Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments[1]

John M. Airey, John H. Rohlf†, and Frederick P. Brooks, Jr.

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

†Silicon Graphics Computer Systems
Mountain View, CA 94039-7311

Abstract

Two strategies, pre-computation before display and adaptive refinement during display, are used to combine interactivity with high image quality in a virtual building simulation. Pre-computation is used in two ways. The hidden-surface problem is partially solved by automatically pre-computing potentially visible sets of the model for sets of related viewpoints. Rendering only the potentially visible subset associated with the current viewpoint, rather than the entire model, produces significant speedups on real building models. Solutions for the radiosity lighting model are pre-computed for up to twenty different sets of lights. Linear combinations of these solutions can be manipulated in real time. We use adaptive refinement to trade image realism for interactivity as the situation requires. When the user is stationary we replace a coarse model using few polygons with a more detailed model. Image-level linear interpolation smooths the transition between differing levels of image realism.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - display algorithms, viewing algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - color,shading and shadowing.

Additional Key Words and Phrases: model-space subdivision, potentially visible, radiosity, adaptive refinement.

[1]This work was supported by NSF Grant#CCR-8609588 and ONR Grant#N00014-86-K-0680

1. Introduction

Our basic goal is a virtual building environment, a system which simulates human experience with a building, without physically constructing the building. Many components of a building simulator, corresponding to many human senses, are important. We concentrate on techniques to enhance visual simulation.

We find natural motion 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 1 frame per second, the system is painful to use. It is necessary to use a two-dimensional floorplan display, or *map view*, to navigate.

• As the frame rate increases to around 20 frames per second interactivity appears to increase rapidly before leveling off. At around 6 frames per second the virtual building illusion begins to work. It is possible to navigate with only the three-dimensional display, or *scene view*.

However, realistic images are also important. We use a lighting model that realistically simulates the complicated diffuse, interreflections within a building interior. Furthermore, studying real buildings demands large, detailed models.

Each of these simulation enhancements greatly increases the number of primitives the display subsystem must process and has an adverse effect on interactivity. To help our display subsystem cope with this increased load and remain interactive, we use two stategies. We pre-compute as much work as possible and use adaptive refinement techniques during display [Bergman86]. The resulting combination is a system that attempts to keep the update rate above about six frames per second while providing as realistic an image as possible.

user, e.g., a client, may desire a restrictive but more natural interface, (a treadmill with a head-mounted display or big screen display). We expect to report on the research results of the interface subsystem in another publication.

2.4 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 2 depicts the 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 filename extensions.

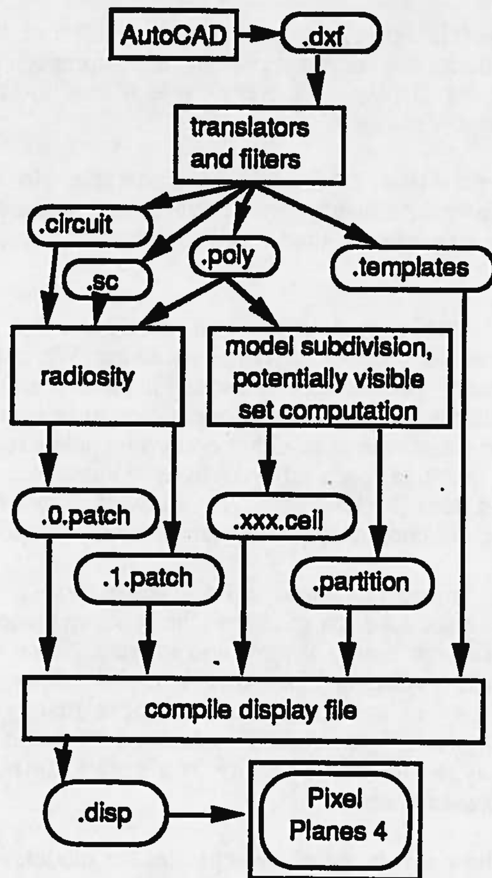


Figure 2. Display File Compilation in UNC Walkthrough System.

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. The display compilation forks into the radiosity process and the model subdivision process.

The radiosity system is described in Section 4. It processes .poly files by automatically dicing the polygons into derived patches and computing color values for each patch for each independent light circuit. The original polygons are given color values by averaging the values of their derived patches. The patches retain indices to parent polygons. The shaded polygons are written to a file with a .0.patch extension and the derived patches are written to a file with a .1.patch extension. This information is used by the next program to construct hierarchical polygons. A *hierarchical polygon* is a polygon that has an associated list of polygons that may be used to replace the polygon. We have experimented only with one level of refinement and with refining all polygons to the same level simultaneously, but our data structures and display code allow arbitrary subdivision for each polygon independently.

The model subdivision process is described in Section 3. It generates a recursive subdivision of the model space. 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.

3. Display Acceleration by Model-Space Subdivision and Volume-to-Polygon Visibility Testing

Architectural databases possess special characteristics. We list some of these properties here and then describe how we exploit them.

1. The model is changed much less often than the viewpoint, which makes pre-processing desirable.

2. Many buildings have high average depth complexity. Any image computed from an interior viewpoint will have many surfaces covering every pixel. A related observation is that most of the model does not contribute at all to any given image. Furthermore, the depth complexity of a building is basically independent of tesselation due to shading and independent of the amount of detail modelled.

3. Most polygons are axial, that is parallel to two of the coordinate axes. Additionally many polygons are rectangles.

4. The set of polygons that appears in each view changes slowly as the viewpoint moves, except when crossing certain thresholds, e.g., doors, which we call *portals*.

The metrics we use 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:

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

To satisfy Objective 2, the process terminates when no partitioning plane has a partition priority exceeding a user-defined threshold or when other limits, such as tree-depth, are exceeded. The process generates a tree with interior nodes representing binary separating planes and leaf nodes representing cell volumes.

If we ran this function on the "planes" in our simple example floor plan, the wall that separates room 2 and room 3 from room 1, the plane y=1, would have 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 4 and 5).

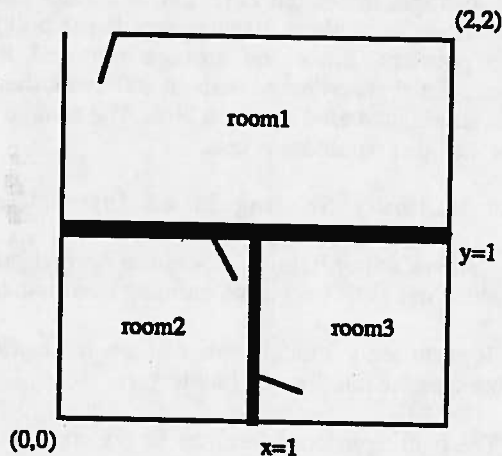


Figure 4. The Subivided Floor Plan.

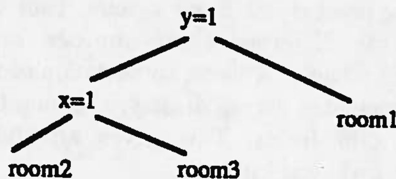


Figure 5. The Corresponding Tree Data Structure for Figure 4. Interior Nodes Represent Splitting Planes and Leaf Nodes Represent Cell Volumes.

3.2 Volume-to-Polygon Visibility Testing

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 actual doors. In real-life datasets, hallways, stairwells, windows, and oddly shaped rooms give rise to other portals. Algorithms that compare co-planar sets of polygons can compute the actual polygonal definitions of the portals [Ottman85],[Weiler81].

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

Unfortunately, this is also a difficult problem. We need to know what is visible from an *area*, an infinite albeit bounded number of viewpoints. We call this problem the view*area* problem.

This is fundamentally equivalent to computing the polygons that receive direct illumination from an area light source [Nishita85]. 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 viewarea problem are complex and inefficient, we have developed two complementary classes of algorithms to compute approximations to the exact solution. These are detailed in [Airey90].

One class 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, it is implemented with the same ray-polygon intersection library used by our radiosity implementation, Section 4.

Another class establishes occlusion relationships among polygons. This is based on the computation of shadow volumes [Crow77]. Since exhaustive computation of shadow volumes is expensive, we 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, we hope they can be combined into a more accurate algorithm in the future.

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.

4.1 A Ray-Casting Approach

We use a modified shooting approach to compute the radiosity solution. The sampling process uses ray-casting based on a jittered hemispherical distribution, rather than a Z-buffer based hemi-cube [Airey89].

At each iteration step, we adapt the resolution of the hemispherical sampling distribution as a function of unshot radiosity to keep the radiosity per ray constant. Airey and Ouh-young observed, empirically, that the unshot radiosity at each step decreases as a negative exponential. Thus, the number of rays fired at each step also decreases as a negative exponential.

A new ray-polygon intersection algorithm tuned to architectural databases accelerates the ray casting. It 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 6.

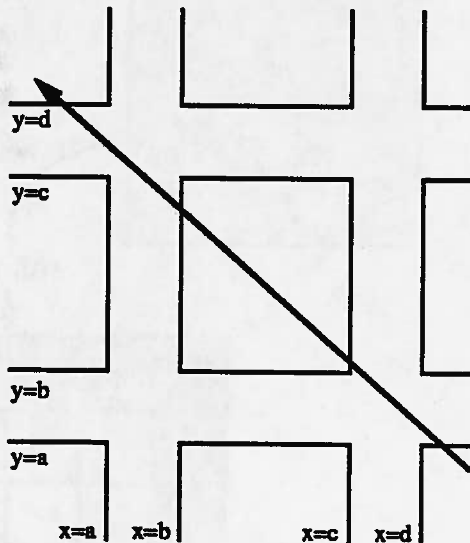


Figure 6. Ray-Line Segment Intersection.

The ray 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 the normal direction. This data structure can be 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 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.

The process works in three dimensions similarly. The small percentage of non-axial polygons are put into 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 primary 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.

4.2 Interactive Light Manipulation

We have extended our radiosity program to compute the contributions of several different light circuits. For each patch we simultaneously 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, large physical memories and virtual memory ease this problem. However, we did not have enough display memory for some of our models. 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,

$\langle r,g,b\rangle_k$, with $0 <= r,g,b < 256$,

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

$\langle R,G,B\rangle = \Sigma\ (\langle r,g,b\rangle_k)$;
max = MAX(MAX(R,G),B);

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.

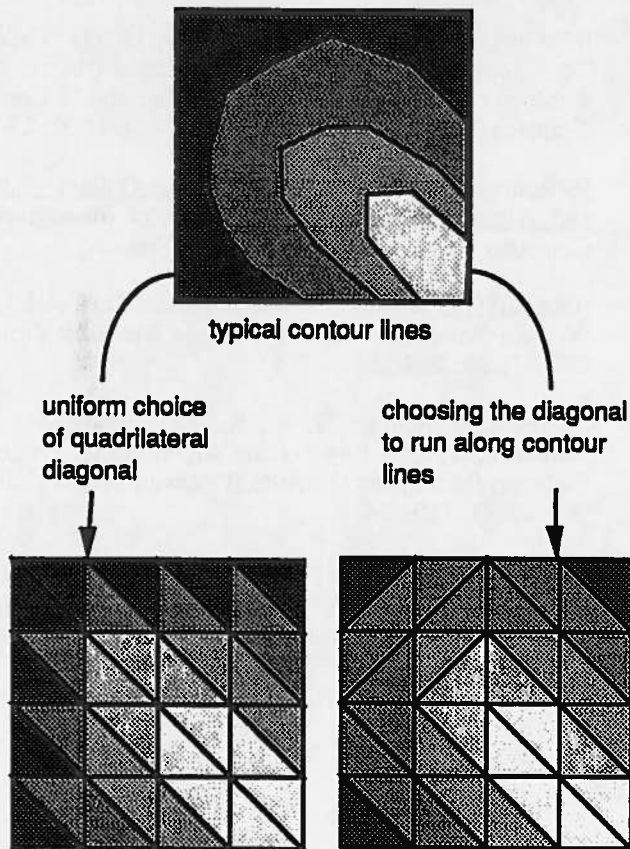


Figure 8. Uniform Choice of Quadrilateral Diagonal vs. Difference Directed Choice. In the Actual Image, the Colors are Transfered to Patch Vertices and Linear Interpolation Provides Smooth Shading.

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 8).

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.

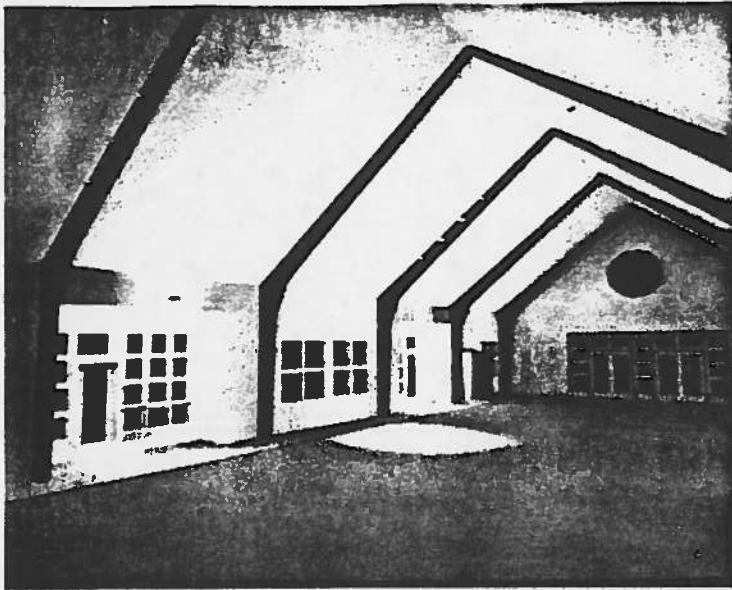
6. Color Plates

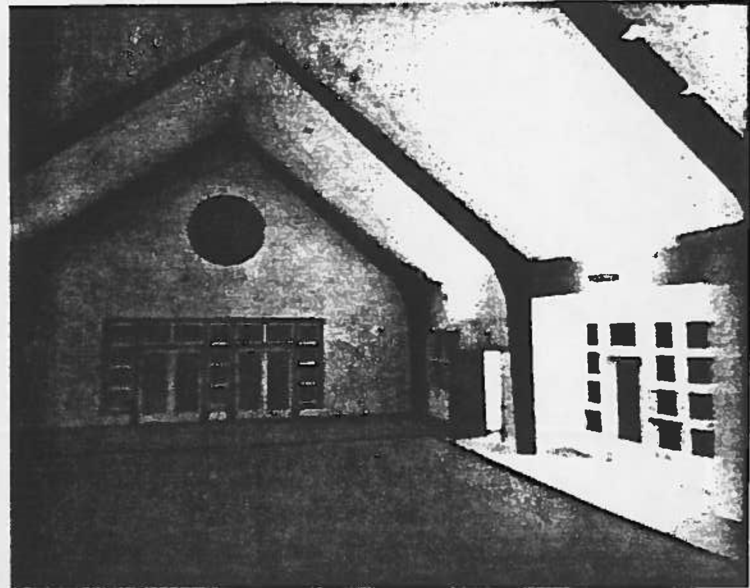
The church2 model of Orange United Methodist Church Fellowship Hall has served as one of our primary system evaluation databases. It has 6037 polygons drawn from a larger 12,000+ polygon model. The model subdivision process partitioned it into 16 cells. As a result, the display subsystem needs to process 1887 polygons on average and 3477 in the worst case. Pixel-Planes 4 can display the basic model at more than 8 frames per second.

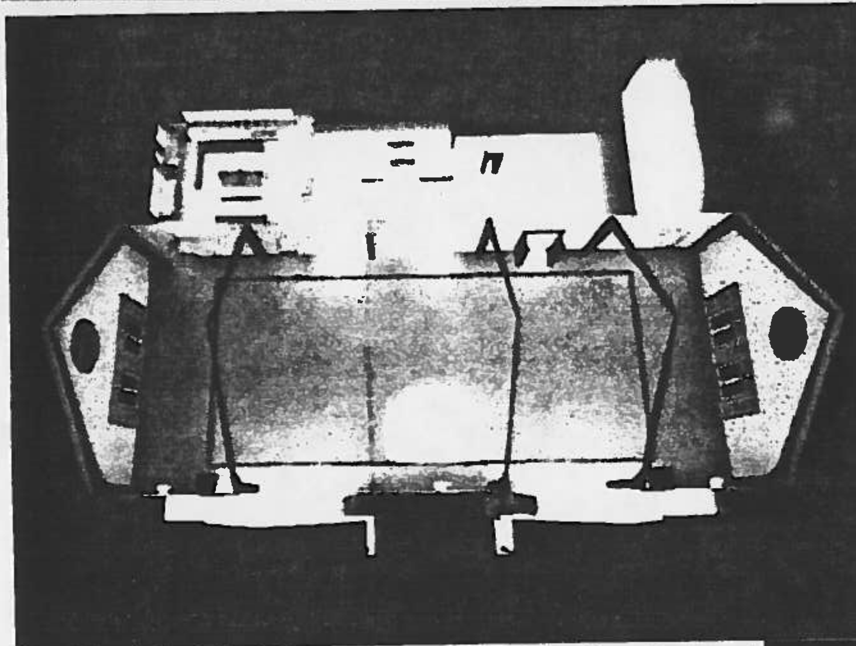
All five color plates show the image that is produced when the viewer is stationary and adaptive refinement has replaced the coarse model with the radiosity-shaded model and anti-aliasing has smoothed the jaggies. The radiosity process produced 26,794 patches with 65,627 vertices from the original 6037 polygons by dicing at a resolution of 21 square inches. A radiosity solution was computed for 13 different lighting circuits. These may be manipulated interactively. The radiosity solution was computed overnight on a DECstation 3100.

The building was designed by Wesley McClure and Craig Leonard of McClure NBBJ. The modelling was done by Penny Rheingans and John Alspaugh with AutoCAD.

Plate 3. is a perspective plan view. We allow the user to choose whether or not back-facing polygons are rendered. In







1. 2.

3.

4. 5.

Airey et al.