DESIGN AND SPEEDUP OF AN INDOOR WALKTHROUGH SYSTEM

Bin Chan, Wenping Wang
The University of Hong Kong, Hong Kong
Mark Green
The University of Alberta, Canada

ABSTRACT

This paper describes issues in designing a practical walkthrough system for indoor environments. We propose several new rendering speedup techniques implemented in our walkthrough system. Tests have been carried out to benchmark the speedup brought about by different techniques. One of these techniques, called *dynamic visibility*, which is an extended view frustum algorithm with object space information integrated. It proves to be more efficient than existing standard visibility preprocessing methods without using pre-computed data.

1. Introduction

Architectural walkthrough models are useful in previewing a building before it is built and in directory systems that guide visitors to a particular location. Two primary problems in developing walkthrough software are modeling and a real-time display. That is, how to easily create the 3D model and how to display a complex architectural model at a real-time rate.

Modeling involves the creation of a building and the placement of furniture and other decorative objects. Although most commercially available CAD software serves similar purposes, it is not flexible enough to allow us to easily create openings on walls, such as doors and windows. Since in our own input for building a walkthrough system consists of 2D floor plans, we developed a program that converts the 2D floor plans into 3D models and handles door and window openings in a semi-automatic manner.

By rendering we refer to computing illumination and generating a 2D view of a 3D scene at an interactive rate. The radiosity method is well known for global illumination computation, and has been used for preprocessing of some walkthrough applications. However, we chose not to use it at this stage for the following reasons. First, the radiosity method is complicated to implement and time consuming to apply, especially since our model consists of several floors of a large building. Second, the large number of polygons that would be generated by the radiosity method is prohibitive on the moderate geometry engine of our median-level graphics workstation. Representing the results of radiosity computation in a texture map is a possible solution, but that would increase the demand on the limited and expensive texture memory of our workstation. Hence at this stage we choose to use illumination textures to simulate the lighting effects in the virtual environment. We also use texture mapping to simulate some other lighting effects, such as specular reflection. Satisfactory results have resulted from these expedient treatments.

Rendering-speedup techniques are most important in designing a walkthrough application since they strongly affect the frame rate and hence the usefulness of the system. We have implemented and tested a few standard techniques, such as the potential visible set method [4], and compared them with a new runtime visibility computation technique: the dynamic visibility method. Our tests show that the new visibility computation method yields superior performance.

1.1 OVERVIEW

The virtual environment we built consists of two floors of a building housing the Department of Computer Science at the University of Hong Kong. Figure 1 shows a typical scene captured from the walkthrough engine. Our walkthrough system was written in *C* and *OpenGL* to allow maximum flexibility in incorporating new algorithmic improvement at the low level of the geometric processing.

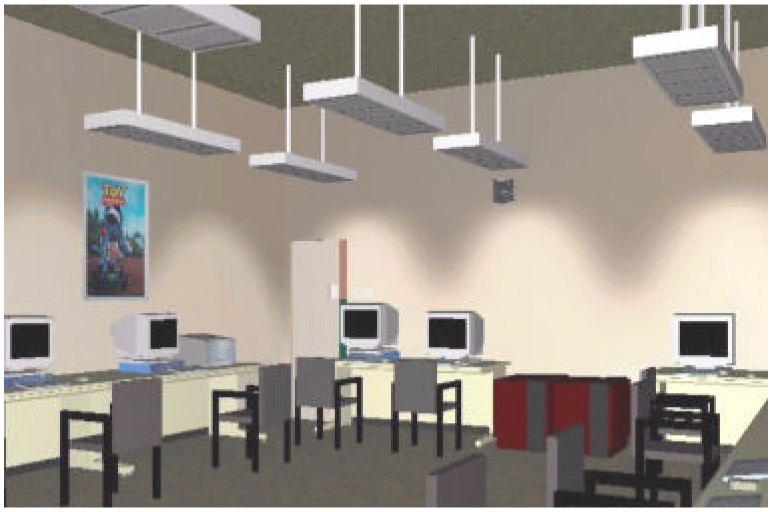


Figure 1. A snapshots of a test system.

This paper is organized as follows. Section 2 is a brief survey of related existing work. Section 3 discusses some issues about modeling and texture mapping for illumination. Section 4 describes the different speedup techniques. Section 6 presents the benchmark results. Section 7 contains conclusions.

2. Related work

Since our emphesis is on speedup techniques, we start by reviewing some speedup-related techniques. Airey et al. [3] and Sequin [4] built systems that compute the potential visible set (PVS) to effectively eliminate many hidden polygons in a densely occluded indoor environment. The PVS is generated offline. The information generated is of two types: cell-to-cell visibility and cell-to-object visibility. The algorithm first divides an environment into cells. These cells are then scanned one by one in such a way that all other cells and objects that are visible to a particular cell are recorded. During the rendering phase all cells and objects visible to the cell containing the viewpoint are extracted from the PVS database and are rendered.

Luebke and Georges [5] proposed a run-time method to compute visibility information in 1995. That method is to project portals and objects on screen space and compare screen space bounding rectangles to determine objects' visibility. Since that paper does not give very detailed descriptions and results, we cannot tell how the screen space projection and comparison is done. They used a proprietary graphics workstation, that is not common. In common OpenGL systems, screen space coordinates are not returned. In that case projection must be done explicitly by the walkthrough system, which may waste large amount of CPU time. Using numerical computation instead of screen space comparison should give better results and be much simpler.

Aliaga and Lastra [8] proposed a system that uses textures to cover portals. Multiple views through each portal in different directions are rendered either offline or at runtime. In the walkthrough the portals are replaced by warped textures, and no cell or object behind the portal is rendered except when the viewpoint is very close to the portal. Such a treatment involves a transition from frames using portal texture to those not using it. For the transition to be less noticeable, more than one texture per portal must be prepared. The textures used for this purpose cannot be too small, for otherwise a portal will look too blurry to be realistic. The textures used in their system are of size 256x256 with 8 bit per color component, i.e. 256KB per texture. A few such textures used at the same time could easily overload the texture memory.

Recently Zhang, Manocha, Hudson, and Hoff [7] proposed a hierarchical occlusion map algorithm similar to the hierarchical z-buffer algorithm. Their algorithm requires no special graphics hardware except texture mapping hardware. It chooses some objects as occluders and then eliminates those objects behind the occluders. A multi-resolution occlusion map is maintained to support quick checking of potential occlusion. Unlike the hierarchical z-buffer algorithm, the hierarchical occlusion maps are scaled down using the texture mapping hardware. The method is efficient if there are many objects behind the occluders. That is, the occluders should be sufficiently large as to block many objects. However, it is usually not easy to choose such occluders for indoor architecture models.

Other image-based systems [14, 15, 16] and semi-image-based systems [1, 6, 8] have recently become common. Nevertheless, most of these methods either require special image hardware to be efficient or they consume large amounts of CPU time and memory. Those attributes make them not very practical for high-resolution indoor walkthrough.

Some other systems explore Level of Detail (LOD) techniques [20] for display speedup. In LOD different models of varying resolutions are prepared for an object, and are applied at different distances from the viewer. Currently we concentrate on the study of visibility culling algorithms, so LOD is not considered in this paper.

3. Modeling

There are many papers describing different techniques in modeling. Since our main concern is not in this area, we will only briefly describe the methods we examined, most of which are straightforward.

The 3D models are built from 2D AutoCAD floor plans. The floor plans are divided in cells loosely based on the division of rooms. All line segments intersecting the cell boundaries are trimmed so all those belonging to a cell must lie entirely inside the cell. Thus the extruded 3D models have the property that all the polygons of a cell lie entirely inside the 3D bounding rectangular block of the cell. Openings on the boundaries between cells are marked as portals. These cell boundaries and portals are drawn directly in AutoCAD and exported as DXF files as well. Later they are converted to the internal cell and portal database files.

After 3D wall models have been generated, they are passed to the object-placing program, which is used to place furniture and other objects interactively in the environment. There is a library of standard furniture for the user to select from and put in the room. A gravity-based model is used in the object placement program to facilitate the process. A simple collision detection algorithm is used to check collision between the bounding boxes of objects. After all objects have been placed, the object IDs and their transformation matrices are stored in a cell file.

3.1 TEXTURE MAPPING FOR ILLUMINATION

Our graphics hardware uses Gouraud shading when displaying the 3D scene. To make the scene look more realistic with illumination effects, texture mapping is used to simulate two special and subtle illumination effects: soft shadows and the reflection of light sources on the floor (see figure 2).

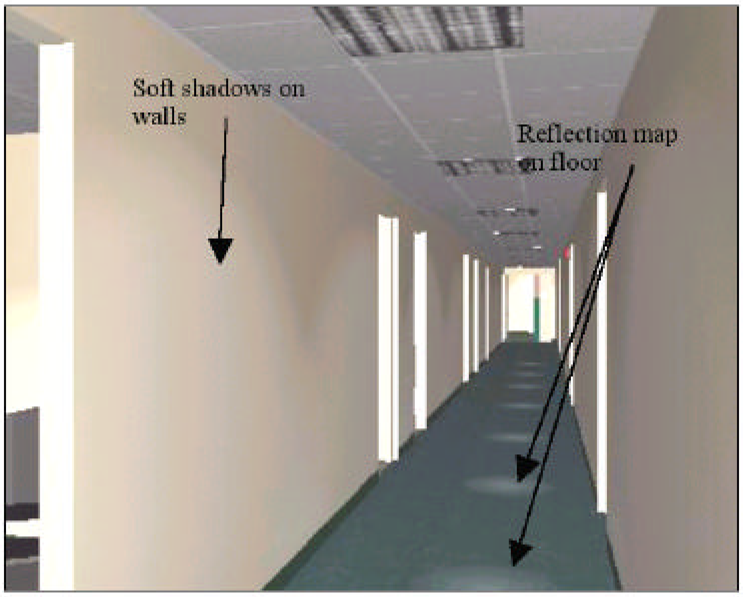


Figure 2. Illumination textures maps

Soft shadows on a wall cast by light sources on the ceiling usually take hyperbolic shapes. Therefore a shadow map of hyperbolic shape is used to define an illumination texture on the wall.

Texture mapping is also used to simulate the effect of specular reflection of a semi-reflective rough floor. Since specular reflection moves with the user's viewpoint, the texture cannot be mapped directly onto the model. Instead, a texture, called a reflection map, is mapped onto a floating rectangle above the ground and moves with the viewpoint to simulate the specular reflection of a light source on the floor.

4. Visibility Computation

4.1 PRE-COMPUTATIONS

Among various speedup techniques for indoor environments, the visibility pre-computation is one of the most widely applied. Pre-computation alleviates run-time CPU usage by performing visibility computation offline. The *potential visible set* (PVS) method [4] is the best-known visibility pre-computation method. After cutting the environment by cell and portal partitioning, visibility between cells and objects are pre-computed and stored in a database by the PVS method. Polygons are culled according to information stored in the database. The culling performance depends highly on the size and number of cells; so also does the visibility database and runtime memory requirement. Thus there is a tradeoff between performance and memory requirement in the PVS method.

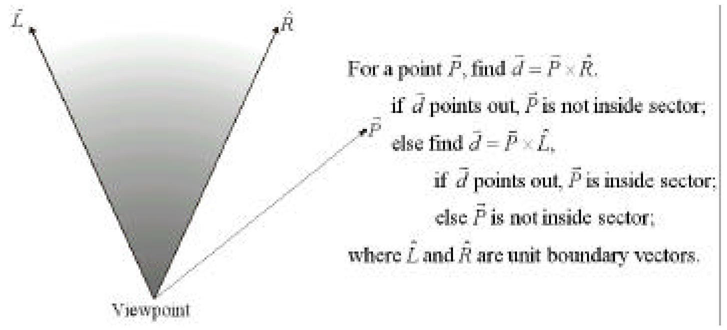
Since the PVS itself does not provide very tight culling, view frustum culling is ususally put behind PVS culling. With the two methods working together, the speedup is rather significant. There has been little work on runtime visibility computations, partly as a result of the success of the pre-computation based techniques.

4.2 RUN-TIME SPEEDUP TECHNIQUES

All speedup techniques used in our system are runtime techniques. Runtime techniques have some important properties not shared by techniques based on pre-computation: (1) they support dynamic scenes; (2) they need no preprocessing and pre-computed data; (3) they use little runtime memory. In the following we shall discuss the idea and evaluation of the dynamic visibility method.

4.3 VIEW FRUSTUM CULLING

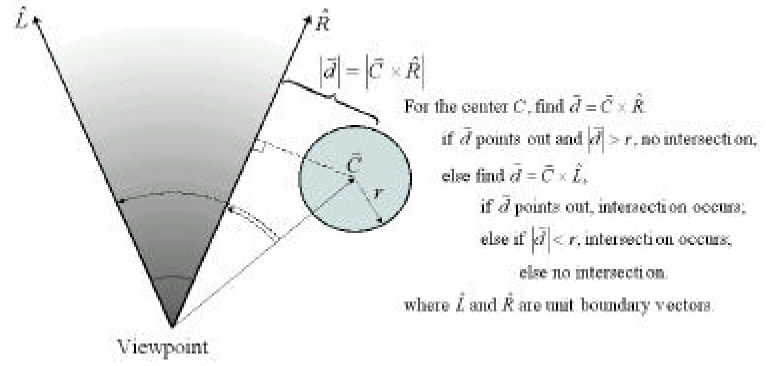
View frustum culling is a classic visibility computation method. It has been used in numerous systems, including systems that use pre-computations. A careful implementation of view frustum culling can be very fast, requiring only three multiplications and a few comparisons per point on the average (see figure 3).



```
For a point $\vec{P}$, find $\vec{d} = \vec{P} \times \hat{R}$.
    if $\vec{d}$ points out, $\vec{P}$ is not inside sector;
    else find $\vec{d} = \vec{P} \times \hat{L}$,
        if $\vec{d}$ points out, $\vec{P}$ is inside sector;
        else $\vec{P}$ is not inside sector;
where $\hat{L}$ and $\hat{R}$ are unit boundary vectors.
```

Figure 3. Testing if a point is in the view frustum. In 2D case, each cross product needs only two multiplications and one addition. On average, only three multiplications are needed since about half of the points will fail the first test.

For an object with a rectangular bounding box, the above computation may need be repeated up to four times, depending on the position of the object. For an object with a circular bounding box, the computation is similar to testing one single point, making it more efficient than the rectangular bounding box (see figure 4).



```
For the center C, find $\vec{d} = \vec{C} \times \hat{R}$
    if $\vec{d}$ points out and $|\vec{d}| > r$, no intersection;
    else find $\vec{d} = \vec{C} \times \hat{L}$,
        if $\vec{d}$ points out, intersection occurs;
        else if $|\vec{d}| < r$, intersection occurs;
            else no intersection.
where $\hat{L}$ and $\hat{R}$ are unit boundary vectors.
```

Figure 4. Testing if an object with a circular bounding box is in the view frustum. Finding the magnitude of $\vec{d}$ here does not involve any square root because vector $\vec{d}$ does not have x and y components.

Despite its simplicity, view frustum culling does not exploit object space coherence, except for bounding box information. Cell and portal partitioning provide useful object space information, which the PVS method uses to attain considerable speedup. We observe that runtime techniques can even make better use of similar object space information to achieve speedup. Based on this observation we modify the classic view frustum algorithm by integrating cell and portal partitioning information to obtain a new visibility algorithm. That algorithm is called the dynamic visibility method because it is executed at runtime and supports dynamic environments.

4.4 DYNAMIC VISIBILITY

Dynamic visibility is quite similar to Luebke's method [5] in the sense that the object space is traversed. However, it does not rely on screen space comparison. Instead, it computes numerically the visibility in 2D. It is not only much simpler, but also faster. This enables us to extend culling to polygon level.

Dynamic visibility is an extended view frustum culling algorithm with integrated object space information. The bounding boxes of objects are tested for intersection with a varying view frustum by comparing the sector formed from the viewpoint to the bounding box of an object with the view frustum. These tests are recursively done cell by cell along different portal sequences. The view frustum is modified by intersecting a visible portal with the view frustum. As the checking goes down the recursion, the view frustum is narrowed when it goes though a portal of a new visible cell. The recursion stops when there is no additional portal cutting the view frustum (see figure 5).

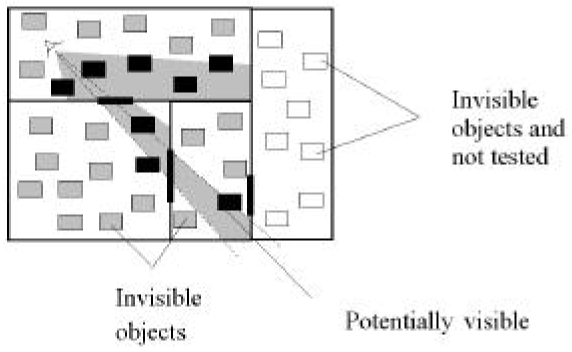


Figure 5. View frustum culling is integrated with cell and portal partitioning.

Dynamic visibility, besides possessing the advantages of a runtime technique, generally provides a tighter culling than the PVS method (compare figures 5 and 6). This is mainly because the system knows the exact viewer position at runtime so it yields more accurate culling. The PVS method, in contrast, can only assume the same visibility status for a large area of potential viewer positions, i.e. inside the same cell in PVS. It is conceded that the PVS method can also achieve a similar level of culling accuracy if cells are made smaller; but in this case the total number of cells increases drastically, further increasing the time for preprocessing and storage of the pre-computed data.

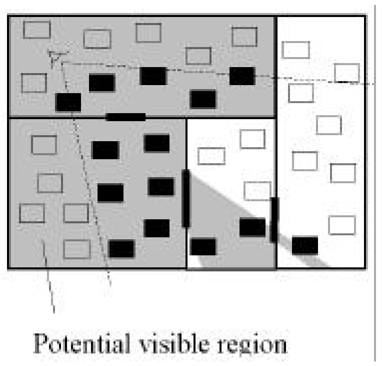


Figure 6. The results of PVS plus view frustum culling. Note that more potentially visible objects are reported than using the dynamic visibility method.

The dynamic visibility method described above has been applied at the object level. All polygons of objects that are visible or just partially visible are rendered. To further tighten culling, the same technique is further extended to the polygon level. In our experience the set of remaining polygons that cannot be culled by polygon level dynamic visibility should be very close to the set of visible polygons.

5. Software Vertex Color Calculations

Another speedup technique we tested is software vertex color calculation. Conventionally, in interactive display of a 3D polygonal model with Gouraud shading, vertex colors are computed by passing normal vectors at the vertices to some primitive drawing functions, which are OpenGL functions in our case. By software vertex color calculations we mean that the vertex colors of a polygon are calculated by the walkthrough engine at runtime, instead of OpenGL functions. In this way, the colors of all vertices can be found before starting the graphics pipeline. For the repeated vertices, only their colors are passed to graphics hardware so that no lighting calculation by graphics hardware is needed, and only the color interpolating and texture mapping functions of graphics hardware are utilized.

This speedup technique is motivated by the observation that many vertices of an object are repeated in more than one, and often up to six, triangles. When adjacent triangles cannot be specially arranged to take advantage of fast triangle drawing functions, such as *triangle_strip(),* the vertex colors of these triangles are normally calculated more than once by graphics hardware in one display cycle. They are computed only once by the walkthrough engine per display cycle in software vertex color calculation.

To make the software computation of vertex color more efficient, we choose to compute the diffuse reflection only, and as might be expected, this assumption turns out not to adversely affect the illumination result since most indoor objects consist of predominantly diffuse surfaces. Consider the process of using graphics lighting hardware that handles diffuse and specular reflection as well as many other lighting effects. Even when the specular terms of most objects are set to zero, the graphics hardware still has to calculate the specular reflection, which costs some extra time. As a comparison, in software computation only the diffuse term in the lighting model is calculated, so the computation is simpler and faster. Besides, by using the software approach, we can define many light sources regardless of the OpenGL limitations; for example, OpenGL supports only up to 8 light sources. Moreover, color calculating hardware exists only in high-end graphics hardware, such as the SGI Onyx2 Infinite Reality. On most systems color calculation is carried out by OpenGL software. Software lighting is a simpler replacement of the relatively complicated OpenGL lighting model.

We found that putting one light source at the viewpoint is very useful in illuminating the environment. The placement of such light source also has one benefit of efficient back-face culling. The dot product of the view vector from the viewpoint to the polygon and its normal must be computed in calculating its vertex color. This information can be used directly to determine the visibility of that polygon. This further releases graphics hardware of the work on back-face culling.

6. Experimental results

Most papers in the literature about virtual reality systems do not give detailed benchmark results. This is due to the fact that benchmarking of virtual reality systems depends on many factors, such as machine platform, model size, display quality and the amount of texture used. There is no standard for comparing different systems. Nevertheless, a benchmark is very useful for understanding speedup techniques. It also gives hints about the directions of improvement. Therefore we shall give some benchmark results of our system.

The data set used is a model of two floors of Chow Yei Ching Building housing the Department of Computer Science at the University of Hong Kong (see figure 1 again). The tests were carried out on an SGI Maximum IMPACT workstation with an R10000 CPU, 192MB main memory, and 4MB texture memory. The whole model, including furniture, consists of 104102 triangles. The walkthrough was run full screen with a resolution of 1280x1024 pixels.

Figure 7 shows the speedup effect of each non-polygon-culling techniques along a typical path of 1409 frames with all other speedup techniques turned off. We can see from the figure that the three methods give quite uniform speedup across the entire path. In other words, they are not very sensitive to the position of the viewer and the structure of the cells and objects. This is a desirable property since it ensures a certain amount of speedup, though not necessarily significant, in all cases. Object-culling algorithms, on the other hand, do not have this property, as we shall see later.

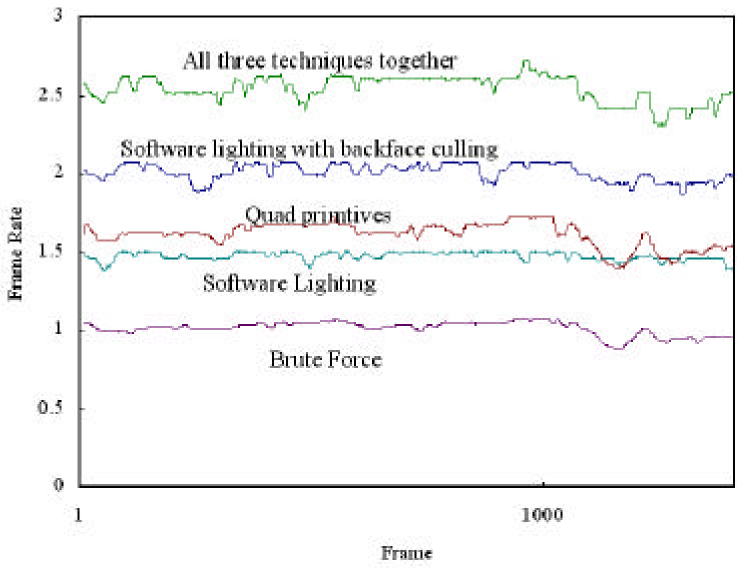


Figure 7. Relative performances of different non-polygon-culling methods. Note that the 'quad primitives' curve means the frame rates when most of the triangles are combined into quadrilaterals before rendering.

The dynamic visibility method has been tested in the same environment as the last test. The model is divided into 47 rectangular cells. For convenience in modeling, each cell is mapped to a room, a corridor or a large open area, which all turn out to be rectangular in our model. In the following evaluation the dynamic visibility method is compared with the PVS method. Figure 8 shows the speedup effects of different polygon-culling techniques across a typical path of 1409 frames.

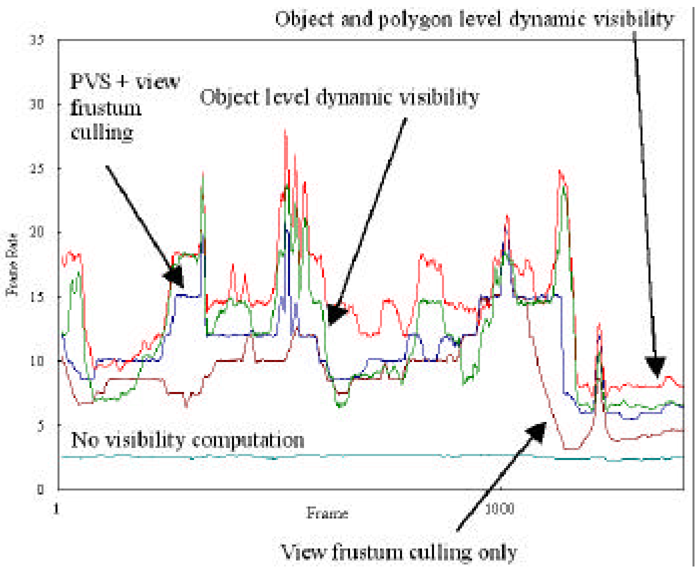


Figure 8. Relative performances of different culling methods.

We can see from figure 8 that the dynamic visibility method gives overall higher frame rate than the PVS plus the view frustum culling along the entire path, even without the aid of pre-computed visibility data. This means that the amount of time spent on runtime visibility computation is justified by the time saved in rendering fewer remaining polygons. Although object level dynamic visibility alone can produce a higher frame rate than the PVS method, sometimes the performance is quite close (see the rightmost portion of figure 8). For example, the performance is quite similar when looking along a corridor with the doors of many rooms open (see figure 9). In these situations polygon level dynamic visibility can further save some time.

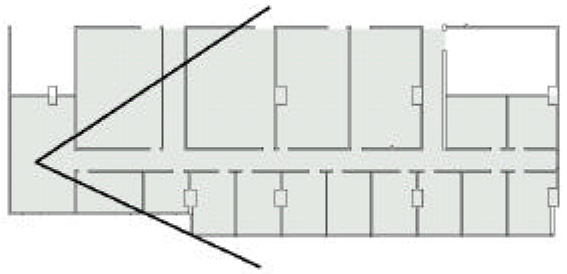


Figure 9. A situation in which the dynamic visibility method performs poorly. The view point is in the lower left cell looking at a long corridor. There are over twenty potential visible cells. In this situation, dynamic visibility performs like PVS.

With dynamic visibility our system can achieve an average frame rate of 16 .1 frames per second, which represents significant speedup from 2.5 fps if no visibility culling technique is used.

To gain a better understanding of the distribution of computation time, another set of tests was carried out. The tests were designed to reveal the relationship between the polygon culling time and the graphics rendering time. In each test a different percentage was set to determine the number of objects or polygons to be dynamically culled. A random number generator is used to choose the polygons to be tested, and those remaining were passed directly to rendering hardware .

Figure 10 shows the relation between total frame time and computation time. The total time drops with increasing object-culling percentage due to the decreasing number of polygons to be rendered. The computation time was recorded by running the same walkthrough path, but skipping the graphics pipeline function calls. Besides the time needed for dynamic visibility computation, it also includes the time spent on some other processes, such as back face culling and data structure traversal. That computation time drops with increasing object-culling percentage is mainly due to the fact that there are still a substantial number of polygons remaining after object level dynamic visibility. These remaining polygons have to undergo further processing before they are rendered. This subsequent processing time dominates the computation time when object-culling percentage is low, explaining the decreasing curve of computation time.

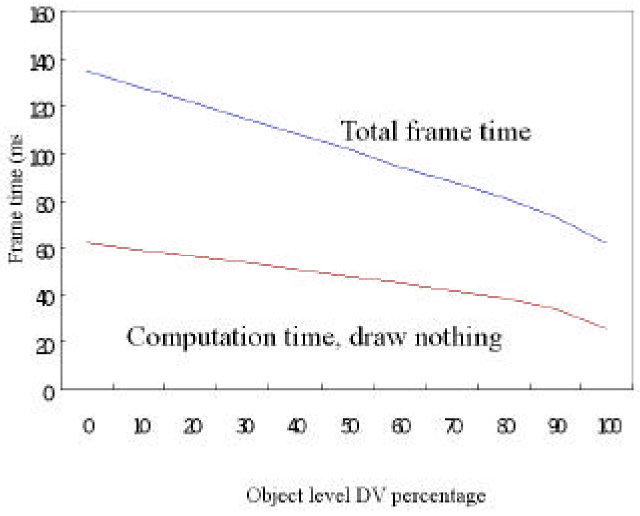


Figure 10. Relationship between rendering time and computation time of object level DV.

If polygon level dynamic visibility is applied after object level culling, the situation becomes a bit different (see figure 11), as the computation time is now a slightly increasing curve. That is because polygon level computation consumes much more CPU time than object level culling. Each object contains a few hundred polygons, but not many objects are left after 100% object level culling. That is, processing only few objects by polygon level culling can incur a large amount of computation. Hence the dynamic visibility computation time dominates the overall computation time in polygon level culling.

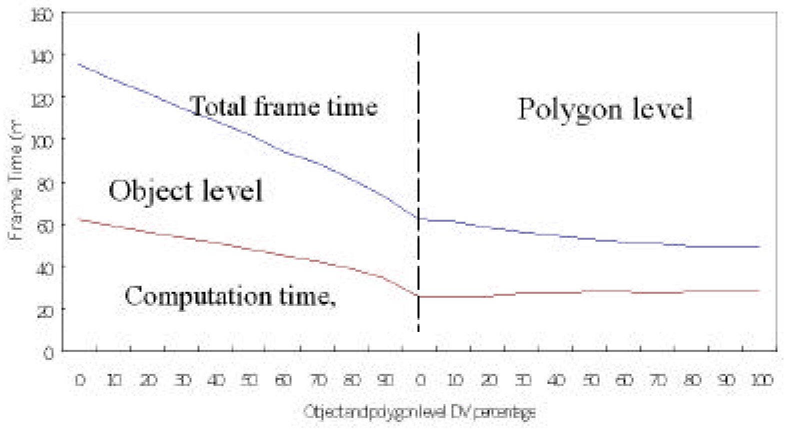


Figure 11. Relationship between rendering time and computation time of object level DV followed by polygon level DV.

We can see from figure 12 that with both object level culling and polygon level culling we can achieve an average frame rate of 16.1 frames per sec. That the frame rate curve levels off at the rightmost region of the chart means that the set of polygons actually rendered closely approximates the set of visible polygons, but the computation time is greater than the rendering time. In this case increasing the CPU speed has a greater affect than improving the performance of the graphics board. With the current technological development, the pace of CPU speed improvement is much faster than that of graphics board, especially for PC-based systems, the runtime visibility computation technique introduced above is expected to become more promising in the future. With the combination of all speedup techniques mentioned, we can achieve a frame rate of about 20 frames per second, compared to about 1 frame per second with no speedup at all.

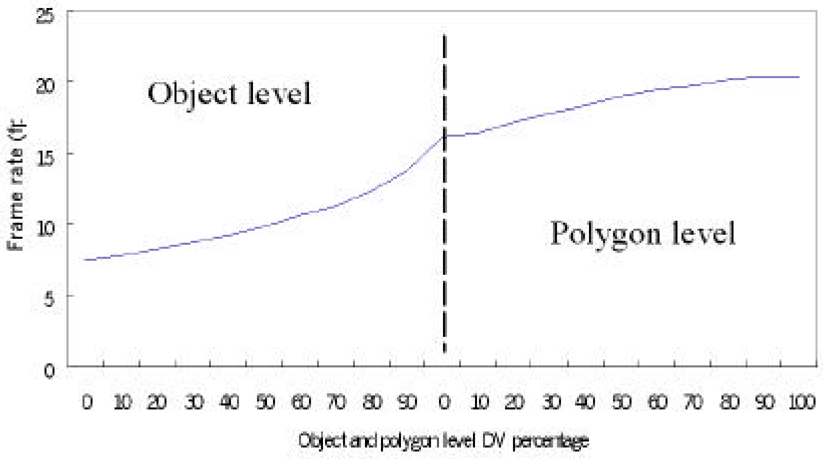


Figure 12. Frame rate against object level and polygon level DV percentage.

7. Conclusions

We have described a practical walkthrough system of architecture models and discussed some issues in modeling, illumination, and display speedup techniques. The speedup techniques used are all runtime methods, so the system supports dynamic environments. No images-based technique is used and no special hardware is needed, so the system can be implemented on most platforms. The paper also gives a benchmark of performance gain using different techniques for display speedup.

We have described a simple and fast visibility computation technique, called the dynamic visibility method. The method culls 99.9% of the polygons in our test indoor environment on average. This figure is model dependent but should not vary too much for typical densely occluded indoor environments.

Our method is a runtime method so it inherits all good properties of general runtime techniques. Our experiments show that it also yields tighter culling than the PVS method. Performance benchmarking has also been conducted to analyze the efficiency of this method for display speedup. The results show that CPU speed is important to the efficiency of this method since CPU-related processing consumes nearly 60% total frame time.

Besides visibility computation, we also described some other non-visibility speedup technique such as software lighting, which yields a uniform speedup along the walkthrough path.

REFERENCES

[1] Hujun Bao, Shang Fu and Qunsheng Peng, “Accelerated walkthrough of complex scenes based on visibility culling and image-based rendering”, *Proceedings of CAD and Graphics’97,* pp. 75-80.

[2] [2] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes, *Computer Graphics: Principles and Practice*. Addison Wesley 1990.

[3] John M Airey, “Towards image realism with interactive update rates in complex virtual building environments.”, *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*.

[4] S. Teller and C. Sequin, “Visibility preprocessing for interactive walkthroughs”, *Computer Graphics*, Vol. 25, No.4. pp. 61-69, 1991.

[5] D. Luebke and C. Georges, “Portals and mirrors: simple, fast evaluation of potentially visible set” April 1995 *Symposium on Interactive 3D Graphics,* pp. 105-106.

[6] J. Shade, D Lischinski, D. Salesin, T. DeRose and J. Snyder, “Hierarchical image caching for accelerated walkthroughs of complex environments”, *ACM SIGGRAPH ’96,* pp. 75-82.

[7] H. Zhang, D. Manocha, T. Hudson and K.E. Hoff, “Visibility culling using hierarchical occlusion maps”, *ACM SIGGRAPH ’97,* pp. 77-88.

[8] D. Aliaga and A. Lastra, “Architectural walkthroughs using portal textures”, *IEEE Visualization 97,* pp. 355-362

[9] N. Greene M. Kass and G. Miller “Hierarchical z-buffer visibility”, *Proceedings of SIGGRAPH ’93*, pp. 231-238.

[10] F. Brooks, “Walkthrough – a dynamic graphics system for simulating virtual buildings”, *Workshop on 3D Graphics 1986*, pp. 9-12.

[11] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, “Fast rendering of

complex environments using a spatial hierarchy", *Computer Interface '96.*

[12] T.A. Funkhouser and C.H. Sequin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments", *Proceedings of SIGGRAPH '93*, pp. 247-254.

[13] D. Kurmann and M. Engeli, "Modeling virtual space in architecture", *ACM Symposium on Virtual Reality Software and Technology 1996*, pp. 77-82.

[14] J. Torborg and J. Kajiya, "Talisman: commodity realtime 3D graphics for the PC", *Proceedings of SIGGRAPH '96*, pp. 353-363.

[15] M. Levoy and P. Hanrahan, "Light field rendering", *Proceedings of SIGGRAPH '96*, pp. 31-42.

[16] S. Gortler, R. Grzeszczuk, R Szeliski and M. Cohen, "The lumigraph", *Proceedings of SIGGRAPH '96*, pp. 43-54.

[17] Myszkowski, Karol and Kunii, "Texture Mapping as an Alternative for Meshing During Walkthrough Animation", *5th Eurographics Workshop on Rendering 1994,* pp. 375-388.

[18] R. Bastos, M. Goslin and H. Zhang, "Efficient Radiosity Rendering using Textures and Bicubic Reconstruction", *1997 Symposium on Interactive 3D Graphics*, pp. 71-74.

[19] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran and Paul Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping", *Proceedings of SIGGRAPH '92*, pp. 249-252.

[20] H. Hoppe, "View-dependent refinement of progressive meshes", *Proceedings of SIGGRAPH '97*, pp. 189-198.

BIOGRAPHIES

Bin Chan is a Ph.D. student in Department of Computer Science and Information Systems in the University of Hong Kong. He received B.Eng. and M.Phil. degrees at the the University of Hong Kong. His research interests include Virtual Reality and Computer Animation.

Contact information:

Mr. Bin Chan
Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong, China.
Phone: +852-28578457
Fax: +852-25598447
Email: mailto:bchan@csis.hku.hk

Wenping Wang is associate professor of computer science at the University of Hong Kong. He received B.Sc. and M.Eng. degrees in Computer Science from Shandong University, China, in 1983 and 1986, respectively. He received a Ph.D. in Computer Science from University of Alberta in 1992. His research interests include Computer Graphics, Geometric Modeling, and Computational Geometry.

Contact information:

Dr. Wenping Wang
Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong, China.
Phone: +852-28597074
Fax: +852-25598447
Email: mailto:wenping@csis.hku.hk

Mark Green is a Professor in the Computing Science Department at the University of Alberta. He is also the director of the Research Institute for Multimedia Systems (RIMS), a university-wide research institute that addresses all aspects of multimedia. Dr. Green received a Ph.D. degree in Computer Science from the University of Toronto, and has been active in the area of interactive computer graphics for over twenty years. Since 1988 he has been working on software tools and architectures for virtual reality. One of the main results of this work is the MR Toolkit, a collection of software tools for the production of VR applications that have been distributed to over 600 sites worldwide. Dr. Green was also one of the founders of the Art and Virtual Environments Project at the Banff Centre for the Arts, which has produced some of the most sophisticated artistic virtual environments to date.

Contact information:

Dr. Mark Green
Director, Research Institute for Multimedia Systems (RIMS)
Department of Computing Science
University of Alberta, Edmonton, Alberta, T6G 2H1, Canada
Phone: + (780) 492-4584
Fax: + (780) 492-4584
Email: mailto:mark@cs.ualberta.ca