Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations

TR90-027

July, 1990

John Milligan Airey

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Research supported in part by Office of Naval Research Contract #N00014-86-K-0680, and by the National Science Foundation, Grant #CCR-8609588.

Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations

by
John Milligan Airey

A Dissertation submitted to the faculty of the The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
1990

Approved by

[illegible]

Frederick P. Brooks Advisor

[illegible]

James Coggins Reader

Jan F. Prins

Jan Prins Reader

John Milligan Airey. **Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations** (Under the direction of Frederick P. Brooks, Jr.)

Abstract

Pre-processing some building models can radically reduce the number of polygons processed during interactive building walkthroughs. New model-space subdivision and potentially visible set (PVS) calculation techniques, used in combination, reduce the number of polygons processed in a real building model by an average factor of 30, and a worst case factor of at least 3.25.

A method of recursive model-space subdivision using binary space partitioning is presented. Heuristics are developed to guide the choice of splitting planes. The spatial subdivisions resulting from binary space partitioning are called *cells*. Cells correspond roughly to rooms.

An observer placed in a cell may see features exterior to the cell through transparent portions of the cell boundary called *portals*. Computing the polygonal definitions of the portals is cast as a problem of computing a set difference operation on co-planar polygons. A plane-sweep algorithm to compute the set operations, union, intersection and difference, on co-planar sets of polygons is presented with an emphasis on handling real-world data.

Two different approaches to computing the PVS for a cell are explored. The first uses point sampling and has the advantage that it is easy to trade time for results, but has the disadvantage of under-estimating the PVS. The second approach is to analytically compute a conservative over-estimation of the PVS using techniques similar to analytical shadow computation.

An implementation of the Radiosity lighting model is described along with the issues involved in combining it with the algorithms described in this dissertation.

Acknowledgements

Andries Van Dam and the Brown Computer Graphics Group introduced me to computer graphics. That was a once in a lifetime experience that I won't forget.

I feel fortunate to have been a part of the graphics cluster at UNC. Everyone in it, (and those who claim not to be in it), contribute to a great research environment. Dr. Fred Brooks, in particular, sets an example that has inpired many students, myself included.

Finally, my family and my friends have given endless support and I thank them all, with special thanks to Hali.

Table of Contents

List of Figures

List of Tables

Chapter I

Overview and Results

This dissertation research was performed as part of the UNC Building Walkthrough project [Brooks86], [Brooks88], [Airey89d], [Airey90b]. The basic goal of the Walkthrough project is to create a virtual building environment, a system which simulates human visual experience with a building, without physically constructing the building. This is intended to aid the part of building design known as design development, when the architect explores, sometimes with the client, different design options.

Part of the goal of the Walkthrough project can be stated simply as, "accurate images, real-time feel, and big models", (figure 1.1). Choosing these objectives and attempting to work on the area of our system that was most deficient forced us to focus on the pre-processing partitioning approach, the pre-processing radiosity approach, and display-time adaptive refinement. The other part of the Walkthrough goal is a natural man-machine interface.

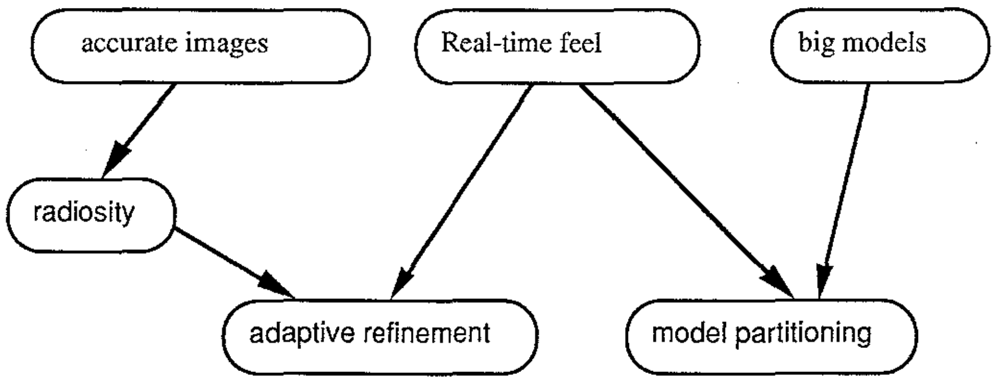


Figure 1.1 The Walkthrough goal, "accurate images, real-time feel and big models", pushed us to develop pre-processing methods to improve update rates, adopt a pre-processing shading model, radiosity, and apply the concept of adaptive refinement during display.

Many components of a building simulator, corresponding to many human senses, are important. My work within the Walkthrough project concentrates primarily on techniques to enhance visual simulation. Visual simulation can be separated into two components, a kinetic component, how the scene moves, and a realism component, how the scene looks. I sought to improve the kinetic visual experience by increasing update rates and to improve the illusion of reality by using the radiosity shading model [Goral84], [Cohen85], [Cohen88], [Wallace89], [Baum89], [Airey89]. For both components, one may use a strategy of pre-computing as much work as possible prior to display. This dissertation describes techniques developed to pre-compute data structures which are used at display time to significantly increase the update rate. Our use of the radiosity shading model is described in chapter IV.

The Walkthrough research team finds natural motion, the kinetic component, to be a very important component of visual simulation. We observe user behavior to be qualitatively different at six updates per second as compared to behavior at one update per second.

• At one update per second (ups) or less, the system is painful to use. It is necessary to use an auxiliary two-dimensional floorplan display, or *map view,* to navigate.

• As the update rate increases from one ups to around twenty ups, interactivity appears to increase rapidly (superlinearly) before levelling off. At around six ups, the virtual building illusion begins to work. It is possible to navigate with only the three-dimensional display, or *scene view.*

In the fall of 1986, we were able to display a relatively spartan model of the Sitterson Hall Computer Science building, composed of 7125 polygons, at about four ups with the Pixel Planes 4 graphics machine [Fuchs85]. We wanted at least six ups and preferably twenty ups.

We perceived the slow update rate as the greatest system deficiency. I developed a process that automatically associates sets of viewpoints, called *cells,* with *potentially visible subsets* (PVS) of the model. The display subsystem has to process only the PVS that is associated with the cell containing the current viewpoint.

This process allows the Sitterson model to be displayed with at least 14 ups and often many more, depending upon the view, on the same Pixel-Planes 4 machine that in 1986 yielded four ups.

The update rate increase due to this process is dependent upon the data. Roughly speaking, the increase in update rate is proportional to the number of rooms in the building.

1.1 UNC Building Walkthrough Summary

A complete building walkthrough system has the following major parts (Figure 1.2).

• A modelling subsystem for the architect, where the canonical model is maintained. We use AutoCAD.

• An image-generation process for constructing a display file from the canonical model and generating the images. This includes the display-compiler subsystem and the display subsystem.

• An interface subsystem that allows the use of many different man-machine interface devices for controlling viewing parameters and illumination of the model.

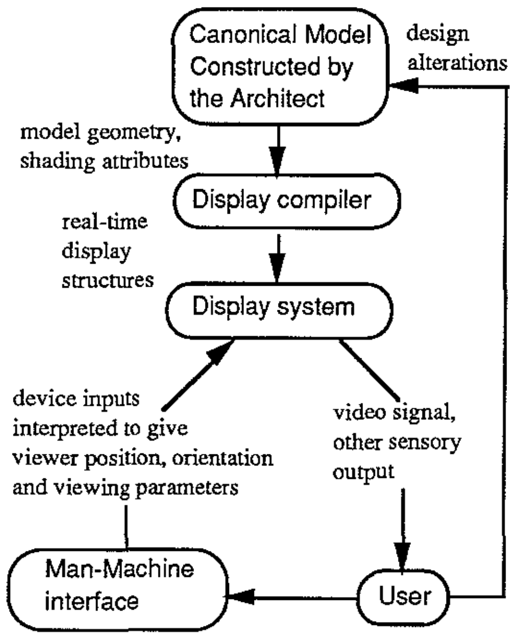

Canonical Model Constructed by the Architect
design alterations
model geometry, shading attributes
Display compiler
real-time display structures
Display system
device inputs interpreted to give viewer position, orientation and viewing parameters
video signal, other sensory output
Man-Machine interface
User

Figure 1.2. Overview of a virtual building system.

1.1.1 Modelling

The modelling subsystem must address ease of construction, (i.e. how many man-hours are required to create the model), model modification, and management of many different versions of the model (a task very similar to source code control [Tichy82]). For our modelling capabilities, we have adopted AutoCAD. This helps us establish partnerships with architects; these partnerships yield us datasets and system evaluation. We have only had limited partnerships so far.

1.1.2 User Interface

The interface subsystem must coordinate input devices with the display system. We have found that a flexible interface to a variety of devices is important. We already use several devices, often concurrently, and need to test new devices often. Frequent users such as architects may want complete freedom of motion (joysticks), while an infrequent user, e.g., a client, may desire a restrictive but more natural interface, (a treadmill with a head-mounted display or big screen display).

1.1.3 Display Compiler

The display compilation task must translate the geometric and surface attribute information from the model into a form suitable for rapid display and interaction with the interface devices. Figure 1.3 depicts our current process of converting an AutoCAD dataset into a form suitable for a virtual world system. The rectangles represent programs. The ovals represent data files. The type of the data file is depicted by the Unix convention of file name extensions.

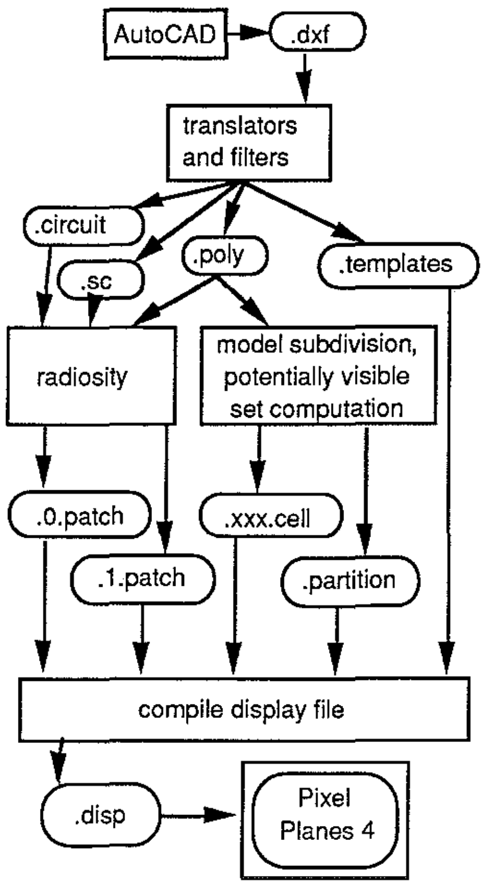


Figure 1.3 Display File Compilation

The AutoCAD external files (.dxf extension) are parsed and converted to a simple format which consists entirely of polygons (.poly extension). Separate files are generated which contain surface attribute information (.sc extension), and the sets of lights for which we want to compute independent radiosity solutions (.circuit extension). Another file (.template extension) allows Phigs+ -like structures to be incorporated into the final display file [Van Dam88]. Instances and templates allow movable objects such as furniture, or temporary diagnostic objects such as cell portals, to be easily added to the display. The display compilation splits into the radiosity process and the model subdivision process.

The radiosity process is briefly described in chapter IV.

The main right fork in Fig 1.1.3.1. is the model subdivision process. It generates a recursive subdivision of the model space [Samat90]. The result of this subdivision process is a tree of splitting planes (.partition extension). The .partition file defines subvolumes, or *cells,*of the model (.cell extension). Each of these cells is processed to determine the polygons that are potentially visible to an observer ranging freely inside the cell. These polygons are then associated with the cell. During display, the cell containing the current viewpoint is found, and only its associated polygons are rendered. Note that a particular polygon may be associated with many cells- i.e., visible from many subvolumes.

The box labelled *model subdivision, potentially visible set computation,* (figure 1.3) contains the programs that are the principal subject of this dissertation. Figure 1.4 is an expansion of that box in figure 1.3.

The box labelled *compile display file* is a program whose main task is to combine all the information generated by previous programs into one coherent structure for display. Even though it serves primarily as a bookkeeper, it is quite a complicated program.

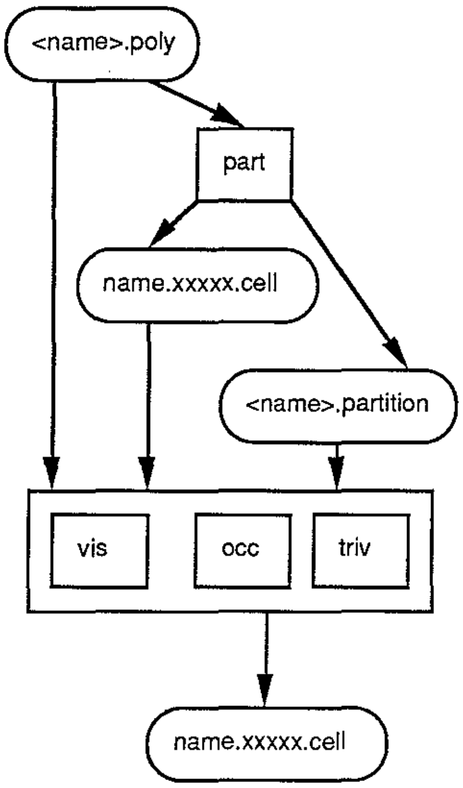


Figure 1.4. The programs used to compute the model subdivision or *part*ition, and identify potentially visible sets. Vis computes the PVS for a cell with a sampling algorithm. Occ computes the PVS for a cell with an analytic algorithm. Triv simply assigns the polygons inside a cell to the PVS and is usually used only to evalute the result of *Part*ition.

1.2 Algorithm Overview

1.2.1 Properties of Architectural Databases

Architectural databases possess special characteristics. Some of these properties follow. Techniques which exploit some of them are described later.

1. Properties of polygon configuration.

   1.1. Most polygons are *axial*, i.e., normal to one of the coordinate axes.

   1.2 Most polygons are rectangles.

   1.3. Large planar surfaces are often structured into multiple, co-planar levels for modelling purposes, shading purposes, and realism detail purposes. For example, the modeller may represent a ceiling with one polygon, but it may be diced into many smaller co-planar polygons to obtain a more accurately shaded image.

1.4. The set of polygons that appears in each view changes slowly as the viewpoint moves, except when crossing certain thresholds, e.g., doors and windows. Consider a simplified three room floorplan (Figure 1.5). The set of polygons visible in a 360 degree field of view from v_1 does not change much until one nears v_2. Then it stays roughly the same until one gets to v_3 where it changes radically again.

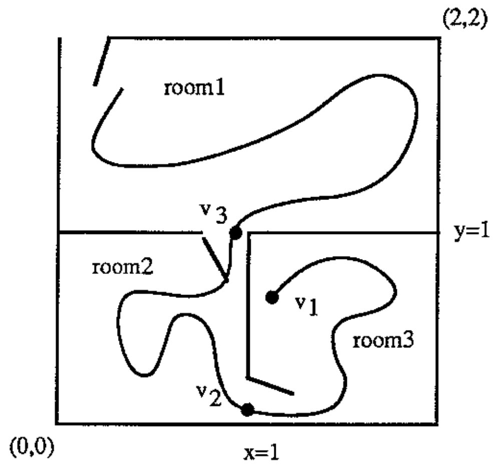


Figure 1.5 Simple three room floorplan with possible viewer path. The set of polygons visible in a 360 degree field of view from v_1 does not change much until one nears v_2. Then it stays roughly the same until one gets to v_3, where it changes radically again.

2. Properties related to the inside-out viewing nature of environment models.

2.1. Many viewpoints have a large portion of the model outside the field of view.

2.2. Surface interreflections in shading calculations are very important for spatial comprehension inside a building.

3. Properties related to depth complexity.

3.1 Any image computed from an interior viewpoint will have many surfaces covering every pixel.

3.2. Many surfaces are completely hidden; they do not contribute to the image at all. Besides the obvious example of back-facing surfaces, forward-facing surfaces in the next room or on the floor above or below do not contribute to the image.

3.3.The fraction of hidden surfaces in a building is basically independent of tesselation required for shading. The number of surfaces hidden is a function of the number of walls, floors and ceilings. Tesselating them uniformly into small patches does not change the fraction of hidden surfaces. Thus the methods outlined in this dissertation should get results independent of shading tesselation.

A simple quantitative analysis comes from [Gharachorloo89], [Sutherland74]. Given P polygons projected onto the image plane, of average area A, a total of $P \bullet A$

pixels must be calculated. Assuming the scene covers a screen of *N* pixels and that the polygons are overlaid in *D* layers everywhere, we have: $P \bullet A = N \bullet D$, where *D* is thedepth complexity. If walls are represented with small individual tiles for shading purposes, then *P* increases while *A* decreases and *D* remains constant.

Additional detail such as fancy window frames and baseboard moldings, etc.has a similar but less easily analyzed effect. The fraction of hidden surfaces increases only slightly.

1.2.2 Model Space Subdivision

The pre-processing algorithm automatically subdivides model space or equivalently, viewpoint space, into *cells*. Define the union of visible polygons for all the viewpoints in a cell as the *potentially visible set* (PVS) for that cell. For any viewpoint in the cell, rendering the potentially visible set for that cell generates an image with no missing polygons. Since the size of the potentially visible set is usually much smaller than the size of the model it came from, it takes less time to render. The rendering process involves at least transforming the polygon to the correct perspective view, and then scan-converting it into the frame buffer if it appears in the viewing frustum and faces the viewer.

For the simple three room floorplan given in figure 1.5, the labelled rooms approximate what one wants in a cell. If the doors were closed (and they had no glass) then one could simply render the polygons in room2 when the viewpoint is in room2 and get roughly a three-fold increase in system update time. If the doors are open, one must add the polygons that can be seen through the doors from room2 to the potentially visible set for that cell.

Even from this simple example it is obvious that certain model subdivisions are better than others. The subdivision process should try to satisfy the following objectives.

Objective 1. Minimize the size of the potentially visible sets. One wants cells whose potentially visible set is not much larger than the visible set of any one viewpoint in the cell. This ensures the best possible speedup.

Objective 2. Minimize the number of cells; split cells only if the resulting child cells have significantly smaller PVSs.

Besides these loosely stated objectives, a subdivision algorithm must satisfy other restrictions.

Restriction 1. It must be easy to find the cell that contains the current viewpoint. This operation is performed for every update during display.

Restriction 2. It must be automatic. No hand-tooling allowed. Architects cannot afford the time and may not have the expertise to hand-weave databases for esoteric display algorithms.

Restriction 1 implies the use some type of data structure suitable for range searching. I chose recursive binary partitioning planes. Furthermore, it is sufficient to restrict their orientation to be normal to one of the coordinate axes. Other options include a regular 3D grid, or adaptive space subdivision techniques such as octtrees or k-d trees [Mehlhorn84], [Samet90]. Other proposed approaches utilize the fact that adjacent cells are typically accessed in a sequential order and usually in the horizontal plane. This would allow a more adaptive subdivision algorithm [Prins90]. Any of these data structures allow the cell containing a viewpoint to be found quickly.

To satisfy Objective 1 and Restriction 2, I devised a heuristic function to choose the splitting planes used in the recursive binary subdivision scheme. Since one wants a splitting plane that is largely opaque, one may limit the choice of splitting planes to those that contain model polygons. The function evaluates each plane containing a polygon for its suitability as a separating plane. Criteria considered are

• how evenly the plane separates the model, called the *balance* of the split,

• how well the plane hides the two sides from each other, called the *occlusion* factor of the split. For example, a floor hides better than a wall with a door in it.

• how little the plane splits individual polygons, since polygons that are split will have to be put in the potentially visible sets of both partitions. This is called the *split* factor.

The metrics used quantify these criteria between 0 and 1. A linear combination of these values, with the occlusion factor weighted most heavily, has proven to be successful, e.g.,

partition priority = .5*occlusion + .3*balance + .2*split.

To satisfy Objective 2, the recursive process terminates when no partitioning plane has a partition priority exceeding a user-defined threshold or when a maximum tree-depth is exceeded. The process generates a tree with interior nodes representing binary separating planes and leaf nodes representing cell volumes. Several people have noted that feedback from the second phase of the pre-processing algorithm, described in the next section, could be used to adjust the model-subdivision [Prins90].

If one runs this function on the "planes" in our simple example floor plan, the wall that separates rooms 2 and 3 from room 1, the plane y=1, has a higher partition priority than the wall that separates room 2 from room 3, the plane x=1, based on its higher occlusion factor. This yields two cells, room 1 and the combination of room 2 and room 3. Recursively evaluating our heuristic function on these two cells suggests that room 2 and room 3 can be further split into two cells along the plane x=1 (figure 1.6).

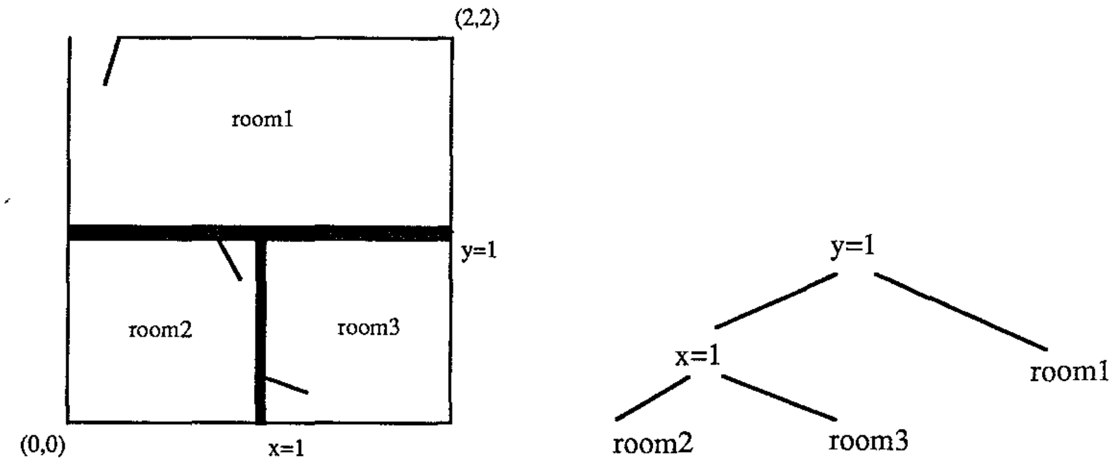


Figure 1.6. Left, the subdivided floor plan. Room1 is separated from room2 and room3 by the plane y=1. Room2 is then separated from room3 by the plane x=1. Right, the corresponding tree data structure. Interior nodes represent splitting planes and leaf nodes represent cell volumes.

1.2.3 Potentially Visible Set calculations

After model-space subdivision, the subset of the model potentially visible to an observer inside each cell is computed and stored with the cell. If the cell is completely sealed, that is, its boundary is composed of opaque surfaces, then this is easy to do. The potentially visible set for the cell is simply the set of polygons that intersect the cell. However, if the cell has holes in its boundary, called *portals,* then the problem is more difficult.

In our simple example, the only portals are doors. In real-life datasets, hallways, stairwells, elevator shafts, windows, and oddly shaped rooms give rise to other portals. Portal geometry is defined only implicitly as the absence of polygons in the boundary of the cell. Explicit polygonal descriptions of the portals are obtained by computing the boolean difference of the cell boundary and polygons lying in the cell boundary planes. Algorithms that perform boolean set operations on co-planar sets of polygons can compute the explicit polygonal definitions of the portals [Ottman85], [Weiler81], [Airey89c]. Section 3.2 describes one such algorithm.

We call the question of what external polygons one should add to the potentially visible set for a cell the *polygon-to-cell* visibility problem. This can be reduced to another problem. One really has to worry only about what can be seen from the cell portals, which can each be represented with polygons. Taking the union of what is visible from all the cell portals of a cell solves the cell-to-polygon visibility problem.

Unfortunately, this is also a difficult problem. One needs to know what is visible from a portal which is an *area,* an infinite albeit bounded set of viewpoints. Call this problem the *polygon-to-portal* visibility problem.

Computing the PVS is equivalent to identifying the polygons that receive direct illumination from an area light source [Nishita85], [Chin89], (the portal acts as an area light source). This is similar to the underlying problem of surface-to-surface reflection that appears in models of global lighting effects, such as the radiosity lighting model. Other researchers have examined a related problem in two dimensions which deals with visibility from an edge [Avis86], [O'Rourke87].

Since algorithms to compute the *exact* solution for the portal-to-polygon visibility problem are very complex, I have developed two complementary classes of algorithms to compute approximations to the exact solution. These are detailed in Chapter III.

One class of algorithms uses point sampling and may underestimate the set of polygons to add to the cell's potentially visible set. This is analogous to the use of point sampling in radiosity solutions. In fact, the same ray-polygon intersection library is used in the Walkthrough radiosity implementation [Airey90b].

Another class of algorithms establishes occlusion relationships among polygons. This is similar to the computation of shadow volumes [Crow77]. Since exhaustive computation of shadow volumes is expensive, the only alternative is to compute a partial solution. This may overestimate the set of polygons to add to the cell's potentially visible set. Since the exact solution is bracketed by these two algorithms, one hopes they can be combined into a more accurate algorithm in the future.

Currently, these approaches are both expensive. In practice we use mostly the sampling-based methods because they are less expensive than the occlusion-relation based methods, although they do occasionally miss potentially visible polygons.

Just as a radiosity solution can be accelerated with graphics workstation hardware [Baum90], the ability to generate environment samples quickly can be used to accelerate a sampling-based approach. For radiosity, the environment samples are not colored pixels, but polygon identifiers. In the case of radiosity, the polygon identifiers and the pixel location are used to construct form factors. In this case, after rendering the view from a sample point on the portal, any polygon whose identifier appears in the frame buffer is added to the PVS under construction. Note that reading the sample values, the polygon identifiers, back from the frame buffer must be efficient also; low bandwidth here can defeat this idea.

1.3 Results

I have run this algorithm on a few databases and compiled statistics to document the speedup results. The databases include

• A 7125-polygon model of Sitterson Hall, *Sitterson1*. Walls are represented by single polygons with separate colors for the front-facing and back-facing sides. AutoCAD was not used for this model. (Modelled by Dana Smith from plans by Phil Freelon of O'Brien and Atkins)

• A second model of Sitterson Hall was constructed with AutoCAD, *Sitterson2*. This model consists of over 22,000 polygons. It consists mostly of polygons that are designed to be seen from only one side. The walls have thickness and are modelled with a pair of polygons. The lobby portion of this model, *lobby2*, appears in a Siggraph '89 video [Airey89b]. *Lobby2* has 3949 polygons. (Modelled by Penny Rheingans.)

• The Orange United Methodist Church Fellowship Building. An early version with 7812 polygons is called *Chur*ch1. (Model by Penny Rheingans from plans by Wesley McClure and Craig Leonard of McClure NBBJ.)

• A later version of the Church consists of over 12,000 model polygons. Since the radiosity process increases the number of shading patches that must be stored in display memory by about an order of magnitude, a 6037-polygon subset was used because of display memory limitations. This subset, called *Church2,* consists of the main meeting hall and a few adjoining rooms, including a fully furnished kitchen.

• This dataset was later re-enlarged to 6974 polygons by adding part of the basement. This version of the church is called *Church3*. (John Alspaugh helped model and manage later church databases). The *Church3* database is featured as a demo for the SGI Iris 4D VGX series machines.

Table 1.1. summarizes the results of the model-subdivision algorithm on these datasets.

The best results, as one would expect, comes from the two complete models, *Sitterson1* and *Church1*, buildings with many rooms, easily detected pockets of PVS coherence. These have the highest depth complexity. *Sitterson1* was subdivided into 269 cells. The cell with the largest potentially visible set had 2195 polygons to display. The average number of polygons to display was a little more than 230. The speedup was 3.25 in the worst case and 30 in the average case.

Data	polys	cells	polys/cell avg.	max.	speedup avg.	min.
Sitterson1	7125	269	230	2195	30.98	3.25
Lobby2	3949	54	466	2550	8.47	1.55
Church1	7812	108	291	2055	26.85	3.80
Church2	6037	16	1887	3477	3.20	1.74
Church3	6974	106	767	3768	9.09	1.85

Table 1.1. Summary of Model Subdivision speedup results.

The additional display memory required to store the data structure generated by the visibility pre-computation is reasonable, about 20%. The main requirement is the need to store potentially visible sets for each cell. Since several cells may see each polygon, there is a potential for large display memory use unless polygon descriptions are shared among cells. The polygons need be represented only once; the PVS for each cell is composed of references to the polygons. From the numbers in Table 1.1 for the Sitterson model, one can see that, on average, about ten cells can see each polygon. This means one needs about 10 more words per polygon to store the pointers. Since the storage required for a color-interpolated quadrilateral is about 200 bytes, the total display file size is increased by about 20%. The storage requirement for the other databases is less.

1.4 Speedup Methods Used by Other Interactive 3D Systems

Most interactive 3D applications can be implemented with some general-purpose technology. All use some form of the well-known image generation pipeline [Molnar89].

In general, however, it is possible and usually desirable to capitalize on certain properties of a given application to increase performance. Specialized hardware may be built to capitalize on application specifics if economics allow it. A less drastic solution is to build specialized software.

When a software approach is used, the strategy of pre-computation is common. Some classes of applications and the methods they use to achieve usable speed follow.

1. *Vehicle* simulations, such as flight simulators, large ship simulators for harbor pilots, or automobile simulations [Schachter83],[Deyo88].

2. *Object* simulation for the visual appearance of mechanical parts or anatomical structures.

3a. Man-made indoor *environment* simulations for buildings or ships, (UNC's Building Walkthrough is in this category).
b. Man-made outdoor *environment* simulations for parks, planned communities, etc.

1.4.1 Vehicle simulations

Vehicle simulations, such as flight simulators, can attain 60 Hz update rates with textured, anti-aliased images including atmospheric effects such as haze and fog. To help achieve such high performance, designers often assume constraints on the models. Grossly-accurate, mostly-static databases composed of relatively few, large, textured polygons are typical of landscapes used in flight simulators [Schachter83],[Abram87].

Because the emphasis is on the simulation experience and not the model, it is reasonable to spend a large amount of time hand-tooling a model that fits the constraints of the system. Since the model will be used over and over again, modeling expense is amortized over many simulations.

Limited depth complexity and large polygons make hard-wired, image-order, or scan-line algorithms, and priority graph approaches very effective[Sutherland74]. The bottleneck in the most general purpose graphics workstations [Akeley89], which typically use the Z-buffer method for hidden surface elimination, is the limited bandwidth to the frame buffer and Z buffer. Image-order algorithms that work on databases with low depth complexity can avoid this bottleneck because visibility is determined before information is written to the frame buffer [Molnar89].

The lighting models used in flight simulators may be relatively simple because direct sunlight is responsible for most of the shading effects seen on the landscape. Surface interreflections, which are responsible for much of the expense in other lighting models, such as ray tracing or radiosity, can be disregarded [Hall88].

1.4.2 Object display applications

Graphics workstations suitable for computer aided engineering (medical and mechanical) have been available for several years. They have taken advantage of the application specifics to achieve the needs of the users with minimal cost. The displayed data is typically not an environment. It is a tie-rod or distributor cap or kidney some other object than can be conveniently viewed through the computer window. An object may be comprised of many small polygons derived from a higher order surface description. Since the object typically is inspected individually and is mostly convex, expensive lighting models that account for surface interreflections are usually unnecessary.

Graphics workstations from Silicon Graphics, Stardent, Hewlett-Packard, Evans and Sutherland and others provide enough computational power to transform and shade over 100,000 polygons a second [Akeley89]. On the other hand, at this date they provide frame buffer bandwidth for about 40 million pixels a second. This directly implies an average polygon size with a dimension of about 20 pixels on a side. If the polygons are large, say 256 pixels on a side, the frame buffer bandwidth bottleneck will prevent the display of more than about 600 polygons a second [Molnar89]. Furthermore, they often cannot generate more than 30 frames a second for any model. For example, the Stellar GS1000 graphics supercomputer renders into virtual pixel maps in main memory and then must copy them into video memory. While they can render a large number of polygons, this prevents them from achieving the 30-60 updates per second that is necessary for out-the-window simulation.

Of course, workstation vendors are continually working to improve their machines, so these limitations may be overcome in the near future.

1.4.3 Environment simulations

Simulations of man-made environments such as buildings and ships have a different emphasis than either of the above applications. The accuracy of the model being displayed is more important than the simulation experience. Environment simulators cannot move building walls and ship bulkheads to simplify visibility priority calculations as features on a landscape can be moved to help vehicle simulation image generation. Furthermore, they cannot justify spending many man-hours hand-tooling the database and preparing it for display because the database will not be used for many hours of lucrative simulation, but only long enough to find potential problems or advantages of the design.

Poorer image quality (aliasing effects) and slightly slower update rates may be tolerated because of the emphasis on the model more than the kinetic experience. On the other hand, a building or ship is primarily designed to house human beings and cannot be evaluated in the same manner in which one evaluates the effectiveness of an engine crankshaft. It must be experienced *somehow*. The structure is most easily examined within a simulation environment similar to the vehicle simulations, but without a vehicle. A balance between model realism and kinetic fidelity must be achieved. Something approaching real-time, say greater than six updates per second, is necessary.

As an aside, the Walkthrough team has noticed that fewer updates are necessary for the individual operating the interactive controls than for observers. The operator appears to be doing some type of interpolation between frames because he knows where he wants to go. He has some idea what the next frame will look like and can use the time between frames to mentally smooth the transition. The observers are less sure what the next image will look like. Did the operator turn left, right, up or down, stop, speed up, etc?. Thus he cannot interpolate to the next image, he must wait for it to appear. It is likely that the observer is doing some type of extrapolation, i.e., if the last few images were from a straight line path then the observer will expect the next image will be also. When the next image turns out to be a hard left turn, he may be disoriented, whereas the operator will not.

The shading requirements of architectural simulations are also different from those of vehicle simulations. Surface interreflections are very important to the architectural evaluation. The radiosity lighting model, which models diffusue interreflection, is substantially more effective than Phong and Gourand lighting models which model local reflections.

Since the penalty for a poorly-designed building is usually only dissatisfaction and not a threat to life, as poor flying is for a pilot, the economic forces have not been sufficiently strong to push development of specialized commercial systems for architectural visualization.

1.4.4 Other Pre-Computation-based Speedup Schemes

Several researchers have devised software approaches that use pre-computation to take advantage of some aspect of a certain class of applications so as to increase performance.

Baum, Winget and Garlick use a spatial subdivision based on an octtree to accelerate clipping operations in a large architectural structure [Garlick90]. They note that the number of polygons actually in the viewing frustum is often a small fraction of the total number of polygons in the model. Their clipper first considers the root volume. If it does not intersect the viewing frustum, then it does not need to be displayed. If it is completely contained within the viewing frustum, then all the polygons in the volume must be rendered. If the volume intersects the viewing frustum but is not contained completely within the frustum, the volume is split and the above operations are applied recursively to its children.

Hubschman and Zucker introduced the idea of using frame-to-frame coherence in animations to decrease the time required for hidden-line removal [Hubschman81]. Their method is restricted to convex non-intersecting polyhedra.

The Binary Space Partition tree (BSP-tree) has been used to display rigid polyhedral scenes in near real-time by automatically pre-computing a structure which gives relative depth-ordering for faces in the model [Naylor81]. The display of a single frame from some viewpoint with hidden-surfaces removed involves traversing the BSP-tree to generate a list of polygons in back-to-front order. This allows polygons painted later to overwrite those painted earlier. Each frame is generated by a separate and complete traversal of the BSP-tree structure. The BSP-tree approach requires drawing all of the polygons in the model for each frame. There is a potential for the number of polygons to increase up to the cube of the original number of polygons due to splitting, but there is much experimental evidence to suggest that the number is increased by a only a single digit multiple in practice [Fuchs83].

Shelley and Greenberg used frame-to-frame coherence for the generation of an animation sequence corresponding to a smooth viewpath through a 3D environment [Shelley82]. A smooth, interactively defined viewpath is represented as a B-spline, and exploited to reduce the expense of the sorting and culling operations for visible line/surface computation. Although the viewpath was specified interactively, the computation of the appearance of the scene along the viewpath was done off-line.

Denber and Turner describes a differential compiler for computer animation [Denber86]. This is designed to playback pre-computed raster images at animation rates. The differential compiler performs temporal domain image data compression using frame replenishment coding on successive frames of animation stored in memory as bitmaps and saves only the differences. A small run-time interpreter then retrieves and displays the differences in real-time to create the animated effect.

Plantinga, Seales and Dyer present a complicated algorithm to pre-compute the visible portion of an object with respect to sets of viewpoints. This work is based on theoretical computer vision work on aspect graphs [Plantinga89]. They demonstrate their algorithm on two small databases, 416 and 384 polygons. They have implemented wire-frame rendering with orthographic projections and viewpoints limited to great circles on a view sphere. The algorithm may be extended to arbitrary viewpaths with true hidden surface removal although perspective projection makes this very difficult. This algorithm cannot be used with existing hidden surface removal algorithms. It remains to be seen whether the storage required to store the ASP (aspect graph) structure will be manageable for large databases.

1.5 Guide to the Chapters.

The Chapters in this dissertation are arranged in a very straightforward fashion. This chapter provided an overview of the methods and results. The pre-processing method is naturally divided into two phases, model space subdivision and PVS computation. A chapter is devoted to each phase. Prior work and projected future work is discussed integrally with each chapter.

One problem that is only briefly mentioned in the overview is the computation of the geometric definitions of a cell's portals. This requires sophisticated methods akin to solid modelling methods. A large part of Chapter 2 is dedicated to an algorithm to solve this problem and similar problems with set operations on co-planar polygons. Techniques from the theoretically oriented computational geometry literature are adapted to real-world data.

Chapter 3 is devoted to techniques for computing the PVS for a cell. Two different approaches are explored.

Chapter 4 discusses the Walkthrough implementation of the radiosity lighting model and the issues involved in integrating it into our system.

Chapter 5 reviews contributions of this work to visual simulation and recommends future research directions.

Chapter II

Model-Space Subdivision - searching for PVS coherence

Chapter I noted some properties of architectural databases. This chapter concentrates on the following property:

Property 2.1 The set of polygons that contributes to each view changes slowly as the viewpoint moves, except when going through walls and floors or crossing portals.

Our goal is, during a pre-processing phase before display,

1. to construct cells of viewpoints that have coherent PVSs.

2. to identify the potentially visible subset (PVS) of the model polygons for each cell.

If the PVS for each cell is significantly smaller than the entire model, then during display, we generate images faster by only processing the PVS for the cell that contains the current viewpoint. Since the update rate is generally a function of the size of the model, the update rate increases in proportion to the difference in size between the current PVS and the model. This technique improves update rate by a *ratio* that increases with the size of the model.

We are essentially searching for pockets of PVS coherence. Coherence of data is a property that graphics researchers have exploited for many years. Spatial coherence is the tendency for the characteristics of a scene to be locally constant across space. Scan-line algorithms, for example, take advantage of spatial coherence. Temporal coherence is defined similarly.

2.1 The Partitioning Machinery

An overview of the partitioning process is presented in Section 1.3.1. It explains what the algorithm does, but many important details were omitted. This section discusses the subdivision process in more detail. The two Objectives and two Restrictions from Section 2.2.1 that guide the partitioning machinery are repeated here.

Objective 1.	Minimize the size of the potentially visible sets.
Objective 2.	Minimize the number of cells; subdivide a cell only if the child PVSs are significantly smaller
Restriction 1.	It must be easy to find the cell that contains the current viewpoint.
Restriction 2.	The partitioning process must be automatic.

Restriction 1 implies the use of some type of data structure suitable for range searching. I chose recursive binary partitioning planes. An initial volume is split with a plane yielding two child volumes. The process is applied recursively to each of the child volumes. If most polygons are *axial*, normal to one of the coordinate axes, it is usually sufficient to restrict splitting plane orientation to axial planes. The result of this process is a binary tree data structure. Splitting planes are represented with interior nodes. The exterior tree nodes, or leaves, are volumes. This data structure allows the cell containing the current viewpoint to be found quickly. Starting from the root, one descends down the tree structure and evaluates the equation of the splitting plane at the viewpoint. Note that in the case of axial planes, this simply a comparison. The choice of which branch to take at each step is based on this comparision. The leaf that is reached in this manner represents the volume containing the viewpoint. The average number of comparisions used to find the cell containing a viewpoint is the same as the average depth of the tree which is proportional to the logarithm of the number of cells, assuming the tree is reasonably balanced.

To satisfy Objective 1 and Restriction 2, I devised a heuristic function to choose the splitting planes used in the recursive binary subdivision scheme. Since one wants a splitting plane that is a natural PVS coherence boundary, we look for a splitting plane that is largely opaque. It is sufficient to limit the choice of splitting planes to those that contain polygons. The function evaluates each plane containing a polygon for its suitability as a separating plane. There are three separate characteristics involved in the choice of a splitting plane:

• How well does the plane hide the two sides from each other: the *occlusion* factor of the split. A floor hides much better than a wall with a door in it. The occlusion factor can be computed by summing the area of polygons in the plane, clipped to the dimensions of the cell, and dividing by the corresponding cross sectional area of the cell. (This assumes no overlapping polygons).

• How evenly does the plane separate the model: the *balance* of the split. Balance is important because one wants the subdivision tree to be reasonably balanced. One cannot afford searching through a linear list of subdivisions to find the cell containing the current viewpoint. The balance factor can be computed by counting the number of polygons on either side of the proposed split and dividing the smaller number into the greater number.

• How little does the plane split individual polygons: the *split* factor. Polygons that are split will have to be put in the potentially visible sets of both partitions. If every polygon is split, the PVS size will not decrease. The split factor can be computed by counting the fraction of polygons that crosses the proposed splitting plane.

The metrics used quantify each criterion between 0 and 1. A linear combination of these values, with the occlusion factor weighted most heavily, has proven successful, e.g.,

partition priority = .5*occlusion + .3*balance + .2*split.

This heuristic function was derived by experience. One of the first things I tried as a graduate research assistant on the Walkthrough project in the fall of 1986 was splitting up a model of Sitterson Hall by hand. Using natural human experience and very little deliberate study, it was clear that splitting up the building by floors was the first thing to do. An early Walkthrough implementation, on the Ikonas display, used a manual partitioning scheme. A value was stored in every polygon description. The user could set a mask that would be compared against the value stored with every polygon as it was being rendered. If the logical-and of the mask and the value was zero, further processing on the polygon was aborted.

It is difficult to choose the best partitions beyond the floors. I devoted some thought to what made floors good splitting partitions, quantitatively, and tried to extend that idea to walls. I wrote some simple software that computed balance, split and occlusion statistics for each plane in the model. The floors were indeed rated highest using these statistics, so there was some reason to believe that this idea could be used successfully to choose further partitions. I tried to use this information and more concentrated study to pick the partitions by hand. It took a long time to subdivide the model with these few simple programs. There was no code written to compute the potentially visible polygons outside the cells. Only the polygons inside the current cell were displayed.

The partitioning scheme that exists now is simply an automation of that effort. It was refined on 2D floorplans that allowed me to see the entire subdivision at a glance. It is certainly conceivable that another good PVS coherence search algorithm exists. For example, one could compute the polygons visible in a 360 degree field of view from uniformly spaced viewpoints throughout the building. If adjacent viewpoints had significantly differing PVSs, a new sample viewpoint halfway between those points could be computed. The process could continue adaptively, until neighboring viewpoints had PVSs that agreed within some tolerance. Likewise, neighboring viewpoints that had similar PVSs could be merged. The cells could be reconstructed from these viewpoints with an octtree.

To satisfy Objective 2, the PVS coherence searching process stops subdividing when any of several user-defined limits is exceeded. This list is not meant to be complete.

• If no partitioning plane has a partition priority exceeding a user-defined threshold, the cell is not split. This will happen when the cell is a single room and there is nothing that will effectively hide one half from the other.

• If the volume of the cell is below a certain value, the cell is not split. This can be used to prevent excessive subdivision if the size of an average room is known.

• If predefined limits on the depth of the subdivision tree are exceeded, no further splitting of that branch is allowed. This allows a hard limit to be placed on the number of cycles used to identify the cell containing the current viewpoint.

• If the number of polygons inside the cell is less than a specified value the cell is not split. If one knows how many polygons the system can display safely within some time limit, this can be used to prevent unnecessary subdivision.

Now we examine the operating details of the partitioning machinery and split selection functions. There are several points that have not been revealed.

For example, implementing the partition priority function naively is expensive. The balance and split criteria each involve looking at all the other faces for each of the candidate planes. This would be quadratic in the number of faces. I use an approximation that takes linear time. Note that databases with 10,000 faces imply that a linear time algorithm can run 10,000 times faster than a quadratic algorithm, assuming the constant factors are similar.

Optimizing the speed of partitioning is important because although the partitioning step takes much less time than identifying the PVSs for each of the resultant cells, it is largely sequential in nature. The PVSs for all the cells can be computed independently of each other. This makes it relatively easy to parallelize, assuming the existence of tools for distributed computing that allow available computers to be assigned to different cells. In such a distributed computing environment, it is important to accelerate the sequential parts of the algorithm.

2.1.1 Partition.c: C code to compute interior node splitting planes and exterior node cell volumes.

The source code reveals the details. Highest level source code is in appendix A. The supporting code can be found in appendix C.

2.1.1.1 partition()

First we will examine the highest level function, `partition()`. Following this we will examine the function that selects the splitting planes, `select_split2()`, and see how a linear approximation can replace the naive quadratic algorithm for choosing the best splitter.

Figure 2.1 provides a graphical overview of the operation of `partition()`.

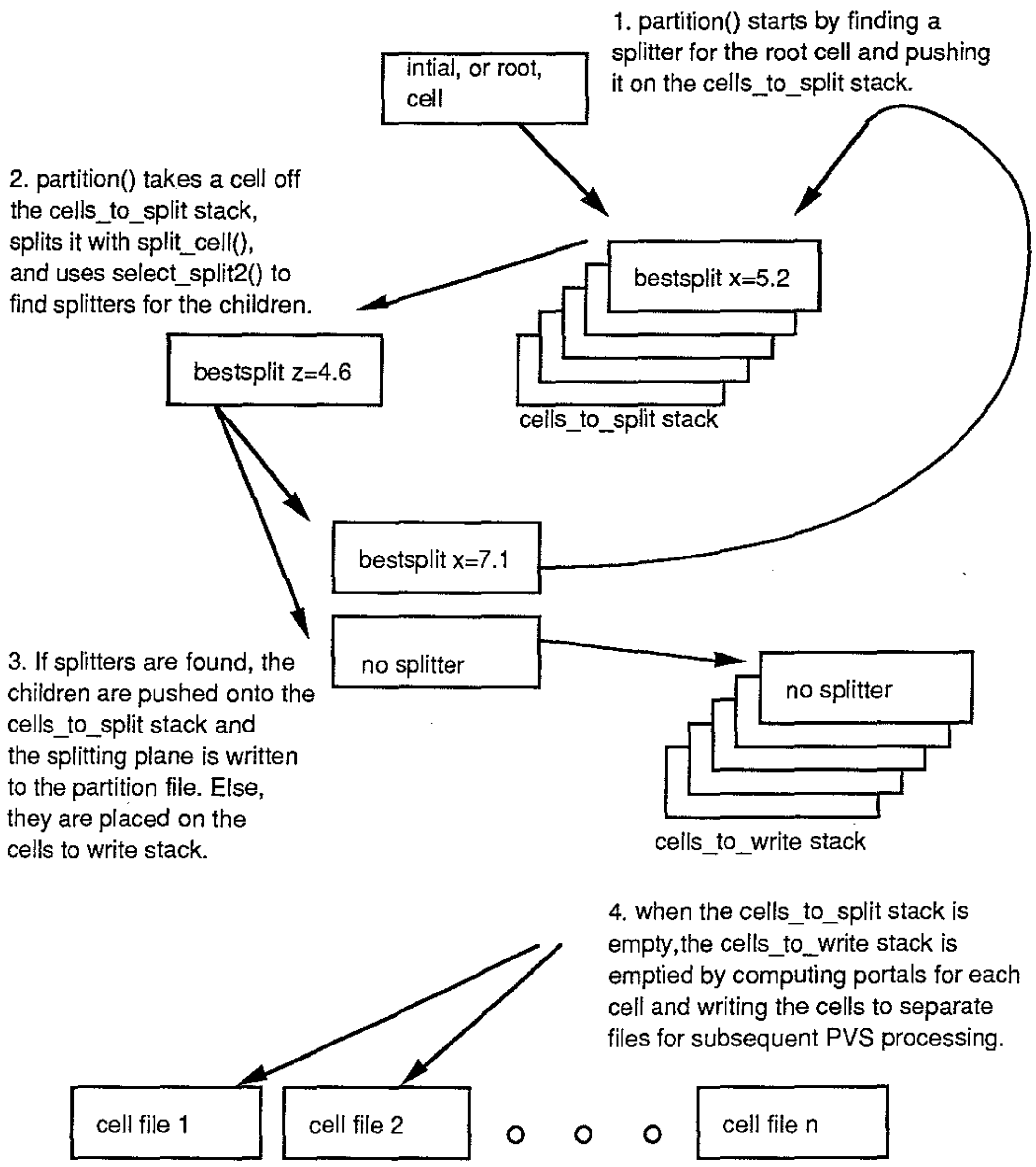

1. partition() starts by finding a splitter for the root cell and pushing it on the cells_to_split stack.
intial, or root, cell
2. partition() takes a cell off the cells_to_split stack, splits it with split_cell(), and uses select_split2() to find splitters for the children.
bestsplit x=5.2
bestsplit z=4.6
cells_to_split stack
bestsplit x=7.1
no splitter
3. If splitters are found, the children are pushed onto the cells_to_split stack and the splitting plane is written to the partition file. Else, they are placed on the cells to write stack.
no splitter
cells_to_write stack
4. when the cells_to_split stack is empty,the cells_to_write stack is emptied by computing portals for each cell and writing the cells to separate files for subsequent PVS processing.
cell file 1
cell file 2
o o o
cell file n

Figure 2.1 A graphical overview of the operation of `partition()`. Two stacks of cells are used. Both start empty. `partition()` pushes the root cell on the cells-to-split stack and begins to split cells. The results of the split are two new child cells. If splitting planes can be found for the child cells they are pushed on the cells-to-split stack. Otherwise, they are pushed on the cells-to-write stack. The splitting plane is written out to the partition file. The cells-to-write stack is processed after the cells-to-split stack empties.

The function `partition(fp,rootname,topinst)` is the highest level function called by the subdivision program. It takes a file pointer and a geometry description of the model. It generates a description of the splitting planes in the interior nodes of the subdivision tree and calls functions to compute the portals for each of the cells at the exterior nodes of the partition tree.

The first thing `partition()` does is establish the orthogonal extents of the database as the *root* cell. The function `initialize_first_cell()` handles this task along with other housekeeping chores. The computed extent serves to define the initial volume of space to be subdivided. If we expect to view the database from the exterior, we can increase the extents of the database by some fixed amount. Note that this does not necessarily limit the user to travel within this extended volume. For practical purposes, there is some fixed distance, dependent upon the building, at which the PVS for the cell does not change appreciably as the viewer retreats. I have simply tripled the dimensions of the original geometric extent of the building volume to get the new extent.

The function `index_build()` constructs a table of indices that help `select_split2()` choose good splitting planes.

The section of code that computes the splitting planes is a stack-based operation. The initial cell, which contains the entire model, is processed by `select_split2()`. If the routine `select_split2()` can find a suitable splitting plane, the cell is pushed on the cells-to-split stack. If `select_split2()` fails to find a suitable splitting plane, the cell will be pushed on the output cell stack.

Assuming that `select_split2()` does find a splitting plane for the initial cell , or tree root, there will be a cell on the stack. The `while` loop pops a cell off the stack and proceeds to use the splitting plane found by `select_split2()`, and stored with the cell, to split the cell with the routine `split_cell(t,t->ng,t->nl)`. The variable `t` is a pointer to a tree node. The `ng` and `nl` fields point to the child cells that are "not greater" and "not less" than the splitting plane field, respectively. Admittedly, names such as *le* and *ge* for less-than-or-equal and greater-than-or-equal may have been wiser. The term *equal*, however, was tied up in my mind with the meaning of lying in a plane.

The splitting plane is written to the partition file with `write_node_ascii()`.

The `select_split2()` function is then applied to the two child nodes. If a splitting plane is found, they are pushed on the cells-to-split stack. Otherwise, they are placed on the output cell stack. When the stack is empty, the partition file is closed.

The second `while` loop processes the cells-to-write stack. It calls `compute_portals()` for each of the six boundary cell faces and writes each cell and its portals to its own file. The function `compute_portals()` is quite complicated and involves thousands of lines of source code. Section 2.2 is devoted to explaining the algorithm used by that function.

2.1.1.2 select_split2()

The `select_split2(t)` routine determines the best subdivision plane for a cell. It takes a pointer to a tree node, which includes the current cell description. If it finds a suitable splitting plane, it stores that information in the cell and returns a true value.

`Select_split2()` uses data structures built by `index_build()` to choose the best splitting plane (figure 2.2). These data structures are also important for other algorithms that process successive parallel planes containing polygons. This is a fundamental operation that we use in ray-polygon intersection computation and occlusion testing. Stored with each plane in the data structure built by `index_build()` is a count of the polygons that lie on each side of the plane and a count of the polygons that are split by the plane.

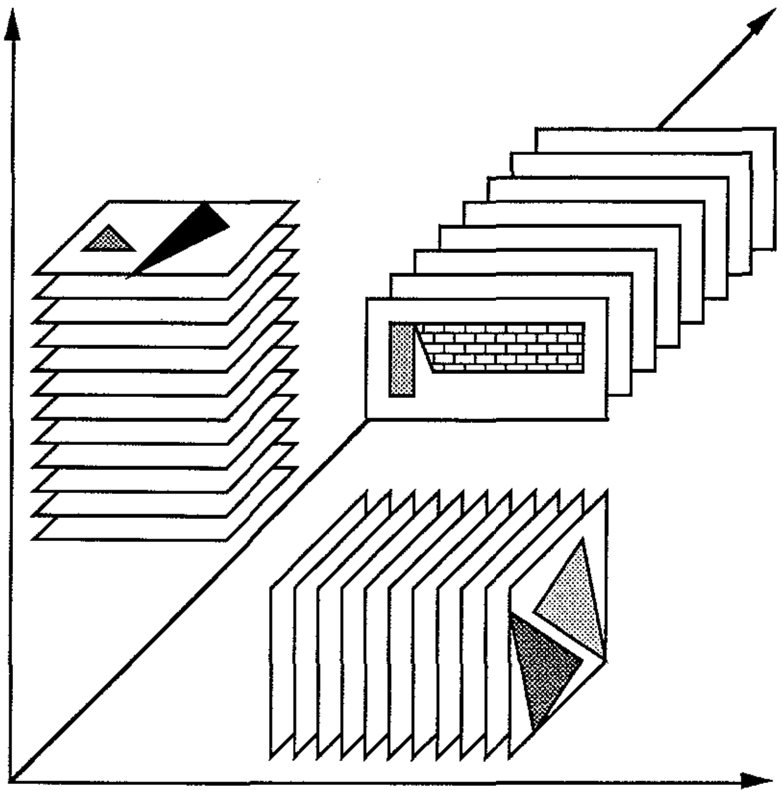


Figure 2.2 Index_build() constructs three sorted lists of parallel planes. Each plane is composed of a list of polygons that lie in it, counts of polygons on each side of the plane and a count of the polygons split by the plane.

The occlusion criterion can be computed in constant time per plane, since it merely involves summing the area of the polygons in the current plane and dividing by the cross section of area in the current cell. Thus only linear time is required to evaluate the occlusion factor for all the planes in a cell.

The balance and split criteria however, require that we compute the relationship of each plane to all the polygons in the current cell. This takes at least linear time for each plane, resulting in quadratic time to compute the value for each plane. To reduce the requirement to constant time per plane we use the values that have been pre-computed and stored in the `coord_index` data structures. For each plane, we have pre-computed a count of all the polygons that are less than or equal (called not greater or "ng" here) and greater than or equal ("nl") to the splitting plane, respectively. A count of polygons split by the current plane is also pre-computed and stored. `Select_split2()` uses these extra statistics for each plane to evaluate the *balance* and *split* values for each plane without comparing every polygon against each plane every time it looks for a splitter. Each plane is compared against every polygon only once.

To get a count of the number of polygons to the "left" and "right" of a plane we compute the difference between the count of polygons to the left of the plane and the count of polygons to the left of the left boundary of the cell. For the right side we compute the difference of the count of polygons to the right of the right boundary of the cell and the count of polygons to the right of the plane. It would be sufficient to use a count for only one side.

The counts are exact only for planes in cells where the corresponding cell dimension is the same size as the initial cell. The counts become less and less accurate as the cell is split into successively smaller and smaller cells. For this reason, `select_split()` attenuates the weight of these two criteria by the depth of the cell.

See Appendix A.1 for the C source code.

2.2 Computing Portals

To compute the portal geometry from a set of polygons and the cell, an axially aligned box, we form the difference of the each of the rectangular sides of a cell and the polygons that lie in the same plane (Figure 2.3).

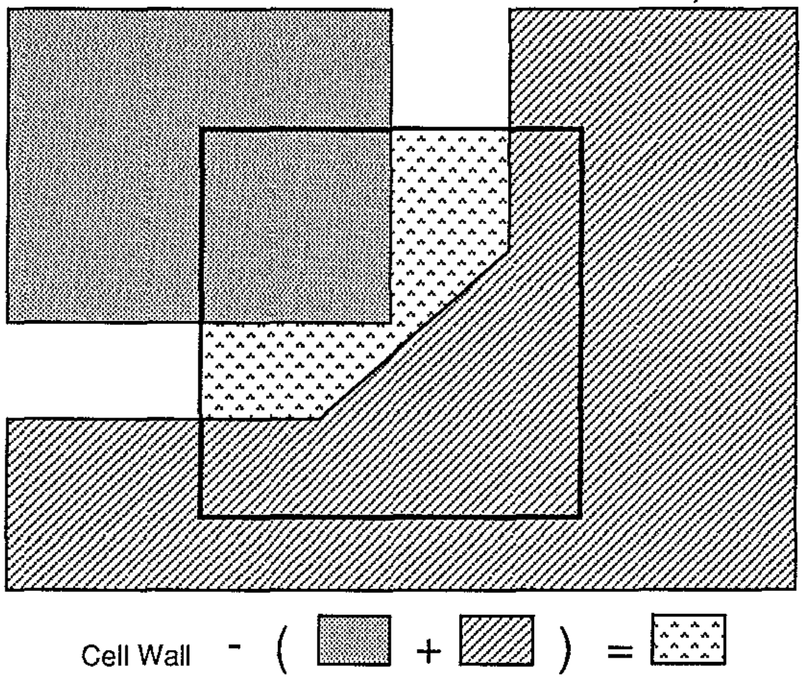


Figure 2.3 Portal computation. The rectangular cell wall is depicted by the bold rectangle. Polygons lying in the same plane as the cell wall are shown with thinner boundary lines. The portals are the set difference between the cell wall and the two polygons lying in the same plane as the cell wall.

The process of computing the difference between the cell walls and the polygons lying in the plane of the cell wall can be accomplished with algorithms that perform set operations on coplanar polygons [Ottman85], [Weiler81], [Airey89c]. Such algorithms are quite complicated. Since we are always dealing with a rectangular cell wall and always perform a difference operation, it is possible that some restricted algorithm could do the job faster and with less coding effort. The Sutherland-Hodgman clipping algorithms appear to be the basis upon which a simple algorithm could be developed [Sutherland74b]. This family of *algorithms provides a simple way to clip polygons to a convex polygon.*

Unfortunately, we are, in essence, clipping the window to the *exterior* of the other polygons. This is equivalent to clipping to a *concave* polygon. This is a fatal stumbling block, and it appears to prevent any simple modification of the Sutherland-Hodgman algorithm to an algorithm that will perform the difference operation.

No small modification of a simple algorithm would do the job, so I looked at the literature for polygon comparison. The most well-known paper in the practical literature is Weiler's polygon comparison algorithm. It definitely can perform the job, but it involves elaborate data structures that did not fit well with my own elaborate data structures. Secondly, the most demanding part of any polygon comparison job is finding all edge intersections efficiently. Weiler's paper did not really address that problem, but rather concentrated on the comparison job, assuming some other algorithm could be used to determine the intersections

I looked to the theoretical literature, in particular [Nievergelt82] and [Mehlhorn84]. These algorithms both use the plane-sweep paradigm, which is simply a rigorous abstraction of the scan-line algorithms used to compute visible surfaces, such as [Sequin85] and all its predecessors. (There seem to be very few cross references between papers in the two fields which suggests parallel and independent development of this idea.)

The theoretical literature usually ignores the harsh reality that floating point number representation is ill-suited for geometric computing. Geometric computing must often compare two very similar numbers. Relative values are the important thing. Traditional floating point was developed to handle absolute values over a large range. Unfortunately, there are also very few satisfactory answers to this problem in the practical literature.

Theoretical literature assumes restrictions in the input which ease exposition of the algorithm. Unfortunately, the generalizations necessary to transform a theoretically advanced algorithm into a practical tool are often non-trivial.

Technology transfer from computational geometry to computer graphics has been slow. The published algorithms are efficient and can be proven correct, but the programmer attempting to implement these algorithms faces many obstacles. The focus of the computational geometry literature is primarily theoretical and secondarily pragmatic whereas the emphasis is reversed for most graphics programmers. The main difficulty faced by the practicing programmer is the restrictive assumptions made by many algorithms about the input. In particular, published geometric algorithms often assume that the input is in *general position*, i.e., that no two input vertices coincide, that no two vertices have the same x-coordinate or that edges intersect only at end points, etc. This makes the analysis of the algorithm much simpler but makes the task of implementation much harder. Input that is not in general position is termed *degenerate* even though it may occur more often in practice than input in general position. Secondary difficulties are those of data representation. The universal unit of information for computer graphics is the oriented convex polygon represented by a list of vertices. Algorithms in the literature may assume some other representation for input or output, such as a doubly-connected edge list (DCEL) which requires translation of data structures.

I have attempted to develop an algorithmically advanced solution to the polygon comparison problem which handles input that appears in practice. Once the algorithm was implemented and fairly stable, many other applications became evident. For that reason, the algorithm is presented in a general setting. The original purpose of the algorithm, to compute portal geometry, is now just one of the many uses of this code.

We have developed an algorithm that handles degenerate input while maintaining good performance. Simply stated, it computes the set operations, union, intersection and difference on polygons. The input may consist of such normally troublesome entities as concave polygons with holes. The output is a triangular subdivision of the plane over which the set operation is true. Although this appears to be rather abstract, many problems encountered by graphics programmers can be cast as set operations on polygons. We list some applications here briefly and discuss them in greater detail in Section 2.2.6.

1) Many graphics algorithms demand convex polygons. Concave polygons, possibly with holes, must be decomposed into convex parts. Obviously, triangles are convex, so triangulation yields a convex decomposition. This is the simplest application of our algorithm but may be the most useful. In section 2.2.2 we discuss less obvious advantages of the triangular subdivision.

2) Generalized clipping operations appear often in graphics applications. For example, Constructive Solid Geometry (CSG) algorithms which operate on boundary representation data need to deal with coincident coplanar faces. If two cubes are abutted and one takes the union of the cubes, it is necessary to remove the portions of the faces that touch since those portions are no longer boundaries; they are inside the union of the cubes. Computing the exclusive-or of those faces solves the problem. The problem of computing portal geometry falls into this category. Another example is simply detecting and counting overlapping polygons (without computing their exact overlap). A wall in a building should not be modelled with overlapping polygons because this can cause rendering errors. Detecting regions where two polygons overlap can be formulated as a modified set expression problem. The algorithm can also be modified to count overlaps.

3) In other applications, it is necessary to cover a curved surface with polygons so that T-junctions do not occur. This structure is known in computational geometry as a *planar subdivision*. If a simpler *tiling*, which allows T-junctions, is used, cracks may appear. Analogously, if a flat surface is shaded we can think of the intensity values as a curved surface. With a tiling, the cracks in the intensity surface appear as shading discontinuities. Since the algorithm can transform a tiling into a triangular subdivision with a union operation, it can be used to eliminate cracks and shading discontinuities; this is important for radiosity applications. See section 2.2.5.3 and figure 2.9 for details.

The portal algorithm is a generalization of a O(n logn) plane sweep algorithm to triangulate a simple polygon [Mehlhorn84]. Plane sweep algorithms without the triangulation feature have been used by the VLSI community to analyse circuits through boolean combinations (set operations) on mask artwork [Syzmanski85], [Ottman85], [Nievergelt82]. The plane sweep paradigm offers several advantages which make it possible to run the algorithm on huge VLSI circuit designs with only a moderately-sized machine. It is iterative rather than recursive, so external devices may be used for very large input sets. Furthermore, the algorithm sweeps a line across the plane and needs memory only for the data along the current sweep line, so the amount of primary memory required at any one time is usually about the square root of input size.

However, VLSI applications can make assumptions about the input such as restricting edges to lie horizontally or vertically. Our presentation is the first we know of which addresses the difficult problems raised by the unrestricted input encountered in graphics applications. We offer a new interpretation of the local geometry of the transition vertex which eliminates the extensive enumeration of cases that would otherwise be necessary to handle real-world input.

We begin with a short note on representation of data in section 2.2.1. Then we present a simple triangulation algorithm in section 2.2.2 and extend it in section 2.2.3 to get the main algorithm. In Section 2.2.4 we discuss implementation issues. Section 2.2.5 presents applications we have implemented ourselves and applications which might interest others.

2.2.1 Why Triangulation?

Although there are many ways to represent polygonal regions of the plane, such as the DCEL (doubly connected edge list) representation of a planar graph [Preperata85], or boolean (union, intersection and difference) combinations of halfspaces [Dobkin88], we chose to represent regions of the plane with a triangular subdivision. Fortunately, it is known that any polygonal region may be triangulated and in fact many different triangulations may exist for the same polygonal region. We represent a triangular subdivision simply by listing the triangles. The triangles are represented by listing the three vertices in counterclockwise order. The coordinate data of the edges is thus represented indirectly. We represent the connectedness information of the edges with a pointer to the three neighboring triangles. Thus a triangle is a list of three coordinate vectors, the vertices, and a list of three pointers to the triangles that share its edges.

The primary reason we have developed our algorithm to output triangles is that many display devices and rendering algorithms require this representation of a surface. However, a triangular subdivision has many other desirable attributes. For example, the number of triangles is proportional to the number of vertices. This means that any operation that was originally implemented in time proportional to the number of vertices, such as computing the area of the region or testing point inclusion can be implemented with the same order of performance by operating on the triangles. If it is of great importance to reduce the number of convex components of a region, we can join some of the triangles in time proportional to the number of vertices to get a good convex decomposition This is fortunate because computing the optimal convex decomposition takes cubic time! Many other problems in computational geometry are trivial given a triangular subdivision [O'Rourke87], [Mehlhorn84]. These include path-planning and two-dimensional visibility calculations both of which are useful to robotics.

We do not require that the input be a triangular subdivision. The input unit is the directed line segment. By convention, the region the line segment bounds lies to the left as we travel from the first vertex to the next. This allows concave polygons with holes to be used in the input. It does, however, preclude self-intersecting polygons such as figure-8's since, in that case, the orientation of some edges are not defined.

2.2.2. A Plane-Sweep Algorithm For Triangulation.

Before we consider the triangulation of an arbitrary region of the plane defined by a set operation on polygons we review a plane-sweep algorithm [Mehlhorn84] to compute a triangulation of a non-self-intersecting, or *simple,* polygon, P.

The plane sweep algorithm puts the vertices of P into a priority queue, which we call the Xqueue, with the vertices ordered lexicographically from left to right and then from top to bottom. A vertex is tagged as a start, bend or end vertex depending upon whether its neighbors in P both follow it in the Xqueue, one follows and one precedes, or both precede, respectively. A typical procedural interface to such a data structure is:

```
xq_init();                  /* initializes the priority queue for use */
xq_delete_min(vertex);      /* returns the vertex with min x coord */
xq_insert(vertex);          /* inserts the passed vertex in the Xqueue*/
xq_term();                  /* frees any memory used by the Xqueue */
```

The Xqueue should be implemented with something that guarantees O(log n) performance for `xq_insert()` and `xq_delete_min()`, such as a heap [Sedgewick88].

The algorithm sweeps a vertical line across the plane from left to right, stepping from vertex to vertex using `xq_delete_min()`. At any point in time, the sweep line defines a vertical ordering on the edges of P that it intersects. Between the edges are *regions*. The regions will be either inside or outside P and furthermore, the sweep line will intersect these *in* and *out* regions alternately. The edges and regions currently intersected by the sweep line and their vertical order are represented by a data structure which we call the Ytable. A typical procedural interface to such a module is:

```
ytbl_init();                /* initializes the data structure for use */
ytbl_insert(edge);          /* inserts the edge into the Ytable */
ytbl_delete(edge);          /* deletes the edge from the Ytable */
ytbl_findabove(vertex);     /* returns the edge (above) the vertex*/
ytbl_findbelow(vertex);     /* returns the edge (below) the vertex*/
ytbl_pred(edge);            /* returns the edge above the passed edge */
ytbl_succ(edge);            /* returns the edge below the passed edge */
ytbl_term();                /* frees any memory used by the Ytable */
```

Ideally, the Ytable should be implemented with a balanced tree such as a red-black implementation of top-down 2-3-4 trees [Guibas78] to ensure O(log n) cost for each of the above operations, excluding `ytbl_init()` and `ytbl_term()`.

The position of the sweep line is advanced by taking a vertex from the Xqueue using `xq_delete_min()` and the Ytable is updated by deleting edges that end at the current vertex and inserting edges that start at the current vertex. This reflects the changes in intersection order of edges of P with the sweep line. This maintenance of the Ytable is common to the *invariant* of all plane sweep algorithms. An invariant is a condition that is maintained throughout the loop of a program. Algorithm proof techniques proposed by Edsger Dijkstra and David Gries depend heavily upon the idea of a loop invariant [Gries81]. Algorithms can be characterized by their invariants and the amount of work that must be done to maintain the invariant during each loop iteration.

The skeleton of any plane sweep algorithm takes the form:

```
sweep()
```

```
{
   vertex v;

   xq_init( list of input polygons);
   ytbl_init( );
   while ((v = xq_delete_min()) is not null)
      transition (v);
   xq_term();
   ytbl_term();
}

transition(v)
vertex v;
{
   1. maintain the Ytable ordering invariant by deleting
      edges that end at v and inserting edges that start at v.

   2. maintain the invariant particular to this plane sweep algorithm.
}
```

Generally a plane sweep algorithm will have some other processing at each transition. In this plane sweep algorithm we associate with every *in* region a chain of vertices, $v_1,...,v_k$ where v_1 and v_k are endpoints of the boundary edges of the region in the Ytable and edges (v_i,v_{i+1}) will become edges of the triangulation. The invariant maintained at each transition that is specific to the triangulation plane sweep is that no triangle can be constructed from any chain. Basically, the chain must be concave or have less than three vertices, i.e. if we closed the chain with an edge from v_1 to v_k we would get either a polygon that is oriented clockwise or a simple line segment.

The important property of this processing step is that if we are given a chain satisfying the invariant, and a new point is added at either end of the chain, it is sufficient to check for a possible output triangle with the new point and its two closest neighbors in the chain. If no counterclockwise triangle can be constructed, then no triangle can be constructed anywhere in the chain. If a triangle can be constructed, the chain is reduced by one vertex and the process may be repeated on the reduced chain. This means that if k triangles can be constructed, $O(k)$ steps will find them and construct them. This action of triangulating up or down a chain when given a chain and a new point is the action that maintains the invariant and produces the triangles.

The maintenance of this second invariant is depicted below (figure 2.4). The left hand side shows a chain of five vertices associated with an *in* region and a new vertex, *vnew*, which has been added to the top of the chain. Since *vnew,v1,v2* form a triangle, *v1* is removed from the chain and the triangle is output. The process is repeated twice more with *vnew, v2, v3* and *vnew, v3, v4*. This leaves the reduced chain seen on the right.

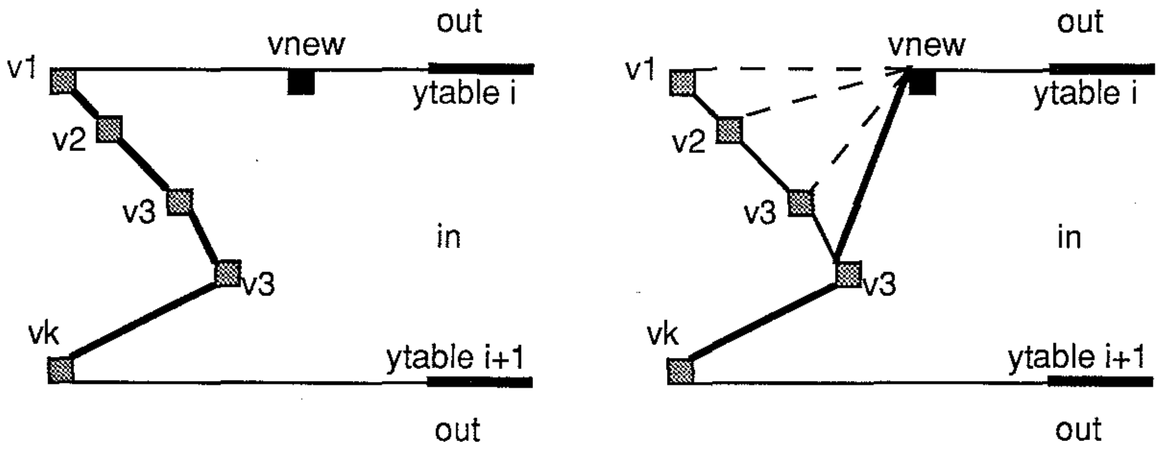


Figure 2.4 Maintenance of the second invariant. Before on the left. After on the right. When a new point, *vnew*, is added to the chain, we attempt to form a counter-clockwise triangle with the new point and its neighbors. The process continues until no more triangles can be constructed.

The algorithm must be designed to maintain this invariant and the Ytable invariant on each transition point. There are three main transition cases to consider depending upon whether the transition vertex has been tagged as a start, bend or end vertex. We consider the action of the algorithm on an example. The progress of a sweep line and the resulting triangles is depicted in figure 2.5. The interior triangulation edges are the same thickness as the polygon if they are part of a chain and are thinner if they have been output. The sweep line appears bent in some instances because vertices with equal x-coordinate values are processed from top to bottom.

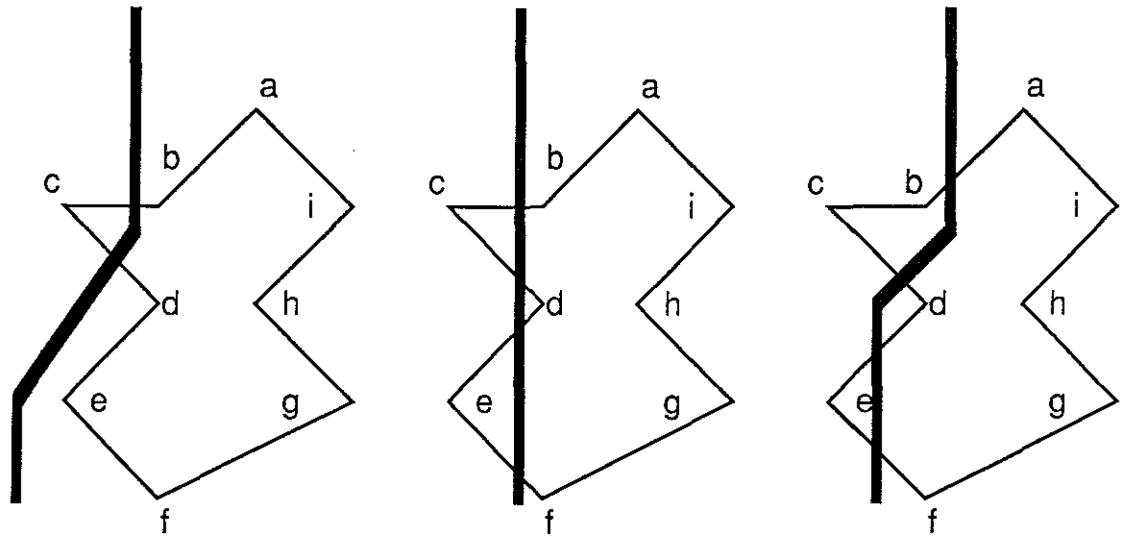


figure 2.5 continued next page

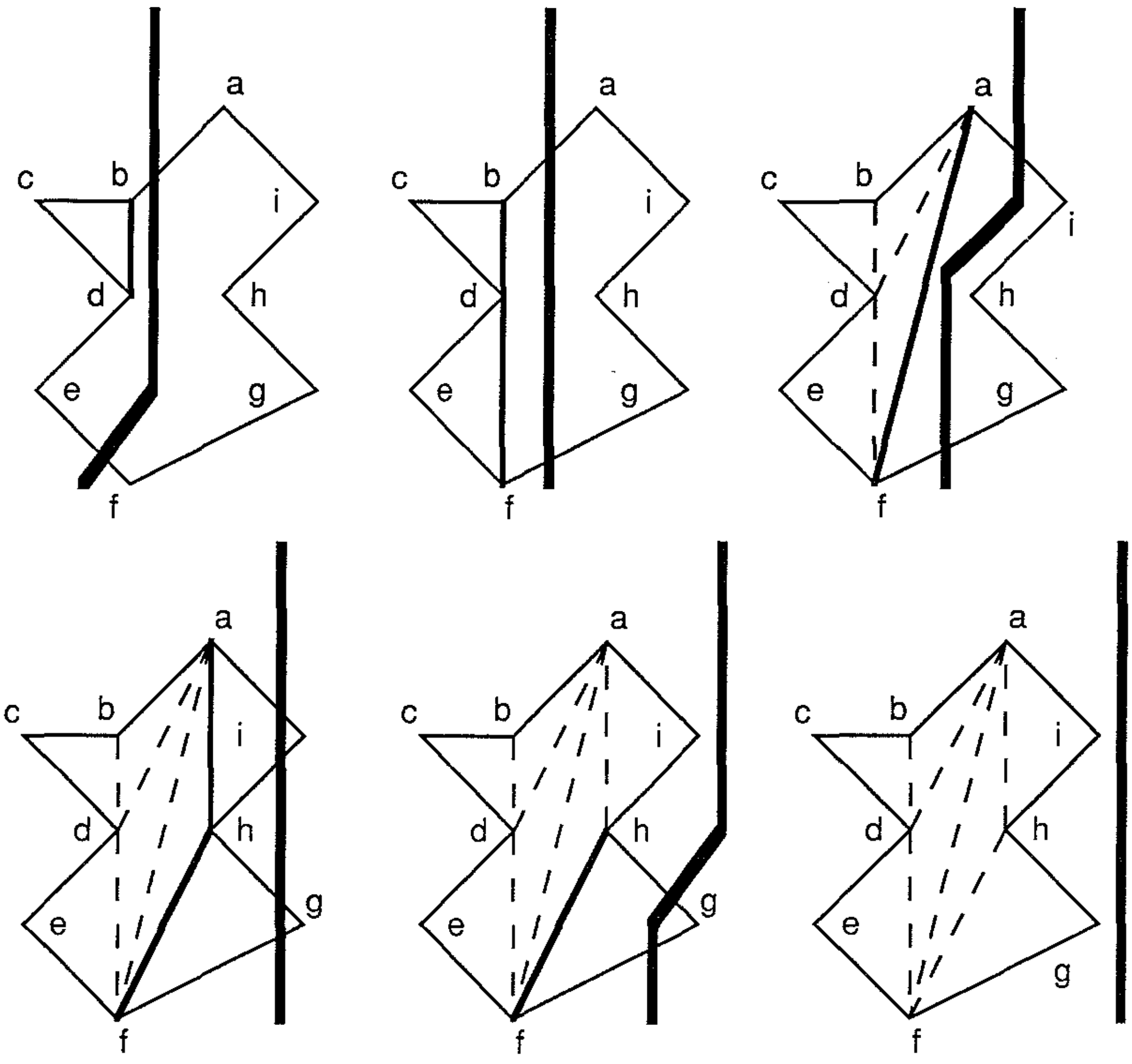


Figure 2.5 An example. The sweep line is depicted crossing a polygon in nine stages, one per vertex. The resulting triangles are drawn with dashed lines. The chains for the current sweep line are drawn with heavy lines.

There are two possibilities for a start vertex. It can appear in an *out* region in which case the action is especially simple. The two edges that emanate from the vertex are inserted into the Ytable. They form a new *in* region and their chain is simply the new vertex. The transition on vertex *c* is an example of this case. If the point appears in an *in* region, the action is slightly more complex. The transition on vertex *h* is an example of this case. The chain that "surrounds" the transition vertex is broken into two chains and the transition vertex is appended to the end of the upper chain and the head of the lower chain. The old chain is broken at its right most point. Any triangles that can be constructed from these new chains are output. In this case the old chain *a, f* becomes *a,h* and *h, a, f.* The latter chain yields one triangle and is reduced to *h, f.*

If the vertex is a bend vertex, we add the new vertex to the appropriate end of the chain and then output triangles if the new addition allows us to do so. Vertex *a* is an example of this type of transition. The chain *b,d,f* becomes *a,b,d,f* and the triangles *abd* and *adf* are output and the chain becomes simply *a,f*. The entry in the Ytable which ended in *a* is replaced by the edge that starts at *a*.

As with the start vertex, there are two possibilities for an end vertex. The transition on vertex *d* illustrates the first possibility where the vertex appears in an out region. The two chains, *b,c* and *e*, are triangulated with the new vertex and then they are joined. The two edges that bounded the *out* region are deleted from the Ytable and the neighboring regions become one region. In this case one triangle was produced when the chain *b,c,d* was reduced to *b,d*. The chains *b,d* and *d,e* are then joined to become chain *b,d,e*. The second possibility, that the vertex appears in an *in* region is illustrated by vertex *i*. The chain becomes closed and is entirely reduced to triangles. In this case there is only one triangle produced. The edges that ended in vertex *i*, *ia* and *hi*, are deleted from the Ytable and the neighboring *out* regions are merged into one *out* region.

The correctness of any part of the triangulation rests upon the idea that the chains are kept "concave" and that maintaining the invariant does not result in overlapping triangles. The only questionable case is when an end point appears in an *out* region. Mehlhorn provides a rigorous proof of correctness for this case. All plane sweep algorithms have the property that maintaining the Ytable and getting the next value from the Xqueue can be performed in O(log n) time. Since there are *n* vertices, the result is an O(n log n) algorithm. Maintaining the chain invariant can also be done within this bound. This follows from the fact that the number of triangles is proportional to the number of vertices and only a constant amount of work was done for each triangle.

2.2.3. Generalizing the Simple Triangulation Algorithm

We now present the extensions and generalizations of the plane-sweep triangulation algorithm necessary to triangulate a region of the plane defined by union, intersection and difference operations on sets of polygons rather than simply the interior of one polygon. In figure 2.3 we need to triangulate the region defined by the difference of the cell wall and the union of the two grey polygons.

First, we will need some mechanism to handle transition vertices formed by the intersection of polygon edges. We will also need some technique to determine what regions of the plane satisfy the set operation. We cannot rely on alternating *in* and *out* regions as in the simple triangulation algorithm.

The last problem is the most difficult to handle in practice. If the input is not in general position, we also must expect vertices to coincide with other vertices and edges. A difficult consequence is that a transition vertex may have any number of edges entering it from behind the sweep line and any number of edges exiting it ahead of the sweep line. This means that each transition vertex cannot be neatly characterized as a start, bend or end vertex. Trying to process vertices that coincide as independent events is not sufficient.

A programmer could methodically transform the explanation of what to do in each of the transition configurations in the simple polygon triangulation algorithm with one case for each of the start, bend and end transitions described in the simple triangulation algorithm above. However, now the number of configurations is no longer finite and cannot be handled with such a taxonomic approach. We propose a new interpretation for these complicated transitions which allows them to be processed without special cases.

2.2.3.1 Handling Transitions Introduced by Edge Intersections

Intersections of polygon edges in the input must be detected so that they may be treated as transition points in the same sense as vertices of the input polygons. As each vertex is processed, newly adjacent edges are checked for intersections. Each time an edge is inserted into the Ytable we check whether it intersects the edges that are adjacent in the Ytable. Similarly, when an edge is deleted from the Ytable we check whether the edges that are now adjacent intersect. When an intersection is processed, the intersecting edges are exchanged in the Ytable and checked against their new neighbors. This catches all intersections (figure 2.6). When an intersection is detected and it is ahead of the sweep line it is inserted into the Xqueue using the `xq_insert()` routine. This will not change the O(logn) complexity of operations done at each transition but it may increase the number of transitions. In the worst case the number of intersections could be $O(n^2)$ but in practice it is much less than n.

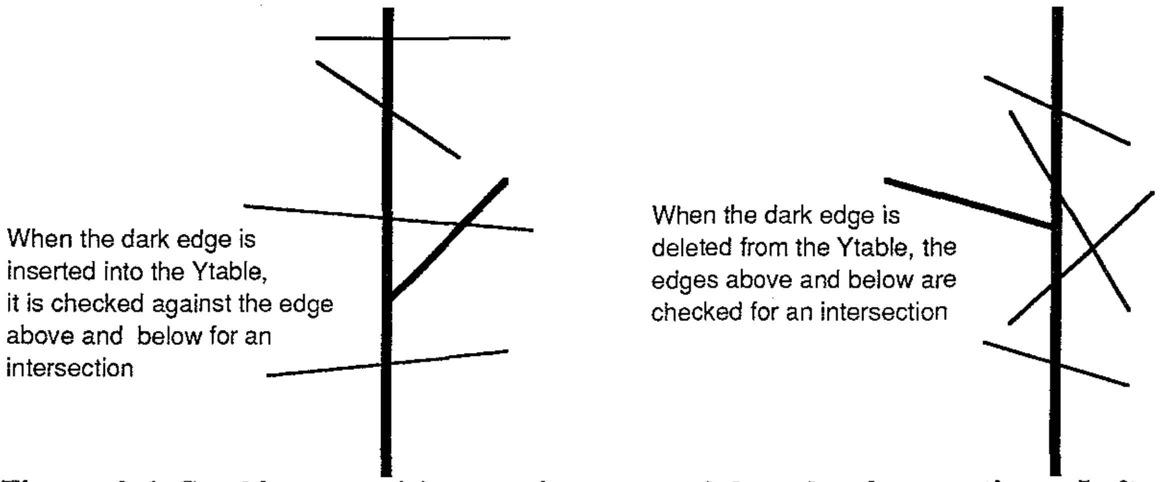

When the dark edge is inserted into the Ytable, it is checked against the edge above and below for an intersection
When the dark edge is deleted from the Ytable, the edges above and below are checked for an intersection

Figure 2.6 Catching transition vertices created by edge intersections. Left: when an edge is added to the Ytable, it is checked against the edges in the Ytable above and below it. Right: when an edge is deleted from the Ytable, the edges above and below it become adjacent and are checked for intersections. Intersections that are to the right of the sweep line are entered into the Xqueue for subsequent processing.

2.2.3.2 Determining which Regions Satisfy the Set Operation

In the triangulation algorithm above we kept track of whether a region was an *in* region or an *out* region. The regions alternated and were bounded on either side by neighbors in the Ytable. This implies that each edge in the Ytable could be treated as a transition from an *in* region to an *out* region. Now we will keep track of whether a region between two edges in the Ytable is *in* or *out* with respect to our desired set operation. It is no longer true that regions will alternate so our interpretation of edges in the Ytable as transitions from *in* regions to *out* regions or vice versa will have to be augmented by the concept of a *non-transition edge.* If two neighboring regions are *in* regions, or *out* regions, the edge between them is a *non-transition* edge.

To determine whether a region satisfies the set operation it necessary to know whether any polygons from a particular set cover the region. This is done by counting the number of polygons from each input set which cover each region. This array of counts can be evaluated to determine whether the region is *in* or *out,* i.e, whether it satisfies the set operation. When an edge from a set of polygons is inserted into the Ytable, the count for that set in the Ytable region bounded by the edge is incremented. Similarly, an edge deletion requires decrementing the appropriate counter.

2.2.3.3 Processing Complex Transitions Caused by Degenerate Input

If we assumed that the input was in general position, the extensions outlined in sections 4.1 and 4.2 would be sufficient. The algorithm will run in time O((n+s)(v+logn)) where *n* is the number of vertices in the input, *s* is the number of vertices created by edge intersections and *v* is the number of distinct sets in the set operation expression. Typically *v* will be a small integer so that the complexity can be simplified to O((n+s)(logn)).

However, if the program is to be useful it must be modified to handle degenerate input. When many vertices in the input coincide with other vertices or with other edges, the transition vertex may have any number of edges entering it from behind the sweep line and any number of edges exiting after the sweep line. The following diagram (figure 2.7) gives an example of such a situation.

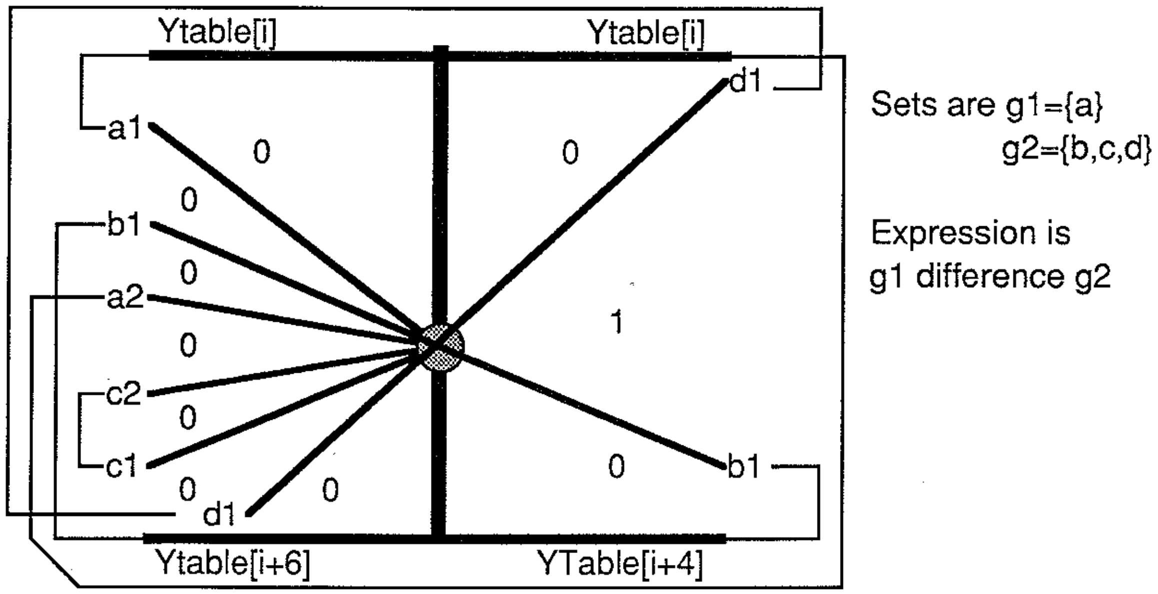


Figure 2.7. An stylized example of many edges converging on one transition vertex. The scan-line is drawn vertically as a thick line. The transition vertex is the circle. The entries in the Ytable that intersect the transition vertex before and after the transition are shown as polygon edges while the entries in the Ytable above and below the transition vertex are shown as horizontal thick lines. The regions between edges are labeled with either a 1 or 0 depending upon whether the evaluation of the set operation DIFFERENCE(g1,g2) is true or false. The labeling of the edges indicates whether the edge extends through the vertex or ends or starts at the vertex. For example, edges *a1* and *a2* are sequential edges in polygon *a* and they both end at the transition vertex. Edges *b1* and *d1* intersect at the transition vertex and have been drawn over the the transition vertex to emphasize that they do not end or start at the vertex. The numbering of the edges reflects the counterclockwise orientation of the polygons. Possible global connections of these edges are suggested by the thin lines connecting edges from a common polygon.

We now show the geometric interpretation of the transition vertex that allows it to be processed without special cases for degenerate situations. Our algorithm will consider each of the edges in counterclockwise order around the transition vertex. If an edge is a transition from a 0-region to a 1-region we will call it a R(ising) edge and if it is a transition from a 1-region to a 0-region we will call it a a F(alling) edge. Other edges are C(onstant) edges. In our example, *b1* is an R edge to the right of the scan-line and *d1* is an F edge to the right of the scan-line. All other edges are C edges. We will look for pairs of R and F edges as we travel counterclockwise around the transition vertex. Clearly, two R edges separated by zero or more C edges cannot occur, similarly for two F edges. This implies that R and F edges separated by zero or more C edges will occur in pairs. This is the necessary abstraction; we consider pairs of R and F edges instead of simply pairs of edges.

Each R-F pair may be treated just as the pair of edges bounding the *in* region were treated in the simple polygon triangulation algorithm. We need a mildly complicated loop structure to detect the R-F pairs but once they are found they may be handled with a simple case structure:

1. The R and F pair are to the left of the scan-line.
   Depending upon which edge was encountered first, this corresponds to one or the other of the end cases described in the triangulation algorithm.

2. The R edge is on one side of the scan-line, the F edge is on the other.
   This is equivalent to the bend case in the triangulation algorithm.

3. The R and F pair are to the right of the scan-line.
   Depending upon which edge was encountered first, this corresponds to one or the other of the start cases described in the triangulation algorithm.

We may now consider the details of the loop used to detect the R, F pairs of edges in the order described. The Ytable will provide the counterclockwise ordering if we find Ytable[i] using `ytbl_findabove()` and use `ytbl_succ()` to traverse the Ytable from *a1* down to *c1*, and then traverse the Ytable from *b1* back to *d1* using `ytbl_pred()`. Any edges that ended at the transition vertex are deleted during the downward traversal. Any edges that begin at the transition vertex are inserted after the downward traversal. Edges that extend through the vertex do not need to be deleted and re-inserted, they can have their order in the Ytable reversed before beginning the upward traversal.

It is necessary to associate with each vertex entered into the Xqueue the edges that emanate from that vertex so that they can be inserted into the Ytable. It is no longer necessary or even possible to associate a *bend*, *start* or *end* type as was done in the simple polygon triangulation algorithm. After the vertex has been processed the edges closest to the edges above and below the transition vertex are checked for intersections and if an intersection is found it is entered into the Xqueue.

Another detail facing the programmer during the transition operation is the problem of correctly maintaining the set count arrays associated with the new regions. Although the edges entering the transition vertex were deleted in top to bottom order, the order in which exiting edges are inserted into the Ytable is arbitrary. It depends upon the order in which the exiting edges associated with the transition vertex were inserted into the Ytable. Until the last edge is inserted, the Ytable is in a state that may not correspond to any possible physical configuration and hence it is impossible to update any of the set count arrays until the last edge is inserted. After the last edge is inserted, and the upwards traversal begins, the set count arrays can be updated as each is encountered. We copy the set count array from the previous edge and then increment or decrement the entries that correspond to the sets that the current edge bounds.

2.2.3.4 Handling Edges that Inhabit the Same Line

Edges may exist on the same line and intersect along line segments rather than at single points. This degenerate condition gives rise to dangling edges and polygons with no area.

The generalized transition algorithm is not changed appreciably by the steps taken to handle this problem. We let the entries in the Ytable be characterized by a line equation and append a list of the edges that share that line equation. This makes checking for intersections between new neighbors in the Ytable slightly more complicated, because we have to ascertain that an intersection of two parent lines is contained by at least one edge lying on each line. Deletions and insertions are also complicated somewhat, because we cannot delete an entry from the Ytable until all the edges on the line are deleted. Similarly, we have to check to see whether an edge we are inserting already has a parent line in the YTable.

2.2.4. Implementation

The generalized algorithm has been implemented in a just under two thousand lines of C. A streamlined implementation suitable for one particular application could no doubt be implemented in much less. Our implementation is augmented to optionally animate its progress and compute certain other information associated with the input. We implemented the Xqueue on top of an implementation of top-down 2-3-4 red-black trees, a balanced tree algorithm [Sedgewick88]. Originally a heap was used but because a heap is only a partially ordered data structure, it can not combine identical vertices until they are removed together. Since coincident vertices are the norm rather than the exception in our applications, the balanced tree implementation has performed as well as the heap.

Because our applications have typically involved thousands of vertices and not millions we have gotten away with implementing the Ytable with an array. In practice one can often achieve good performance simply by making sure that the Xqueue is implemented efficiently due to the fact that the Ytable is usually far below worst case capacity at any given time. Intuitively, if the region being swept is roughly square and edges are distributed evenly over the region, any vertical line will intersect roughly the square root of the total number of edges. Thus, the Ytable will be filled to the square root of maximum capacity on average. These intuitive ideas are made precise and confirmed by Bentley, Haken and Hon [Bentley80].

The implementation was coded so that a pointer to the function that evaluates the set operation could be passed as an argument to the transition routine. This was a good decision, because it allows experimentation with new applications without the duplication of code for the transition routine, which is fairly complex.

Our biggest single implementation problem has been tuning the algorithm to allow for floating point errors. Since all decisions in the algorithm are local, it really only cares about the relative ordering of values and not about their absolute value. Unfortunately, this is exactly what floating point is not designed to do. The optimal data type to use for vertices and other geometric values would be rational numbers represented with a big integer for numerator and denominator since that would allow exact calculations but this is not possible in our version of C, since we are limited to 32 bit integers. Double precision floating point, 64 bit IEEE format, was used for the coefficients of edges in the Ytable but even then epsilon values were necessary to prevent edges being inserted into the Ytable incorrectly.

High level code for the algorithm appears in Appendix A.2. There is a small modification to the algorithm that has not yet been discussed. Some applications, such as radiosity re-tilings, need to keep all edges between polygons that are considered to be in the same set (figure 2.8). The modification treats non transition edges that normally would be ignored, the interior edges, as a Falling and Rising pair of edges rather than as a Constant edge. Since the function passed in as an argument determines whether an edge is a Rising, Falling or Constant edge, this change is fully backward compatible with the implementation as described above. See Appendix A.2 and A.3 for C code for `sweep()` and `transition()`.

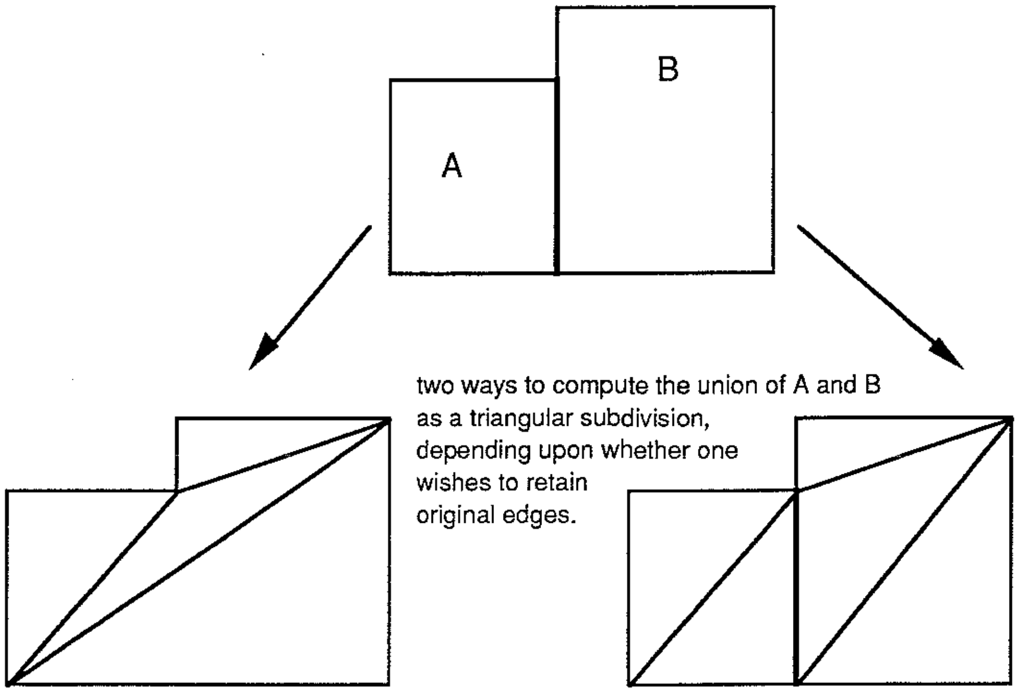

A
B
two ways to compute the union of A and B as a triangular subdivision, depending upon whether one wishes to retain original edges.

Figure 2.8 Modification: allow all edges present in the original input to be retained in the triangular subdivision output.

2.2.5. Applications

We list here three applications that we have implemented using the decomposition algorithm. We also describe how the algorithm might be employed to solve the hidden surface problem. The key to applying the algorithm is to recognize that a problem can be viewed as a set operation (boolean expression). This is not always obvious.

2.2.5.1. Arbitrary Region Clipping

Clearly, the polygon clipping operation is a set operation. It is the intersection (boolean AND) of the clipping window and each of the rest of the polygons which may be grouped into one set. While we do not recommend implementing clipping an environment to a single rectangular window with this algorithm, there are other clipping applications that crop up in computer graphics research that may need the power of this algorithm.

Computing polygonal definitions of portals is an arbitrary region clipping example. Subtracting the polygons inhabiting the plane of the cell boundary from the rectangular boundary of the cell gives us the portals. The subtraction operation is the AND-NOT operation. Conversely, a programmer could place doors and windows in the boundary of a rectangular volume and subtract them away from the boundary to model a room. This type of coplanar CSG operation can be troublesome for CSG algorithms that operate on boundary representation data.

Another example of an unusual clipping application is the occlusion operation that appears in chapter 4. We are given an odd shaped region known to be the window in a plane through which two polygons on opposite sides of the plane must look to "see" each other. Determining that the window was completely covered allows us to deduce that those two polygons could not see each other. We used the difference operation to subtract polygons known to inhabit the plane of the given window from the window. If anything remains after the operation, the polygons can see each other.

2.2.5.2 Detecting Coincident Polygons

Two physical objects should not be modelled so that inhabit the same area in space, however, many CAD programs do not support operations to catch this problem. A wall can be mistakenly overlapped with a window frame. If the two objects have different colors, the result of rendering these objects is undefined.

The algorithm can be used to detect coplanar polygons in a model. Each polygon is a member of a distinct set. The set operation is the union of any two or more polygons. Because our implementation maintains an array with one entry for each set in each element of the Ytable this is not very efficient. However in this case it works efficiently enough. An implementation which used linked lists instead of the array or hard wired the set operation in some way could be used if efficiency became of utmost concern. It is also possible to use something more than a true set operation to determine the region of the plane to triangulate. For example a count of the number of polygons covering a region might be used.

2.2.5.3 Transforming a Planar Tiling into a Triangular Planar Subdivision

We define a *tiling* of a planar region as a decomposition of the region into polygons such that the region is covered at every point by one and only one polygon. A tiling in which edges intersect other edges only at vertices is a *planar subdivision.* The advantage of a planar subdivision over a tiling becomes clear when the tiling is laid over a curved surface. The tiling of a curved surface may have cracks in it while the planar subdivision avoids this problem. Von Herzen notes this problem in approximating curved surfaces with restricted quadtrees [Von Herzen87].

Even if the surface tiled is actually flat, such as the wall of a building, radiosity shading calculations produce a variation in intensity over the surface that can be perceived as a curved surface. Here the cracks will appear as shading discontinuities. Transforming the tiling to a planar subdivision will remove the shading discontinuities (figure 2.9).

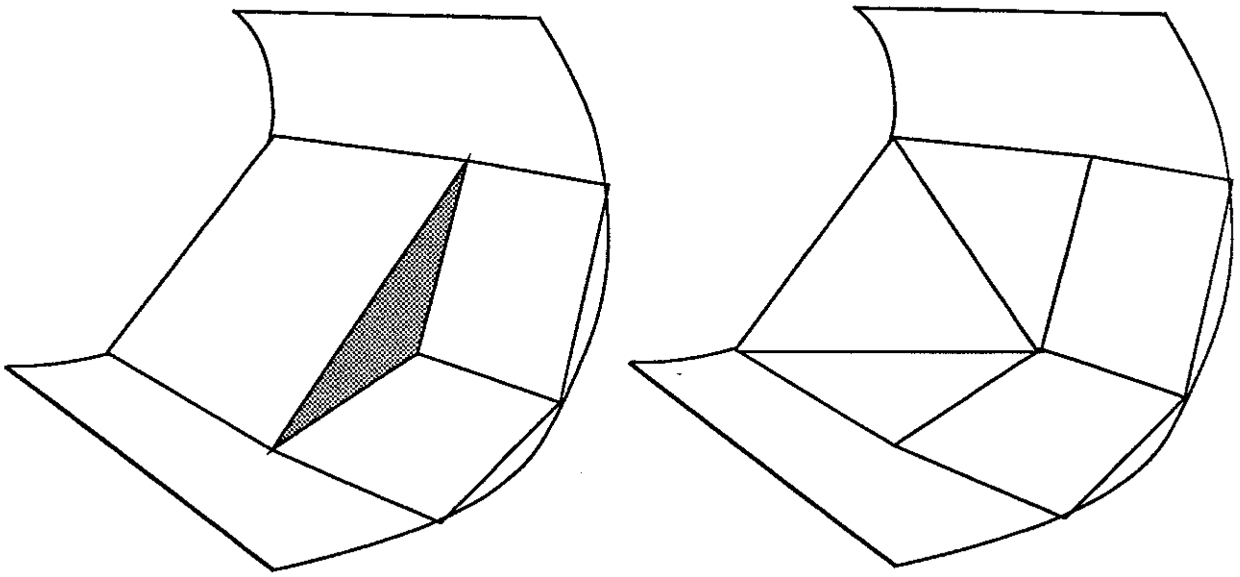


Figure 2.9 Left, a tiling allows cracks to appear between polygons when laid over a curved surface. The planar subdivision on the right has no cracks. Shading intensities can be interpreted as curved surfaces. The cracks in the tilings will appear as shading discontinuities.

Chapter III

Potentially Visible Set (PVS) Calculation

This chapter addresses the problem of determining the set of polygons that are visible to an observer allowed free range in a *cell,* an axially aligned box of viewpoints. The set of polygons is called the cell's *potentially visible set* (PVS).

If the contents of the cell are assumed visible, then the problem reduces to identifying the polygons visible to viewpoints constrained to lie on the non-opaque boundary, or *portals,* of the cell. This is evident because any line of sight from a viewpoint in the interior of the cell must intersect the portal. Thus, that line of sight can be replaced by one which starts at the intersection point. We can further constrain our attention to the non-opaque portions of the cell boundary. Section 2.2 presented an algorithm to compute a triangular subdivision of the portals. Thus, the problem is to compute the set of polygons that can be seen by all the viewpoints lying on a convex polygon (figure 3.1).

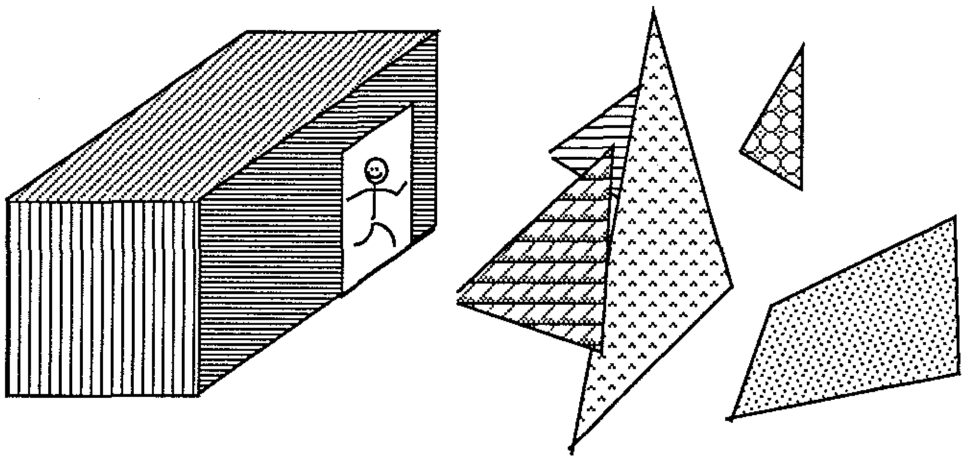


Figure 3.1 The PVS Problem: Identify any polygons that the observer can see through the cell portal.

An equivalent problem is determining the set of polygons illuminated by an area light source. Note that this is not the same as estimating the illumination on each point of a surface due to an area light source, which is the usual form of the problem for shading purposes.

There are also several related two-dimensional problems which have appeared in the theoretical computational geometry literature.

Avis, Gum and Toussaint describe a linear-time algorithm for determining visibility between two edges in a polygon [Avis86]. They define four nested types of edge visibility:

complete, strong, weak and partial (figure 3.2). Similar definitions can be made for polygons embedded in $\mathcal{R}^3$, although, as the authors note in their last sentence, "...all the problems in three dimensions corresponding to those discussed in this paper are open."

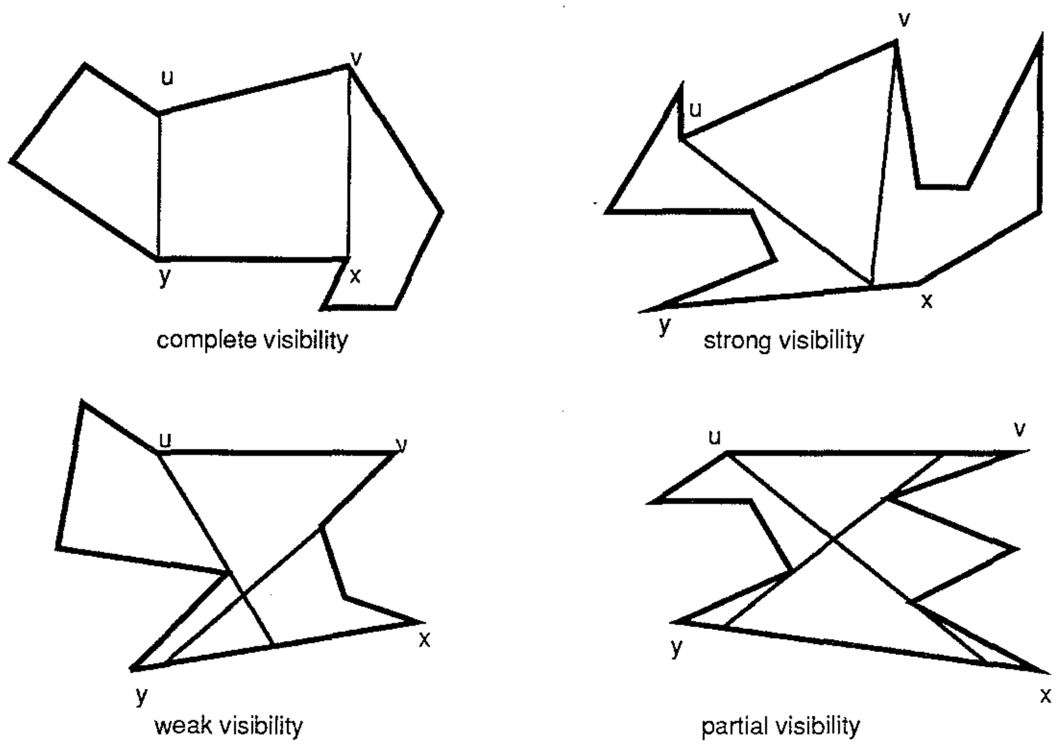


Figure 3.2 Four types of visibility. [Avis86]
1) Edge *uv* is said to be *completely visible* from edge *xy* if for all points *z* on edge *xy* and all points *w* on edge *uv, w* and *z* are visible.
2) Edge *uv* is said to be *strongly visible* from edge *xy* if there exists a point *z* on edge *xy* such that for all points *w* on edge *uv, w* and *z* are visible.
3) Edge *uv* is said to be *weakly visible* from edge *xy* if for each *w* on edge *uv*, there exists a point *z* on edge *xy* such that *w* and *z* are visible.
4) Edge *uv* is said to be partially visible from edge *xy* if there exists a point *w* on edge *uv* and a point *z* on edge *xy* such that *w* and *z* are visible.

O'Rourke and Suri's algorithm, O(n^4) in the number of line segments (which is worst case optimal!), for computing the boundary of the visible region from an edge in a general two-dimensional environment of line segments shows that generalizing problems even within two dimensions can lead to significant difficulties [O'Rourke87].

Another concept that appears in O'Rourke's monograph is the concept of a *visibility graph.* Given a collection of geometric objects embedded in space, we can define a graph with the objects as nodes. An arc between two nodes represents visibility of some type between the two objects represented by the nodes. This allows graph theory and terminology to be applied to these visibility problems. For example, one can represent the solution to the problem of computing polygon-to-cell visibility as a bipartite graph. One type of node represents cells, the other, polygons. The density of the ideal solution graph indicates the potential speedup. The methods presented in this dissertation are appropriate for sparse graphs, corresponding to scenes with high average depth complexity.

3.1 Over-Estimating Visibility vs. Under-Estimating Visibility

Computing an exact answer to the portal-polygon visibility problem appears to be very difficult. For this reason, I have implemented approximation algorithms. If we use graph terminology and think of the portal-polygon problem as the problem of computing a bipartite visibility graph, there are a couple of obvious ways to proceed. The first is to start with a graph with no edges and add edges whenever we can establish visibility between a portal and a polygon. The second is to start with a complete graph, and delete edges whenever we can establish occlusion between a portal and a polygon.

The edge addition approach will be familiar to those in the computer graphics community. Essentially, one samples lines of sight between a portal and a polygon. If any of the lines of sight are unobstructed, then the polygon is visible.

Most solutions to the area light source problem involve point sampling. In one approach the environment is sampled from the point of view of the light source, usually with a fixed sampling pattern. An alternative is to sample the light source from selected points within the environment [Wallace89]. I call these sampling methods *environment-independent* sampling and *environment-dependent* sampling, respectively. A better pair of terms might be *fixed* and *targeted* sampling, respectively, because in one case a fixed pattern of samples is fired into the environment and in the other case the samples are aimed at interesting things in the environment. As with any pair of algorithms for one problem, the possibility of hybrids arises.

Although point sampling may be accomplished by any number of techniques, e.g., object-order z-buffer or scan-line algorithms [Rogers85], we have based our experiments upon a ray-casting sampling method. Although slower than many other sampling methods, it is simple and flexible.

The fixed sampling approach fires a predetermined pattern of sample rays into the environment from regularly spaced source points on the portals. The number of rays fired and the spacing of the source points on the portals are fixed at run time. There are many different ways to choose the sample patterns. Section 3.2.1 covers these choices.

The second approach is to target the rays at the polygons. We then have to choose how many rays should be fired at each polygon and how to distribute those rays. Section 3.2.2 covers the details involved in implementing this approach.

One advantage of sampling methods is their simplicity. This leads to efficient, flexible, and easy-to-maintain programs. Furthermore, one may adjust the grain of the sampling to match the available computation power and time constraints. Note that if a sampling algorithm, such as the Z-buffer algorithm used by most commercial workstations, is used for display, the grain of visibility sampling can be matched to the grain of display sampling because the same hardware is being used for both purposes.

A disadvantage of point sampling methods is that they underestimate the PVS. Unless the sampling frequency is high, some polygons will be missed. There are two aspects of sampling. Firing rays into the environment means one will miss some directions.

Furthermore, the rays are fired from sample source points on the portal. This means one may miss some weakly visible polygons, even if we could sample all directions from the source points. Any edge-addition methods, whether point-sampling or not, approach the true value from below, so they in general are subject to under-estimating.

The penalty for missing polygons is severe. The illusion of reality can be damaged if parts of the model are missing. Thus, I also examine algorithms that attempt to compute the exact PVS, but if they err, do so by overestimating the PVS. Using the area light source analogy again, we want to identify the polygons that lie in complete shadow, i.e., the umbra cast by occluding objects. Section 3.3 details algorithms to compute this type of solution as well as ideas for computing the exact solution.

3.2 Sampling Algorithms

This section covers sampling algorithms. First, the ray-polygon intersection algorithm that was used is discussed. The subject of ray-polygon intersection algorithms has been extensively researched [Hanrahan89]. Most approaches use some type of hierarchical subdivision of the environment in an attempt to get O(logN) performance for computing the intersection of a single ray. I could have written my ray-polygon intersection code to use the model subdivision developed in chapter 3, as that provides a suitable hierarchical subdivision of the model.

Instead, I used another data structure I already had, the parallel plane sorted index list, which I had originally developed to allow me to choose partitions efficiently. This approach appears to a new way to compute ray-polygon intersections efficiently. See figure 2.2.

The algorithm takes advantage of characteristics such as the large proportion of axial rectangles. The basic idea can be easily described in two dimensions. Consider the problem of computing the closest intersection of the ray and line segments depicted in figure 3.3.

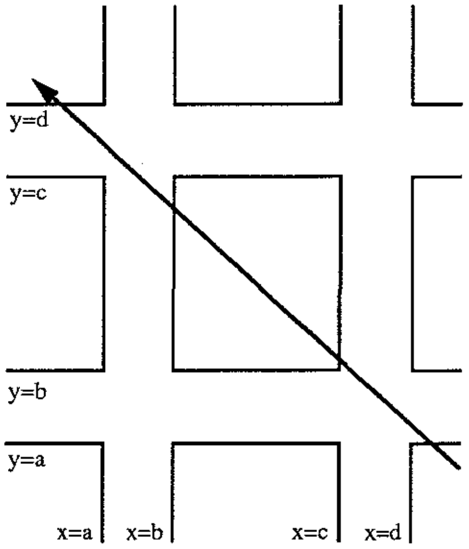


Figure 3.3. Ray-Line Segment Intersection.

The ray shown intersects the lines containing segments parallel to the x-axis, in order from bottom to top. Similarly, the ray intersects the lines containing segments parallel to the y-axis, in order, from right to left.

This suggests a data structure which groups line segments lying in the same line together. Each set of parallel lines is sorted along their normal direction. This data structure is pre-computed.

To compute the intersections in order, we check the intersection parameter for the closest line in each of the two sorted lists. In our example, the line y=a is closer to the initial point of the ray than the line x=d. When we check the segments lying in the line y=a, we halt and report the intersection.

If we had not found an intersection, we would have computed the intersection parameter for the next line in the x-parallel list, y=b, and compared it with the intersection parameter for the line x=d. We continue to effectively merge the two lists until we find an intersection. A priority queue data structure can be used to maintain the order of the lists that are being merged [Sedgewick88]. The priority of each list is simply the closest element in each list that has not yet been processed. If the number of lists is only two the implementation is trivial. The implementation with three is simple, but may not be immediately obvious to someone reading the C code. The C code that appears later in this section uses some tables to define the start and end points of the list. If the ray start point is in the middle of the model, we don't want to waste time running through planes that are behind the ray. A binary search is used to find the planes that are closest to the start point of the ray, then the sign of each component of the ray tells us whether we should move backwards or forwards through each list. All this information is encoded into several tables.

The small percentage of non-axial polygons in our models are stored in a standard BSP tree. After an intersection is found for the axial polygons, the BSP tree is searched from front to back until we find an intersection or exceed the intersection parameter found for the axial polygons.

The technique is very effective for bare buildings without a great deal of detail. For example, it runs very efficiently on the basement of the church, which is a fairly standard layout of unfurnished small rooms with a minimum of detail. It performs poorly on the upper portion of the church which consists of one large and irregular room, the fellowship hall, and one very detailed and polygon-rich kitchen.

To run efficiently on the upper portion of the church, the algorithm would have to be extended. Currently, it computes an intersection for an entire plane's worth of polygons, but then searches naively through the list of polygons for that plane to see if any of those polygons contain the intersection point. Some type of 2-D organization, such as a grid or quadtree, could be induced on that simple list of polygons to accelerate this operation.

3.2.1 Environment-Independent, or Fixed, Sampling

The advantages of fixed sampling methods are as follows:

• The number of samples can be easily controlled. This can be used to predict the running time.

• The sampling density is known in advance. This can be used to help predict the accuracy of the result.

• The sampling pattern is known in advance. This allows one to design sampling algorithms to take advantage of the coherence of the pattern, such as scan-line methods based on regularly spaced scan-line samples, or to design hardware based on the sampling pattern, such as the Pixel-Planes 5 computing surface or SGI Iris 4D Geometry Pipeline [Baum90], [Akeley89].

The main disadvantage is:

• If we cannot characterize the distribution of polygons in the environment at the time the pattern is determined, a lot of samples may be fired at the same polygon, a lot more may be fired at nothing, and some polygons may be missed.

3.2.1.1 Hemi-cube sample pattern

The first sample pattern tried was based on the hemi-cube pattern used in the early radiosity papers[Cohen85]. The hemi-cube was used as an approximation to a hemisphere because it allowed standard hidden surface techniques, such as the Z-buffer method, to be used to compute the samples (figure 3.4). Esssentially, the polygons are projected onto the hemi-cube faces as if they were viewing screens.

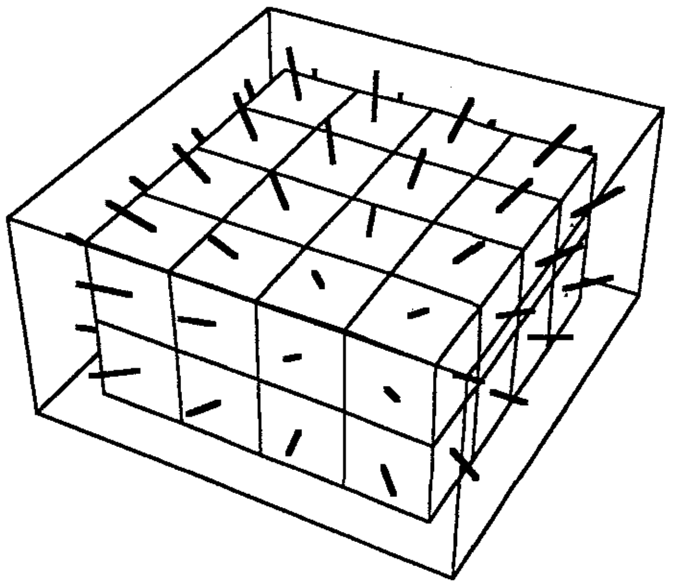


Figure 3.4 The hemi-cube sampling pattern. Sample rays are shown extending through the sample areas. The sample rays start at the center of the base of the hemi-cube. This allows the (hardware) algorithm to take advantage of the regular pattern of samples and use iterative methods.

These papers note the importance of rotating the hemi-cube to reduce sampling artifacts. Architectural models are usually axially aligned. An aligned sampling grid will hit many polygons many times and completely miss many others (figure 3.5).

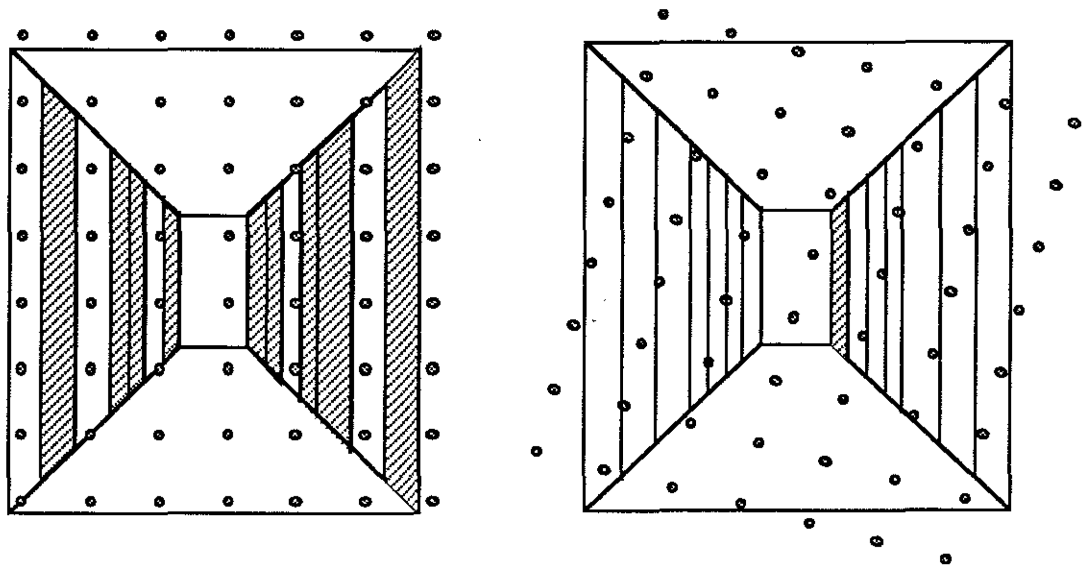


Figure 3.5 Two stylized views down a hallway. The sample points are represented with dots. Note that the axially aligned pattern on the left hits many polygons several times and others not at all. The rotated sampling grid on the right hits many more polygons with the same number of total samples. The missed polygons are shaded.

An alternative to random rotation of the sample grid is to jitter each sample point (figure 3.6)

The probability of hitting a tall thin rectangle with an axially aligned regular hemi-cube sampling pattern is just the ratio of the width of the rectangle to the width between samples. The chance of hitting the polygon is independent of its total area; it can be infinitely tall and still get missed. If the rectangle is rotated slightly, or equivalently, the sampling grid is rotated, the expression for the probability of hitting the polygon becomes more complex and appears to be unrepresentable in closed form except for certain special cases. However, it is evident that the chance of hitting the polygon increases. In the case of axially aligned jittered sampling, it is clear that the height of the polygon plays a role in the hit probability. The probability of missing a tall rectangle is the product of the probability that each pixel will miss the rectangle. Thus the chance that the polygon will escape detection with a jittered grid is very slight if the polygon is tall.

I did not use random rotation of the hemi-cube or sample jittering in this early implementation. As might be expected, the performance of this method was poor. The implementation was intimately connected with the research into the use of radiosity lighting models in the Walkthrough project. I wrote software so that it could be used for either purpose. At that time, I rewrote the radiosity software to use a hemisphere based sample distribution pattern and at the same time replaced the hemi-cube visibility point-sampling code with a hemisphere point-sampling method.

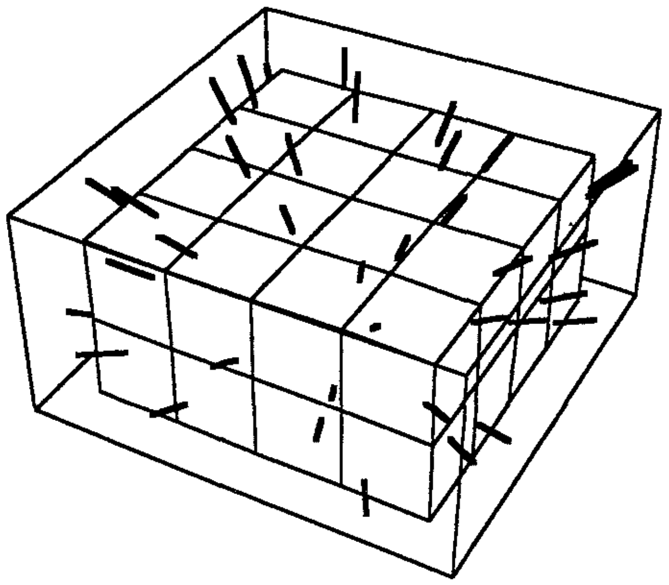


Figure 3.6 The jittered hemi-cube sampling pattern. Sample rays are shown extending through the sample areas. The sample rays start at the center of the base of the hemi-cube. This pattern transforms signal aliasing due to the coherence of the samples into (less objectionable) noise.

3.2.1.2 Cosine weighted hemisphere sample pattern

To partition the surface of the hemisphere into sample regions, I used a mapping from planar polar coordinates to the surface of the hemisphere that was convenient for radiosity calculations. For radiosity calculations it was important that each sample area on the hemisphere have equal area when projected orthogonally down onto the base disk (figure 3.7). This hemisphere partition mimics the distribution of energy emitted or reflected from a purely diffuse material.

The mapping is generated by taking equal radial subdivisions and angular subdivisions and then taking the square root of the radial division to adjust for the fact that area on the disk is a function of the square of the radius. Given the radius and theta coordinates of a point on the disk, the cartesian coordinates of a point on the hemisphere in the sample ray direction are

x = radius*cos(theta),
y = radius*sin(theta),
z = sqrt(1-radius*radius).

The hemisphere sampling pattern does not suffer the same magnitude of sampling problems that the axially aligned hemi-cube pattern does. It is prone to similar problems in polar coordinates, but most building models are aligned along cartesian coordinates. I further jittered the hemisphere samples to avoid polar coordinate coherence effects. To generate the correct radius for point samples that are in the center of the sample areas I simply generated twice the number of radius divisions and took every other value. Note that one cannot simply take the average of the radial limits of the sample area. Doing so will not give the radial center of the sample area. The average must be done before the square root is computed. No such problem exists when generating the angular center of a sample patch. Because the Unix system random number generator, `random()`, only returns uniformly distributed samples, I only jittered angularly (figure 3.7 and figure 3.8).

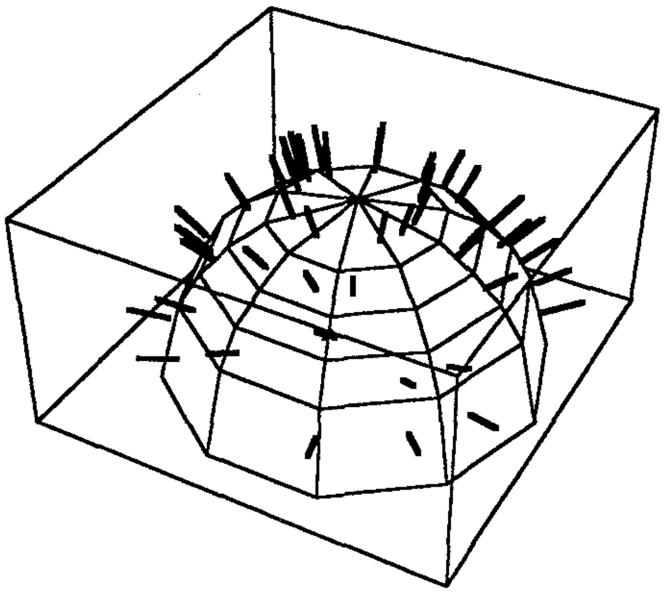


Figure 3.7 The angularly-jittered cosine-weighted-hemisphere sampling pattern. Sample rays are shown extending through the sample areas. The sample rays start at the center of the base of the hemisphere. This pattern has the property that each sample area has equal area when projected orthogonally down onto the base disk.

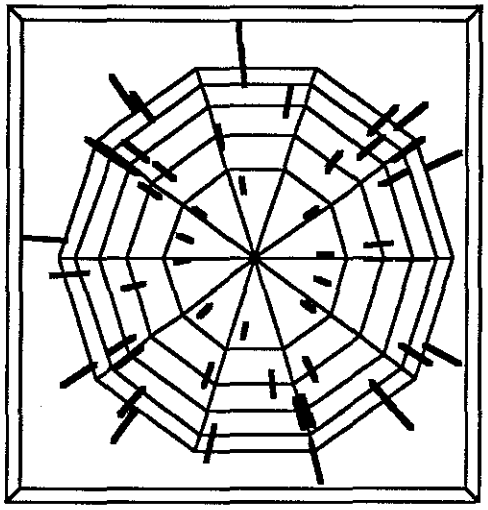
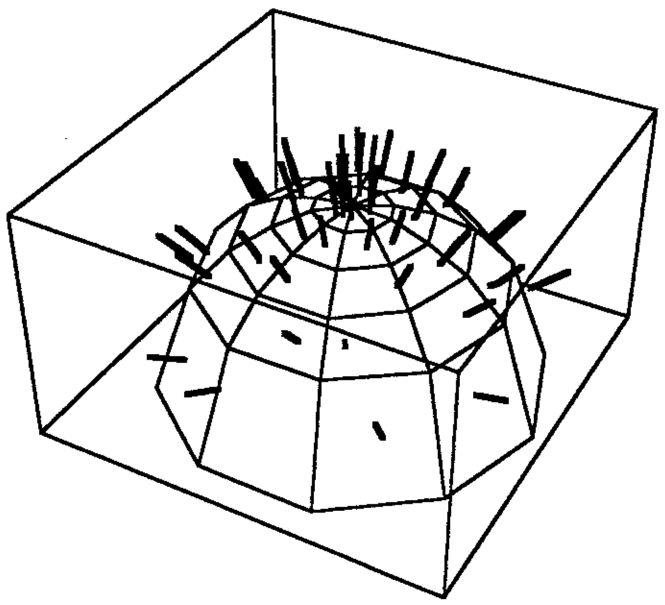


Figure 3.8 The angularly-jittered cosine-weighted hemisphere sampling pattern (top view). The sqrt (radius) progression is clear in this view.

This sample pattern worked better than the hemi-cube method, but I still wanted to improve performance with a small number of samples. The main problem is due to the spaces between samples at the top of the hemisphere. The area of this space gets larger as the radius of the hemisphere increases. In the case of the view down the hallway, the perspective transformation results in nearby polygons appearing as large polygons at the edge of the image and far away polygons appearing as small polygons in the center of the image. Thus there was a tendency for the cosine-weighted hemisphere sampling pattern to miss polygons down at the very end of long hallways. To combat this annoying problem, I looked for a simple modification that would bunch the rays closer to the top of the hemisphere.

3.2.1.3 Linear radius hemisphere sample pattern

The obvious trick is to take out the area-correcting square root of the radius subdivisions. This bunches the samples closer to the top of the hemisphere (figure 3.9).



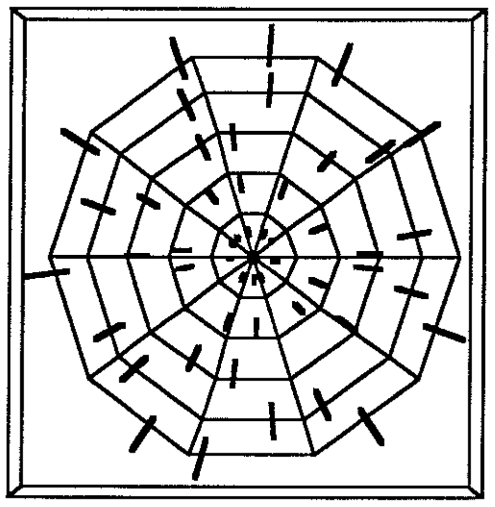


Figure 3.9 Linear radius-subdivision angularly-jittered sample pattern, off-axis and top views.

3.2.1.4 Other sample patterns

The linear radius hemisphere modification helps the "end of the hallway" problem, but it will not magically allow one to find all visible polygons with only a few samples. The patterns I have proposed here will make the difference between finding most of the visible polygons with the hemi-cube and almost all the polygons with the jittered patterns.

However there are those among us for whom "almost all the polygons", is not good enough. In the end, if one wants to get all polygons above pixel size, a whole lot of samples must be used. So while playing with sample patterns is interesting, time may be better spent finding (or building!) a faster sampling engine.

Several people have suggested interesting ideas for improving the effectiveness of sampling. Jan Prins suggested an adaptive method which uses knowledge of portals. The idea is that portals in neighboring cells can be interpreted as regions of increased image complexity. It is likely that many polygons will be visible through a portal. Thus, one could fire rays at portals of neighboring cells after firing a fixed set of rays into the environment.

3.2.2 Environment-Dependent, or Targeted, Sampling

The advantages of targeted sampling:

• The sampling is concentrated in the area where it is believed that it will do the most good.

The disadvantages:

• The number of samples depends upon the complexity of the environment. It is possible for many samples to go in the same direction if there are a number of polygons, one behind the other, in the same direction.

• One cannot pipeline the algorithm by capitalizing on the coherence of sampling pattern. On the other hand, a lack of coherence can enable the use of massive non-pipelined parallelism, as in parallel ray-tracing algorithms [Delaney88].

The main variables in applying targeted sampling are the choice of how many samples to target at each polygon and how to distribute the samples. I simply used a run time constant to determine how many rays to fire at each polygon and a run time specification of how to space the sample viewpoints on the portal. This was very effective on the real-world models I was dealing with, which had on the order of 7,000 polygons. I typically fired a small number (1 to 7) of rays at each polygon. It might have been worthwhile to weight the number of samples based on the apparent size of the polygon, but the success of the initial attempt neutralized motivation to try many alternatives.

To generate sample target points on the polygons I generated a random number for each vertex and normalized them so that their sum was one. Then I used these as weights to get a point on the polygon. This will tend to weight the samples towards the center, but the effect is not excessive (figure 3.10). The alternatives require more coding effort. Greg Turk has published a note which describes how to generate uniformly distributed values on a polygon [Glassner90]. Either this method or a method using a jittered grid would probably be preferable to the one used. See Appendix A.4 for C code to implement sampling methods.

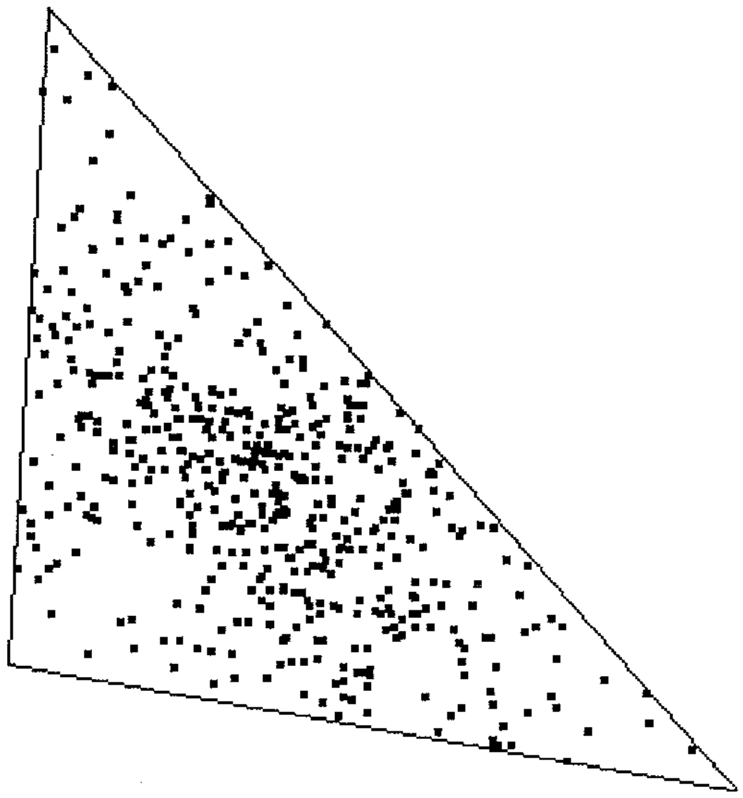


Figure 3.10 Random points generated on a polygon by generating random values for each vertex, normalizing them so they sum to one, and using them as vertex weights to get an interior point.

3.3 Over-Estimation Methods

It is likely that any practical application will use some type of sampling algorithm. A sampling algorithm can be matched to the available compute power, and many graphics workstations, such as the Silicon Graphics Iris 4D Power Series, have hardware that can be used to generate environment samples very efficiently. The rendering pipeline can be used to scan convert polygon identifiers instead of color values into the frame buffer [Baum90]. However, the development of an analytical, or closed form solution, which computes the exact solution or over-estimates the solution, is also important.

This section discusses the ideas involved in trying to compute an analytic solution. Most analytical algorithms are similar to analytic algorithms for computing shadowing effects from an area light source. Existing analytic shadow algorithms and their limitations to individual convex shadow casters are discussed. The limitations are illustrated by dropping down a dimension. An algorithm to compute the solution in two dimensions without any limitations is outlined, followed by a rough sketch of its extension to three dimensions. Several other ideas for computing the exact solution in three dimensions are briefly mentioned. This section finishes with a description of the algorithm I implemented.

Crow introduced the idea of shadow volumes for a point or directional light source, a source with zero area [Crow77]. Simply stated, the shadow volume for a polygon from a point source is the cone defined by the point light source and the polygon. Call these *shadow cones,* (figure 3.11)

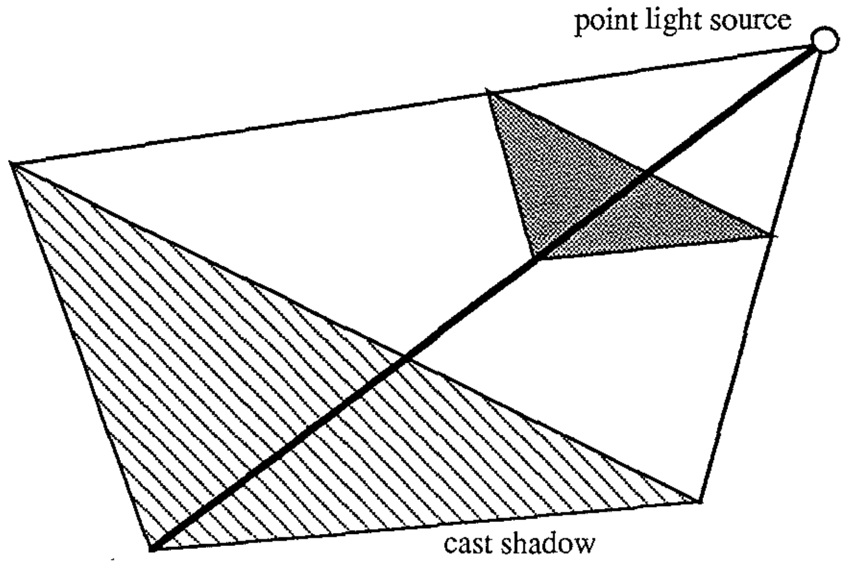


Figure 3.11 A Shadow Cone. All surfaces within the shadow cone receive no direct illumination from the point light source.

Nishita and Nakamae published an analytical solution for computations with area light sources [Nishita83]. This description covers the shadowing effect of one convex polygon or polyhedron. The complex interactions of many polygons are not addressed.

Nishita and Nakamae extended the idea of a shadow cone to shadow volumes formed by a *convex* polygon or polyhedron and a convex light source. Essentially, the shadow umbra is the *intersection* of the shadow cones computed from each vertex of the area light source. The shadow penumbra is the convex hull of the shadow cones, (figure 3.12). Note that the extra faces defined by the convex hull are actually constructed from the vertices of the shadow caster and an edge of the light source. The faces of the shadow cones, on the other hand are constructed from vertices of the light source and edges of the shadow caster.

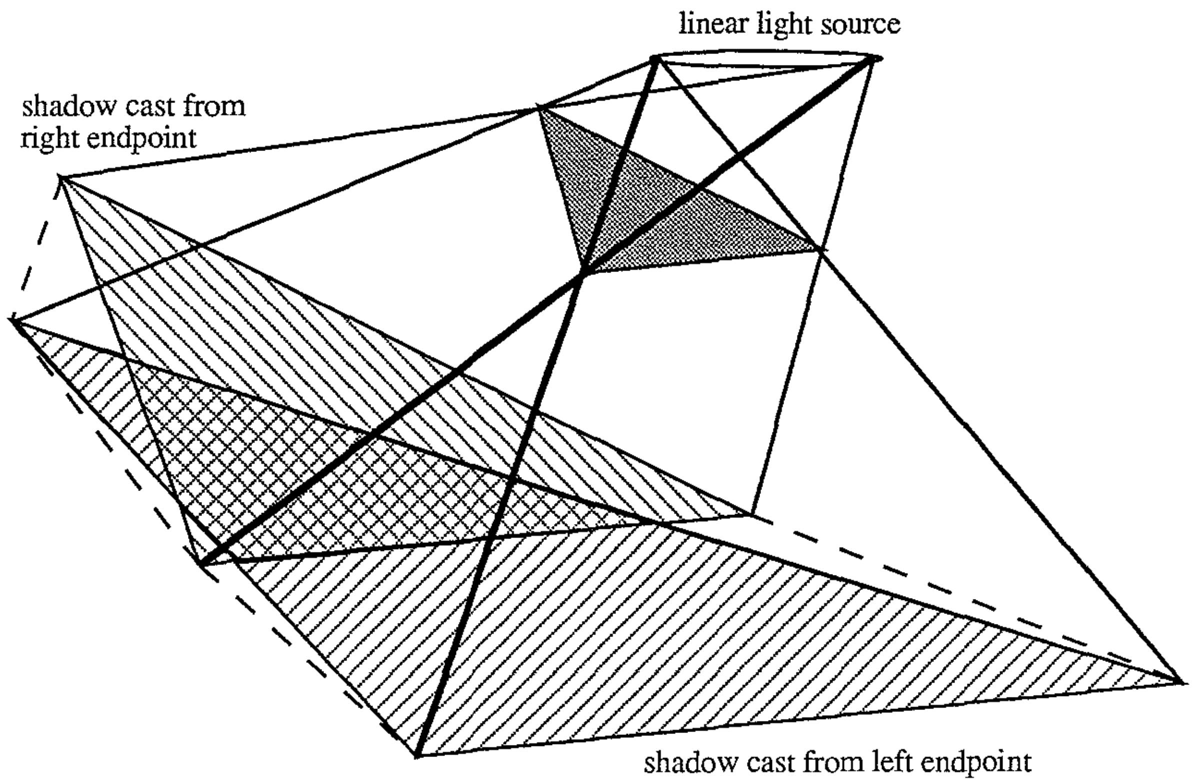

linear light source
shadow cast from right endpoint
shadow cast from left endpoint

Figure 3.12 A Shadow Volume. The surfaces within the umbra receive no direct illumination from the area source. Surfaces within the penumbra but outside the umbra receive some fraction of the direct illumination of the linear source.

3.3.1 Concave Shadow Casters

The restriction to convex shadow casters in Nishita's and Nakamae's algorithm is important. It is easy to construct an example using concave shadow casters and a linear light source where the intersection of the shadow cones formed with the line source endpoints is larger than the true umbra, (figure 3.13). Note that in this figure we have only drawn the shadows, so the reader might imagine a light and the shadow caster above the page.

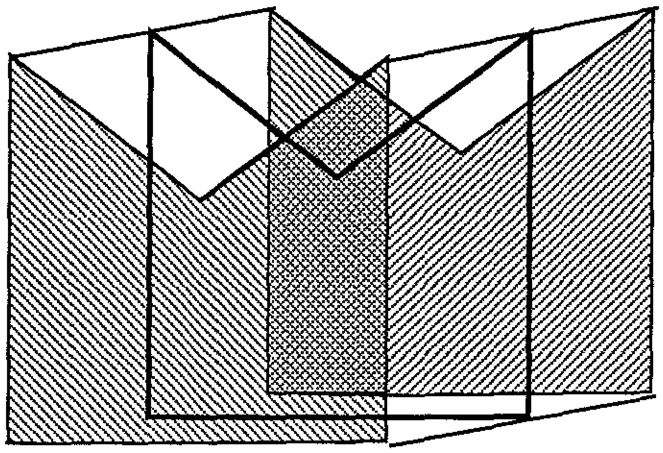


Figure 3.13. Shadows cast by a concave polygon from a linear light source. (Only the shadows are shown). By definition, the umbra is inside the shadow cast from all interior points of the linear light source. Here the shadow cast by some interior point is drawn in bold. Some of the computed umbra falls outside it.

To correctly compute the umbra for a concave polygon, we need to consider the difference between the convex hull of the polygon and polygon itself, i.e. the parts that would "fill in" the polygon and make it convex. These extra portions may be thought of as windows, and the cones formed with them and the light source vertices are visibility cones. Taking the convex hull of the visibility cones yields everything that can be seen through the windows. Thus, for concave polygons, we must subtract the hull of visibility volumes from the intersection of the shadow cones to get the true umbra volume, (figure 3.14).

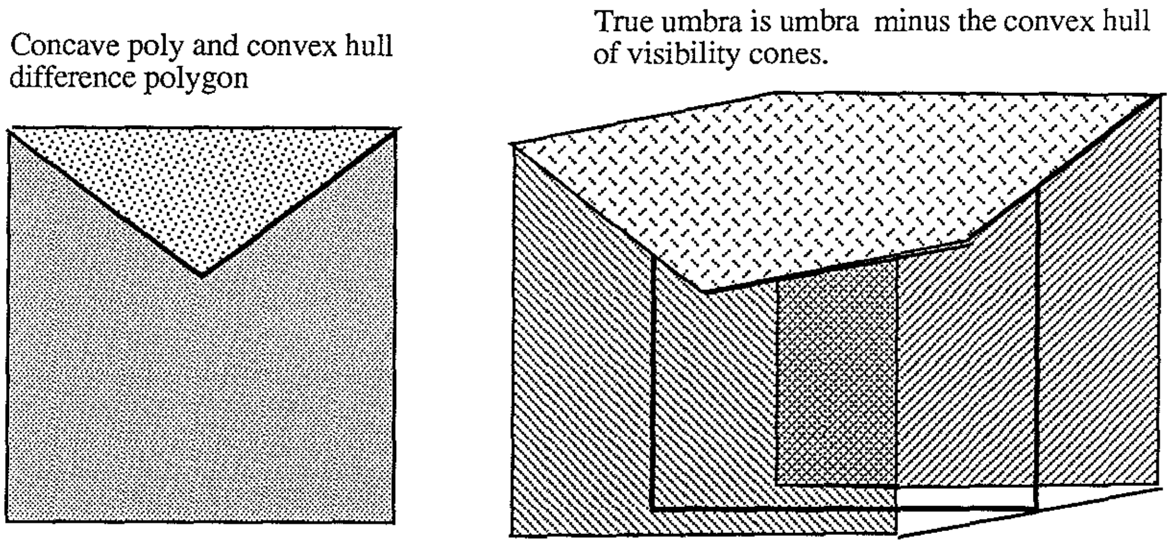

Concave poly and convex hull difference polygon
True umbra is umbra minus the convex hull of visibility cones.

Figure 3.14. Shadows cast by a concave polygon from a linear light source. (Only the shadows are shown). The true umbra is determined by computing the convex hull of the visibility cones and subtracting it from the intersection of the shadow cones.

This solution for a single concave shadow caster could be computed by extending

Chin's SVBSP trees with Thibault's 3D CSG algorithms [Thibault87], [Chin89].

3.3.2 Multiple Shadow Casters (possibly concave)

Generalizing the analytic computation to the shadowing effects of several disjoint non-coplanar polygons is more difficult than generalizing to concave shadow casters. The umbrae cannot be simply computed independently and then combined. The obvious counter-example to this proposal can be constructed by simply dividing one original polygon into two polygons. The union of the umbrae cast by each half is less than the original umbra. These problems can be explained best by dropping down a dimension and dealing with line segments in the plane (figure 3.15).

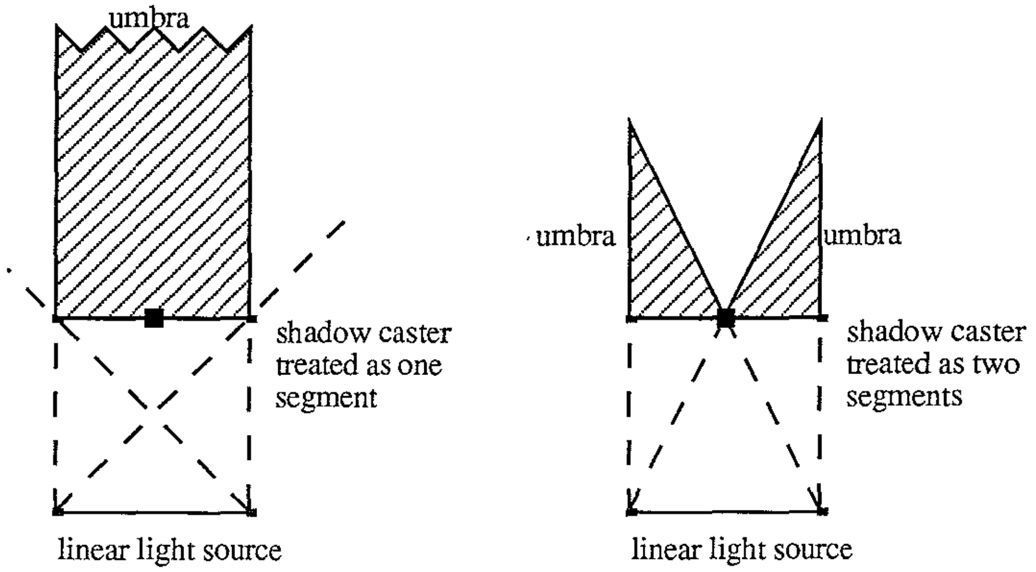

umbra
shadow caster treated as one segment
linear light source
umbra
umbra
shadow caster treated as two segments
linear light source

Figure 3.15 Umbras cannot be computed for each shadow caster independently. This example shows that dividing a shadow caster in two parts and computing the umbra yields a much smaller total umbra than that obtained by treating the two line segments as one segment.

It is also possible to construct examples where the two segments appear as one segment to one endpoint of the light source and as two to the other. Even in this case, treating the shadow casters independently leads to a computed umbra that is smaller than the true umbra, (figure 3.16).

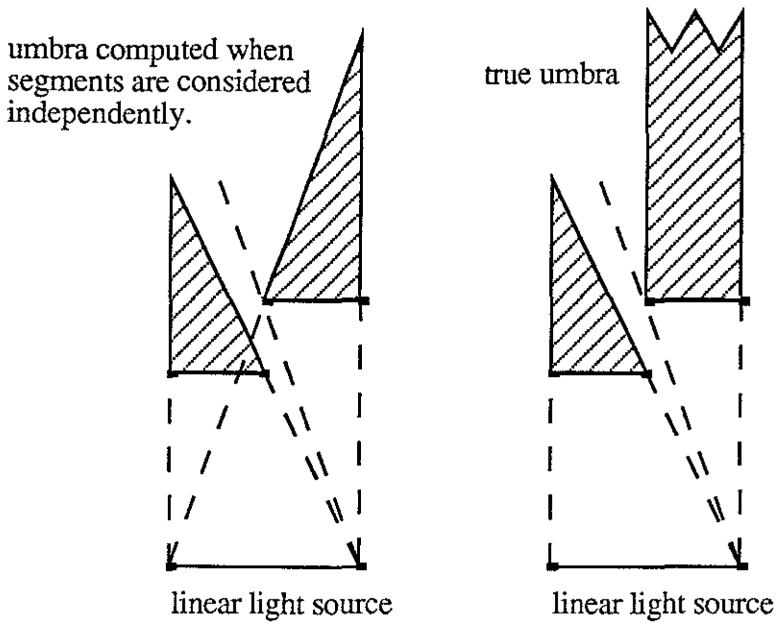


Figure 3.16 Even in the case where the segments appear as one segment to one vertex, we cannot simply union the umbrae of the two shadowing segments.

In general, given two line segments in the plane (disjoint, non-collinear), there are a pair of lines, called linear separators. The linear separators divide the plane into two regions where the two segments will appear as one unbroken unit and two regions where the segments will appear as two separate parts (figure 3.17). The light source can be subdivided according to these regions and the pieces can be processed accordingly.

In figure 3.16, one of the linear separators is the vertical line that intersected the light source at its midpoint. The correct umbra is formed by identifying the point on the light source at which the two shadow casters appear as two segments. Then we process the two halves of the light source independently. The intersection of the umbrae for each of the halves is the true umbra, (figure 3.18).

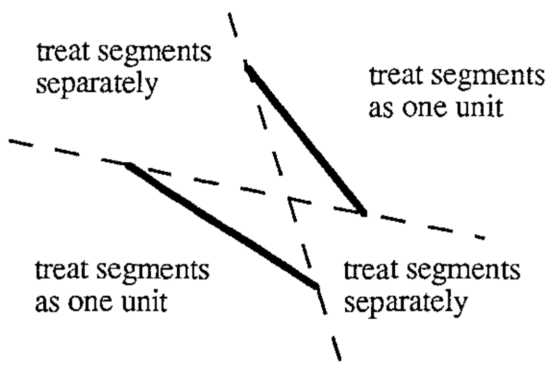

treat segments separately
treat segments as one unit
treat segments as one unit
treat segments separately

Figure 3.17 Two linear separators exist for a pair of segments. These lines separate regions of the plane where the silhouette of the segments appears as one segment from regions where the silhouette appears as two segments.

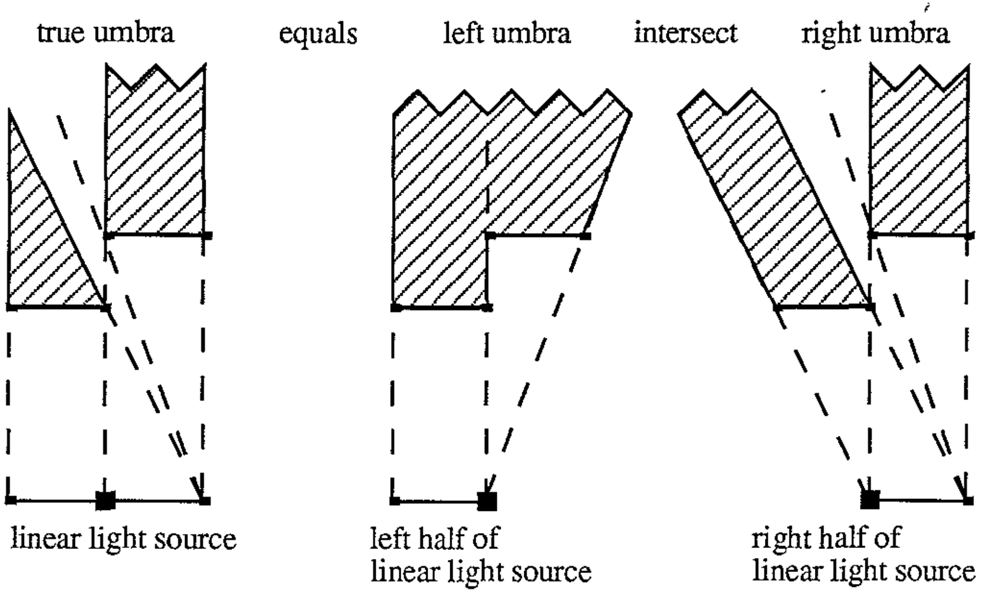


Figure 3.18 The correct umbra is computed by partitioning the light source into a subsegment that can treat the two shadow casters as one segment and a subsegment that must treat them as two separate segments. Intersecting the umbrae computed for each light source subsegment gives the true umbra for the entire light source segment.

3.3.3 A Naive Algorithm to Analytically Compute 2D Shadow Relationships

This leads to a naive algorithm for computing shadow relationships between N line segments with respect to a linear light source.

1. Compute linear separators for all pairs of line segments and subdivide the light source as necessary.

2. For each light source segment,
   2.1 Classify the shadow casting segments into subsets according to whether they can be treated as one unit or not. Note that this subset structure is an equivalence relation. For each light source subsegment, the subsets of occluding segments will have no intersection with each other and their union will exhaust all the segments.

   2.2. Compute the umbra regions for each of the subsets

3. Intersect umbra regions.

4. Any segment that is in some umbra for each of the light subsegments cannot be seen.

Note that actually computing the umbra regions is more than we need. We really only want to know whether any segments fall in the umbra regions. We can replace steps 2.2 and 3. with the following idea.

For each endpoint of the current light source subsegment, project the segments onto a line beyond all the segments, and keep track of the depth ordering of the segments. Thus we have a series of depth lists associated with each section of the projection line. The sublists are sorted by depth (figure 3.19).

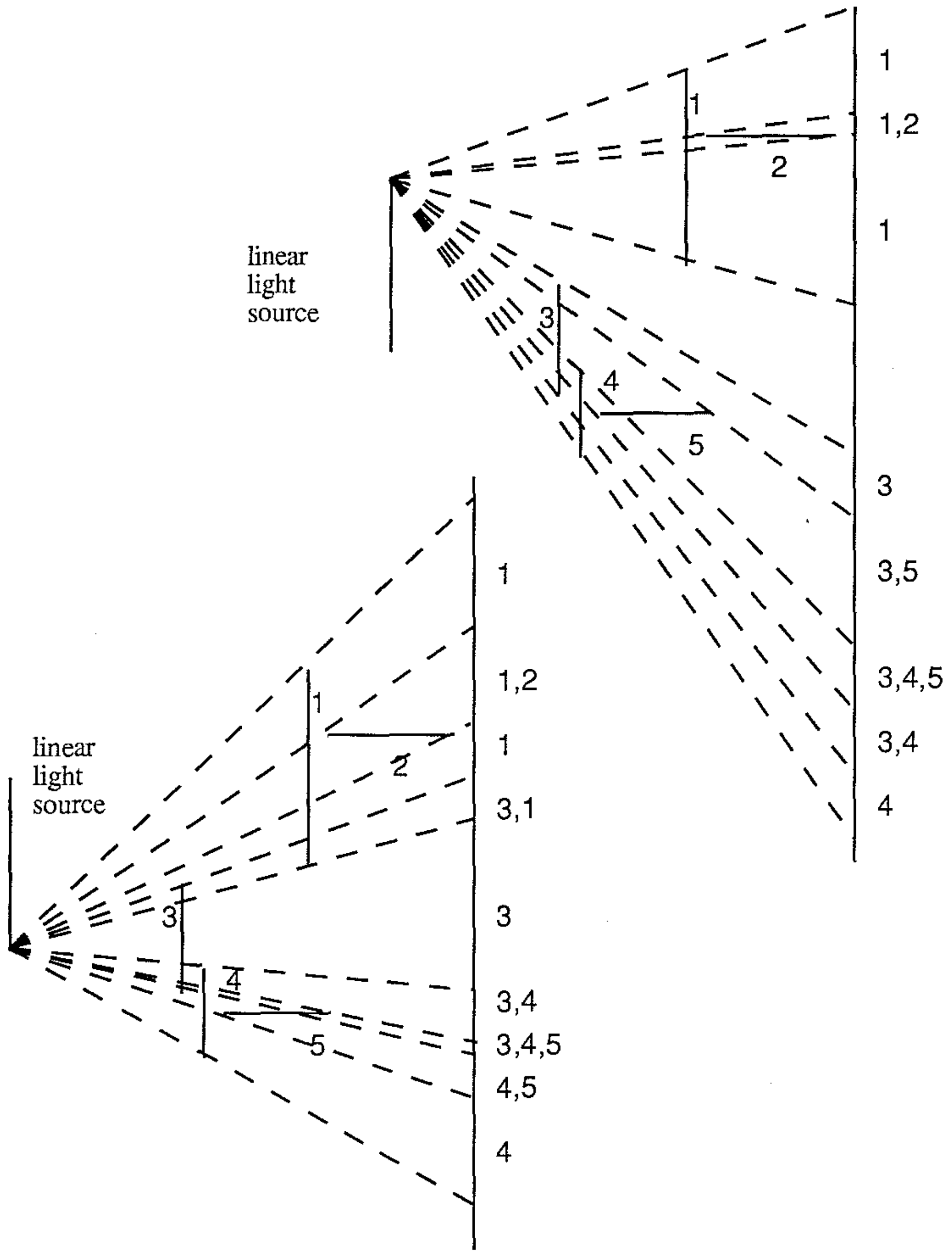


Figure 3.19. Projections of shadow casting line segments onto a segment beyond the shadow casters.

In this case we can see that segment 2 is covered by segment 1 in the projections from each of the light source endpoints. Thus segment 2 is in the umbra of segment 1. The situation of segment 5 is more complicated. Segments 3 and 4 cover segment 5. Since the light source lies in the quadrant of space defined by the linear separators of 3 and 4 that allow them to be treated as one unit, we can classify segment 5 as lying in the combined umbra of segments 3 and 4.

3.3.4 O'Rourke's worst case n^4 example

O'Rourke and Suri present an algorithm to compute the boundary of the region visible to a segment. The algorithm is different from the one I presented in many ways. They are constructing the boundary of the visible region. My algorithm sketch constructs the boundary of the hidden regions in the first variant, and identifies hidden segments in the second variant. The interesting thing about their (more rigorous) presentation is that they provide a construction that shows that the visible region can have $\Omega(n^4)$ vertices on its boundary. This places a lower bound on any algorithm that explicitly constructs its boundary. Since my second variant does not explicitly construct the boundary of the umbra regions, it may escape this worst case performance. The analysis of that variant is an open problem.

The main idea of their example is to let the light segment be horizontal. Place n closely spaced line segments immediately above and parallel to the light segment. The gaps between these segments permit $\Theta(n)$ cones of light to emerge above them. Place a second row of segments above the first, again parallel to the light segment. $\Theta(n^2)$ beams of light escape above this second row. These beams intersect $\Theta(n^4)$ times above the second row, creating a visibility region with $\Theta(n^4)$ vertices and edges (figure 3.20).

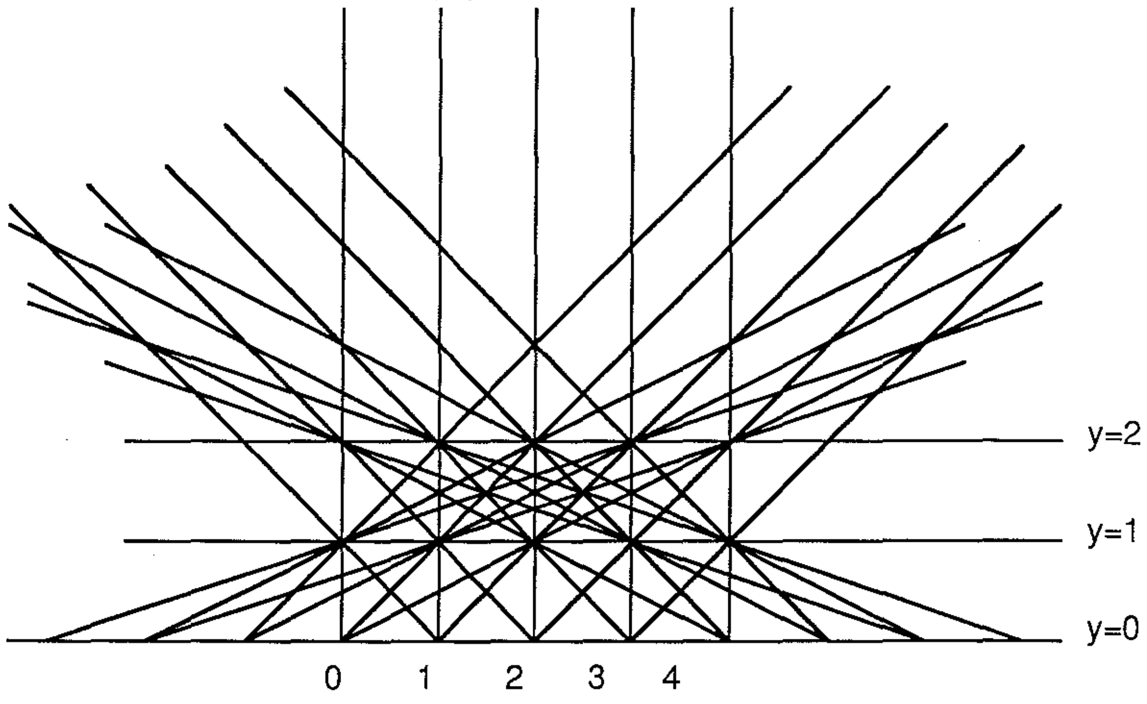


Figure 3.20 Five gaps on two parallel lines (y = 1 and y = 2) above the light segment (y = 0) produce 29 distinct intersections above the top line; in general, n gaps produce $\Theta(n^4)$ intersections. (taken from [O'Rourke87])

3.3.5 Extending the Analytic Algorithm to Three Dimensions

With no small effort, the two-dimensional algorithm can be extended to three dimensions. I can offer only a rough outline as to how this should be accomplished.

The notion of linear separator in three dimensions is more complicated than in two dimensions. Consider two triangles and the planes defined by a vertex of one triangle and an edge of the other (figure 3.21). Not all such planes formed will be separator planes.

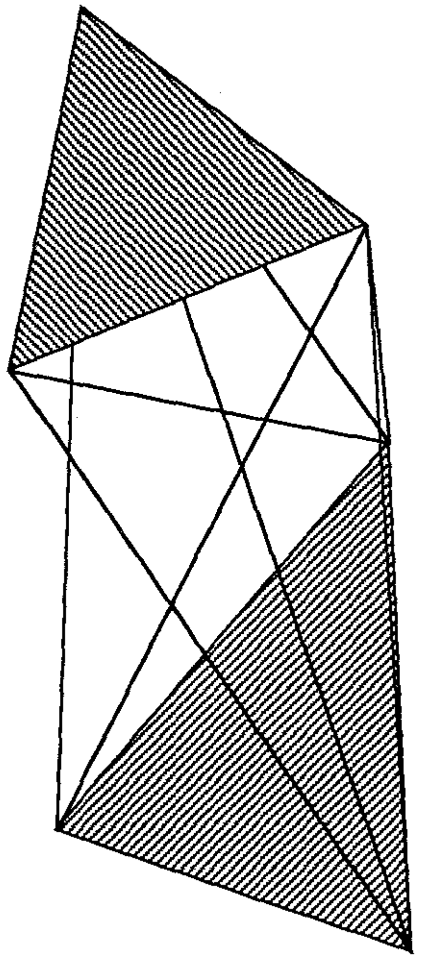


Figure 3.21 Define separators for two triangles embedded in 3D space by forming the planes defined by a vertex from one triangle and an edge from the other. Only some of these planes will qualify as separators. Each triangle must lie completely on opposite sides of the plane for it to qualify as a separator.

If the observer and one triangle are on the same side of all the separator planes then that triangle will overlay some part of the other triangle and those triangles may be treated as one concave polygon for purposes of the occlusion calculation. As in the two-dimensional case, these separator planes are used to subdivide the viewing, or light polygon.

For each fragment of the viewing polygon, the set of occluding polygons is classified into subsets that are treated as one concave polygon for purposes of computing occlusion relationships.

An analytical visible surface algorithm [McKenna87] computes a list of depth-sorted surfaces for every image fragment from each vertex of each subpolygon of the viewing polygon.

The image data structures produced by the analytic visible surface algorithm from each vertex of the viewing subpolygon must then be analyzed. These contain the information that allows us to check if a concave polygon (or set of polygons that can be treated as one concave polygon) occludes another polygon.

3.3.6 Other approaches

This dissertation is one of the first to try to solve the 3D problem in a practical way. The problem admits many different solutions. Some ideas that have been proposed but not implemented are mentioned briefly here.

3.3.6.1 BSP Shadow Volume Tree methods

If the shadow volumes are convex, a standard Boundary Representation CSG modeller can be used to explicitly construct the intersection of the shadow volumes for each vertex to get the umbra [Mantyla83][Requicha85][Laidlaw86][Putnam86]. The operations can also be computed with BSP trees [Rohlf90][Thibault87][Chin89].

However this is more difficult when the shadow volumes for each vertex are concave. If the shadow volumes for each light vertex are generated by unions of the shadow volumes for individual convex polygons, the result is almost always concave. Although no solution to the general problem has been proposed, it is likely that this approach could succeed.

3.3.6.2 Direct visibility-segment operations

This approach has been suggested by several people.

1. Construct the convex hull of two polygons . This represents the bundle of line segments joining points on the two polygons.

2. Cut out parts of this volume with occluding polygons. It is not necessary to construct the true umbra cast by the occluding polygons from the two polygons. We only need to know when the containing volume becomes disconnected. However, counter examples may be constructed to this idea (figure 3.22).

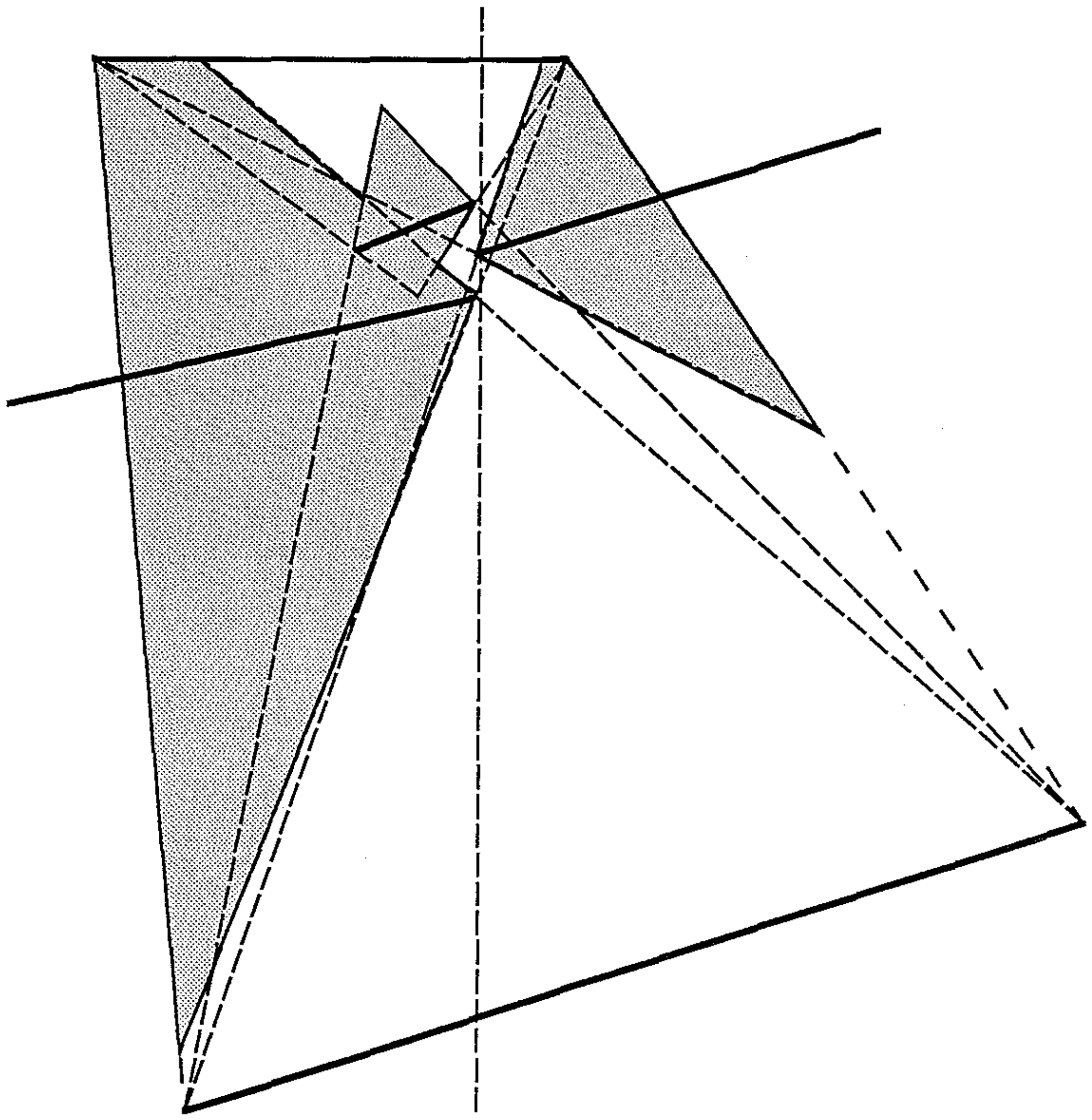


Figure 3.22 A counter-example to the idea of "cutting out the invisible portion between two polygons that might see each other". In this case, the dotted line represents the only way the polygon at the top can see the polygon at the bottom. The shaded area represents parts of the space between the two polygons that would be cut out by the effects of the occluding polygons. Moving any of the segments slightly to cut off the only line of sight does not separate the two polygons at top and bottom.

3.3.7 An Over-Estimation Implementation

Implementing the algorithm outlined in 3.3.4 would not only be a formidable task, but even assuming we could implement it reliably, there is no question that it would run so slowly as to be purely a research curiosity. If an implementation is to be attempted, it should have some chance of running on today's machines in a manageable amount of time. I consider a month or so to be the limit of *manageable*, and that does not imply *practical* in any sense. However, since what runs in a month on today's machines may run in hours or even seconds on future machines, I will present an algorithm that may take on the order of a month to run on a 7000 polygon, 269 cell model such as that of Sitterson Hall with a 10 MIP workstation.

Although the Sitterson Hall model has 7125 polygons, the number of distinct planes containing polygons is much smaller, 718 axial planes (338 normal to each of the X and Y axes and 42 normal to the Z axis) plus the planes defined by skew polygons (324 skew polygons in all). This is likely to be the case in most architectural models. Note that a factor of 10 in a N^3 algorithm means a factor of 1000 in running time.

We exploit this property. Restrict the set of occluding polygons to a set of polygons lying in an axial plane. The combination of a wall and a ceiling is not considered. The wall and the ceiling are considered independently. Thus some overestimation of the PVS will occur. Overestimation results in a time penalty because we render polygons that may not contribute to the image. However, we are guaranteed that the illusion of reality is not damaged by missing polygons.

Now we must only determine whether a set of co-planar polygons occludes a polygon with respect to the view polygon. After reading the prequel, it is natural to expect that we will project the set of coplanar polygons onto the plane containing the possibly visible polygons and then use the method described for computing the shadow cast by a concave polygon to determine whether the polygon is occluded. However the projection operation is troublesome. Points may need to be projected to infinity on the plane of the possibly visible segment. However, since we are restricting the occluding polygons to lie in a plane, we can project the possibly visible polygon up on the the plane of the occluding polygons. This operation has no problems with points going to infinity, (figure 3.23)

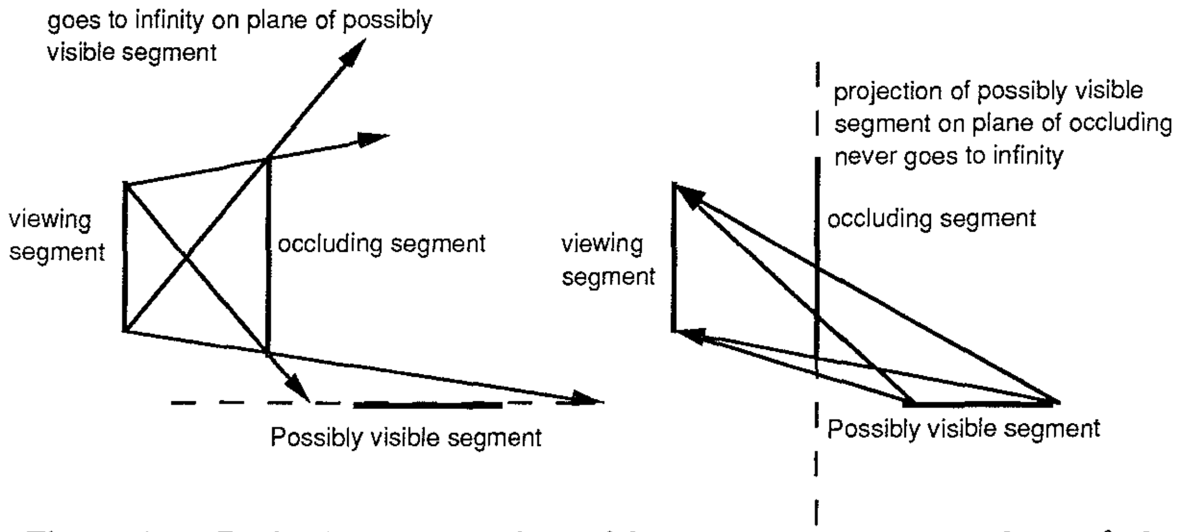

goes to infinity on plane of possibly visible segment
viewing segment
occluding segment
Possibly visible segment
projection of possibly visible segment on plane of occluding never goes to infinity
occluding segment
viewing segment
Possibly visible segment

Figure 3.23 Projecting the possibly visible segment onto the plane of the occluding segment instead of projecting the occluding segment onto the plane of the possibly visible segment avoids the problem of projecting points to infinity.

In this case we form the convex hull of the projections of the possibly visible polygon on the plane of the occluding polygons. This is essentially the window through which viewpoints must look to see the possibly visible segment. It is then a matter of determining if this window is opaque, i.e., that the occluding polygons cover it up. I use the portal computation algorithm developed in section 3.2 to compute the difference of the window and the occluding polygons. If the difference is null then the possibly visible polygon is occluded (figure 3.24)

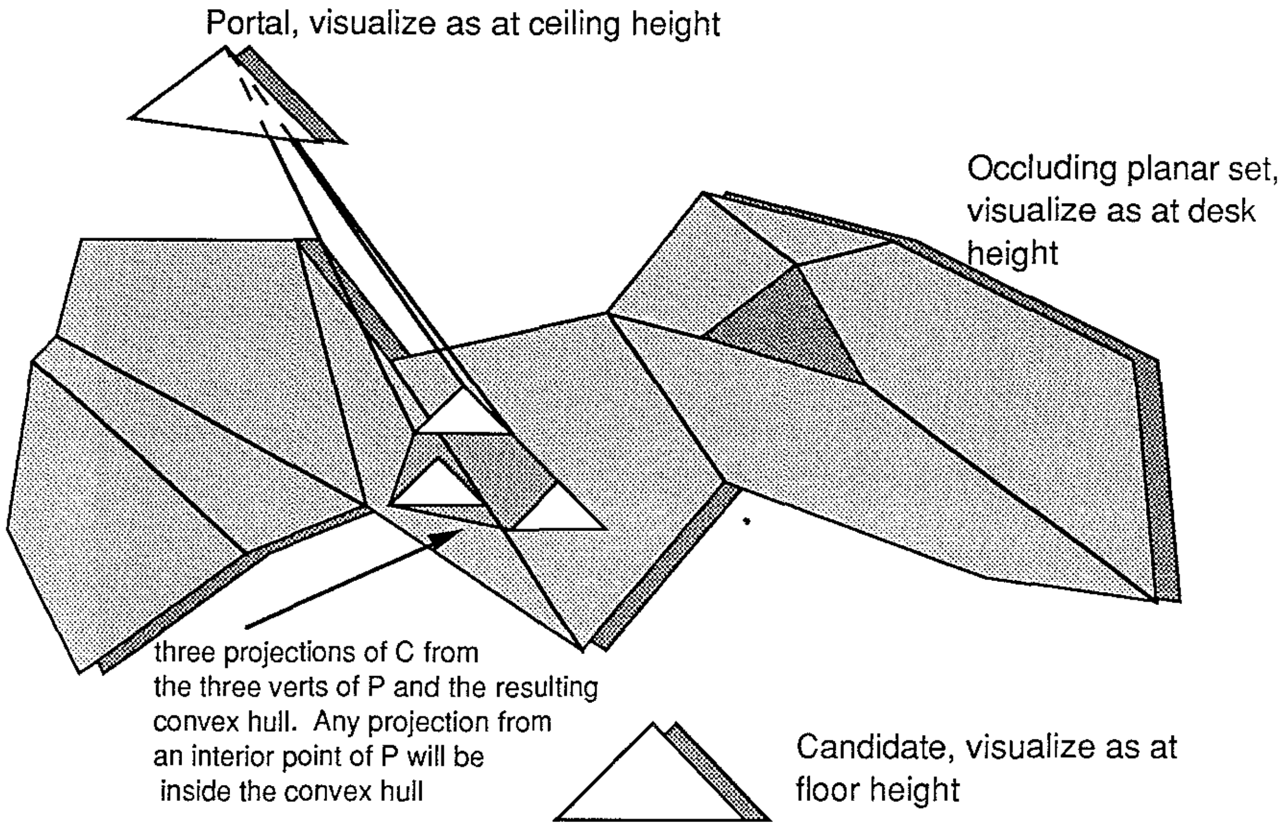

Portal, visualize as at ceiling height
Occluding planar set, visualize as at desk height
three projections of C from the three verts of P and the resulting convex hull. Any projection from an interior point of P will be inside the convex hull
Candidate, visualize as at floor height

Figure 3.24 The occlusion operation. The polygon begin tested is projected onto the occluding set of co-planar polygons from each vertex of the potal. The convex hull of the projections is computed. This is the window in the occluding plane through which the candidate can be seen. If the window is completely opaque then the candidate is occluded.

Computing this operation once is not very expensive. The expense comes from the fact that we will want to compute this operation many times:

```
for each portal
    for each polygon
        for each plane lying between the portal and polygon
            compute the projection, hull and difference operation.
            classify the polygon as visible or occluded, with respect to the portal.
```

The third *for* loop runs through all planes lying between the portal and the polygon. This is essentially a three-dimensional range-search problem which typically requires a mildly sophisticated data structure and algorithm to solve efficiently. I store the axial planes in three sorted arrays, one array for each set of parallel planes. Thus the three-dimensional problem is decoupled into three one-dimensional problems which are much easier to solve.

It is clear that some method is needed to reduce the number of times the *occlusion operation*, the projection, convex hull and difference operations, is performed. This problem is ripe for any number of slick and incredibly complex data structures. However, since we are aiming for something that can be implemented, the following idea is the one I used.

The principal idea is that if a volume is occluded then all the polygons inside the volume are also occluded. The occlusion operation can be performed on a possibly visible volume in the exact same manner as it is performed on a possibly visible polygon.

Given a portal, a set of co-planar occluding polygons and a hierarchical spatial subdivision of the model, we can try the occlusion operation first on the top level volume. If that succeeds than we are done, and all polygons inside the model can be classified as occluded. If it fails, then we can try the occlusion level on the children volumes in a recursive manner. In any reasonable hierarchical model space subdivision, the number of volume nodes should be linearly proportional to the number of polygons. Thus in the very worst case the algorithm will go twice as slow. It is likely that in the average case on large buildings that this technique would accelerate the algorithm considerably. Note that the result of the hierarchical model space subdivision described in chapter 2 can be used for this phase of the algorithm.

In my experiments on the Sitterson Hall model this technique did not appreciably speed the algorithm because no high level volumes were classified as occluded. If the number of polygons had been much larger, for example, if the model were fully furnished, I think the algorithm would be more successful. See Appendix A.5 for C code to implement an over-estimation method.

3.4 Impressions and Comparisons of the Implemented Algorithms

It is difficult to offer absolute conclusions for the preference of one method over another without specifying the limits on one's computational resources. In my situation, with several general purpose 10-20 MIPS networked workstations, model databases in the range of 10,000 polygons, and roughly 10,000 portals, the targeting methods are the clear practical choice. A description of the system in practice follows.

To build a display file for a large model in this environment I first spend a few hours tweaking run time parameters on the partitioning phase of the computation to get a model subdivision with the desired number of cells. This number is picked based on how much time I think I have to process the cells. For example, if the deadline is tomorrow, I might only use 10 cells. If the deadline is five days away, I might use 50 cells. Then I distribute the cells among as many workstations as possible. After most machines finish their tasks, I kill any unfinished jobs and restart those remaining tasks with less demanding parameters, e.g., either a larger spacing between sample portal viewpoints or fewer sample rays fired at each polygon. Lastly, after interactively viewing the results, I might notice a few cells that have noticeably missing polygons. For these cells I use the edge-deletion or over-estimation methods. Note that all this can be automated; it is simply the process of managing a generic large scale computation.

On machines that possess hardware to accelerate the fixed-sampling-pattern methods, such as Pixel-Planes 5 or the SGI Iris 4D series of machines, the increase in speed would make the fixed pattern methods preferable. It is also worth noting that since the same hardware is used to generate images and pre-compute visible polygons, the sampling density is, by default, the necessary sampling density.

For a highly parallel SIMD machine such as Thinking Machines' Connection Machine or the MasPar, a ray-intersection algorithm might make the targeting approach superior [Delany88]. Such an algorithm processes many rays in parallel. No known coherence of the sample pattern is used, as in a pipelined implementation of a Z-buffer algorithm. Since one can generally get more useful information per ray from the targeted sampling than from fixed pattern sampling and targeted rays cost about the same as rays from a fixed sample pattern, targeted sampling is likely to be more effective in this case.

For the 269-cell Sitterson display file, I used a combination of sampling methods and over-estimation methods. As a first pass, I used a low-sampling-rate fixed-pattern method on each cell. Later, if I encountered a cell that had noticeably missing polygons, I re-processed only that cell with the edge deletion method, or used a higher sampling rate with a fixed-sampling-pattern on that cell. When the 269-cell Sitterson model was computed, I had not yet implemented the targeted sampling method.

All the Orange Church display files, including the 106-cell display file which appears in the SGI demo suite for the VGX series machines, were initially computed using the targeted sampling methods. Depending upon the cell, three to seven rays were targeted at each polygon from 36 inch intervals on the portals. A few exterior cells had very large portals. These cells were processed using much larger intervals on the portals. One cell in the stairway was recomputed using a sample pattern. There are still a few cells with missing polygons, but they are rarely encountered.

When I first developed the targeting method, I performed some timing experiments on a particularly troublesome cell, cell 226 of an early church model. (litchurch.roundoff.poly 7812 polygons). Cell 226 is an entryway alcove in the fellowship hall of the church. The standard 2 foot portal, 392 rays-per-portal cosine-weighted angularly-jittered hemisphere sampling distribution worked poorly. Many small polygons were not hit by the rays fired from the portals. In particular, the small polygons around the circular windows at either end of the great hall were difficult to detect with a fixed sampling pattern. This lead to efforts to find some variant of the sampling method that would work for this cell.

I had thought about implementing the targeted sampling idea earlier, but this particular case forced me to implement it. Since Pixel-Planes 4 could display about 32K quadrilaterals, (39K triangles) per second, I hoped to achieve a partition where no cell would contain more than 2000 polygons, as this gave a guaranteed 16 Hz frame rate. The over-estimation method was declaring about 3200 polygons visible for this particular cell. The initial attempt at using a fixed sample pattern method found 1000 visible polygons. It was my guess that about 2000 polygons were actually visible. So I tried varying the parameters of the fixed sampling methods. It was clear that many samples were required to find all visible polygons, so I implemented the targeted sampling method, which proved to be clearly superior for this particular case.

Comparisons of different ways to compute the PVS for the entryway cell are documented below. The PVS for the cell was computed only a few ways since each computation took anywhere from 40 CPU minutes to 30 CPU hours to complete. Each trial is classified according to the general method used. The specific values of controlling parameters are also listed. The time in seconds to compute the PVS for the cell is given along with the size in polygons of the computed PVS. A relative time is also given, in terms of the fastest method. Finally, although I can not be sure of the exact PVS, I could not visually detect any missing polygons in the PVS computed by one of the targeted sampling methods. That method computed 1587 polygons in the PVS. Each of the other methods has its PVS size listed relative to that result. Since only a few trials were computed, and some reasoning on the results of each trial was used to determine what to do next, comments on each trail appear with the data for each trial rather than in one large commentary after all the trial data.

	time in seconds	# visible polygons	time relative to fastest method	# visible polygons relative to most correct method
Fixed pattern sampling				
Cosine weighted 392 rays at 24 inch portal viewpoint samples	2235.81	1026	1.637	0.647

Comments:

This initial attempt worked poorly. Only 65% of the visible polygons were detected. However, most of the missing polygons were quite small.

Members of the Walkthrough team often used databases with every other polygon removed to double the update rate, which allowed them to test new interfaces, etc. It is suprisingly easy to navigate with only half the polygons present. The 65% of visible polygons detected here was probably greater than 95% visible polygon *area* so the computed PVS could have been acceptable under some circumstances.

Linear radius, 450 rays at 12 inch portal viewpoint samples	7137.24	1259	5.225	0.793

Comments:

This was a simple attempt at roughly doubling the sampling density. This got about 79% of the polygons, a 14% improvement, at 3.2 times the cost.

Linear radius, 5000 rays at 12 inch portal viewpoint samples	54909.91	1490	40.198	0.939

Comments:

The number of samples was increased greatly to see if a huge increase in the number of rays would get all the polygons. I also changed the pattern to the linear radius hemisphere pattern.

A factor of eight in time hit another 15% of the polygons. The result gained here was actually pretty good. It took more than a casual glance to know that any polygons were missing, but the time cost was excessive.

	seconds	polygons	rel. time	rel. polys
Targeted sampling				
1 targeted ray at each polygon, from the center of each portal.	**1365.99**	**1509**	1.000	**0.951**

Comments:

This was the initial trial with the targeted sampling method. A single ray was targeted at each polygon from the center of each portal. The time used was just over half that of the simplest fixed sample pattern method, yet it found more visible polygons than the fixed sampling method which took forty times as long.

It is worth noting that these figures are slightly misleading, because the polygons that the targeted sampling method missed were much larger than the polygons that the fixed sampling method missed. Floors in other rooms are only visible through doors. A sample targeted at the floor will often miss the door and hit the wall. This results in the floor being erroneously declared not visible. The fixed sampling method, on the other hand, only missed polygons because of their size. This suggests that the targeted sampling method should use apparent area to compute the number of targeted samples for each polygon, but I found later that simply using several rays, e.g. 5, made this event very rare. The flip side of this coin is that a few rays targeted at very small polygons could be used to augment a fixed sampling method.

7 targeted rays at each polygon, from the center of each portal	**7366.80**	**1531**	5.393	**0.965**

Comments:

This was an unsuccessful attempt to get those remaining polygons. Only a 1.5% increase in the number of detected polygons for a factor of 5 increase in time. Part of the problem was that all rays were being fired from the center of the portal. It was possible for that position to be degenerate somehow, e.g. a column or other obstruction might be sitting directly in front of that point. The solution is to use more initial points on the portal.

7 targeted rays at each polygon, from points spaced 24 inches apart on each portal	**47,297.51**	**1587**	34.625	**1.0**

Comments:

Many more initial points on the portal were used. The number of targeted rays per polygon was kept at seven, which is overkill. As far as I could determine, there were no missing polygons in this example, but the time was again excessive. When I computed display files for later models of the church, I typically used 3 targeted rays-per-polygon from points spaced 36 inches apart on the portals.

Over-estimation method: **109,936.34** **3211** **80.481** **2.023**

Comments:

This trial missed no polygons, but classified many invisible polygons as visible. In this case those extra polygons bumped me over my self-imposed limit of 2000 polygons per cell, which corresponded to update rates of 16 Hz on Pixel-Planes 4. The cost of this method is too high to allow it to be practical.

Method	**PVS size**	**relative PVS size**	**relative time**
392 ray cosine-weighted hemisphere. Source points spaced 24 inches.	1026	0.647	1.637
450 ray linear-radius hemisphere. Source points spaced 12 inches	1259	0.793	5.225
5000 ray linear-radius hemisphere. Source points spaced 12 inches	1490	0.939	40.198
1 targeted ray per polgon. 1 source point per portal	1509	0.951	1.0
7 targeted rays per polygon. 1 source point per portal	1531	0.965	5.393
7 targeted rays per polygon. Source points spaced 24 inches	1587	1.0	34.625
Over-estimation	3211	2.023	80.481

Table 3.1 Summary of timing comparisons

3.5 PVS computation taking account of viewing direction

If the PVSs are computed and stored for each portal instead of each cell, then it should be possible to display the PVS only for the portals that fall in the viewing frustum. Compute the portals lying in the current view frustum and then render the PVS associated with each portal. The difficulty is in handling overlap in portal lists. Since the portal regions are triangulated, it is common to have many portals per cell. The 269 cell Sitterson model subdivision had an average of almost 30 portal triangles per cell. Many polygons would occur in the PVS of several portal triangles. Each polygon should be rendered only once.

It may also be possible to subdivide the model based not only on viewer position, but also on view direction. This approach has been shown to accelerate ray-tracing [Arvo87].

Chapter IV

A Radiosity Implementation

The radiosity lighting model has several properties that make it desirable for virtual building environments.

• It accurately models the diffuse interreflections that dominate the interior of a building.

• The lighting information may be pre-computed and stored as color values at polygon vertices. These values are linearly interpolated by hardware during display. This effectively eliminates any lighting calculations at display time, and yields rapid rendering.

• The process is a linear system. Thus the contributions of several different light sources may be computed independently. A linear combination of these solutions may be computed during display, allowing the user to brighten and dim lights. This gives an added dimension of interactivity at little cost.

For several years the best-known solution to the radiosity lighting model used time and space quadratic in the number of patches [Cohen85]. This made it prohibitive for use in practical systems with real building models. The shooting algorithm used in the progressive refinement solution to radiosity makes radiosity practical [Cohen88]. The algorithm runs in linear space, and usually only linear time is required to converge to an acceptable solution. It is no longer a research curiosity but a tool for virtual environments.

The UNC Building Walkthrough Project uses the radiosity shading model. A brief description of our implementation follows.

4.1 A Ray-Casting Approach

We use a modified shooting approach to pre-compute the radiosity solution [Cohen88]. The sampling process uses an adaptive ray-casting based on a jittered hemispherical distribution [Airey89], the same used for the PVS calculation in the previous chapter. This differs from the Z-buffer based hemi-cube introduced by Cohen, et. al. The hemi-cube is an approximation to the illumination hemisphere. The hemi-cube has five faces, each subdivided into a regular grid. The Z-buffer algorithm capitalizes on the regular grid to accelerate the hidden-surface calculations necessary to compute energy transfer coefficients, known as *form factors*. The Z-buffer method computes samples faster than ray-casting, but requires the overhead of transforming the database first. The Z-buffer method is faster than the ray-casting technique when the number of samples taken outweighs the cost of transforming the database.

Cohen et. al. developed a two-level patch and element structuring scheme to keep the number of samples shot from the same point high. Energy is shot from a coarse grid and collected by a finer grid. The two-level structuring scheme is unnecessary in the ray-casting method, since the cost of casting any ray is essentially the same, no matter what shooting point is used. The energy may be shot from a fine grid at the same expense as shooting from a coarse grid.

At each iteration step, we adapt the resolution of the hemispherical sampling distribution as a function of unshot radiosity to keep the radiosity-per-sample constant. Airey and Ouh-young observed, empirically, that the unshot radiosity at each step decreases as a negative exponential [Airey89]. Thus, the number of samples needed at each step also decreases as a negative exponential. Once the number of samples taken at each step drops below a certain point, the ray-casting approach is faster than a Z-buffer method. The cross-over point and the number of iterations determines which method is faster. In our experiments, the ray-casting approach was slightly faster. Ming Ouh-young and I proposed a hybrid algorithm that used the Z-buffer method for early iterations and switched over to the ray-casting method at the cross-over point [Airey89].

An advantage of ray-casting sampling algorithms is flexibility. We have been able to experiment with light-emitter distributions other than true diffuse emission, such as spotlight-like distributions, with only small changes in our software. Wallace, et al., use ray-casting to sample the light source from the model vertices to decrease solution errors due to limited sampling distributions [Wallace89]. They also note other advantages, such as the ability to use exact parametric descriptions of objects rather than polygonal approximations. The exact parametric descriptions allow accurate shadows and avoid polygonized silhouettes.

4.2 Interactive Light Manipulation

We have extended our radiosity program to compute the contributions of several different light circuits. For each patch we compute a vector of radiosities, one entry for each light circuit. Since a value for the red, green and blue channels must be stored for every patch for every independent set of lights, the storage requirement is large.On workstations used to compute the radiosity solution one may run the radiosity program several times, once for each light circuit, and combine the results as a post-process. However, the results must fit into display memory. We devised an approximation to save space. An average color is computed from the colors due to each light circuit, and an 8 bit intensity value is computed for each light circuit.

The radiosity process computes an array of color values for each vertex,

$<r,g,b>k$, with $0 <= r,g,b < 256$,

one for each of the k light circuits. We compute an average color,

$<R,G,B> = \sum (<r,g,b>k);$
max = MAX(MAX(R,G),B);
$<R,G,B> = <R/max,G/max,B/max>;$

Then we compute an eight bit intensity value for each of the k light circuits,

$<I>k = (rk/R + gk/G + bk/B)/3.$

The storage required is k+4 bytes rather than 4*k bytes, assuming word boundary restrictions. The penalty for this savings is that the color of each surface stays constant, regardless of the light circuit settings. Although many lights encountered in real models are not white (especially incandescent lamps), this approximation has been useful.

During display, the user may alter global settings for each of the k light groups, i.e,. turn some off, brighten others, etc. We scale the average <R,G,B> value stored at each vertex with the dot product of the global settings and the light group intensity values stored at each vertex. This takes roughly one extra frame time to compute. The result is then stored at the vertex until the user changes the global settings again. Thus, any combination of k lights can be interactively modified during display. In our system, k is 20.

4.3 Using a Physically Based Lighting Model on Non-Physical Models

A physically based rendering method requires physically based models. Although AutoCAD is a powerful modelling tool, it does not guarantee topological consistency of the models it produces. Several problems must be handled.

• Most radiosity programs expect polygons to be oriented so that, for example, the vertices appear in counter-clockwise order when the viewer looks at the front of the polygons. This reduces the cost of the radiosity computations and allows back face culling to be used during display. It is difficult, if not impossible, to construct correctly oriented polygons from many modelling primitives provided by AutoCAD, such as extruded polylines. We attempt to solve this problem with a radiosity program that can keep track of the radiosity for both sides of every polygon, and only allocates storage for a patch when it is hit by light. Display polygons can be generated for both sides of a patch if both sides received a certain amount of energy, or alternatively, the side which received the most light can be used as the front of the polygon. This idea requires only light-emitting polygons to be correctly oriented.

• Z-buffer algorithms suffer problems caused by coincident co-planar surfaces. The plane-sweep algorithm in Chapter 2 can be used to detect coincident co-planar surfaces. This allows the modeller to fix the problem before the radiosity program is run. To handle any remaining coincident surfaces, our radiosity program only transfers energy to one of the coincident patches, preventing the other from being generated. This avoids many display problems.

• Improper edge adjacencies. See the discussion at the end of Chapter 2. The retesselation program described in Chapter 2 can transform polygonal surface tilings into planar subdivisions, a tesselation in which every edge joins two and only two polygons except at surface boundaries. This is necessary to prevent cracks in curved surfaces and shading discontinuities in planar surfaces.

• There are other problems related to improper edge adjacencies. The largest problem is polygons that incorrectly abut other non-coplanar polygons, i.e., they do not share edges. For example, wall polygons may abut floor polygons without sharing edges. Thus one polygon spans the floor in two separate rooms. If the light is on in one room, but not in the other, the linear interpolation of vertex values allows light to *leak* under the wall. This problem can be solved by detecting the incorrect adjacency and cutting the floor polygon where it meets the wall polygon.

4.4. Adaptive Refinement

Pre-computation is a good strategy and should be applied to viewpoint-independent image features. Unfortunately, only a few tasks, such as visibility relations and diffuse shading in static environments, fall into this category. To deal with image features that cannot be handled by pre-computation, and features that strain the limits of the display subsystem even with pre-computation, we turn to adaptive refinement.

An object can be approximated at various levels of detail. We use the approximation that most closely fits the needed level of interactivity at the moment. This idea is well-known and regarded as a common-sense notion among flight simulator developers. However, since our projected user, an architect, also constructs the model, we have concentrated on automatic applications of the principle

The radiosity process dices model polygons into patches. In our experience, this increases the number of display polygons by a factor of four to ten. Since our display system, Pixel-Planes 4, takes a constant amount of time to render any color-interpolated quadrilateral, regardless of screen size, a radiosity shaded model takes four to ten times longer to display than the original model. (For commercial graphics workstations, which tend to be pixel-fill limited, this effect may be much less noticable.)

The dicing due to radiosity can be used to produce levels of detail automatically. We have adopted hierarchical polygons as our display primitive (Figure 4.1). This is sometimes called a pyramidal representation.

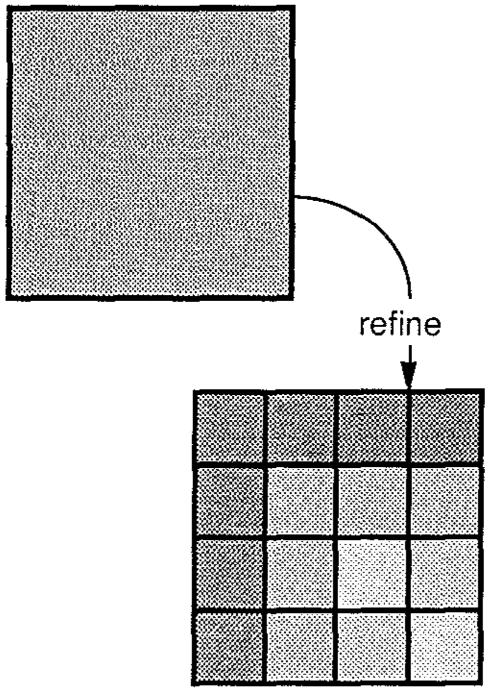


Figure 4.1. A Hierarchical Polygon. In the Actual Image the Patch Values are Stored at Vertices and Interpolated to Obtain Smooth Shading.

Each polygon has an associated list of polygons that can be used to refine it. When the user stops, the image "sweetens." The resolution level of the hierarchical polygons displayed is increased; we display the patches.

We smooth the transition from one quality level to the next with pixel-level blending to minimize user distraction. The blending takes advantage of the huge aggregate SIMD computing power of the Pixel-Planes 4 machine by computing the blending function at every pixel simultaneously. The blending implementation uses fifty interpolation steps and occurs in a fraction of a second.

The level of resolution refinement is fixed by the choice of patch size made during the radiosity pre-computation. We have developed secondary levels of refinement that are dependent upon the current view and light circuit settings. The secondary levels of improvement are slower since they involve computation during display, but they can markedly improve an image that suffers from coarse patch sampling.

We approximate bilinear interpolation across a quadrilateral patch with two triangles so the shading can be expressed as a Pixel-Planes 4 linear expression [Fuchs85]. This can cause problems. Note that if the color values at the four corners of the quadrilateral are a,b,c,d, then the bilinearly-interpolated color at the center of the patch is (a+b+c+d)/4. Since a quadrilateral can be triangulated in two ways, the value at the center is either (a+c)/2 or (b+d)/2, depending upon which diagonal is chosen.

During the first adaptive refinement step, we choose the diagonal uniformly. As a secondary adaptive refinement step, we choose the diagonal that connects the two vertices that are more closely matched in color. This tends to make the diagonals run perpendicular to the shading gradient (Figure 4.2).

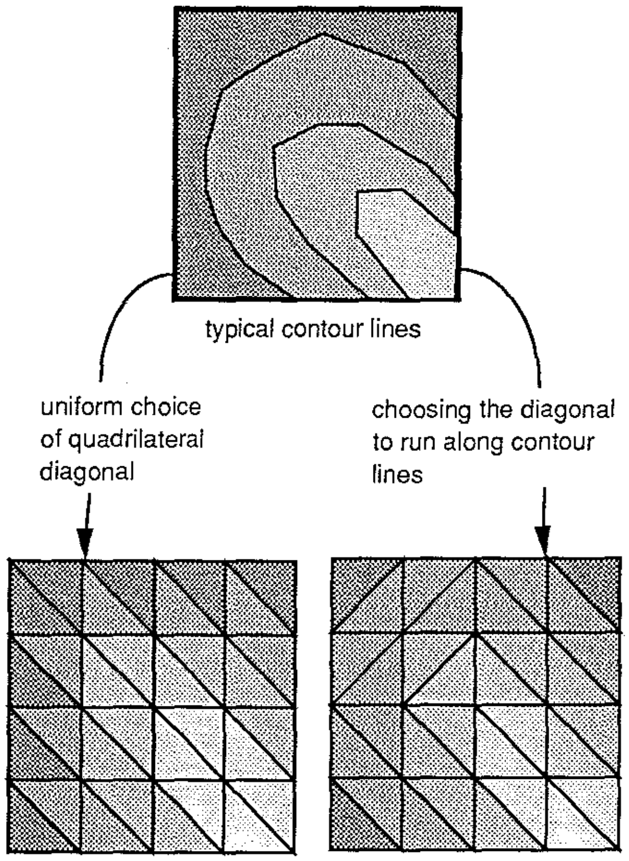

typical contour lines
uniform choice of quadrilateral diagonal
choosing the diagonal to run along contour lines

Figure 4.2 Uniform Choice of Quadrilateral Diagonal vs. Difference Directed Choice. In the Actual Image, the Colors are Transfered to Patch Vertices and Linear Interpolation Provides Smooth Shading.

Even after choosing the best diagonal, the approximation may be inaccurate. A patch can be subdivided into four patches. The color value at the new center vertex is computed with bilinear interpolation. The process is applied to each subpatch recursively.

Following adaptive refinement of shading, the image is anti-aliased. We use an algorithm developed by Fuchs et al. that builds the anti-aliased image using supersampling [Fuchs85]. A new image is computed for each supersample and blended smoothly into an accumulated image using the supersample filter weights.

Chapter V

Contributions and Future Work

This dissertation has focused on improving update rates in a visual simulation system for buildings, UNC's Building Walkthrough. The contributions to visual simulation include:

• An implementation of a complete interactive visual simulation system for buildings.

• A characterization of the properties of architectural databases that can be exploited to improve system performance. The most important property is the observation that potentially visible sets (PVSs) of features are similar across local regions of space called *cells,* which approximate rooms. Potentially visible sets change drastically on the boundaries of cells. Transparent portions of the cell boundary are called *portals*. Any algorithm that attempts to improve the display update rate through pre-computation is likely to use these concepts.

• A cell's PVS is defined as the union of polygons interior to the cell and those exterior to cell which are visible from points on the portals. Thus, the problem of computing the PVS for a cell is reduced to the easy problem of identifying polygons inside a cell and the hard problem of computing weak visibility between portals and polygons.

• A model-space subdivision algorithm to locate cells and their portals is described and implemented. Special attention is given to an algorithm to correctly compute the polygonal definitions of portals, which are typically defined only implicitly in the model.

• Two different algorithms to compute visibility between two convex polygons are presented. The problem is similar to computing illumination from a polygonal area light source. One approach is an analytic algorithm which computes a conservative over-estimation of the PVS. That algorithm is similar to analytic shadow computation. The second approach uses point sampling methods. This approach may under-estimate the PVS, but it is easy to trade cost for results and is easy to implement. This is the clear choice for practitioners. Note that if point sampling methods are used to estimate visibility, it is unnecessary to compute the polygonal definitions of the portals. It is only necessary to know whether a sample point on the boundary of the cell is transparent or opaque.

5.1 Future work

In any visual simulation system, the effectiveness of the whole and not the success of individual parts is the important thing. So while this dissertation demonstrates the quantitative success of increasing update rates with a pre-processing algorithm, its integration into the Walkthrough research project was at least as important. With that in mind, we cast a critical eye at the system and suggest improvements.

5.1.1 A better modelling system

Without a doubt, modelling is the effort that consumes the most human resources. AutoCAD is not a successful commercial product without reason, but it is clear that it did not evolve with 3-D architectural modelling as the focus. Writing a modelling system is hard; one needs a critical mass of users to test and drive system development. For this reason, it is difficult for specialized modellers to develop and become accepted. However, the problem remains: we need better modelling systems. A good metric for modelling systems would be the time it takes to accurately model a structure of a certain square footage.

Another related comment is that the modelling system could be written with anticipation of the display process that will follow. The creators of AutoCAD did not know that we would be attempting to interactively move through a model shaded with radiosity and created with their system. The modeller itself could use the notion of cell and portal to aid the subsequent display process.

5.1.2 A better man-machine interface

The graphics cluster in the computer science department at UNC at Chapel Hill has a history of experimentation with graphical interface devices that comes from the long standing GRIP molecular modelling project run by Dr. Brooks. Many physical input devices and often many more logical models for each device have been tried on many projects. The Walkthrough project itself has several different logical models for the box containing two three-degree of freedom joysticks and one slider. The sticks can control rotation and translation in many different ways. We have used a treadmill with bicycle handlebars to navigate through the simulated building. Work in this area is continuing. The head-mounted display project underway at UNC has already accomplished a lot and promises much more.

There is also the aspect of interaction other than navigation. A limited example of this is our ability to turn on and turn off lights. One could imagine the ability to open and close doors and move furniture. Our system allows the user to pass through walls, ceilings and floors which is nice sometimes, but can allow a user to get lost inside interstitial spaces. Adding collision detection would be a noticeable improvement on the existing system, although Brice Tebbs has noted that if the simulation is very good, the users shouldn't bump into walls anymore than they do in the real buildings.

5.1.3 Better update acceleration methods

Although the system described in this dissertation succeeds at increasing the update rate, the large amounts of time needed for pre-processing make it impractical to make small changes in the model. The user feels compelled to make several changes at once so that less time is spent pre-processing the model for display. The pre-processing methods could be significantly accelerated with large scale parallelism. I have used parallelism on the scale of the number of cells in a model by distributing cells to individual workstations. This can cut the total time to compute the PVS for all the cells down to the time needed for the most difficult cell. However, it requires many independent machines. More efficient parallel methods using truly parallel machines could be developed.

It is also quite possible that some other methods which require much less pre-processing could be used. It may be possible to transfer some of the work back into the display time computation, although this can be difficult because time is at a premium there.

The open problem in theoretical hidden-surface algorithms is an output-size sensitive hidden surface algorithm, one that runs in polylog time in the number of visible surfaces. Practitioners need an output-size sensitive sampling algorithm. Several people have suggested variants on an algorithm that draws surfaces from front to back and stops when the image is sufficiently complete but before all surfaces have been processed. This requires some type of *a priori* knowledge about the database, but would require much less pre-processing than the methods proposed in this dissertation.

5.2 Different algorithm paradigms- randomization and sampling.

Many early hidden surface algorithms were analytical in nature or were easily adapted from taking point samples to producing analytical solutions. Scan-line algorithms are the best example. However, it is apparent that sampling algorithms are used for fast image generation as well as realistic image generation. Hardware Z-buffer hidden-surface implementations are used in most graphics workstations. Software Monte-Carlo integration methods, distributed ray-tracing and radiosity among them, produce very realistic images.

Although I think analytical algorithms are important theoretically, I believe that sampling based algorithms, in particular those with an element of randomization will be superior for practical purposes. Analysis of randomized sampling algorithms is a relatively new field in algorithm design. That knowledge could be transferred to many algorithms in computer graphics.

References

[Abram87] Abram, G. (1987) Parallel Image Generation with Anti-Aliasing and Texturing Ph.D Dissertation, The University of North Carolina at Chapel Hill

[Airey89d] Airey, J. and F. P. Brooks Jr. (1989) Walkthrough - Exploring Virtual Worlds Second Annual Report University of North Carolina at Chapel Hill TR89-012

[Airey89] Airey, J. and M. Ouh-young (1989) Two Adaptive Techniques Let Progressive Radiosity Outperform the Traditional Radiosity Algorithm Department of Computer Science University of North Carolina at Chapel Hill TR89-020

[Airey89b] Airey, J., J. Rohlf and P. Rheingans 1989 The Virtual Lobby Video SIGGRAPH '89 Film and Video Show

[Airey89c] Airey, J. M. (1989) Solving Computer Graphics Problems Through Boolean Combinations of Polygons University of North Carolina Department of Computer Science TR89-031

[Airey90b] Airey, J. M., J. H. Rohlf and F. P. Brooks Jr. (1990). "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments." *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics).* **24**(2): 41-50

[Akeley89] Akeley, K. (1989). "The Silicon Graphics 4D/240GTX Superworkstation." *IEEE CG & A.* **9**(4): 239-246.

[Arvo87] Arvo, J. and D. Kirk. (1987). "Fast Ray Tracing by Ray Classification." *ACM Computer Graphics (Proceedings SIGGRAPH '87).* **21**(4): 55-64.

[Avis86] Avis, D., T. Gum and G. Toussaint. (1986). "Visibility between two edges of a simple polygon." *The Visual Computer.* **2**(6): 342-357.

[Baum89] Baum, D. R., H. E. Rushmeier and J. M. Winget. (1989). "Improving Radiosity Solutions Through the Use of Analytically Determined Form-Factors." *ACM Computer Graphics, (Proceedings of SIGGRAPH '89).* **23**(3): 325-334.

[Baum90] Baum, D. R. and J. M. Winget. (1990). "Real Time Radiosity Through Parallel Processing and Hardware Acceleration." *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics).* **24**(2): 67-76.

[Bentley80] Bentley, J. L., D. Haken and R. W. hon (1980) Statistics on VLSI Designs Carnegie-Mellon University, Pittsburgh PA, 1980 CMU-CS-80-111

[Brooks86] Brooks, F. P., Jr. (1986) Walkthrough- A Dynamic Graphics System for Simulating Virtual Buildings 1986 Workshop on Interactive Computer Graphics. University of North Carolina at Chapel Hill

[Brooks88] Brooks, F. P. B., Jr. (1988) First Annual Technical Report Walkthrough Project University of North Carolina at Chapel Hill TR88-035

[Chin89] Chin, N. and S. Feiner. (1989). "Near Real-Time Shadow Generation Using BSP Trees." *ACM Computer Graphics (Proceedings SIGGRAPH '89).* **23**(3): 99-106.

[Cohen88] Cohen, M. F., S. E. Chen, J. R. Wallace and D. P. Greenberg. (1988). "A progressive Refinement Approach to Fast Radiosity Image Generation." *Computer Graphics (Proceedings of SIGGRAPH '88).* **22**(4): 75-84.

[Cohen85] Cohen, M. F. and D. P. Greenberg. (1985). "A Radiosity Solution for Complex Environments." *Computer Graphics (Proceedings of SIGGRAPH '85).* **19**(3): 31-40.

[Crow77] Crow, F. C. (1977). "Shadow Algorithms for Computer Graphics." *Computer Graphics (Proceedings of SIGGRAPH '77).* **11**(2): 242-248.

[Delany88] Delany, H. C. (1988) Ray Tracing On A Connection Machine 1988 International Conference on Supercomputing St. Malo, France, July 4-8, 1988

[Denber86] Denber, M. and P. Turner. (1986). "A differential compiler for computer animation." *ACM Computer Graphics (Proceedings of SIGGRAPH '86).* **20**(4): 21-27.

[Deyo88] Deyo, R., J. A. Briggs and P. Doenges. (1988). "Getting Graphics in Gear: Graphics and Dynamics in Driving Simulation." *ACM Computer Graphics (Proceedings SIGGRAPH '88).* **22**(4): 317-326.

[Dobkin88] Dobkin, D., L. Guibas, J. Hershberger and J. Snoeyink. (1988). "An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon." *ACM Computer Graphics (Proceedings of SIGGRAPH '88).* **22**(4): 31-40.

[Fuchs83] Fuchs, H., G. D. Abram and E. D. Grant. (1983). "Near Real-Time Shaded Display of Rigid Objects." *ACM Computer Graphics, (Proceedings of SIGGRAPH '83).* **17**(3): 65-69.

[Fuchs85] Fuchs, H., J. Goldfeather, J. P. Hultquist, S. Spach, J. Austin, J. Brooks Frederick P. , J. Eyles and J. Poulton. (1985). "Fast Spheres, Textures, Transparencies, and Image Enhancements in Pixel-Planes." *Computer Graphics (Proceedings of SIGGRAPH '85).* **19**(3): 111-120.

[Garlick90] Garlick, B., D. Baum and J. Winget. (1990). "Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations." SIGGRAPH '90 Course Notes on Parallel Algorithms and Architectures for 3D Image Generation (Course #28). :

[Gharachorloo89] Gharachorloo, N., S. Gupta, R. F. Sproull and I. E. Sutherland. (1989). "A Characterization of Ten Rasterization Techniques." *ACM Computer Graphics (Proceedings SIGGRAPH '89).* **23**(3): 355-368.

[Glassner90] Glassner, A. (1990). Graphics Gems.

[Goral84] Goral, C. M., K. E. Torrance and D. P. Greenberg. (1984). "Modeling the Interaction of Light Between Diffuse Surfaces." *Computer Graphics (Proceedings of SIGGRAPH '84).* **18**(3): 213-222.

[Gries81] Gries, D. (1981). The Science of Programming. Texts and Monographs in Computer Science. Springer-Verlag.

[Guibas78] Guibas, L. J. and R. Sedgewick (1978) A Dichromatic Framework For Balanced Trees 19th Annual Symposium on Foundatins of Computer Science, IEEE, 1978

[Hall88] Hall, R. (1988). Illumination and Color in Computer Generated Imagery. Monographs in Visual Communicatino. Springer-Verlag.

[Hanrahan89] Hanrahan, P. (1989). Chapter 3. A survey of Ray-Surface Intersection Algorithms. An Introduction to Ray Tracing. Academic Press.

[Hubschman81] Hubschman, H. and S. W. Zucker. (1981). "Frame-to-Frame Coherence and the Hidden Surface Computation:
Constraints for a Convex World." *Computer Graphics (Proceedings of SIGGRAPH '81).* **15**(4): 45-54.

[Laidlaw86] Laidlaw, D., B. Trumbore and J. F. Hughes. (1986). "Constructive Solid Geometry for Polyhedral Objects." *ACM Computer Graphics (Proceedings SIGGRAPH '86).* **20**(4): 161-170.

[Mantyla83] Mantyla, M. and M. Tamminen. (1983). "Localized Set Operations for Solid Modeling." *CG&IP.* **17**(3): 279-288.

[McKenna87] McKenna, M. (1987). "Worst-case optimal hidden surface removal." *ACM Trans. on Graphics.* **6**(1): 19-28.

[Mehlhorn84] Mehlhorn, K. (1984). Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

[Molnar89] Molnar, S. and H. Fuchs. (1989). Chapter 18 Advanced Raster Graphics Architecture. Fundamentals of Computer Graphics.

[Naylor81] Naylor, B. F. (1981) A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes University of Texas at Dallas

[Nievergelt82] Nievergelt, J. and F. P. Preparata. (1982). "Plane-Sweep Algorithms for Intersection Geometric Figures." *CACM.* **25**(10):

[Nishita83] Nishita, T. and E. Nakamae (1983) Half-Tone Representation of 3-D Objects Illuminated by Area Sources or Polyhedron Sources Proceedings of The IEEE Computer Society's International Computer Conference and Applications Conference (COMPSAC)

[Nishita85] Nishita, T. and E. Nakamae. (1985). "Continous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection." *Computer Graphics (Proceedings of SIGGRAPH '85).* **19**(3): 23-30.

[O'Rourke87] O'Rourke, J. (1987). Art Gallery Theorems and Algorithms. International Series of Monographs on Computer Science. Oxford University Press.

[Ottman85] Ottman, T., P. Widmayer and D. Wood. (1985). "A Fast Algorithm for the Boolean Masking Problem." *CVGIP*. **30**: 249-268.

[Plantinga89] Plantinga, W. H., C. R. Dyer and W. B. Seales (1989) Real-time hidden-line elimination for a rotating polyhedral scene using the aspect representation University of Pittsburgh, 89-3

[Preperata85] Preparata, F. P. and M. I. Shamos. (1985). Computational Geometry. Texts and Monographs in Computer Science. Springer-Verlag.

[Prins90] Prins, Jan. (1990). Personal communication.

[Putnam86] Putnam, L. K. and P. A. Subrahmanyam. (1986). "Boolean Operations on n-Dimensional Objects." *IEEE CG&A*. **1986**(June): 43-51.

[Requicha85] Requicha, A. A. G. (1985). "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms." *Proceedings of the IEEE*. **73**(1): 30-44.

[Rogers85] Rogers, D. F. (1985). Procedural Elements for Computer Graphics. McGraw-Hill.

[Rohlf90] Rohlf, J. H. (1990). Personal communication.

[Samet90] Samat, Hanan. (1990). Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addision-Wesley.

[Schachter83] Schachter, B. J. (1983). Computer Image Generation. John WIley & Sons.

[Sedgewick88] Sedgewick, R. (1988). Algorithms. Computer Science. Addison-Wesley.

[Sequin85] Sequin, C. H. and P. R. Wensley. (1985). "Visible Feature Return at Object Resolution." *IEEE CG&A*. **5**(5): 37-50.

[Shelley82] Shelley, K. L. and D. P. Greenberg. (1982). "Patch Specification and Path Coherence." *Computer Graphics (Proceedings of SIGGRAPH '82)*. **16**(4): 157-166.

[Sutherland74b] Sutherland, I. E. and G. W. Hodgman. (1974). "Reentrant Polygon Clipping." *CACM*. **17**(1): 32-42.

[Sutherland74] Sutherland, I. E., R. F. Sproull and R. A. Schumacker. (1974). "A Characterization of Ten Hidden Surface Algorithms." *ACM Computing Surveys*. **6**(1): 1-55.

[Syzmanski85] Syzmanski, T. G. and C. J. V. Wyk. (1985). "GOALIE: A Space Efficient System for VLSI Artwork Analysis." *IEEE Design and Test*. **2**(3): 64-72.

[Thibault87] Thibault, W. C. and B. F. Naylor. (1987). "Set Operations on Polyhedra Using Binary Space Partitioning Trees." *ACM Computer Graphics (Proceedings SIGGRAPH '87)*. **21**(4): 153-162.

[Tichy82] Tichy, Walter F. (1982). Design, Implementation, and Evaluation of a Revision Control System. IEEE 6th Conference on Software Engineering, Toyko, Japan.

[Van Dam88] Van Dam, Andries, ed. "PHIGS+ Functional Description Revision 3.0", *Computer Graphics*.**22**(3): 125-218.

[Von Herzen87] Von Herzen, B. and A. H. Barr. (1987). "Accurate Triangulations of Deformed, Intersection Surfaces." *ACM Computer Graphics (Proceedings of SIGGRAPH '87)*. **21**(4): 103-110.

[Wallace89] Wallace, J. R., K. A. Elmquist and E. A. Haines. (1989). "A Ray Tracing Algorithm for Progressive Radiosity." *ACM Computer Graphics (Proceedings of SIGGRAPH '89)*. **23**(4): 315-324.

[Weiler81] Weiler, K. (1981). "Polygon Comparison using a Graph Representation." *ACM Computer Graphics (Proceedings of SIGGRAPH '81)*. **15**(4): 10-18.

Appendix A. Primary source code.

Tips on reading this source code:
No supporting source code appears here. That source code can be found in appendix C.
I try to follow a few self-imposed conventions. Names in all capitals are simple string substitutions (macros) performed by the C pre-processor. Names with a leading capital letter are global variables. Function names and local variables are all lower case.

The software is structured into a hierarchy of libraries. Each Walkthrough header (.h) file represents a software package. The header files from the lowest level packages are included first so that definitions from low level packages can be used in high level packages. Circular dependencies between the Walkthrough software packages do not exist.

Appendix A.1 partition() and select_split2().

```
/* Copyright 1990, John M. Airey, UNC CS Dept.  All Rights Reserved. */

/* usr include files */
#include <stdio.h>
#include <math.h>

/* walkthru include files representing separate software packages */
#include "../../include/error.h"
#include "../../include/base_type_io.h"
#include "../../include/mem.h"
#include "../../include/geom_data.h"
#include "../../include/hash.h"
#include "../../include/hierarchy.h"
#include "../../include/coord_index.h"
#include "../../include/cell.h"

/* library interface definition file */
#include "vgraph.h"


void
partition(fp,rootname,topinst)
FILE*           fp; /* used for .partition file and cell files */
char*           rootname;
Instance*       topinst;
{
   Node*                troot = alloc_tree_node((Node*) 0,ROOT);
   Node*                t = troot;
   NodeLink*            stack = (NodeLink*) 0;
   NodeLink*            cell_list= (NodeLink*) 0;
   char                 filename[81];
   int                  i;
   Face*                facep;
   double               width,depth,height;

   flatten_template(topinst->template);

   initialize_first_cell(t->cell= alloc_cell(topinst->template));

   /**/
   width  = t->cell->east  - t->cell->west;
```

```
    depth  = t->cell->north - t->cell->south;
    height = t->cell->sky   - t->cell->earth;

    t->cell->west  -= width;
    t->cell->east  += width;
    t->cell->south -= depth;
    t->cell->north += depth;
    t->cell->earth -= height;
    t->cell->sky   += height;


    if (Params.MaxDepth) /* only do this if we have to */
       index_build(topinst->template);

    if (select_split2(t)) push_tnode(&stack,t);
    else                  push_tnode(&cell_list,t);

    while (t = pop_tnode(&stack)){
       t->ng = alloc_tree_node(t,NGSIDE);
       t->nl = alloc_tree_node(t,NLSIDE);
       split_cell(t,t->ng,t->nl);

       write_node_ascii(fp,t);

       if (select_split2(t->ng)) push_tnode(&stack,t->ng);
       else                      push_tnode(&cell_list,t->ng);

       if (select_split2(t->nl)) push_tnode(&stack,t->nl);
       else                      push_tnode(&cell_list,t->nl);
    }
    if (fp) fclose(fp);

    while (t = pop_tnode(&cell_list)){

       /* compute portals for cell */
       t->cell->westpl =compute_portals(topinst->template->gd,
                                        t->cell,WEST,0);
       t->cell->eastpl =compute_portals(topinst->template->gd,
                                        t->cell,EAST,0);
       t->cell->southpl=compute_portals(topinst->template->gd,
                                        t->cell,SOUTH,0);
       t->cell->northpl=compute_portals(topinst->template->gd,
                                        t->cell,NORTH,0);
       t->cell->earthpl=compute_portals(topinst->template->gd,
                                        t->cell,EARTH,0);
       t->cell->skypl  =compute_portals(topinst->template->gd,
                                        t->cell,SKY,0);

       if (!(fp=fopen(sprintf(filename,"%s.%05d.cell",
             rootname,t->id),"w")))
          die("write_tree","could not open cell file",1);
       write_node_ascii(fp,t);
       write_cell_ascii(fp,t->cell,1);
       fclose(fp);
    }
}
```

```
/*
** split selection routine
*/
int
select_split2(t)
Node*   t;
{
   int            i,j,min,max;
   int            nl;              /* the number not less than sfp */
   int            ng;              /* the number not greater than sfp */
   double         sp;              /* the number split by sfp */
   double         extensionarea;   /* if split covered whole cell */
   double         eqarea;          /* area of faces co-planar to split */
   double         best_criteria_so_far = Params.MinPriority;
   double         criteria;
   IndexNode**    index;
   double         o,s,b,vb;
   double         oc = Params.OcclusionCoefficient;
   double         bc = Params.BalanceCoefficient/(t->cell->depth+1);
   double         sc = Params.SplitCoefficient/(t->cell->depth+1);
   FaceLink*      eql;
   FaceCellRelation fcr;

   /* check that we are within the limits
      before trying to find a splitter
   */
   if (t->cell->depth >= Params.MaxDepth ||
       t->cell->num_in < Params.MinFaces ||
       volume_of_cell(t->cell) < Params.MinVolume){
      return 0;
   }
   /* else within all limits so look for a splitter*/

   normalize_split_coefficients(&oc,&bc,&sc,&vbc);

   for (j = X; j <= Z; j++){

      switch (j){
         case X:
            index = IndexYZ;
            min = find_index(j,t->cell->west);
            if (index[min]->v <= t->cell->west) min++;
            max = find_index(j,t->cell->east);
            if (index[max]->v >= t->cell->east) max--;
            break;
         case Y:
            index = IndexZX;
            min = find_index(j,t->cell->south);
            if (index[min]->v <= t->cell->south) min++;
            max = find_index(j,t->cell->north);
            if (index[max]->v >= t->cell->north) max--;
            break;
         case Z:
            index = IndexXY;
            min = find_index(j,t->cell->earth);
            if (index[min]->v <= t->cell->earth) min++;
            max = find_index(j,t->cell->sky);
            if (index[max]->v >= t->cell->sky) max--;
            break;
        }

        extensionarea = cell_dim_area(j,t->cell);

        for (i = min; i <= max; i++){
           nl = index[i]->nl - index[max]->nl;
           ng = index[i]->ng - index[min]->ng;
           sp = index[i]->sp;
           eqarea = 0;
           for (eql = index[i]->eq; eql; eql = eql->next){
              fcr = set_face_cell_relation(eql->f,t->cell);
              if (!(fcr & TRIVIALLYOUTSIDE))
                 eqarea += clipped_area_of_face(t->cell->west,
                                                t->cell->east,
                                                t->cell->south,
                                                t->cell->north,
                                                t->cell->earth,
                                                t->cell->sky,
                                                eql->f);
           }

           o = eqarea/extensionarea;
           if      (ng == nl)      b = 1.0; /* also catches denom == 0.0 */
           else if (nl  > ng)      b = ((double) ng)/nl;
           else                    b = ((double) nl)/ng;
           s = 1.0 - sp/t->cell->template->num_f;

           criteria = (o == 0.0) ? 0.0 : oc*o +bc*b +sc*s;
           if (criteria > best_criteria_so_far){
              best_criteria_so_far = criteria;
              t->si = j;
              t->sv = index[i]->v;
           }
        }
    }

    return (t->si != NC);
}
```

Appendix A.2 sweep.h: the library header file

```
/* Copyright 1988, John M. Airey, UNC CS Dept.  All Rights Reserved. */

/********************* CPRE DEFINES ****************************/
/* left to right, top to bottom ordering. */
#define XQVERT_LESS(a,b)
\
((a)[XQx] < (b)[XQx] || ((a)[XQx] == (b)[XQx] && (a)[XQy] > (b)[XQy]))

#define XQVERT_GREATER(a,b)
\
((a)[XQx] > (b)[XQx] || ((a)[XQx] == (b)[XQx] && (a)[XQy] < (b)[XQy]))

#define INFINITY          (&InfinityFace)
#define MINUSINFINITY     (&MinusInfinityFace)
```

```
#define ENABLE           1
#define DISABLE          0

#define NEGNEG          1
#define NEGPOS          2
#define POSNEG          3
#define POSPOS          4

#define RADINFO         1

#define MAXOWNERS       16
/********************* END CPRE DEFINES ****************************/

/*********************    TYPES *********************************/
/* type declarations */
typedef struct  XQVert                    XQVert;
typedef struct XQOwnLink                  XQOwnLink;

typedef enum VertEdgeRelation             VertEdgeRelation;
typedef struct Yel                        Yel;
typedef struct YOwnLink                   YOwnLink;

typedef struct YelSave                    YelSave;

/* definitions */
struct XQOwnLink {
   Face*        own;    /* pointer to owning face */
   VertPtr      rv;     /* second vert of edge */
   int          s;      /* is edge in clockwise order */
   XQOwnLink*   next;   /* other owners */
};

struct XQVert {
   VertPtr      v;      /* pointer to vert in geom dbase */
   XQOwnLink*   owners; /* list of faces who have an edge strtng at v */
};

enum VertEdgeRelation   { ABOVE, BELOW, ONTHELINE };

struct YOwnLink {
   VertPtr      v1;
   VertPtr      v2;
   int          s;      /* v1 and v2 appear in opposite order in face
array */
   Face*        own;
   YOwnLink*    next;
};

struct Yel {
   YOwnLink*    owners;
   double       a,b,c;      /* b >= 0 for ABOVE, BELOW and ONTHELINE */
   int*         avec;       /* set counts for current above interval */
   int*         bvec;       /* set counts for current below interval */
   VertLink*    h;          /* points to the head of a list
                                 if avec evaluates to 0 and bvec
                                 evaluates to 1 */
   VertLink*    t;          /* points to the tail of a list
```

```
                                      if avec evaluates to 1 and bvec
                                      evaluates to 0.*/
   Yel*         u;          /* the edge that points to the head of the
                                      list when we point to the tail. */
   Yel*         d;          /* the edge that points to the tail of the
                                      list when we point to the head */
};

struct YelSave {
   Yel*         self;       /* pointer to self */
   Yel*         u;          /* edge on other side of  region */
   Yel*         d;          /* edge on other side of  region */
   VertLink*    h;          /* header of vert list for region */
   VertLink*    t;          /* tail of vert list for region */
   int          left;       /* left side of sweep line? */
   int          found;      /* am I valid */
   int          i;          /* index counterclockwise
                                      around cur xqvert */
   int          nown;       /* number valid entries in owners array */
   Face*        owners[MAXOWNERS];
};

/********************** END TYPES ******************************/

/*********************** EXTERNS ***************************/
/* functions */
extern void             xq_init();
extern void             xq_term();
extern XQVert*          xq_insert();
extern XQVert*          xq_min();

extern void             ytbl_init();
extern void             ytbl_term();
extern Yel*             ytbl_insert();
extern void             ytbl_delete();
extern Yel*             ytbl_succ();
extern Yel*             ytbl_pred();
extern Yel*             ytbl_findabove();
extern Yel*             ytbl_findbelow();
extern void             ytbl_fixbvecs();
extern VertEdgeRelation vert_edge_relation();
extern void             set_edge_coefficients();

extern void             ytbl_print();
extern void             ytbl_check_order();
extern void             ytbl_consistency_check();

extern FaceLink*        sweep();
extern int              sweep_detect();
extern VertLink*        triangulate_counter();
extern VertLink*        triangulate_clockwise();


extern XQVert    XQCurV;          /* access to min of XQueue */
extern int       XQx;             /* projection indices */
extern int       XQy;
extern int       XQz;
extern double   XQplaneq[];

extern int      BitVecSize;
extern int      NumTransitions;
extern Face     InfinityFace;
extern Face     MinusInfinityFace;
extern Yel**    YTable;
extern int      MaxInYTable;
extern int      NumInYTable;
extern int      NumTransitions;

/******************** END EXTERNS ************************/
/* end sweep.h */
```

Appendix A.3 Sweep.c: sweep() and transition()

```
/* Copyright 1988, John M. Airey, UNC CS Dept.  All Rights Reserved. */

/* usr include files */
#include <stdio.h>
#include <math.h>

/* walkthru include files */
#include "../../include/error.h"
#include "../../include/hash.h"
#include "../../include/mem.h"
#include "../../include/geom_data.h"
#include "../../include/hashedge.h"
#include "../../include/X11display.h"

/* library interface definition */
#include "sweep.h"

int     transition();
void    check_for_intersection();
int     construct_tri();
int     detect_tri();
static HASHequip*       Etable;

/*
*/
FaceLink*
sweep(pobj,fl,boolean_area_func,dbg)
GeomData*       pobj; /* where the results go */
FaceLink*       fl;
int             (*boolean_area_func)();
int             dbg;
{
   XQVert*      min;
   FaceLink*    rfl = (FaceLink*) 0;
   FaceLink*    cur;
   int          lastface = pobj->num_f;
   int          faces_so_far = lastface;
   int          i;

   Etable = hashedge_init();
   if (!fl) return rfl;

   xq_init(fl,fl->f->o);
   ytbl_init(fl);
   while (min = xq_min()) {
      NumTransitions++;
      (void) transition(pobj,min,boolean_area_func,construct_tri);
   }
   xq_term();
   ytbl_term();

   hashedge_term(Etable);
   Etable = (HASHequip*) 0;

   for (i = lastface; i < pobj->num_f; i++){
      ALLOCN(cur,FaceLink,1,"sweep");
      cur->f = &pobj->f[i];
      cur->next = rfl;
      rfl = cur;
   }
   return rfl;
}
/*
   This transition procedure is very similar to the transition procedure
   outlined in Kurt MelHorn's Multi-Dimensional Searching and
   Computational Geometry starting on page 160 except that it has
   been generalized in two ways. The first is that it triangulates
   the portion of the plane that satisfies some boolean combination of
   sets of polygons (Set operations on polygons).
   The second is that it has been written to deal
   with so called "degenerate input" or polygons that are not
   in "general position", i.e. vertices from one polygon are permitted
   to coincide with vertices or worse, edges, from  another polygon.

   The first generalization is relatively simple. Each region has
   a count of the number of polygons from each input set which cover the
   region. A function passed in as a parameter runs through this
   count and returns either true or false depending upon whether the
   region is "in" or "out" of the boolean set we are interested in.

   The second generalization is much harder from a programming
   standpoint.
   Theoretically it is not such a big deal, but it entails a formidable
   amount of detail as you will see if you try to understand the code.

   Given a particular vertex, we may have any combination of incoming
   and outgoing edges. Four bends, one end, an intersection, three
   starts,whatever. No simple case statement on the type of vertex as
   described in the Computational Geometry literature is possible.

   Here is an example of the type of difficulty encountered.
   The code in the published algorithms for the bend cases just
   specifies
   that we can simply "replace" (not insert the old and delete the new)
   the entering edge with the exiting edge.
   However if we have two bends going through the vert, they can
   act topologically as either two bends, one above the other or
   as an intersection vert in which case the exiting edges have to
   be exchanged. So just processing these two bends as two bends
   independently of each other could result in an incorrect Ytable.

   We need to generalize our view of the transition operation.
   I view the vertex as having three types of incident edges, those
   that end at the vertex, those that pass through the vertex and
   those that start at the vertex. We can classify the sectors
   surrounding
   the vert as to whether they satisfy the boolean function we are
   evaluating. If we move counterclockwise around the vertex we will
   move through alternating positive and negative sets of adjacent
   sectors. This function travels counterclockwise around the vertex
   performing the necessary operations on the YTable and
   processing the pairs of edges that bound a set of adjacent positive
   sectors. Depending upon whether the edges are to the left or right of
   the sweep line we either do processing equivalent to an end,
   one of the two  bends or a start.
   That is the generalization in a nutshell.

   Some Details.
   One might ask how do we travel counterclockwise around the vert.
   One possibility is to let the XQueue code associate a list of all
   edges that end at, extend through or begin at the vertex and
   keep them in a list sorted counterclockwise. This is sort of the
   natural way to go after reading a published description of
   a plane sweep algorithm since the description implies that we
   use information associated with the XQueue vertex to
   know which case transition should execute.

   However, there is a way to avoid this extra sorting.
   Since the ytable maintains the edges in sorted order we let it
   do the sorting for us. We can find the edge directly above the vert
   and use ytbl_succ to move down to the edge directly below the vert.
   As we move along, we delete
   from the YTable those edges who end at the vert.
   When we hit the edge below the vert,
   we reverse in the YTable all edges that extended through the vert
   and insert into the YTable all edges that begin at the vert.
   We then travel up from the lower edge to the edge above the vert
   using ytbl_pred. This is a counterclockwise traversal.
   Note that this means we are letting the YTable code determine what
   edges go through or end at vertex. The edges do not have to be known
   to the XQueue code. It still is unknown to me whether this invites
   floating point woes more than the first option does, but I have
   not experienced any problems.

*/
int /* return 1 if a triangle was detected */
transition(pobj,v,boolean_area_func,tri_func)
GeomData*       pobj;
XQVert*         v;
int             (*boolean_area_func)();
int             (*tri_func)();
{
   Yel                    *h,*l,*e,*succ,*prev_pos;
   int                     done = 0;
   YelSave                 pos,neg,prevneg,nextpos;
```

```
YOwnLink*           yown;
static YelSave      init={(Yel*)0,(Yel*)0,(Yel*)0,
                          (VertLink*)0,(VertLink*)0,0,0,0,0};
int                 going_down = 1;
int                 tran;
VertLink*           vl1;
VertLink*           vl2;
VertLink*           tvl;
VertLink*           tvlhold;
VertLink*           uppermaxrightvl;
VertLink*           lowermaxrightvl;
int                 rotindex = 0;
int                 tri = 0;
Face*               commonowners[MAXOWNERS];
int                 ncomown;
int                 i,j;

nextpos = prevneg = pos = neg = init;

h = ytbl_findabove(v);
l = ytbl_findbelow(v);

e = h;
succ = ytbl_succ(e);
while (!done){

   /* check if pos found previously on POSPOS transition */
   if (nextpos.found){
      pos = nextpos;
      nextpos = init;
   }

   while (!pos.found && !done){
      if (going_down) e = succ;
      else            e = ytbl_pred(e);

      if (e == l)  {
         insert_exiting_edges(v,l,h);
         going_down = 0;
         check_for_intersection(pobj,h,ytbl_succ(h),v);
         check_for_intersection(pobj,ytbl_pred(l),l,v);
      }
      else if (e == h) done = 1;
      else {
         tran = (*boolean_area_func)(e);
         if      ((tran==NEGPOS &&  going_down) ||
                  (tran==POSNEG && !going_down)){
            pos.self= e;
            pos.h = e->h;
            pos.t = e->t;
            pos.u = e->u;
            pos.d = e->d;
            pos.found = 1;
            pos.left = going_down;
            pos.i = rotindex++;
         }
         else if ((tran==POSNEG &&  going_down) ||
                    (tran==NEGPOS && !going_down)){
                if (prevneg.self){
                   ytbl_print(ErrFile);
                   die("transition","NEG,NEG error",1);
                }
                else {
                   prevneg.self = e;
                   prevneg.h = e->h;
                   prevneg.t = e->t;
                   prevneg.u = e->u;
                   prevneg.d = e->d;
                   prevneg.found = 1;
                   prevneg.left = going_down;
                   prevneg.i = rotindex++;
                }
             }
             else if (tran == POSPOS){

                if (prevneg.self){
                   ytbl_print(ErrFile);
                   die("transition","NEG,NEG error 2",1);
                }
                else {
                   prevneg.self = e;
                   prevneg.h = e->h;
                   prevneg.t = e->t;
                   prevneg.u = e->u;
                   prevneg.d = e->d;
                   prevneg.found = 1;
                   prevneg.left = going_down;
                   prevneg.i = rotindex;
                }
                pos.self= e;
                pos.h = e->h;
                pos.t = e->t;
                pos.u = e->u;
                pos.d = e->d;
                pos.found = 1;
                pos.left = going_down;
                pos.i = rotindex++;
             }
          }

          if (going_down) {
             succ = ytbl_succ(e);
             ytbl_delete(e);
          }
       }  /* end while pos not found and not done */

       while (!neg.found && !done){
          if (going_down) e = succ;
          else            e = ytbl_pred(e);

          if (e == l) {
             insert_exiting_edges(v,l,h);
             going_down = 0;
             check_for_intersection(pobj,h,ytbl_succ(h),v);
             check_for_intersection(pobj,ytbl_pred(l),l,v);
```

```
    }
    else if (e == h) done = 1;
    else {
       tran = (*boolean_area_func)(e);
       if       ((tran==NEGPOS &&  going_down) ||
                  (tran==POSNEG && !going_down)){
          ytbl_print(ErrFile);
          die("transition","POS,POS error",1);
       }
       else if ((tran==POSNEG &&  going_down) ||
                  (tran==NEGPOS && !going_down)){
          neg.self= e;
          neg.h = e->h;
          neg.t = e->t;
          neg.u = e->u;
          neg.d = e->d;
          neg.found = 1;
          neg.left = going_down;
          neg.i = rotindex++;
       }
       else if (tran == POSPOS){

          neg.self= e;
          neg.h = e->h;
          neg.t = e->t;
          neg.u = e->u;
          neg.d = e->d;
          neg.found = 1;
          neg.left = going_down;
          neg.i = rotindex;

          nextpos.self= e;
          nextpos.h = e->h;
          nextpos.t = e->t;
          nextpos.u = e->u;
          nextpos.d = e->d;
          nextpos.found = 1;
          nextpos.left = going_down;
          nextpos.i = rotindex++;
       }
    }

    if (going_down) {
       succ = ytbl_succ(e);
       ytbl_delete(e);
    }
} /* end while neg not found and not done */


/* try to use the neg found while looking for first pos */
if (!neg.found && prevneg.found)
   neg= prevneg; /* structure copy */

/* make sure we have both a pos and a neg */
if ((!neg.found && pos.found) || (neg.found && !pos.found)) {
   ytbl_print(ErrFile);
   die("transition","missing NEG or POS",1);
```

```
}

if (pos.found && neg.found){
   /* now we have a pos and neg pair */

   if (pos.left && neg.left){
      /* do end processing */
      if (pos.i < neg.i){
         /* triangulate the whole vert link list and free mem. */
         for (tvl = pos.h; tvl->next; tvl = tvlhold){
            tvlhold = tvl->next;
            if ((*tri_func)(pobj,v->v,tvl->v,tvl->next->v))
               tri = 1;
            freevl(tvl);
         }
         freevl(tvl);
      }
      else{
         pos.d->u = neg.u;
         neg.u->d = pos.d;

         vl1 = allocvl(v->v);

         /* triangulate upwards from v */
         if (neg.t->next)
            fprintf(ErrFile,"something is screwy\n");
         vl1->prev= triangulate_clockwise(pobj,neg.t,
                                          vl1,tri_func,&tri);
         vl1->prev->next = vl1;

         /* triangulate downwards from v */
         if (pos.h->prev)
            fprintf(ErrFile,"something is screwy2\n");
         vl1->next= triangulate_counter(pobj,pos.h,
                                        vl1,tri_func,&tri);
         vl1->next->prev = vl1;
      }
   }
   else if (pos.left && !neg.left){
      /* do bend processing  */
      vl1 = allocvl(v->v);

      neg.self->d = pos.d;
      pos.d->u = neg.self;
      neg.self->h = vl1;

      if (pos.h->prev) fprintf(ErrFile,"something is screwy3\n");
      vl1->next =
            triangulate_counter(pobj,pos.h,vl1,tri_func,&tri);
      vl1->next->prev = vl1;
   }
   else if (!pos.left && neg.left){
      /* do bend processing  */
      vl1 = allocvl(v->v);
      pos.self->u = neg.u;
      neg.u->d = pos.self;
      pos.self->t = vl1;
      if (neg.t->next) fprintf(ErrFile,"something is screwy4\n");
      vl1->prev = triangulate_clockwise(pobj,neg.t,
                                         vl1,tri_func,&tri);
      vl1->prev->next = vl1;
   }
   else /* if (!pos.left && !neg.left) */{
      vl1 = allocvl(v->v);

      /* do  start processing  */
      if (neg.i > pos.i) { /* case 1.1 in Melhorn */
         pos.self->t = neg.self->h = vl1;
         pos.self->u = neg.self;
         neg.self->d = pos.self;
      }
      else{ /* case 1.2 in Melhorn */
         vl2 = allocvl(v->v);

         for (prev_pos = ytbl_pred(pos.self);
              prev_pos->h == (VertLink*) 0;
              /* (*boolean_area_func)(prev_pos) != NEGPOS; */
              prev_pos = ytbl_pred(prev_pos)) /*NOP*/;

         /* fix the other entry links and region links now */
         neg.self->d = prev_pos->d;
         prev_pos->d->u = neg.self;
         pos.self->u = prev_pos;
         prev_pos->d = pos.self;

         pos.self->t = vl1;
         neg.self->h = vl2;

         /* find right most vertlink */
         lowermaxrightvl = prev_pos->h;
         for(tvl = prev_pos->h->next; tvl; tvl = tvl->next)
            if (XQVERT_GREATER(tvl->v,lowermaxrightvl->v))
               lowermaxrightvl = tvl;

         uppermaxrightvl = allocvl(lowermaxrightvl->v);
         uppermaxrightvl->prev = lowermaxrightvl->prev;
         if (lowermaxrightvl == prev_pos->h)
            prev_pos->h = uppermaxrightvl;
         else{
            lowermaxrightvl->prev->next = uppermaxrightvl;
            uppermaxrightvl->prev = lowermaxrightvl->prev;
         }

         /* now triangulate up from uppermaxrightvl */
         vl1->prev = triangulate_clockwise(pobj,uppermaxrightvl,
                                            vl1,tri_func,&tri);
         vl1->prev->next = vl1;

         /* now triangulate down from lowermaxrightvl */
         vl2->next = triangulate_counter(pobj,lowermaxrightvl,
                                          vl2,tri_func,&tri);
         vl2->next->prev = vl2;
      }
   } /* end start case */
```

```
      } /* end pos found and neg found */
      pos = neg = init;
   } /* end while not done */
   return tri;
}
```

Appendix A.4 Visible.c: C code to implement sampling methods

```
/* Copyright 1990, John M. Airey, UNC CS Dept.  All Rights Reserved. */

/* usr include files */
#include <stdio.h>
#include <math.h>

/* included walkthru header files for lower level software libraries */
#include "../../include/error.h"
#include "../../include/mem.h"
#include "../../include/geom_data.h"
#include "../../include/hash.h"
#include "../../include/hierarchy.h"
#include "../../include/coord_index.h"
#include "../../include/cell.h"

/* library interface definition */
#include "vgraph.h"

extern char*    sprintf();
extern double   drand48();


/* definition of declared externs in vgraph.h */
double  PortalPatchSize = 96.; /* 2 foot sq. patches on 1/4 inch scale
*/
double  NumPortalRays = 2700.; /*comparable to 30x30 hemi-cube*/
int     RayDistribution;
int     HemisphereSample = 1;
int     RayTries = 7;

/*
This routine is passed a cell data structure. This contains information
that allows access to all polygons in the database and all portals for
the cell.

For each of the six faces of the cell, it samples the visible polygons
from the portals that comprise the open parts of that cell side.

Lastly, it marks faces that intersect the boundary of the cell as
visible. Whenever possible, simple tests are put at the beginning of
logical or (||) tests because C will not evaluate the remainder of the
conditional expression if it is
not necessary. Thus the FACEINSIDE macro is placed before the more
expensive clip_face_to_box function call.
*/
void
mark_visible_faces(cell)
Cell*           cell;
```

```
{
   FaceLink*     fl;
   int           i;
   Vert          wb[MAXVERTSINFACE];
   Matrix        pmat;

   /* rotation matrix to transform 0,0,1 to -1,0,0 */
   pmat[0][0] =  0.0; pmat[0][1] =  0.0; pmat[0][2] =  1.0;
   pmat[1][0] =  0.0; pmat[1][1] =  1.0; pmat[1][2] =  0.0;
   pmat[2][0] = -1.0; pmat[2][1] =  0.0; pmat[2][2] =  0.0;

   for (fl = cell->westpl; fl; fl = fl->next){
      fl->f->flags |= REVERSED;
      sample_portal_visibility(cell,fl->f,pmat);
   }

   /* rotation matrix to transform 0,0,1 to 1,0,0 */
   pmat[0][0] =  0.0; pmat[0][1] =  1.0; pmat[0][2] =  0.0;
   pmat[1][0] =  0.0; pmat[1][1] =  0.0; pmat[1][2] =  1.0;
   pmat[2][0] =  1.0; pmat[2][1] =  0.0; pmat[2][2] =  0.0;

   for (fl = cell->eastpl; fl; fl = fl->next){
      sample_portal_visibility(cell,fl->f,pmat);
   }

   /* rotation matrix to transform 0,0,1 to 0,-1,0 */
   pmat[0][0] =  1.0; pmat[0][1] =  0.0; pmat[0][2] =  0.0;
   pmat[1][0] =  0.0; pmat[1][1] =  0.0; pmat[1][2] =  1.0;
   pmat[2][0] =  0.0; pmat[2][1] = -1.0; pmat[2][2] =  0.0;

   for (fl = cell->southpl; fl; fl = fl->next){
      fl->f->flags |= REVERSED;
      sample_portal_visibility(cell,fl->f,pmat);
   }

   /* rotation matrix to transform 0,0,1 to 0,1,0 */
   pmat[0][0] =  0.0; pmat[0][1] =  0.0; pmat[0][2] =  1.0;
   pmat[1][0] =  1.0; pmat[1][1] =  0.0; pmat[1][2] =  0.0;
   pmat[2][0] =  0.0; pmat[2][1] =  1.0; pmat[2][2] =  0.0;

   for (fl = cell->northpl; fl; fl = fl->next){
      sample_portal_visibility(cell,fl->f,pmat);
   }

   /* rotation matrix to transform 0,0,1 to 0,0,-1 */
   pmat[0][0] =  0.0; pmat[0][1] =  1.0; pmat[0][2] =  0.0;
   pmat[1][0] =  1.0; pmat[1][1] =  0.0; pmat[1][2] =  0.0;
   pmat[2][0] =  0.0; pmat[2][1] =  0.0; pmat[2][2] = -1.0;

   for (fl = cell->earthpl; fl; fl = fl->next){
      fl->f->flags |= REVERSED;
      sample_portal_visibility(cell,fl->f,pmat);
   }

   /* rotation matrix to transform 0,0,1 to 0,0,1 */
   pmat[0][0] =  1.0; pmat[0][1] =  0.0; pmat[0][2] =  0.0;
   pmat[1][0] =  0.0; pmat[1][1] =  1.0; pmat[1][2] =  0.0;
```

```
    pmat[2][0] =  0.0; pmat[2][1] =  0.0; pmat[2][2] =  1.0;

    for (fl = cell->skypl; fl; fl = fl->next){
       sample_portal_visibility(cell,fl->f,pmat);
    }

    cell->num_vis = 0;
    for (i = 0; i < cell->template->num_f; i++){
       if (i != cell->template->faces[i]->id)
          die("mark_visible_faces","bad assumption",1);
       if (cell->fcr[i] & VISIBLE) cell->num_vis++;
       else if (FACEINSIDE(cell->fcr[i]) ||
                clip_face_to_box(cell->west,cell->east,
                                 cell->south,cell->north,
                                 cell->earth,cell->sky,
                                 cell->template->faces[i],wb)){
          cell->fcr[i] |= VISIBLE;
          cell->num_vis++;
       }
    }
}
/*
This routine is passed a cell data structure which allows it to access
all polygons in the database, a Face data structure which defines a
portal, and a matrix which can be used to transform a canonical pattern
of samples constructed for a portal lying in the x,y plane and facing
upwards, to the actual orientation of the portal.

The routine generates a grid of sample points on the face of the portal.
The implementation uses a little index magic to allow the same code to
work regardless of the orientation of the portal.

Then the sample points on the portal are constructed. If the portal does
not completely cover the sample square, the center of the fragment
inside the square is used as an initial point instead of the center of
the square. Depending upon the value of HemisphereSample, either
hemisphere_sample or fire_rays_at faces is called to actually compute
the samples.
*/
void
sample_portal_visibility(cell,f,pmat)
Cell*   cell;
Face*   f;
Matrix  pmat;
{
   int          i,j,n;
   Extent       clipbox;
   double       area;
   Vec3         center;
   Vert         wb[MAXVERTSINFACE];
   int          x1,x2,x0;
   int          x1dim,x2dim;
   double       dx1,dx2;

   /* switch on the orientation of the face to set variables to
      constants so that the same code to generate a grid on the portal
      will work regardless of the portal orientation
   */
```

```
    switch (f->o){
       case X: x1 = Y; x2 = Z; x0 = X; break;
       case Y: x1 = Z; x2 = X; x0 = Y; break;
       case Z: x1 = X; x2 = Y; x0 = Z; break;
       default:
          fprintf(ErrFile,"f->o = %d, f->planeq = %lg,%lg,%lg,%lg\n",
             f->o,f->planeq[A],f->planeq[B],f->planeq[C],f->planeq[D]);
          die("sample_portal_visibility","portal skew",1);
    }
    x1dim=rint((f->ex[MAXEX(x1)] - f->ex[MINEX(x1)])/PortalPatchSize);
    x2dim=rint((f->ex[MAXEX(x2)] - f->ex[MINEX(x2)])/PortalPatchSize);
    if (!x1dim) x1dim = 1;
    if (!x2dim) x2dim = 1;
    dx1 = (f->ex[MAXEX(x1)] - f->ex[MINEX(x1)])/x1dim;
    dx2 = (f->ex[MAXEX(x2)] - f->ex[MINEX(x2)])/x2dim;

    clipbox[MINEX(x0)] = f->ex[MINEX(x0)];
    clipbox[MAXEX(x0)] = f->ex[MAXEX(x0)];

    clipbox[MINEX(x1)] = f->ex[MINEX(x1)];
    clipbox[MAXEX(x1)] = clipbox[MINEX(x1)] + dx1;
    for (i = 0; i < x1dim; i++){
       clipbox[MINEX(x2)] = f->ex[MINEX(x2)];
       clipbox[MAXEX(x2)] = clipbox[MINEX(x2)] + dx2;
       for (j = 0; j < x2dim; j++){
          if ((n = clip_face_to_box(clipbox[MINX],clipbox[MAXX],
                                    clipbox[MINY],clipbox[MAXY],
                                    clipbox[MINZ],clipbox[MAXZ],
                                    f,wb)) > 2){

             area = center_and_area(n,wb,x1,x2,1.,center);
             pmat[3][0] = center[0];
             pmat[3][1] = center[1];
             pmat[3][2] = center[2];
             if (HemisphereSample) hemisphere_sample(cell,pmat);
             else                  fire_rays_at_faces(cell,pmat);
          }
          clipbox[MINEX(x2)] += dx2;
          clipbox[MAXEX(x2)] += dx2;
       }
       clipbox[MINEX(x1)] += dx1;
       clipbox[MAXEX(x1)] += dx1;
    }
}

/*
Tables used by ray firing mechanism. The function
init_ray_intersection_tables
must be called once before visibility calculations begin.
*/

static IndexNode** Index[6];
static int          End[6];
static int          Incr[6];
static int          X1[6];
static int          X2[6];
static double       Rand[17];
```

```
/*
*/
void
init_ray_intersection_tables()
{
   int i;

   Index[MINX] = IndexYZ; Index[MAXX] = IndexYZ;
   Index[MINY] = IndexZX; Index[MAXY] = IndexZX;
   Index[MINZ] = IndexXY; Index[MAXZ] = IndexXY;

   X1[MINX] = Y; X1[MAXX] = Y;
   X1[MINY] = Z; X1[MAXY] = Z;
   X1[MINZ] = X; X1[MAXZ] = X;

   X2[MINX] = Z; X2[MAXX] = Z;
   X2[MINY] = X; X2[MAXY] = X;
   X2[MINZ] = Y; X2[MAXZ] = Y;

   End[MINX] = -1; End[MAXX] = NumYZ;
   End[MINY] = -1; End[MAXY] = NumZX;
   End[MINZ] = -1; End[MAXZ] = NumXY;

   Incr[MINX] = -1; Incr[MAXX] = 1;
   Incr[MINY] = -1; Incr[MAXY] = 1;
   Incr[MINZ] = -1; Incr[MAXZ] = 1;

   for (i=0; i< 17; i++) Rand[i] = drand48();
   for (i=0; i< 17; i++) fprintf(ErrFile,"Rand[i] = %g\n",Rand[i]);
}
/*
```

This routine receives a cell data structure which allows it to access all polygons in the model and a matrix which defines the position and orientation of the starting point of a hemisphere of rays. The routine generates a canonical distribution of rays for a portal centered at the origin in the x,y plane and facing upwards to the positive z axis. The portal_mat is used to transform this canonical distribution of rays to the actual position and orientation of the portal.

A data structure which groups polygons that lie in the same axial plane together and sorts parallel planes is used to accelerate ray intersections with axial polygons. A BSP tree is used to do mop up work on the (hopefully) small number of skew polygons. The IndexXY,IndexYZ, and IndexZX arrays contain the sorted planes. Each entry is a structure which contains a few stats used for other purposes and a list of polygons which satisfy the plane equation of the entry. That list is accessed through the 'eq' field.

The order of intersections for each of the three set of sorted parallel planes is evidently the sorted order of the list or the reverse sorted order. To compute the intersections in order for three three sets of axial planes, a merge is computed using a simple three element priority queue. A set of global index lists is used to help manage all the different combinations of positive, negative, x, y or z that arise in tracing the ray through the data structure.

Once a plane ray intersection point is determined. The list of polygons that lies in that plane is searched to see if any of them contain the hit point. Note that the feature of C whereby the remainder of a logical OR operation, (||) is not evaluated if an early expression evaluates true is taken advantage of to increase efficiency. Faces (polygons) are tagged with a RECTANGLE bit. If the polygon is a rectangle (a common occurence in architectural databases), a simple point inclusion test is performed before a more involved function call is made for general convex polygons.

Once the closest axial polygon is determined, the BSP tree is checked for a closer intersection.

```
*/
void
hemisphere_sample(cell,portal_mat)
Cell*   cell;            /* cell owning portal */
Matrix  portal_mat;      /* matrix defining center and orientation of
portal */
{
   FaceLink*    fl;
   Face*        close_face;
   Face*        skew_face;
   Vec3         od,c,d,hit;
   double       numr,numt,r,t,radius,theta,offset;
   int          x1,x2;
   int          q1,q2,q3; /* micro priority queue (heap) indices */
   double       parm[6];
   int          start[6];
   int          s[6];
   static int   rand = 0;


   numr = rint(sqrt(NumPortalRays/2.));
   numt = 2*numr;

   c[X] = portal_mat[3][0];
   c[Y] = portal_mat[3][1];
   c[Z] = portal_mat[3][2];

   start[MINX] = start[MAXX] = find_index(X,c[X]);
   if (start[MINX] != -1){
      if (IndexYZ[start[MINX]]->v >= c[X]) start[MINX]--;
      if (IndexYZ[start[MAXX]]->v <= c[X]) start[MAXX]++;
   }
   else start[MAXX] = NumYZ;
   start[MINY] = start[MAXY] = find_index(Y,c[Y]);
   if (start[MINY] != -1){
      if (IndexZX[start[MINY]]->v >= c[Y]) start[MINY]--;
      if (IndexZX[start[MAXY]]->v <= c[Y]) start[MAXY]++;
   }
   else start[MAXY] = NumZX;
   start[MINZ] = start[MAXZ] = find_index(Z,c[Z]);
   if (start[MINZ] != -1){
      if (IndexXY[start[MINZ]]->v >= c[Z]) start[MINZ]--;
      if (IndexXY[start[MAXZ]]->v <= c[Z]) start[MAXZ]++;
   }
   else start[MAXZ] = NumXY;
```

```
for (r = 0; r < numr; r++){

   switch (RayDistribution){
      case 0:
         /* weight forward more heavily */
         radius = (2.*r+1.)/(2.*numr);
         break;
      case 1:
         radius = sqrt((2.*r+1.)/(2.*numr));
         break;
      default:
         die("hemisphere_sample","unknown ray distribution",1);
         break;
   }
   /* radius = sqrt((2.*r+1.)/(2.*numr)); */

   offset = Rand[rand]*(2.0*M_PI); /* random rot. of hemi-sphere */
   rand = (rand+1)%17;

   for (t = 0; t < numt; t++){

      theta =        Rand[rand] * t * (2.0*M_PI)/numt
            + (1.0-Rand[rand])* (t+1.) * (2.0*M_PI)/numt;

      rand = (rand+1)%17;

      /* compute Original Direction from radius and theta */
      od[X] = radius*cos(theta+offset);
      od[Y] = radius*sin(theta+offset);
      od[Z] = sqrt(1.- radius*radius);

      /* transform direction to portal orientation */
      d[X] = portal_mat[0][0]*od[X] +
             portal_mat[1][0]*od[Y] +
             portal_mat[2][0]*od[Z];
      d[Y] = portal_mat[0][1]*od[X] +
             portal_mat[1][1]*od[Y] +
             portal_mat[2][1]*od[Z];
      d[Z] = portal_mat[0][2]*od[X] +
             portal_mat[1][2]*od[Y] +
             portal_mat[2][2]*od[Z];

      /* set up three element priority queue */

      if (d[X] < 0.) q1 = MINX;
      else           q1 = MAXX;
      s[q1] = start[q1];
      if (fabs(d[X]) < FEPS || s[q1] == End[q1]) parm[q1] = HUGE;
      else parm[q1] = (Index[q1][s[q1]]->v - c[X]) / d[X];

      if (d[Y] < 0.) q2 = MINY;
      else           q2 = MAXY;
      s[q2] = start[q2];
      if (fabs(d[Y]) < FEPS || s[q2] == End[q2]) parm[q2] = HUGE;
      else parm[q2] = (Index[q2][s[q2]]->v - c[Y]) / d[Y];
```

```
/* insert element in micro priority q */
if (parm[q2] < parm[q1]) { int _t = q1; q1 = q2; q2 = _t; }

if (d[Z] < 0.) q3 = MINZ;
else           q3 = MAXZ;
s[q3] = start[q3];
if (fabs(d[Z]) < FEPS || s[q3] == End[q3]) parm[q3] = HUGE;
else parm[q3] = (Index[q3][s[q3]]->v - c[Z]) / d[Z];

/* insert element in micro priority q */
if (parm[q3] < parm[q1]) { int _t = q1; q1 = q3; q3 = _t; }

close_face = (Face*) 0;
while (!close_face && s[q1] != End[q1]){
   x1 = X1[q1];
   x2 = X2[q1];

   hit[X] = c[X] + d[X]*parm[q1];
   hit[Y] = c[Y] + d[Y]*parm[q1];
   hit[Z] = c[Z] + d[Z]*parm[q1];

   for (fl = Index[q1][s[q1]]->eq; fl; fl = fl->next){
      if (((fl->f->flags&RECTANGLE) &&
           hit[x1] >= fl->f->ex[MINEX(x1)] &&
           hit[x1] <= fl->f->ex[MAXEX(x1)]
           &&
           hit[x2] >= fl->f->ex[MINEX(x2)] &&
           hit[x2] <= fl->f->ex[MAXEX(x2)])
          ||
           (!(fl->f->flags&RECTANGLE) &&
            point_in_face(hit, fl->f))) {
         close_face = fl->f;
         break;
      }
   }

   if (!close_face){
      /* Increment Index */
      s[q1] += Incr[q1];

      /* reset parm value */
      if (s[q1] != End[q1])
         parm[q1] = (Index[q1][s[q1]]->v - c[q1>>1])
                    /
                    d[q1>>1];
      else
         parm[q1] = HUGE;

      /* perform down heap */
      if (parm[q2] < parm[q3]){
         if (parm[q2] < parm[q1]) { int t= q1; q1= q2; q2= t;}
      }
      else {
         if (parm[q3] < parm[q1]) { int t= q1; q1= q3; q3= t;}
      }
   }
} /* end while */
```

```
            /* check skew faces for closer intersection */
            /*
            if (skew_face = naive_intersect(SkewList,p,d,&near_t))
               near_face = skew_face;
            */
            if (skew_face = skew_intersect(SkewTree,
                           HUGE,POINTSON,c,d,&parm[q1])) {
               close_face = skew_face;
               hit[X] = c[X] + d[X]*parm[q1];
               hit[Y] = c[Y] + d[Y]*parm[q1];
               hit[Z] = c[Z] + d[Z]*parm[q1];
            }


            if (close_face){

               if (d[X]*close_face->planeq[A] +
                   d[Y]*close_face->planeq[B] +
                   d[Z]*close_face->planeq[C] < 0.){

                   /* making a dangerous assumption for efficiency's sake:
                      assuming the face id corresponds to the index in
                      the face cell relation table for the cell.
                      if a bug arises, make sure this is not the culprit.
                   */
                  cell->fcr[close_face->id] |= VISIBLE;
               }
               else{ /* hmmm, backside of face is being hit */
                  /*fprintf(ErrFile,"hit backside of face %d\n",
                              close_face->id);*/
                  cell->fcr[close_face->id] |= VISIBLE;
               }
            }
            else
               /* fprintf(ErrFile,"Rays in Space....\n")*/ ;
         }
      }
   }
   /*
   This routine is very similar to the previous routine,
   hemisphere_sample(). It differs primarily in the way in which rays are
   constructed. Here, up to RayTries rays are fired at every polygon. A
   random number is computed for each vertex, the random numbers are
   normalized so that they sum to one, and a linear combination of the
   vertices, using the random numbers as weights is used to compute a point
   on the polygon. Note there is some monkeying with a floating point
   epsilon, FEPS which I believe is defined as 1e-6, to avoid floating
   point error that would crash the program.
   */
   void
   fire_rays_at_faces(cell,portal_mat)
   Cell*   cell;           /* cell owning portal */
   Matrix  portal_mat;     /* matrix defining center and orientation of
   portal */
   {
      FaceLink*     fl;
      Face*         close_face;
```

```
Face*         skew_face;
Face*         f;
Vec3          c,d,hit,fc;
double        mag;
int           x1,x2;
int           q1,q2,q3; /* micro priority queue (heap) indices */
int           i,j,k;
VertPtr       v;
double        parm[6];
double        rand[MAXVERTSINFACE];
int           start[6];
int           s[6];


c[X] = portal_mat[3][0];
c[Y] = portal_mat[3][1];
c[Z] = portal_mat[3][2];

start[MINX] = start[MAXX] = find_index(X,c[X]);
if (start[MINX] != -1){
   if (IndexYZ[start[MINX]]->v >= c[X]) start[MINX]--;
   if (IndexYZ[start[MAXX]]->v <= c[X]) start[MAXX]++;
}
else start[MAXX] = NumYZ;
start[MINY] = start[MAXY] = find_index(Y,c[Y]);
if (start[MINY] != -1){
   if (IndexZX[start[MINY]]->v >= c[Y]) start[MINY]--;
   if (IndexZX[start[MAXY]]->v <= c[Y]) start[MAXY]++;
}
else start[MAXY] = NumZX;
start[MINZ] = start[MAXZ] = find_index(Z,c[Z]);
if (start[MINZ] != -1){
   if (IndexXY[start[MINZ]]->v >= c[Z]) start[MINZ]--;
   if (IndexXY[start[MAXZ]]->v <= c[Z]) start[MAXZ]++;
}
else start[MAXZ] = NumXY;

FOREACH_FACE_IN_TEMPLATE(cell->template,f,i){
   if (cell->fcr[f->id]&VISIBLE) continue;

   /* To avoid being classed as visible at least RayTries rays to
      random points on f should be stopped by some other polygon */
   close_face = (Face*) 0;
   for (k = 0; k < RayTries && f != close_face; k++){
      close_face = (Face*) 0;

      rand[f->n] = 0.0;
      FOREACH_VERT_IN_FACE(f,v,j){
         rand[j] = drand48();
         rand[f->n] += rand[j];
      }
      FOREACH_VERT_IN_FACE(f,v,j) rand[j] /= rand[f->n];
      /* compute random point on f */
      fc[X] = fc[Y] = fc[Z] = 0.0;
      FOREACH_VERT_IN_FACE(f,v,j){
         fc[X] += rand[j]*v[X];
         fc[Y] += rand[j]*v[Y];
   fc[Z] += rand[j]*v[Z];
}

d[X] = fc[X] - c[X]; d[Y] = fc[Y] - c[Y]; d[Z] = fc[Z] - c[Z];
if ((mag = sqrt(d[X]*d[X] + d[Y]*d[Y] + d[Z]*d[Z])) > FEPS) {
   d[X] /= mag; d[Y] /= mag; d[Z] /= mag;
}
else d[X] = d[Y] = d[Z] = 0.0;

if (d[X]*portal_mat[2][X] +
    d[Y]*portal_mat[2][Y] +
    d[Z]*portal_mat[2][Z] > FEPS){

   /* now check if any other face lies between the two. */

   /* set up three element priority queue */
   if (d[X] < 0.) q1 = MINX;
   else           q1 = MAXX;
   s[q1] = start[q1];
   if (fabs(d[X]) < FEPS || s[q1] == End[q1]) parm[q1] = HUGE;
   else parm[q1] = (Index[q1][s[q1]]->v - c[X]) / d[X];

   if (d[Y] < 0.) q2 = MINY;
   else           q2 = MAXY;
   s[q2] = start[q2];
   if (fabs(d[Y]) < FEPS || s[q2] == End[q2]) parm[q2] = HUGE;
   else parm[q2] = (Index[q2][s[q2]]->v - c[Y]) / d[Y];

   /* insert element in micro priority q */
   if (parm[q2] < parm[q1]) { int _t = q1; q1 = q2; q2 = _t; }

   if (d[Z] < 0.) q3 = MINZ;
   else           q3 = MAXZ;
   s[q3] = start[q3];
   if (fabs(d[Z]) < FEPS || s[q3] == End[q3]) parm[q3] = HUGE;
   else parm[q3] = (Index[q3][s[q3]]->v - c[Z]) / d[Z];

   /* insert element in micro priority q */
   if (parm[q3] < parm[q1]) { int _t = q1; q1 = q3; q3 = _t; }

   while (!close_face && s[q1] != End[q1]){
      x1 = X1[q1];
      x2 = X2[q1];

      hit[X] = c[X] + d[X]*parm[q1];
      hit[Y] = c[Y] + d[Y]*parm[q1];
      hit[Z] = c[Z] + d[Z]*parm[q1];

      for (fl = Index[q1][s[q1]]->eq; fl; fl = fl->next){
         if (((fl->f->flags&RECTANGLE) &&
               hit[x1] >= fl->f->ex[MINEX(x1)] &&
               hit[x1] <= fl->f->ex[MAXEX(x1)]
               &&
               hit[x2] >= fl->f->ex[MINEX(x2)] &&
               hit[x2] <= fl->f->ex[MAXEX(x2)])
              ||
               (!(fl->f->flags&RECTANGLE) &&
                point_in_face(hit,fl->f))){
```

```
            close_face = fl->f;
            break;
         }
      }

      if (!close_face){
         /* Increment Index */
         s[q1] += Incr[q1];

         /* reset parm value */
         if (s[q1] != End[q1])
            parm[q1] = (Index[q1][s[q1]]->v - c[q1>>1])
                          / d[q1>>1];
         else
            parm[q1] = HUGE;

         /* perform down heap */
         if (parm[q2] < parm[q3]){
            if (parm[q2] < parm[q1]) {
               int _t = q1; q1 = q2; q2 = _t;
            }
         }
         else {
            if (parm[q3] < parm[q1]) {
               int _t = q1; q1 = q3; q3 = _t;
            }
         }
      }
   } /* end while */

   /* check skew faces for closer intersection */
   /*
   if (skew_face = naive_intersect(SkewList,p,d,&near_t))
      near_face = skew_face;
   */
   if (skew_face=skew_intersect(SkewTree,HUGE,
                                 POINTSON,c,d,&parm[q1])){
      close_face = skew_face;
      hit[X] = c[X] + d[X]*parm[q1];
      hit[Y] = c[Y] + d[Y]*parm[q1];
      hit[Z] = c[Z] + d[Z]*parm[q1];
   }


   /* if close face is f, then f is visible. */
   if (close_face == f){
       /* making a dangerous assumption for efficiency's sake:
          assuming the face id corresponds to the index in
          the face cell relation table for the cell.
          if a bug arises, make sure this is not the culprit.
       */
      cell->fcr[close_face->id] |= VISIBLE;
   }
   else if (!close_face)
      fprintf(ErrFile,"fire_rays_at_faces: no face hit\n");
   else {
      /* if we hit somebody else, mark him visible */
```

```
                    cell->fcr[close_face->id] |= VISIBLE;
                 }
              }
              else{
                 /* we could break out of the for loop if we knew all of the
                    polygon was behind the portal */;
              }
           }
        }
     }
```

Appendix A.5 Occlusion.c: C code to implement an over-estimation method

```
/* Copyright 1990, John M. Airey, UNC CS Dept.  All Rights Reserved. */

/* usr include files */
#include <stdio.h>
#include <math.h>

/* walkthru include files */
#include "../../include/error.h"
#include "../../include/base_type_io.h"
#include "../../include/mem.h"
#include "../../include/geom_data.h"
#include "../../include/hash.h"
#include "../../include/hierarchy.h"
#include "../../include/coord_index.h"
#include "../../include/cell.h"

/* library interface definition file */
#include "vgraph.h"

extern char*    sprintf();

static OcclStackEl*      OcclS = (OcclStackEl*) 0;
static int               OcclSNum    = 0;
static int               OcclSMaxNum = 0;

/*
The objective of this function is to identify all faces in obj that
cannot be seen from any viewpoint inside the cell. Because any view from
inside the cell must look through the boundary, it suffices to identify
faces that cannot be seen from the "free" points on the boundary of the
cell. A point on the boundary of the cell is free if it is not contained
in some face. We call the regions on the boundary of the cell that are
not occupied by faces the "portals". Since the portals are not
represented directly in the model, we have to compute them.

The portals are computed with a plane sweep boolean area algorithm.

Basic Algorithm:
Once the portals have been computed we consider each of the faces in the
object in turn. If it has any portion inside the cell or on the boundary
then it is immediately classified visible.
```

```
If it is definitely outside then we try to establish that some occluding
set of faces exists for each portal. If such a set exists for each
portal we classify the face as not visible.

Enhanced Algorithm:
Similar but rather than consider each of the faces of the object totally
independently of each other we will consider groups of them at once. The
tree of cells is a convenient structure to use for this.

Essentially we initialize a stack with the root.
We then pull an element of this stack and if it is a cell we run the
occlusion operation on the cell itself. If it is classified as occluded,
all the faces inside it are classified as occluded.If it is visible, it
is split into its child cells and all faces that are SPLIT by that
division. The child cells and the SPLIT faces are pushed onto the stack.
If the cell is a leaf cell then all the faces inside it are put into the
stack.

The computation continues until the stack is empty. A count is kept of
the number of faces that were classified occluded as members of a group
and the number of faces classified as occluded individually. This can be
used to assess the success of the enhanced algorithm.
*/
void
mark_occluded_faces(pdata,cell,root)
GeomData*        pdata;  /* the object that holds the portals */
Cell*            cell;
Node*   root;
{
   OcclStackEl*          cand;
   Vert                  wb[MAXVERTSINFACE];/* a Work Buffer */
   int                   i,num_individual_occ = 0,num_group_occ = 0;
   FaceCellRelation      fcr;
   Face*                 facep;

   cell->num_vis = 0;

   occls_init(root,cell->template->num_f);

   while (cand = occls_pop()){

      if (!((num_group_occ+ num_individual_occ+cell->num_vis)%100) &&
          num_group_occ+num_individual_occ+cell->num_vis > 0){
         fprintf(ErrFile,"num_g_occ= %d, num_i_occ = %d, num_vis= %d\n",
             num_group_occ,num_individual_occ,cell->num_vis);
         fflush(ErrFile);
      }

      if (candintersectcell(cand,cell,wb)) {
         if (cand->face) {
            cell->fcr[cand->face->id] |= (VISIBLE | CHECKED);
            cell->num_vis++;
         }
         else             stack_children(cand->node,cell);
      }
      else {
         if (vis_from_boundary(pdata,cell,WEST,  cell->westpl,cand,wb)  ||
             vis_from_boundary(pdata,cell,EAST,  cell->eastpl,cand,wb)  ||
             vis_from_boundary(pdata,cell,SOUTH,cell->southpl,cand,wb)||
             vis_from_boundary(pdata,cell,NORTH,cell->northpl,cand,wb)||
             vis_from_boundary(pdata,cell,EARTH,cell->earthpl,cand,wb)||
             vis_from_boundary(pdata,cell,SKY,   cell->skypl,   cand,wb)){

            if (cand->face) {
               cell->fcr[cand->face->id] |= (VISIBLE | CHECKED);
               cell->num_vis++;
            }
            else stack_children(cand->node,cell);
         }
         else{
            if (cand->face){
               cell->fcr[cand->face->id] &= ~VISIBLE;
               cell->fcr[cand->face->id] |= CHECKED;
               num_individual_occ++;
            }
            else{/*mark all faces definitely inside the cell as occ. */
               FOREACH_FACE_IN_TEMPLATE(cell->template,facep,i){
                  fcr = set_face_cell_relation(facep,cand->node->cell);
                  if (FACEINSIDE(fcr)){
                     num_group_occ++;
                     cell->fcr[i] &= ~VISIBLE;
                     cell->fcr[i] |= CHECKED;
                  }
               }
            }
         }
      }
   }
   occls_term();
   fprintf(ErrFile,"f= %d, g_occ= %d, i_occ= %d, vis= %d\n",
      cell->template->num_f,num_group_occ,
      num_individual_occ,cell->num_vis);
}

/*
we need the CHECKED bit because faces can be in the boundary of two
cells
and thus FACEINSIDE could be true for 1 face and two cells.
*/
void
stack_children(t,c)
Node*   t;  /* stacking this nodes children */
Cell*           c;  /* computing occlusions for this cell */
{
   int                    i,split;
   FaceCellRelation       fcr;
   FaceCellRelation       fcrng;

   if (!t->ng /* && !t->nl*/){                              /* leaf node */
      for (i = 0; i < t->cell->template->num_f; i++){
         fcr = set_face_cell_relation(t->cell->template->faces[i],
                                      t->cell);
         if (FACEINSIDE(fcr) && !(c->fcr[i] & CHECKED)){
            c->fcr[i] |= CHECKED;
```

```
                occls_push(t->cell->template->faces[i],(Node*) 0);
             }
          }
       }
       else{
          occls_push((Face*) 0,t->ng);
          occls_push((Face*) 0,t->nl);

          switch(t->si){
             case X: split = EASTSP; break;
             case Y: split = NORTHSP; break;
             case Z: split = SKYSP; break;
          }
          for (i = 0; i < t->cell->template->num_f; i++){
             fcr = set_face_cell_relation(t->cell->template->faces[i],
                                          t->cell);
             fcrng= set_face_cell_relation(t->cell->template->faces[i],
                                           t->ng->cell);
             if (FACEINSIDE(fcr) &&
                 (fcrng & split) &&
                 !(c->fcr[i] & CHECKED)){
                c->fcr[i] |= CHECKED;
                occls_push(t->cell->template->faces[i],(Node*) 0);
             }
          }
       }
    }
    /*
    */
    int
    candintersectcell(cand,cell,wb)
    OcclStackEl*    cand;
    Cell*           cell;
    Vert            wb[MAXVERTSINFACE];
    {
       if (cand->face){
          if (!(cell->fcr[cand->face->id] & TRIVIALLYOUTSIDE) &&
              ((FACEINSIDE(cell->fcr[cand->face->id]) ||
                clip_face_to_box(cell->west,cell->east,
                                 cell->south,cell->north,
                                 cell->earth,cell->sky,cand->face,wb))))
             return 1;
          else
             return 0;
       }
       else{
          if (cand->node->cell->east  <= cell->west  ||
              cand->node->cell->north <= cell->south ||
              cand->node->cell->sky   <= cell->earth)
             return 0;
          else
             return 1;
       }
    }
    /*
    */
    void
```

```
occls_init(root,stacksize)
Node*   root;
int              stacksize;
{
   CALLOCN(OcclS,OcclStackEl,stacksize,"occl_init");
   OcclS[0].face = (Face*) 0;
   OcclS[0].node = root;
   OcclSNum = 1;
   OcclSMaxNum = stacksize;
}

/**/
void occls_term() { free((char*)OcclS); }

/*
*/
OcclStackEl*
occls_pop()
{
   if (OcclSNum > 0)
      return &OcclS[--OcclSNum];
   else
      return (OcclStackEl*) 0;
}
/*
*/
void
occls_push(face,node)
Face*            face;
Node*   node;
{
   if(OcclSNum < OcclSMaxNum){
      OcclS[OcclSNum].face = face;
      OcclS[OcclSNum++].node = node;
   }
   else die("occls_push","stack too small",1);
}


/* Foreach portal we want to establish the existence of a
   face (or (sub)set of faces(s)), occl,  such that
   the convex hull of the projections of cand onto the plane of occl
   is contained by the occl face(s).
*/
int
vis_from_boundary(pobj,cell,bid,pl,cand,wb)
GeomData*                 pobj;
Cell*                     cell;
int                       bid;              /* boundary id, SKY,EARTH etc */
FaceLink*                 pl;               /* portal list */
OcclStackEl*              cand;             /* candidate face */
Vert                      wb[MAXVERTSINFACE]; /* work buffer */
{
   FaceLink*     plp;
   Vert          ccv[MAXVERTSINFACE];/* cand clipped verts */
   int           ccvn;
   Extent        cex;
```

```
int           i,c;
int           bc;             /* cell boundary normal axis */
double        bv;             /* cell boundary plane value */
int           split;
int           clip;
int           vis;

int           max,min;
IndexNode**   index;

if (!pl) return 0;

if (cand->face){
   switch (bid){
      case WEST:
         if (cell->fcr[cand->face->id] & (WESTNL | WESTEQ)) return 0;
         split = (cell->fcr[cand->face->id] & WESTSP);
         bc = X; bv = cell->west; clip = 1;
         break;
      case EAST:
         if (cell->fcr[cand->face->id] & (EASTNG | EASTEQ)) return 0;
         split = (cell->fcr[cand->face->id] & EASTSP);
         bc = X; bv = cell->east; clip = -1;
         break;
      case SOUTH:
         if (cell->fcr[cand->face->id]&(SOUTHNL | SOUTHEQ)) return 0;
         split = (cell->fcr[cand->face->id] & SOUTHSP);
         bc = Y; bv = cell->south; clip = 1;
         break;
      case NORTH:
         if (cell->fcr[cand->face->id]&(NORTHNG | NORTHEQ)) return 0;
         split = (cell->fcr[cand->face->id] & NORTHSP);
         bc = Y; bv = cell->north; clip = -1;
         break;
      case EARTH:
         if (cell->fcr[cand->face->id]&(EARTHNL | EARTHEQ)) return 0;
         split = (cell->fcr[cand->face->id] & EARTHSP);
         bc = Z; bv = cell->earth; clip = 1;
         break;
      case SKY:
         if (cell->fcr[cand->face->id] & (SKYNG | SKYEQ))   return 0;
         split = (cell->fcr[cand->face->id] & SKYSP);
         bc = Z; bv = cell->sky; clip = -1;
         break;
      default: die("vis_from_boundary","bad boundary id",1); break;
   }

   cex[MINX] = cand->face->ex[MINX];
   cex[MAXX] = cand->face->ex[MAXX];
   cex[MINY] = cand->face->ex[MINY];
   cex[MAXY] = cand->face->ex[MAXY];
   cex[MINZ] = cand->face->ex[MINZ];
   cex[MAXZ] = cand->face->ex[MAXZ];

   /* clip cand->face to the correct side of the boundary */
   if (split){
      for (i=0; i < cand->face->n; i++){
         wb[i][X] = cand->face->verts[i][X];
```

```
            wb[i][Y] = cand->face->verts[i][Y];
            wb[i][Z] = cand->face->verts[i][Z];
         }
         ccvn = clip_to_ortho_plane(cand->face->n,wb,bc,bv,clip,ccv);
         if (clip == -1) cex[MINEX(bc)] = bv;
         else            cex[MAXEX(bc)] = bv;
      }
      else{
      /* cand is on correct side of cell boundary so just copy to ccv */
         for (i=0; i < cand->face->n; i++){
            ccv[i][X] = cand->face->verts[i][X];
            ccv[i][Y] = cand->face->verts[i][Y];
            ccv[i][Z] = cand->face->verts[i][Z];
         }
         ccvn = cand->face->n;
      }
      if (ccvn < 3) die("mark_occluded_faces","bad cand->face clip",1);
   }
   else{ /* construct ccv from cell extent */
      cex[MINX] = cand->node->cell->west;
      cex[MAXX] = cand->node->cell->east;
      cex[MINY] = cand->node->cell->south;
      cex[MAXY] = cand->node->cell->north;
      cex[MINZ] = cand->node->cell->earth;
      cex[MAXZ] = cand->node->cell->sky;

      switch (bid){
         case WEST:
            if      (cex[MINX] > cell->west) return 0;
            else if (cex[MAXX] > cell->west) cex[MAXX] = cell->west;
            break;
         case EAST:
            if      (cex[MAXX] < cell->east) return 0;
            else if (cex[MINX] < cell->east) cex[MINX] = cell->east;
            break;
         case SOUTH:
            if      (cex[MINY] > cell->south) return 0;
            else if (cex[MAXY] > cell->south) cex[MAXY] = cell->south;
            break;
         case NORTH:
            if      (cex[MAXY] < cell->north) return 0;
            else if (cex[MINY] < cell->north) cex[MINY] = cell->north;
            break;
         case EARTH:
            if      (cex[MINZ] > cell->earth) return 0;
            else if (cex[MAXZ] > cell->earth) cex[MAXZ] = cell->earth;
            break;
         case SKY:
            if      (cex[MAXZ] < cell->sky) return 0;
            else if (cex[MINZ] < cell->sky) cex[MINZ] = cell->sky;
            break;
         default: die("vis_from_boundary","bad boundary id",1); break;
      }

      extent_to_box(cex,ccv);

      ccvn = 8;
   }

   /* now try to show no portal can see cand */
   for (plp = pl; plp; plp = plp->next){
      vis = 1;
      for (c = Z; c >= X && vis; c--){
         switch (c) {
            case Z: index = IndexXY; break;
            case Y: index = IndexZX; break;
            case X: index = IndexYZ; break;
         }
         /* We only allow zero-area contact with an occluding plane.
            This is subtle so think about it for a bit.
         */

         if       (plp->f->ex[MAXEX(c)] < cex[MINEX(c)]){
            if ((min= find_index(c,plp->f->ex[MAXEX(c)])) == -1)
                                                                   continue;

            if (index[min]->v <  plp->f->ex[MAXEX(c)] ||
                index[min]->v == plp->f->ex[MINEX(c)])                min++;

            if ((max= find_index(c,cex[MINEX(c)])) == -1)
                                                                   continue;

            if ( (cand->node && index[max]->v >= cex[MINEX(c)]) ||
                 (cand->face && (index[max]->v >  cex[MINEX(c)] ||
                             index[max]->v == cex[MAXEX(c)]) ) ) max--;
         }
         else if (plp->f->ex[MINEX(c)] > cex[MAXEX(c)]){
            if ((min= find_index(c,cex[MAXEX(c)])) == -1)
                                                                   continue;

            if ( (cand->node && index[min]->v <= cex[MAXEX(c)]) ||
                 (cand->face && (index[min]->v <  cex[MAXEX(c)] ||
                             index[min]->v == cex[MINEX(c)]) ) ) min++;

            if ((max= find_index(c,plp->f->ex[MINEX(c)])) == -1)
                                                                   continue;

            if (index[max]->v >  plp->f->ex[MINEX(c)] ||
                index[max]->v == plp->f->ex[MAXEX(c)])                max--;
         }
         else continue;
         for (i = min; i <= max && vis; i++)
            vis= !occlusion_exists(pobj,plp,ccv,ccvn,c,index[i],wb);
      }
      /* if no occlusion exists for this portal, return a 1 */
      if (vis) return 1;
   }
   return 0; /* if this happens we are very happy */
}

/* check if some part of index[i] occludes cand:
   Reverse project cand onto the plane of
   index[i] by computing the intersection of the segments defined by
   a vertex of the portal and the vertices
   of cand. If the convex hull of the reverse projections from all
   the vertices of the portal minus index[i]->eq is empty
   then cand is occluded for that portal.
*/

int
occlusion_exists(pobj,p,ccv,n,c,indexnode,wb)
GeomData*       pobj;                  /* for sweep intersections etc */
FaceLink*       p;                     /* portal in question */
Vert            ccv[MAXVERTSINFACE];   /* verts of clipped cand */
int             n;                     /* count of verts */
int             c;                     /* orientation of indexnode plane
*/
IndexNode*      indexnode;             /* plane of hopefully occluding
faces*/
Vert            wb[MAXVERTSINFACE];    /* work buffer for projections */
{
   int                  i,j;
   VertPtr              v;
   double               t,numerator;
   int                  x,y,z;
   int                  hn;
   FaceLink             hflink;
   FaceLink*            fl;
   static VertPtr       vertps[MAXVERTSINFACE];
   static Face          hfacedata;
   Face*                hface = &hfacedata;
   Vert                 hull[MAXVERTSINFACE];
   int                  savenumv;

   switch (hface->o = c) {
      case X: x = Y; y = Z; z = X; break;
      case Y: x = Z; y = X; z = Y; break;
      case Z: x = X; y = Y; z = Z; break;
      default: die("occlusion_exists","c < X || c > Z",1);
   }
   /* compute the convex hull of the set of "reverse" projections of ccv
      onto the indexnode plane from the vertices of p.
   */
   hn = 0;
   FOREACH_VERT_IN_FACE(p->f,v,j){
      numerator = (indexnode->v - v[c]);
      for (i = 0; i < n && hn < MAXVERTSINFACE; i++){
         t = numerator/(ccv[i][c] - v[c]);
         if         (t > 1.0 && t < 1.0 + FEPS) t = 1.0;
         else if    (t < 0.0 && t >  -FEPS)     t = 0.0;
         if (t > 1.0 || t < 0.0) {
            fprintf(ErrFile,"t = %g,ccv = %g %g %g indexnode->v=%g\n",
                  t,ccv[i][X],ccv[i][Y],ccv[i][Z],indexnode->v);
            die("occlusion_exists","bad t value",1);
         }
         wb[hn][x] = t*(ccv[i][x] - v[x]) + v[x];
         wb[hn][y] = t*(ccv[i][y] - v[y]) + v[y];
         wb[hn][z] = indexnode->v;

         /* limit the precision of the computed value to two digits */
         wb[hn][x] = rint(wb[hn][x]*1e2)/1e2;
         wb[hn][y] = rint(wb[hn][y]*1e2)/1e2;
```

```
            /* if any points are outside the extent of indexnode, ret 0 */
            if (wb[hn][x] < indexnode->ex[MINEX(x)] ||
                wb[hn][x] > indexnode->ex[MAXEX(x)] ||
                wb[hn][y] < indexnode->ex[MINEX(y)] ||
                wb[hn][y] > indexnode->ex[MAXEX(y)]) return 0;
            hn++;
         }
         if (i != n && hn == MAXVERTSINFACE)
            die("occlusion_exists","hn > MAXVERTSINFACE",1);
      }

      /* compute the difference of the hull and the faces in the index
         plane.if the result is 0 then ccv was occluded with respect to the
         portal
      */
      if ((hface->n = convex_hull(wb,hull,hn,c)) < 3)
         return 1;
      hface->set_id = 0;
      hface->flags = CONVEX;
      hface->verts = vertps;
      for (i = 0; i < hface->n; i++) hface->verts[i] = (VertPtr) hull[i];
      hflink.f = hface;

      hflink.next = indexnode->eq;

      for (fl = hflink.next; fl; fl = fl->next) fl->f->set_id = 1;

      /* the vertices in pobj created by intersections detected by sweep
         can be reused
      */
      savenumv = pobj->num_v;
      if (sweep_detect(pobj,&hflink,portal_func,0)) {
         pobj->num_v = savenumv;
         return 0;
      }
      else {
         pobj->num_v = savenumv;
         return 1;
      }
   }
```