Design of a Walkthrough System for Indoor Environments from Floor Plans

Bin Chan Hiu Ming Tse Wenping Wang

Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
[bchan,hmtse,wenping]@cs.hku.hk

Abstract

This paper is to share some experience in designing a practical walkthrough system for indoor environments. We discuss issues from the design of efficient utility programs for building 3D models to the design of a walkthrough engine. Our utility programs allow the creation of the interior of a several storied building in a few man-days. We also propose several rendering speedup techniques implemented in our walkthrough system. Tests have been carried out to bench mark the amount of speedup brought about by different techniques. One of these techniques, called dynamic visibility, proves to be more efficient than existing standard visility preprocessing methods.

1 Introduction

Walkthrough of architecture models finds applications in previewing a new building before it is built physically or in directory systems that guide visitors to reach a particular location. Two main problems in building a walkthrugh application are modeling and real-time display. That is, how the 3D model can easily be created, and how to display a complex architecture model at real-time rate.

Modeling involves the creation of a building and the placement of furniture and other decorative objects. Althrough most commercially available CAD software serves similar purposes, they are not flexible enough to allow us to create openings on walls, such as doors and windows. Since the input for building a walkthrough system are 2D floor plans in our case, 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 due to the following reasons. Firstly, the radiosity method is complicated to implement and time-consuming to apply, especially for our model consisiting of several floors of a large building. Secondly, 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 texture map is a possible solution, but that would then increase the demand on the limited and expensive texture memory on 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 the most important issues 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], runtime eye-to-object visibility, 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.

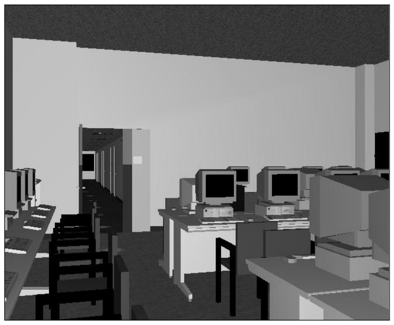

Figure 1

This system was written in C and OpenGL to allow maximum flexibility in incorporating new algorithmic improvement at the low level of geometric processing. The system can be divided into five parts, shown as follows:

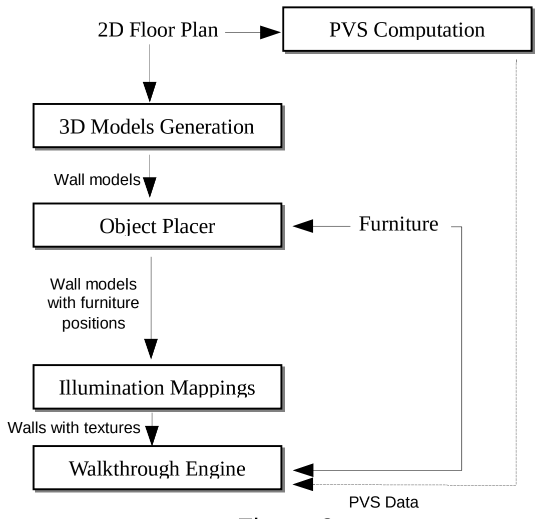

Figure 2

This paper is organized as follow. Section 2 is a brief survey of related existing work. Section 3 discusses some issues about modeling. Section 4 is about texture mapping for illumination. Section 5 describes the walkthrough engine and different speedup techniques. Section 6 presents the benchmark results. Section 7 contains the conclusion and comments on future research directions.

2 Related Work

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 environment. Greene, Kass and Miller[5] described a system using hierarchical *z*-buffer to quickly cull invisible polygons. However, since their algorithm requires special hardware, it is not yet practical on ordinary graphics workstations.

Recently, Zhang, Manocha, Hudson and Hoff [10] proposed a hierarchical occlusion map algorithm similar to the hierarchical *z*-buffer algorithm that requires no special hardware but hardware texture mapping only. It chooses some objects as occluders and tries to ignore those objects behind them. The method is efficient if there are many objects behind the occluders. That is, occluders should be sufficiently big to block many objects, but it is usually not easy to choose such occluders for indoor architecture models; the most effective occluders in such models are walls. But if walls are chosen as occluders, then everything in the room still needs to be displayed. In this case, the method is reduced to the PVS algorithm.

Another approach is image based rendering that uses texture mapping extensively. Texture mapping, generated at runtime, can be used to replace complicated geometry. These textures are usually not updated until adsolutely necessary. Systems like Microsoft's Talisman [9] are based this approach. However, its implementation again requires special image processing hardware.

Some other systems explore Level of Details (LOD) techniques for display speedup. In LOD different models exist for the same object, and are used 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

The data needed for building our models are floor plans and some height information like ceiling height, door height and windows positions. It begins with AutoCAD 2D DXF files of floor plans. First the files have to be cleaned up because there is some information, such as sewage pipes and electric wiring, that is not needed by the walkthrough system. Only the information about walls, doors and windows is extracted and stored in 2D DXF files.

The floor plans are divided in cells loosely based on the division of rooms. In order to speed up the subsequent operations like 3D extrusion, furniture placement and PVS calculations, rooms are not further subdivided. The subdivision process is done manually, since automatic algorithms, such as BSP, usually do not provide a subdivision which fits room partitions very well. Figure 3 shows the way a part of a floor is divided into cells.

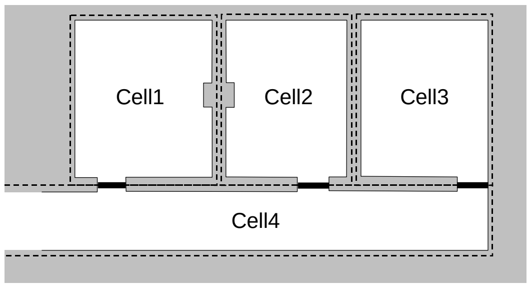

Figure 3

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 and later converted to the internal cell and portal database files.

3.1 2D to 3D Extrusion

The DXF files of the floor plan are used as the input to the 2D-to-3D converter. The converter first triangulates floors and ceilings; the ceilings have the same triangulation as the floors. The walls are truangulated in a special way. All the resulting triangles are right angle triangles. As shown in Figure 4.

The number of triangles in such a triangulation may not be minimum. However, there are two reasons for keeping all triangles being right angle triangles. First, the triangles thus generated can easily be combined into rectangles so quads drawing calls can be used, instead of triangles drawing calls. Note that one quad drawing call can save color calculations for two vertices when compared with two triangle drawing calls. Second, since texture mapping is used to simulate the illumination effect, irregular triangulation would affect the ease of specifying texture mapping. This will be explained in detail in the next section.

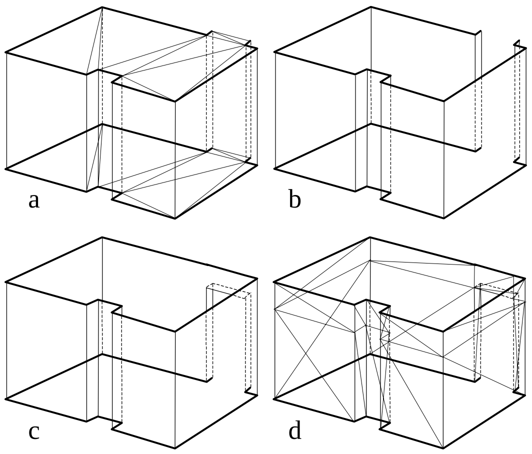

Figure 4

Figure 4 shows the 2D to 3D conversion. In figure 4a the ceiling and the floor are triangulated. In figure 4b, ceiling and floor are not shown, but only the wall directly extruded from the 2D boundary (The opening is the door position). Figure 4c is same as figure 4b except that the wall above the door is corrected. Figure 4d shows the final triangulation.

The resulting model is a connected surface with openings only on doors and windows. Note that there are no T joints between triangles on walls, except along the boundary between walls and floors or ceilings. Those T joints can be ignored since walls and floors or ceilings do not lie on the same plane, and they usually have different materials settings or texture mappings, so the problem of unmatched colors between adjacent triangles near those T joints is not an issue. The extrusion operation is done cell by cell since different cells may have different heights. The heights of ceilings, windows and door frames are entered manually since they are not available in the input 2D DXF files.

3.2 Objects Placement

After 3D wall models have been generated, they are passed to the object placing program, which is used to place funiture 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. The direction of the gravitation can be set to be along the three main axes to make it easy to align an object against the wall. After all objects have been placed, the object IDs and their transformation matrices are stored in a cell file.

4 Texture Mapping for Illumination

Gouraud shading is done by graphics hardware when displaying the 3D scene. To make the scene look more physically realistic, texture mapping is used to simulate two special and subtle illumination effects: soft shadows and the reflection of light sources on the floor.

4.1 Soft Shadows

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

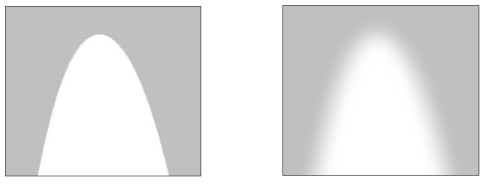

Figure 5

Figure 5a is the original texture map with a parabola shape. 5b is the gaussian blurred image of 5a.

The illumination texture is created by heavily blurring an image containing a parabolic shape. Figure 5b shows the texture map generated from Figure 5a.

In the case where several lights are situated close to each other, the bright regions of two adjacent lights may overlap each other. To handle this case two more textures are created as shown in Figure 6. There are altogether three different kinds of textures to simulate the most common illumination effects on a wall.

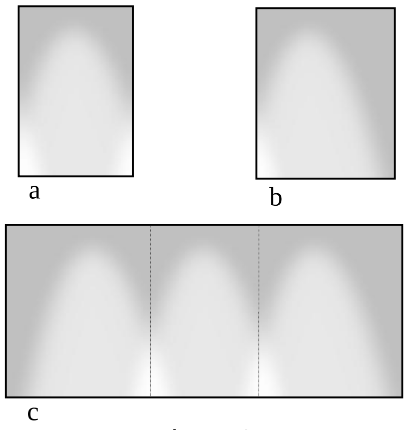

Figure 6

Figure 6a is the texture used for a light with other lights at both sides. 6b is the texture used when there is a light at one side. 6c is the effect of the combined texture for 3 closely situated lights.

The shadow texture placement is done automatically by analyzing the positions of lights placed by the object placer. Each light in a cell is checked to see whether there is a wall near enough; a distance threshold of one meter is set. If a wall is more than one meter away from the light, it is assumed that the light has no special illumination effect on that wall and only hardware Gouraud shading is used. If the distance is within one meter, the texture maps is then centered at the nearest point of that wall from the light. The vertical position is determined by the distance of the wall from the light. Usually a big wall formed by two big triangles may have more than one bright regions cast by multiple light sources. Since it is not possible to map more than one kind of texture on it or to map one texture at two or more arbitrary positions on the triangle, the big triangles must be subdivided. Figure 7 shows how the triangles are subdivided.

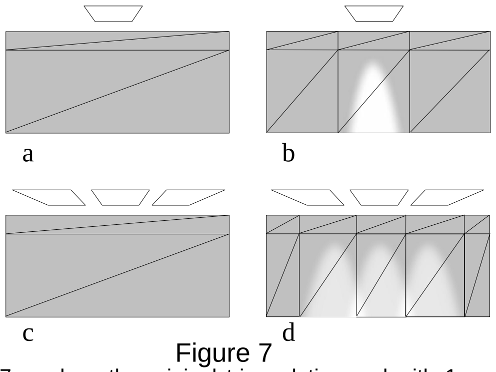

Figure 7

Figures 7a,c show the original triangulation and with 1 and 3 light sources respectively. Figures 7b,d show the triangulation for the illumination texture mapping for a and c, respectively

If the distance between two light sources is smaller than a preset threshold of 2 meters, the bright regions they cast on the wall are assumed to have some overlapping. Thus the subdivision is done as shown in Figure 7d. The boundary between two textures is chosen to be the equi-distance line on the wall next to the two adjacent light sources.

The main advantage of using texture mapping to simulate illumination effects as described above is that very few texture maps need to be prepared and stored in texture memory for a complex indoor enviroment. This is in contrast with the radiosity approach, in which different walls usually have different illumination distributions and so need different texture maps; this easily makes the number of different texture maps beyond the capability of texture memory. Our texture map approach also produces “reasonable” soft shadows in the sense that bright regions on a wall appear at positions where they are expected, though brightness and shapes of the regions are not as physically accurate as the results of the radiosity method.

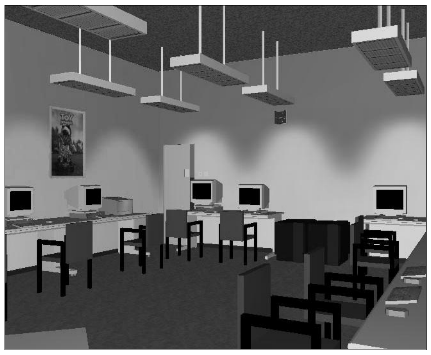

Figure 8

Figure 8 shows the effect of illumination by illumination textures mapping. Note that there are only three kinds of illumination textures.

4.2 The Reflection Map

Texture mapping is also used to simulate the effect of specular reflection of a semi-reflective rough floor. It is observed that such a floor usually produces reflections of bright objects and the reflections are so blurred that they look quite round. Since specular reflection moves with the user’s viewpoint, the texture cannot be directly mapped onto the model. Instead, a texture, called 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 ceiling.

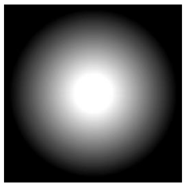

Figure 9

Figure 9 shows a reflection map's alpha component (white for opacity).

The reflection map used is a transparent texture map, with its alpha component shown in Figure 9 (white for opacity). The coordinates of light sources are used to determine the coordinates of the floating reflection maps, through some simple transformations.

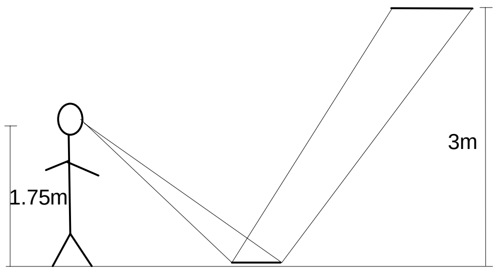

Figure 10

5 The Walkthrough Engine

The walkthrough engine refers to the program that displays the virtual environment in an interactive manner. A key to displaying a complex model in real-time is to quickly eliminate invisible polygons so to reduce the burden on graphics pipelines. In this section we discuss a few existing speedup techniques, as well as some new ones, implemented in the walkthrough system. To understand the performance gain of these speedup techniques, benchmarking tests have been carried out, and we will present the results in the next section.

5.1 Potential Visible Set Precomputation

We have used the standard PVS algorithm [4] as a primary speedup technique. This algorithm has two levels: cell-to-cell visibility and cell-to-object visibility. Cell-to-cell visibility records the cells that are potentially visible to a cell in which the viewer is in. Cell-to-object visibility further records the potential visible objects in those visible cells because usually not all of the objects in those visible cells are visible to the viewer. This method has been used successfully in many applications, and has also made a big difference in our system.

5.2 Runtime Eye-to-Object visibility

The pre-computed PVS data can help eliminate most of the triangles. However, there are objects that are not visible but are still rendered by the hardware after PVS processing (see Figure 11). That is because the PVS data is recorded at the cell level, the visibility at different positions in the same cell cannot be told from the PVS data. Thus, some runtime calculations are needed to find out which objects are not visible outside the cell. At runtime, the bounding boxes of objects from the precomputed potentially visible set are used to test against the 2D view frustum for tighter visibility computation. We note that some invisible objects are still not eliminated even after this step of eye-to-object visibility computation. See Figure 11

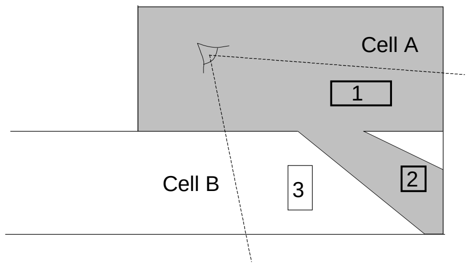

Figure 11

Figure 11 shows the case where some objects cannot be eliminated by the PVS information. Cell B is the adjacent cell of cell A, where the viewer is in. Cell B and all objects in it are marked potentially visible. Object 3 is reported to be potentially visible even after the view frustum test.

5.3 Dynamic Visibility Computation

Dynamic visibility computation refers to a new method for fast and tight visibility computation, which provides an alternative to the PVS algorithm. It is termed dynamic visibility because it computes cell-to-cell visibility information at runtime rather than offline as preprocessing. The advantages of this new method are: 1) it is not a preprocessing scheme, so does not consume extra memory for holding visibility information as required by the PVS algorithm; 2) it is easy to implement; 3) it yields tighter visibility information and leads to shorter overall rendering time than the PVS method. Nontheless, the dynamic visibility method is suitable only for indoor environments or similar densely occluded environments.

Dynamic visibility is computed by recursively checking the intersection between the portals and the view frustum. The view frustum will be narrowed down in new visible cells as the checking goes down the recursion.

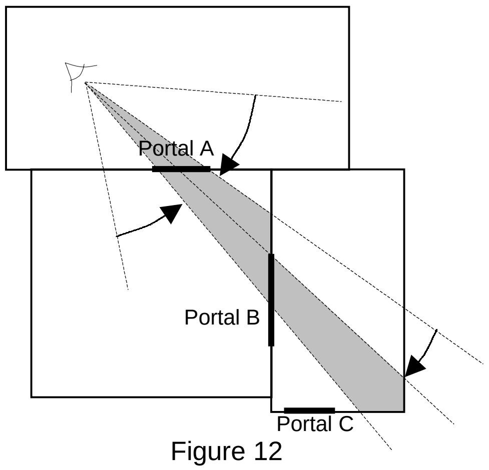

Figure 12

Figure 12 shows how to find the visible cells via portal A, by continuously narrowing the view angle through the portals, and stops when no more portal intersects the view region.

Testing the intersection of a portal with the view frustum can be treated as testing the intersection of two 2D sector regions with their apices both at the view point. Each test is extremely efficient, involving only four multiplications, two additions and two comparisons. The number of portals in one cell is usually not large, typically one to four. The number of portals in portal sequences is bounded by the maximum number of cells across a building. Therefore, the number of testings done per frame is not a big burden.

By comparing the sector formed from the view point to the bounding box of an object in an visible cell and the view frustum, we can determine the visibility of the object to the viewpoin. This test differs from the eye-to-object visibility mentioned in section 5.3 in the way that the number of objects tested in dynamic visibilty is less. Consider Figure 12. If objects from the PVS are to be tested for eye-to-object visibilty, then some objects in the cell below portal C will be tested. However, in the dynamic visibility method, objects in the cell below portal C are not considered in eye-to-cell testing step, because portal C is not visible to the current viewpoint.

The dynamic visibility method provides more accurate visibility information than the PVS algorithm because the former is done at runtime, and is fine tuned to return tighter visibility information with respect to the viewpoint. Since no pre-computation is involved, it can be used in dynamic virtual environments but must still be a densely occluded environment for maximum efficiency. Without the need for large memory as required by the PVS data, the dynamic visibility method is more suitable for walkthrough system on the web.

5.4 Software Vertex Color Calculations

Conventionally, in an 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 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, often up to six, triangles. When adjacent triangles cannot be specially arranged to take the 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. But they are computedonly once by walkthrough engine per display cycle.

To make the software computation of vertex color more efficient, we choose to compute the diffuse reflection only, and this assumption turns out not to adversely affect the illumination result as might be expected, since most indoor objects consist of predominantly diffuse surfaces. Consider the process of using graphics lighting hardware that handles diffuse and specular reflection and many other lighting effects. Even when the specular terms of most objects are set to zero, 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 the limitation by OpenGL implementation; OpenGL supports only up to 8 light sources, for instance.

One extra benefit of the software vertex color calculation is efficient backface culling. The dot product of the view vector and a polygon normal must be found in calculating its vertex color. This piece of information is used directly to determine the visibility of that polygon. This further releases graphics hardware of the work on backface culling.

6 Benchmark Results

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. The whole model, including furniture, consists of 93628 triangles. The platform used for running the walkthrough system is an SGI Maximum IMPACT workstation with R10000 CPU, 192MB main memory and 4MB texture memory. The table below shows the average frame rate and the number of triangles actually rendered during a walkthrough of a guided path with 1409 frames, using different combinations of speedup techniques.

Guided Tour	Benchmark results (1409 frames)									
PVS Cell to Cell	No	Yes	No	Yes	Yes	Yes	No	Yes	No	Yes
PVS Cell to objects	No	No	No	Yes	No	Yes	No	No	Yes	Yes
Eye to objects	No	No	Yes	No	Yes	Yes	X	X	X	X
Dynamic Visibility	No	No	No	No	No	No	Yes	Yes	Yes	Yes
Average # of cells	11.79	8.50	8.58	8.50	7.75	7.75	5.87	5.87	5.87	5.87
Average # of objects	168.78	118.21	42.29	49.69	38.77	33.78	28.61	28.61	28.61	28.61
Average # of primitives	13326.28	9273.20	3714.11	3905.07	3369.58	2780.27	2266.96	2266.96	2266.96	2266.96
Average # of textures	415.77	310.99	226.12	235.24	208.74	206.65	165.10	165.10	165.10	165.10
Average time in ms	**330359**	**251193**	**153384**	**153811**	**144558**	**130608**	**109060**	**108999**	**108744**	**108821**
Average fps	**4.27**	**5.61**	**9.19**	**9.16**	**9.75**	**10.79**	**12.92**	**12.93**	**12.96**	**12.95**
Speedup	**0%**	**32%**	**115%**	**115%**	**129%**	**153%**	**203%**	**203%**	**204%**	**204%**

X means not care. Dynamic visibility

Table 1 shows the comparison between the various polygon culling techniques under the worst condition, that is, all doors are opened. The average numbers are average numbers per frame. The average number of objects is the smallest when the dynamic visibility is turned on. These comparisons are done without backface culling, using OpenGL lighting.

Guided Tour	Benchmark results (1409 frames)									
PVS Cell to Cell	No	Yes	No	Yes	Yes	Yes	No	Yes	No	Yes
PVS Cell to objects	No	No	No	Yes	No	Yes	No	No	Yes	Yes
Eye to objects	No	No	Yes	No	Yes	Yes	X	X	X	X
Dynamic Visibility	No	No	No	No	No	No	Yes	Yes	Yes	Yes
Average # of cells	11.79	2.10	8.58	2.10	2.05	2.05	1.81	1.81	1.81	1.81
Average # of objects	168.78	13.00	42.33	12.23	11.97	11.95	11.63	11.63	11.63	11.63
Average # of primitives	13326.28	1037.06	3717.62	949.89	940.79	939.17	912.60	912.60	912.60	912.60
Average # of textures	415.77	69.53	226.12	68.32	66.98	66.96	64.47	64.47	64.47	64.47
Average time in ms	**330359**	**60501**	**153092**	**58535**	**58326**	**58317**	**56885**	**56880**	**56812**	**56819**
Average fps	**4.27**	**23.29**	**9.20**	**24.07**	**24.16**	**24.16**	**24.77**	**24.77**	**24.80**	**24.80**
Speedup	**0%**	**446%**	**116%**	**464%**	**466%**	**466%**	**481%**	**481%**	**481%**	**481%**

X means not care. Dynamic visibility

Table 2 shows the comparison between the various polygon culling techniques under the best condition, that is, most of the doors are closed. The average numbers are average numbers per frame. The average number of objects is the smallest when the dynamic visibility is turned on. These comparisons are done without backface culling, using OpenGL lighting.

Guided Tour	Benchmark results (1409 frames)						
Quad/Tri/Tri-strip	Tri	Quad	Tri-strip	Tri	Tri	Tri	Tri
Lighting	Hardware	Hardware	Hardware	Software	Hardware	Hardware	Hardware
Pre-transform	Hardware	Hardware	Hardware	Hardware	Software	Hardware	Hardware
Back face culling	No	No	No	No	No	OpenGL	Software
Average # of cells	48.00	48.00	48.00	48.00	48.00	48.00	48.00
Average # of objects	1231.00	1231.00	1231.00	1231.00	1231.00	1231.00	1231.00
Average # of primitives	93628.00	54320.00	54320.00	93628.00	93628.00	N/A	46153.45
Average # of textures	2197.00	1146.00	1146.00	2197.00	2197.00	N/A	1754.21
Average time in ms	**1243416**	**798527**	**787947**	**654927**	**1195299**	**1223720**	**668355**
Average fps	**1.13**	**1.76**	**1.79**	**2.15**	**1.18**	**1.15**	**2.11**
Speedup	**0%**	**56%**	**58%**	**90%**	**4%**	**2%**	**86%**

Table 3 shows the comparison between software and hardware approaches under the worst condition, that is, no PVS, no dynamic visibility, not even view frustum culling. Note that OpenGL backface culling is not very efficient. This may be due to the fact that a triangle cannot be detected to be front or back facing in OpenGL until the third vertex is passed to it. By the time the third vertex is passed, the graphics pipeline has already started to process the first two vertices' information. If that is a back face, the pipeline still has to be flushed.

Guided Tour #2	Slowest	Fastest for the best case	Fastest for the worst case
PVS Cell to cell	No	Yes	Yes
PVS Cell to objects	No	Yes	Yes
Eye to objects	No	X	X
Dynamic Visibility	No	Yes	Yes
Quad/Tri/Tri-strip	Tri	Tri-strip	Tri-strip
Lighting ***	Hardware	Software	Software
Pre-Transform	Hardware	Software	Software
Back face culling	No	Software	Software
All portals status	Open	Closed	Opened
Average # of cells	11.79	1.81	5.87
Average # of objects	168.78	11.63	28.61
Average # of primitives	13326.28	321.51	715.37
Average # of textures	415.77	33.48	84.41
Average time in ms	**330359**	**41220**	**62107**
Average fps	**4.27**	**34.18**	**22.69**
Speedup	**0%**	**701%**	**432%**

Table 4 shows the fastest speedups taken for the best and the worst cases, respectively.

With the benchmark results, we try to find the relative proportions of time that different processes take. The following assumptions are made:

1. The total time can be divided into two parts: calculation time and rendering time.
2. The rendering time is proportional to the number of primitives rendered.
3. When no speedup technique is used, the time taken is the pure rendering time. That is, there is no calculation time. The time in column 1 of Table 3 is therefore the pure hardware rendering time of 1409 frames of 93628 triangles.

From Table 3 we find that the speedup by using software lighting is about 90%. The hardware uses at least 47% of the time to do the lighting and the rest is for rasterization and texture mapping, etc. The fact that hardware takes such a long time to compute the lighting has not been expected, but this agrees with the data in columns 2 and 3 of Table 3 where quad and triangle strips are used. Quads and triangle strips drawing can save one third of the vertex color calculations; there are less primitives to draw if quads or triangle strips are used instead of triangles. This results in less graphics pipeline startup and shut down, and hence less overhead. Table 1 and 2 show that on average the dynamic visibility method outperform the PVS in all situations.

With the combination of various speedup techniques, our system can achieve up to 701% speedup. The average frame rate can reach 34.18 frames per second, compared with 4.27 frames with no speedup technique being used.

7 Conclusion and Future Work

We have described a practical walkthrough system of architecture model and discussed some issues in modeling, illuminations and display speedup techniques. The paper also gives a benchmark of performance gains of different techniques for display speedup.

The new potential visibility determination we use can cull on average 98% of the polygons of the environment. We observe that now the efficiency of hardware rendering is no longer the most important issue that stops us from further improving the frame rate. The frame rate is now limited by the speed of polygon culling techniques and software lighting. Further eliminating polygons seems to have no more effect on improving the frame rate

Furtherwork is to be done on fast polygon culling techniques. PVS is a good method in that it requires no runtime computation but its culling is not tight enough. Some refinement like subdivision of cells into smaller cells may help increase its accuracy.

Dynamic visibility computation is the approach at the other extreme. It tends to uses more runtime computation than the PVS approach. Althrough its accuracy makes it out-perform the PVS, the amount of computation required is a hindrance to further increasing the frame rate, especially when the number of objects is huge. From the data shown in Table 2, the frame rate is around 24 even if there are less than one thousand triangles to render per frame. This is much below the maximum capability of our graphics hardware. Therefore, much of the time has been used on computation by CPU and this keeps graphics hardware idling and waiting for CPU to complete visibility computation, rather than rendering more polygons. More work should be done to increase the speed of dynamic visibility computation.

Acknowledgments

We wouldlike to thank K.S.Cheng, S.M. Lee, Y. Cheung, S.K. Ng, C.Y. Lau, K.L. Ko and M.L. Lam for the modeling and coding of the system. We are grateful to the Estate Office of the University of Hong Kong for providing us with the floor plans of Chow Yei Ching Building.

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] 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] N. Greene M. Kass and G. Miller “Hierarchical z-buffer visibility”, *Computer Graphics Proceedings* August 1993, pp.231-238.

[6] N. Greene, “Hierarchical polygon tiling with coverage masks”, *ACM SIGGRAPH ’96*, pp.65-74.

[7] 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.

[8] 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.

[9] J. Torborg and J. Kajiya, “Talisman: commodity realtime 3D graphics for the PC”, *ACM SIGGRAPH ’97*, pp.353-363.

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