Optical Engineering 47(9), 097005 (September 2008)

Stereoscopic urban visualization based on graphics processor unit

Türker Yılmaz
Uğur Güdükbay
Bilkent University
Department of Computer Engineering
06800 Bilkent
Ankara, Turkey
E-mail: gudukbay@cs.bilkent.edu.tr

Abstract. We propose a framework for the stereoscopic visualization of urban environments. The framework uses occlusion and view-frustum culling (VFC) and utilizes graphics hardware to speed up the rendering process. The occlusion culling is based on a slice-wise storage scheme that represents buildings using axis-aligned slices. This provides a fast and a low-cost way to access the visible parts of the buildings. View-frustum culling for stereoscopic visualization is carried out once for both eyes by applying a transformation to the culling location. Rendering using graphics hardware is based on the slice-wise building representation. The representation facilitates fast access to data that are pushed into the graphics procesing unit (GPU) buffers. We present algorithms to access this GPU data. The stereoscopic visualization uses off-axis projection, which we found more suitable for the case of urban visualization. The framework is tested on large urban models containing 7.8 million and 23 million polygons. Performance experiments show that real-time stereoscopic visualization can be achieved for large models. © *2008 Society of Photo-Optical Instrumentation Engineers.* [DOI: 10.1117/1.2978948]

Subject terms: urban visualization; slice-wise representation; vertex buffer object (VBO); OpenGL graphics library; stereoscopic visualization.

Paper 080297R received Apr. 18, 2008; revised manuscript received Jul. 12, 2008; accepted for publication Jul. 18, 2008; published online Sep. 22, 2008.

1 Introduction

Visualizing urban environments is one of the most challenging areas in computer graphics, mainly because of the unorganized geometry and their complex nature. Attempts to reduce this complexity include either preprocessing or assuming simpler geometry for the buildings in the urban environment or both. And since virtual reality applications need twice the processing power of their monoscopic counterparts, it is crucial to send only the visible parts of the geometry to the rendering pipeline.

There are three ways to increase rendering performance. *View-frustum culling* (VFC) discards the objects that are out of the field of view. *Back-face culling* discards those polygons whose normals are facing away from the viewer. *Occlusion culling* eliminates the parts that are occluded by objects in front.

Urban environments provide the opportunity to detect a lot of occlusion during a walkthrough, which can be eliminated from the graphics pipeline as it does not contribute to the final view. Therefore, previous work has mostly concentrated on determining these occluded parts. The quality of a visibility algorithm depends on how fast it determines the visible parts of the model for different views, which are called *potentially visible sets* (PVSs), and the degree of tightness of the PVSs.

The advances in graphics hardware allow detection of occluded regions of urban geometry, even with complex 3-D buildings. Visual simulations, urban combat simulations, and city engineering applications require highly detailed models and realistic views of an urban scene. Occlusion detection using preprocessing is a very common approach, because of its high polygon reduction and its ability to handle general 3-D buildings.

Virtual reality applications require special treatment because the geometry is rendered twice, once for each eye. Generally, performance-enhancing techniques such as view-frustum culling (VFC) are applied twice for both eyes; this increases the overhead. We apply VFC only once for a viewpoint that is well placed for both eye coordinates rather than twice for stereoscopic visualization. The view calculated from this location has the same coverage as both eyes together.

We use the slice-wise representation of buildings for occlusion culling and rendering based on graphics processing unit (GPU). We assume that the PVSs are determined in preprocessing time, and the resultant visibility list is stored using a slice-wise building representation. We improve rendering performance using this representation through GPU-based rendering. In particular, we demonstrate how GPU can achieve high frame rates during stereoscopic visualization.

In the next section, we discuss related work in terms of occlusion culling, stereoscopic visualization, and slice-wise representation. In Sec. 3, we summarize the slice-wise representation of buildings. In Sec. 4, we describe the proposed stereoscopic urban visualization framework. In Sec. 5, we outline the performance study. Last, we provide our conclusions.

2 Related Work

Visibility determination is a well-studied area in computer graphics.[1] In order to achieve good stereoscopic visualiza-

tion, a good monoscopic correspondent must first be achieved. Therefore, we initially deal with the problem of speeding up monoscopic visualization by using powerful occlusion culling and VFC algorithms.

2.1 *Occlusion Culling*

In the special case of urban environments, most geometry is hidden behind other buildings; occlusion culling therefore provides significant gains in performance. In addition, most of the buildings are partially visible for different views during a walkthrough. Thus, identifying occluded parts of the buildings quickly and representing partial visibility is of vital importance.

Much work has focused on visualizing urban scenes composed of 2.5-D buildings—buildings constructed using their footprints. These have mainly used *object space* methods, which iterate over the scene objects and decide whether or not they are visible.[2–4] For example Ref. 5 discusses cell-to-cell visibility—a portal sequence is constructed from one cell to others where a sight line exists. *Image space algorithms* perform visibility computation for each frame by checking whether the projections of the bounding volumes of occluded buildings fall entirely within the image area covered by the occluders.[6–11]

Occlusion culling is performed either during visualization (on-line) or before visualization (off-line). On-line algorithms calculate the visibility during run-time.[12] However, the scalability is limited if no simplifying assumptions are made. To overcome this, geometry-reduction techniques such as view-dependent simplification schemes can be incorporated.[13,14] Off-line algorithms calculate visibility for a given region by discretizing the scene and determining the navigable area,[15] called *view-cells*. In this way, the preprocessed information can be calculated and stored for later use.

Occluder shrinking is a common approach of off-line algorithms. Using occluder shrinking, it is possible to determine occlusion from a specific point and use it for the entire view-cell region, because the occluders are shrunk by the maximum distance that a user can go in the view-cell (see Fig. 1). Wonka et al.[12] shrink occluders by using a sphere constructed around 2.5-D occluders.In Ref. 16, instead of a sphere, the authors calculate erosion of the occluder using a convex shape, which is the union of the edge convex hulls of the object. These two approaches are applicable to 2.5-D urban environments. Exact shrinking can be carried out only by using Minkowski differences of the view-cells and the occluders.[17] In Ref. 18, a Minkowski-difference-based occluder shrinking method is proposed; it can shrink 3-D objects and use them as occluders.

One of the biggest disadvantages of off-line occlusion culling algorithms is the difficulty of storing the visibility information for run-time use, especially when the scene is large, containing tens of millions of polygons. Since visibility information must be stored for each view-cell, the number of view-cells can total hundreds of thousands. Recently, a storage scheme for buildings, called the *slice-wise representation*, was developed; this facilitates the storage of partial visibility information for urban walkthroughs.[18] It can significantly reduce the size of PVS storage when compared to other commonly used storage schemes, such as octrees. The partial visibility information can be represented with 50% reduced polygons and 80% speed-up in frame rates when compared to occlusion culling using building-level granularity. The high reduction in storage requirements for partial visibility allows the visualization of large and complex urban models.

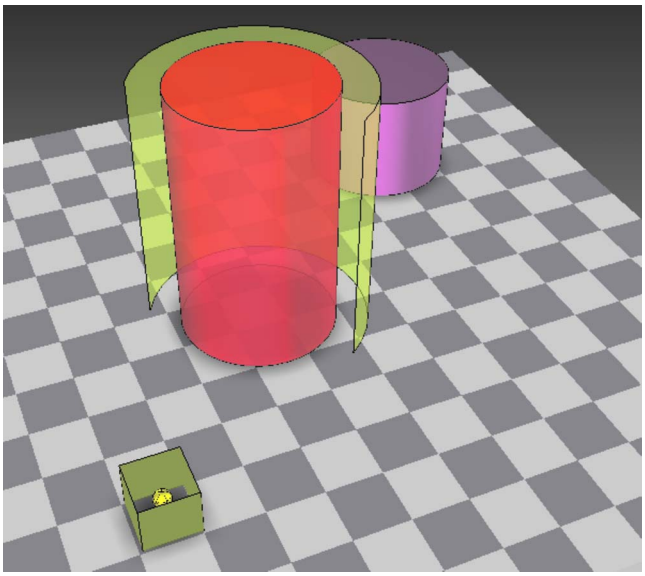

Fig. 1 Occluder shrinking: if the tested object (the rear cylinder) is occluded by the shrunk version of the occluder (the inner front cylinder) with respect to the center of the cube, then it is also occluded by the occluder itself (the outer front cylinder) if viewed from any point within the view-cell (the small cube). This facilitates the determination of the occluded regions for each view-cell.

We determine the occluded regions in the scene as a preprocessing step.[18] The comparison of our work with the state-of-the-art is summarized in Table 1. Here, we particularly focus on stereoscopic visualization of large urban models using the slice-wise representation. We show how the slice-wise representation perfectly fits the graphics hardware architecture; the GPU can be used, allowing faster frame rates for stereoscopic visualization.

2.2 *Stereoscopic Visualization*

Stereoscopic visualization is used in many applications such as simulators and scientific visualizations. It uses spe-

Table 1 The comparison of our approach for occlusion culling with the state-of-the-art.

Property	Previous work	Our approach
Object-space approach	Refs. 2–4	
Image-space approach	Refs. 6–11	√
On-line occlusion culling	Ref. 12	
Off-line occlusion culling	Ref. 15	√
Simplification incorporated	Refs. 13 and 14	
Occluder Shrinking	Ref. 16	√

cifically designed hardware—four frame buffers for the stereoscopic display. One of the most commonly used pieces of hardware is the time-multiplexed display system that is supported by liquid crystal shutter (LCS) glasses and virtual reality (VR) gears. Detailed information about these systems can be found in Refs. 19 and 20.

Stereoscopic viewing requires a display technique that allows each eye see the image generated for it. Most of the applications support stereoscopic display by generating the two images for the left and right eyes completely separately. The application must be able to generate 50 or more images per second to achieve a frame rate that approximates the same real-time visualization as the monoscopic correspondent.[21] Obviously, when a monoscopic application is converted to stereo without any improvement, the frame rate decreases by half.

Earlier works on speeding up stereoscopic rendering generally utilize the mathematical characterizations of an image. These works make use of the invariant characteristics of the image when the eye-point shifts horizontally as in a typical stereo application, such as the scan lines toward which an object projects.[19] In Ref. 22, the authors present a sterescopic ray-tracing algorithm that infers a right-eye view from a fully ray-traced left-eye view, which is further improved in Ref. 23. In Ref. 24, a non-ray-tracing algorithm is described that speeds up second-eye image generation in the processes of polygon filling, hidden surface elimination, and clipping. Methods that take advantage of the coherence between the two halves of a stereo pair for ray-traced volume rendering are discussed in Ref. 25. In Ref. 26, the authors present an algorithm using segment composition and linearly interpolated reprojection for fast direct volume rendering. Hubbold et al.[27] propose extending a direct volume renderer for use with an autostereoscopic display in radiotherapy planning. In Ref. 21, the authors present a framework to speed up stereoscopic visualization of terrains represented as height fields by generating the view for one eye from the other with some modifications; this speeds the process by approximately 45%, as compared to generating two eye-views separately from scratch. Mansa et al. provide an extensive analysis of coherence strategies that can be utilized for stereo occlusion culling.[28]

3 Slice-Wise Representation of Buildings

In Ref. 18, the slice-wise representation of buildings is discussed in detail. Here we give a brief summary of slice-wise representation and the usage of it in an urban visualization system for completeness. The slice-wise representation is based on the observation that the visible parts of the buildings in a typical urban walkthrough are mostly in one of the following three cases (see Fig. 2):

- The visible section is an *L-shaped* one with different orientations.
- The visible section is a *vertical rectangular* block, from the left or right of the building if the occluder perspectively seems taller than the occludee.
- The visible section can be seen as a *horizontal rectangular* block.

A significant feature of this representation is that it facilitates the storage of partial visibility in case a building is partially visible for a viewpoint. The slice-wise representation of buildings can facilitate the visualization of urban environments in an urban visualization system. The visualization framework utilizing this representation is shown in Fig. 3.

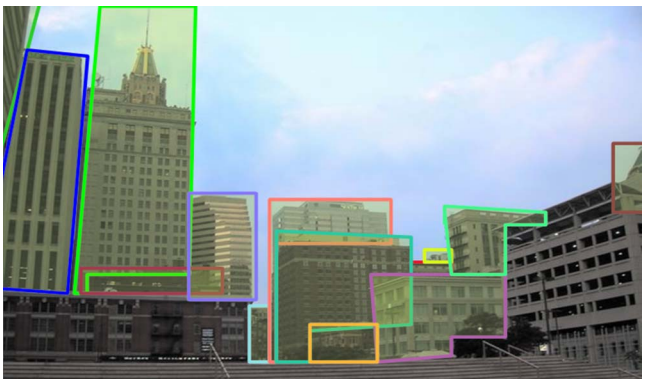

Fig. 2 The visibility forms that can be experienced during a typical urban walkthrough.

In the first phase, the scene data is read and converted to a temporary data structure having enough information for the internal processes. Next, a uniform subdivision is applied, and the cells are clustered into slices. The navigable area for the user is divided into view-cells. Then, the visibility determination using occluder shrinking is performed. The shrunk versions of occluders are constructed using the Minkowski differences of the occluders and the view-cells in object-space. The occlusion determination takes place

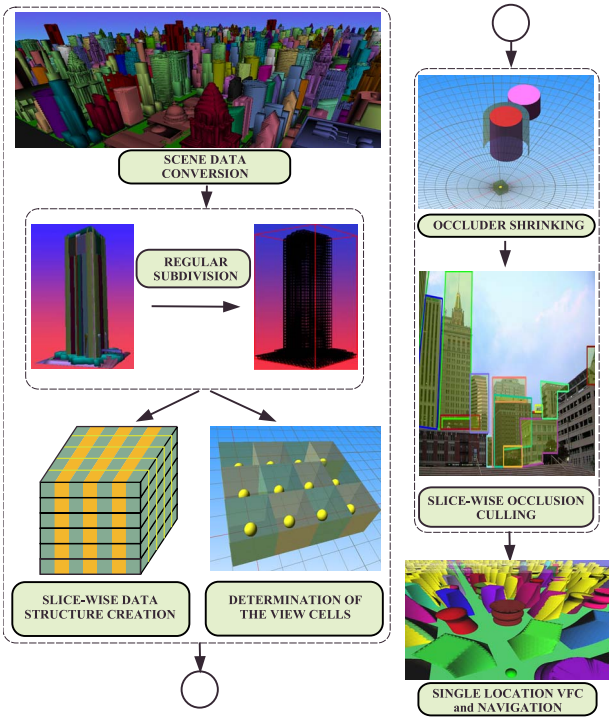

Fig. 3 The flow diagram of a visualization system using the slice-wise representation. The phases in dashed blocks are performed in the preprocessing phase.

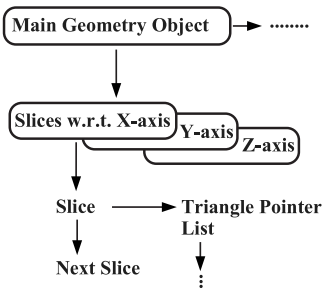

Fig. 4 The data structure for the slice-wise representation.

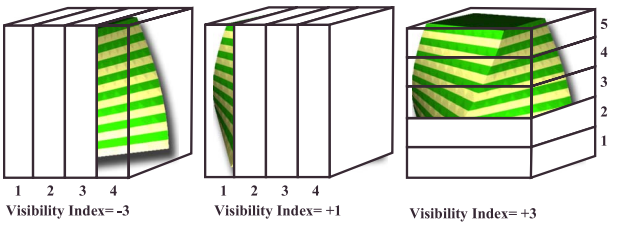

Fig. 5 Visibility index determination using the slice-wise representation: The index number to be stored depends upon the occluded section of the object. + or − signs are used to define the occlusion side.

after this step using the slice-wise representation, and partial visibility information is determined throughout the urban model for each view-cell.

The slice-wise representation is constructed by applying a regular subdivision to a building and then combining these subdivided blocks into slices for each axis. For each building, a separate list of slices is maintained. Since the slices are formed for each axis, a triangle of a building can be accessed by any of them (see Fig. 4).

In order to achieve conservative visibility by sampling the visibility from discrete locations, the occluders have to be shrunk by the maximum distance that can be traveled in view-cells. It is necessary to shrink the possible occluders so that the objects behind the occluder become visible and are added to the visibility list in case the user moves to the farthest available location in the view-cell. The shrinking is performed using Minkowski differences as described in Ref. 18.

In order to determine the occlusion for a building, that building is drawn in its original size, and other buildings are drawn in their shrunk versions. Hardware occlusion queries are used to determine the portions that are visible with respect to the center of each view-cell, i.e., square blocks on the ground. To speed up the process, several techniques such as quadtree-based culling and building-level culling are used in order to cull large portions before entering the slice-wise tests.

During the finest grained occlusion culling phase—the slice-wise occlusion culling step—the slices—not individual triangles—are tested for occlusion. A building is tested for occlusion using the shrunk versions of other objects as occluders and the slices of buildings parallel to each axis as occludees. The vertical slices are tested by gradually increasing their height, and the first visible heights are recorded for each. The horizontal slices are checked for complete occlusion. After determining the slices and portions of each building that are visible, the resultant list is optimized, and partial visibility is represented with only 3 bytes, one for each axis. As a result, visibility becomes encoded by the first visible slice numbers of vertical and horizontal axes (see Fig. 5). For the sake of simplicity, 3 bytes are stored for each building, including the unused axis. A separate visibility list is maintained for each navigable view-cell.

The rendering method employed in Ref. 18 uses dynamic display list compilation in OpenGL. This can cause bottlenecks if there is a large amount of visible geometry. To reduce this, the authors construct display lists on-line for nine view-cells, including the neighbors of the user's view-cell. This approach provides a suitable environment for visualization and eliminates frame dips that may arise because of the compilation. In the worst case, this has the disadvantage of replicating display lists of the buildings with little visual differences for all neighboring view-cells, which may lead to memory overflows.

4 Stereoscopic Urban Visualization Framework

In this section, we first explain how we use the GPU and the slice-wise representation for the monoscopic case. GPU utilization is based on the memory configuration for the vertices of the buildings. During visualization, we use only the indices for the vertices, which denote the locations of the vertices of the slices for partially visible and completely visible buildings.

4.1 *Using Slice-Wise Representation on the GPU*

GPU usage is becoming commonplace, not only in rendering but also in performing tasks such as collision detection,[29] database sorting,[30] and others.[31] Our aim is not to develop a new GPU-based algorithm, but to optimize the rendering of the scene using slice-wise representation for buildings.

Using slice-wise representation, it is possible to access any triangle by three orthogonal axes slices. In order to use this representation with the display list mechanism, the triangles pointed by each axis have to be compiled in the memory as display lists with different identifiers. Usually, this pointer duplication wastes memory, because a linked list of slices and their triangles must be maintained, (see Fig. 4). This is an undesirable property. However, if there were a way to represent this accessibility in some other terms, it would be very handy and would permit the visualization of larger urban models. This is what we achieve by using the GPU architecture, the buffer objects stored in the GPU.

4.1.1 *OpenGL: vertex buffer objects (VBOs)*

OpenGL provides a mechanism for the client-server type execution of the graphics commands. For a single machine, the server side is the graphics card (GPU), and the client side is the CPU. When a drawing command is issued, the data moves back and forth between the graphics card and the CPU. At this point, a vertex buffer object (VBO) becomes a powerful feature allowing the storage of the data in the GPU and eliminates the movement of the data to be drawn between the graphics memory and main system

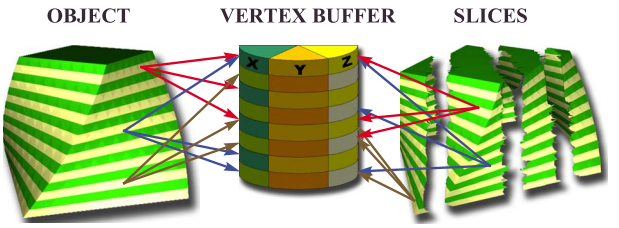

Fig. 6 The VBO data structure used in GPU-based visualization. The object triangles are constructed using the index buffers created in the GPU and accessed as needed for each building and for each slice.

memory.[32] With VBOs, the vertices are stored in a memory-efficient fashion in the GPU, and the data becomes encapsulated in storage schemes called "buffer objects." If the available graphics card memory is not sufficient, it can automatically swap with the main memory. In order to use a VBO, only a pointer to the actual encapsulated data in the GPU needs to be accessed by the CPU. This is a pointer to the memory location in the GPU that is used as a *buffer*, and it will be called a *binding pointer* throughout this paper.

4.1.2 *VBO creation for the buildings*

Our VBO configuration is shown in Fig. 6. The *vertex buffer* is filled with the x, y, and z vertex coordinates for each building. A second buffer, the *index buffer*, is created for each building which stores the indices of the vertices for each triangle. This index buffer is used to represent completely visible buildings during navigation. Next, other index buffers are created for each slice so as to represent partial visibility. It should be noted that the index buffers required for each slice can be constructed during walkthrough by storing the indices in main memory. The triangles and vertices in the memory are not needed after the VBOs for a building are constructed and stored in the GPU.

Figure 7 gives the VBO creation algorithm. In the first part of the algorithm, vertex coordinates, normals, and color data are sent to the GPU. These data will be used once with the rendering commands for the buildings, regardless of their visibility class. In the second part, the vertex index data for the triangles of a completely visible object are sent. Next, the same kind of data is sent for the slices. In the last part, the vertices, triangles, and other related data are deleted from the main memory through linked lists. To implement this algorithm, the data structure shown in Fig. 4 must be modified slightly to include binding pointers for complete visibility and for the slices for partial visibility (see Fig. 8).

4.1.3 *Implications of using VBOs for slices*

The slice-wise representation coupled with VBO provides a suitable environment for visualization, because the only memory overhead of this representation is the index buffers that are needed. It has several benefits: it supports partial visibility; it provides the lowest potentially visible set storage cost; and it facilitates a fast visualization environment.

As a result, the storage and accessibility representation of each slice is fully utilized, although the amount of GPU memory may cause slight limitation on this issue. However,

```
foreach Object in the model do
    /* 1st part */
    Generate a VBO Array Buffer for the vertices
    Vertex_List_Binding<--VBO_vertex_buffer
    Send vertices in the main memory to GPU through Vertex_List_Binding
    Generate a VBO Array Buffer for the normals of the triangles
    Normal_List_Binding<-- VBO_normal_buffer
    Send normals in the main memory to GPU through Normal_List_Binding
    Generate a VBO Array Buffer for the colors of the triangles
    Color_List_Binding<-- VBO_color_buffer
    Send colors in the main memory to GPU through Color_List_Binding
    /* 2nd part */
    Generate a VBO Element Array Buffer for index list of triangles
    CV_Element_Buffer_Binding<-- VBO_index_buffer
    Send array of indices to GPU as Element Array for the whole building
    forall Slices of the building in X, Y and Z axes do
        Generate a VBO Element Array Buffer for index list of triangles
        Slice_Element_Buffer_Binding<-- VBO_index_buffer
        Send array of indices to GPU as Element Array for the slice
    /* 3rd part */
    Delete vertices, triangles, normals and color data from the main memory
```

Fig. 7 The VBO creation algorithm. This algorithm is used to send the vertex coordinates, normals, and color data along with the vertex indices of the triangles to the GPU. In the first part, the necessary information for the vertices is sent. In the second part, we send the indices of the vertices for the triangles of a completely visible object and its slices. In the last part, these data are deleted from the main memory after they are transferred to the GPU.

VBOs have the advantage of being able to swap with the main memory, if the GPU memory becomes full. We have performed tests even with 32 MB of GPU memory–there were no memory overflows, and it automatically performs swapping with the main memory without causing notice-

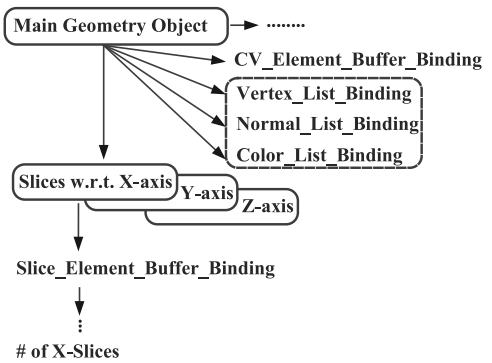

Fig. 8 The modified data structure for slice-wise representation to facilitate GPU implementation: the vertex, normal and color list bindings point to their memory locations in the GPU. These data are referenced by the element buffer bindings (*CV_Element_Buffer_Binding* and *Slice_Element_Buffer_Binding*) depending on visibility status during run-time.

```
Determine the active view-cells where the user eyes are located;
Apply Single Location VFC;
forall the objects attached to the active view-cells that are in the frustum do
    Get X, Y and Z indices;
    BindObject();
    if Y≡1 then
        /*object is completely visible */
        draw_CV_object(object);
    else
        foreach axes X, Y and Z do
            if slice_index≡0 then
                loop;
            pass_the_slice←—0;
            if slice_index<0 then
                /*Right part is visible */
                while slices are not finished do
                    pass_the_slice←—pass_the_slice+1;
                    if pass_the_slice>abs(slice_index) then
                        draw_slice();
            else
                /*Left part is visible */
                while slices are not finished do
                    pass_the_slice←—pass_the_slice+1;
                    if pass_the_slice≤slice_index then
                        draw_slice();
```

Fig. 9 The algorithm for selecting the slices to be rendered. The selection is performed based on the visibility index assigned to the slice as described in Ref. 18. The BindObject( ) function is used to inform the GPU that the object is to be accessed for rendering.

able frame dips. The representation of each slice does not need to be changed. However, instead of keeping display lists and triangles in the main memory, they are kept in the high-speed memory of the graphics hardware. This produces a huge decrease in the amount of main memory used because of the driver optimization of OpenGL. Figures 6 and 8 show the resultant configuration and the memory-resident structures for GPU-based visualization using the slice-wise representation.

4.1.4 *VBO referencing during run-time*

Run-time VBO access is depicted in Fig. 9 In this algorithm, the slice-wise representation of buildings is exploited. This algorithm uses the visibility information, which is produced using the occlusion culling algorithm and the slice-wise representation. In this algorithm, the following operations are performed:

1. First, the active view-cell (or view-cells, since two eyes may be in two different cells) are determined by looking at the user location in the navigable space.[15] Visible objects are determined and stored as a linked list for each view-cell.
2. Next, this list is traversed and any completely visible objects are rendered using the *CV_Element_Buffer_Binding* index of the object. If the object is partially visible, then we traverse the slices of the object. The occlusion can be either on the left or right of the vertical axes or in the lower part of the object (see Fig. 5).
3. If the object is occluded from the left and the right part is visible, which is denoted by a negative visibility index, we increment the variable and do not render the slices. We just skip the slices until the incremented variable becomes greater than the absolute value of the visibility index. Then, we send the *Slice_Element_Buffer_Binding* indices of the visible slices for rendering.
4. If the object is occluded from the right and the left part is visible, which is denoted by a positive visibility index, we render the slices until the incremented variable becomes greater than the visibility index.

4.2 *Stereoscopic Rendering*

The following conditions are required to achieve the best performance in stereoscopic visualization:

- The rendering rate should be sufficient to achieve interactive visualization, i.e., it should be at least 17 frames per second.
- The ghosting effect (cross talk), which is caused by drawing a geometry for one eye and not drawing it for the other eye, should be reduced or eliminated.
- The strongest stereo effect with the lowest values of parallax should be provided. Parallax values should not exceed 1.6 deg.[33]

The main problems incurred with stereoscopic visualization include the ghosting effect and the resultant eye disturbance problems. The ghosting effect, or cross talk, is the faded image seen by the untargeted eye. This effect is undesirable because it may cause eye fatigue and other visualization problems. The main causes of the ghosting effect or cross talk stated in the literature are the late decaying of the phosphor and shutter leakage.[34–37] The phosphor persistence causes a faded image to be seen when the image for the other eye is being displayed on the screen.[38] Most of the research in this area is devoted to reducing this disturbing effect. This effect is experienced particularly when the background is dark and the image just drawn has high-intensity colors.

4.2.1 *Stereoscopic projection method*

We applied off-axis projection with parallel frustums (Fig. 10) for stereoscopic visualization, i.e., two projections are performed for each viewing direction and for each eye and converge at infinity. Since an urban scene contains many buildings at a distance, we found that using off-axis projection with a single convergence point (*toe-in projection*) causes a lot of ghosting effects on the screen (see Fig. 10). Because of the convergence angle and varying scene depth,

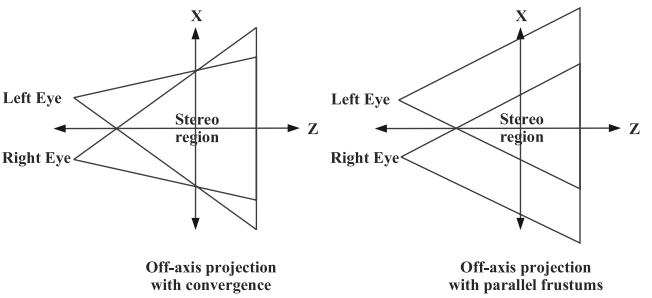

Fig. 10 Off-axis projection using convergence is shown on the left. If the user converges to the assumed location in the scene, then perfect stereo is achieved. However, for urban scenes where there are lots of buildings, assuming a single convergence point is not realistic. On the right, off-axis projection with parallel view frustums is shown. Converging viewing directions at infinity decreases the ghosting effect if the viewing parameters are kept within reasonable limits.

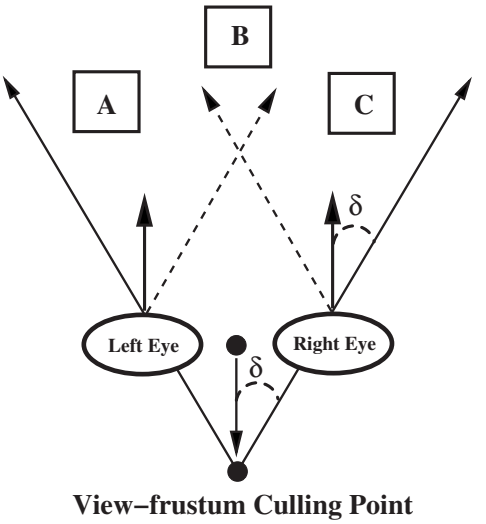

Fig. 11 Changing the VFC location: since we know the projection angle, the exact distance to move backward becomes a simple function of half of the eye separation distance and half of the projection angle [*backward_distance*=*half_interocular_distance*/tan(δ)]. By moving the VFC location, a single test can cover all the volume that can be viewed in stereo.

locations other than the convergence point can have noticeable ghosting effect, even when the viewing parameters are kept within reasonable limits. In real life, the human eyes can converge easily to any point the viewer wants. In computer-generated stereo, it is not easy to determine the point where the user's eyes are converging; there has been some work in this area, but the results are not easily applicable.[39,40] Using a convergence point works better for observing a single object. Therefore, we choose to use off-axis projection with parallel view frustums converging at infinity. If the stereo parameters, such as interocular distance and user-screen distance, are kept within reasonable limits, the ghosting effect on the inner parts of the screen becomes unnoticeable. We do not use on-axis projection because it causes image distortions at the peripheries of the screen due to projection transformations.

4.2.2 *View-frustum culling*

View-frustum culling (VFC) is one of the most important methods of eliminating primitives that do not contribute to the final image during navigation. It is generally performed twice for stereoscopic visualization. We made a simple change to decrease the number of VFC operations for stereoscopic visualization from two to one. Instead of performing VFC according to the locations of the eyes, we move backward a calculated distance and put the culling location at the spot indicated in Fig. 11. This location is determined by using the midpoint of both eyes, the frustum angle, and the interocular distance. The viewing frustum becomes enlarged by moving the user position virtually backward, until the new frustum edges coincide with the right edge of the frustum with respect to the right eye and the left edge of the frustum with respect to the left eye. Thus, we are able to cover the whole region that can be observed during stereoscopic visualization. Although this single-location VFC increases the number of polygons to be processed for rendering, it is much less costly than performing VFC twice.

VFC can be performed on the unoccluded objects by making an in-order traversal of the scene quadtree. Another solution is to test the bounding boxes of each unoccluded object one by one. Our experiences show that when the scene quadtree subdivision depth is too high, it may take longer to cull the objects from the frustum than testing unoccluded objects one by one. Since the scene is large and the number of visible objects is much smaller than the number of quadtree nodes, for ground-based navigation, it is faster to test only the bounding boxes of individual buildings in urban scenes.

VFC can be done using stencil tests on the quadtree blocks of the unoccluded geometry. It can also be carried out by applying hardware occlusion queries for the quadtree blocks. If the scene hierarchy is to be used for the VFC operation, then the in-frustum information for each node of the hierarchy is needed, in order to determine the tests for deeper level nodes. However, this requires a hardware occlusion query setup and retrieval operation for each quadtree block, and the setup time for hardware occlusion culling is longer than the setup time for the stencil buffer mechanism. This is not the case for testing the bounding boxes of each object individually; all of the bounding boxes can be sent to the GPU in a single batch using hardware occlusion query, and the ones returning visible pixels can be quickly rendered. These options are scene dependent, and we have chosen to test the bounding boxes of the objects using hardware occlusion queries; we use an empty buffer as an occluder buffer and test the bounding boxes of each object individually.

5 Performance Study and Comparisons

The proposed framework is implemented using C language with OpenGL libraries. The test platform is an Intel Pentium IV, 3.4-GHz computer with 4 GB of RAM and a NVidia Quadro Pro FX 4400 graphics card with 512 MB of memory supporting the quad buffering needed for stereoscopic visualization. Crystal Eyes LCS glasses are used for viewing in stereo. The purpose of the empirical study is to test:

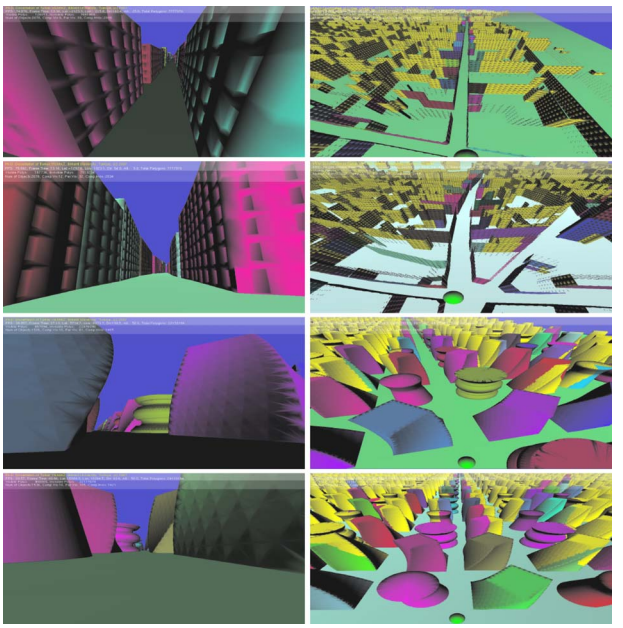

Fig. 12 Still frames from navigations through the Vienna2000 model (the first two rows) and the procedurally-generated model (the last two rows) in monoscopic view. On the left, still frames from a given viewpoint are shown. To the right of each frame, the view from above the user position represented by the small ellipsoid, shows the rendered buildings using occlusion culling based on the slice-wise representation. Invisible buildings are shown in yellow (faded out). (Color online only).

- whether single-location VFC brings an advantage over multiple VFC, given that the enlarged frustum may decrease performance because of containing more polygons;
- GPU performance with the slice-wise building representation.

We performed tests using both the Vienna2000 Model, which consists of 7.8 million polygons in 2,086 buildings, and a procedurally-generated city model composed of 23 million polygons in 1,536 buildings with six different architectures. Still frames from navigations through these models are shown in Fig. 12.

In Fig. 13, we compare the frame rates obtained using different VFC schemes. Our aim is not to test the advantage of VFC but to test the gain in performance from using single-location VFC instead of multiple-location VFC. However, we also give performances when VFC is not applied for the sake of completeness. The reason for the fluctuations in these graphs is the changing polygon counts as the navigation is carried out. Different parts of an urban model can be represented with different numbers of polygons, depending on the complexity of the buildings.

The average frame rates for the Vienna2000 Model are 281.8, 231.0, and 215.8 frames per second (fps) for the single-location, multiple-location, and no-frustum culling schemes, respectively. The average frame rates for the procedurally-generated model are 34.24, 30.5, and 10.2 frames per second (fps) for the single-location, multiple-location, and no-frustum culling schemes, respectively. The procedurally-generated model has long streets, which means that a lot of geometry is instantly visible in each frame. The culling ratios (including view-frustum culling and occlusion culling) are 98.53%, 98.53%, and 96.43% for the Vienna2000 Model and 97.00%, 97.00%, and 91.82% for the procedurally generated model for the single-location VFC, multiple-location VFC, and no-frustum culling schemes, respectively. Using single-location VFC with the Vienna2000 model produces a 22.0% gain in frame rates when compared to using multiple location VFC; for the procedurally-generated model, the gain is 12.3%.

The advantage of using a GPU-based rendering ap-

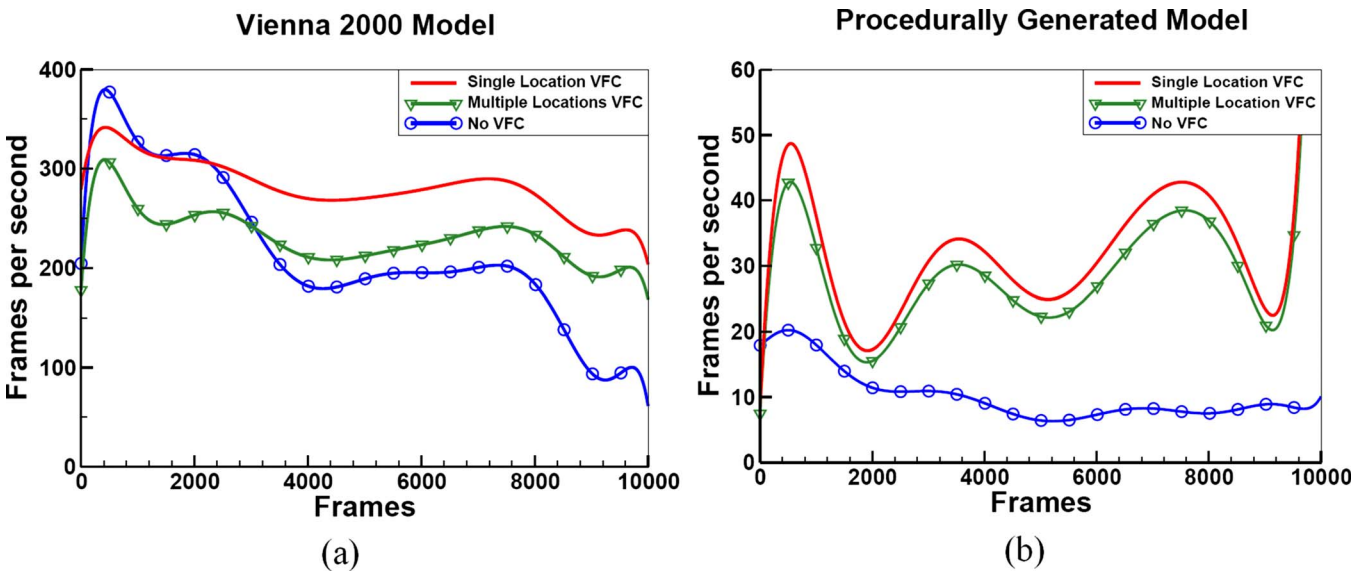

Fig. 13 Frame rate comparison of the VFC schemes in stereoscopic visualization: (a) frame rates for the Vienna2000 model with 7.8 million polygons. (b) frame rates for the procedurally-generated model with 23 million polygons. These graphs show the advantage of using single-location VFC with respect to multiple location VFC and not performing VFC. Note that we render two images for each frame.

Table 2 The summary of test results using the proposed framework.

Model name	Vienna2000	Procedurally-generated
Number of polygons	7.8 million	23 million
Number of buildings	2,086	1,536
Number of slices	94,480	30,392
Main memory usage	1.3 MB	425.5 KB
GPU memory usage	298 MB	904 MB
Single-location VFC (stereo)	281.8 fps	34.24 fps
Multiple-location VFC (stereo)	231.0 fps	30.5 fps
Using display lists (mono)	135.1 fps[a]	Not available
Speed-up using VBOs	315%	Not available
Speed-up of single-location VFC	22%	12.3%

[a]The reported average frame rate in Ref. 18 (using the same test platform).

proach with the slice-wise building representation can be examined in two aspects: rendering speed-up and memory usage. The reported average frame rate for the monoscopic rendering of the Vienna2000 Model using OpenGL display lists is 135.1 fps.[18] The frame rate for GPU-based stereoscopic rendering is 281 fps on average. Since we render two images for each frame, this corresponds to a 315% speed-up when compared to using OpenGL display lists. The reported main memory usage for the slice-wise representation of the Vienna2000 model is 218.7 MB. For the GPU-based approach, the main memory usage is only 1.3 MB (14 bytes per each of 94,480 slices). Thus, GPU-based rendering confers significant advantages both in terms of the rendering speed and main memory usage. Test results are summarized in Table 2.

6 Conclusion

In this paper, we propose a framework for the stereoscopic visualization of urban environments. We make use of an occlusion-culling approach based on a slice-wise building representation that can capture partial visibility. The stereoscopic visualization framework uses a GPU-based rendering method that exploits slice-wise representation. The framework also uses a modified view-frustum culling approach, in which only one culling is performed. The resultant view-frustum has the same coverage as the view-frustums for each eye in stereoscopic visualization.

The visualization is done using off-axis stereoscopic projection with parallel frustums. The framework is tested on large urban models: the Vienna2000, which is a real-world model containing 7.8 million and a procedurally-generated model containing 23 million polygons. The empirical study shows that using the single-location VFC brings a significant gain in frame rates when compared to using multiple-location VFC. The GPU-based rendering of the urban model using the slice-wise representation is significantly faster than the one using OpenGL display lists. This shows that the slice-wise representation fits perfectly onto the GPU architecture by the use of vertex buffer objects. This study shows that the proposed framework allows a real-time stereoscopic visualization of urban scenes.

Acknowledgments

The work described in this paper is supported by the Scientific and Research Council of Turkey (TÜBİTAK) under Project Codes 104E029 and 105E065. The Vienna2000 Model is courtesy of Peter Wonka and Michael Wimmer. We are grateful to Kirsten Ward for proofreading and suggestions.

References

1. D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *IEEE Trans. Vis. Comput. Graph.* **9**(3), 412–431 (2003).
2. J. Heo, J. Kim, and K. Wohn, "Conservative visibility preprocessing for walkthroughs of complex urban scenes," in *Proc. ACM Symposium on Virtual Reality Software and Technology*, pp. 115–128, ACM Press/Addison-Wesley (2000).
3. J. T. Klosowski and C. T. Silva, "Efficient conservative visibility culling using the prioritized-layered projection algorithm," *IEEE Trans. Vis. Comput. Graph.* **7**(4), 365–379 (2001).
4. G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion, "Conservative volumetric visibility with occluder fusion," in *Proc. SIGGRAPH*, pp. 229–238, ACM Press/Addison-Wesley (2000).
5. T. A. Funkhouser, C. H. Sequin, and S. J. Teller, "Management of large amounts of data in interactive building walkthroughs," *ACM Computer Graphics (Proc. ACM Symposium on Interactive 3D Graphics)* **25**(2), 11–20 (1992).
6. D. Bartz, M. Meißner, and T. Hüttner, "OpenGL-assisted occlusion culling for large polygonal models," *Comput. & Graphics* **23**(5), 667–679 (1999).
7. B. Chen, J. E. Swan, E. Kuo, and A. E. Kaufman, "LOD-sprite technique for accelerated terrain rendering," in *Proc. IEEE Visualization*, pp. 291–298 (1999).
8. N. Greene, "Efficient occlusion culling for Z-buffer systems," in *Proc. Computer Graphics International*, pp. 78 (1999).
9. M. Wimmer, M. Giegl, and D. Schmalstieg, "Fast walkthroughs with image caches and ray casting," *Comput. & Graphics* **23**(6), 831–838 (1999).
10. F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative visibility preprocessing using extended projections," in *Proc. SIGGRAPH*, pp. 239–248, ACM Press/Addison-Wesley (2000).
11. M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer, "The randomized z-buffer algorithm: interactive rendering of highly complex scenes," in *Proc. SIGGRAPH*, pp. 361–370, ACM Press/Addison-Wesley (2001).
12. P. Wonka, M. Wimmer, and F. X. Sillion, "Instant visibility," *Computer Graphics Forum (Proc. Eurographics)* **20**(3), 411–421 (2001).
13. C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet, "Integrating occlusion culling and levels of detail through hardly visible sets," *Comput. Graph. Forum* **19**(3), 499–506 (2000).
14. J. A. El-Sana, N. Sokolovsky, and C. T. Silva, "Integrating occlusion culling with view-dependent rendering," in *Proc. IEEE Visualization*, pp. 371–378 (2001).
15. T. Yılmaz and U. Güdükbay, "Extraction of 3D navigation space in virtual urban environments," in *Proc. 13th European Signal Processing Conference*, EURASIP (2005).
16. X. Decoret, G. Debunne, and F. Sillion, "Erosion based visibility preprocessing," in *Proc. 14th Eurographics Workshop on Rendering*, P. Christensen and D. Cohen-Or, Eds., pp. 281–288 (2003).
17. P. K. Agarwal and M. Sharir, "Arrangements," in *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds., pp. 49–119, Elsevier, North-Holland, Amsterdam, (1999).
18. T. Yılmaz and U. Güdükbay, "Conservative occlusion culling for urban visualization using a slice-wise data structure," *Graphical Models* **69**(3–4), 191–210 (2007).
19. L. F. Hodges, "Tutorial: time-multiplexed stereoscopic computer graphics," *IEEE Comput. Graphics Appl.* **12**(2), 20–30 (1992).
20. L. F. Hodges and D. McAllister, "Stereo and alternating-pair techniques for display of computer-generated images," *IEEE Comput. Graphics Appl.* **5**(9), 38–45 (1985).
21. U. Güdükbay and T. Yılmaz, "Stereoscopic view-dependent visualization of terrain height fields," *IEEE Trans. Vis. Comput. Graph.* **8**(4), 330–345 (2002).

22. J. D. Ezell and L. F. Hodges, "Some preliminary results on using spatial locality to speed up raytracing of stereoscopic images," in *Stereoscopic Displays and Applications I*, *Proc. SPIE* **1256**, 298–306 (1990).
23. S. J. Adelson and L. F. Hodges, "Stereoscopic ray-tracing," *Visual Comput.* **10**(3), 127–144 (1993).
24. S. J. Adelson, J. B. Bentley, I. S. Chong, L. F. Hodges, and J. Winograd, "Simultaneous generation of stereoscopic views," *Comput. Graph. Forum* **10**(1), 3–10 (1991).
25. S. J. Adelson and C. D. Hansen, "Fast stereoscopic images with ray traced volume rendering," in *Proc. Symposium on Volume Visualization*, pp. 3–9, ACM Press (1994).
26. T. He and A. Kaufman, "Fast stereo volume rendering," in *Proc. IEEE Visualization*, pp. 49–56 (1996).
27. R. Hubbold, D. Hancock, and C. Moore, "Stereoscopic volume rendering," in *Proc. Visualization in Scientific Computing*, pp. 105–115 (1998).
28. I. Mansa, A. Amundarain, L. Matey, and A. Garcia-Alonso, "Analysis of coherence strategies for stereo occlusion culling," *J. Comput. Animation Virtual Worlds* **19**(1), 67–77 (2008).
29. N. K. Govindaraju, M. C. Lin, and D. Manocha, "Fast and reliable collision culling using graphics hardware," *IEEE Trans. Vis. Comput. Graph.* **12**(2), 143–154 (2006).
30. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 325–336 (2006).
31. A. Lefohn, "Glift: an abstraction for generic, efficient GPU data structures," in *GPGPU: General-Purpose Computation on Graphics Hardware*, ACM SIGGRAPH Course Notes, pp. 140–151, ACM Press, New York (2005).
32. nVidia Corp. "Using vertex buffer objects (VBOs)," pp. 1–15 (2003).
33. N. A. Valyus, *Stereoscopy*, Focal Press, London and New York (1962).
34. T. Haven, "A liquid-crystal video stereoscope with high extinction ratios, a 28% transmission state, and 100 μs switching," *Proc. SPIE* **761**, 23–26 (1987).
35. L. Lipton, J. Halnon, J. Wuopio, and B. Dorworth, "Eliminating π-cell artifacts," *Proc. SPIE* **3957**, 264–270 (2000).
36. J. Lipscomb and W. Wooten, "Reducing crosstalk between stereoscopic views," *Proc. SPIE* **2177**, 92–96 (1994).
37. P. Bos, "Time sequential stereoscopic displays: the contribution of phosphor persistence to the 'ghost' image intensity," in *Proc. Three-Dimensional Image Technologies (ITEC)*, pp. 603–606 (1991).
38. A. J. Woods and S. S. L. Tan, "Characterizing sources of ghosting in time-sequential stereoscopic video displays," *Proc. SPIE* **4660**, 66–77 (2003).
39. Z. Zhu and Q. Ji, "Robust real-time eye detection and tracking under variable lighting conditions and various face orientations," *Comput. Vis. Image Underst.* **98**(1), 124–154 (2005).
40. J.-G. Wang, E. Sung, and R. Venkateswarlu, "Estimating the eye gaze from one eye," *Comput. Vis. Image Underst.* **98**(1), 83–103 (2005).

Türker Yılmaz received his BSc degree in finance management from the Turkish Military Academy, Ankara, Turkey, in 1991. He finished the one-year automated data processing program at Middle East Technical University, Ankara, Turkey, in 1998. He then received his MSc and PhD degrees, both in computer engineering, from Bilkent University, Ankara, Turkey, in 2001 and 2007, respectively. Currently, he is an instructor at the Turkish Military Academy. His research interests include visualization of complex graphical environments, virtual reality, and simulation programming.

Uğur Güdükbay received his BSc degree in computer engineering from Middle East Technical University, Ankara, Turkey, in 1987. He received his MSc and PhD degrees, both in computer engineering and information science, from Bilkent University, Ankara, Turkey, in 1989 and 1994, respectively. He then conducted research as a postdoctoral fellow at the University of Pennsylvania, Human Modeling and Simulation Laboratory. Currently, he is an associate professor at Bilkent University, Department of Computer Engineering. His research interests include various aspects of computer graphics, including physically based modeling, human modeling and animation, and visualization of complex graphical environments. He is a senior member of IEEE and professional member of ACM.