# Tight Conservative Occlusion Culling for Urban Visualization

TÜRKER YILMAZ and UĞUR GÜDÜKBAY
Bilkent University

In this paper, we propose a framework for the visualization of urban environments. In this context, we also propose a novel tight conservative occlusion culling algorithm, which brings a solution to the partial occlusion problem. A conservative visibility culling algorithm accepts an object as completely visible even only a small portion of it is unoccluded. In our algorithm, we tried to avoid this by decomposing the objects into slices to achieve tight conservativeness. This sliced structure is queried for visibility at predefined grid locations in the scene. Since the slices of objects are checked for visibility instead of complete objects, only a very small portion of polygons of an object are sent to the graphics pipeline unnecessarily if a small portion of the object is unoccluded. The grid cells with approximately the same visibility lists are clustered to decrease the size of the visibility information that will be used during the visualization process. Empirical results show that an average speed up of 41% can be achieved using our tight conservative occlusion culling algorithm as compared to the classical conservative occlusion culling approach.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three Dimensional Graphics and Realism; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Curve, surface, solid, and object representations*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: 3D navigation, clustering, conservative visibility, occlusion culling, urban visualization, visibility preprocessing.

## 1. INTRODUCTION

Visualization of large geometric environments has always been an important field of computer graphics. Modern graphics workstations allow rendering millions of polygons per second. On the other hand, the amount of data that is needed to be processed increases dramatically, as well. No matter how much graphics hardware evolves, it cannot catch up with the quality demanded from these systems. Thus, speed up techniques that reduce the workload of the graphics pipeline become important.

Each visualization application has different requirements. For example, view-dependent visualization is especially suitable for terrain data. Performing occlusion culling on terrain data has of secondary importance, even negligible. However, occlusion culling has utmost importance in the case of urban scenes. For interactive fly-through of a city, a system must

Authors' Address: Department of Computer Engineering, Bilkent University, 06800 Bilkent, Ankara, Turkey. Tel: +90 (312) 290 13 86, Fax: +90 (312) 266 40 47. e-mail: {yturker, gudukbay}@cs.bilkent.edu.tr. The first author is partially supported by Turkish Scientific and Technical Research Council (TÜBİTAK).

store in memory and render only what is seen from the current viewpoint. The most important challenge is to identify the relevant portions of the model, swap them into memory and render at interactive frame rates, as the user changes position and viewing direction.

The efficiency of the visibility algorithm has vital importance for making an urban visualization system usable on common hardware. For view-frustum culling and back-face culling, which are the ways to speed-up the visualization, the current algorithms seem to be efficient enough. However, occlusion culling algorithms are still very costly. Especially, the object-space algorithms strongly need precomputation of the visibility for each viewpoint and viewing direction. Besides, most occlusion culling algorithms make assumptions that restrict the scenes to be visualized, such as the ones that use building footprints to construct the city [Wonka et al. 2000; Wonka et al. 2001].

One property of conservative object-space algorithms is that an object could be sent to the graphics pipeline even if a small portion of it becomes visible. However, in most cases this results in unnecessary overloading of the hardware, especially if the objects are very complex, containing ten to hundred thousands of polygons. Therefore, a smart approach is necessary to create a tight conservative set of the visibility without causing further overheads. One solution could be subdividing large objects into smaller ones and create hierarchies for the objects. Although it is possible to traverse the nodes of the hierarchy of an object to see which parts are visible, it becomes impractical to access the visibility lists in most cases.

In this paper, a novel object data structure is presented, where the objects are subdivided into slices at each dimension. For this purpose, the objects are first uniformly subdivided into cells. Then, the slices are formed using these cells. While performing occlusion culling during the preprocessing phase, we exploit the properties of this slice-wise structure and create a tight conservative display list of the scene for each viewpoint. The visibility data to be stored for the visualization during run time is also minimized.

Discretization and using a suitable object hierarchy are not enough for a visualization system. Since the data generated by the occlusion culling process is very large, the cellulization of the navigation space to reduce the size of the visibility information is crucial. Besides, if the implementation is a *fly-through* application instead of a *walkthrough* application, the calculation of the visibility on a cluster basis is inevitable. Our occlusion culling approach calculates the visibility on a grid base in three dimensions and makes use of the similarity between visibility lists of neighboring cells to create clusters of visibility. The created clusters are *not* uniform and may have any type of volume as long as the similarity criteria of the visibility lists are satisfied.

The contributions of the paper can be summarized as follows:

—A scene discretization and navigation space extraction algorithm, which automatically detects and establishes the navigation space through the entire scene, regardless of the architectural complexity of the objects appearing in the scene.

—A new data structure especially suitable for urban scenes, called *slice-wise structure*, which automatically leads to exploiting real world occlusion in urban scenes. The proposed occlusion culling approach reduces the data to be used during run-time to three bytes for each object and viewpoint, which indicates the visible slice indices for each dimension.

—A novel occlusion culling algorithm that creates a tight conservative visibility of the scene, exploiting the benefits of the proposed slice-wise object structure. The algorithm

creates occluder fusion by taking all occluders into account.

—A simple and efficient clustering algorithm, which creates clusters of visibility for 3D navigation through the entire scene. These clusters correspond to view-cells used in common urban walkthrough systems. One important aspect of the cluster creation scheme is that it is done without human intervention and can easily be used for any type of scene. Besides, it can also be used for 2D walkthrough systems.

In the next section, we discuss related work on scene discretization and navigable space extraction, occlusion culling, and visibility clustering. We describe the proposed framework for urban visualization in Section 3. The details of scene discretization and navigable space extraction are given in Section 4. We describe the slicing process in Section 5. The proposed tight conservative occlusion culling algorithm is explained in Section 6. The 3D clustering process and the navigation algorithm are explained in Sections 7 and 8, respectively. We give results of the empirical study in Section 9. Finally, we give conclusions and possible future extensions in Section 10.

## 2. RELATED WORK

### 2.1 Scene Discretization and Navigable Space Extraction

The extraction and cellulization of the navigation space is one of the most commonly used techniques to construct a suitable structure for visibility computations. Interestingly, there is not satisfactory work done for the extraction of the navigable area in 3D automatically. Discretization refers to the process of specifying object surfaces up to a certain roughness criterion independent of the size and shape of the polygons compromising the objects. The specification of the scene objects independent of their size and shape simplifies the occlusion culling process and the construction of clusters for visibility computations.

Generally, navigation space extraction for building interiors is not necessary, because rooms of the architectural models naturally correspond to cells [Funkhouser et al. 1992; Teller and Sequin 1991; Saona-Vázquez et al. 1999; Andújar et al. 2000]. However, the parts of the buildings should be explicitly specified as room, door, wall, window, etc. In [Funkhouser et al. 1992], cell-to-cell visibility is defined, where a portal sequence is constructed from a cell to the others if a sightline exist, thereby making a whole cell navigable. In [Wonka et al. 2000; Wonka and Schmalstieg 1999], the city model is extruded from building footprints where the navigable space information is directly available.

Sometimes, it is necessary to determine the navigable area during model design time. In [Schaufler et al. 2000], the developed system accepts streets or paths as navigable, which are explicitly specified. Their occlusion culling algorithm is suitable for extending the user navigation into 3D-space although it is not very straightforward. In [Downs et al. 2001], the user is assumed to be at two meters above streets. Besides, the created model has straight streets, making the navigation space determination process straightforward. In [Wonka et al. 2000; Wimmer et al. 1999; Durand et al. 2000], the navigation is implemented by assuming that the user is on the ground, where navigable space information is available in 2D.

### 2.2 Scene Representation

Scene representation has crucial impact on the performance of the visibility algorithm in terms of the memory requirement and the processing time. Many data structures were adopted for scene and object representation such as octrees [Samet 1984], or scene graph

hierarchy [OpenSG Forum 2000]. Especially, scene graph usage that provides fast traversal algorithms is very popular, such as [Staneker et al. 2004]. However, these are for the definition of object hierarchies. Direct usage of these structures may yield insufficient performance. Therefore, these structures are augmented with suitable fields to make them appropriate for occlusion culling. Besides, natural object structure is also modified in some applications. In [Saona-Vázquez et al. 1999], the triangles that belong to many nodes of the octree are used to separate the nodes by using their supporting planes for robust traversal. In [Baxter III et al. 2002], the objects could be divided into subobjects to create a balanced scene hierarchy, if necessary. In this study, we propose a slice-wise data structure for the objects in a scene that decreases the memory and processing costs of the visualization algorithm.

Occlusion culling on octree representations of objects is costly because the traversal on octree nodes requires too much processing time. Therefore, if the scene objects are stored using octree structure, the applications become suitable only for scenes where there are large occluders and a large portion of the whole model mostly stays behind these occluders. Visibility determination by traversing a scene hierarchy requires the the selection of occluders fast enough, which is a difficult task [Wonka et al. 2000; Funkhouser et al. 1992; Schaufler et al. 2000; Durand et al. 2000; Heo et al. 2000; Law and Tan 1999].

In the case of architectural walkthroughs, the rooms correspond to cells. These object structuring methods are suitable for indoor walkthroughs of buildings or ships where large occluders exist [Funkhouser et al. 1992; Saona-Vázquez et al. 1999; Gotsman et al. 1999; Teller 1992]. In [Lerner et al. 2003], the scene is partitioned using an approach that works for both indoor and outdoor environments. They accept occlusion of vertical polygons only and create additional portals on the scene to calculate the cell-to-cell visibility by applying a two pass algorithm on the half edges extracted from the scene.

Scenes may also be constructed using the extrusion of building footprints, as in [Wonka et al. 2000; Wonka et al. 2001]. However, this limits the complexity of the scene created and may not be applicable to all kinds of city plans.

### 2.3 Occlusion Culling

Occlusion culling methods determine the parts of the scene that are occluded with other objects and do not contribute to the images, which therefore should not be sent to the graphics pipeline. Since the amount of occlusion is high in city or architectural walkthroughs, the application of these algorithms are inevitable. In the case of terrains, it was reported that occlusion culling does not bring much performance gain [Hoppe 1998].

Occlusion culling algorithms can be classified based on the target environments. Some of them accept the scene as densely occluded, whereas LOD control and impostors contribute mostly in sparsely occluded situations. In the former, many of the geometry should be hidden behind large, preferably manually selected occluders [Funkhouser et al. 1992; Teller and Sequin 1991; Gotsman et al. 1999; Cohen-Or et al. 1998; Coorg and Teller 1997]. Otherwise, the cost of checking for visibility becomes high, which degrades the performance. The LOD control and impostors are most suitable for general scenes and usually require integration of different approaches together, as described in [Andújar et al. 2000; Law and Tan 1999; El-Sana et al. 2001; Germs and Jansen 2001].

Occlusion culling algorithms can be classified as *object space* and *image space*. The object space algorithms computationally decide on the visibility status of an object and does not use hardware properties [Schaufler et al. 2000; Heo et al. 2000; Cohen-Or et al.

1998; Coorg and Teller 1997; Klosowski and Silva 2001]. In the case of image space algorithms, the basic idea is to perform visibility computation for each frame by scan converting some potential occluders and checking if projection of the bounding volumes of the occludees fall entirely within the image area presented by the occluders [Wimmer et al. 1999; Durand et al. 2000; Greene 1993; Bartz et al. 1999; Greene 1999; Zhang et al. 1997; Wand et al. 2001].

Occlusion culling algorithms can also be classified as *conservative* and *approximate* [Cohen-Or et al. 2003]. Conservative algorithms mostly calculate the visibility on object basis and overestimate the visible set of objects. Instead of traversing an object hierarchy for fine tuned visibility, they either accept the entire object as visible or reject it. In order to tighten the conservativeness of visibility determination algorithms where large portions of the scene become hidden behind large occluders, traversal of hierarchical object representations is a useful approach since the traversal time pays for the rendering time of the occluded part.

In the case of approximate occlusion culling algorithms, such as [Klosowski and Silva 2001; 1999], the visibility primitives are estimated up to a certain threshold, i.e., some of them may not be sent to the graphics pipeline although they should be accepted as visible. There are also parallel approaches to occlusion culling, such as [Wonka et al. 2000; Baxter III et al. 2002; Davis et al. 1999]. A detailed survey on the occlusion culling algorithms can be found in [Cohen-Or et al. 2003].

Sending an object to graphics pipeline if a very small portion becomes visible causes unnecessary loading of the graphics hardware, degrading the performance. Therefore, a tight conservative determination of the visibility is desirable. In our work, the visibility is determined using the slice-wise data structure in the preprocessing phase, which allows us to determine a tight conservative approximation of the visibility in terms of the slices of the objects. This information is used for visibility determination during visualization, which requires constant time for each object.

## 2.4 Clustering

Clustering can be used for two different purposes. One of them is to create visibility list for occlusion, which is *from region visibility*. The other is used to compress the data produced by visibility computations. Generally, building interiors are used as natural clusters for from region visibility. Triangles of paths are used or the space is regularly subdivided for from region visibility in outside environments. In order to compress the data, there are many different approaches like [Bajaj et al. 1999; Popović and Hoppe 1997; Rossignac 1997; Yagel and Ray 1996; Panne and Stewart 1999].

Our approach to clustering for from region visibility is similar to the one described in [Panne and Stewart 1999]. Our technique depends on the visibility information gathered from discrete grid locations. However, if the application does not constrain user movements to the sampling locations, a naive approach may yield visual errors while moving between grid locations, i.e., it may regard an object as invisible, although it is visible. Since we do not limit user movements, we create additional clusters, called *passage clusters*, using the existing ones to get rid of this problem. The visibility list of a passage cluster is the union of the visibility lists of the neighboring clusters which form passage clusters.

## 3. THE VISUALIZATION FRAMEWORK

The general framework for urban visualization using tight conservative occlusion culling is shown in Figure 1. It mainly consists of a preprocessing phase and a navigation phase. In the preprocessing phase, we take the following steps:

—read the scene data and create bounding boxes for each object,
—discretize the objects,
—create slice-wise data structure for the objects,
—determine the navigation octree forest for the scene,
—perform occlusion culling for each grid location, and
—cluster the visibility list in 3D.

During navigation, we use the data collected in the preprocessing phase, while performing view-frustum culling and perform 3D navigation with unconstrained user movements.

## 4. SCENE DISCRETIZATION

This process is composed of three steps. First, a uniform subdivision is performed for each object where the grid cells occupied by each object are determined. Then, the navigable space is extracted as the complement of the grid cells occupied by the objects. This step produces an octree structure for each object using adaptive subdivision to reduce memory requirements. Finally, the occupied cells for each object in the uniform subdivision is converted to a slice-wise structure where the objects are represented with axis-aligned slices.

### 4.1 The Discretization Algorithm

We start by reading the scene data. The next thing is to calculate the bounding boxes of each object in the scene. The object discretization algorithm is based on grid cells with a user-defined size threshold. This threshold defines the roughness of the extracted mold of the object. The algorithm travels inside the bounding box of the object to find the occupied grid cells. A grid cell and a triangle may intersect in three ways, which are shown in Figure 2.

The first case is where any vertex (or vertices) of the triangle is inside the cell. This case is the easiest to determine, in which a range test gives the intended result (Figure 2 (a)). The second case, none of the vertices of the triangle is inside the cell but the triangle plane intersects the edges of the cell, is handled by performing ray plane intersection test (Figure 2 (b)). In the algorithm to detect this case, the main idea is to shoot rays from each corner of the cell to each coordinate axis direction. The last case (Figure 2 (c)), where the triangle penetrates the cell without touching any of its edges is handled in a similar way, but this time the rays are shot from the vertices of the triangle and checks are made against the surfaces of the cell. This process is repeated until all locations in the bounding box of the object is tested. A sample discretization for an object in 2D is shown in Figure 3 (a). With this approach, it is possible to use all holes and passages through the objects as part of the navigable area (see Figure 3 (b)).

### 4.2 Extraction of Navigable Space

Although the uniform subdivision provides the occupied cell information, which is enough to determine the navigable space, its memory requirement is high. In order to overcome

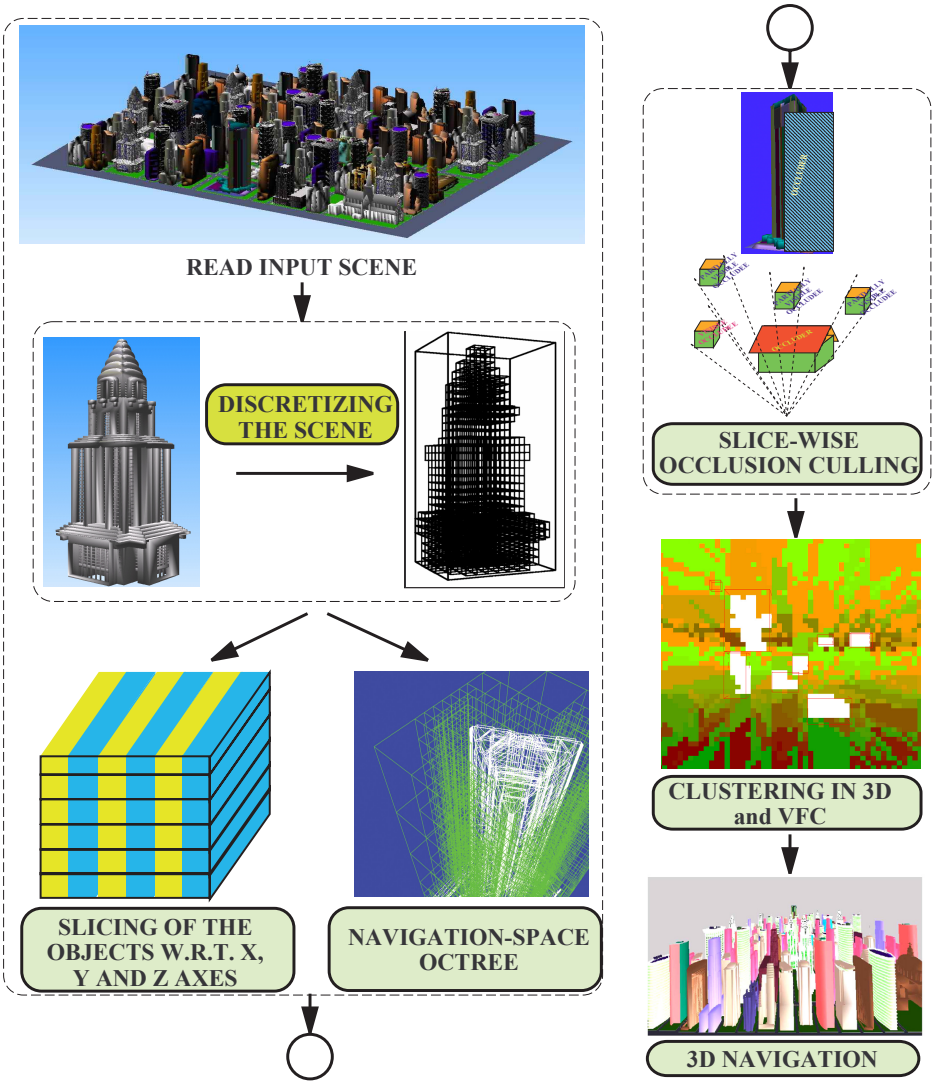



Fig. 1. General overview of the proposed framework: in the first phase, we read the scene and calculate bounding boxes of objects. Next, we discretize each object by checking each triangle if it intersects with a predefined threshold sized cube. After discretizing the object, we check the cubes for fullness to create slices and the navigation octree of the bounding boxed object. These slices are checked for occlusion and a tight conservative visibility determination is performed for each grid location. After clustering the visibility information for the grid cells in 3D, navigation in 3D can be done while performing view frustum culling on the visible data.

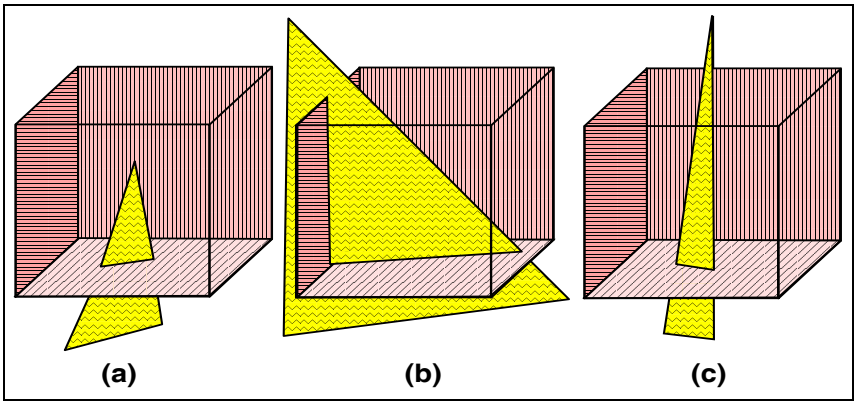
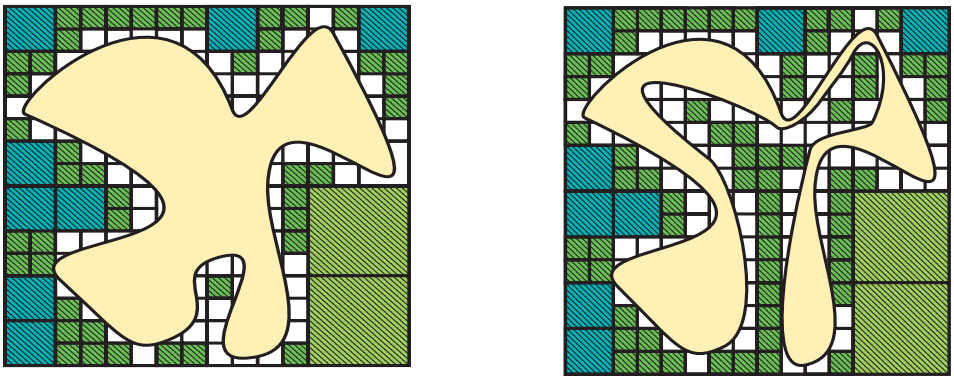



Fig. 2. Test cases: (a) *any vertex is inside the cell*; (b) *the vertices of the triangle is not inside the cell, but the cell edges intersect with the triangle surface*; (c) *the triangle edges intersect with the surfaces of the cell.*



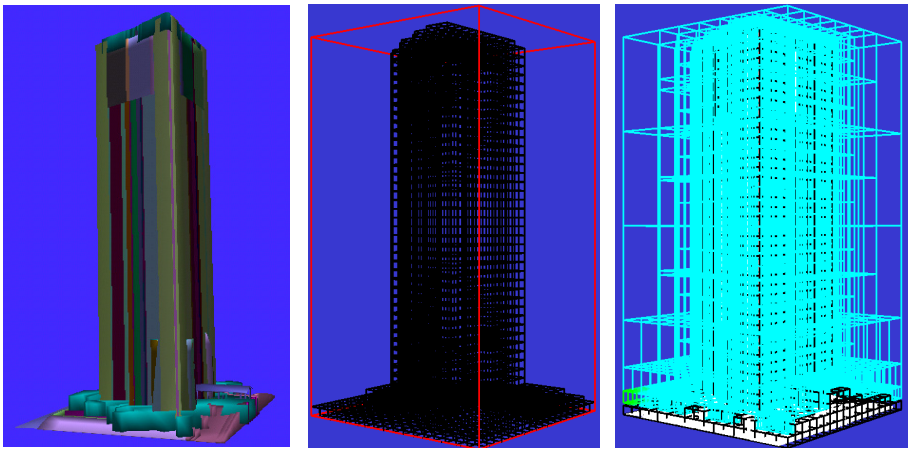
(a) (b)

Fig. 3. (a) Discretization of an object in 2D for easy interpretation. It is normally performed in 3D. The object is represented with uniform grid cells (filled boxes show the complement of the object space, which corresponds to the navigable area represented as an octree using adaptive subdivision to reduce memory requirements.) (b) Any holes and passages can safely be represented as part of the navigable area.

this problem, an adaptive subdivision is applied to the bounding box of the object to extract the navigable area as an octree structure. This is done using the occupied cell information provided by the uniform subdivision. An example of the created structure is shown in Figure 4. The navigation octrees for each object are tied up to the spatial forest of octrees that corresponds to the whole scene. The empty area outside the objects in the scene is also a part of the navigable space.

We did not make any assumptions on the type of scene objects, or on their respective locations, while determining the navigable space information. The objects may have

any type of architectural property, such as pillars, holes, balconies etc. Our algorithm indiscriminately finds the locations not occupied by any object part. This property makes our approach very suitable for the models that are created from different sources such as *LIDAR*, because the only information needed is triangle information, which most model formats have, or otherwise the primitives that are convertible to them.



(a) (b) (c)

Fig. 4. Navigation octree construction: (a) *the original object*; (b) *cells of the object, where the triangles of it pass through*; (c) *the created navigation octree, in which navigable space information is embedded. In the figure, the black lines show the occupied cells and the green lines show the boundaries of the navigation octree for the object.*

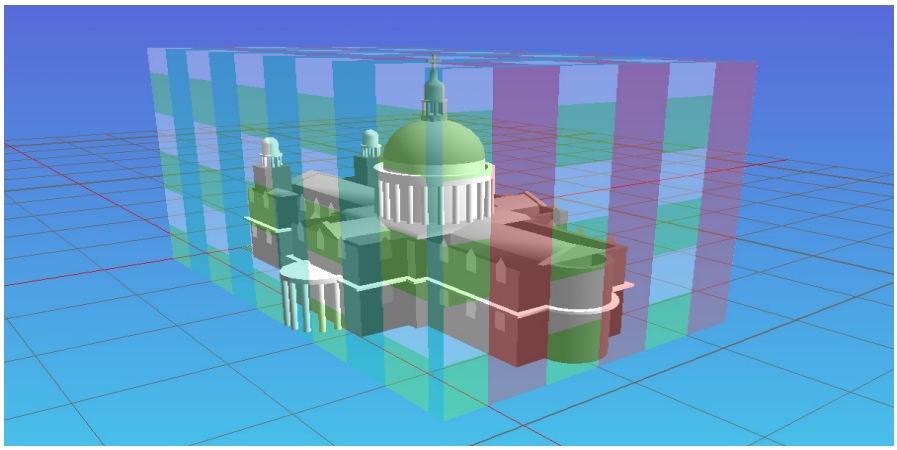
## 5. SLICE-WISE STRUCTURING OF OBJECTS

### 5.1 Slicing Objects

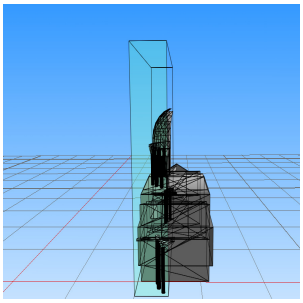
Slicing is an object representation where an object is equally divided into vertical and horizontal slices parallel to the coordinate axes and each slice knows which triangles pass through it. In order to create a structure like this, we make use of the uniform subdivision used to discretize the objects. During the discretization phase, the triangles are tested against grid cells and the cell occupancies of triangles are determined. Here, we check for the coverage of the slices over 3D grid cells and determine the slices along with their triangles. This procedure is repeated for each coordinate axis. The process of slicing an object is shown in Figure 5. The resultant scene data structure is shown in Figure 6.

### 5.2 Object Visibility Characteristics

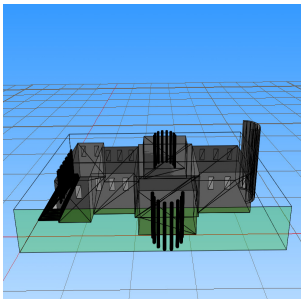
Our slice-wise approach is based on the observation that while a person is navigating through a city, the visible parts of the objects mostly have one of the following three forms (see Figure 7), which are the main points of our visibility algorithm.



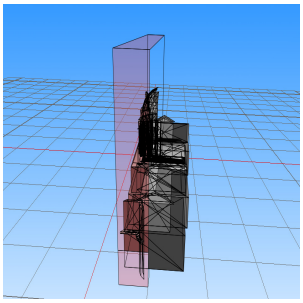
(a)



(b)



(c)



(d)

Fig. 5. The process of slicing an object: (a) *a complete view of the object where the positions of slices are shown on the bounding box*; (b) *slicing the object with respect to the x–axis. The triangles, which pass from this slice is attached to it*; (c) *slicing with respect to the y–axis*; (d) *slicing with respect to the z–axis.*

(1) If a building is occluded in part by a smaller occluder, the visible part looks like an *L-shaped* block in different orientations as in Figure 7 (a).
(2) If the occluder appears taller than the occludee, the visible part looks like a *vertical rectangular* block, from the left or right of the building (see Figure 7 (b)).
(3) 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 7 (c).

If the visible part of a building has a form other than the ones listed above, we can accept it as completely visible. Obviously, a visibility culling algorithm could be developed using even without an object data structure without characterizing visible parts of the buildings. However, this may cause a large number of polygons to be sent to the graphics pipeline

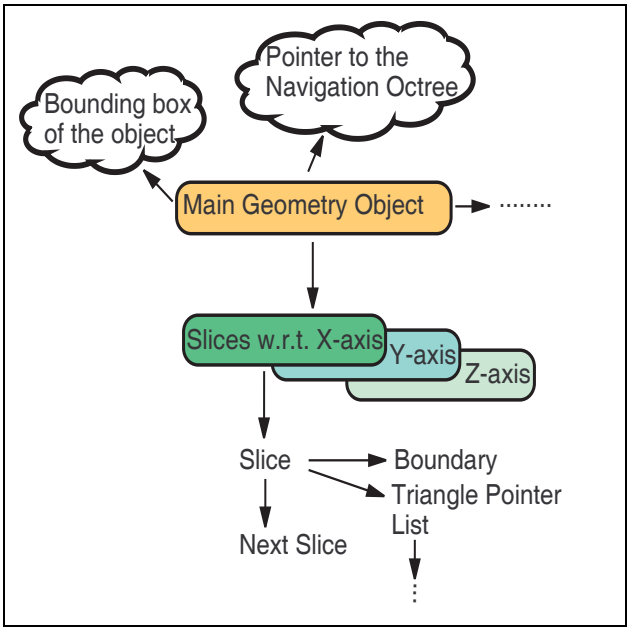
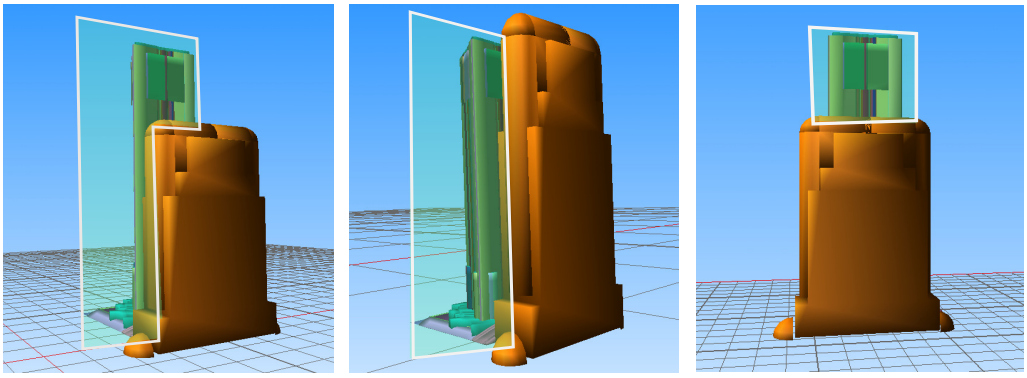



Fig. 6. The scene data structure produced by slicing operations.

unnecessarily for the sake of conservative visibility. If we could find a way to exploit these visibility characteristics with a little overhead, we could reduce the number of polygons sent to the graphics pipeline, which is what we achieve with slice-wise structure.



(a) (b) (c)

Fig. 7. Forms of visibility during urban navigation: (a) *L-shaped* visibility; (b) *vertical rectangular* visibility; (c) *horizontal rectangular* visibility. In the figure, the visible part of the occludee is the green transparent area.

### 5.3 Benefits of Slicing

In addition to facilitating the exploitation of different visibility characteristics for tight conservative visibility culling, the benefits of slicing objects are listed as follows:

—Slicing the objects provides a fast way to access portions of an object.

—Each triangle is covered by all three axes' slices. Therefore, we can use any combination during visibility, i.e. the axis that has the most occlusion.

—Memory requirements 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. This data specify the indices of the visible slices for each axis.

—The approach prevents unnecessarily overloading the graphics pipeline with invisible triangles just because only a small portion of the object is visible.

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

Since slicing is done in a sequential order, the definition of the visible portion only requires the cardinality of the number of the visible slices, depending on the position of occlusion as shown in Figure 8. Therefore, we only need to keep the last visible slice index for each axis. The slice indices are assigned with respect to each axis after they are checked for occlusion. In this way, we can access the visible portions of the objects very fast during navigation.

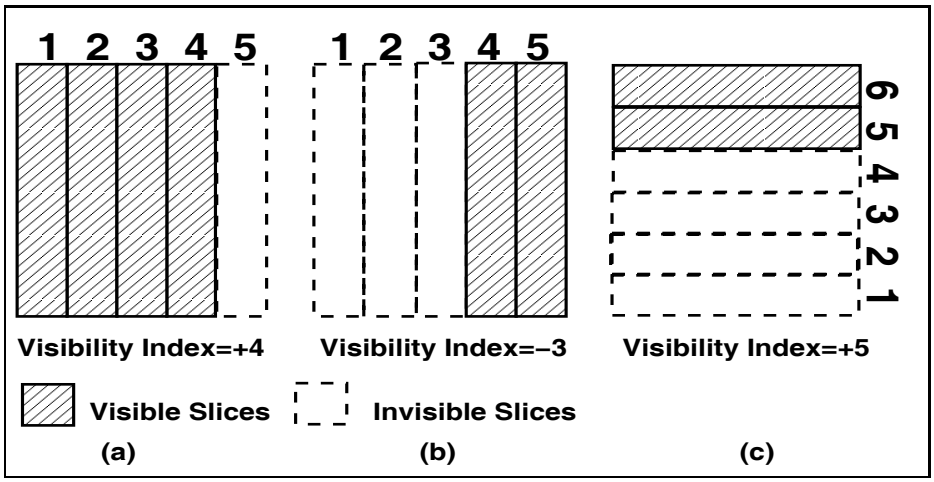



Fig. 8. Defining visibility indices for objects: the visible slice indices are determined for each axis during occlusion determination. (a) If an object is occluded in part from right, the index of the last visible slice is stored with a "+" sign. (b) If the object is occluded from left, then the index of the last invisible slice is stored with a "−" sign. (c) If the object is occluded from bottom, we keep the first visible horizontal slice of the object.

Accessing a triangle by using any of the three axial components gives us the flexibility for defining the visible portion. After occlusion determination, we can choose the maximally occluded side since the nature of slice-wise approach allows us to define it by using any of the axis. If a triangle is visible, it means it can definitely be accessed using any axial slicing. Choosing maximally occluded side allows us to tighten the conservativeness

as much as possible, as will be explained in the next section. After the elimination of unnecessary slices, we are left with a tight conservative visibility of the scene.

## 6. TIGHT CONSERVATIVE OCCLUSION CULLING

Current object space occlusion culling algorithms mostly accept an object as visible when a small portion of it becomes visible. During navigation over a city model, it is very frequent that a building is visible only by small portions. In these cases, it should not be necessary to send the whole object to the graphics pipeline for rendering. This results in unnecessarily overloading of the pipeline, especially for the cities that have individual building models with highly complex architecture.

### 6.1 The Occlusion Culling Algorithm

The occlusion culling algorithm works in the preprocessing phase. It tests all scene objects against each other and determines the occlusion amounts of each slice of the objects with respect to all occluders. This process is repeated for each navigable grid location.

The pseudo-code of the occlusion culling algorithm is given in Algorithm 1. The algorithm sorts the objects in a grid cell with respect to their bounding box corners. The reason of sorting is to detect a possible occlusion earlier to reduce further occlusion tests. We start from the nearest building and determine the visible sides of it by testing the viewpoint against its bounding box planes. If the test reports as *outside*, it means that the side is visible from the viewpoint. The visible sides are found to define the centroid of the object along with its left and right corners with respect to the viewpoint. This information is necessary to construct a correct transformation towards the occluder, as if looking at it during navigation.

In order to start the occlusion test, we use OpenGL's stencil buffer [Neider et al. 1994]. We draw the occluder as seen from a tested viewpoint to the stencil buffer for this purpose. Then, we start testing other buildings for occlusion against the image drawn to the stencil buffer. We test bounding boxes of candidate occludees to determine which objects may be occluded by the occluder. Current occlusion culling algorithms stop after this step and accept an object invisible, if all eight corners of the occludees bounding box remains in the stencil area of the occluder. However, this is too conservative. Therefore, we go through further steps and determine a tight conservative visibility for the object. In this step, we mark these objects as having intersection with the occluder. For all of the marked objects, we extract tight visibility information and optimize the number of visible slices.

The tight conservative occlusion culling test is given in Algorithm 2. It checks the slices of a candidate occludee on each axis using the slice-wise structure. We start from the lowest unoccluded point and increment the height by the threshold defined for the grid intervals as shown in Figure 9. Each time the stencil buffer is checked for a difference, which means there is an occlusion. This process is repeated for all vertical slices. For the horizontal slices, we check for complete occlusion (see Figure 10).

### 6.2 Optimizing the Visible Slice Counts

After finding occlusions in an object caused by many occluders, the appearance may be an irregular shape that may not be represented easily. However, our aim is to decrease the amount of information to represent the visibility, hence the access time to the visible parts of the object. In particular, the purpose of optimization is to represent the same

```
for each grid location in 3D do
    sort objects;
    for each building in the scene do
        if the building is visible then
            –Determine the two visible sides of the bounding box of the occluder;
            –Construct stencil and draw occluder;
            –Test bounding boxes of objects for intersection;
            if an intersection is found then
                Mark the object
    for each marked object do
        Tighten conservativeness using the test in Algorithm 2;
    for each building in the scene do
        Optimize visible slice counts (See Algorithm 4);
        Determine the building visibility status;
```

**Algorithm 1:** The occlusion culling algorithm: this algorithm finds a tight approximation of the visibility for each grid cell in 3D. The information is then passed to the clustering algorithm and the created clusters are used during navigation.

```
for each marked object that is visible do
    while X and Z slices are not finished do
        if the slice is not completely occluded by another object then
            slice_occlusion_height ⟵ find_exact_occlusion (occluder, slice);
            //(See Algorithm 3);
            if the slice is completely occluded by another object then
                mark the slice as INVISIBLE;
    while Y slices are not finished do
        if the slice is not occluded by another object then
            determine if the whole slice is occluded;
            if the slice is completely occluded by another object then
                mark the slice as INVISIBLE;
    if all slices are occluded then
        Mark the object as INVISIBLE;
    else
        Mark the object as PARTIALLY_VISIBLE;
```

**Algorithm 2:** The tight conservative occlusion test: slice-wise object structure is used to test all slices of the candidate occludee against the occluder.

visible area by using a small number of slices. Therefore, we have to release tightness of conservativeness a little to reduce the access time and memory requirement (see Figure 11).

```
Data    : (Occluder, Slice of the occludee)
Result  : The height of the minimum visibility
level ⟵ 1;
// Find how many grid units exist in the unoccluded part;
last ⟵ Ceil ( (height_of_slice − occluded_height_of_slice ) / grid_threshold);
slice_occlusion_height ⟵ occluded_height_of_slice;
while level ≤ last do
    to_be_tested_height ⟵ level ∗ grid_threshold + occluded_height_of_slice;
    draw the slice with to_be_tested_height into the stencil;
    is_hit ⟵ check_stencil_buffer( );
    // If yes, it means it is occluded;
    if is_hit then
        draw slice with to_be_tested_height out of the stencil;
        is_hit ⟵ check_stencil_buffer ( );
        //If yes, it means it is partially visible;
        if is_hit then
            slice_occlusion_height ⟵ ( level − 1 ) ∗ grid_threshold +
            occluded_height_of_slice;
            break;
        level ⟵ level + 1;
    else
        break;
return [slice_occlusion_height]
```

**Algorithm 3:** Finding exact occlusion: we first calculate the number of blocks to make occlusion calculation in discrete units. We test if a drawn block lies in the stencil area covered by an occluder, which means occlusion. If this is the case, the block is checked in the outside of the stencil. If a buffer difference occurs, then the block should be treated as visible.

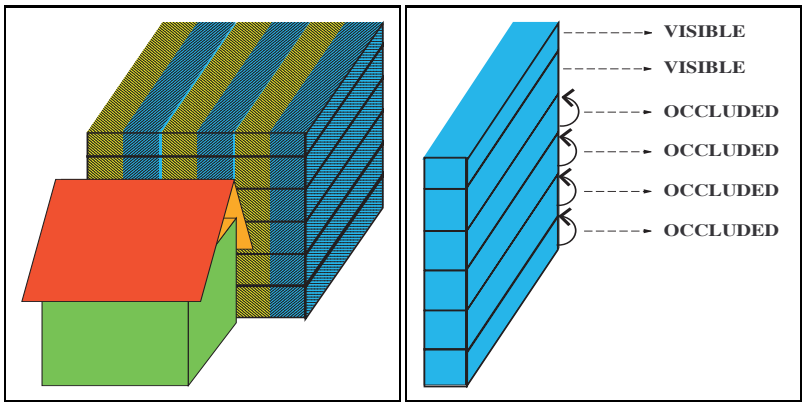
The algorithm for optimizing the slice counts is given in Algorithm 4. The algorithm is used to decrease the number of slices used to represent the visible portion for an occludee. In this algorithm:

—First, the maximally occluded axis is found by calculating the occluded regions and the percentages of occlusion with respect to each axis. Maximally occluded axis is selected for two reasons:
  i) Any triangle of the object is represented by three axes' slices. If a triangle is found to be visible after the occlusion tests, then all three slices to which it belongs will be visible.
  ii) If we discard the minimum occluded side and select the maximally occluded one, we will achieve maximum occlusion thereby decreasing the number of slices used to represent the same visible area.

—The rectangle that represents the occluded area is constructed.

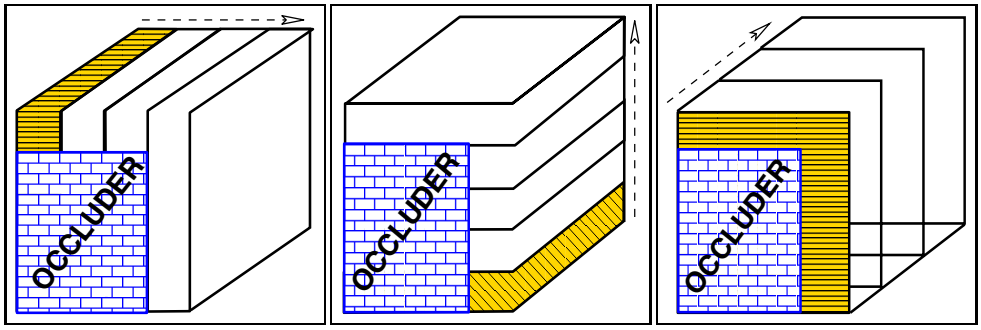
—For all the slices of the maximally occluded axis, we discard the vertical ones up to vertical edge of the rectangle and horizontal ones up to the upper edge.

—Finally, we discard slices of the minimally occluded axis.




(a) (b)

Fig. 9. In order to determine the correct occlusion height that may occur as in *(a)*, the slices are tested and the correct position of occlusion is found from bottom to top, as in *(b)*.




(a) (b) (c)

Fig. 10. Slices of the object are tested for occlusion for x, y and z-axes, respectively. (a,c) While vertical slices are tested for exact heights (b) horizontal slices (y-axis) are tested for being completely occluded. This may result in unnecessarily accepting the slice as visible. However, this case is handled by optimizing the slice counts. The arrows show the testing order.

One final word about this part is that if the occlusion of an object appears in the middle of the slice side, i.e., it divides the side into two, we accept the object as completely visible because this occlusion cannot be represented by our slice index representation (see Figure 8).

## 7. CLUSTERING THE VISIBILITY

Most of the approaches to occlusion culling assume that the user is navigating on the ground. Generally, these algorithms do not meet the requirements of fly-through applications where the user is not bound to the ground. The occlusion culling algorithm that

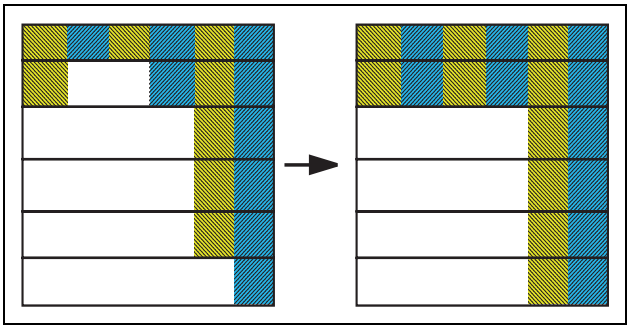


Fig. 11. The resultant shape of the occlusion may have a jaggy appearance. This has to be handled in a conservative way. Because, if we go on to describe the occlusion in its original form, the memory requirement for the partially occluded object will be very high. However, after this smoothing, we only need to describe the visible slice index as in Figure 8.

```
for each object in the scene do
    if the object is VISIBLE then
        // Determine the occluded region;
        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);
        For the working axis, determine the side (left or right) where the occlusion is
        maximum;
        Construct maximum sized rectangle of occlusion;
        while X and Z slices are not finished do
            // Modify slice occlusion values so that the region other than the
            occlusion rectangle is visible;
            slice_occlusion_height ⟵ occlusion_rectangle_height;
            Discard the occlusion values for the other axis;
        while Y slices are not finished do
            if horizontal_slice_height ≤ occlusion_rectangle_height then
                Discard the horizontal slice;
```

**Algorithm 4:** Algorithm for optimizing the occlusion for an object: The algorithm reduces the number of slices used to represent the visible portion for an occludee. First, maximally occluded axis is selected. Then, the rectangle that represents the occluded area is constructed. Finally, the occlusion heights of the slices are modified (see Figure 12).

we propose calculates the visibility for each cell in 3D by finding an optimized number of slices for each object that represent the visible area. The algorithm meets the requirements of fly-through applications where the user is not bound to the ground. Although the memory requirement is not too much, it can further be reduced. We reduce the memory requirement by clustering the grids with the same visibility (up to a pre-defined threshold) into regions. The clustering also eliminates possible visual errors that can occur when the user is moving between successive grid locations.

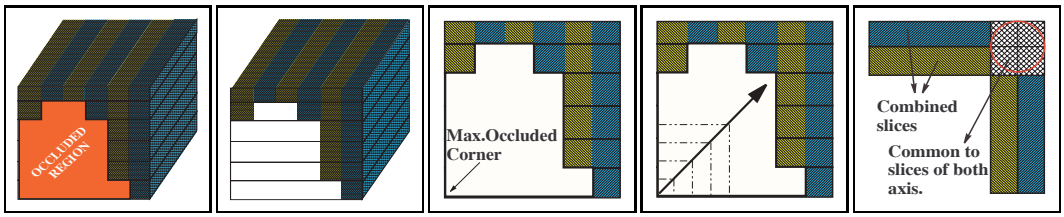



Fig. 12. Optimizing the visible slice counts of an occludee: *(a) and (b)* Due to occluder fusion, the resultant occlusion on an object may have a jaggy appearance. This is handled as described in Algorithm 4. *(c)* After selecting the maximally occluded axis, the starting corner of occlusion is determined. *(d)* The rectangle of occlusion is determined in a conservative way. *(e)* The occlusion heights of the slices are modified, by discarding the vertical ones up to vertical edge of the rectangle and horizontal ones up to the horizontal edge.

### 7.1 The Clustering Algorithm

The clustering algorithm is shown in Algorithm 5. The algorithm works as follows:

—For each grid location in 3D, we check whether the cell was assigned to a cluster before.

—If the cell was not assigned to a cluster previously, then we increment the global cluster id and assign the cell to the new cluster.

—The visibility list of the first cell in the cluster is transferred to a comparison list to compare with the neighboring grids.

—A recursive comparison algorithm (see Algorithm 6) finds the neighbors of the current cell. The similarity of the visibility is calculated by checking the slice indices of the objects for each neighboring grid. If the difference percentage is smaller than the predefined threshold, then the cell is assigned to the new cluster. The algorithm is recursively called until there are no neighboring grids that can be added to the cluster or all neighboring grids are assigned to a cluster. The similarity check is performed as follows:

i) If the signs of the two objects having visibility indices $A$ and $B$ are positive, then the similarity values for x- and z-slices are
$Slice\_index(min(A,B)) + Invisible\_portion(max(A,B))$,
and the similarity values for y-slices are
$Visible\_portion(max(A,B)) + Invisible\_portion(min(A,B))$.

ii) If the signs are negative, then the similarity is
$Visible\_portion(min(A,B)) + Invisible\_portion(max(A,B))$.

iii) If the signs are different, then the similarity is $Abs(A+B)$.

iv) If one of the objects is visible, then the similarity is the other's invisible portion.

—The cluster data structure is created for the determined clusters.

—The superset of each individual list is assigned to each grid location.

—Object pointers and slice indices are assigned to the cluster data structure.

—Passage clusters are created.

After the clustering algorithm is finished, we are left with a structure similar to the one shown in Figure 14. During navigation, we only need a three dimensional integer

```
global_cluster_id ⟵ 0;
for each grid location in 3D do
    if the grid location does not belong to any cluster then
        if the location is navigable then
            global_cluster_id ⟵ global_cluster_id + 1;
            grid_location_belonging_cluster ⟵ global_cluster_id;
            comparison_list ⟵ current_cell_visibility;
            compare(current_cell, comparison_list) (See Algorithm 6);
create cluster blocks;
equalize the visibility lists of grid cells within clusters;
add objects to clusters;
create passage clusters (See Algorithm 7);
```

**Algorithm 5:** The clustering algorithm: the visibility lists assigned to each grid location are used to create a conservative superset visibility list for the cluster. The cells whose visibility lists differ with a user specified deviation threshold are clustered together. After this algorithm, we are left with an array of clusters pointing to the visible list of objects and their slices for each cluster, which will be used during navigation.

```
Data   : (current_cell, comparison_list)
Result : Cells having a visibility list, different from the neighbor no more than the
         deviation threshold
Find neighbors of the current cell, which are not assigned to any cluster;
while neighbors are not finished do
    for each object in the scene do
        similarity_value + ⟵ calculated_similarity;
        difference ⟵ (total_slices – similarity_value) / total_slices;
        if difference ≤ deviation_threshold then
            neighbor_cell_cluster_id ⟵ global_cluster_id;
            compare(neighbor_cell, comparison_list);
```

**Algorithm 6:** The comparison algorithm is used to determine which grid locations can be added to the cluster, by testing neighboring cells' visibility. The created clusters may have any size and shape as in Figure 13.

array, storing the indices of the cluster array, and the cluster data structure along with the geometry object data structure.

### 7.2 Problems with Naive Clustering

Although the visibility is valid while moving inside a cluster, there may be some visual errors while passing to a neighboring cluster. The problem is illustrated in Figure 15. The visibility is determined by sampling at discrete grid locations. While in a border cell of a cluster an object is invisible, it may be visible in the neighboring cell belonging to another cluster. In this case, the object may be regarded as invisible up to the half way of the two

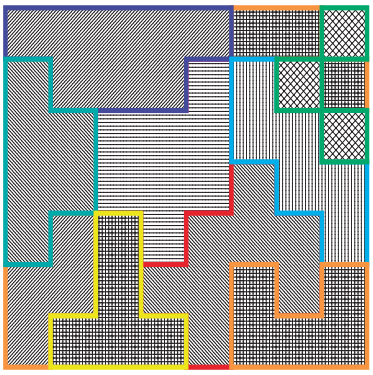
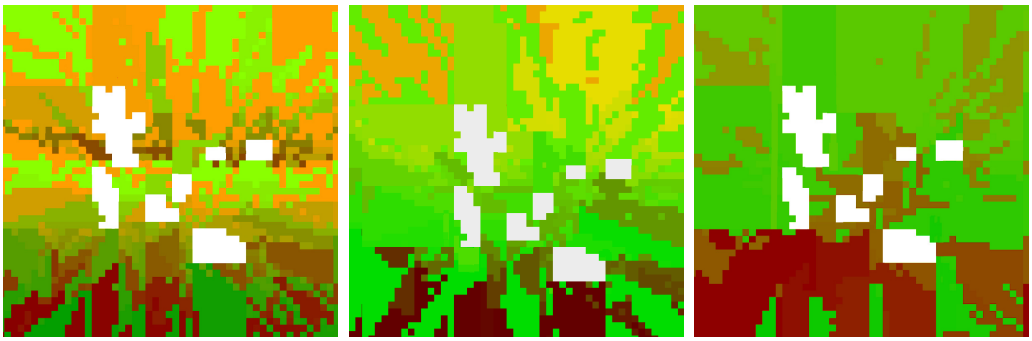


Fig. 13. A possible clustering of the scene, which is represented in 2D for the sake of simplicity.



(a) (b) (c)

Fig. 14. Samples from the created clusters of a small scene: Clusters created with *(a)* 5%, *(b)* 10%, *(c)* 20% deviation threshold, which means the visibility lists of the grid cells in these clusters are different from each other at most with this percentage. White cells show the places where the buildings are located.

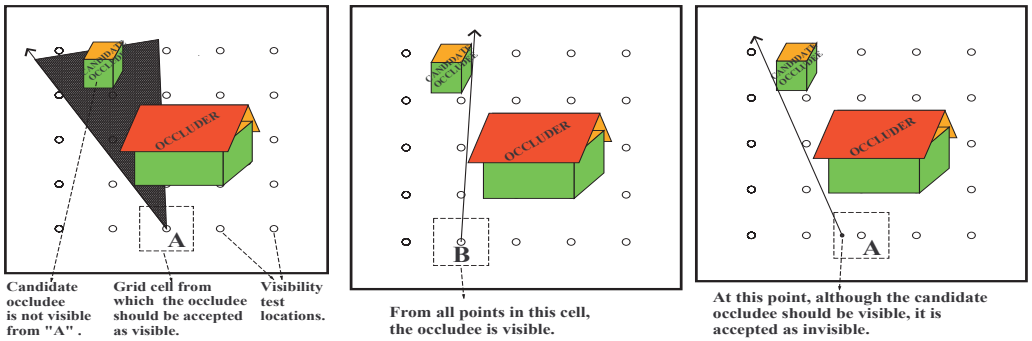
grid cells, which is a clustering problem that must be handled.

There are two possible solutions to this problem. One of them is to inform the system that the user is in the problematic area and the visibility must be handled in a conservative way. The other is to create *passage clusters* between the neighboring clusters having the union of the visibility lists of both clusters and discard the problem. We chose the second approach since it is robust and adapts to the general idea of our approach to visibility determination.

The algorithm for creating *passage clusters* is given in Algorithm 7. In this algorithm:

—For each grid cell, we find the neighbors and check if these neighbors are in the same cluster with the cell under consideration.

—If the cell is not in the same cluster then we check whether the cell's cluster was previously added to the list of neighbor clusters because a cluster may have neighboring relation with other clusters through many grid cells.




(a) (b) (c)

Fig. 15. Illustration of the visibility problem between clusters. The visibility is sampled at discrete grid locations. *(a)* After the sampling at the border grid cells of a cluster, the object could be regarded as *invisible*; *(b)* For the neighboring cluster the object may be regarded as *visible*. *(c)* However, up to the halfway from cell A, the user is accepted to be in *Cluster-A* and it will not be possible for the user to see the candidate occludee although it is visible.

—If the cluster was added before then we add the cell to the list of neighboring locations for the neighbor cluster.

—Otherwise, we add both the neighboring cluster and the cell to the list of neighbors.

—After determining the neighborhood information of the clusters, we go through the newly created *passage clusters* and construct the visibility lists of them by taking the union of the visibility lists of the neighboring clusters.

After handling the visibility problem between the clusters, the appearance looks like the one in Figure 16. These clusters behave as others and incur no overhead on the performance during navigation.

Although, passage cluster creation eliminates the clustering problems mentioned above, it cannot deal with the errors generated due to the nature of discrete visibility sampling methods, which occur rarely. As an example, assume in locations *A* and *B*, the objects *A'* and *B'* are visible, respectively. Again assume that, among the grids *A* and *B* there is another visible object, *C'*, which is not contained in either of the sampling locations. Since *C'* is not contained in either visibility lists, passage cluster that will be created using grids *A* and *B*, will not solve the problem, which is due to the nature of discrete visibility sampling. Excluding this problem, our approach can be accepted as error free.

## 8. THE 3D NAVIGATION ALGORITHM

The resultant structures to be used during navigation mainly include scene data structure, a three-dimensional array storing the cluster indexes and the cluster structure containing pointers to the scene objects for each cluster.

Since a triangle belongs to at least three axial slices, the navigation algorithm should detect existence drawing of a triangle in order to eliminate duplicate drawing. This is performed by assigning an index to each triangle to store current frame number. A triangle

```
for each grid location in 3D do
    active_cluster ⟵ cluster_id;
    Find all neighbors of the current_cell;
    while neighbors are not finished do
        if the cluster of the neighbor grid cell = active_cluster then
            loop;
        else
            check if the neighbor was added to the list of neighbor clusters before;
            if the cluster exists in the list of neighbor clusters then
                add the grid cell to the list of neighbor cells of the neighbor cluster;
            else
                add the cluster and its cell to the neighbor cluster list and neighbor
                grid cells of the neighbor cluster;
for each neighboring cluster do
    create a passage cluster;
for each passage cluster do
    construct conservative superset of the all of the neighboring clusters, in the
    newly created clusters;
    add the passage cluster to the list of clusters;
    redirect pointers of grids to passage clusters;
```

**Algorithm 7:** The passage cluster creation algorithm: first, the neighbors of each cluster are found. We determine the grid cells in which the clusters are neighbors of each other. The adjacent grid cells from two clusters forming the border line form a new cluster. This new cluster's visibility list is the union of the two neighboring clusters' visibility lists.

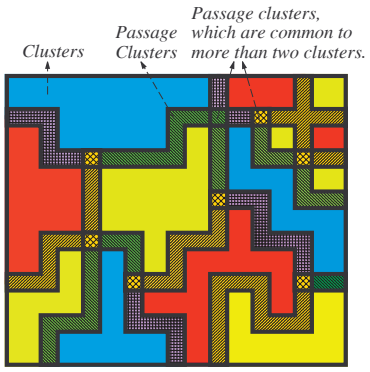



Fig. 16. The illustration of the passage clusters. The passage clusters are treated as normal clusters during navigation, incurring no performance overhead. The visibility set of a passage cluster may be a superset of more than two clusters.

is drawn to the scene if the slice to which it belongs is visible and the frame number assigned to it is different than the current one. By this way repetitive drawing is eliminated.

The algorithm for 3D navigation is given in Algorithm 8. We access the cluster index using the three-dimensional array storing the cluster indexes, which is valid for the current position. The display list contains pointers to the objects and their visible slice indexes.

The user is able to navigate freely on the scene. The navigation algorithm is supported by a view frustum culling algorithm, which eliminates the objects that are completely out of view frustum.

## 9. EMPIRICAL STUDY

The proposed algorithms were implemented on PC and workstation platforms using *C language* with *OpenGL* libraries. During the explanation of empirical study, *our approach* refers to the proposed slice-wise approach, whereas the *classical approach* refers to the conservative occlusion culling algorithms, in which the whole object is regarded as visible even if a tiny portion of it becomes visible.

We determined a navigation path to be used for our comparisons and repeated the navigation for different clustering thresholds. Clustering threshold refers to the percentage of difference in the visibility lists of the neighboring grid locations. As the cluster threshold increases, the degree of conservativeness increases, while the number of actively used clusters decreases. We compared our approach with the classical conservative occlusion culling approach. Our experiments showed that 41% average speed up can be obtained in frame rate with respect to the classical approach using the proposed slice-wise approach. The city model used in experiments consists of 150 very complex buildings with different architectures, each having around 10-30K polygons. The tests were performed on an Intel Pentium IV-2.8 Ghz. computer with 1 GB of RAM and ATI Radeon 9100 graphics card with 64 megabytes of memory. During the tests the scene objects were subjected to neither view dependent simplification nor view frustum culling.

We prepared a navigation of the scene with about 1200 frames (Figure 17). Still frames from the flythrough are given in Figure 18. Figure 19 shows the frame rates obtained during flythroughs using the proposed approach and the classical approach for different cluster thresholds. The graphs are smoothed using regression and running averages functions for easy interpretation.

The best clustering threshold is below 5% (see Figure 19 (a)-(c)). As the size of the clusters grows up, the frame rate decreases due to overconservative estimate of the visible polygons (Figure 19 (e) and (f)).

In Figure 20, the speed up of our approach as compared to the classical approach for different clustering thresholds is given. The best speed up is about 41% on the average and is achieved below 5% cluster thresholds. As the cluster threshold gets larger, the speed up drops down to 38%, 34%, 22% and 16%, respectively. Figure 21 shows the effect of our approach in decreasing the polygon counts with respect to the classical one. As in the case with frame rate speed up, the first three cluster threshold levels *(0%, 2.5% and 5%)*, show up the same gain, which is 33%. The other levels' gains are 31%, 26%, 19% and 14%, respectively.

Another issue to mention is that we experimented with an urban scene consisting of approximately 150 complex buildings because our only aim is to make a comparison with the classical conservative occlusion culling approach. For very large scenes containing thousands of buildings, the tight conservative nature of the proposed occlusion culling approach will show its effect much more since the portion of visible polygons will be

```
frame_index ⟵ frame_index+1;
// Determine the cluster id the user is in ;
x ⟵ round(longitude/grid_size);
y ⟵ round(altitude/grid_size);
z ⟵ round(latitude/grid_size);
active_cluster ⟵ cluster_pointers[x][y][z];
geometry_object ⟵ clusters[active_cluster].objects;
while geometry_object is not NULL do
    if geometry_object is completely visible then
        draw the whole object;
    else
        if geometry_object is partially visible then
            for each axial slice do
                if slice_index = 0 then
                    discard the side;
                else
                    if slice_index is negative then
                        Draw right slices of the absolute index number by checking{;
                        if triangle_index = frame_index then
                            discard the triangle;
                        else
                            triangle_index ⟵ frame_index;
                            Draw the triangle;
                        };
                    else
                        Draw left slices of the index number including itself by
                        checking{;
                        if triangle_index = frame_index then
                            discard the triangle;
                        else
                            triangle_index ⟵ frame_index;
                            Draw the triangle;
                        };
    geometry_object ⟵ geometry_object.next;
```

**Algorithm 8:** The 3D Navigation Algorithm: during run time the cluster index is determined using division of current coordinates by grid size. The cluster structure stores object pointers along with the visible slice index. The algorithm determines the current object list while checking for each triangle if it is drawn before.

much smaller as compared to the conservative occlusion culling approaches.

The empirical study presented reveals the fact that our occlusion culling scheme incurs no overhead on the visualization algorithm, since the determination of the visible portions

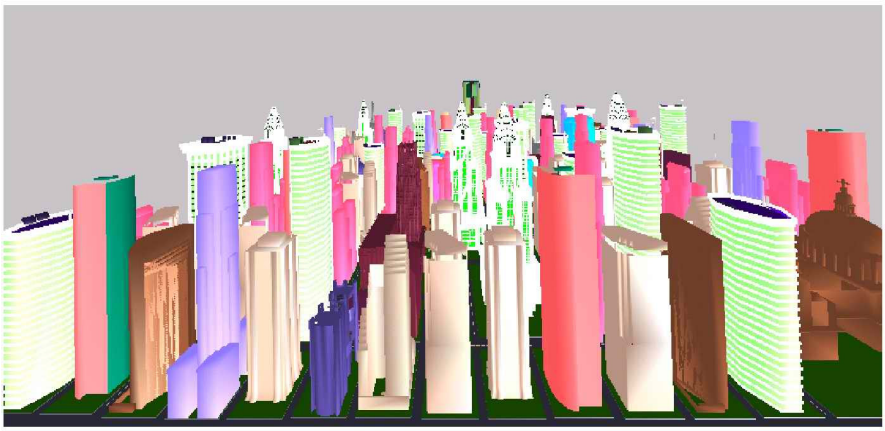


Fig. 17. The urban scene used in the experiments.

of the objects require almost constant time by using a few bytes for each object. Besides, the clustering scheme decreases memory paging need and decreases frame time.

## 10. CONCLUSION

In this paper, we proposed an occlusion culling approach for the visualization of urban environments. The proposed approach is tight conservative in the sense that it avoids sending the whole objects to the graphics pipeline if only a small portion of the object is visible, thereby bringing a solution to the partial occlusion problem. This is done by dividing the objects into slices and checking the slices for occlusion culling instead of the whole objects.

In the proposed visualization framework, the navigable area in 3D is extracted after the scene is discretized. Then, the sliced structure for the scene objects is queried for occlusion. Since we calculate the visible polygons for each cell in 3D, the proposed approach facilitates 3D navigation as compared to the other occlusion culling approaches that can be used in applications where the navigation is either bound to the ground or restricted in different ways. We also proposed a clustering scheme that decreases memory and paging requirements. These steps are performed in the preprocessing phase and the constructed scene structures are used during flythrough by the visualization algorithm. Empirical results showed that a speed up of 41% can be achieved using the proposed tight conservative occlusion culling approach.

Our occlusion culling approach is especially suitable for scenes having arbitrarily complex buildings since the benefits of determining partial occlusion can be realized in these types of scenes. Additionally, our approach can also be used for the visualization of scenes other than the urban scenery, although we did not test it.

The proposed slice-wise structure could also be used to increase the effectiveness of view dependent simplification. As a future work, we plan to use the slice-wise structure in a view dependent visualization framework.

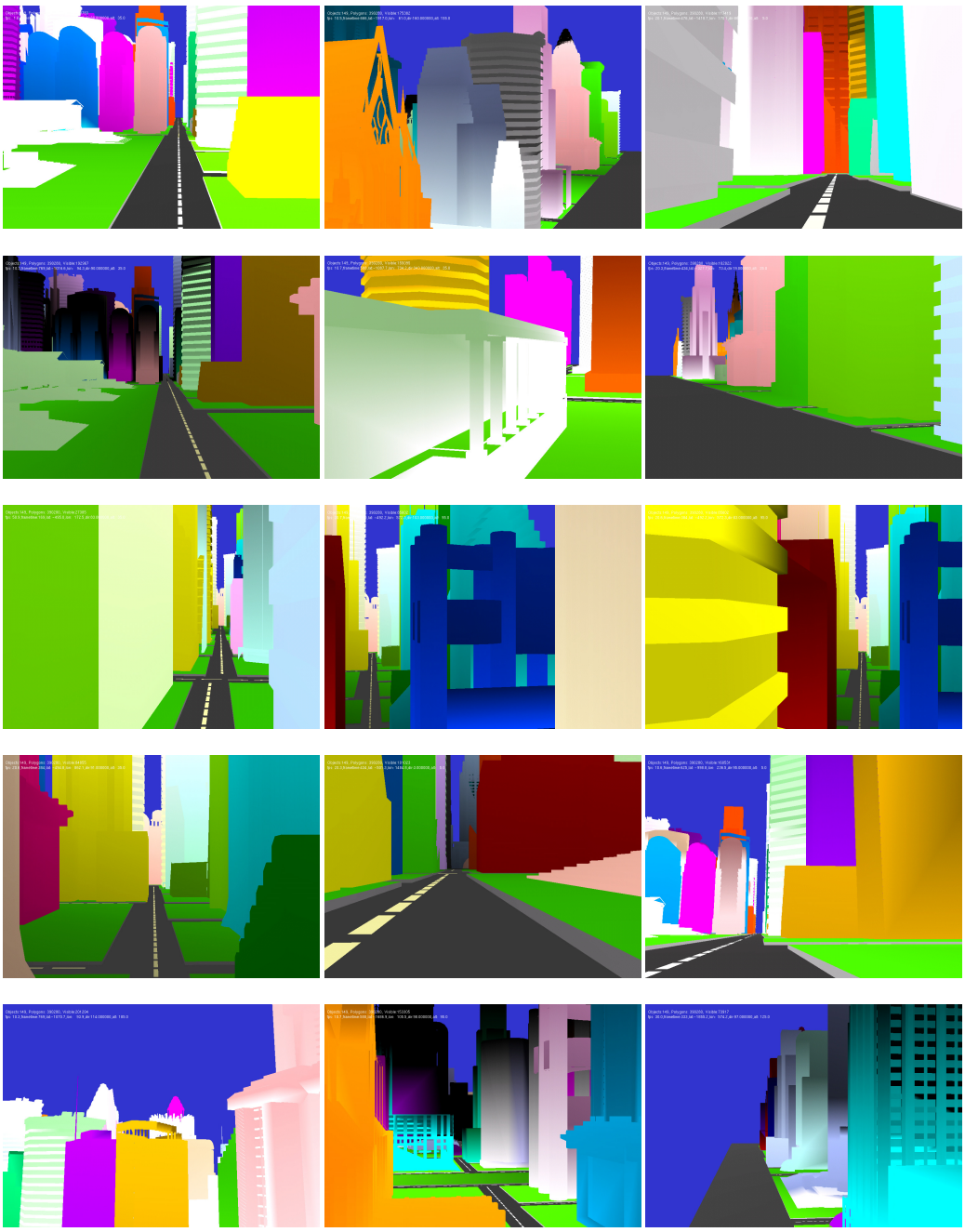


Fig. 18. Still frames from a navigation through the scene used in the experiments.

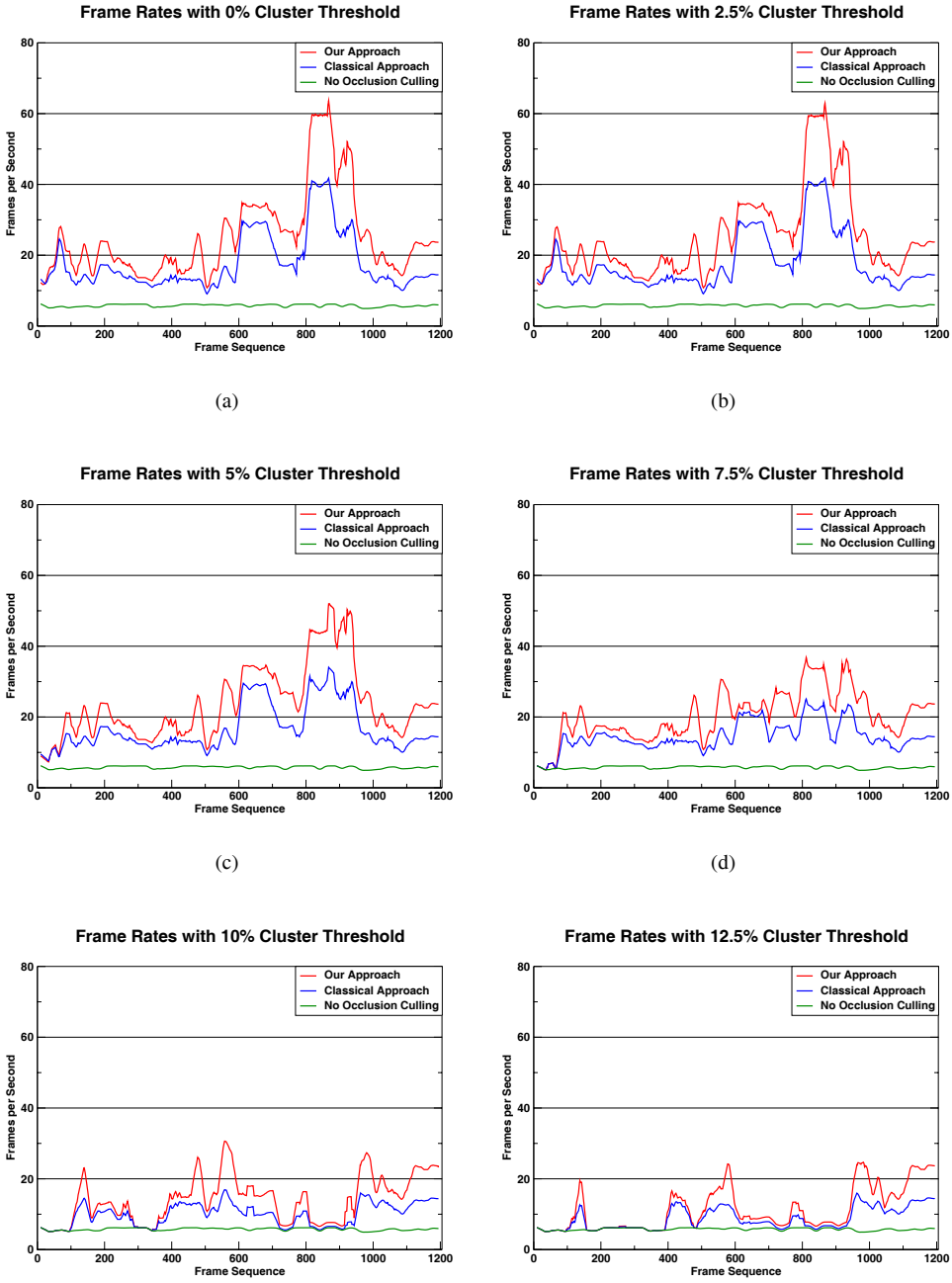



Fig. 19. Frame rates for different clustering threshold values.

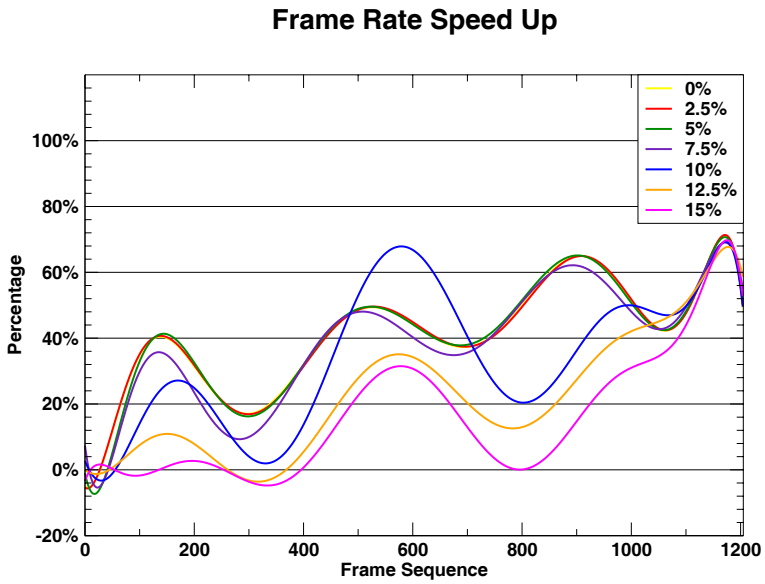



Fig. 20. Frame rate speedups of the proposed approach as compared to the classical approach for each clustering threshold. The gains are approximately the same (41%) for the first three clustering levels (0%, 2.5% and 5%). The average speed up for the remaining four are 38%, 34%, 22% and 16%, respectively.

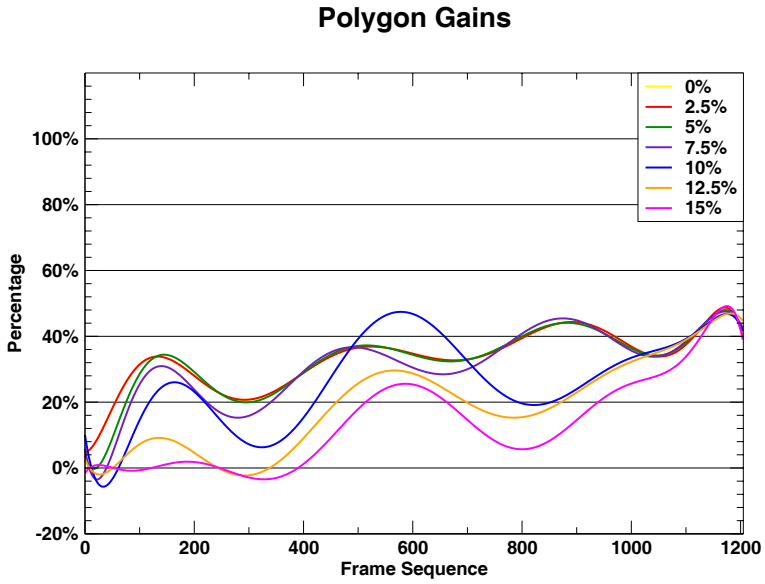



Fig. 21. Decrease in the number of polygons regarded as visible for each clustering threshold as compared to the classical approach. As in frame rate speed up, the gains are about the same (33%) for the first three clustering levels (0%, 2.5% and 5%). The average polygon gains for the remaining four clustering thresholds are 31%, 26%, 19% and 14%, respectively.

## REFERENCES


ANDÚJAR, C., SAONA-VÁZQUEZ, C., NAVAZO, I., AND BRUNET, P. 2000. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum 19,* 3, 499–506.

BAJAJ, C. L., PASCUCCI, V., AND ZHUANG, G. 1999. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings of IEEE Visualization'99*. 307–316.

BARTZ, D., MEISSNER, M., AND HÜTTNER, T. 1999. OpenGL-assisted occlusion culling for large polygonal models. *Computers & Graphics 23,* 5, 667–679.

BAXTER III, W. V., SUD, A., K.GOVINDARAJU, N., AND MANOCHA, D. 2002. Gigawalk: Interactive walkthrough of complex environments. In *Proceedings of 13th Eurographics Workshop On Rendering*. 203–214.

COHEN-OR, D., CHRYSANTHOU, Y., SILVA, C. T., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9,* 3, 412–431.

COHEN-OR, D., FIBICH, G., HALPERIN, D., AND ZADICARIO, E. 1998. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum 17,* 3, 243–254.

COORG, S. AND TELLER, S. J. 1997. Real-time occlusion culling for models with large occluders. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*. 83–90.

DAVIS, D., RIBARSKY, W., JIANG, T. Y., FAUST, N., AND HO, S. 1999. Real-time visualization of scalably large collections of heterogeneous objects. In *Proceedings of IEEE Visualization'99*. 437–440.

DOWNS, L., MÖLLER, T., AND SÉQUIN, C. H. 2001. Occlusion horizons for driving through urban scenes. In *Proceedings of SIGGRAPH'01*. 121–124.

DURAND, F., DRETTAKIS, G., THOLLOT, J., AND PUECH, C. 2000. Conservative visibility preprocessing using extended projections. In *Proceedings of SIGGRAPH'00*. 239–248.

EL-SANA, J. A., SOKOLOVSKY, N., AND SILVA, C. T. 2001. Integrating occlusion culling with view-dependent rendering. In *Proceedings of IEEE Visualization'01*. 371–378.

FUNKHOUSER, T. A., SEQUIN, C. H., AND TELLER, S. J. 1992. Management of large amounts of data in interactive building walkthroughs. *ACM Computer Graphics (Proceedings of ACM Symposium on Interactive 3D Graphics) 25,* 2, 11–20.

GERMS, R. AND JANSEN, F. W. 2001. Geometric simplification for efficient occlusion culling in urban scenes. In *Proceedings of WSCG'01*. 291–298.

GOTSMAN, C., SUDARSKY, O., AND FAYMAN, J. A. 1999. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics 23,* 5, 645–654.

GREENE, N. 1993. Hierarchical Z-buffer visibility. In *Proceedings of SIGGRAPH'93*. Vol. 27. 231–238.

GREENE, N. 1999. Efficient occlusion culling for Z-buffer systems. In *Proceedings of SIGGRAPH'99*. 78–79.

HEO, J., KIM, J., AND WOHN, K. 2000. Conservative visibility preprocessing for walkthroughs of complex urban scenes. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST-00)*. 115–128.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of IEEE Visualization'98*. 35–42.

KLOSOWSKI, J. T. AND SILVA, C. T. 1999. Rendering on a budget: A framework for time-critical rendering. In *Proceedings of IEEE Visualization'99*. 115–122.

KLOSOWSKI, J. T. AND SILVA, C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7,* 4, 365–379.

LAW, F. A. AND TAN, T. S. 1999. Preprocessing occlusion for real-time selective refinement. In *Proceedings of the ACM Symposium on interactive 3D Graphics*. 47–54.

LERNER, A., CHRYSANTHOU, Y., AND COHEN-OR, D. 2003. Breaking the walls: Scene partitioning and portal creation. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications (PG'03)*. 303–312.

NEIDER, J., DAVIS, T., AND WOO, M. 1994. *OpenGL Programming Guide*. Addison–Wesley.

OPENSG FORUM. 2000. *OpenSG – Open Source Scene Graph*. http://www.opensg.org.

PANNE, M. V. D. AND STEWART, A. J. 1999. Efficient compression techniques for precomputed visibility. In *Proceedings of Eurographics Workshop on Rendering*. 306–316.

POPOVIĆ, J. AND HOPPE, H. 1997. Progressive simplicial complexes. In *Proceedings of SIGGRAPH'97*. 217–224.

ROSSIGNAC, J. 1997. Geometric simplification and compression in multiresolution surface modeling.

SAMET, H. 1984. The quadtree and related data structures. *ACM Computing Surveys 16,* 2, 187–260.

SAONA-VÁZQUEZ, C., NAVAZO, I., AND BRUNET, P. 1999. The visibility octree: a data structure for 3D navigation. *Computers & Graphics 23,* 5, 635–643.

SCHAUFLER, G., DORSEY, J., DECORET, X., AND SILLION, F. X. 2000. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH'00*. 229–238.

STANEKER, D., BARTZ, D., AND STRASSER, W. 2004. Occlusion culling in OpenSG PLUS. *Computers & Graphics 28*, 87–92.

TELLER, S. J. 1992. Visibility computations in densely occluded environments. Ph.D. thesis, University of California, Berkeley.

TELLER, S. J. AND SEQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH'91*. Vol. 25. 61–69.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRASSER, W. 2001. The randomized z–buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH'01*. 361–370.

WIMMER, M., GIEGL, M., AND SCHMALSTIEG, D. 1999. Fast walkthroughs with image caches and ray casting. *Computers & Graphics 23,* 6, 831–838.

WONKA, P. AND SCHMALSTIEG, D. 1999. Occluder shadows for fast walkthroughs of urban environments. *Computer Graphics Forum (Proceedings of Eurographics'99) 18,* 3, 51–60.

WONKA, P., WIMMER, M., AND SCHMALSTIEG, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proceedings of Rendering Techniques*. 71–82.

WONKA, P., WIMMER, M., AND SILLION, F. X. 2001. Instant visibility. *Computer Graphics Forum (Proceedings of Eurographics'01) 20,* 3, 411–421.

YAGEL, R. AND RAY, W. 1996. Visibility computation of efficient walkthrough of complex environments. *Presence 5,* 1, 1–16.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. E. 1997. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH'97*. 77–88.