Turk J Elec Engin, VOL.10, NO.1 2002, © TÜBİTAK

Walkthrough in Complex Environments at Interactive Rates using Level-of-Detail

Alper SELÇUK, Uğur GÜDÜKBAY, Bülent ÖZGÜÇ
Bilkent University
Department of Computer Engineering
06533 Bilkent Ankara Turkey
e-mail: alpers@microsoft.com,{gudukbay,ozguc}@cs.bilkent.edu.tr

Abstract

One of the biggest problems in computer graphics is displaying huge geometric models at interactive rates. A lot of work has been done to achieve the required frame-rates in architecture, simulation, computer-aided design and entertainment applications. In this paper, a system that enables walkthrough in complex environments using level-of-detail approximations is explained. The system uses hierarchical triangulated models as input. In the preprocessing phase, multiresolution models of objects are created using polygonal simplification techniques. During walkthrough, fast frustum culling based on bounding boxes is performed to eliminate branches of hierarchy that are not visible. An appropriate level for detail of objects is selected and displayed depending on the distance of the objects to the camera. For far nodes in the hierarchy, geometric data in lower levels is ignored and textured bounding boxes are displayed. The system achieves interactive frame rates for moderately complex models containing up to a million polygons.

Key Words: *level-of-detail, visibility culling, geometric simplification, rendering, walkthrough, frame rate.*

1. Introduction

In recent years, the use of computers in design has become extremely important. Although computer-aided design software helps engineers greatly in designing and viewing individual components, to verify the correctness of design, engineers need to view combined shots of their design. Even a walkthrough of the entire model may be required in some cases. Architects today use computers to design and view buildings. The size of such architectural models can be huge. Customers may want to view the building on computer. Showing rendered pictures or a film prepared by moving a virtual camera inside the building model on a predetermined path alone may not always satisfy a customers. Customer may want to walk inside the model interactively. In addition to this, walkthroughs in outdoor environments are also needed for applications, such as virtual tours in ancient sites for touristic purposes. Virtual models of ancient sites also can be very complex, containing millions of polygons. Generally, the same techniques can be applied for both types of walkthrough applications.

For such walkthroughs, image quality is the most important property. Image quality depends on texture quality, positioning of light sources and rendering quality. Real-time restrictions can be slightly loosened for the sake of image quality. Considering the size of an architectural model, the textures used for walls, floor and furniture and the number of light sources in a building, the power of today's computers are insufficient.

The problem for a walkthrough, a virtual tour by moving the camera, in a complex environment is that the number of polygons to be displayed exceeds the number that the computer can render for each frame. It is trivial that current computer graphics systems cannot meet the required graphics throughput for the above examples. The situation will probably not change in the near future because as graphics systems evolve and get more powerful, the sizes of models also grow larger and larger.

In order to decrease the load of graphics systems in complex tasks, we must consider the capabilities of human visual systems. Without any knowledge about this, graphics output requirements will most likely exceed the limits of current graphics systems. Since the human eye will view generated images, satisfying only the human visual system is enough to simulate reality in a virtual environment. Although the human visual system is complex, it is not perfect. We must consider the limitations of human visual system to achieve perceptual and visceral realism at low cost [1]. This can reduce the workload of graphics systems considerably and enable real-time walkthroughs in million-polygon models.

A system for a walkthrough in complex environments is explained in this paper. The system uses hierarchical geometric models as input. In the preprocessing phase, multiresolution models of an object in the scene are created using polygonal simplification techniques. During walkthrough, fast visibility and frustum culling based on bounding boxes is performed to eliminate objects that are not visible to the camera. An appropriate level of detail (LOD) of objects is selected and displayed depending on the distance of the object to the camera. In addition, textured bounding boxes are used for objects that are too far away from the camera to further reduce processing load. The system achieves acceptable frame rates for moderately complex models containing up to a million polygons.

The rest of the paper is organized as follows. In Section 2 different approaches for a walkthrough in complex environments are discussed. Details of the walkthrough system developed in this work are explained in Section 3. Section 4 presents the results produced by using the implementation. Conclusions and future research areas are given in Section 5.

2. Methods for a Walkthrough in Complex Environments

Methods that enable walkthroughs in complex environments vary from very simple ideas to very complex algorithms. Figure 1 gives a taxonomy of these methods. Algorithms of varying complexity can be used together to access the combined advantages of each other.

2.1. Reducing Geometric Complexity

The main idea of geometric complexity reduction is to get a realistic image without modeling and rendering all the scene.

2.1.1. Polygonal Simplification

Polygonal simplification is the process of transforming a three-dimensional (3D) polygonal model into a simpler version containing less polygons. The transformation tries not to change the original shape and

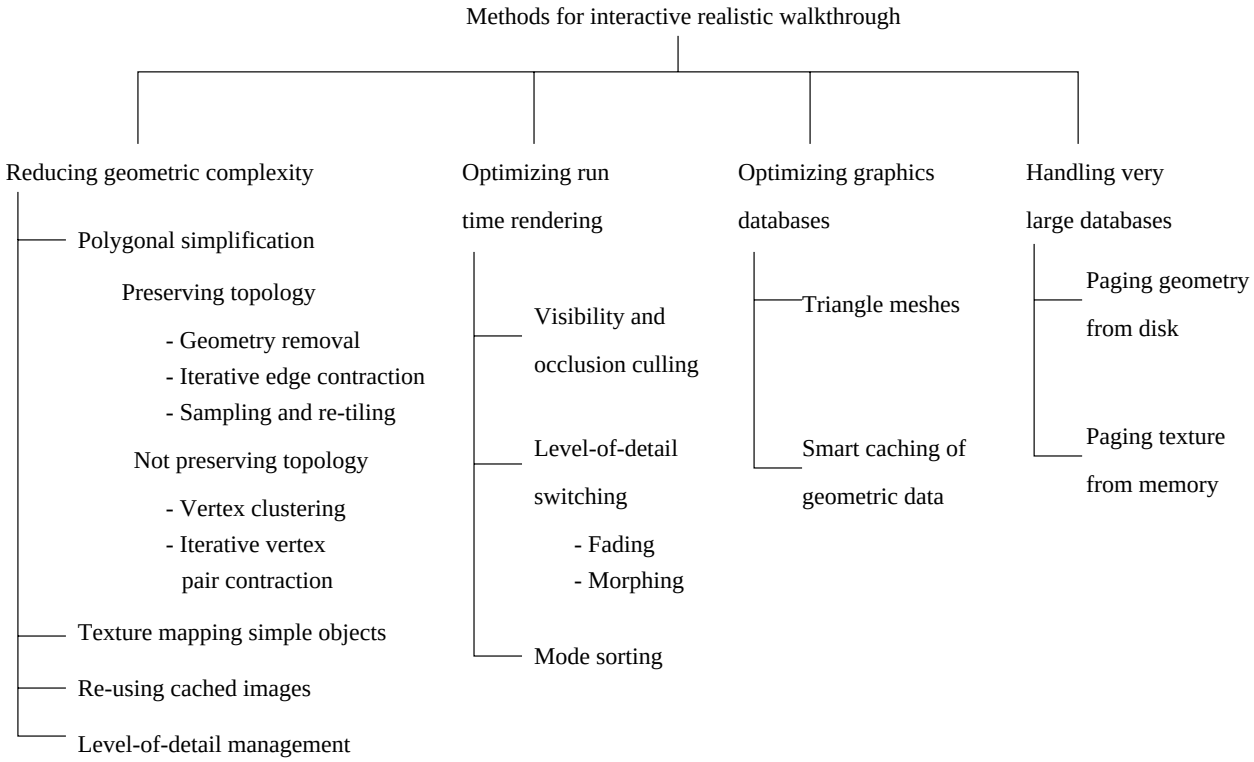

Figure 1. A taxonomy of methods for a walkthrough in complex environments

appearance of the model. In this way, rendering load will be reduced considerably. Storage requirements will also be reduced, simplifying the management of data between disk and memory. Furthermore, the transmission of simplified large models over networks will be faster than original models. If the simplification is good enough, then each stage of the graphics pipeline will have a workload that can be handled in real time.

Polygonal simplification algorithms are categorized into two groups: algorithms that preserve the topology of the original model and algorithms that do not preserve the topology of the original model. There are four basic methods for simplification. These are *geometry removal* [2, 3], *vertex clustering* [4], *iterative edge (or vertex-pair) contraction* [5, 6, 7], and *sampling and re-tiling* [8].

2.1.2. Geometric Level-of-Detail (LOD) Management

The main idea of the geometric LOD management technique is representing objects that do not contribute much to the scene with less primitives. Polygonal simplification algorithms can be used to generate multiresolution models automatically. One important requirement for multiresolution models is the preservation of the appearance of objects. There are measures for determining the success of a LOD management algorithm. Measures for image output are more important than measures for topology or geometry. Therefore, an image-based error metric is generally used.

Some examples of LOD management algorithms are given in [9, 10, 11, 12]. The LOD management algorithms presented in [12, 10] are dynamic algorithms in the sense that they simplify the geometric models on-the-fly in a view-dependent manner while making a walkthrough. However, such algorithms are difficult to implement. Static approaches, like the one used in our implementation, use a number of LOD representations

of the models obtained through a preprocessing step. Since the simplification is done in a preprocessing step, the implementation of the walkthrough algorithms are easier and higher frame rates can be obtained. However, popping artifacts may be severer than dynamic approaches when switching between different LOD representations in static approaches. This is due to the fact that dynamic approaches may simplify different parts of a model in different proportions according to the view position.

Hierarchical data structures for storing LOD are proposed in [13, 14]. In these techniques, the intermediate nodes of the hierarchy contain simplified data of its children. Leaf nodes store original data. A sample hierarchy is given in Figure 2. In [13], the representation of a scene is selected according to the distance of the scene to the viewpoint and the area that the scene covers on the screen. In [14], different search strategies, such as depth-first search, and best-first search are proposed for selecting the representation.

Another algorithm makes use of frame-to-frame coherence by caching images of objects rendered in one frame for possible reuse [15]. The algorithm uses a Binary Space Partitioning tree to store geometric primitives and cached images of objects.

2.1.3. Texture Mapping

The algorithms explained so far use geometric simplifications of the original model. Replacing the original model with a texture or a colored cube is another method. A simple polygonal object with a texture can be used instead of a complex polygonal model if it is in a distant position on the screen. In such methods, geometric simplification cost is eliminated [16].

2.2. Optimizing Run-Time Rendering

To optimize the run-time rendering in a walkthrough application, visibility and occlusion culling techniques are used to reduce the number of objects to be rendered for each frame [17, 18, 19]. To eliminate visual defects caused by switching between different representations of objects, *fading* and *morphing* techniques are used. Grouping the polygons in terms of texture and material properties also increases run-time rendering performance.

2.3. Optimizing Graphics Databases

Geometric data should be organized in such a way that graphics processors should be able to manipulate the geometric data efficiently. Triangulating the model as a preprocessing step reduces the rendering time of the scene. In addition the arrangement of data in memory should reduce the delays during rendering. For example, data can be arranged in contiguous memory locations. Also, secondary processors for pre-fetching data can be used to reduce delays caused by page faults.

2.4. Handling Very Large Databases

Another problem with complex models is that the model can be too large to fit into the memory. This problem has two components. The first one is that the size of the model exceeds the RAM and the second one is that the size of the texture does not fit into the texture memory. When paging geometric data from disk, full I/O bandwidth should be used. This requires structuring the data so that it can be read in large blocks; preferably being transferred using Direct Memory Access into the application's address space. Paging operation should not affect the frame rate. Therefore, it can be performed synchronously between

frames. However, the amount of data being transferred between frames can be too small to utilize full I/O bandwidth. To avoid this problem in multi-threaded systems, asynchronous loading can be performed by creating a thread for load operation.

For achieving fast texture mapping, graphics subsystems have their own texture memories that are used for caching textures. Texture memories are much more expensive than conventional RAM and maximum available texture memory size is much less than RAM. Therefore, the management of texture memory is critical. Unlike paging data from disk, paging texture cannot be done asynchronously with rendering because on most graphics architectures texture loading shares the same data paths as normal rendering. A fraction of the rendering time must be reserved for texture loading. Memory management issues, such as fragmentation, should also be considered. The simplest solution to this problem is to load textures of the same size.

3. The Walkthrough System

In the preprocessing phase, the system builds a hierarchy of the scene and generates simplified versions of the objects in the scene (Figure 2). The simplification algorithm is based on removing nearly coplanar triangles from the objects. A number of simplified versions can be produced by the simplification algorithm each with different LODs. In the walkthrough phase, a virtual camera is moved inside the model. As the camera moves inside the model, the appropriate version of the objects is selected and displayed. The block diagram of the system is given in Figure 3.

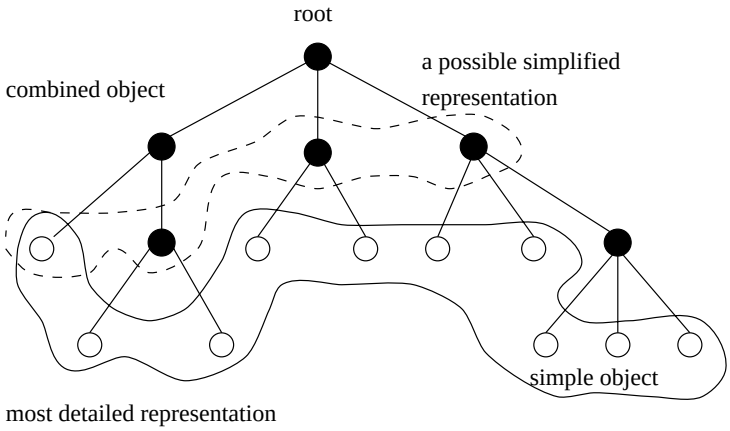

Figure 2. A sample hierarchy

3.1. Input Model

For the system, the triangle representation is selected for the models. The geometric model should have a hierarchical structure. Objects should consist of other objects and only the primitives at the lowest level should have geometric data. Hierarchical structures are very suitable for simplification and management purposes.

Object Separated Triangle Format (OSTF) [20] is selected for importing models from CAD/CAM software. OSTF meets all the requirements explained above. It stores objects in a hierarchical structure. Triangle representation is used at the lowest level of the hierarchy. Each triangle has its vertex, normal and texture data.

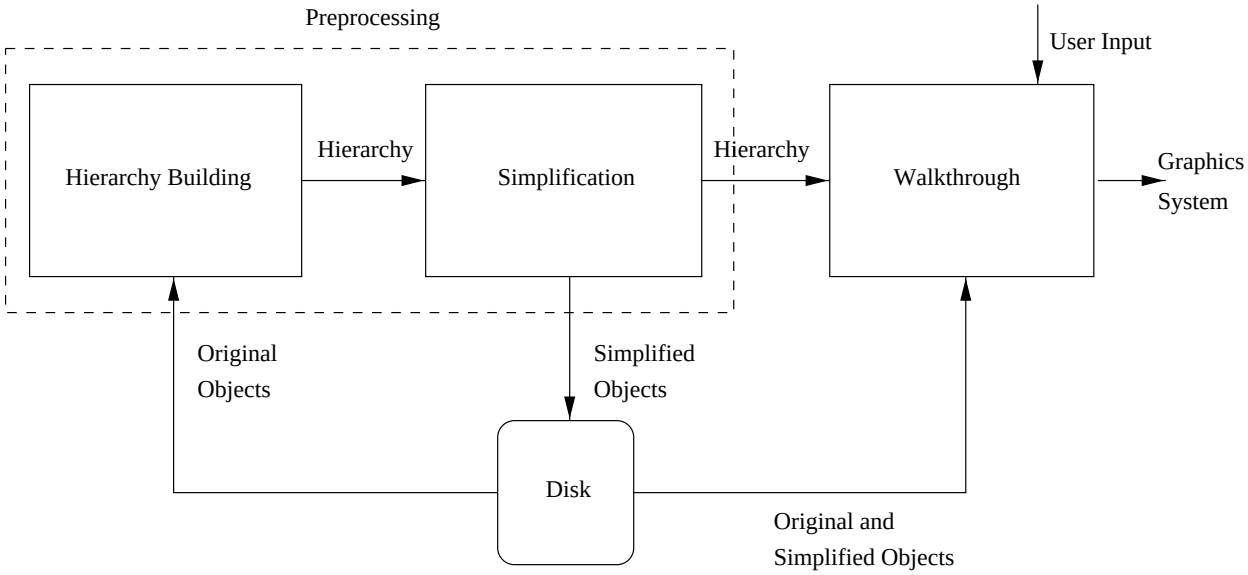

Figure 3. Block diagram of the walkthrough system

3.2. The Preprocessing Phase

In the preprocessing phase, scene data is read from the OSTF file and an object hierarchy is built in memory. Then, simplified versions are built and stored in separate files for later use. Each phase of the preprocessing is explained below.

3.2.1. Building the Hierarchy

The first step in preprocessing is building the hierarchy of the geometric model in memory. The hierarchy is in the form of a list. Each element of the list is the root of a tree representing an object. The person building the model by using CAD/CAM software should decide how to partition the scene into objects and how to model each object. Each different decision ends with a different hierarchy. Figure 4 shows a column's hierarchical representation. The system creates a tree for each object and inserts the root of the tree into the list representing the entire scene.

The object tree has two types of nodes. Intermediate nodes contain bounding box data and pointers to lower levels. These nodes do not contain any geometric data related to the objects. On the other hand, leaf nodes contain a list of triangles making up the object. Both types of nodes contain texture data. The texture of leaf nodes is taken from the OSTF file. For intermediate nodes, texture data is obtained by mixing the textures of its child nodes. Figure 6 shows an example of texture mixing for an object tree with 5 nodes.

In the hierarchy building phase, the OSTF files are read, and the hierarchy list and the object trees are created. For finding the bounding boxes of leaf nodes, triangle data is also read; however, it is not stored. Instead, they are used for computations and then discarded. The hierarchy list of the column in Figure 4 is shown in Figure 5. Black nodes represent intermediate nodes and white nodes represent leaf nodes. The depth of the hierarchy can increase depending on the complexity of the model. After the hierarchy list is built in memory, it is traversed for setting the texture and the bounding box data of child nodes. Bounding box of an intermediate node is the smallest box that contains all the bounding boxes of its child nodes.

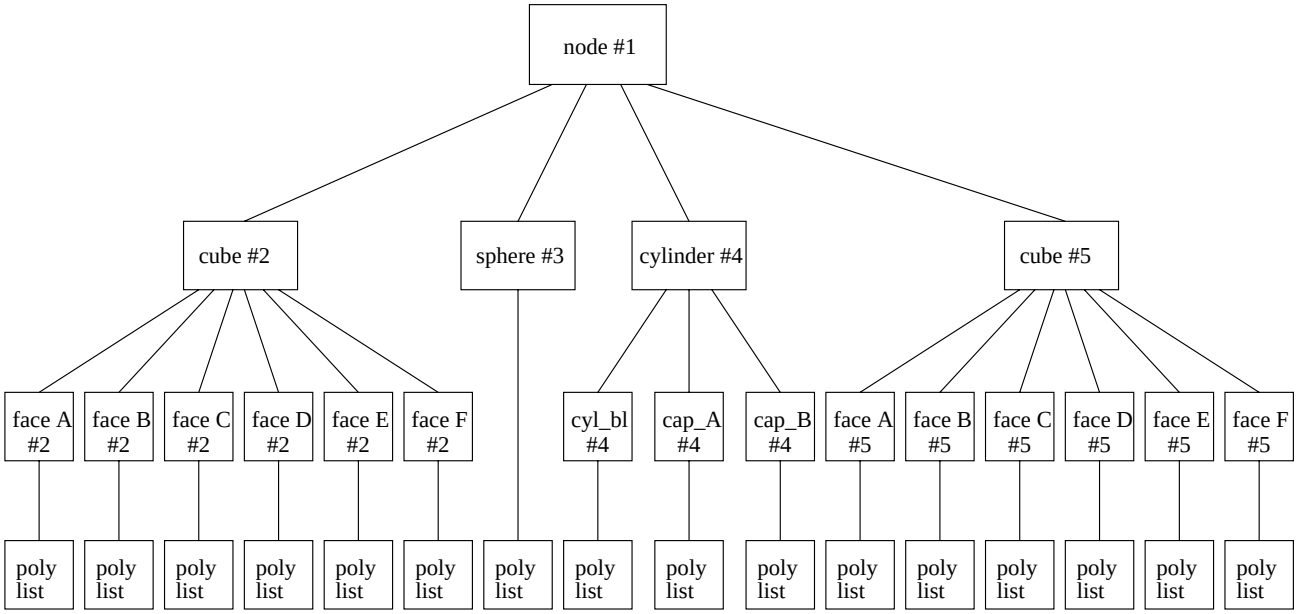

Figure 4. Hierarchical model of a column

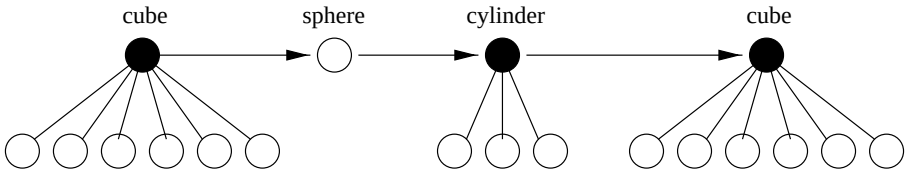

Figure 5. Hierarchy list of the column

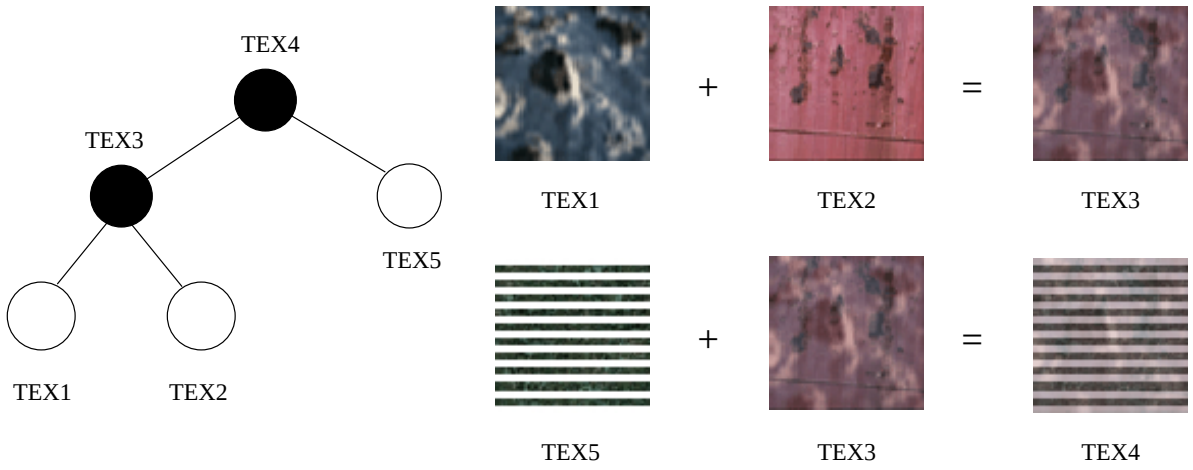

Figure 6. Finding textures of an intermediate node

```
For each object tree in the hierarchy list
   For each leaf node of object tree
      Read geometric data of triangles
      Find normals of triangles
      Find coplanar groups of triangles
      For each coplanar group
         Find bounding polygons
         For each bounding polygon
            Omit planar edges
            Reduce vertex data to 2D
            Triangulate
            Save triangles to object file
```

Figure 7. Geometric optimization algorithm

3.2.2. Simplification

The second phase of preprocessing is the simplification of geometric models. The geometric optimization method proposed in [3] is selected for polygonal simplification. The technique is simple to implement and is very effective on planar triangulated surfaces. In addition, the amount of simplification can be adjusted by giving threshold values enabling the generation of multiple LODs for a single object. Figure 7 gives the steps of a simplification algorithm. Details of each step are given in the sequel.

The operation of finding coplanar groups of triangles is applied to each leaf node in the hierarchy separately. Triangles of the node are grouped according to their normal vectors. Threshold value $\epsilon 1$ is used for grouping. It affects the number of polygons in coplanar groups. A small $\epsilon 1$ will cause lots of groups with few triangles and a large $\epsilon 1$ will cause the opposite. For each coplanar group created in the previous step, a boundary polygon is found by omitting the shared edges between triangles. Then, planar edges are removed from the bounding polygons. The second threshold $\epsilon 2$ is used for this purpose. This step causes a great number of vertices to be removed from the bounding polygon.

The boundary polygons are then triangulated. Depending on the value of threshold $\epsilon 1$ for creating coplanar groups, the vertices of a bounding polygon may not be on the same plane. Therefore, vertices of the bounding polygon are reduced to 2D coordinates. This guarantees that the vertices are on the same plane and simplifies triangulation. The 2D polygons formed may not be convex. Triangulation algorithms for non-convex polygons are complex and have restrictions. Triangulation in the system is a part of the simplification phase and triangles created in this phase will be displayed only when the object is far from the camera. Therefore, a fast and simple triangulation algorithm that can cause visual artifacts is implemented. The algorithm first creates the convex hull of a bounding polygon and then triangulates the convex polygon.

The Graham Scan [21] algorithm is used for finding convex hull. The convex polygon is easily triangulated. The triangulation changes the original polygon by omitting vertices that cause concavities. This results in different views of original and triangulated polygons. However, changes are usually not noticed by the user because simple versions are only displayed when the object is far from the camera.

3.3. The Walkthrough Phase

In the walkthrough phase, a virtual camera is moved inside the geometric model interactively. The user can control the direction and position of the camera. As the camera moves, a fast frustum culling is performed to discard objects that are out of the viewing frustum. An appropriate LOD of objects is selected and displayed according to the distance of the object to the viewpoint. Each step of the walkthrough phase is explained in detail in the following parts.

3.3.1. Frustum Culling

First, a viewing frustum is formed using a virtual camera model, which is defined using a direction vector, current position and field of view angle. These parameters can be changed interactively. After the viewing frustum is formed, objects in the scene are culled against that frustum. Hierarchical structure, which is built in the preprocessing phase, is used for calculations. The system creates four planes each of which are sides of the frustum for each camera movement. Frustum culling calculations are performed first on each element of the hierarchy list using the bounding box of the element. If all the vertices of a bounding box are outside the frustum planes, then the object represented by the element and its child nodes are ignored. It is also checked whether the volume formed by the vertices are cut across the frustum. To understand this, we check whether the vertices are outside any single plane of the frustum. If the bounding box is not outside the frustum, the algorithm continues with lower level nodes of the object tree. The algorithm ignores each node that is out of the frustum and does not process its children.

The explicit frustum culling used in the system introduces overhead when a large amount of objects are in the frustum. However, for cases when explicit frustum culling prunes object trees, its overhead is compensated.

3.3.2. Managing Level of Detail

In this phase, the system selects an appropriate LOD objects in the scene. Hierarchy list consists of roots of object trees. The intermediate nodes of object tree contain bounding boxes and texture data. Leaf nodes contain geometric data in addition to the bounding box and texture data. During walkthrough, each object tree in the viewing frustum is traversed. For each node, bounding box data and current camera position are used for calculations. The distance between the camera position and the center of each plane of the bounding box is calculated. The minimum of these six distance values gives an estimate of distance of the node to the viewpoint. If the distance is less than a user-defined threshold, children of the node are processed. For nodes that are far away from the camera, the bounding box is rendered using the texture of the node. In this the way, rendering operation for far objects is reduced to a textured rectangular prism rendering no matter what the geometric complexity is.

Distance calculation for near objects proceeds to the leaf nodes of the object tree. For leaf nodes, more than one threshold value is defined. Threshold values $\epsilon 1$, $\epsilon 2$ and distance of the leaf node to the camera determine which representation of the object to render. In addition to the representations created in the simplification phase, a textured bounding box of a leaf node is also a candidate for rendering.

Switching between different representations of an object introduces two problems to the system. The first one is the visual defects, such as popping, and the second one is the loading time of different representations. The solution of popping is through the blending of different representations. Since this is an expensive operation, the popping problem is not handled by the system. Thus, in exchange of performance increase, image quality is degraded.

For solving the disk-loading problem, memory caches are used by the system. There are two caches for each leaf node of an object. If the node is out of the viewing frustum, both caches are empty. For nodes that are in the viewing frustum, the caches contain two different representations of the node. The caching especially helps in reducing the delay of continuous switching for each back and forth camera movement, since both representations will be in memory through caching. The pseudo-code of LOD management is given in Figure 8.

```
For each element of hierarchy list
   If node in viewing frustum
      Find estimate of distance
      If distance greater than threshold
         Render textured bounding box
      Else
         If leaf node
           Select appropriate level of detail
            If representation not in cache
              Load representation to cache
            Render representation
         If intermediate node
            Process children
```

Figure 8. Level of detail management algorithm

4. Results

The results obtained by using the implementation is given in this section. The results of LOD management and walkthrough are provided separately.

4.1. Level of Detail Management Results

The parameters that affect the management of LOD are the hierarchy of objects in the scene and distance values for changing LOD. To explain the effect of both factors, the scene in Figure 9 is selected. The spheres are identical except their textures. Each sphere consists of 960 triangles. `Sphere 1`, `Sphere 2` and `Sphere 3` are grouped together to form `Node 28`.

The still frames in Figure 9 show LOD management for different camera positions. It should be noted that in order to determine an LOD of an object to be used for a particular frame, the distance from the camera to the center of the object is measured. The first change in LOD occurs when the distance an object center to the camera is larger than 40. If the distance of objects is larger than 80, a bounding box representation of the object is displayed. If an intermediate node's distance to the camera is larger than 160, a mixed-textured bounding box is displayed instead of the bounding boxes of its children. The only intermediate node in the example is `Node 28`. The initial scene in Figure 9 has 2,880 triangles, whereas the final scene has only a bounding box.

4.2. The Walkthrough System

The system differs from a straightforward walkthrough application in three aspects. A straightforward walkthrough application directly draws all the polygons in the scene without any simplification for each viewing position.

The first difference is explicit frustum culling. Although graphics libraries make automatic frustum culling, a frustum control based on bounding boxes for faster elimination of vertices that are not visible can be used to reduce the amount of work for this process. The frustum is recalculated for each camera movement and each vertex of the bounding boxes of the nodes in the hierarchy is checked for each new frame. It is an expensive operation because a 3D dot-product is required for each vertex. Therefore, if a great number of objects cannot be eliminated using this operation, the performance drops considerably.

The second difference is using distance calculations for LOD selection. To find an estimation of distance between visible objects and camera, the minimum distance between the camera and the bounding

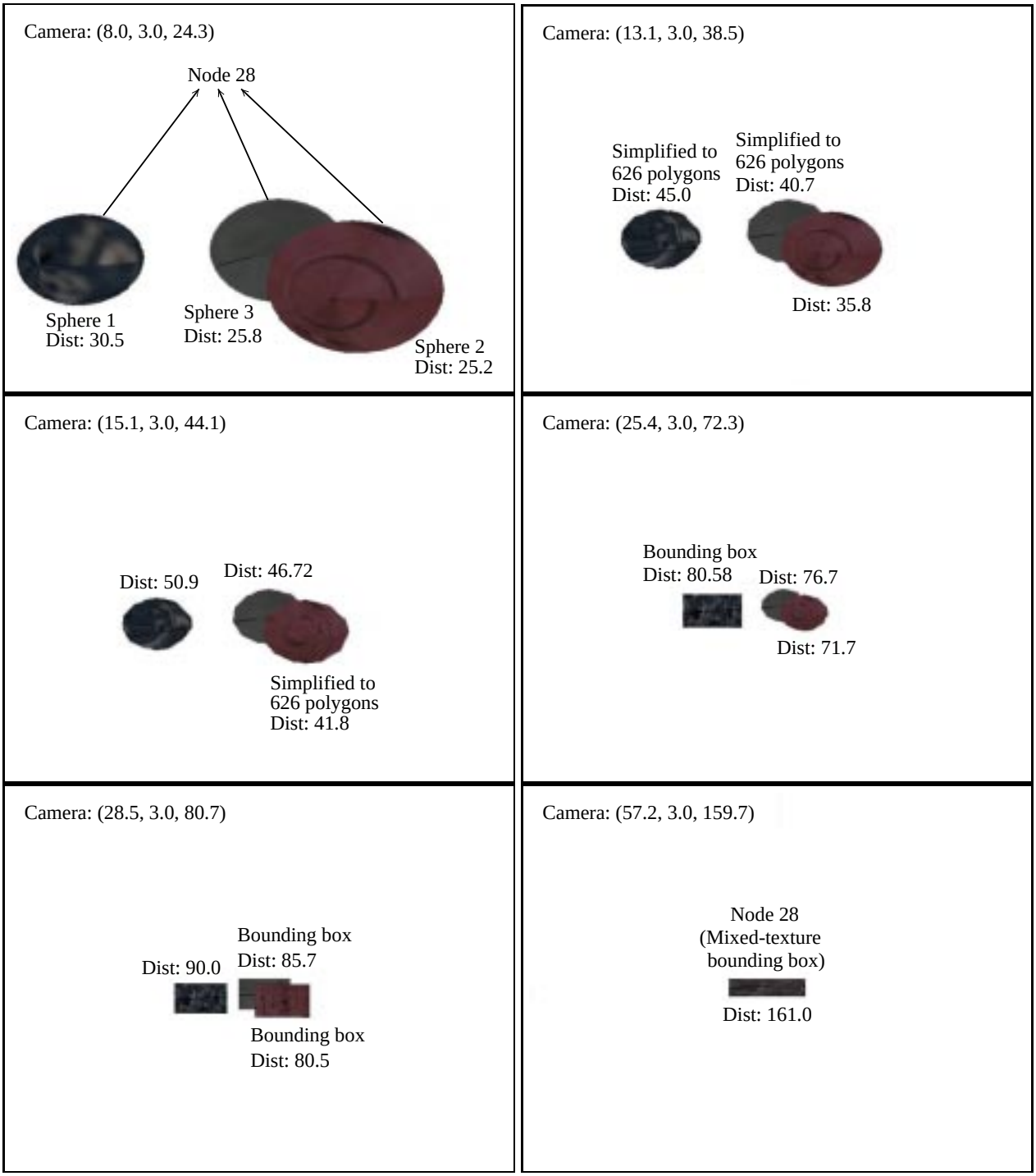

Figure 9. Level of detail management

box of the object is calculated. Distance calculations are also expensive operations. If detailed versions are rendered after distance calculations, the overall performance drops below normal walkthrough applications.

The third and most important difference is loading different representations of objects from disk. If the system reads data from disk during a walkthrough, the performance will drop considerably. The double-memory-cache helps to reduce the number of disk accesses for two-level-of-detail scenes. However, if the number of representations for objects is more than two, a new model should be loaded for each representation change. The number of memory caches can be increased, which will lead to memory shortage problems.

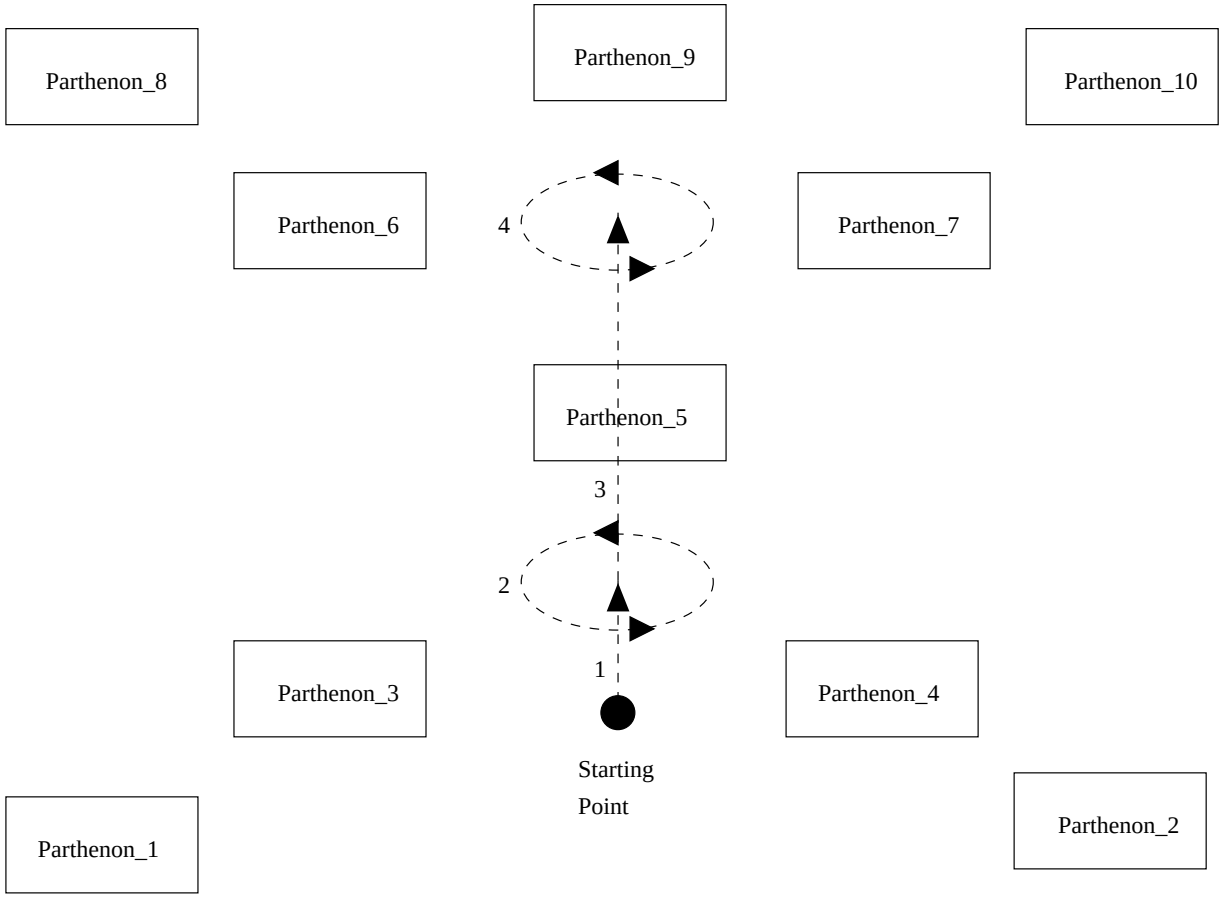

Figure 10. The top view of the test scene

To demonstrate these features, a scene containing 10 Parthenons (with over 400,000 polygons) is prepared. The top-view of the test scene is displayed in Figure 10. The dashed lines show the walkthrough path in the scene. Figure 11 shows still frames from the walkthrough of this scene. The orientations of the parthenons are different. All objects in the scene have two LOD representations. The walkthrough is divided into four parts and each part contains 8 frames. The numbers in Figure 10 correspond to these four parts. In the first part, the camera moves towards *Parthenon_5*. In the second part, the camera rotates around itself 360 degrees. In the third part, the camera moves inside *Parthenon_5* and exits it. In the fourth part, the camera rotates around itself 360 degrees.

Table 1 shows the improvements over a simple walkthrough system where all of the objects in the scene are rendered without any simplification regardless of the camera position. The system is executed in wire-

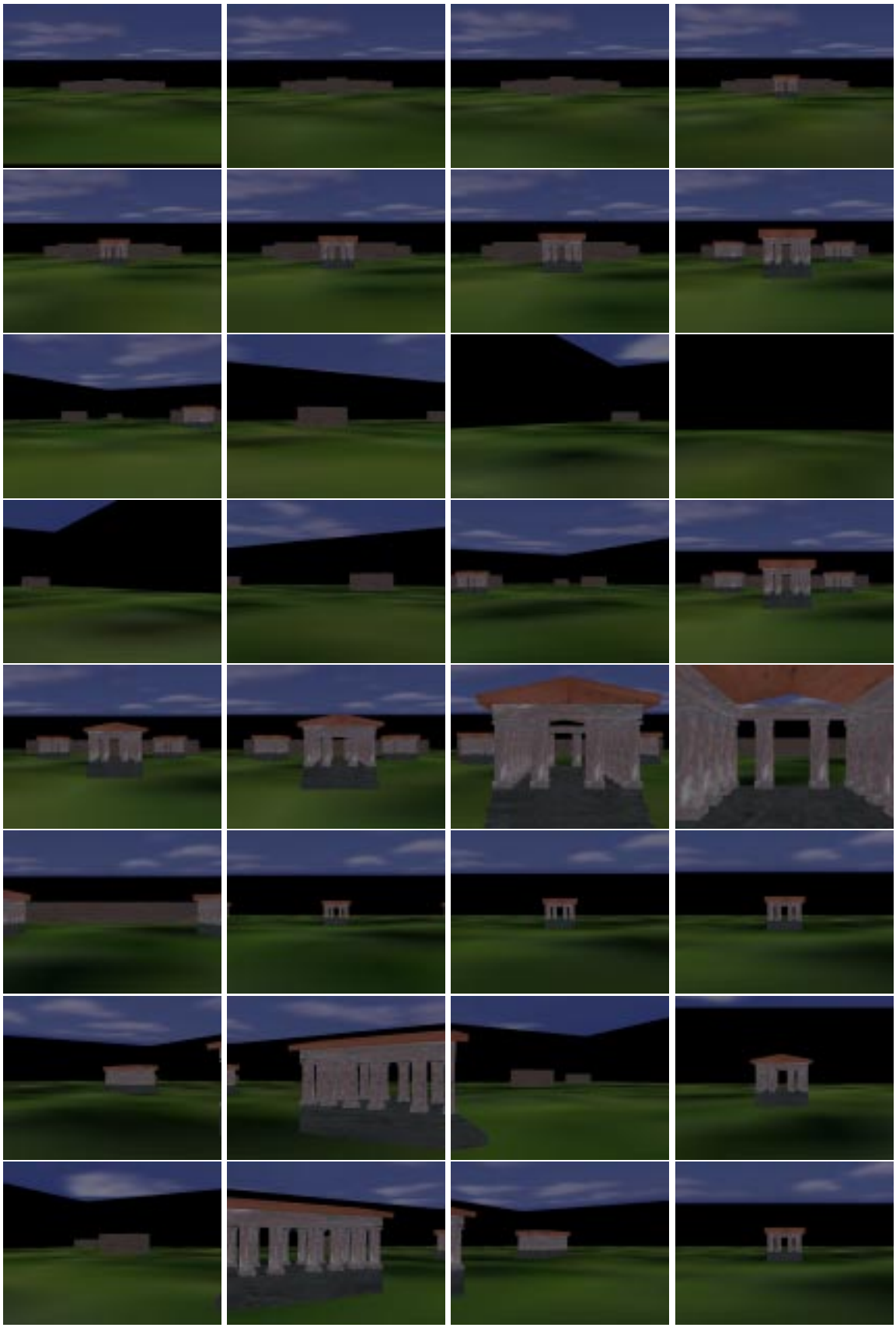

Figure 11. The still frames; the pictures are ordered rowwise from top-to-bottom and left-to-right

Table 1. Processing times for the walkthrough system (seconds)

Operations	**Frustum**	**Distance**	**Reading**	**Display**
Bounding box displayed	~ 0.00	~ 0.00	~ 0.00	~ 0.00
Bounding box displayed	0.02	0.03	0.00	0.00
Loading a simpler representation	0.02	0.06	0.70	0.05
Loading different representations	0.08	0.10	1.82	0.34
Loading different representations	0.06	0.08	1.77	0.55
No loading from disk	0.06	0.08	0.00	0.45
No loading from disk	0.02	0.05	0.00	0.48

frame mode on a Silicon Graphics Onyx with four 200MHz R4400 CPU, Reality Engine, 512MB RAM and 64MB texture memory (Silicon Graphics Onyx is a registered trademark of Silicon Graphics International). The simple walkthrough system, where the objects are rendered without any simplification, would require between 1.48 and 1.53 seconds per frame for this environment depending on the camera position. Table 4.2 gives the execution times of each step in the system for different camera positions on the path.

The first two rows show the processing times when the camera is far from the scene. Only the higher level nodes of object trees in the hierarchy list are tested. Since they are all far away from the camera their bounding boxes are displayed. Therefore, all values are near zero. As the camera gets closer, lower level nodes are also tested and, for some objects, simpler representations are loaded from disk to cache (row 3). Rows 4 and 5 show the worst cases of the system. Too many disk accesses are required for these shots. After most of the caches are filled, the execution times remain constant, as can be observed from, last two rows.

These results show that the display times are improved at worst more than three times compared to a simple system. Depending on the distance of camera, much better results can be obtained. The disadvantage of the system is disk-access times. When many objects are needed to be loaded to the cache, the system's performance gets worse than that of the simple walkthrough application.

5. Conclusions and Future Work

A system for a walkthrough in complex architectural environments and outdoor scenes is explained in this paper. The system takes as input triangulated hierarchical models. First, the object hierarchy is built in memory. Then, simplified versions of objects are created. The simplification algorithm is based on removing nearly coplanar triangles from the model. Triangles with nearly the same normals are grouped together. For each group, boundary polygons are found. Planar edges are removed from the boundary polygons. The remaining vertices of the boundary polygons are triangulated. The simplification algorithm is very successful on objects with planar surfaces. It reduces the number of triangles considerably without affecting the image quality.

For walkthrough, a virtual camera and a viewing frustum is defined. The user moves the camera inside the model as if he/she walks in the model. As the camera moves, the nodes in the hierarchy are traversed. Nodes that are out of the viewing frustum are discarded. Frustum culling calculations are based on bounding boxes. For near objects, detailed representations are displayed, whereas for far objects simplified versions are shown. To reduce the model-loading overhead, double-memory caches for storing recently used representations of an object are used.

To improve the performance of the system both in terms of simplification and frame-rate, the following

could be done.

i. Since the system has a modular structure, the geometric simplification algorithm can be improved or replaced.

ii. The intermediate nodes in the object hierarchy are represented by textured cubes. The textures of the cubes are obtained by mixing the textures of lower level nodes. More accurate representations can be used for intermediate nodes.

iii. The system currently performs only visibility culling. For architectural models, occlusion culling algorithms are very important for reducing the number of vertices. An occlusion culling method could be adapted to the system.

iv. For reducing the number of mode changes when rendering a model, primitives with similar textures can be grouped together.

v. Current double-memory cache system solves the problem of model loading for two LODs. The caching system can be optimized for more LODs by increasing the number of caches.

Acknowledgment

This project is supported by an equipment grant from the Scientific and Technical Research Council of Turkey (TÜBİTAK) the grant no. 198E018.

References

[1] K. Chiu and P. Shirley, "Rendering, complexity and perception," in *Proc. of the 5th Eurographics Rendering Workshop*, 1994.

[2] W. Schroeder, A. Jonathan, and E. Lorensen, "Decimation of triangle meshes," in *Proc. of SIGGRAPH'92*, pp. 65–70, 1992.

[3] P. Hinker and C. Hansen, "Geometric optimization," in *Proc. of Visualization'93*, pp. 189–195, 1993.

[4] J. Rossignac and P. Borrel, "Multi-resolution 3d approximations for rendering complex scenes," in *Modeling in Computer Graphics: Methods and Applications*, pp. 455–465, 1993.

[5] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh optimization," in *Proc. of SIGGRAPH'93*, pp. 19–26, 1993.

[6] R. Ronfard and J. Rossignac, "Full-range approximation of triangulated polyhedra," *Computer Graphics Forum*, vol. 15, no. 3, pp. C68–C76, 1996.

[7] M. Garland and P. Heckbert, "Surface simplification using quadric error metrics," in *Proc. of SIGGRAPH'97*, pp. 209–216, 1997.

[8] G. Turk, "Re-tiling polygonal surfaces," in *Proc. of SIGGRAPH'92*, pp. 55–64, 1992.

[9] T. Funkhouser and C. Sequin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," in *Proc. of SIGGRAPH'93*, pp. 247–254, 1993.

[10] D. Luebke and C. Erikson, "View-dependent simplification of arbitrary polygonal environments," in *Proc. of SIGGRAPH'97*, pp. 199–208, 1997.

[11] J. Xia, J. El-Sana, and A. Varshney, "Adaptive real-time level-of-detail based rendering for polygonal models," *IEEE Trans on Visualization and Computer Graphics*, vol. 3, no. 2, pp. 171–183, 1997.

[12] H. Hoppe, "View-dependent refinement of progressive meshes," in *Proc. of SIGGRAPH'97*, pp. 189–198, 1997.

[13] S. Belblidia, J. Perrin, and J. Paul, "Generating levels of detail of architectural objects for image-quality and frame-rate control rendering," in *Proc. of Computer Graphics International'96*, 1996.

[14] P. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Computer Graphics*, pp. 95–102, 1995.

[15] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, "Hierarchical image caching for accelerated walkthroughs of complex environments," in *Proc. of SIGGRAPH'96*, pp. 75–82, 1996.

[16] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, "Fast rendering of complex environments using a spatial hierarchy," in *Proc. of Graphics Interface'96*, pp. 132–141, 1996.

[17] J. Clark, "Hierarchical geometric models for visible surface algorithms," *Communications of ACM*, vol. 19, no. 10, pp. 547–554, 1976.

[18] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proc. of SIGGRAPH'93*, pp. 231–236, 1993.

[19] S. Teller and C. Sequin, "Visibility preprocessing for interactive walkthroughs," in *Proc. of SIGGRAPH'91*, pp. 61–69, 1991.

[20] Silicon Graphics International, Inc., *SGI Alias Menu Book*. 1996.

[21] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*. Berlin: Springer-Verlag, 1985.