, we depict subdivision depth and the number of nodes needed for each subdivision. The number of nodes for octrees refers to regularly subdivided octree including the bits needed for the previous levels. In an adaptively subdivided octree, the number of nodes is below these levels. However, giving exact costs and approximations on adaptive version is very difficult. Instead, we give an informal comparison of the results of the empirical study with octrees and triangle level PVS sizes in Section 5. A comprehensive study of the costs for various construction schemes of octrees is presented in [38].

PVS storage costs are depicted for various storage schemes in Fig. 7. In this comparison, we assume that all objects are *visible* or *partially visible*. We also assume that each node of octree and regular grids can be identified with 1 bit and we discard additional information to be stored along them, such as corner coordinates. The pointers to polygons are not taken into account because they are needed in all types of structures. Additionally, we provide the data needed for occlusion culling at the polygon level, assuming that the visibility of each triangle is encoded in bits. The figure 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.

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.

It is also possible to define some semantic properties and store occlusion information with respect to this information. For example, in [23], the authors define floors for the buildings, called as $2.5\mathrm{D}+\epsilon$; these buildings have more vertical complexity than 2.5D buildings. Their occlusion-culling algorithm tests triangles and determine the visibility on a floor basis. However, this prevents its application to real city models obtained from sources, such as airborne laser scanners. Our algorithm is capable of handling all types of buildings, do not need floor information and determines occlusion with respect to three axes. In this way, it captures a tighter PVS than the floor-based visibility calculation. Since it requires more processing time, it is applied as a preprocessing step.

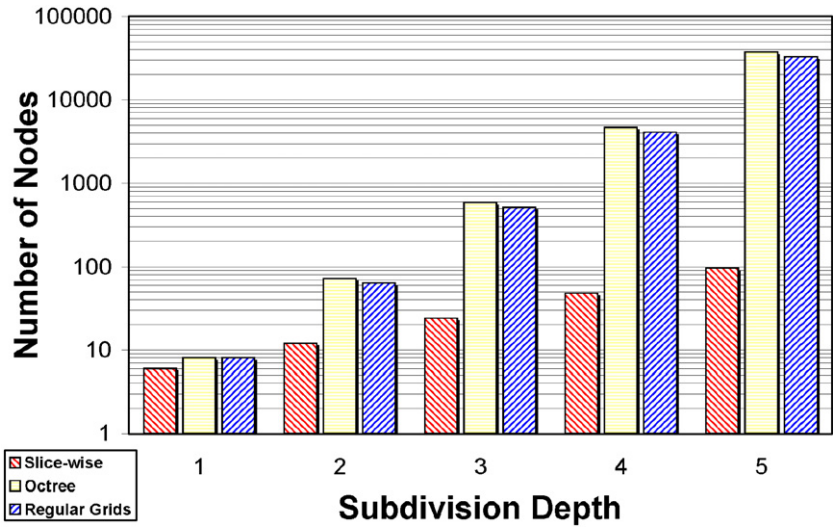

Fig. 6. The comparison of the number of nodes needed for each subdivision scheme. The graph shows the number of nodes needed. The values are in logarithmic scale.

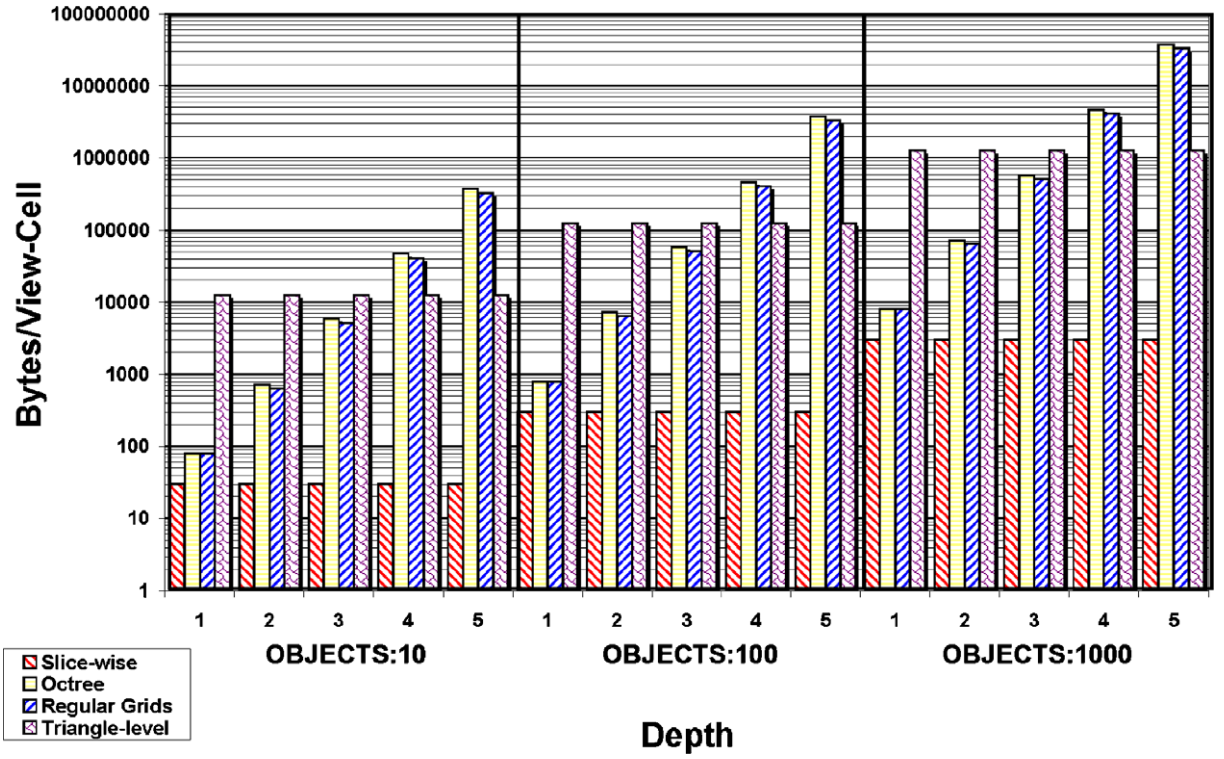

Fig. 7. The comparison of the PVS storage requirements for each subdivision scheme. The graph shows the amount of memory required for a view-cell in bytes. We assume that each object is composed of 10K polygons. The slice-wise data structure provides a huge decrease in the use of memory. The values are in logarithmic scale.

4. Slice-based from-region visibility

Fig. 8 shows the framework for urban visualization using the slice-wise representation. It mainly consists of a preprocessing phase and a navigation phase. To test the effectiveness of the slice-wise representation, we developed a conservative from-region visibility algorithm. To achieve conservative occlusion culling, we made use of the shrinking idea first proposed by Wonka et al. [11]. Our shrinking algorithm can be applied to any kind of scene object, including nonconvex ones.

4.1. Occluder shrinking

The purpose of shrinking is to achieve conservative occlusion culling by sampling from discrete locations. 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. To achieve conservativeness, it is necessary to shrink occluders so that the objects behind the occluder is visible even if the user moves to the farthest possible location in the view-cell.

Wonka et al. [11] shrink occluders by using a sphere constructed around 2.5D occluders. Decoret et al. [16] generalize the shrinking by a sphere to erosion by a convex shape, which is the union of the "edge convex hulls" of the object. This is performed to create tighter visibility sets and to increase the occlusion region of the objects. They compute the shrunk versions of objects using an image-based algorithm at each view cell using a voxelized representation. This makes it very difficult to apply the presented image-based approach to general 3D objects.

4.1.1. Shrinking general 3D objects

The exact shrinking can only be performed by calculating the Minkowski differences of the object and the view-cell [39,40] and using the volume constructed inside the object as the shrunk version (see Fig. 9).

Consider a general 3D object O and a 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:

$$O \oplus X = \{M + x | M \in O, x \in X\} \quad (1)$$

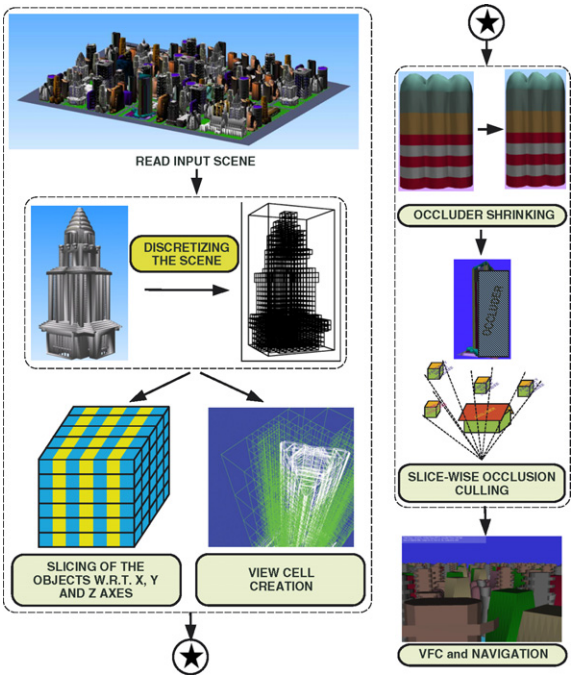

Fig. 8. The urban visualization framework: in the first phase, we read the scene and calculate the bounding boxes of objects. Next, we apply uniform subdivision to each object. Then, we cluster the cells of the uniform subdivision into slices. After creating the shrunk versions of the objects, these slices are checked for occlusion and a tight visible set is determined 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.

Here, X is commonly called the *structuring element* [16]. Thus, the inner volume, which composes the shrunk shape S of the object, can be defined as:

$$\begin{aligned} O \ominus X &= \{S | \forall x \in X, S + x \in O\} \\ &= \{S \mid \{S\} \oplus X \subset O\} \\ &= \{S\} \subseteq \{O \ominus X\} \end{aligned} \tag{2}$$

Although there exist methods for exact Minkowski sums [41], it is practically very hard to find the exact Minkowski differences of general 3D objects. Our aim is to find a shrunk version of an object and retain the conservativeness of the view-cell visibility. Thus, we try to find an approximation to the difference, which will also satisfy the conservativeness of the occlusion-culling process. In this way, we can shrink any complex 3D object.

4.1.2. Shrinking using the Minkowski difference

We shrink an object by moving the vertices in the reverse direction of their normals. Although architecturally we do not make any assumption on the buildings, connected and closed meshes shrink better. Our aim is to use an approximate Minkowski difference of the object and the view-cell and still achieve conservative occlusion culling. It should be noted that the vertices are not moved with a constant distance (see Fig. 10).

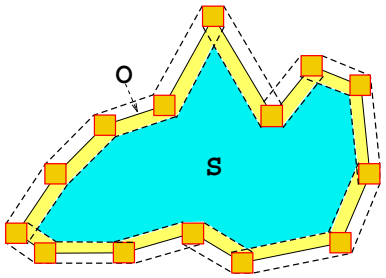

Fig. 9. Shrunk volume extraction using Minkowski difference calculation. O denotes the object and S denotes the shrunk volume of the object.

The traveled distance of a vertex during shrinking affects the final position of a face. For the exact Minkowski difference calculation, the movement distances of the faces towards inside vary with respect to the orientation of the view-cell, as shown in Fig. 11a. As an approximation that guarantees conservativeness, the faces should be moved at least the distance ε, which is the longest movement distance within a view-cell (see Fig. 11b).

Theorem 1. *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 Fig. 12).*

Proof. According to Eq. (2), 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 difference, that is $d = a$ or $d = b$ (see Fig. 11a). Hence, $\{S_d\} \subseteq \{O \ominus X_d\}$. Since ε is the maximum distance to be moved, then $\{a,b\} \leqslant \varepsilon$. 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. □

4.1.3. Calculating shrinking distance for the vertices

Using the notations of Figs. 11b and 13,

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

$$i = 1, 2, \ldots, n$$

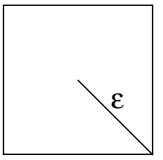

Fig. 10. A sample view-cell, in which the user can move at most with ε distance.

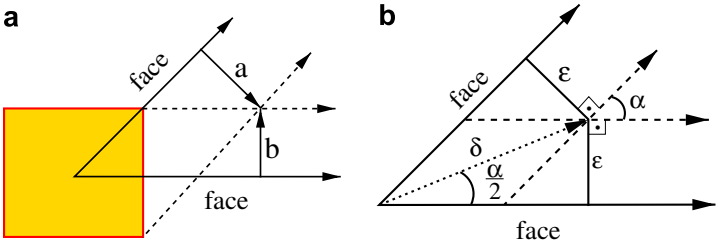

Fig. 11. Shrinking using the Minkowski difference: (a) in an exact Minkowski difference calculation, the movements of the faces towards inside are different with respect to the view-cell position. The two faces move with distances a and b. (b) The face movements are assumed to be at least the distance ε of Fig. 10 to guarantee conservativeness. In this case, the vertex movement distance becomes δ. For easy interpretation, only an instance of the process where two faces sharing a vertex is shown.

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 yields to the longest distance and guarantees conservativeness. The calculated shrinking distance becomes a conservative bound on the real Minkowski differences of the model and the view-cell.

4.1.4. Shrinking occluders

The calculation of a correct shrinking distance is not enough to create conservative shrunk versions

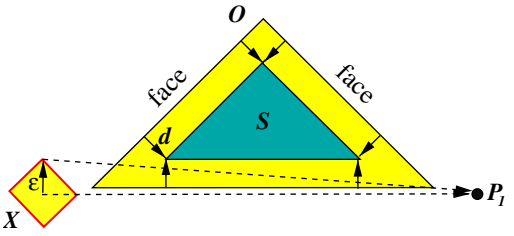

Fig. 12. If a point P_1 is visible from the center of the view-cell, then it should also be visible with respect to its shrunk version S, even if the user moves to the farthest distance available in the view-cell, ε. If the inner movement distance d of the faces for the shrunk shape calculation is greater than or equal to ε, the conservativeness is guaranteed and the point P_1 becomes visible with respect to the shrunk version S. The reader is referred to [11] 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.

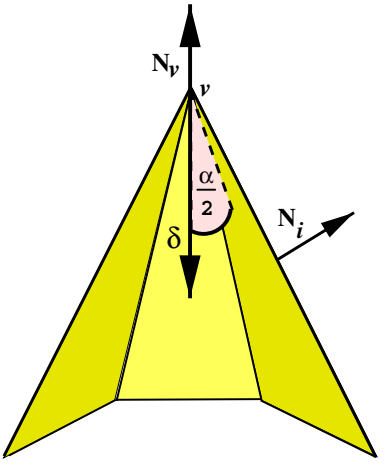

Fig. 13. The object is shrunk by moving the vertex in the opposite direction of the vertex normal. The shrinking distance δ is calculated based on the view cell parameter ε and the angle $\frac{\alpha}{2}$.

of the occluders; some faces may go inside one another and invalidate conservativeness (see Fig. 14). To prevent such cases, we check the intersection of the volumes formed by the movement of the faces in the occluder and the corresponding faces in the shrunk version and remove the intersected ones. In order to accomplish this;

- 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 (see Fig. 15).

We also remove other bad cases such as triangles with no area and overlapping triangles. Shrinking examples are given in Figs. 15 and 16.

Our shrinking algorithm has some similarities with simplification envelopes [42]. Simplification envelopes are used to create simplified versions of 3D models. The simplified models are obtained by moving the vertices at most δ distance from their original position. When many triangles come close to each other, they are removed and smaller number of triangles are inserted. In simplification envelopes, it is guaranteed that the movement distance of the vertices are *at most* δ, whereas in our shrinking algorithm it is guaranteed to be *at least* δ. In simplification envelopes, the vertices are moved with small steps and the triangles are checked for intersection at each step. In our case, since the movement distance has to be at least δ, we calculate the necessary distance once and check for intersections later. Since our aim is to shrink the objects, we do not create new triangle patches as opposed to simplification envelopes since the triangle patch creation may invalidate conservativeness.

The shrinking approach used in [21] is also applicable to general 3D scenes. They accept triangles or groups of connected triangles as occluders. They use the supporting planes of the triangles or a combination of supporting planes to construct an occluder umbra with respect to a selected projection point. The shrinking is performed in this umbra by calculating the inner offset of the supporting planes towards the center of the occluder umbra. Since they use planes of the triangles, the view-cell size changes with respect to the geometry. This approach facilitates the load balancing of the geometry for each view-cell. However, in large environments, the view-cell partitioning may go deeper and may result in a large number of view-cells. As reported by the authors, they may have up to 500K view-cells ranging from several inches to a few feet wide for which the occluder shrinking should be performed. In our approach, we use the objects as a whole and calculate their shrunk versions once.

As a result of occluder shrinking, some triangles may be removed from the model if there is any intersection. Conservativeness of the calculated visibility may be invalidated if we let these intersected triangles in the occluder model. This triangle removal may be overly conservative especially for the models with object sizes less than the shrinking distance. A minimally conservative solution requires the calculation of exact Minkowski differences. Currently, conservativeness degree is lowered by adjusting the view-cell size with respect to the average size of

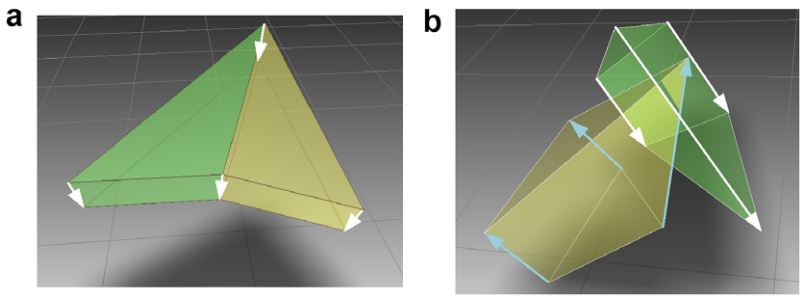

Fig. 14. (a) Shrinking without any intersection: the neighboring triangles do not intersect because of common vertices. (b) The movement volumes of the two triangles intersect and this case results in removal of the triangles.

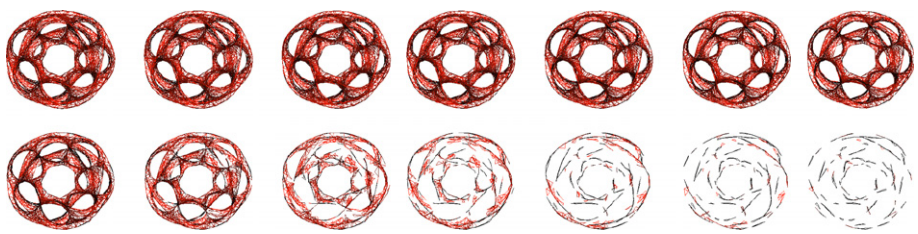

Fig. 15. Shrinking applied to a heptoroid: the rims around holes shrink to null when the triangles from opposite sides start to intersect with each other (δ values are increasing in row-wise order).

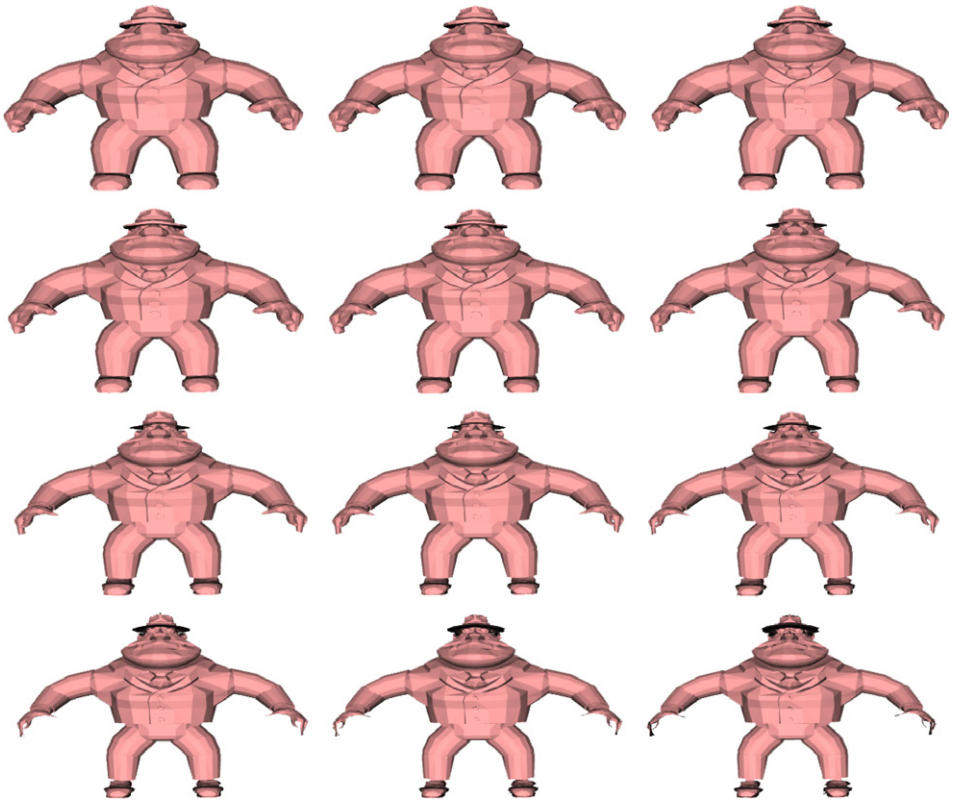

Fig. 16. Shrinking applied to a general 3D object: the vertices touching the ground are not moved vertically (δ values are increasing in row-wise order).

the buildings. The view-cells for different city models are shown in Fig. 17.

4.2. 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. At each step, slices of the horizontal axis are checked for complete occlusion. The other two-axis slices are checked for partial occlusion. An object is tested against a combination of the shrunk versions of all other objects; this creates occluder fusion and determines the occlusion amounts. This process is repeated for each navigable view-cell. The culling scheme is applied from a coarse-grained to fine-grained occlusion-culling tests (see Algorithm 1).

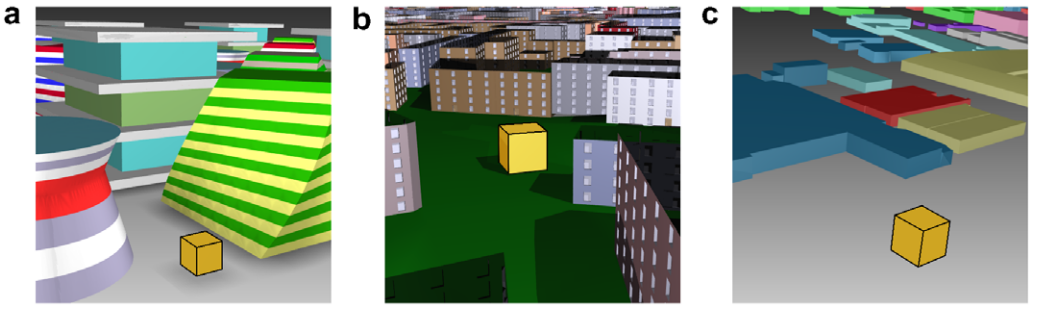

Fig. 17. View-cells (cubes) for different city models: (a) the view-cell for the procedurally generated model. (b) The view-cell in the Vienna2000 model. (c) The view-cell in the Glasgow Model.

```
foreach grid location in 3D do
    Draw the shrunk versions of the visible objects in the frustum as occluder;
    Cull portions of the scene using a quadtree;
    foreach candidate occludee belonging to visible quadtree blocks do
        Construct frustum towards the center of the occludee;
        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.

4.2.1. Coarse-grained culling

The visible buildings are classified as either partially visible or completely visible. Determining these two forms require more occlusion queries to be performed. Algorithm 1 performs a coarse-grained occlusion culling to eliminate large invisible portions of the scene. It performs the following operations:

- Draw shrunk versions of all objects as occluders and disable color and depth buffer updates.
- We perform projection towards all 90° sections of the viewpoint and send quadtree blocks constructed from the ground locations of the buildings for being culled. We use hardware occlusion queries for this purpose. This step eliminates most of the invisible buildings quickly without testing them one by one.
- We calculate the projection of the ocludees belonging to visible quadtree blocks. In practice, although the calculated shrink versions are conservative, due to the use of graphics hardware to detect occlusion and hence the rasterization errors can be encountered, the conservativeness can be violated. For example, a far away object may project to less than a pixel but an occluder right in front of the object may still cover the entire pixel due to the rasterization errors. These errors may be caused by projection, image sampling, and depth-buffer precision errors [43]. We overcome these problems by adjusting the viewing parameters for the projection so that the occludee is zoomed to the maximum extent on the occlusion test screen, which is 1024×768 pixels. This results in a very large view of the redrawn object. In other words, the outer contour of an occluder in the current view becomes tested with a very high precision. Besides, the bounding box of the candidate occludee is tested while generating two fragments for each pixel as in [43] and using antialiasing. In this way, rasterization errors that can be faced due to hardware occlusion queries are prevented.
- We test the bounding box of the candidate occludee. If the bounding box test returns visible pixels, we mark the object as *visible*, otherwise as *invisible*.

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 determine a tighter visibility set for the object. If the bounding box of the occludee is visible, we submit occlusion queries for the slices; we then determine the maximum occlusion height for each slice of the occludee using Algorithm 2. Thus, we classify the buildings as *completely visible*, *partially visible* and *invisible*. The visibility information for the slices is sent to Algorithm 3 to decrease the number of slices and determine the visibility indices to be used during navigation.

4.2.2. Fine-grained culling

Algorithm 2 performs fine-grained occlusion-culling. It checks the slices of a candidate occludee. To find the exact occlusion, we first submit occlusion queries and test the vertical slices with blocks of size Δ (see Fig. 18). Next, we collect the query results. Finding the last invisible Δ allows us to determine the occluded height of the slice. Horizontal slices are checked for complete occlusion.

4.2.3. Optimizing the visible slice counts

An object occluded by several occluders may have an irregular appearance that cannot be easily represented (see Fig. 19). However, our aim is to decrease the amount of information needed to represent visibility, and therefore reduce the time to access the visible parts of the objects. In particular, the purpose of this optimization is to represent the visible area by using a small number of slices and determine a single index for each axis. We have to sacrifice tightness of visibility somewhat to reduce the access time and memory requirement (see Fig. 19).

```
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
            if slice_occlusion_height≡0 then
                Mark the slice as COMPLETELY_VISIBLE;
            else
                Mark the slice as PARTIALLY_VISIBLE;
    forall horizontal slices do
        if The query returns any visible pixels then
            Mark the slice as COMPLETELY_VISIBLE;
        else
            Mark the slice as INVISIBLE;
end
if all slices are COMPLETELY_VISIBLE then
    Mark the object as COMPLETELY_VISIBLE;
else
    Mark the object as PARTIALLY_VISIBLE;
```

Algorithm 2. Testing the slices for occlusion: each slice is tested against the shrunk occluder. The vertical slice bounding boxes are drawn from the bottom to the top incrementing them gradually as in Fig. 18b.

The algorithm for optimizing the slice counts is given in Algorithm 3. This algorithm is used to decrease the number of slices representing 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 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 horizontal 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.
- We discard the slices of the minimally occluded axis.
- Finally, the indices are calculated for each axis (see Fig. 5).

```
X_occlusion ⟵ Percentage of occlusion for X-axis slices;
Z_occlusion ⟵ Percentage of occlusion for Z-axis slices;
work_axis ⟵ 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;
Assign the visibility indices to each axis for being stored;
```

Algorithm 3. Optimizing the visible slice counts: the algorithm reduces the number of slices used to represent the visible portion for an occludee (see Fig. 19).

The created PVS size is a function of the number of view-cells and the number of the objects. The size of the PVS does not change with respect to the model complexity (see Fig. 7).

4.3. Rendering

Vertex arrays [21] and vertex buffer objects [22] are two popular techniques used for rendering the complex environments. We perform rendering by creating OpenGL display lists for the view-cells on-the-fly. OpenGL display lists are more flexible for run time purposes than using vertex buffer objects. We also create display lists for the neighboring view-cells of the user location to prevent performance degradation. This degradation might occur because of the display list compilation of the new PVS for the view-cell the user moves into. The neighboring view-cells are updated on-the-fly gradually without decreasing the frame rate below 25 fps. It should be noted that this operation is also highly parallelizable. The rendering process is shown in Fig. 20. We avoid multiple renderings of the triangles

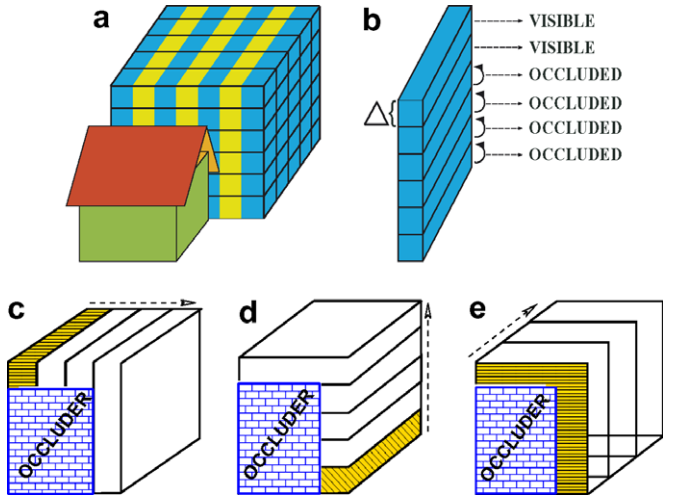

Fig. 18. In order to determine the correct occlusion height in (a), the slices are tested beginning from the lowest unoccluded height and the point of occlusion is found as in (b) by iteratively eliminating blocks of size Δ from the vertical slice; this creates occluder fusion (the viewpoint is in front of the occluder). 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.

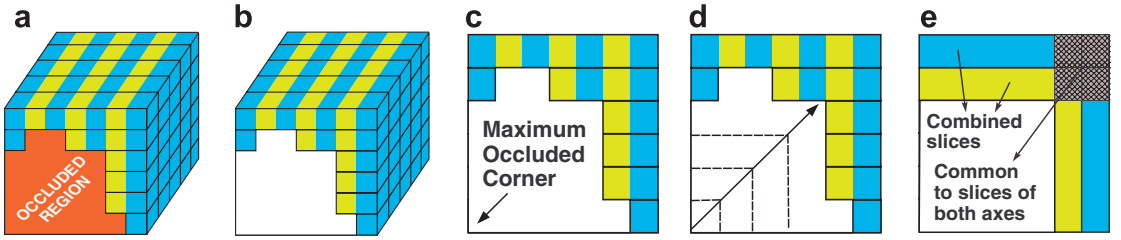

Fig. 19. The resultant shape of the occlusion may have a jaggy appearance and we need to smooth it to represent with the slice-wise structure (a and b). This is handled as described in Algorithm 3. After selecting the maximally occluded axis, the starting corner of the occlusion is determined (c). The 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).

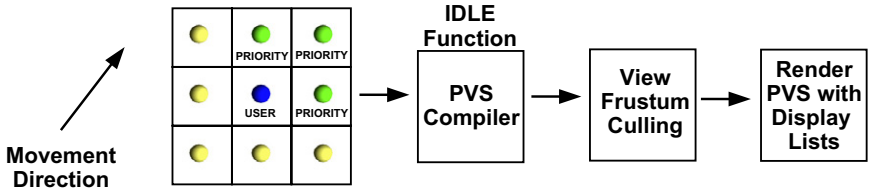

Fig. 20. The rendering process: the user is in the blue view-cell. The display list for the view-cell is compiled before entering to the navigation stage, along with the neighboring view-cells. During navigation, the compilations of the display lists for the neighboring view-cells in the movement direction are given high priority. The display list compiler is attached to the idle function of the OpenGL so that the neighboring view-cell compilation does not cause bottlenecks. In this way, frame dips caused by the compilation of the display lists are prevented. After view frustum culling is performed on the quadtree blocks of the ground locations of the buildings, the constructed display lists for each building in the view frustum are rendered. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this paper.)

at the intersections of the horizontal and vertical axes of the partially visible objects (see Fig. 19e). The tests were performed in 1024×768 screen resolution. The navigation algorithm is supported by a view-frustum-culling algorithm, which eliminates the objects that are completely out of the view frustum.

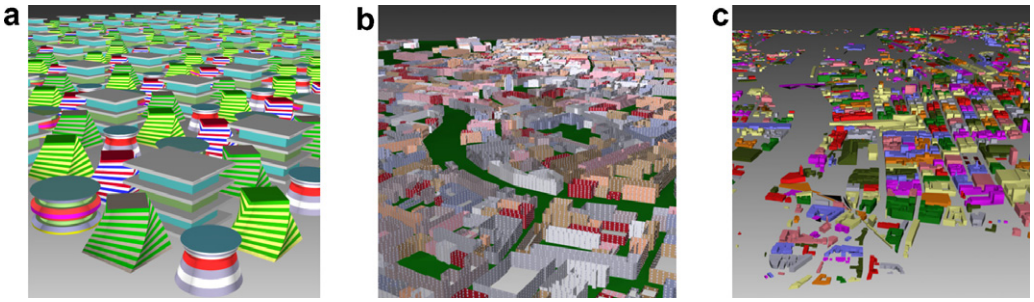

Fig. 21. The models used in tests: (a) the procedurally generated 40M-polygon test model is composed of 6144 buildings ranging from 5K to 8K polygons each; (b) the Vienna model is a 7.8M-polygon model that has 2078 blocks of buildings ranging from 60 to 30,768 faces. In this model, contrary to previous works, each surrounding block of buildings is accepted as a single object during the tests. (c) Glasgow model has originally 290K polygons. However, the mesh structure is not well-defined and has intersecting, long and thin triangles. Therefore, the mesh structure has been refined and a total of 500K polygons are used during the tests.

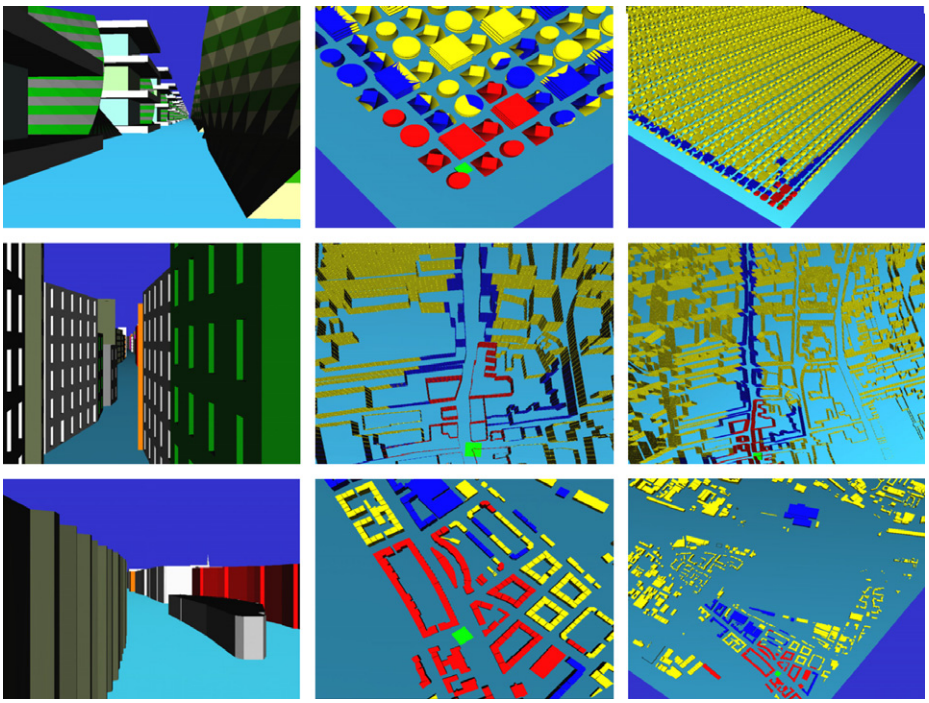

Fig. 22. Still frames from the navigations through the scenes used in the experiments. On the left column, still frames from the current viewpoint are shown. In the middle, the views above the view-points are shown. The view-cell is the green box. On the right, larger areas showing the results of the slice-wise occlusion culling are shown. Partially visible buildings are in blue, completely visible buildings are in red, and invisible buildings are in yellow color. The rows belong to procedurally generated, Vienna, and Glasgow models, respectively.

5. Results and discussion

5.1. Test environment

The proposed algorithms were implemented using *C language* with OpenGL libraries; they were tested on 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. We use three different urban models for the tests (Fig. 21). The first one is a procedurally generated model using a few detailed building models, which is composed of 40M triangles. The second one is the model of the city of Vienna composed of 7.8M triangles (Vienna2000 Model with detailed buildings). The last one is the Glasgow model, which is a relatively small (500K) model. Table 1 shows various statistics about these models. The results of the tests are

Table 1
Statistics of the models used in the tests

Model	Procedurally generated	Vienna 2000	Glasgow
No. polygons	40M	7.8M	500K
No. buildings	6144	2086	1461
Model size	$100K \times 63K$	2385×2900	4246×3520
View-cell size	200×200	10×10	15×15
No. navigable cells	45.5K	72K	66K

Table 2
Summary of the test results using the slice-wise structure

Model	Procedurally generated	Vienna 2000	Glasgow
Total PVS size on disk (MB)	52	18	65
No. slices	377,920	30,392	11,948
No. triangle pointers	136.2M	27.3M	1.6M
Slice-wise memory usage (MB)	1094	218.7	12.4
PVS calculation time/cell (ms)	323	292	436
Shrinking time/building (s)	30.0	13.8	3.7
Total PVS calculation (h)	4.08	5.6	8.0
Total shrinking time (h)	0.05[a]	8	1.5

[a] Since the procedurally generated model contains six different types of buildings repetitively, total shrinking time is low.

summarized in Table 2. We discuss the results for the largest one, 40M-polygon procedurally generated model, for space considerations. The interpretation of the test results for two real city models are similar.

For the procedurally generated model, the navigation area is divided into 200-pixel grids. The area of the city is $100K \times 63K$ pixels. There are about 45.5K navigable grid points in the scene, from where the visibility culling is done. The test city model used in the experiments consists of 6144 complex buildings with six different architectures, each having from 5K to 8K polygons with a total of 40M polygons. The slices are 200 pixels wide, the same width as the grid cells, although they can be different to adapt to the dimensions of the buildings. On the average, there are 15 slices on the x and z axes. The number of slices of the y axis depends on the heights of the buildings, which in our case is around 25 slices. As a result each object has about 55 slices. Preprocessing takes approximately 323 ms for each view-cell. We perform a navigation containing 12,835 frames (Fig. 22). The navigation is performed on the ground to make the occlusion results comparable with other works. It should be noted that flythrough-type navigation is also possible without any modification.

The first aim of the empirical study is to test whether our slice-wise structure and the shrinking algorithm provide an advantage in occlusion culling over one where an object is sent to the graphics pipeline completely even if it is only partially visible. This is performed to test if there are any overheads that will prevent its usage for fine-grained-visibility testing. *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. The second aim is to compare the PVS storage requirements of an occlusion culling approach using a slice-wise data structure and other subdivision schemes, such as octree and triangle level occlusion culling.

5.2. Rendering performance

Fig. 23 shows the frame rates obtained using the slice-wise and building-level occlusion culling. The graphs are smoothed for easy interpretation using a regression function. The average frame rate of the building-level occlusion culling is 61.36 frames per second (fps). We achieve average frame rates of 111.06 fps, 81% faster than using building-level granularity. In our tests, 99.26% of the geometry is culled on the average. The culling percentages strongly depends on the size of the view-cell used. As the size of the view-cell increases, the number of preprocessed occlusion-culling operations decreases, whereas the number of the triangles unnecessarily accepted as visible increases. As examples of geometry culling performance: in [26] from

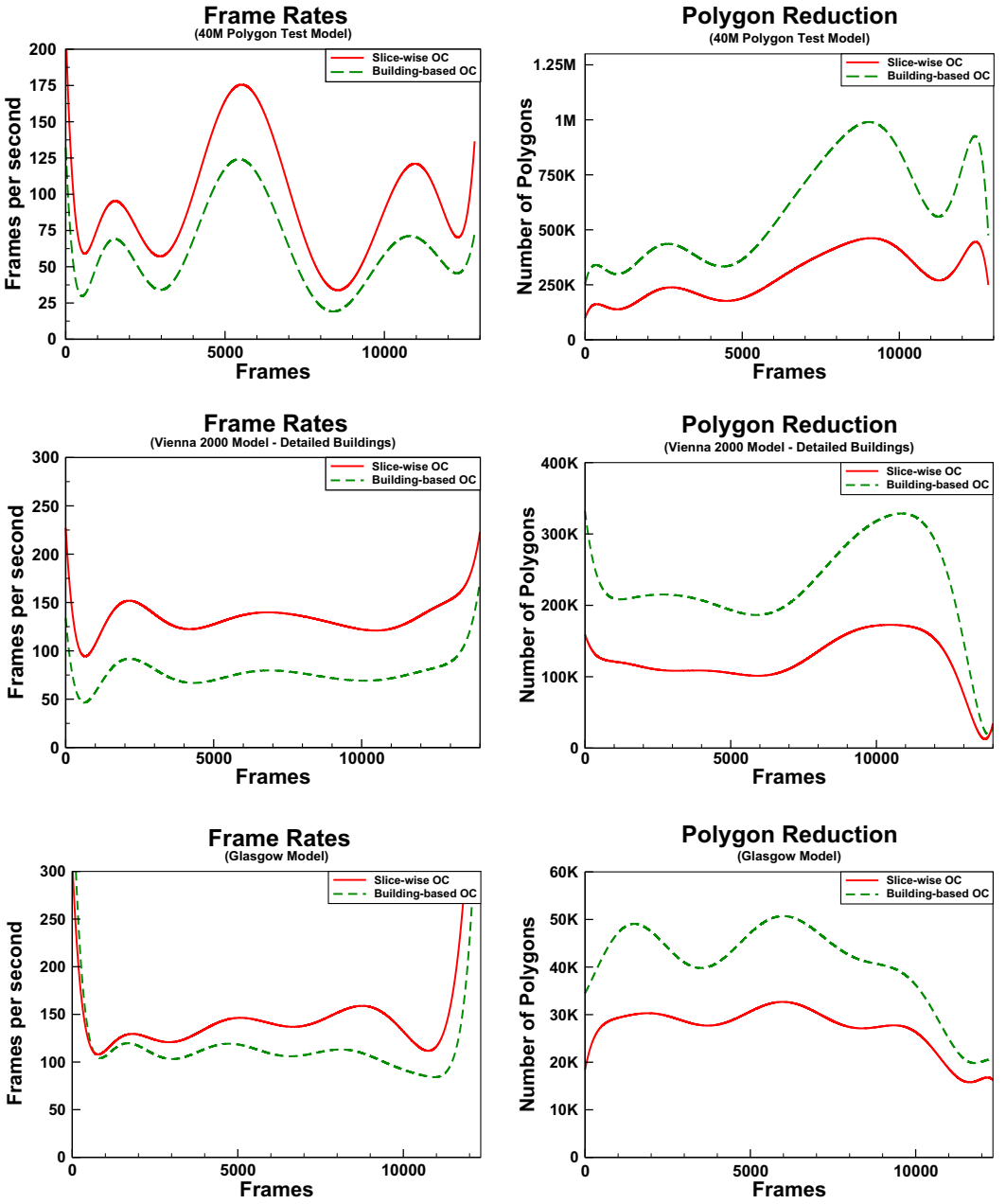

Fig. 23. Frame rates (left column) and number of polygons rendered (right column) of the proposed slice-wise approach as compared to the building-level approach for each model. The average frame rate speedups are 81%, 73.7% and 26.7%; and the average polygon reductions are 51%, 46.3% and 34.6% for the procedurally generated, Vienna2000, and Glasgow models, respectively. One reason for the lower performance increase in the Glasgow model is that the average number of the polygons rendered are very small for both granularities and the GPU is not fully utilized. The other reason is that the mesh structure has long and thin triangles belonging to several slices, thereby decreasing the exploitation of the data structure.

72% to 99.4%; in [23] from 99.86% to 99.95%; in [11] 99.34% culling ratios are reported.

Fig. 23 also gives the number of polygons rendered for the slice-wise and building-level occlusion culling. Using the building-level granularity, 93 buildings and 592K polygons are drawn on the average for each frame. Using the slice-wise occlusion culling, 89 of these buildings are accepted as partially visible. This decreases the number of rendered polygons to 290K on the average, which is

Table 3
Comparison of the average frame rates and rendered polygon counts for slice-wise occlusion culling and building-level occlusion culling

	Model	Procedurally generated	Vienna 2000	Glasgow
Frame rate	Slice-wise	111.06	135.1	152.2
	Building-level	61.36	77.8	120.1
Polygon count	Slice-wise	290K	122.1K	26.2K
	Building-level	592K	227.6K	40.1K

Frame rate is in frames per second (fps); polygon counts are the average number of polygons rendered per frame.

approximately 49% of the number of polygons rendered with building-level occlusion culling. Table 3 gives the average frame rate and rendered polygon count comparisons for slice-wise and building-level occlusion-culling methods for the three test models.

5.3. PVS storage

The proposed slice-wise occlusion-culling algorithms are optimized to exploit their benefits. Applying each subdivision scheme and performing tests on their performances would require different optimizations. This would make the comparisons unbalanced. Therefore, we only give informal results of using octrees and polygon level occlusion-culling processes for their effects in terms of the resultant PVSs.

We compare PVS storage requirements of the slice-wise structure and other subdivision schemes, namely the octree-based and triangle-based PVS storage (see Fig. 7). In 45.50K navigable view-cells, there are four completely visible, 89 partially visible buildings and about 290K visible polygons on the average. The PVS created using the slice-wise structure is 52 MB, where each partially visible buildings is represented with 55 slices on the average.

For the octree structure, a building is represented with 4680 nodes to obtain the same granularity with the slice-wise structure, which requires a subdivision depth of 4. We assume 75% of this amount for the adaptive octree case, which is 3510 bytes. We further assume that each node of octree for the buildings is in the memory and the visible nodes are represented in bits, hence decreasing down to around 438 bytes/building. Totally, the octree-based PVS storage requirement is about 1.95 GB.

For the triangle level PVS storage, there are about 13.20 billion triangles (290K visible polygons for 45.5K view-cells), which should be encoded into bits resulting in about 1.65 GB. Thus, the storage requirement of the slice-wise structure is about 3.15% of the triangle-level PVS storage and 2.67% of the octree-based PVS storage. Since, the PVS storage requirement for the slice-wise data structure is the same for all subdivision levels, as the subdivision goes deeper, it becomes more advantageous to use it.

A rough comparison of the PVS storage requirement of the proposed approach with some well-known approaches is as follows. In [11], the model used consists of 82K view-cells with 7.8M triangles and a PVS with a size of 55 MB (building-level occlusion culling is used). In [21], the model used consists of 90K view-cells with 34M triangles. The authors employ a very powerful compression and decompression scheme for the PVS and they have 1.1 GB-size PVS for their environment (polygon-level occlusion culling is used). In our test environment, we have 45.5K view-cells and 40M triangles. The PVS created by our scheme is 52 MB without any compression.

For the scenes that have a lot of connectivity, it may be necessary to subdivide the scene into clusters as in [6,21,22]. The clustering approach is suitable for the cases where there is no natural object definition. However, the buildings are mostly disconnected for urban models. Thus, the cluster formation process is not very useful since the quadtree or k-d tree-based hierarchy for the ground locations of the buildings serves the same purpose, as shown in [23].

Our approach can also capture occlusion in bird's-eye view. However, since the occlusion becomes less and the amount of the visible geometry increases, it may be more suitable to combine the approach with LOD (Level-of-Detail) rendering approach, especially for the completely visible buildings.

6. Conclusions

In this paper, we proposed 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 axis-aligned slices and the slices rather than the whole objects are checked for occlusion.

We also showed how to shrink objects in a scene, including nonconvex ones, in order to use them as occluders for from-region conservative occlusion culling. Our shrinking algorithm can be used for any kind of object, not just 2.5D buildings in an urban scene. Our experiments showed that the proposed slice-wise occlusion culling provides a significant increase in frame rates and decrease in the number of processed polygons per frame as compared to a visualization using building-level occlusion culling. In addition, the slice-wise structure drastically reduces the PVS storage requirement.

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 of our method would be 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 as in [6,21,22].

The proposed approach works for flythrough-type navigations where the user can be above buildings. It is also suitable for tilted viewing directions. Since, the occluded parts in the urban model become less as the flying altitude increases, it would be helpful for real time rendering to integrate our method with other approaches, such as view-dependent refinement.

Acknowledgments

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 Codes 104E029 and 105E065. Al Model is courtesy of Viewpoint Datalabs International, Inc. The heptoroid model is courtesy of the University of California, Berkeley. Vienna2000 Model is courtesy of Peter Wonka and Michael Wimmer. Glasgow Model is courtesy of ABACUS–Architecture and Building Aids Computer Unit of the department of Architecture, University of Strathclyde. We are grateful to the anonymous reviewers for their constructive comments.

References

[1] H. Samet, The quadtree and related data structures, ACM Computing Surveys 16 (2) (1984) 187–260.

[2] OpenSG Forum, OpenSG—Open Source Scene Graph, <http://www.opensg.org>, 2000.

[3] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, D. Manocha, MMR: an interactive massive model rendering system using geometric and image-based acceleration, in: Proceedings of the Symposium on Interactive 3D Graphics, 1999, pp. 199–206.

[4] D. Staneker, D. Bartz, W. Strasser, Occlusion culling in OpenSG PLUS, Computers & Graphics 28 (2004) 87–92.

[5] C. Saona-Vázquez, I. Navazo, P. Brunet, The visibility octree: a data structure for 3D navigation, Computers & Graphics 23 (5) (1999) 635–643.

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

[7] D. Cohen-Or, Y. Chrysanthou, C.T. Silva, F. Durand, A survey of visibility for walkthrough applications, IEEE Transactions on Visualization and Computer Graphics 9 (3) (2003) 412–431.

[8] J.T. Klosowski, C.T. Silva, Efficient conservative visibility culling using the prioritized-layered projection algorithm, IEEE Transactions on Visualization and Computer Graphics 7 (4) (2001) 365–379.

[9] J.T. Klosowski, C.T. Silva, Rendering on a budget: a framework for time-critical rendering, in: Proceedings of IEEE Visualization, 1999, pp. 115–122.

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

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

[12] D. Davis, W. Ribarsky, T.Y. Jiang, N. Faust, S. Ho, Real-time visualization of scalably large collections of heterogeneous objects, in: Proceedings of IEEE Visualization, 1999, pp. 437–440.

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

[14] J. Bittner, J. Prikryl, P. Slavik, Exact regional visibility using line-space partitioning, Computers & Graphics 27 (4) (2003) 569–580.

[15] T.A. Funkhouser, C.H. Sequin, S.J. Teller, Management of large amounts of data in interactive building walkthroughs, ACM Computer Graphics (Proceedings of ACM Symposium on Interactive 3D Graphics) 25 (2) (1992) 11–20.

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

[17] J. Bittner, P. Wonka, M. Wimmer, Fast exact from-region visibility in urban scenes, in: Proceedings of the 15th Eurographics Workshop on Rendering Techniques, 2005, pp. 223–230.

[18] L. Downs, T. Möller, C.H. Séquin, Occlusion horizons for driving through urban scenes, in: Proceedings of SIGGRAPH, 2001, pp. 121–124.

[19] D. Bartz, M. Meißner, T. Hüttner, OpenGL-assisted occlusion culling for large polygonal models, Computers & Graphics 23 (5) (1999) 667–679.

[20] C. Andújar, C. Saona-Vázquez, I. Navazo, P. Brunet, Integrating occlusion culling and levels of detail through hardly-visible sets, Computer Graphics Forum 19 (3) (2000) 499–506.
[21] J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, S. Venkatasubramanian, D.S. Johnson, vLOD: high-fidelity walkthrough of large virtual environments, IEEE Transactions on Visualization and Computer Graphics 11 (1) (2005) 35–47.
[22] S.-E. Yoon, B. Salomon, R. Gayle, Quick-VDR: out-of-core view-dependent rendering of gigantic models, IEEE Transactions on Visualization and Computer Graphics 11 (4) (2005) 369–382.
[23] T. Leyvand, O. Sorkine, D. Cohen-Or, Ray space factorization for from-region visibility, ACM Transactions on Graphics 22 (3) (2003) 595–604.
[24] G. Schaufler, J. Dorsey, X. Decoret, F.X. Sillion, Conservative volumetric visibility with occluder fusion, in: Proceedings of SIGGRAPH, 2000, pp. 229–238.
[25] F. Durand, G. Drettakis, J. Thollot, C. Puech, Conservative visibility preprocessing using extended projections, in: Proceedings of SIGGRAPH, 2000, pp. 239–248.
[26] J. Heo, J. Kim, K. Wohn, Conservative visibility preprocessing for walkthroughs of complex urban scenes, in: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 2000, pp. 115–128.
[27] V. Koltun, Y. Chrysanthou, D. Cohen-Or, Virtual occluders: an efficient intermediate PVS representation, in: Proceedings of the Eurographics Workshop on Rendering Techniques, Springer-Verlag, Berlin, 2000, pp. 59–70.
[28] C.L. Bajaj, V. Pascucci, G. Zhuang, Progressive compression and transmission of arbitrary triangular meshes, in: Proceedings of IEEE Visualization, 1999, pp. 307–316.
[29] J. Popović, H. Hoppe, Progressive simplicial complexes, in: Proceedings of SIGGRAPH, 1997, pp. 217–224.
[30] J. Rossignac, Geometric simplification and compression in multiresolution surface modeling, in: SIGGRAPH Course Notes #25, 1997.
[31] M.V.D. Panne, A.J. Stewart, Efficient compression techniques for precomputed visibility, in: Proceedings of Eurographics Workshop on Rendering, 1999, pp. 306–316.
[32] V. Koltun, Y. Chrysanthou, D. Cohen-Or, Hardware-accelerated from-region visibility using a dual ray space, in: Proceedings of the 12th Eurographics Workshop on Rendering Techniques, Springer-Verlag, London, UK, 2001, pp. 205–216.
[33] J. Bittner, P. Wonka, M. Wimmer, Visibility preprocessing for urban scenes using line space subdivision, in: Proceedings of Pacific Conference on Computer Graphics and Applications, 2001, pp. 276–284.
[34] T. Hüttner, M. Meissner, D. Bartz, Opengl-assisted visibility queries of large polygonal models, Tech. Rep. WSI-98-6, Dept. of Computer Science (WSI), University of Tübingen (1998).
[35] J.T. Klosowski, C.T. Silva, The prioritized–layered projection algorithm for visible set estimation, IEEE Transactions on Visualization and Computer Graphics 6 (2) (2000) 108–123.
[36] T. Yılmaz, U. Güdükbay, Extraction of 3D navigation space in virtual urban environments, in: Proceedings of the 13th European Signal Processing Conference, 2005.
[37] C. Gotsman, O. Sudarsky, J.A. Fayman, Optimized occlusion culling using five-dimensional subdivision, Computers & Graphics 23 (5) (1999) 645–654.
[38] B. Aronov, H. Brönnimann, A.Y. Chang, Y. Chiang, Cost-driven octree construction schemes: an experimental study, in: Proceedings of 19th Annual ACM Symposium on Computational Geometry, 2003, pp. 227–236.
[39] P.K. Agarwal, M. Sharir, Arrangements, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science Publishers B.V., North-Holland, Amsterdam, 1999, pp. 49–119.
[40] M. Schmitt, J. Mattioli, Morphologie Mathématique, Masson, Paris, 1993.
[41] G. Varadhan, D. Manocha, Accurate minkowski sum approximation of polyhedral models, in: Proceedings of the Pacific Conference on Computer Graphics and Applications, IEEE Computer Society, 2004, pp. 392–401.
[42] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright, Simplification envelopes, in: Proceedings of SIGGRAPH, 1996, pp. 119–128.
[43] N.K. Govindaraju, M.C. Lin, D. Manocha, Fast and reliable collision culling using graphics hardware, IEEE Transactions on Visualization and Computer Graphics 12 (2) (2006) 143–154.