A virtual walkthrough system for complex indoor environments

Chan Bin

A thesis submitted for the Degree of Master of Philosophy

Department of Computer Science
The University of Hong Kong
August 1998

Abstract of thesis titled "A virtual walkthrough system for complex indoor environments"
Submitted by Chan Bin
for the Degree of Master of Philosophy
at the University of Hong Kong in August 1998

We have built a virtual walkthrough system for indoor environment. Our system has the following characteristics: it is designed for mid-range to low-end graphics workstations, which usually have fast CPU but slow graphics board. Reasonably realistic illumination effects are implemented by using a small number of texture maps rather than accurate illumination since accurate illumination is hard to represent in low-end workstations. The system involves no pre-computation so it supports dynamic environments. The speedup techniques used are efficient and simple so the walkthrough engine is easy to implement and can be rebuilt easily on any platform.

In this thesis, we talk about many issues in building such a system, from the creation of models, placement of objects, to the design of the walkthrough engine. The emphasis is on speed-up techniques. We also consider geometric modeling and illumination modeling issues. Among the speedup techniques, two methods are of most importance: software lighting and a dynamic visibility method.

In software lighting, the vertex colors of a polygon (diffuse term only) are computed by the walkthrough engine rather than the graphics hardware and the graphics hardware's lighting function is totally turned off. We intentionally offload some of the graphics hardware's work to CPU because of a few reasons: 1) there are many repeated vertices; 2) only the diffuse term is used in most situations for indoor environments; 3) by carefully placing light sources, we can get back-face culling information as a side product of the process; 4) we notice that the recent advancement of graphics hardware tends to be slower then that of CPU.

The dynamic visibility method means the visibility information is calculated at runtime instead of by pre-computation. The benefit of doing so is that we can get more accurate culling so fewer polygons are eventually rendered and therefore dynamic environments can be supported, which is very important to some applications like building preview with modification functions. Compared with some pre-computation methods, such as the Potential Visibility Set method, our method provides more accurate culling and the save in time for rendering fewer polygons justifies the time used for online culling. Besides the two methods mentioned, there are some others techniques used in our system: texture grouping, which is for reducing I/O traffic between main memory and texture memory in graphics board and extensive use of quadrilaterals, instead of triangles, to reduce the total overhead in rendering startup.

Table of Contents

1 Introduction

Stepping into the 90s, the term "Virtual Reality" has become more and more popular. Virtual Reality is a technology that can immerse the user into a virtual three-dimensional world, with which the user can directly interact using 3D input devices like a 3D mouse and glove. Interactiveness, head motion parallax and stereoscopic view are the basic factors for the immersed feeling. Virtual Reality technology is in fact a combination of different technologies: computer graphics, 3D input technology like position sensing and output like head mount display technology. There are many kinds of VR systems, distinguished by different combinations of input and output devices. For instance, totally immersed VR uses head mount display, glove and voice recognition for spoken commands. Screen based VR does not rely on head mount display but monitors. No matter which kind of VR, computer graphics is still the key and driving technology behind. In recent years, the emergence of high-speed 3D graphics chips have made the VR technology more and more popular as witnessed in the computer games industry. However, true 3D immersion has never been popular due to a few factors. First is the lack of inexpensive and accurate input devices. The noise and latency in current 3D sensors is too annoying for real applications and their effective range is too limited. Second is the lack of affordable and high quality head mount display. Currently, most of the head mount display suffers from a few common problems, such as very narrow view angles, low resolution and unbearable weight. One variation of VR we want to talk about here is the screen based VR. The monitor is used as the output device instead of head mount display. Usually it is used with shutter glasses to provide stereoscopic views. Although it cannot provide total immersed feeling, as with head mount display, it is capable of providing much higher resolution, richer color compared with those LCD head mount display, and most importantly: less weight on the head. Monitor is a well-developed device and is more affordable. That is why most of the new VR computer games, such as those car racing or shooting games, use screen based VR. It is beyond our scope to talk about computer games here but usually we can tell from the games industry how new technologies are developed to the commercial level. The importance is that a new technology becomes useful only when it is commercially available.

As mentioned, computer graphics technology is the most important factor in a VR system. It directly affects the usefulness of the system itself. Here, we will mainly talk about the computer graphics issues rather than input and output devices. The biggest issue in computer graphics for VR is speed. Frame rate of at least 10 to 20 frames per second is essential or a user may get sick of the jumpy motion. The time for rendering a frame is directly proportional to the number of polygons. The larger is the models, the longer is the rendering time and hence lower frame rate. The models used in architecture walkthrough are usually very complex, with polygon count from a few tens of thousands to millions. Even the most powerful graphics workstation in the world may have difficulties in rendering a simple architecture model 30 time per second by brute force. Some speed up techniques must be used in order to achieve the interactive rate. These techniques can be divided into two kinds. The first kind is to reduce the number of polygons rendered. This is based on the observation that typically only a small portion of polygons in an environment are visible to a particular eyepoint at the same time. In other words, most of the polygons are hidden or too far away to be visible. Although

there is z-buffer hidden surface removal algorithm to accurately remove hidden surfaces, it is too slow for environments of high depth complexity. Some other techniques are needed to quickly cull unnecessary polygons. These techniques include visibility pre-computation, level of details and fast polygon culling techniques. Usually, over 90% of the polygons for indoor environments can be eliminated. The second kind is image-based approach, which relies on images to represent object details. Texture mapping is the classic method of such an approach. Instead of using polygon meshes to represent surface details, images are used so polygon count is greatly reduced. Sometimes, geometric models can even be dropped, as in purely image based VR. Views at different positions and directions are pre-rendered or photographed and stored. Frames in the walkthrough are generated by blending or morphing the pre-rendered images. In image-based VR the display speed does not depend on the complexity of the environment. It only depends on the size of the window. However, it needs huge amount of storage space. Many gigabytes of memory are needed for walkthrough of a medium size environment in a window as small as 256x256 pixel [11]. There is a real example of image-based VR application. In Japan, there are electronic museums. All the exhibits are shown on HDTV with a trackball like device to rotate the viewpoint. All the exhibits are photographed at different angles, some even at 0.2-degree increment to allow smooth rotation. Image-based VR is practical in this situation because the environment is simple and the viewer's path is limited and fixed. In general, image-based VR is not very practical yet but it is a very promising technique. Another drawback of such systems is that only static scenes are supported. Hybrid approaches that combine both polygon-reducing techniques and image based techniques are also possible. One way is to replace objects far away from the viewpoint by images but render objects near the viewpoint. This can reduce the memory needed for storing too many images and still greatly reduces the number of polygons [1, 8].

1.1 Visibility

Hidden surface removal algorithms are essential in 3D rendering. The z-buffer algorithm is the most commonly used for hidden surface removal and is implemented in hardware in many graphics workstations. However, this algorithm is not very efficient for environments with high depth complexity. This is largely due to its pixel level nature, since only screen space coherence is explored in this algorithm and no object space information is used. The problem is that every pixel of a polygon has to be tested for visibility even if it is entirely invisible. So most of the time is wasted in testing such polygons in a highly occluded environment. Another factor is that graphics pipeline startup and shutdown is exceptionally time consuming. The pipeline goes through one cycle whenever a polygon is drawn. That is why the time needed for rendering a fixed number of big and small polygons does not differ vary much with the same size of polygons. That also explains why the rendering time is roughly proportional to the number of polygons but not the total area of polygons rendered. Since the z-buffer algorithm is implemented at pixel level, it cannot tell whether a polygon is visible before the graphics pipeline is started. In the graphics pipeline, every polygon is rendered no matter it is visible or not. No polygon is actually saved. That is why brute force rendering of complex environments is extremely slow.

Polygon culling techniques, on the other hands, take object space information into account. There are many polygon-culling techniques. Some classic methods, such as backface culling and view frustum

culling, are very common and are implemented in almost every system. Some recent methods, such as hierarchically grouping of nearby objects [5] and Potential Visible Set pre-computation [4], are common too. Most of these methods extract spatial coherence of an environment, some by pre-computation and some using texture assisted methods. These methods are very useful for environments with high depth complexity or indoor environments. Objects in the same room are highly related with each other. Whenever one object is invisible, the other objects in the same room are very likely to be invisible too. Polygon culling algorithms usually cull polygons at the object level so a large number of polygons can be eliminated in one step. Since no graphics pipeline startup is needed for those invisible polygons, the overall rendering efficiency is increased. A polygon culling technique must be used together with the z-buffer algorithm because it is not an accurately hidden surface removal algorithm by itself and it does not provide any depth information of polygons to the graphics hardware to do the correct rendering. It can quickly eliminate a large number of invisible polygons that the z-buffer algorithm cannot eliminate fast enough. In terms of polygon count, an ideal culling algorithm is one that returns only the polygons that are eventually visible, i.e. the optimal set, which are usually a few percent of the total number of polygons for indoor environments. However, if we take the total time needed for each frame into account, the situation is a bit different. A polygon culling algorithm is usually more computationally expensive than brute force rendering. It is because some calculation is needed to determine what can be eliminated. Even if the computation is done offline like in PVS, extra time is needed to scan through the database to determine what to render. The computation needed differs from algorithm to algorithm but there is a general behavior that the more polygons it culls, the more computational time it needs but the less rendering time is required. Thus, there is a trade off between the computational time and rendering time. On one hand, the computational time for polygon culling should not be longer than the rendering time, for otherwise, the CPU just cannot feed data fast enough to the graphics board and the graphics board will keep waiting so resources are wasted. On the other hand, the polygon culling algorithm should be efficient enough to cull polygons and return a set of potential visible polygons which is tight enough to make the rendering time needed no longer than the culling time. Different culling techniques cull polygons in different ways so it is common that several techniques are used together to provide a better culling percentage. Usually, these techniques are applied sequentially in a pipeline style. In general, the first stage is a fast and rough method. The subsequent stages are more accurate and usually take longer time but cull much less polygons. For instance, PVS culling is carried out before view frustum culling and then followed by back face culling.

1.2 Geometric Simplification

Level of details (LOD) is another technique that is widely used to reduce the number of polygons. The rationale is to use simpler models for objects far away from the viewpoint. It is because we cannot see objects clearly if they are too far away. For instance, a chair a thousand meters away will be projected to the same pixel on screen no matter what it looks like, so why bother to use the full detailed model to waste the graphics pipeline's time? There are many LOD algorithms. All of them use some kinds of metrics for measuring the importance of a particular model to the screen. Some use distance, while some others use projected area. Some algorithms generate several distinct models

offline, while some generate continuous models. Most of them work well for connected, well-defined polygon meshes. However, many practical models are highly disjoint with a lot of parts and different material properties. Most of the algorithms fail to handle models with different materials. The situation is worse if there are textures mapped on the models. LOD seems to be a useful technique but there are still no standard methods.

1.3 Realism

Another issue is about realism. VR is a technique about immersion. If the frames generated look too artificial, say flat shading everywhere, the immersion feeling will be affected. So textures are used extensively to increase the fidelity of the rendered images. Texture mapping is very useful in representing surface details by pasting images on polygons. It is an image-based technique used to highly reduce polygons needed. It is just not possible to represent surface details by polygons for objects like a piece of carpet. Millions of polygon would be needed if the surface details of a carpet were model geometrically. Texture mapping hardware is common nowadays. They even appear in low-end PC 3D graphics accelerators. Besides surface details, illuminations are essential to the realism of the rendered image. One of the main illumination effects in an indoor environment is the bright region on a wall cast by light sources on the ceiling. Shadows represent another important effect. There are many ways of generating shadows. In ray tracing, shadow is generated very easily but ray tracing is too slow to be used in interactive systems and the shadows it generates are too sharp to look real. Radiosity method, on the other hand, can produce more realistic soft shadows. Although radiosity method is more computational intensive than ray tracing, it can be used as a pre-computation process, so no extra computation is needed in rendering. The shadows generated can either be represented by polygon mesh or textures. However, the large number of polygons or textures it generates may be a problem for complex environments. The time needed for the computation of radiosity is also a critical factor for complex environments, since it may take hours or days to process a floor of an architecture model.

2 Related Works

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. The PVS is generated offline. The information generated has two types: cell-to-cell visibility and cell-to-object visibility. The algorithm first divides the environments into cells, roughly following the division of rooms and walls. The cells are then scanned one by one. All other visible cells and objects to that cell are recorded. In rendering, the visible cells and objects to the cell containing the viewpoint are extracted from the PVS database and only these visible cells and objects are rendered. By applying the PVS method to our environment, usually over 70% of the objects can be eliminated.

Greene, Kass and Miller [5] described a system using a hierarchical z-buffer to quickly cull invisible polygons. This is a hidden surface removal algorithm which takes the object space coherence into account. The object space is divided in an octree manner. When checking the visibility of an object, the faces of the bounding cube are checked first. If they are not visible, the objects inside the cube must be invisible. Otherwise, the checking will recursively go down the octree. To quickly check the visibility of bounding cubes, a multi-resolution z-buffer or hierarchical z-buffer is maintained which

requires a special piece of hardware for it to be efficient. Hence this method 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. Their algorithm requires no special hardware but hardware texture mapping only. 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 by using the texture mapping hardware. 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 simply reduced to the PVS algorithm.

Aliaga and Lastra [13] proposed a system that uses textures to cover portals. Multiple views through each portal at different directions are rendered either offline or at runtime. In 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. This involves a transition from frames using portal texture and frames not using it. For the transition to be less noticeable, more than one texture per portal should be used. The textures used for this purpose cannot be too small or the portal will look too blurry to be realistic. The textures used in their system is of size 256x256 with 8 bit per color component, i.e. 256KB per texture. For machines with limited texture memory, a few such textures used at the same time could easily overload the texture memory, not including other textures required for the models.

Microsoft's Talisman [9] is a system for real time animation. Texture maps, generated at runtime, are used to replace complicated geometry. The real 3D objects are not rendered until absolutely necessary. The system utilizes frame to frame temporal coherence by transforming the resulting images. 2D transformations of the rendered images are used to simulate 3D transformations. However, its implementation involves many image processing steps like image compressing and image warping, so special hardware is used. The system cannot be implemented on ordinary graphics workstation yet.

Apple's Quicktime VR uses an image-based approach. First, a panorama of an indoor environment is constructed from photos taken at different angles or by image synthesis like ray tracing with a cylindrical view plane. This panorama, which is a wide image, is then used to render the scene of walkthrough. A simple model must be defined to tell the system how the panorama image should be warped. This is a very handy way of construction walkthrough systems. However, a user cannot move freely in the environment, and he can only examine objects from one side because the other side is not defined in the panorama. If a user moves too far away from the default position, the scene will be distorted and blurred very much due to over warping of the panorama.

Levoy, Hanrahan [11], Gortler, Grzeszczuk, Szeliski and Cohen [12], two groups from Stanford University and Microsoft, respectively, proposed recently similar systems, called Light Field Rendering or Lumigraph, that do not rely on geometry models. Flow of light in space is represent by 4D functions. Plane pairs are used to define the 4D functions. Each entry of the 4D function corresponds to an image. 2D slices are extracted from the 4D functions as the view at different angles

and positions. The major problem of this approach is that the storage of 4D functions consumes too much memory so compression is crucial to the algorithm. However, even with compression, display cannot be too large, usually smaller than 256x256 pixels. Another drawback of these kinds of methods is that they can only handle static environments.

3 Overview

The thesis is organized into a few sections. Section 3 is an overview of the system; section 4 is about modeling; section 5 is about object placement; section 6 is about illumination; section 7 is about the walkthrough engine and speedup techniques; section 8 presents some results; section 9 is about the applications of the system and section 10 contains conclusions.

3.1 System Goals

Some of the intended applications of the system are a computer directory system for visitors of a building, virtual tour of buildings through the Internet and preview of to-be-constructed buildings. There are a few goals of the system:

1) Designed for mid-range to low-end graphics workstations, because we cannot expect a computer used in a directory system to be a high-end graphics workstation, which costs hundreds of thousands dollars, and most of the Internet users do not have such computers. To this end, we intentionally shift some of the graphics hardware's workload to the CPU because we believe that the CPU's evolution is much faster then the graphics hardware, especially for PC workstations. Moreover, it is much easier and cheaper to scale up CPU performance by using more processors than adding graphics pipelines. Now, we have PCs with two to four CPUs with just hundreds of US dollars per additional CPU, but it would cost thousands to tens of thousands dollars for any additional graphics pipeline and sometimes there is no option at all.
2) Implement reasonably realistic illumination effects rather than accurate illumination since accurate illumination is hard to represent in low-end workstations. There should be a good balance between interactive performance and realism.
3) The support for dynamic environments is essential in applications like building preview since the architects might need to move the objects in the environments or even adjust the position of walls.
4) Quick creation of new floors is also a goal of the system. We have created some automation tools, such as floor plan extruder, and auto texture mapper, to automatically create the wall models from floor plans. From our experience with our tools, a new floor with about fifty rooms needs only about a few man-days to construct.
5) The speedup techniques used should be as simple as possible so that the walkthrough engine can be made simpler and smaller and can be rebuilt easily on any platform.
6) There should be no extra data except the essential models, textures and cell division information needed for the walkthrough engine to function, so to reduce the data downloading time for applications across the Internet, whose network bandwidth is limited for most users.

3.2 System Overview

The virtual environment we built consists of a few floors of a building housing the Department of Computer Science and Information Systems at the University of Hong Kong. Figure 1 shows a typical

scene captured from the walkthrough engine. Typically, each floor consists of about 100,000 triangles.

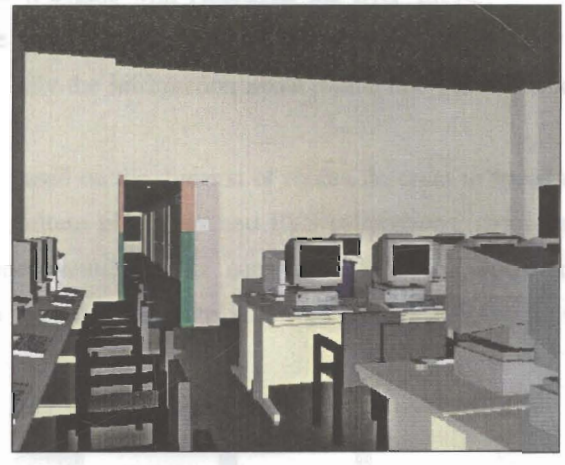


Figure 1 A typical scene captured from the walkthrough engine. It shows a view inside a computer lab.

The system is divided into four steps. The first one is the generation of 3D models, which is a step that extrudes the floor plans into 3D wall models. It also adds door and window openings. Textures of floor and ceiling are added in this step too. This step is done cell by cell and the resulting wall models are represented by many different model files. The next step is object placement where objects from an object library are selected and placed inside the wall models. This step is done manually since no information about object positions is automatically available. The third step is illumination mapping that analyses the positions of light sources placed in the previous step and maps illumination textures to the bright regions on the walls. The last step is the walkthrough engine which is responsible for the rendering of the scene using several speedup techniques. The PVS computation step is not necessary and is only implemented for comparison with our new visibility algorithm.

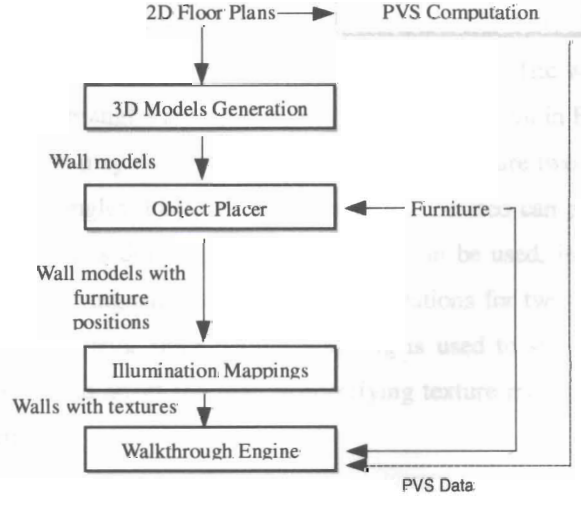


Figure 2 shows the organization of the system. Note that the PVS computation step is included just for comparison and is not actually used in the final system.

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 in figure 2.

4 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 some information about sewage pipes and electric wiring 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 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.

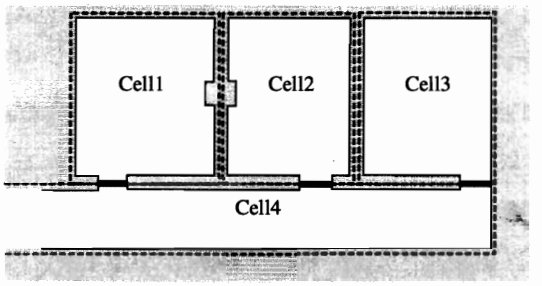


Figure 3 shows the cutting of cells. It loosely follows the division of rooms. The floor plan is then clipped to each cell before extrusion. The cell boundaries are set to touch each other if there is a portal between them so after clipping, the clipped floor plans of adjacent cells touch each other with no gap between them.

All line segments intersecting the cell boundaries are trimmed so line segments 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.

4.1 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 triangulated 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 more efficient quads drawing calls in OpenGL can be used, instead of triangle drawing calls. Note that one quad drawing call can save color calculations for two vertices, 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.

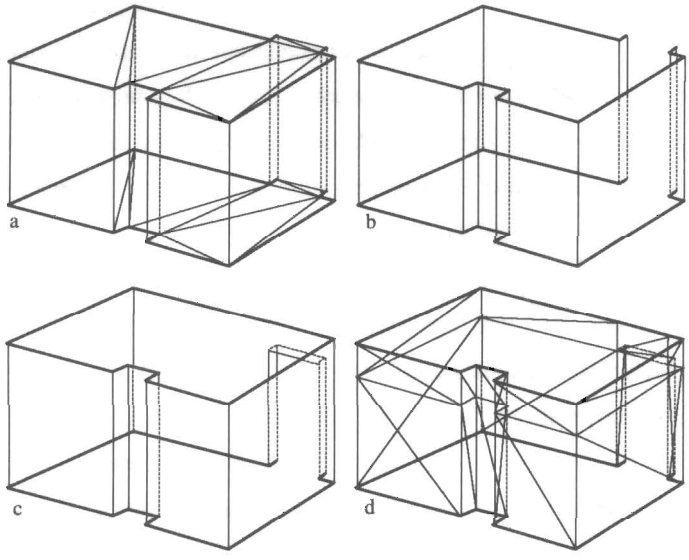


Figure 4 shows the 2D to 3D conversion. In figure 4a the ceiling and the floor are triangulated. In figure 4b, ceiling and floor are not shown, but only the wall directly extruded from the 2D boundary (The opening is the door position). Figure 4c is same as figure 4b except that the wall above the door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

4.2 Texture Maps Processing

Texture maps for the ceiling and the floor were obtained by photographing the real scene. However, photographed images cannot be used directly as textures due to a few problems. They usually do not give seamless joint and they produce repeated patterns when the images repeat themselves. In choosing textures, it should be noted that those images with outstanding pattern should be avoided. We choose one with the least global pattern, then use some image retouching tools to remove the global pattern. A simple process as shown in figure 5 is used to blend the edges. This process can create self repeating textures with no noticeable edges. However, it cannot remove the periodical pattern. There is a method that, by analyzing the textures at multiresolution [14], can produce excellent self-repeating texture. This method is not implemented because of its complexity, which involves some frequency domain analysis. We prefer our simpler way because it can be carried out in some commercial image retouching tool like Adobe Photoshop.

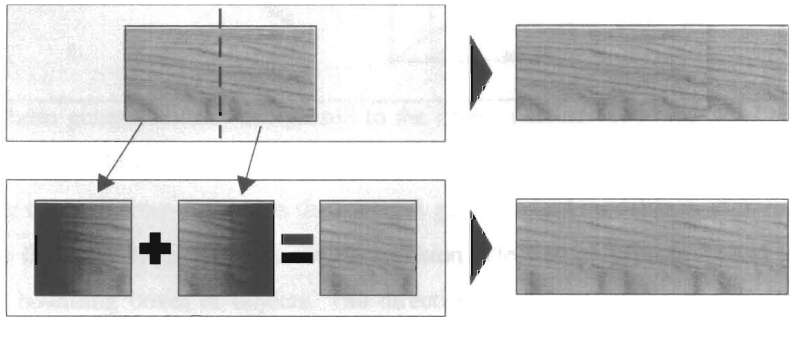


Figure 5 shows the procedures used to make an image becomes self-repeating

Most of the textures are made from photographs. Normally, the camera is positioned directly in front of a target. Flash light is usually used for indoor environments but it may cause problems for shiny surfaces. If possible, flash light should not point to the target in the same direction of the camera. That is because the high intensity of the flash light will make the specular reflection of the surface boom the whole scene and result in a high-contrast photo which cannot be used. In some situations, when shooting the photograph of the wall at the end of a long corridor or the flash light is not detachable, for instance, the flash light cannot be placed in a direction far away from the camera direction. In these cases, the photo should be taken at an inclined direction. The target will then look perspectively distorted in the photo, like that shown in figure 6 and figure 7. The images are adjusted after scanning. By measuring the positions and size of the target, the position of the camera and the camera's view angle, a rough estimate of the projection matrix can be found. The target is rendered under such projection and we can get a shape similar to the one on the image. The viewpoint's position is fine tuned interactively to match the shape of the image and a more accurate projection matrix results. The distorted image is re-sampled using the projection matrix to get the correct image.

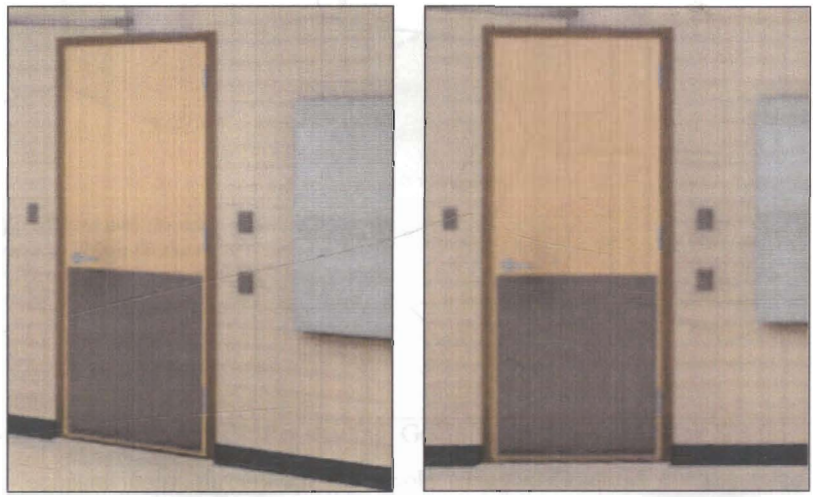


Figure 6 shows the textures before and after processing.

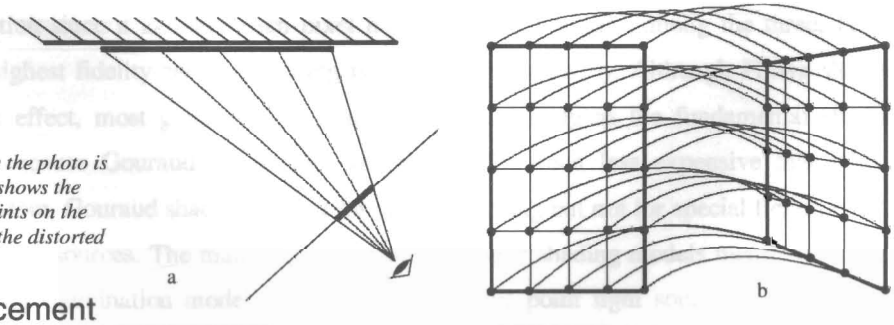


Figure 7a shows why the photo is distorted. Figure 7b shows the relations between points on the restored texture and the distorted

5 Objects Placement

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. The direction of the gravitation can be set

freely to make it easy to align an object against the wall. The collision detection algorithm used here is very simple, as only collision between rectangular blocks are tested. No complicated algorithm is used because speed is not the main concern and there are no complicated models such as curved surfaces. The only function we need is to align an object's bounding box with a plane. With this function, most of the object placement can be achieved. The only difficulty is to put an object inside another object; for instance, putting a book inside the bookshelf because the plane of alignment is a plane inside the bounding box of the bookshelf. In that case, the bookshelf is broken down into many triangles. Each triangle of the bookshelf will be transformed into a very thin rectangular block, with its own bounding box. The same algorithm can then be used to do the alignment. After all objects have been placed, the object IDs and their transformation matrices are stored in a cell file.

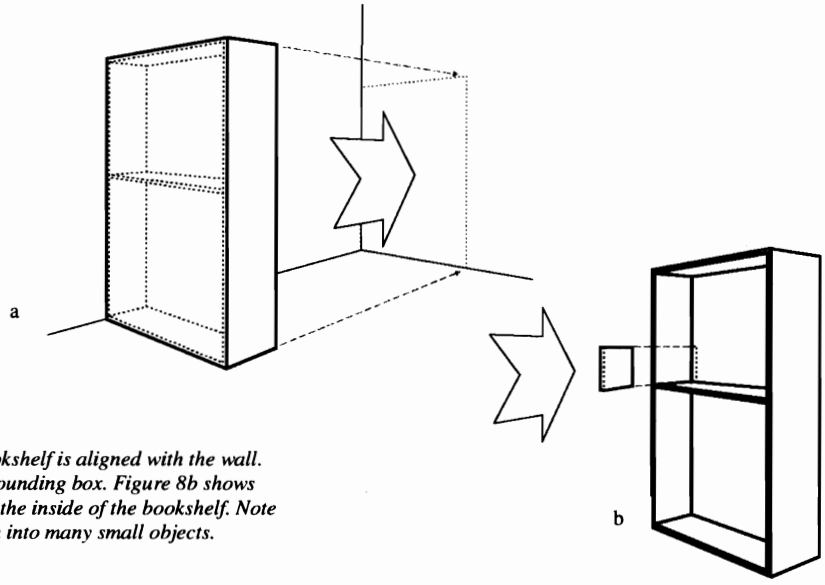


Figure 8a shows how a bookshelf is aligned with the wall. The thick lines shows the bounding box. Figure 8b shows how a book is aligned with the inside of the bookshelf. Note that the bookshelf is broken into many small objects.

6 Illumination

There are three kinds of common shading methods: flat shading, Gouraud shading and Phong Shading. Flat shading means using a constant color throughout the whole triangle. Gouraud shading is to calculate the colors at the three vertices of a triangle and the colors at the middle and edges are linearly interpolated. Phong shading means that colors for all pixels are calculated from an illumination equation since it assumes each pixel has a different normal. Among the three, Phong shading has the highest fidelity and flat shading is of the poorest quality. Although Phong shading produces the best effect, most graphics boards use Gouraud shading as the fundamental shading method. That is because Gouraud shading is computationally much less expensive than Phong shading. In our system, Gouraud shading is used for general lighting but not for special lighting effect cast by extended light sources. The main problem is that the three shading models mentioned are all based on the Phong illumination model, which works only for point light sources. However, our environment is full of extended light sources. Therefore, we cannot rely on the graphics hardware or Gouraud shading to produce special illumination effects like bright regions on walls. Another reason is that graphics hardware usually supports only a fixed number of light sources; for instance, eight on most SGI workstations. This is definitely not enough to simulate the effect of so many light sources in our environment.

To accurately generate the illumination effect, radiosity method could be used. Radiosity method is a simulation based on light energy distribution. It can generate realistic soft shadow but there is a problem in how to present the results it generates. In radiosity method, the original triangles in the model are sub-divided according to the variation of the shadows. To represent complicated shadow variations, it is not uncommon to divide one triangle into a few hundred smaller triangles. For an environment with hundreds of light sources and thousands of objects there are often shadows that are too complicated to handle this way, not to mention the time needed for the process. If time is not a factor, images can be used to represent the shadow variations. Doing this way can reduce the number of triangles needed but will greatly increase the number of texture maps to even tens of megabytes, since almost every triangle will have its own textures. Most of the low-end to mid-range workstations are installed with only 16MB or less texture memory. Too many textures will overload the texture memory and cause excessive texture swaps in and out of it. Usually, the system bus of mid range workstations can transfer data at a rate of a few hundred megabytes per second. That is, for frame rate of 30 fps (frames per second), the bandwidth limit is about 10-20 MB per frame. If the size of textures is around this range, the system bus will be flooded by textures all the time, leaving no bandwidth for other system activities. If the size is substantially larger, the system will not be able to deliver the textures fast enough to the graphics pipeline for rendering, no matter how big the texture memory is, i.e. the system bus will become the bottle neck of performance. Therefore, radiosity-based illumination results of complex environments cannot be used in an interactive system easily, especially in low-end to mid-range workstations. To make the scene look more physically realistic, texture mapping is used to simulate two special and subtle illumination effects in our system: soft shadows on walls and the reflection of light sources on the floor.

6.1 Soft Shadows

Soft shadows on a wall cast by light sources on the ceiling usually take parabolic shapes. The shape is parabolic because the light sources are extended. From the technique in calculating the illumination in radiosity method, we know that a point light source at the ceiling will cast an oval bright region on the vertical walls nearby. The position and size of the bright region is determined by the nearest distance between the point light source and the wall. The further away is the light source, the lower and bigger is the bright region but dimmer.

Figure 9 shows the eqi-brightness curve on a wall that is illuminated by a small flat light source L. I_L is the intensity of the light source L. I_P is the brightness of the point P.

$$I_P = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} I_L \qquad (1)$$

$$I_P = \frac{ab}{\pi r^4} I_L$$

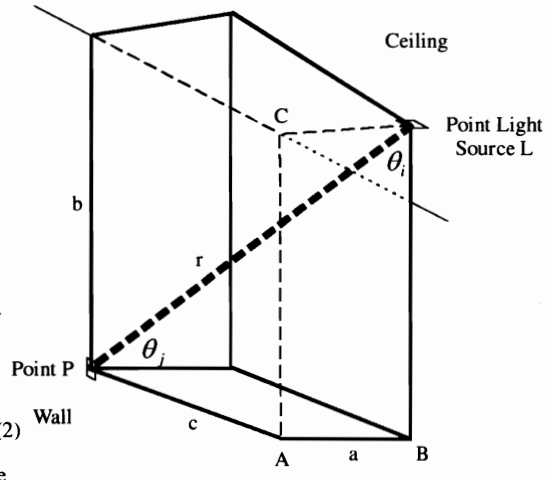
$$I_P = \frac{ab}{\pi\left(a^2+b^2+c^2\right)^2} I_L$$

set I_P = constant k for equi - brightness contour

$$\frac{ab}{\pi\left(a^2+b^2+c^2\right)^2} I_L = k$$

$$\frac{abI_L}{\pi k} = a^4+b^4+c^4+2\left(a^2b^2+b^2c^2+c^2a^2\right) \qquad (2)$$

Taking b, c as coordinates for the wall plane, the equation is oval.



If the light source is extended, say a line perpendicular to the wall, the brightness on the wall will be the sum of illumination of infinitely many point light sources like those shown in the figure 11. Illumination map of parabolic shape is used to simulate the illumination on walls. The texture is not generated for any particular light source, rather it is used to represent the illumination by a general light source. Here, we assume that the bright region cast by light sources with different shapes look similar, as long as the light sources have the similar size. Thus, we can use the same kinds of textures for different light sources. In mapping the illumination textures, the vertical position should be set according to the distance between the light source and the wall. The relation between the vertical position of the bright region and the distance of a light source from the wall can be determined. Due to symmetry, we only consider the points on the plane ABLC in figure 9. Equation (1) becomes:

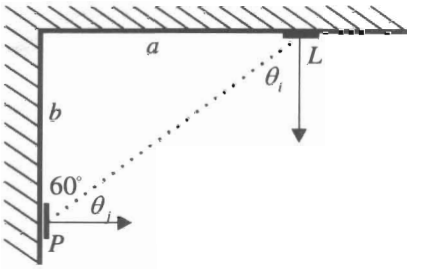


Figure 10 shows the side view of figure 9. It is found that the positon of the maximum brightness of the bright region cast by a small flat light source on the ceiling is linearly related to the distance of the light source from the wall.

$$I_P = \frac{\cos\theta_i \sin\theta_i}{\pi r^2} I_L \qquad \text{since } \theta_j = \frac{\pi}{2} - \theta_i \text{ when } c = 0$$

$$I_P = \frac{\cos\theta_i \sin\theta_i}{\pi (a/\sin\theta_i)^2} I_L = \frac{\cos\theta_i \sin^3\theta_i}{\pi a^2} I_L \qquad (3)$$

$$\frac{dI_P}{d\theta_i} = \frac{3\cos^2\theta_i \sin^2\theta_i - \sin^4\theta_i}{\pi a^2} I_L = 0$$

$$\theta_i = \pi/3$$

That means every light source at the ceiling will cast the brightest spot on vertical walls at a direction 60 degrees to the vertical direction. The texture should be placed lower if the light source is farther away from the wall.

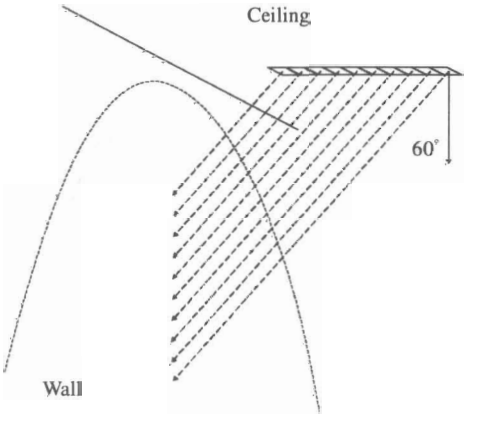


Figure 11 shows the bright regions of an extended light source. It can be considered as the sum of a group of small flat light sources aligned along a line.

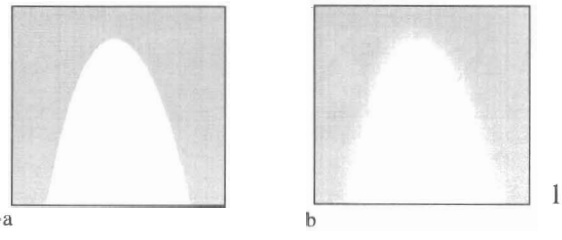


Figure 12a is the texture map with a parabola shape. 12b is the gaussian blurred image of 12a.

The illumination texture is created by heavily blurring an image containing a parabolic shape. Figure 12b shows the texture map generated from Figure 12a. In the case where several lights are situated close to each other, the bright regions of two adjacent lights may overlap. To handle this case two more textures are created as shown in Figure 13. There are altogether three different kinds of textures to simulate the most common illumination effects on a wall. One limitation is that the closely situated light sources should have the same distance from walls, otherwise there are no suitable textures to join the textures of different heights together.

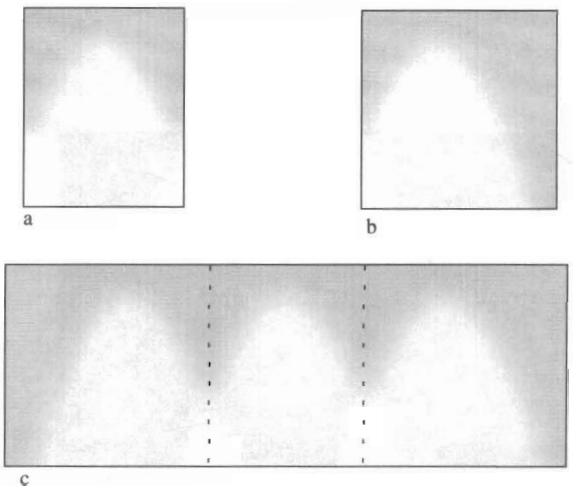
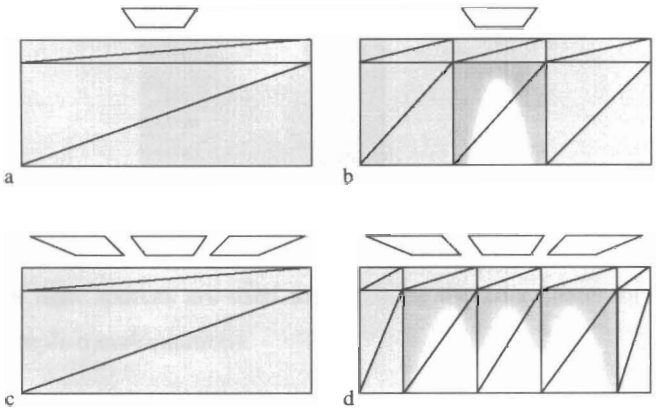


Figure 13a is the texture used for a light with other lights at both sides. 13b is the texture used when there is a light at one side. 13c shows the combined texture for three closely situated lights.

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. Usually a big wall formed by two big triangles may have more than one bright regions cast on it by multiple light sources. Since it is not possible to map more than one kind of texture on a triangle or to map one texture at two or more arbitrary positions on a triangle, such big triangles must be subdivided. Figure 14 shows how the triangles are subdivided. If the distance between two light sources is smaller than a preset threshold of two meters, the bright regions they cast on the wall are assumed to have some overlapping. Thus the subdivision is done as shown in Figure 14d. The boundary between two textures is chosen to be the equi-distance line on the wall next to the two adjacent light sources.



Figures 14a,c show the original triangulation and with 1 and 3 light sources respectively. Figures 14b,d show the triangulation for the illumination texture mapping for a and c, respectively.

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 environment. 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 "reasonably realistic" 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. See figure 12 for a scene with illumination rendered with our texture mapping method.

Figure 15 shows the effect of illumination by illumination texture mapping. Note that there are only three kinds of illumination textures applied.

6.2 The Reflection Map

Texture mapping is also used in our system 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 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 which moves with the viewpoint to simulate the specular reflection of a light source on the ceiling.

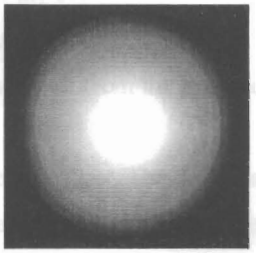


Figure 16 shows a reflection map's alpha component (white for opacity).

The reflection map used is a transparent texture map, with its alpha component shown in Figure 16 (white for opacity). The coordinates of light sources are used to determine the coordinates of the floating reflection map, through some simple transformations.

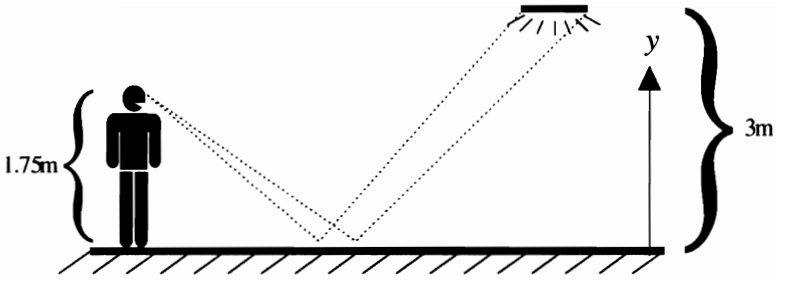


Figure 17 shows the position of the reflection of a light source.

Transform = $(-T) \cdot S \cdot T$

where T is the translation in x, z dimension that moves the origin to the viewer's position. S is the scale transform in x, z dimension by a factor of 1.75/3.0=0.5833. In other words, the coordinates are scaled down by 41.7% about the viewer's position. The y dimension is not changed since the y value of the floor can be used directly.

7 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. The display by the walkthrough engine is done frame by frame. In each frame, the display process can be divided into two phases: the computation phase and the rendering phase. The computation phase means computations like polygon culling is carried out by CPU. The rendering phase means that projections, lighting computation, and rasterization are carried out by the graphics hardware. Theoretically, the two processes can work concurrently; however, in practice it is usually not the case. When a large number of polygons are fed at a fast rate, the graphics hardware is not fast enough to render then and polygons' data will accumulate in the input buffer. The computation process will be blocked whenever the input buffer is full. For complex environments, this situation occurs very often that the input buffer is always full and thus there is virtually no concurrency between the CPU computation and hardware rendering, so we can treat the execution of the two phases as a sequential process. Usually, polygon culling techniques can be used to cull more polygons and make the graphics hardware's job easier. Hence the trade-off between the computation and rendering can be adjusted by controlling the number of polygons rendered by using different polygon culling technique. Some of the speedup techniques we used offload some of the work from the graphics hardware to the CPU.

7.1 Texture Grouping

Triangles with the same textures should be rendered together. This is not a polygon culling technique, but a means to utilize the graphics pipeline more efficiently. Although there is hardware's assistance, texture mapping is still a slow process compared with rendering without texture. The texture memory serves as a buffer to temporarily store textures. Whenever a textured triangle is rendered, the texture memory is searched first. If the texture is in the texture memory, no transfer from the main memory is needed. The ideal situation is that texture memory is large enough to hold all different textures in use. In this case, all the textures can be downloaded to the texture memory in the first frame, and all the subsequent frames will need no more texture transfer. However, the amount of texture memory is

usually very limited. Suppose a workstation has 4MB of texture memory and it can transfer data from main memory to the graphics board at a rate of 600MB per second. For a 30 frames per second requirement, the bandwidth needed is 20MB per frame. A typical 128x128 full color texture is about 64KB. Therefore, only 64 such textures can be stored in the texture memory and 320 can be transferred through the system bus during one frame. If the amount of textures is much higher than the texture memory allows and the polygons are drawn in random order, most of the textures will have to be removed from the texture memory, without being re-used, for other textures to move in. Those removed textures will later be transferred in again for rendering the same frame, which highly increases the workload of the system bus to even over the limit of the bus. Rendering triangles with the same texture together can reduce the traffic between the main memory and the texture memory. Each kind of texture is transferred only once during one display cycle. So as long as the amount of different kinds of textures needed per frame is less than the average bandwidth limit, the system bus will not be overloaded.

7.2 Drawing Primitives

Triangles are the basic drawing primitives of graphics hardware but most of the surfaces of indoor environments are rectangular so a better match is a quadrilateral, or a quad. Although quads will be decomposed into two triangles in rendering, some of the information between the two triangles are the same and can be shared because there are two common vertices. To understand this, we first need to know how the hardware renders a triangle. Some information like vertex colors and z-values are essential to the correct rendering of the triangle. Since the hardware uses Gouraud shading, only the colors at the vertices are calculated and the colors and z-values of pixels in the middle are interpolated. The colors and z-values are generated scanline by scanline by scan conversion (see figure 18).

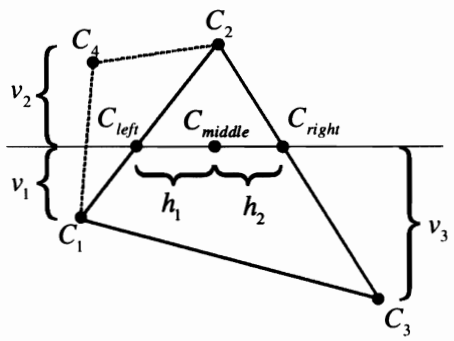


Figure 18 shows how the color of the pixel C_{middle} is found. Note that the colors are computed incrementally that only additions are used after startup.

$$C_{left} = C_1 + dv_{left} \times v_1 \quad \text{where } dv_{left} = \frac{C_2 - C_1}{v_1 + v_2}$$

$$C_{right} = C_3 + dv_{right} \times v_3 \quad \text{where } dv_{right} = \frac{C_2 - C_3}{v_2 + v_3}$$

$$C_{middle} = C_{left} + dh \times h_1 \quad \text{where } dh = \frac{C_{right} - C_{left}}{h_1 + h_2}$$

Here, we consider color interpolation only; z-values interpolation is similar. In graphics pipeline startup, values C_1, C_2, C_3, dv_{left} and dv_{right} are calculated, where the C_x are colors. These calculations are very time consuming. If a quad is divided into two triangles and passed to the graphics hardware separately, the values C_1, C_2 and dv_{left}, or dv_{right} for triangle $C_1C_2C_4$ will be re-calculated. For rendering two triangles, there are six color calculations and six dv calculations. For rendering one quad, there are only four colors calculations and five dv calculations. Thus, 33% of color calculations are saved by quad drawing. Like quad, triangle strips can provide an even bigger saving, up to 66% of total color calculations and 33% of dv calculations. From the specification of a typical mid-range graphics workstation, a triangle-strip's rendering speed is about three times faster than that of a quad. Supposing a quad is rendered as two triangles, a triangle strip's rendering speed is about 50% faster

than quad drawing. However, an undesirable aspect about triangle strip drawing is that it requires the vertices to be traversed in a special order, i.e. in either a zigzag or a fan-like manner (see figure 19). It is easy to organize vertices of a regular polygon mesh in this way but not so easy for a general mesh. The overhead for rendering triangle strips is also bigger than quad drawing. If there are not many triangles in one triangle strip, the saving may not be able to cover the overhead. For these reasons triangle strips are not used in our system. Only quads and triangles are used.

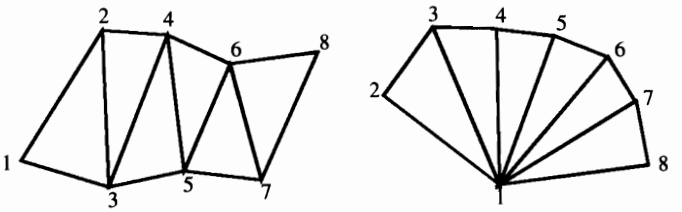
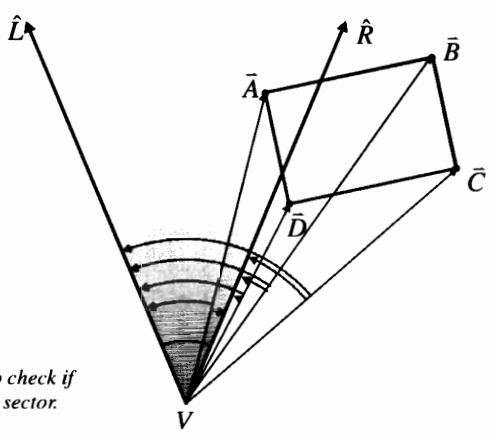


Figure 19 shows two types of triangle strips: the strip type and the fan type. The numbers indicate the order of traversing the vertices.

7.3 View frustum culling

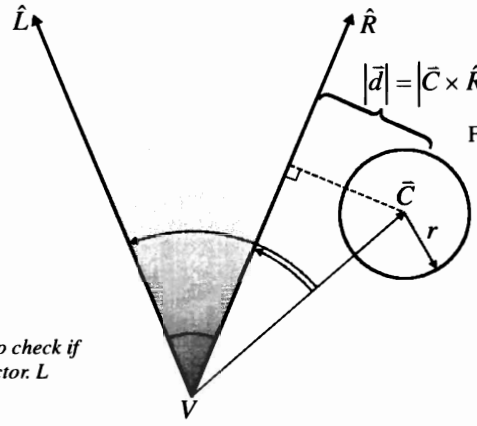
View frustum culling is one of the classic polygon culling techniques. Assuming that objects are evenly distributed and the view angle is 90 degrees, this method can on average eliminate three quarters of all the objects. The test involved is an intersection test between a view frustum and a bounding box. For indoor environments it is a good idea to make it a 2D test by projecting the view frustum and the bounding boxes to the floor. That is because the model of a sufficiently big indoor environment looks like a thin board. When the viewer is inside the model, there are not many things in the up and down directions, and most of the objects are in the horizontal directions. In 2D the test reduces to the checking the intersection between a sector area and a rectangles or a circle, depending on the shape of the bounding area. To make sure a rectangle does not intersect the view sector, at most eight cross products are needed, two for each vertex. It needs one or two cross products to test if a point is in the sector. So on average one and a half cross products per vertex and hence six cross products per bounding rectangle are required. Computing cross product in this special 2D case involves two multiplications and one addition. Thus, altogether twelve multiplications and six additions are needed. If the rectangle intersects the sector, the test can be completed a bit faster since if any one of the vertices is detected to be in the sector, the test can stop with intersection reported.



For a point P, find $\vec{d} = \vec{P} \times \hat{R}$.
if $\vec{d}$ points out, P is not inside sector;
else find $\vec{d} = \vec{P} \times \hat{L}$,
if $\vec{d}$ points out, P is inside sector;
else P is not inside sector;
Repeat for the four vertices of the rectangle.

Figure 20 shows how to check if a rectangle intersects a sector.

A bounding sphere, on the other hand, projects to a circle in 2D. With the special properties of a circle, the test can be made even simpler. Suppose the center and radius of the circle are available. Referring to figure 21, either one or two cross products is needed so on average it needs about three multiplications and three additions.



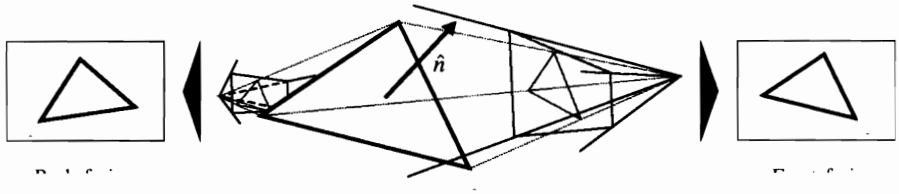
For a center C, find $\vec{d} = \vec{C} \times \hat{R}$.
if $\vec{d}$ points out and $|\vec{d}| > r$, no intersection;
else find $\vec{d} = \vec{C} \times \hat{L}$,
if $\vec{d}$ points out, intersection occurs;
else if $|\vec{d}| > r$, intersection occurs;
else no intersection.

Figure 21 shows how to check if a circle intersects a sector. L and R are unit vectors.

7.4 Back Face Culling

Back face culling is another classic method. It means culling all the polygons that do not face the viewer. This method can cull roughly half of the polygons in the field of view. When rendering polygons, normal vectors are required for lighting computation. The normal vector can be used to indicate which side of a polygon is the front face. Usually, a dot product is required for the test. Let $\vec{V}$ be the vector from the polygon to the viewpoint. The face is front facing if $\vec{V} \bullet \hat{n} > 0$. In OpenGL, back facing triangles are determined by the orientation of the projected vertices. It is assumed that all triangles' vertices are passed to OpenGL in the same convention of the right hand rule (see figure 22). A triangle is front facing if its projected vertices are anti-clockwise; otherwise, it is back facing. OpenGL's back face culling algorithm uses this approach because it has to process triangles with no normals specified or with different normals for different vertices. The disadvantage is that all three vertices must be provided to tell whether or not the triangle is back facing. In practice, the vertices are passed to OpenGL one by one. OpenGL will instruct the graphics hardware to start the pipeline right after the first vertex is passed. When the third vertex is passed, graphics pipeline has already done some of the initialization. If the triangle turns out to be back facing, the whole pipeline will be stalled and flushed. If the hardware keeps encountering such situations, its efficiency will be severely affected. If every triangle has one and only one normal, the dot-product approach can be more efficient.

Figure 22 shows the different orientations of projected vertices for front and back faces.



Backface culling is usually done polygon by polygon. Since there is no hardware assisted dot product

computation, one extra dot-product per polygon could be a big burden to the CPU when polygon count is very large. Object space coherence can again be exploited to facilitate backface culling. In a typical indoor environment the walls and furniture are very likely to be aligned. Many polygons are lying on the same plane. For instance, a wall may consist of hundreds of polygons. Repeating dot-product computation for hundreds of time for backface culling for these coplanar polygons does not make any sense. To reduce the redundant dot-product computation, the model is checked for aligned polygons. Each of these polygons is associated with the plane it is lies on, called an alignment plane. Each of these planes has a flag to show whether it is back facing in each frame. At the beginning of each frame, the flags are cleared. In performing backface culling of a polygon, the flag of its alignment plane is checked first. If the flag is marked back facing, the polygon is back facing. If the flag is marked front facing, the polygon is front facing. If the flag is empty, the plane is tested and the flag is marked. The number of dot-products is then reduced to the number of distinct alignment planes associated with the polygons. A tolerance is allowed to reduce the number of alignment planes and this solves problems arising from numerical inaccuracy in defining the polygon-plane association. Polygons that are very close to an existing alignment plane but are not on it are made to be associated with it. This does not produce obvious visual artifacts.

In our system, the back face culling algorithm is integrated with a special computation for lighting calculation, which will be described in later sections. The dot product $\vec{V} \bullet \hat{n}$ needed for back face testing is available as a side product in that computation.

7.5 Potential Visible Set Pre-computation

This PVS method relies on dividing an indoor environment into cells connected by portals, which are transparent. The visibility of cells and objects to a particular cell C is tested by checking the intersection between those cells' and objects' bounding boxes and the visible region of cell C. This pre-computation produces two levels of visibility information: 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. Cell-to-object visibility further records the potentially visible objects in those potentially visible cells because usually not all of the objects in those potentially visible cells are visible to the viewer. Figure 23 shows a typical situation of cells and objects and their visibility to a cell located in the middle. The PVS database can help eliminate a large potion of objects without any runtime computation. If the floor is sufficiently large with a lot of rooms, the elimination will be substantial. However, the visibility information stored in the PVS database is still too conservative. For instance, figure 23 shows that there are many invisible objects marked by the PVS method as potentially visible to the middle cell. Note that all objects in the adjacent cells, which have direct portal connection to the middle cell, are marked. Clearly, there is no way for the viewer to see all the objects in all adjacent cell at the same time. Far too many objects are marked due to the off-line nature of the algorithm. In pre-computation, the viewer's exact position and view direction in each frame at runtime is not known. The algorithm just assumes that the viewer is inside a particular cell and the viewer can move to anywhere in the cell and look in any direction. Hence, further processing is needed to reduce the set of potentially visible objects. However, the further processing of all the potentially visible objects recorded in the PVS in every single frame is already a considerable burden to the CPU. Motivated by

this, a new runtime method, called dynamic visibility method, that does not rely on PVS has been developed and is described in the later sections.

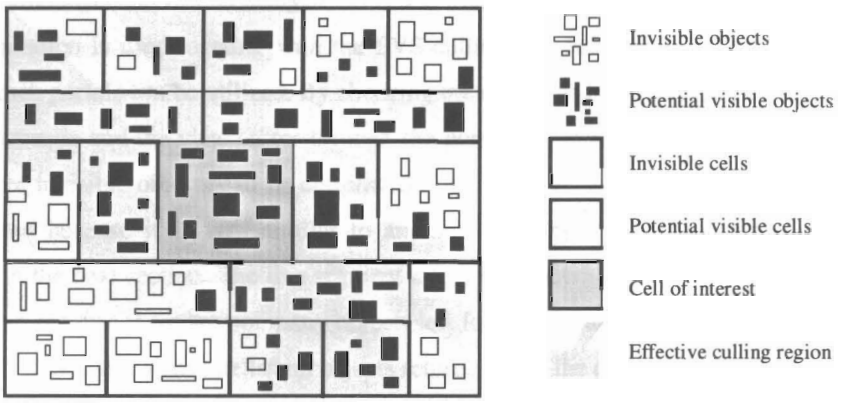


Figure 23 shows the potential visible cells and objects recorded in the PVS database for the center cell. Cell of interest is the cell in which the viewer is.

7.6 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. 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. View frustum culling is carried out after the PVS to give a tighter culling but some invisible objects are still not eliminated even after these two processes. Therefore, the runtime eye-to-object visibility is used to further eliminate the invisible objects. The most straight-forward way is to find the intersection between the line segments connecting the viewpoint to the four corners of a bounding box with the walls of the related cells. This treatment is computationally expensive since the walls could be very complicated. There could also be mis-report of some visible objects as invisible. For instance, this occures for a long object placed behind a narrow door (Figure 24).

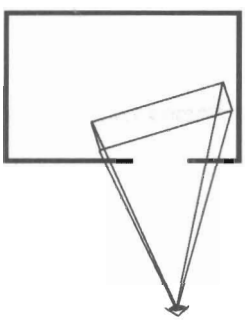


Figure 24 shows the situation when a visible objects is mis-reported as invisible by the method that checks the intersection between line segments connecting the viewpoint and the four bound box corners and the walls.

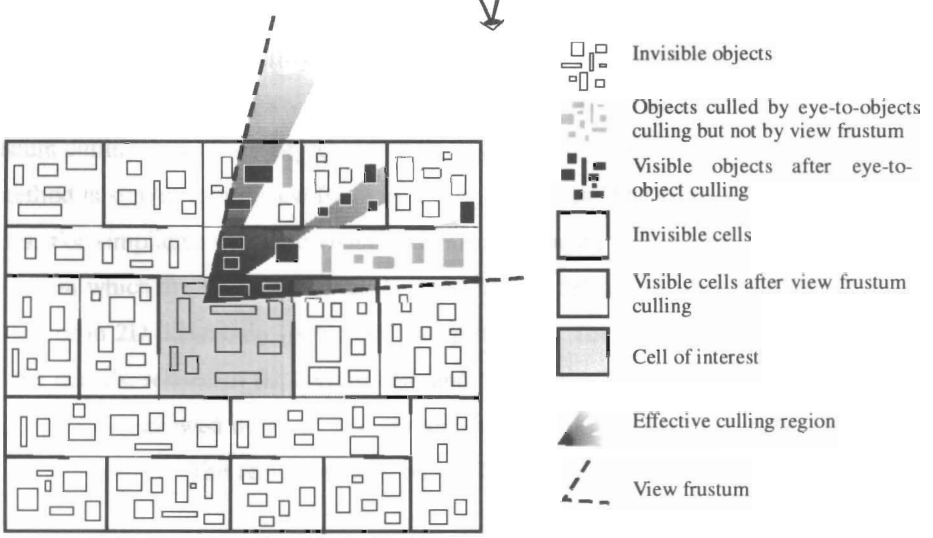


Figure 25 shows the potentially visible cells and objects returned by the eye-to-object culling.

The mis-report situation occurs when the four corners are hidden but the middle part is visible. A computationally less expensive variation is by checking the four line segments with the portals but the mis-reporting situations still exists and further analysis is needed to eliminate the error. Since the eye-to-object visibility computation is used together with the PVS culling, the information about which cell is visible through which portals can be utilized. By checking the existence of intersection between the object's bounding rectangle and the view sector through the portal just in front of the cell where the object is located, more invisible objects can be eliminated. This view sector checking method is, however, not implemented because it is very similar to another visibility computation technique which will be described in the next section. The line segment intersection method is used despite it is computationally expensive because there are not many objects left for the test after the PVS and view frustum culling. Figure 25 shows the visible cells and objects returned after the eye-to-object visibility test. The dark sector region represents the effective culling region. Since the algorithm tests the line segments only with the cell where the tested object is, the effective culling region is a bit larger than the actually visible region.

7.7 Polygon Level Culling

Back face culling is polygon level culling. Besides it, techniques like PVS, view frustum culling and eye-to-object culling can also be extended to the polygon level. However, the effect of their performance gain at the polygon level is not as obvious as object level algorithms. Usually, an object may consist of hundreds of polygons. Polygon level culling is roughly two order more computationally expensive than object level culling. Since there is no special hardware to assist these tests, too many runtime tests may increase the CPU's workload so high that it becomes a bottom-neck in the rendering cycle. Therefore, except back face culling, no other polygon level culling technique is used.

7.8 Dynamic Visibility Computation

By generalizing the eye-to-object algorithm, a new algorithm has been devised: dynamic visibility computation, which refers to a method for fast and tight visibility computation. 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 in 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. Nonetheless, the dynamic visibility method is suitable only for indoor environments or similar densely occluded environments. In fact, it is used in our system as a replacement of the PVS algorithm, view frustum culling, and eye-to-object visibility algorithm together.

The dynamic visibility method is similar to the view frustum culling algorithm. The difference is that the view angle is not fixed. For simplicity, dynamic visibility is also computed in 2D. It is done cell by cell, starting with the cell in which the viewpoint is. Each object in the first cell is checked against the view frustum, a view sector in 2D. Then only visible cells are processed. This is done by checking the intersection between the portals, related to the current cell, and the view sector. If the portal cuts the view sector, the view sector is narrowed down to look through the portal and only the objects in the cell behind the portal are processed. The recursion goes on until no more visible portal remains.

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. In figure 26, $\vec{A}$ and $\vec{C}$ are the left and right vectors respectively. Each test is extremely efficient, involving only three multiplications, one and a half additions and two comparisons, on average. 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 tests 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 a visible cell and the view frustum, we can determine the visibility of the object to the viewpoint. From the description of the algorithm, it looks very similar to the eye-to-object culling algorithm and in fact they yield the same set of visible objects. However, they differ in that the number of objects tested for visibility in dynamic visibility is less. Consider figure 27. If objects from the PVS are to be tested for eye-to-object visibility, 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.

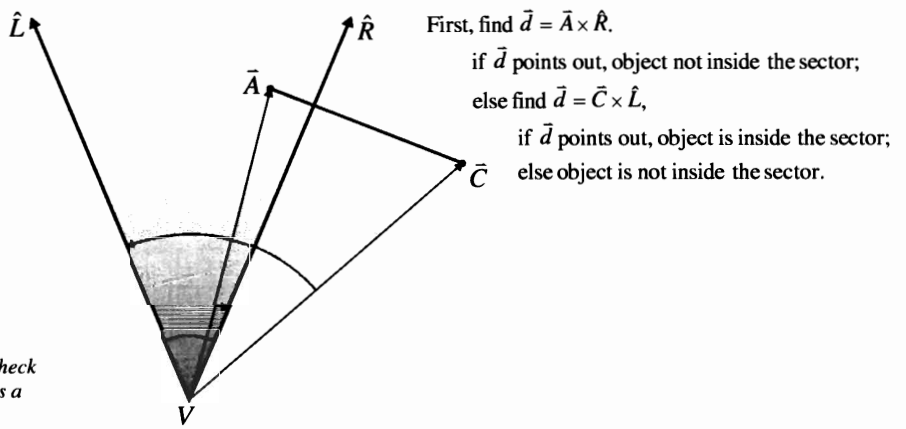


Figure 26 shows how to check if a line segment intersects a sector.

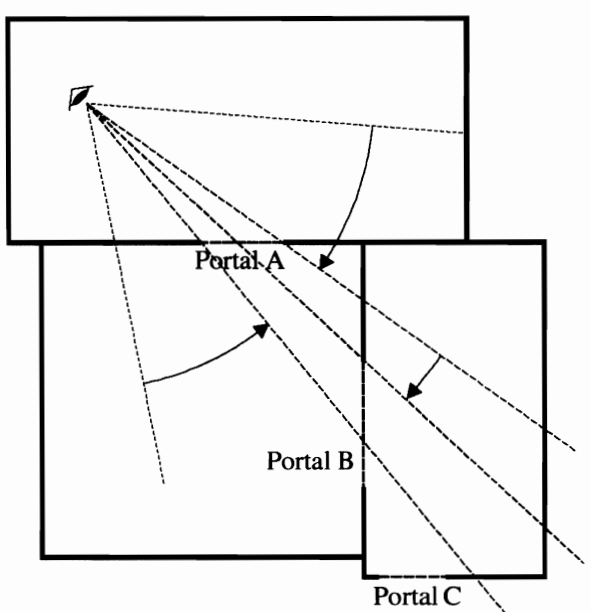


Figure 27 shows how to find the visible cells via portal A, by continuously narrowing the view angle through the portals, terminating when no more portal intersects the view region. The shaded area shows the visible area seen through portal A.

Note that the objects returned by the algorithm are not occluded by walls, but can only be occluded by other objects. 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. The effective culling region of this method is exactly equal to the visible region. Since no pre-computation is involved, it can be used in dynamic virtual environments, but such environments must still be densely occluded for maximum efficiency.

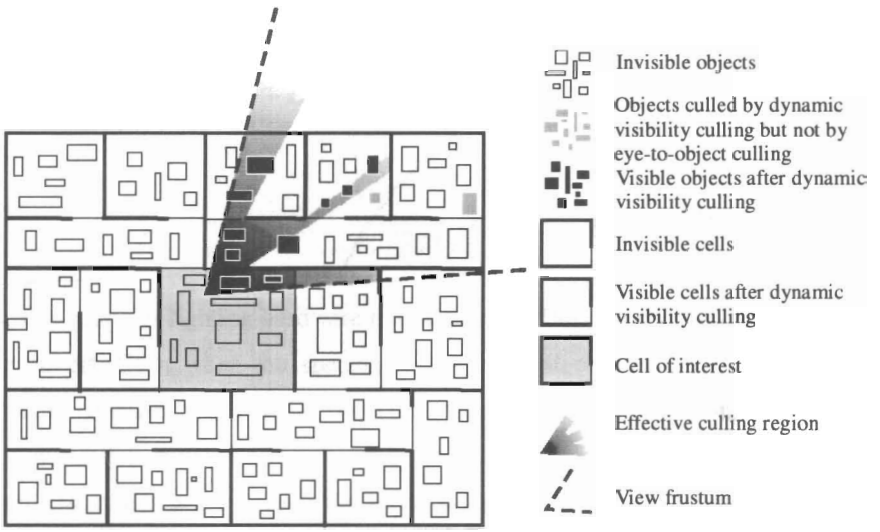


Figure 28 shows the potential visible cells returned by the dynamic visibility method

7.9 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 system. 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. Since vertex colors are provided to graphics hardware, 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. By using software, they are computed only once 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. The definition of the Phong illumination model shows that the ambient and diffuse terms are easy to calculate. The $\cos\theta$ in the diffuse term is the dot-product $\hat{N} \bullet \hat{L}$, which is not hard to compute. The specular term, on the other hand, is computationally much more expensive.

$$I = \underbrace{I_a k_a}_{\text{Ambient}} + \underbrace{I_d k_d \cos\theta}_{\text{Diffuse}} + \underbrace{I_s k_s \cos^n \alpha}_{\text{Specular}}$$

where I_x is the light source intensity for each component and k_x is the color coefficient for each component.

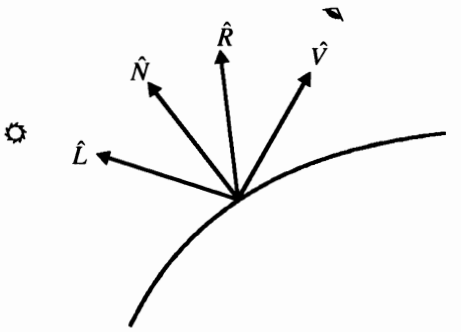


Figure 29 shows the essential vectors used in the Phong illumination model. L is the light vector, N is the nomal vector, R is the reflection of L about N and V is the view vector.

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 as many light sources as needed, regardless the limitation of OpenGL implementation; OpenGL supports only up to eight light sources, for instance. Such a limitation in the number of light sources could prevent a system from mapping the light sources to light models on the ceiling in the environment, which has a large number of light models. So by using software computation, there can be a direct match of light sources and light models. Of course, not all of them will be used for color calculation at the same time. The light sources out of range are not included or the scene will be very bright and the computation will be very slow. Since only ambient and diffuse color are considered, all the computation can be done offline. Therefore, more complicated lighting properties like attenuation can be included as long as the properties do not rely on the position of the viewpoint. However, if there is a moving light source, the computation for this moving light source must be done online. The sum of the colors calculated online and offline will be the final color. In our implementation, there is a moving light source put at the viewpoint. It is used because if the light models are not placed evenly, some part of the model could appear very dark. The light source at the viewpoint serves to illuminate the area in front of the viewpoint so nowhere in view will look too dark. In color calculation, the dot-product $\hat{N} \bullet \hat{L}$ of the diffuse term is calculated online for this moving light source for every triangle. This dot-product is not only used for color calculation, but also for backface culling. The sign of this value tells whether or not the triangle faces the viewpoint, because in this case $\hat{N} \bullet \hat{L} = \hat{N} \bullet \hat{V}$. That is, the information needed for backface culling is available as a side product. Unlike in OpenGL, back facing triangles can be discarded in our system during the attempted color calculation of the first vertex, not the last. Figure 30 shows the difference between the effects of OpenGL lighting and software lighting with the ambient and diffuse components only.

Figure 30 shows the visual difference between OpenGL lighting and software lighting

OpenGL Software

8 Experimental Results

The system is developed on an SGI Indigo² workstation with Maximum IMPACT graphics board, one 195MHz R10000 processor, 192 MB memory and 4MB texture memory. The benchmark tests are also carried out on this machine. Another set of tests is carried out on an SGI O_2 with 180MHz R5000 processor and 64 MB memory. The operating system used is IRIX 6.2 and screen resolution is 1280x1024. The environment consist of 104,102 triangles, when rendered by brute force on SGI Indigo², the average frame is about 1 fps. The tables and charts below show 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. Here, we generally call the three methods: quad rendering, software lighting and software backface culling as non-object-culling (NOC) speedup techniques, while the other techniques, PVS culling and dynamic visibility methods, are called object-culling (OC) speedup techniques. Chart 1 shows the relative speedup of NOC and OC techniques.

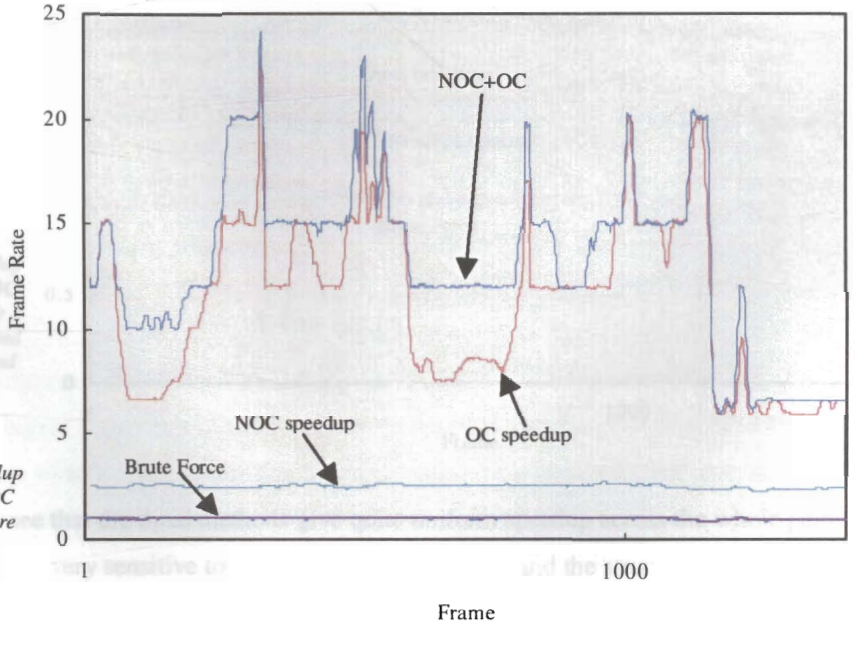


Chart 1 shows the speedup effect of the NOC and OC techniques. The tests were carried out on an SGI Indigo² IMPACT workstation.

We can see that the NOC speedup techniques alone do not give very large speedup. OC speedup techniques, on the other hand, produce a much higher speedup. From chart 1 we can see that the NOC speedup techniques on average give one to two times speedup while OC techniques give as large as ten times. We first present the detailed speedup statistics of the various NOC methods in table 1. The

tests were carried out with other speedup techniques turned off and tested on an SGI Indigo² IMPACT workstation.

NOC techniques	***Benchmark results (1409 frames)***				
Quad/Tri	*Tri*	*Quad*	*Tri*	*Tri*	*Quad*
Lighting	*Hardware*	*Hardware*	*Software*	*Software*	*Software*
Back face culling	*OpenGL*	*OpenGL*	*OpenGL*	*Software*	*Software*
Average # of cells	*48.00*	*48.00*	*48.00*	*48.00*	*48.00*
Average # of objects	*1317.00*	*1317.00*	*1317.00*	*1317.00*	*1317.00*
Average # of primitives	*104102.00*	*59547.00*	*104102.00*	*52123.87*	*29525.70*
Average # of textures	*8392.00*	*4252.00*	*8392.00*	*5812.71*	*2959.36*
Average time in ms	***988.26***	***623.65***	***679.45***	***498.89***	***395.82***
Average fps	***1.01***	***1.60***	***1.47***	***2.00***	***2.53***
Speedup	***0%***	***58.42%***	***45.54%***	***98.02%***	***150.50%***

Table 1 shows the comparison between software and hardware approaches under the worst condition, that is, no PVS, no dynamic visibility, not even view frustum culling, tested on an SGI Indigo² IMPACT workstation. Note that in our implementation, software backface culling is tightly integrated with the software lighting algorithm so software backface culling could not be turned on alone.

The results of table 1 show that using quad drawing call can give a 58% speedup. Software lighting gives 46% speedup and if with software backface culling turned on, together they give a speedup of 98%. All the three techniques together give a 1.5 times speedup. Chart 2 shows the detailed speedup in frame rate along the guided path.

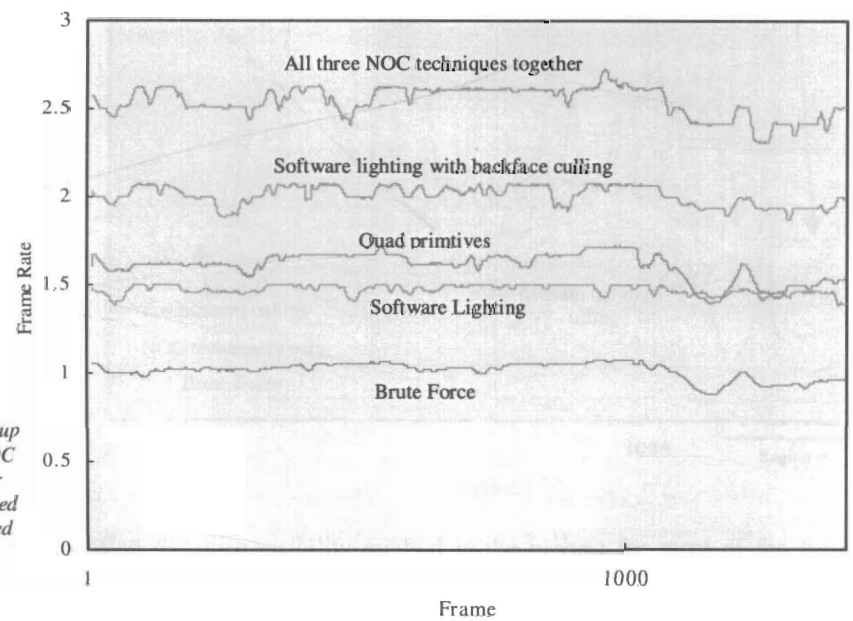


Chart 2 shows the speedup effect of the different NOC techniques with all other speedup techniques turned off. The tests were carried out on an SGI Indigo² IMPACT workstation.

From chart 2, we can see that the three methods give quite uniform speedup across the whole 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 very good property since it ensures a certain amount of speedup, although not very large, to the whole system. Object-culling algorithms, on the other hand, do not have this property, as we can see later. Table 2 shows the comparison of speedup of various OC speedup techniques.

OC techniques	***Benchmark results (1409 frames)***				
View Frustum	*No*	*Yes*	*Yes*	*Yes*	*No*
PVS	*No*	*No*	*Yes*	*Yes*	*No*
Eye-to-object	*No*	*No*	*No*	*Yes*	*No*
Dynamic visibility	*No*	*No*	*No*	*No*	*Yes*
Average # of cells	*48.00*	*11.83*	*8.52*	*7.77*	*5.86*
Average # of objects	*1317.00*	*183.91*	*53.04*	*36.16*	*30.80*
Average # of primitives	*29525.70*	*4493.94*	*1626.39*	*1273.22*	*1051.15*
Average # of textures	*2959.36*	*728.01*	*514.29*	*474.37*	*386.19*
Average time in ms	***395.82***	***134.71***	***101.36***	***86.47***	***85.79***
Average fps	***2.53***	***7.42***	***9.87***	***11.56***	***11.66***
Speedup	***0%***	***193.28%***	***290.12%***	***356.92%***	***360.87%***

Table 2 shows the comparison of the various object culling techniques. The tests were carried out on an Indigo² R10000 IMPACT workstation. The average numbers are average numbers per frame. The average number of objects is the smallest when the dynamic visibility is turned on. All the NOC techniques are also turned on to show the highest frame rate achievable.

The combination of the techniques: PVS culling, view frustum culling, and eye-to-object culling gives speedup about 3.5 times while the dynamic visbility method alone can give the same or even greater speedup. The following chart shows the speedup statistics of the various OC speedup techniques along the pre-defined path of the viewer.

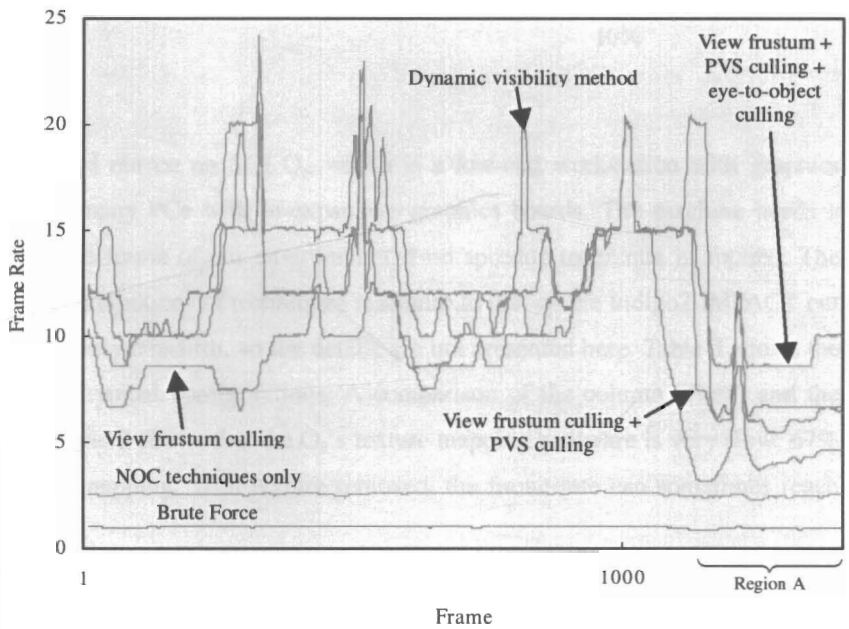


Chart 3 shows the speedup effect of the different OC techniques with all NOC speedup techniques turned on. The test is carried out on an SGI Indigo² IMPACT workstation.

In chart 3, the curve representing dynamic visibility method is the highest for most of the frame except for region A. Region A represents a sequence of frames that the viewer is at the end of a long corridor with seventeen rooms. Such a situation is very problematic, especially for algorithms with higher per-object computation like the dynamic visibility method. The test path was specially designed to include such a region to show the limitation of the dynamic visibility method. In region A, the combined method is more efficient than the dynamic visibility method because there are many visible cells with no or very few visible objects in them. The combined method can quickly remove those invisible objects by pre-computation, but the dynamic visibility method tries to test every object in those visible cells and hence leads to lower frame rate. Although on average the dynamic visibility method can give a higher speedup than the combined method in most of the paths we tested, if the

viewer's path covers a large portion of area like that in region A, the overall frame rate could be lower than the combined method. Chart 4 shows a comparison between the combined method and the dynamic visibility method only.

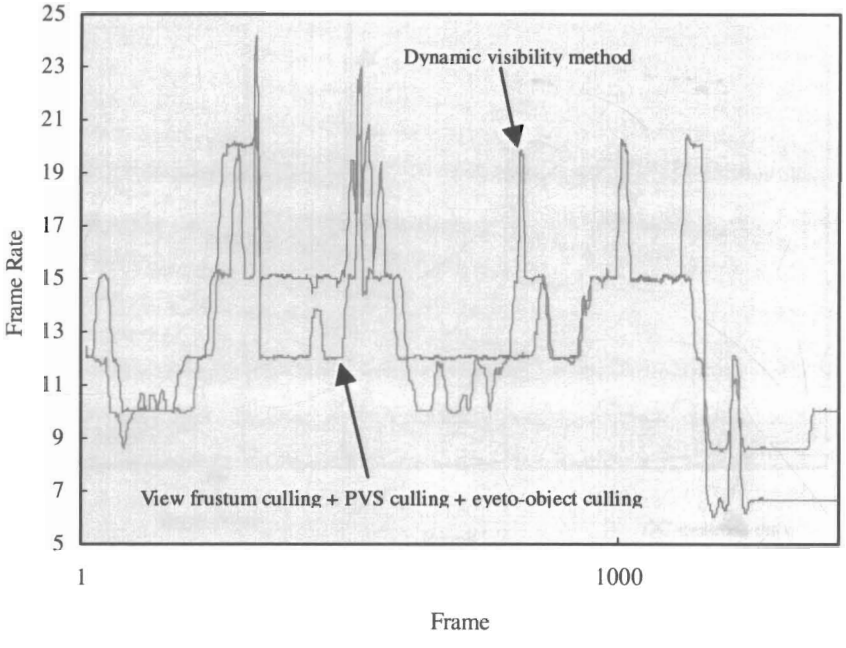


Chart 4 shows the comparison between the combined method and the dynamic visibility method. The tests were carried out on an SGI Indigo² IMPACT workstation.

Another set of tests was carried out on an SGI O_2, which is a low-end workstation with graphics performance even lower than many PCs with in-expansive graphics boards. The machine needs a couple of seconds to render one frame of our environment if no speedup technique is applied. The speedup behavior of various combinations of techniques is similar to that on the Indigo2 IMPACT but with much lower speed, about only one-fifth, so the details are not presented here. Table 3 shows the summary of speedup of a few typical configurations. A comparison of the column "Best" and the column "Best, no texture" of table 3 shows that the O_2's texture mapping hardware is very slow; 67% of the time is spent in texture mapping. With texture removed, the frame rate can sometimes reach interactive rate.

O_2 results	***Benchmark results (1409 frames)***			
Configuration	*Brute force*	*OC method only*	*Best*	*Best, no texture*
Average # of cells	*48.00*	*5.86*	*5.86*	*5.86*
Average # of objects	*1317.00*	*30.80*	*30.80*	*30.80*
Average # of primitives	*104102.00*	*3290.32*	*1051.15*	*1051.15*
Average # of textures	*8392.00*	*994.04*	*386.19*	*0.00*
Average time in ms	***3258.50***	***491.84***	***374.74***	***122.87***
Average fps	***0.30***	***2.03***	***2.67***	***8.14***
Speedup	***0%***	***576.67%***	***790.00%***	***2613.33%***

Table 3 shows the comparison between the various configurations of speedup techniques carried out on SGI O_2. "OC method only" means the NOC methods are turned off. "Best configuration" means that dynamic visibility and the three NOC techniques are turned on. the "best, no texture" configuration is the same as the best one except that texture mapping is turned off.

Chart 5 shows the speedup on SGI O_2 of the four configurations in table 3 along the same path as in the other charts.

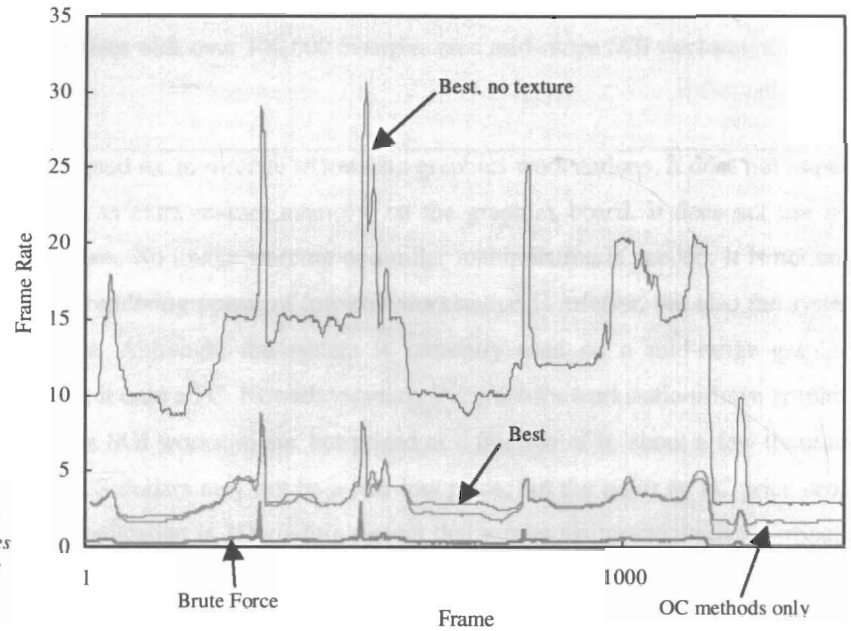


Chart 5 shows the comparison between four typical speedup techniques configurations. The test is carried out on an SGI O_2 workstation.

All the above results are obtained under "heavy" load, which means that all the doors in the environment are open. Most of the doors are usually closed in real applications. Table 4 and chart 6 show the difference in speedup of the best configuration on SGI Indigo^2 IMPACT and O_2 under "heavy" and "light" load.

	Benchmark results (1409 frames)			
Configuration	*IMPACT, heavy*	*IMPACT, light*	*O_2, heavy*	*O_2, light*
Average # of cells	*5.86*	*1.81*	*5.86*	*1.81*
Average # of objects	*30.80*	*12.19*	*30.80*	*12.19*
Average # of primitives	*1051.15*	*465.87*	*1051.15*	*465.87*
Average # of textures	*386.19*	*174.98*	*386.19*	*174.98*
Average time in ms	***85.79***	***56.67***	***374.74***	***240.60***
Average fps	***11.66***	***17.64***	***2.67***	***4.16***

Table 4 shows the difference in speedup of the best configuration on an SGI Indigo^2 IMPACT and an O_2 under heavy and light load. The "heavy" load configuration means that all the doors in the environment are open. The "light" load configuration means that all of the doors in the environment are closed.

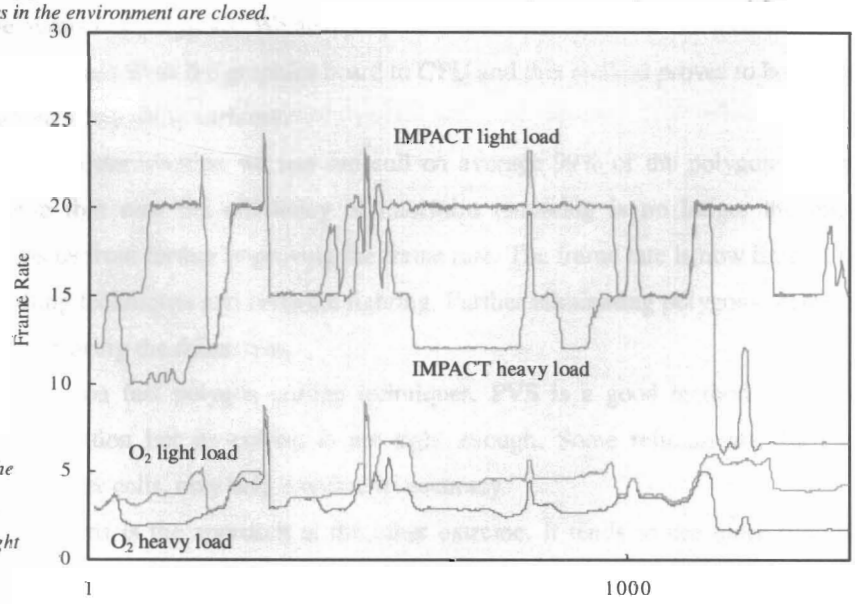


Chart 6 shows the difference in speedup of the best configuration on an SGI Indigo^2 IMPACT and an O_2 under heavy and light load along a pre-defined path.

In short, with the dynamic visibility method, quad rendering, software lighting, software backface lighting, and other techniques mentioned applied in the walkthrough engine, we can achieve a frame rate of about 11-12 fps (c.f. 1 fps in brute force rendering), without a pre-computed visibility database, for a texture mapped environment with over 100,000 triangles on a mid-range SGI workstation.

9 Applications

The system developed is designed for mid-range to low-end graphics workstations. It does not impose very high requirements, such as extra texture memory, on the graphics board. It does not use any image based speedup techniques. No image warping or similar manipulation is needed. It is not only because the computation and rendering power of low-end workstation is inferior, but also the system bus bandwidth is too limited. Although, the system is currently used on a mid-range graphics workstation, the goal is to port it onto a PC. Nowadays, many PC graphics workstations have graphics power equivalent to mid-range SGI workstations, but priced at a fraction of it, about a few thousand US dollars. A few thousand US dollars may not be a real low price, but the point is, PC price drops very fast. A straightforward application is 3D guiding system that guides visitors to reach a particular room in a building or to show the way to the fire exit and washroom. It can also be used in virtual tour and office decoration preview. Since no visibility pre-computation is needed, it can work with dynamic environments. The system can be used for interactive furniture placement or even change of the room structures like moving the walls. It is also well suited for multi-user environments. Beside the essential information like the models, textures, furniture position information and cell/portal information, there is no large data file like PVS data needed. Therefore, it is also suitable for applications across the Internet, which usually has very slow transmission rate between two points.

10 Conclusion and Future Work

We have described a practical walkthrough system of architecture model, and discussed some issues in modeling, illumination and display speedup techniques. We also present a benchmark of performance gains of different techniques for display speedup. We propose a new polygon culling algorithm, called dynamic visibility, that replaces various existing methods together as an all-in-one technique to reduce the number of polygons. We also propose to use software to do color calculations and shift most of the calculations from the graphics board to CPU and this method proves to be useful for an environment without many shiny surfaces.

The new potential visibility determination we use can cull on average 99% of the polygons of the environment. We observe that now the efficiency of hardware rendering is no longer the most important factor 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.

Further work 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 refinements, such as subdivision of cells into smaller cells, may help increase its accuracy.

Dynamic visibility computation is the approach at the other extreme. It tends to use more runtime computation than the PVS approach. Although its accuracy makes it out-perform the PVS method, 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 11-12 even if there are only about one thousand triangles to render per frame, which is much below the maximum capability of the 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. One way to go is to take advantage of multiprocessor machines by parallelizing the method and run it on more than one processor since multiprocessor machine is readily available now. Many dual-processor or quad-processor PCs are now commercially available at a few thousand US dollars.

11 Glossary

3D sensors	Sensors to detect position and orientation. It is usually made of sensor coils and works together with a transmitter of magnetic field. It is essential in totally immerse VR to detect head and hand positions and orientations.
Color computation	It refers to the step that polygon vertices' colors are computed in rendering. It is a step before rasterization.
Gouraud shading	It is a polygon shading method implement in most graphics hardware. The idea is that color calculations are done only for the vertices of polygons. The interior colors are interpolated.
Graphics board	A computer's part dedicated for display related computation such as 3D projection, pixel filling to producing video signal. They are isolated from the CPU because the computation is intensive and rather independent.
Head mount display	A helmet with two displays, LCD or CRT, directly covering the user's eye. It can provide stereoscopic view. Usually, a 3D sensor is attached to the display to check the position of the head. Most of the head mount display block the user's normal view so traditional input devices cannot be used. Therefore, some other input devices like glove, spaceball or microphone are used to issue commands.
Image morphing	Generate intermediate images from two key images to produce a continuous animation of a smooth change from one to another. The generation is usually feature-based that requires human input to match special features.
Image synthesis	Synthesis of artificial image. There are two kinds of techniques: ray tracing and radiosity method. Modern image renderers combine the two techniques together to produce good looking images.
Image warping	Like image morphing, image warping is also used to generate intermediate images from two or more key images. The different is that image warping take camera positions into account and does not require user to match features.
LOD	Level of Details is a technique to reduce number of polygons to be rendered. Multiple models are used to represent the same object at different distance from the viewer.
Phong shading	It is a polygon shading method. It produces realistic shading at an expense of much more computation because color calculation is computed for every pixel in the polygon. It is not commonly implement in hardware due to it's complexity.
Polygon culling	It is a group of speedup techniques in polygon rendering. The main idea is to remove the invisible polygon before rendering. Techniques like PVS, view frustum culling and back face culling belong to this group.
Projection	It refers to the step that 3D coordinates are projected to screen coordinates in

	polygon rendering. It involves a few matrix operations per projection. It is usually done by hardware in high-end graphics workstation but not in low-end ones.
PVS	Potential Visible Set visibility pre-computation is a pre-computation of mutual visibility between cells and objects in a cell divided environment.
Radiosity method	Radiosity is a simulation technique to produce realistic images. The technique is based on energy distribution. It can produce images with very realistic soft shadows. Compare with ray tracing, radiosity method is a relatively expensive process.
Rasterization	It refers to the step that the pixels of a polygon are filled in rendering. It is usually done by the hardware even in low-end graphics board.
Ray tracing	Ray tracing is a simulation technique to produce realistic images. The technique traces how light ray travels before coming into the viewer's eye. It can produce realistic images of reflective and transparent objects but not soft shadows.
Soft shadow	Shadows of objects under extended light sources. It can be accurately computed using radiosity method.
Texture mapping	Use raster images to paste on polygons in 3D to replace complicated details. It is very useful to model different textures of surfaces.
Texture memory	A cache to hold texture images in graphics board to reduce system bus traffic for texture transfer. The size is usually about 4MB to 16MB.
Virtual environment	An environment that the user perceive in a virtual reality system.
Virtual reality, VR	A technique that use 3D input and output devices to give the user a feeling of 3D immersion hence to provide a more real and interactive environment of working or playing.
VR Gloves	A glove with many sensors to detect the bending of joints of hand. It is a device to let the user issue commands using gestures or to let the user touch and move objects in virtual environment.
z-buffer algorithm	Z-buffer, also known as depth buffer, is a special memory space used to record the shortest distance between objects and the viewer for each pixel. It is the most common hidden surface removal algorithm.

12 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", *Proceedings of SIGGRAPH '91,* pp.61-69.

[5] N. Greene M. Kass and G. Miller "Hierarchical z-buffer visibility", *Proceedings of SIGGRAPH '93,* pp.231-238.

[6] N. Greene, "Hierarchical polygon tiling with coverage masks", *Proceedings of 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", *Proceedings of SIGGRAPH '96,* pp.75-82.

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

[10] H. Zhang, D. Manocha, T. Hudson and K.E. Hoff, "Visibility culling using hierarchical occlusion maps", *Proceedings of SIGGRAPH '97,* pp.77-88.

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

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

[13] D. Aliaga and A. Lastra, "Architectural walkthroughs using portal textures", *IEEE Visualization 97*

[14] J.S. De Bonet, "Multiresolution sampling procedure for analysis and synthesis of texture images", *Proceedings of SIGGRAPH '97,* pp. 301-368.

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

[16] S.M. Seitz and C.R. Dyer, "View morphing", *Proceedings of SIGGRAPH '96,* pp.21-30.

[17] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide,* Addison Wesley, 1993.

[18] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, "Fast rendering of complex environments using a spatial hierarchy", *Computer Interface '96*

[19] 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.

[20] Q. Wang, M. Green, and C. Shaw, "EM – an environment manager for building networked virtual environments", *IEEE Virtual Reality Annual International Symposium '95,* pp.11-18.

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

[22] G. Schaufler, "Exploiting frame to frame coherence in a virtual reality system", *Proceedings of VRAIS '96,* pp.95-102.

[23] T.L. Kay and J.T. Kajiya, "Ray tracing complex scenes", *Proceedings of SIGGRAPH '86,* pp.269-278.