# Fast and Accurate Radiosity-Based Rendering

Kevin P. Smith

Master's Project Report
Under the Guidance of
Carlo H. Séquin

May 23, 1991

**Abstract**

Current techniques of radiosity-based rendering are inadequate for complex scenes with 100,000 polygons or more. Existing approaches either take too long, or produce poor images. By fully exploiting the special organization of some recent graphics hardware, fast computers, and some new algorithms, we can generate accurate images far more quickly than existing application programs.

Our approach pushes the frontiers of both speed and correctness. It combines the speed of progressive radiosity with the flexibility provided by adaptive subdivision. We have developed a new visibility algorithm specifically designed to take advantage of the fast graphics hardware available on the Silicon Graphics IRIS machines.

Another difficulty addressed is the problem of having to start from poor input models. Many of the existing architectural models of buildings have been developed with 2-dimensional drafting tools such as AutoCAD. These models have various problems including randomly oriented faces, intersecting or poorly shaped faces, and overlapping coplanar features. With a number of preprocessing tools, such input files are converted into a more consistent description that is directly suited for radiosity analysis.

These experiments and improvements have been developed in an environment that permits us to replace and update individual program modules so that we can compare and evaluate, for instance, different patch and element meshing algorithms, and study the effect of different algorithms on the quality of the final image.

## 1 Introduction

Architects have been designing buildings for years, but until recently they have had no good ways of generating accurate images showing what a building will look like. Specifically, generating realistic images of the interior of buildings has been a tough challenge. A technique known as radiosity, however, can provide this capability.

Radiosity was introduced in 1984 [7], and since then, intense research has focused on extending the original ideas and finding ever more efficient and more accurate algorithms. Yet despite a lot of research, radiosity is still not used by its intended audience, the architect. There are two primary reasons for this. The first reason is that existing radiosity techniques are still either too slow or insufficiently accurate to apply to reasonably large scenes. The second reason is that most of the existing techniques are also too difficult to use.

The radiosity technique was introduced in 1984 as a method of simulating the interreflection of light within a diffusely reflective environment [7]. In this approach, if there are $n$ polygons in the environment, a $nxn$ matrix is created where each entry in the matrix describes what percent of the energy leaving one polygon arrives on the other polygon. Combined with the energy emitted by the polygons representing light sources, this matrix represents a system of linear equations. Determining the final energy of each polygon is then a matter of solving this system. While this matrix method works well for small scenes, the memory requirement scales as a factor of $n^2$, and solving the matrix takes at least $n^2$ time. For large scenes, this cost is too high.

A technique to overcome this difficulty, known as progressive radiosity, was introduced in 1988 [4]. In this algorithm, the polygon with the most energy to distribute is identified, and its energy is distributed to the rest of the scene. This process is repeated numerous times until the remaining undistributed energy in the scene becomes insignificant. This method has proven to be significantly faster than the matrix method with no appreciable loss of accuracy.

While this method has proven to be quite quick, the images produced are of about the same accuracy. Attaining better accuracy for a radiosity solution depends upon creating a reasonable subdivision of the polygons in the scene. It is not possible to capture the detail of a sharp shadow with a coarse polygon mesh.

In 1990, Campbell and Fussell presented a radiosity algorithm that adaptively generated a mesh by projecting shadow volumes onto receiving polygons [8]. This approach appears to be quite adequate for the generation of a reasonable mesh, however, the time required to project these shadow volumes is excessive, and this approach is impractical for large scenes.

Not only are existing techniques too slow or inaccurate to apply to large scenes, but they are also too difficult to use. The problem is that all of these techniques place a number of restrictions on the representation of the input model. For example, these techniques require that there be no intersecting polygons in the input model, and that all connectivity information be stated explicitly. Unfortunately, most CAD packages that architects use to design buildings do not meet these constraints. For this reason, it is not possible to apply existing radiosity techniques to their models without laboriously reworking the model description. This cumbersome process scares away most potential users. In order to overcome this difficulty, a series of filters has been developed that can take a typical input model and automatically generate one that is geometrically and topologically suited for radiosity. This series makes our radiosity system easy to use.

Additionally, speed can be obtained by using a progressive radiosity algorithm in conjunction with fast computers and specialized graphics hardware. The model is first subdivided into a reasonable number of *patches*, where a patch represents a primary or secondary light source. Then the patches iteratively distribute their energy to the scene using the specialized graphics hardware for visibility analysis.

Lastly, accuracy can also be attained through use of a patch/element hierarchy, as introduced in 1986 by Cohen, et. al [6]. By adaptively subdividing patches into *elements*, where an element represents an energy receiving polygon, the polygon mesh can be made as fine as needed locally without unnecessarily increasing the number of patches. Also, by distributing energy to the element vertices instead of to the elements themselves, as suggested by Wallace, et. al [14], additional accuracy is attained.

Overall, the processing of the model follows these steps:

```
preprocess model to make it consistent
subdivide surfaces into patches
subdivide the patches into sufficiently small elements
repeat
  Find P:  the brightest patch in the scene.
  Calculate E:  the energy that P has to distribute.
  Determine which vertices in the scene are visible to P.
  Analytically calculate the energy density distributed from P
    to all visible vertices.
  Adaptively subdivide elements where needed.
  Subtract E from P's energy supply.
until the solution shows adequate convergence.
```

The rest of this paper is spent explaining the various steps in detail. Section 2 describes the preprocessing step, section 3 explains and discusses surface and patch subdivision, section 4 discusses visibility analysis, section 5 explains energy distribution, and section 6 covers adaptive subdivision.

## 2 Input Model Conversion

Most existing architectural models are not directly appropriate for radiosity calculations. First of all, many of the face have arbitrary orientation. These models contain little topology or connectivity information, and many even have intersecting faces. Lastly, due to an insufficiently rich modeling language, concave faces are often described by a number of simpler convex faces.

For radiosity calculations, the ideal model should:

- describe coherent faces in a single construct.
- have unique face orientations, i.e., the contours should be described in a counter-clockwise order when viewed from the accessible side of the face.
- possess full connectivity information, i.e., two faces sharing an edge and its vertices should do so explicitly in the data structure.

In order to construct an ideal description from existing models, a number of pre-processing steps must be applied. We call this process "model cleaning".

### 2.1 UniGrafix

The language used to represent architectural models is Berkeley UniGrafix [13]. The four basic statements that are needed for radiosity are:

```
color stmt:          c_rgb <material-name> r g b ;

material definition: defmat <material-name> ;
                       diffuse_rgb  r g b ;
                       emission_rgb r g b ;
                     end ;

vertex definition:   v <vertex-name> x y z [material-name] ;

face definition:     f <face-name> (vertex-name ... vertex-name)
                       [(vertex-name ...) ...] [material-name] ;
```

The "c_rgb" statement is a shorthand that defines a diffuse material with the specified reflectance values, r, g, and b which all range from 0.0 to 1.0.

The "defmat" statement can be used to define area light sources. In this statement, the "emission_rgb" construct defines the amount of energy per unit area that is emitted. When a face is defined with an energy emitting material, the face becomes a light source. Note that there is no upper limit on the values of r, g, and b for the emission_rgb construct. This is because light sources are allowed to be arbitrarily bright.

The vertex statement defines a vertex by name, location, and color. On input to the radiosity program, vertices are not colored. On output, however, vertices will have color, which can be used to Gouraud shade the final image.

The face statement is by far the most interesting of these four statements. A face is defined by an optional name, a list of contours, and a material. Each contour is defined by a list of vertex names.

The first contour in the list is the outer contour and defines the perimeter of the face.

Faces are one-sided, where the visible side is defined as the side that has the outer contour running counter-clockwise in a right-handed coordinate system. When a face has more than one contour, the additional contours either define holes, or islands in the face. Examples of each are shown in figure 1.

Note that the contours that describe holes run clockwise, whereas the contours that describe islands run counter-clockwise.

The UniGrafix language comes with a library that makes program writing fairly easy. The data structures used by the library are "winged-edged", meaning that the faces point to edges, which point back to all faces that use them. Similarly, all of the edges point to their vertices, which point back to

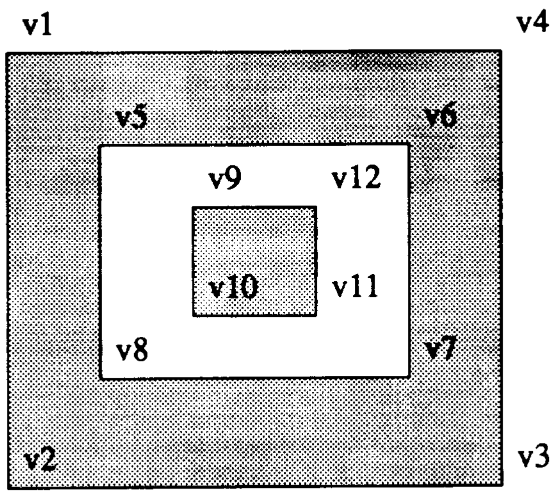


Figure 1: A UniGrafix Face

all of the edges that use them. This type of highly connected data structure makes traversal of polyhedral models easy.

An example of a complete model is shown here. This is a model of a red cube illuminated by a white triangular light. Note that this model is less than ideal because the light is being described as a single face rather than an enclosed volume. It is, however, adequate.

```
c_rgb red 0.9 0.2 0.2;
v XYZ   1 1 1;
v XY    1 1 -1;
v XZ    1 -1 1;
v YZ    -1 1 1;
v X     1 -1 -1;
v Y     -1 1 -1;
v Z     -1 -1 1;
v N     -1 -1 -1;
f x     ( X XY XYZ XZ ) red;
f y     ( Y YZ XYZ XY ) red;
f z     ( Z XZ XYZ YZ ) red;
f a     ( N Z YZ Y ) red;
f b     ( N X XZ Z ) red;
```

```
f c     ( N Y XY X ) red;

defmat whitelight;
  diffuse_rgb 0.4 0.4 0.4;
  emission_rgb 5 5 5;
end;
v 11    2 1 1;
v 12    1 1 2;
v 13    1 2 1;

f light (11 12 13) whitelight;
```

## 2.2 Non-planar face tesselation

The first step in model cleaning is the removal of non-planar faces. Typically, non-planar faces are created by a modeling tool that attempts to approximate a curved surface with a number of quadrilateral polygons. For this reason, non-planar faces are removed by tesselating them into triangles. This tesselation is somewhat arbitrary, as non-planar faces are inherently ambiguous. The UniGrafix program that performs this tesselation is called "ugplanar".

## 2.3 Face flipping

Next, all faces need to be oriented correctly. Remember that faces are one-sided, and the back side of a face should never be visible. The floor of a room, for example, should be oriented to face upwards, because a floor is always viewed from above. If a floor faces downward, then it needs to be "flipped", so that it faces upward.

This step is by far the most difficult step to automate. The reason for this is that there is no sure way to distinguish between the inside of a room and the inside of a solid object. For example, suppose a model consists purely of a cube floating in space. There are two reasonable ways to interpret this scene. One possibility is that it represents a solid cube floating in a vacuum. Another valid interpretation is that it represents a small cubical room in the middle of the earth. Is the scene solid inside the cube, or outside the cube?

If this question can be answered correctly, and the model is flawless except for the randomly oriented faces, then two possible interpretations can be generated, and the user can decide which interpretation is correct.

In fact, if we assume that this scene does not appear in the middle of the earth, then the correct interpretation can be chosen automatically. The problem, however, is that the model is never flawless. The cube floating in space, for example, might be missing a side. In that case, the only choice a program has is to assume that the cube is really a box, and so all of the five faces should be double sided (it can do this by creating five new faces).

But what if the cube has a pin hole in it? Should it be treated as a box with a really small opening? Or should the program assume that the pin hole is a flaw in the model, and treat the object as a cube?

Currently, these problems are not dealt with. Instead, an interactive walk through program allows the user to select individual faces to be flipped, or to drop "light bombs" at various locations. Dropping a "light bomb" means that the program determines what faces are visible to the user, and any of those faces that are pointing away from the user are flipped. This is called "light bombing" because it effectively takes a flashbulb picture of the scene, and any faces that are incorrectly oriented in the snapshot are flipped. These faces have been "bombed" by the flashbulb. Lastly, the user may select faces that should be double sided.

While this process works well, it is less than ideal because it is not automatic. In an ideal situation, a program would carefully analyze the scene, and automatically flip the appropriate faces to produce the desired results. Research along these lines is currently underway at UC Berkeley [10]. By making a few simplifying assumptions, and checking for consistency throughout the scene, it is hoped that an automatic method can eventually be found.

## 2.4 The grouper - ugvlmerge

Most existing models are described as a set of purely convex polygons, where a polygon is defined by a series of geometric coordinates. This rather restricted way of describing a model makes some scenes difficult to represent. An example is a wall with a window in it (shown in figure 2).

The primary fault with the common representation is that it contains T-vertices (see figure 3). These T-vertices cause a number of problems. One problem is that they create discontinuities in a Gouraud shaded image. Another is that finite precision rendering systems will leave gaps between the polygons adjacent to the T-vertex. Lastly, the algorithm used to compute vertex visibility yields incorrect results in the presence of T-vertices.

Another problem with this common representation is that complex faces

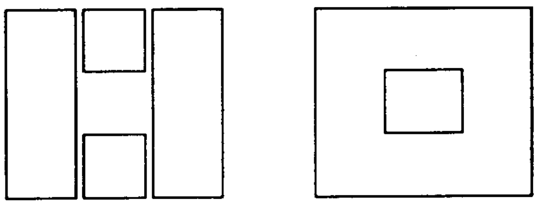


Figure 2: Common versus ideal face representations

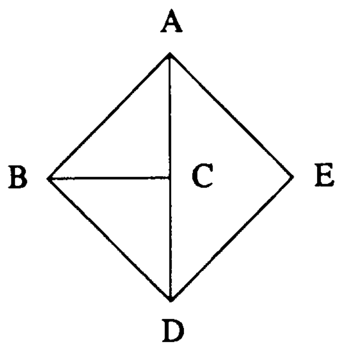


Figure 3: If the face on the right is defined by the vertices A,D,E then there is a T-vertex at C

should be represented as complex faces rather than a number of simple faces. Eventually the complex faces will be subdivided into simple ones, but it would be better to do this in a controlled fashion rather than using the faces provided by the input model.

A program called "ugvlmerge" converts a common representation into an ideal one. This program is also called "the grouper", as it groups sets of simple faces into a collection of complex ones.

In order to construct connectivity information, the first step that the grouper performs is to collapse sets of "nearly coincident" vertices into one vertex (where "nearly" is defined by an epsilon). This is done by storing all vertices in an adaptive octtree. Every time a new vertex is inserted, the tree is examined to determine if another vertex already exists at "nearly" the same position. If so, the new vertex is discared and replaced with the already existing one. Otherwise, the new vertex is inserted.

Next, the grouper finds all vertices which lie "nearly" on the edge of a polygon. Those vertices are then merged into the edges that they lie on. This is done by iterating through all edges of the scene, and recursively searching our octtree for vertices that lie on that edge.

At this point, the grouper has created all connectivity information needed by the patch mesher. The wall with a window in it is fully connected, as shown in figure 4.

The last step performed by the grouper deals with the joining of coplanar polygons that share edges. This is easily achieved by using the information provided by the UniGrafix winged edge data structure. Every edge in the data structure is examined and if two coplanar polygons made of the same material or color share that edge, then the two polygons are merged. This will complete the transformation as shown in figure 5.

### 2.5 Face intersection - ugisect and ugcopl

The next step in cleaning the model is to remove all intersecting faces. The problem with a face that intersects another is that the lighting on one side of the intersection is independent of the lighting on the other side. In order to capture this discontinuity along the line of intersection, the intersection needs to be made explicit.

This is accomplished by identifying pairs of intersecting faces, and cutting them along their line of intersection. A program called "ugisect", which was written by Mark Segal during his graduate work at UC Berkeley does just this [12,11].

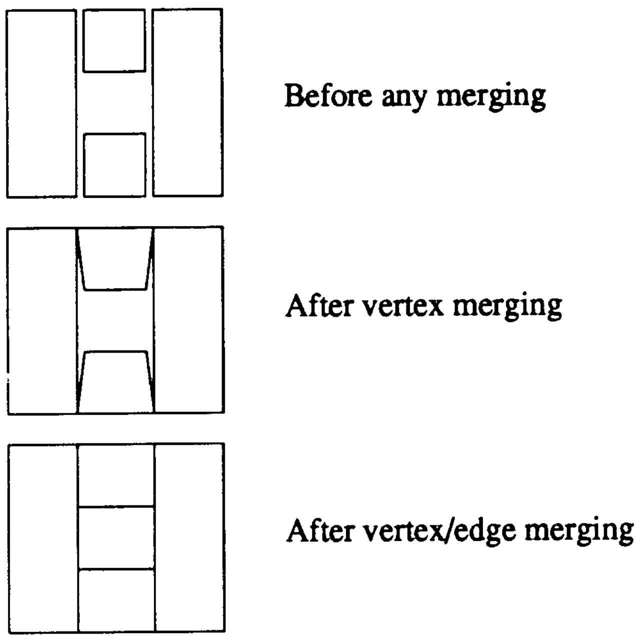


Figure 4: The three stages of vertex merging

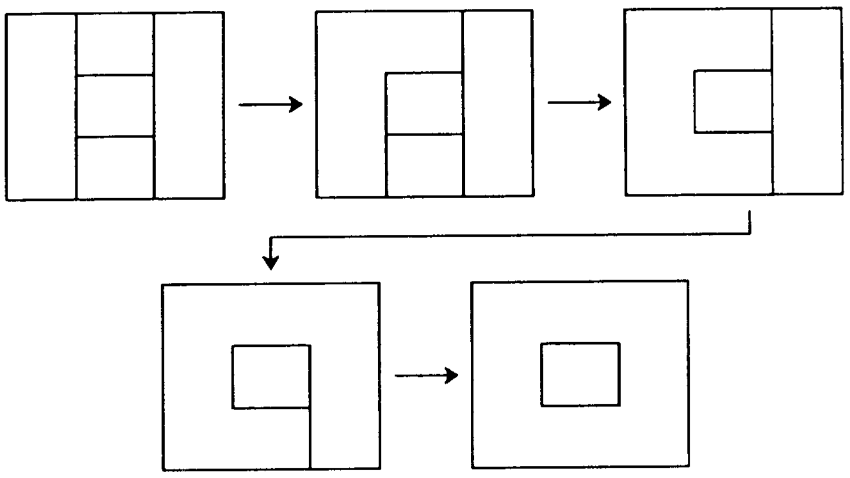


Figure 5: The five stages of merging these polygons

Unfortunately, this algorithm fails to intersect coplanar faces. The reason for this is that coplanar faces do not have a single line of intersection. Instead, they have a plane of intersection. Another program called "ugcopl", however, does intersect coplanar faces. It identifies coplanar faces and uses a sweep line algorithm to determine where the two faces intersect.

## 2.6 Coincident face removal

Another problem that needs to be dealt with is the removal of coincident face pairs. It makes no sense to have two identical faces in the scene, especially when these faces have different material properties. Clearly, the only reasonable way to deal with this problem is to discard one of the two faces.

By examining a number of different models, we have discovered two distinct reasons that these faces are created. In the first case, sometimes multiple faces that describe the same surface overlap. In this case, both faces will be made of the same material, so it does not matter which face is discarded. The second case, however, arises when a piece of paper is placed on a desk. Ugcopl will cut a polygon out of the desk to match the piece of paper, and there will be two coincident faces. The face that should be discarded is the face from the desk, because the piece of paper is supposed to be on top, and should remain visible. In general, it is impossible to know which face was intended to be visible, and which face was intended to be obscured. However, a good guess is that the face that came from a larger surface should be discarded. If a rug is laid upon the floor, for example, the section of the floor that coincides with the rug should be discarded, as the entire floor is larger than the rug. This way, the rug will remain visible. While this heuristic is not perfect, it usually does the right thing. A program called "ugclean" performs this face discarding operation.

# 3 Meshing

After the input model has been cleaned, it is ready for the meshing phase. During the meshing phase, three different hierarchical levels of faces are used. At the hightest level, a *surface* defines the largest connected planar region. A surface might be large like a wall or a ceiling, or it may be small, like the head of a doorknob.

Surfaces are subdivided into *patches*. The purpose of these patches is to act as secondary light sources. Because they are treated as lights of uniform intensity, they need to be sufficiently small so that this assumption does not create noticeable artifacts. This subdivision of surfaces into patches is performed by the UniGrafix filter "ugcut".

During radiosity analysis, these patches are adaptively subdivided into *elements*. Elements represent energy receiving nodes and are used for final rendering of the radiosity image. The purpose of these elements is to provide an accurate image. So that the radiosity algorithm can easily interface with the routines that perform the element meshing, they are provided in the form of an element meshing library.

## 3.1 The patch mesher - ugcut

Ugcut is the patch mesher. It takes an architectural model and chops the surfaces into patches. In doing this, ugcut guarantees two things. First, it guarantees that no T-vertices are created, and second, it guarantees that all patches created are either triangles or convex quadrilaterals. This code was written by Stephen Mann, a graduate student at the University of Washington.

The algorithm itself only takes one parameter: the desired length of a patch edge. Then it iterates over every surface in the model and cuts them one at a time.

The first step in cutting a surface is to divide each of the edges of the surface into segments. This division is done in such a way as to make each of the segments as near to the desired edge length as possible. For example, if the desired length is two units, and a surface edge is 7.6 units long, then that edge is subdivided into four segments, each 1.9 units long.

The purpose of this division is to create some vertices along the edges of the faces from which patches can be cut. Furthermore, the exact same division will be performed on all surfaces that share that edge, and this guarantees that no T-vertices are created at these edges. Figure 6 shows an example of this.

The next step in cutting a surface is laying an imaginary grid of rectangles over it, and creating patches out of any rectangles that lie entirely within the surface. This grid is aligned with the longest edge of the surface. Furthermore, this grid is sized so that it fits perfectly in the bounding box of the surface. This assures that surfaces in the scene that are perfect rectangles are cut into a set of perfectly rectangular patches. Additionally, the rectangles of the grid will have edges of nearly the desired length. An example is shown in figure 7.

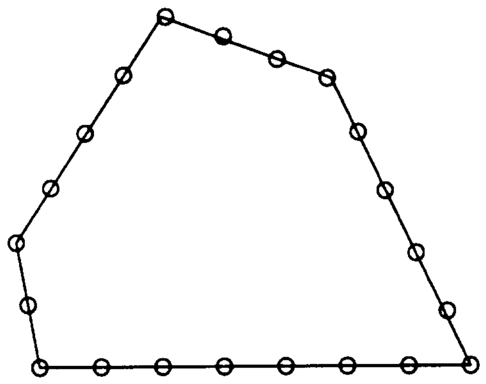


Figure 6: Polygon with vertices inserted along the edges

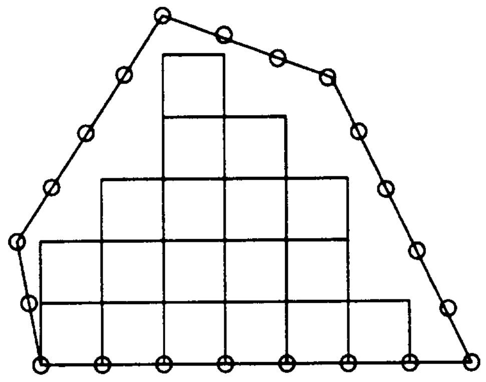


Figure 7: Polygon with overlaid grid

After the rectangular patches have been cut out of the surface, the remaining part of the surface is triangulated with a tesselator that attempts to create nicely shaped triangles. An example is shown in figure 8.

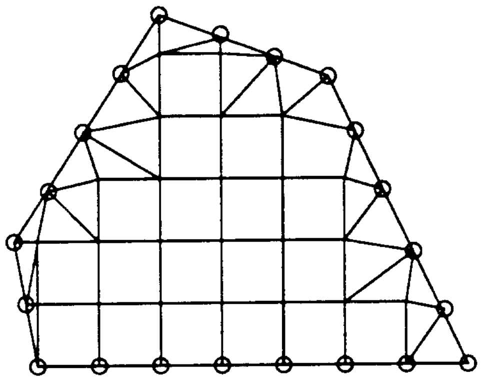


Figure 8: Final output mesh

## 3.2 The element meshing library - libmesh

The element meshing library is a collection of routines for subdividing triangular and quadrilateral meshes. It is used by the radiosity program during the solution phase to adaptively subdivide a patch into many smaller elements in areas where finer resolution is required. Because the patches are always triangles and quadrilaterals, quad trees are used as the basis for subdivision. Two examples of quad trees are shown in figure 9. The use of quad trees guarantees that if the patches provided are nicely shaped, then the children will also be nicely shaped. A technique known as *anchoring* is used to avoid introducing T-vertices into the scene [2]. This code was also written by Stephan Mann.

The first extention to the basic quad tree mesh is that it is *balanced.* If an element is subdivided two levels deeper than one of its neighbors, then its neighbor is also subdivided. This is shown in figure 10.

In addition to being balanced, the quad trees are also *anchored.* An element with T-vertices on at least one of its edges is divided into smaller elements, none of which will contain T-vertices on any of their edges. This anchoring step removes all T-vertices from the mesh. An example is shown in figure 11.

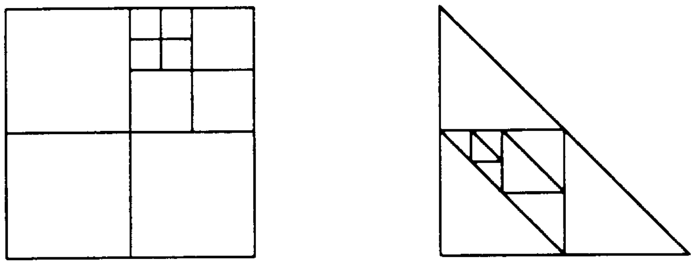


Figure 9: Quadrilateral and triangular quad trees

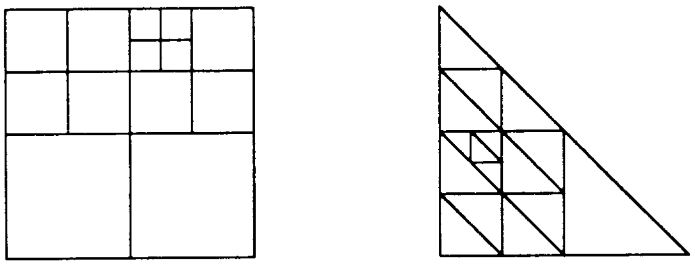


Figure 10: Balanced quad trees

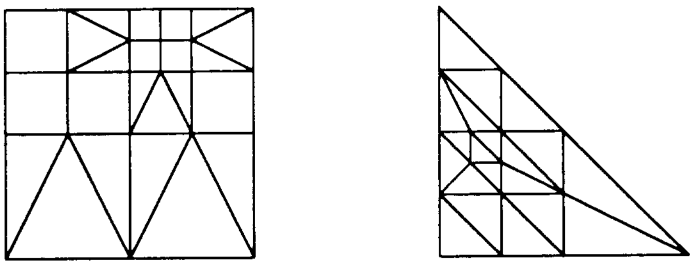


Figure 11: Balanced anchored quad trees

The purpose of this anchoring is to insure that the mesh contains only triangles and quadrilaterals. While this does not provide a significant benefit with the current radiosity algorithm, it is possible that future techniques will borrow more heavily from finite element meshing algorithms, which are usually restricted to triangles and quadrilaterals [9].

A detailed example of a mesh containing triangular and quadrilateral patches is shown in figure 12.

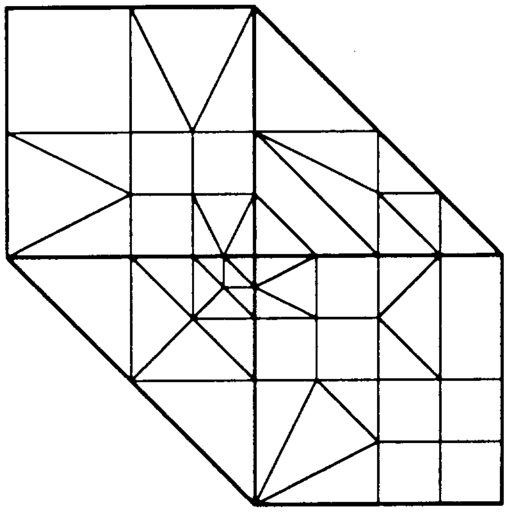


Figure 12: Four patches with balanced anchored quad trees

Across surface boundaries, T-vertices are removed by stitching them into the neighboring polygon's contour rather than balancing and anchoring (see figure 13). This removal is performed differently at surface boundaries because the surface normals and material properties are discontinuous at the boundary, and therefore sophisticated balancing and anchoring techniques are unnecessary.

## 4 Visibility Analysis

The most time consuming part of a progressive radiosity algorithm is visibility analysis. Before energy can be distributed from a light source to the scene, the scene must first be analyzed to determine what parts are visible.

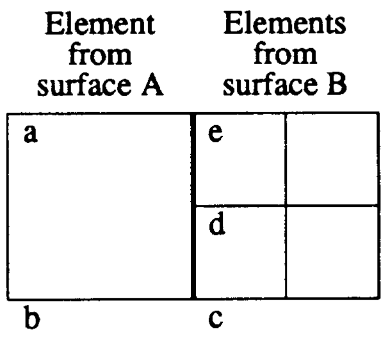


Figure 13: Before stitching, the element from surface A is defined by the vertices (a b c e). After stitching, the same element is defined by the vertices (a b c d e).

As energy is distributed to element vertices, this step requires analyzing what vertices in the scene are visible to a given light source.

## 4.1 Requirements

The two most important concerns for a visibility algorithm are speed and accuracy. In order to achieve the speed, a progressive refinement algorithm that takes advantage of the fast graphics hardware available on the SGI machines is used.

In order to attain the accuracy desired, energy is distributed from patches to vertices using an analytic closed contour integral [3]. There are two primary reasons that this method provides the accuracy desired. The first reason is that when the final image is displayed, the vertices need colors (energy densities) assigned to them for Gouraud shading. Many algorithms estimate these energy values by interpolating the computed energies of the adjacent elements. Instead, by distributing energy directly to the vertices, the final energies used are far more accurate.

The second reason that this algorithm provides a high degree of accuracy is that it does not suffer from traditional hemi-cube aliasing artifacts [3].

Given this framework for an algorithm, we then tried to find a method of using the graphics hardware to quickly determine which vertices in the scene are visible to a given light source. Somehow the scene needed to be rendered into a set of rectangular regions. An obvious algorithm that does this is the hemi-cube algorithm [5]. In this algorithm, the top half of a cube is placed over the center of the light source, and the rest of the scene is projected onto the faces of this cube (see figure 14). In order to use the given hardware, this projection is achieved by rendering the scene five times. In each rendering, one of the five faces from the hemi-cube is chosen, mapped onto the frame buffer, and the scene is rendered. The scene can then be scanned from the frame buffer and analyzed. Several techniques were explored to find the most effective way to render and analyze the scene.

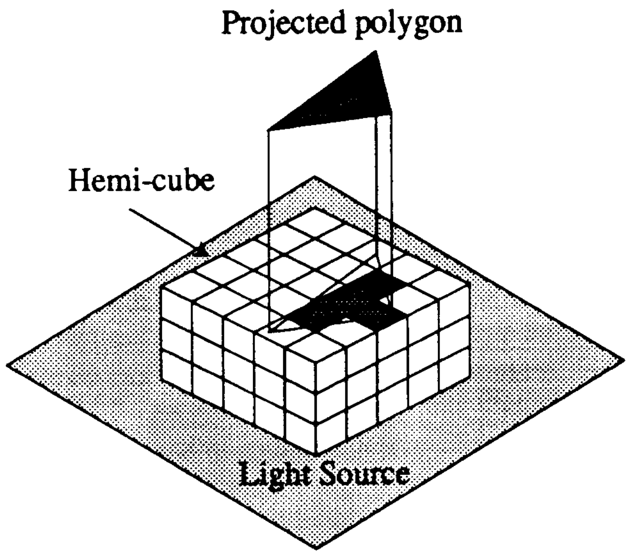


Figure 14: A hemi-cube

## 4.2 Early visibility algorithms

In the initial visibility algorithm, the scene was rendered as a set of elements. It was then scanned to determine which elements were visible, and all vertices from those elements were marked as visible. In this process, the assumption was made that if part of an element is visible, then all of the element's vertices are also visible. There were a number of obvious problems with this approach. The first problem was that the algorithm was too liberal about marking vertices as visible. If even the smallest fraction of an element was visible in the hemi-cube, then all vertices that define that element were assumed to be visible. This incorrectly marked many occluded vertices as visible.

The next visibility algorithm also rendered the scene as a set of elements. Again, the scene was scanned for visible elements, and the vertices of those elements were considered as possible candidates for visibility. This time, however, the vertices were verified to be visible. This was done by computing where the vertex should lie in the scene, and what its z value is. If the element at that location in the scene contained the vertex, or if the vertex's z value was closer than the z value in the scene, then the vertex was marked as visible. This approach solved the problem of incorrectly marking an occluded vertex as visible. However, often the vertices fell on adjacent elements and were marked as not visible due to insufficient z-buffer resolution.

## 4.3 The current algorithm

Our final algorithm comes from a modification to the decal algorithm presented by Kurt Akeley in IRIS magazine [1].

The decal algorithm solves is the problem of surfaces occluding their own vertices. When a surface and its vertices are rendered separately, which is the only way to render them on existing graphics hardware, the z-buffer values for the surface often occludes the vertices. The problem is that the z-buffer value for the surface is calculated at the center of the pixel. The z-buffer value for a vertex, however, is calculated at the location of the vertex. An example showing why this might cause a vertex to lie deeper in the z-buffer than the surface it defines is shown in figure 15. Note that the vertex V has z depth of 9, whereas the polygon at the same pixel has a z depth of 8. Therefore, the polygon has a closer z value and is drawn in front of the vertex.

The decal rendering algorithm solves this problem, and works like this:

```
For every surface in the scene
   render the surface as follows:
             z-buffer    frame buffer        render
     Pass 1: masked off       on             surface
     Pass 2:    on            on       surface's vertices
     Pass 3:    on        masked off         surface
end For
```

With this algorithm, every surface is rendered as a black polygon with vertex-ids on top of it. Since black is the color of the background, the

4 pixels:

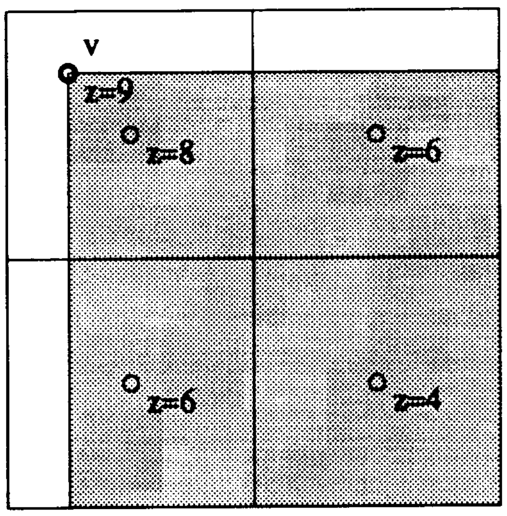


Figure 15: The vertex v, with a depth of 9 is occluded by the surface with a depth of 8

only parts of the scene that are visible are the vertices. This is achieved as follows: First, a particular surface is drawn in black. This erases any vertices that were previously drawn to the hemi-cube that should be occluded by this new surface. However, when the surface is drawn, the z-buffer write mask is turned off so that the z-buffer is not updated. This is done so that the surface's vertices can be rendered into the scene separately without the danger that the surface itself might occlude them (because they have almost identical z-values). Next, the vertices themselves are rendered. They are rendered with both the z-buffer and the frame buffer turned on. Finally, the surface is rendered again, but this time the frame buffer mask is turned off. Since the frame buffer (hemi-cube) is not updated this time, the surface will not accidently erase any of the vertices that were just rendered. The z-buffer, however, is updated so that any vertices drawn later (from other surfaces) will not be rendered if they are occluded by this surface.

A potential problem has to do with vertices that are shared by two surfaces. For example, vertex V might be a part of surface A as well as surface B. Additionally, V might fall on a pixel that is covered by surface B, and have a deeper z value than that surface pixel of B (see figure 16). Now, if surface A is drawn first, then V is drawn into a pixel that is subsequently covered by B. Additionally, the z value there will be set to the z value of V. Later, when B is drawn in black, it will cover the pixel that contains V, and V will be erased. However, vertex V is rendered again (because V is part of surface B), and it will have the same z value as the buffer, because V is the last thing written to the z-buffer at that pixel (B has not been written to the z buffer yet). Care must be taken, therefore, to set the z clipfunction so that objects (pixels) with z values equal to the buffer's value are drawn. Otherwise, V would fail to be redrawn in this case.

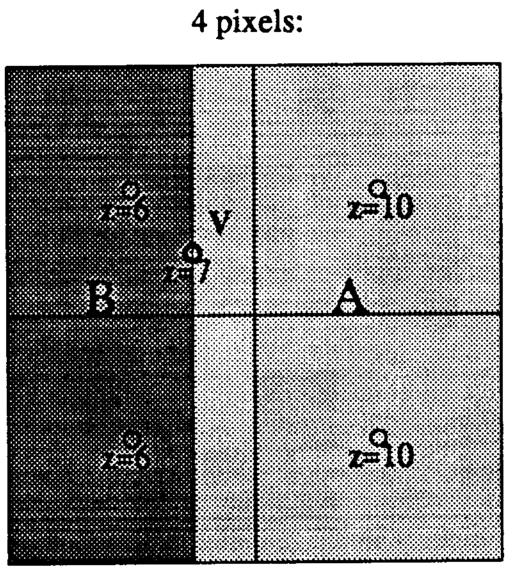


Figure 16: Two surfaces A and B sharing vertex V

## 4.4 Multiple redraws of hemi-cube

Another potential difficulty of this algorithm is that two vertices from the same surface may be rendered into the same pixel. In this case, one of the vertices will not appear in the final image, and therefore will be incorrectly marked as not visible. This problem, it turns out, can be solved by a slight modification to the algorithm. The modified algorithm looks like this:

```
Mark all vertices as not visible
repeat
```

```
    Render scene, only drawing vertices marked as not visible.
    Scan hemi-cube for vertices, mark them as visible.
until no new verties were marked
```

If the vertices U and V are part of the same surface, and lie on the same pixel, then the first time the scene is rendered, only one of them will appear in the hemi-cube. If U is the vertex that appears, then it will be marked as visible. Since at least one vertex was found in the hemi-cube, the scene will be rendered again. This time, however, U will not be rendered, because it is already known to be visible. Therefore, V will appear in the same pixel that U appeared on the previous pass. V will be marked as visible, and the scene is rendered again. This will continue until no more vertices are found. Typically this requires rerendering the scene three or four times.

## 4.5 Ray casting

Unfortunately, there are still some rare situations that are handled improperly by the vertex visibility algorithm. These remaining problems are the result of a hemi-cube of inadequate resolution.

The first problem is that a surface might occlude vertices near the edge of an adjacent surface. If vertex V is part of the interior of surface A and surface B is adjacent to A, and V is within half of a pixel of the edge of surface A (when rendered in the hemi-cube), then V might be incorrectly occluded by B (see figure 17).

The second problem is similar, and occurs at shadow boundaries. In this case, vertices within half of a pixel of the shadow boundary may be incorrectly analyzed. In figure 18, surface A covers all nine pixels and contains vertices V1 and V2. Surface B is closer, and occludes five of the nine pixels. Both V1 and V2 are incorrectly analyzed in this case. V1 should be visible, but it lands on a pixel covered by B, and is marked as occluded. Conversely, V2 should be occluded, but it lands on a pixel covered by A, and is marked as visible.

These problems only create noticeable artifacts when the size of an element is on the order of the size of a pixel or smaller. Therefore, when one vertex belonging to such an element is visible and another is not, the vertices need to be more carefully analyzed.

One way to more carefully analyze these vertices is through oversampling. Oversampling can be accomplished by jittering the image by a variety of sub-pixel positions. The visibililty information from these different images

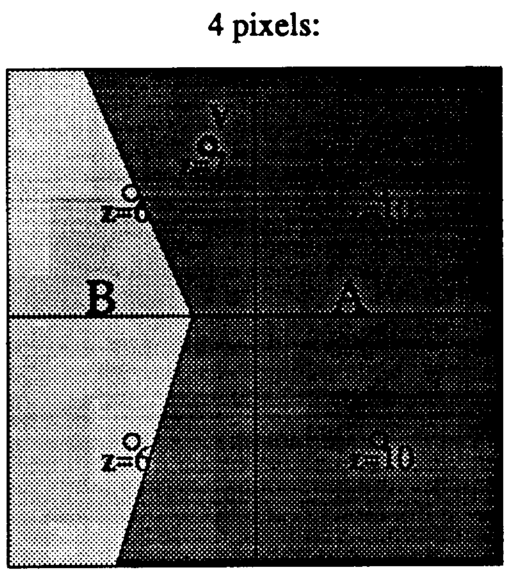


Figure 17: V lands on a pixel owned by B and is incorrectly occluded

can then be accumulated and analyzed. This approach has the disadvantage that a large number of images need to be generated just to analyze a few vertices in the scene; this takes an unnecessarily long time.

Therefore, it makes sense to apply a ray casting algorithm to the few vertices in the scene that are questionable. Because the number of vertices that need special attention is generally low, it is much faster to analyze each one individually than it is to apply a global vertex visibility algorithm multiple times. This analysis consists of casting rays from the center of the light to the questionable vertices. The vertices are then marked as visible if the ray hits them.

## 5 Light shooting

The core of a radiosity algorithm centers around energy distribution. In a progressive radiosity algorithm, the key steps are finding the brightest patch in the scene and distributing its energy. This process is called "light shooting", as the primary or secondary light sources shoot their energy to the rest of the scene.

9 pixels:

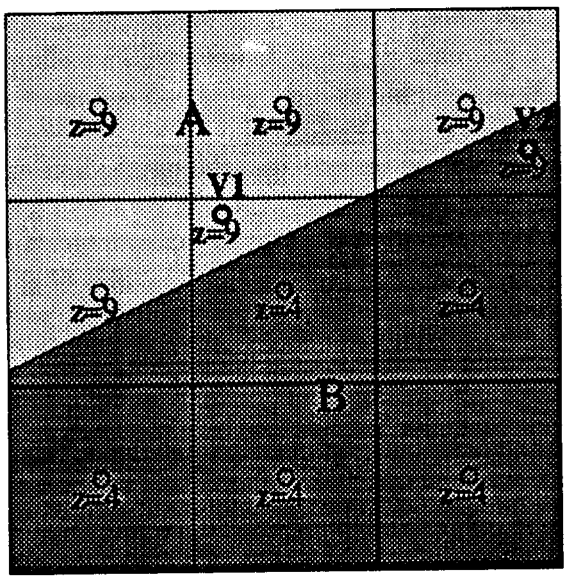


Figure 18: Surface A covers all 9 pixels, and contains V1 and V2. Surface B occludes 5 pixels. V1 is incorrectly determined to be occluded, and V2 is incorrectly determined to be visible.

## 5.1 Light energy calculation

The first step in this process it to calculate the amount of energy that each patch has to distribute. This energy comes from two sources. There is emitted energy and reflected energy. The emitted energy is easy to calculate, since it is defined in the input file as energy per unit area. This number can be multiplied by the area of the patch (for each of red, green, and blue), to determine the emitted energy.

It is also easy to calculate the reflected energy per unit area at each of the vertices. From previous iterations, the energy per unit area received at each vertex has been calculated. Also, the percent of that energy that is reflected back into the environment is known from the material description. Thus, the energy per unit area that is reflected at each vertex location can be calculated. However, the amount of energy reflected by the patch as a whole is still not known.

To compute the energy reflected by the patch, first the energy reflected by each of its elements must be computed. A reasonable way to do this is to define the average energy of an element as the average energy of the vertices that define it. Since all elements are either triangles or quadrilaterals, there will only be three of four vertices to average. This average can then be multiplied by the area of the element to calculate the total energy for that element. Summing the energy of all elements from the patch provides an estimate for the total energy of the patch. Under this scheme, vertices that are part of large elements are weighed heavily, whereas vertices that are only part of small elements are weighed lightly.

After calculating the total energy that a patch has to distribute to the environment, the energy that has been distributed on previous iterations is subtracted out. This determines the differential energy to be distributed in the current iteration.

## 5.2 Bright light selection

Calculating the amount of energy that each patch in the entire scene has to distribute takes a long time, and to do this on every iteration would take too much time. Therefore, the brightest twenty patches are found and they are shot over the next twenty iterations. Since the act of shooting one patch may change the energy of other patches, a patch's energy is recalculated just before it is shot. Also, note that because the energies of the patches may change, the brightest patch might not always be the one shot. This may cause the algorithm to take slightly longer to converge.

## 5.3 Light subdivision

When energy is shot from a patch, it is distributed to all vertices that are visible from the center of the patch. The problem with this is that a vertex might be visible to only half of a light source, and so it should receive energy from only half of the light. This effect creates the penumbra region at the edge of a shadow.

There are a number of ways to create this penumbra region. The method employed here subdivides a light source into a number of smaller lights and then shoots all of these smaller lights separately. The subdivision is performed dynamically through the element meshing library.

Before subdividing, a light source needs to be analyzed to determine if subdivision is necessary, and if so, how much is required. Unfortunately, this analysis depends both upon how far away the shadowed surface is, as well as upon how far away the occluding surface is (the one that casts the shadow). The problem here is that the central algorithm shoots light to the entire scene at once, rather than to a patch at a time. Therefore, the light source needs to be subdivided equally for all receiving patches.

This subdivision is based upon two global numbers. The first number tells what minimum amount of energy can be shot from a patch without creating a significant penumbra. The second number tells how small a bright light needs to be so that it will not create a significant penumbra. A patch is then subdivided until the elements created have too little energy, or are too small to create a significant penumbra.

## 5.4 Analytic energy distribution

After determining which vertices in the scene are visible to the light source, the energy from the light source is distributed to these vertices. This distribution is based upon a form factor calculation that computes the density of energy that is transferred from a triangular or quadrilateral light source to a vertex in the scene. The light source is assumed to be uniform in brightness, and the receiving vertex needs to have a surface normal associated with it.

The form factor calculation that most radiosity algorithms use is an approximation that assumes the light source is a disk. An exact solution can be obtained with the form factor calculation based upon a closed contour integral presented by Baum and Winget [3]. As accuracy is a large concern and this integral calculation is not too expensive for triangles and quadrilaterals, it is used in the algorithm.

This integral calculates exactly what energy density should be distributed to the vertices of the scene and it has worked extremely well in almost all cases. However, in the case where a vertex lies along the edge of a light source, this integral will fail to produce the desired answer. The reason for this is that the integral exhibits a discontinuity at points that lie on the light itself. If the vertex is moved an infinitesimal distance behind the light source, it will receive no energy. If, instead, it is moved an infinitesimal distance in front of the light source, it will receive a significant amount of energy. Figure 19 is a graph showing the relation between the location of the vertex, and the energy received.

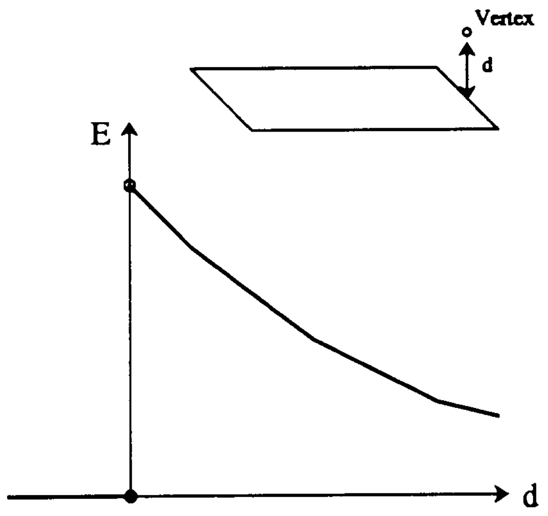


Figure 19: Light to vertex energy transfer function shows a discontinuity at d=0

The problem with this occurs at corners. If patches A and B are adjacent and form a concave 90 degree corner at their shared edge, none of the energy emitted from A will reach B at that shared edge as the integral suddenly drops off to zero there. Thus that edge will appear to be darker than it should be.

In order to solve this problem, the vertex is temporarily moved a very short distance in front of the light source. This distance is calculated based upon the size of the light source, to insure that this approximation is reasonably accurate.

It is also possible to solve this problem analytically, by directly calculating what the energy transfer function should be at d=0. The numerical solution, however, is much easier and faster to implement.

## 6 Gradient Subdivision

In order to capture the complicated lighting effects in the scene, any element that exhibits an energy gradient across one of its edges that is larger than some threshold (the "subdivision gradient") is subdivided. This subdivision forces the mesh to be more detailed where necessary.

### 6.1 Subdivision Criteria

As stated, this algorithm would subdivide along sharp shadows indefinitely. No matter how finely the elements were subdivided, the elements that fall on the shadow boundary would still exhibit high gradients. For this reason, another stopping criterion was devised. This criterion is based upon the size of the element; if the element is too small, then it is no longer subdivided. Initially, this stopping criterion was based upon the area of the element. If the element covered a sufficiently small area, then it would not be subdivided. The problem here, however, is that long skinny triangles were never subdivided because they covered very little area. Instead, it is better to stop the subdivision once the sampling points are close together. Therefore, a criterion was chosen that achieved this goal: if an element's longest edge is sufficiently short, then the element is no longer subdivided.

### 6.2 Subdivision Implementation

There is one difficulty that makes this subdivision extremely tricky. Each level of subdivision introduces new vertices to the scene, and it is difficult to determine what energy density to assign to these vertices. For example, let U and V be two adjacent vertices in the scene, with energy densities of 1 and 2 respectively (in the real program, energy densities are expressed in terms of red, green, and blue). Now, suppose that a nearby light is "shot", and V is visible to the light, but U is not. The new energy densities for U and V might be 1 and 4 respectively (see figure 20). Also, suppose that the subdivision heuristic decides that the edge containing U and V should be split, and a new vertex W is created between U and V. The difficulty lies in determining what energy density to assign to W.

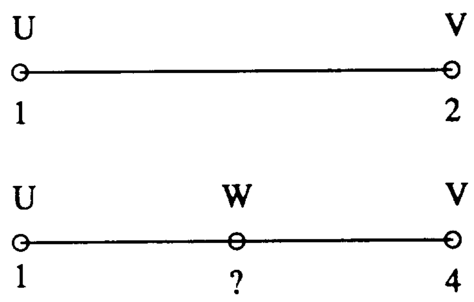


Figure 20: What value should be given to W?

The theoretically correct way to calculate the energy density at W is to reshoot all previous lights at W. That is, if 20 lights have already been shot, then those 20 lights need to be shot at W as well. However, this approach would be slow, as it would require shooting rays from each of the 20 light sources to all newly created vertices in the scene.

Instead, the energy density at W can be estimated by averaging the densities of U and V from before the current light was shot. In this example, that would give a value of 1.5. Then, the current light is reshot at all the new vertices (just W in this case), to accurately add in the energy contributed by the current light. Assuming that W is visible to the current light, its final energy density will probably be about 3.5.

Reshooting the current light at all new vertices is expensive, but it was the current light that created the high gradient across the edge from U to V, so it is probably worth paying this cost. Additionally, if the energy density between U and V is monotonic, then the maximum error of this estimate is guaranteed to be less than one half of the subdivision gradient. This can be seen as follows: First, let us assume (without loss of generality) that the energy density at U is less than the density at V. Furthermore, the gradient of $V - U$ is less than the subdivision gradient (or it would have already been subdivided). Based upon the assumption that the energy density from U to V has no local maximum or minimum, the midpoint W must have an energy density that falls between the energy densities of U and V. Since the estimate for the density at W is $(U + V)/2$, then the maximum possible error is $|(U + V)/2 - U|$ or $|(U + V)/2 - V|$ which is $(V - U)/2$, which is less than one half of the subdivision gradient. Note, however, that an error of this magnitude might be introduced at every level of subdivision.

## 7 Results

To test the accuracy and robustness of our radiosity algorithm, we generated radiosity-based renderings for two different architectural models.

The first model is the house of Dr. Fred Brooks of UNC Chapel Hill. The model was created using AutoCAD by UNC graduate students: Harry Marples, Machael Zaretsky, John Alspaugh, and Amitabh Varshney. The solution was performed on the entire house; the resulting mesh after adaptive refinement contains 420,026 elements. An overhead view of the house (with the roof removed) is shown in color plate 1. A view of the piano room is shown in color plate 2a, and the corresponding mesh is shown in color plate 2b.

The other model was designed by Mark Mack Architects for a proposed theater near Candlestick Park in San Francisco. The theater model was build using GDS software by Carles Ehrlich from the Dept. of Architecture, UC Berkeley. Two views of the theater are shown in color plates 3a and 3b. In color plate 3a note the shadows cast by the rungs of the catwalk onto the adjoining catwalk frame. Timings and statistics for all three models are summarized figure 21.

| | Brooks' House | Candlestick Theater |
|---|---|---|
| # of input surfaces | 8,623 | 5,276 |
| # of patches | 68,186 | 78,094 |
| # of elements | 420,026 | 1,061,542 |
| Grouper time | 2:47 min | 2:50 min |
| Intersection time | 2:23 min | 4:01 min |
| Patch mesh time | 6:30 min | 25:53 min |
| Time per iteration | 2:16 min | 7:12 min |

Figure 21: Program Timings and Statistics (all timings done on a single processor IRIS 4D/310 GTX)

As can be seen, this radiosity algorithm has been successfully used to generate realistic images. However, one flaw still exists. Gradient subdivision will not detect a shadow unless it covers at least one vertex. Thus, small or thin shadows that do not occlude any vertices from the initial element mesh will not be captured properly.

Despite this flaw, we have been very pleased with the images produced by this algorithm.

## 8 Conclusion

In order to make radiosity a more practical technique for generating images, we have tried to make it faster, more accurate, and easier to use. Speed has been achieved through the use of a progressive radiosity algorithm and specialized graphics hardware. Accuracy is maintained through a patch/element hierarchy and adaptive subdivision. Lastly, the models are automatically preprocessed by a serious of filters to guarantee that they are geometrically and topologically suited for radiosity. This preprocessing insures that the desired accuracy can be obtained, and it also allows the radiosity pipeline to be applied to models generated by typical CAD drafting packages. Hopefully, by bringing these ideas to the architects who might use them, we can make radiosity a useful, practical method for interactively viewing buildings which have not yet been built.

## Acknowledgments

This project was conceived, designed, and finished at Silicon Graphics Incorporated in Mountain View, California. Special thanks go to those people whose help was critical to the success of this project: Stephen Mann for writing the element and patch meshing code, Mark Segal for maintaining the polygon intersection code, Jim Winget for demanding perfection in all things, and Dan Baum for the original ideas and funding.

## References

[1] K. Akeley. The hidden charms of z-buffer. *IRIS Universe*, pages 31–37, 1990.

[2] R.E. Bank, A.H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, pages 3–17, 1983.

[3] D.R. Baum, H.E. Rushmeier, and J.M. Winget. Improving radiosity through use of analytically determined form factors. *Computer Graphics (Proc. SIGGRAPH '89)*, 23(3):325–334, July 1989.

[4] M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics (Proc. SIGGRAPH '88)*, 22(4):75–84, August 1988.

[5] M.F. Cohen and D.P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):31–40, July 1985.

[6] M.F. Cohen, D.P. Greenberg, D.S. Immel, and P.J. Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications*, pages 26–35, March 1986.

[7] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics (Proc. SIGGRAPH '84)*, 18(3):213–222, July 1984.

[8] A.T. Campbell III and D.S. Fussell. Adaptive mesh generation for global diffuse illumination. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):155–164, August 1990.

[9] A. Kela. *Automatic Finite Element Mesh Generation and Self-Adaptive Incremental Analysis Through Geometric Modeling.* PhD thesis, Unversity of Rochester, 1987.

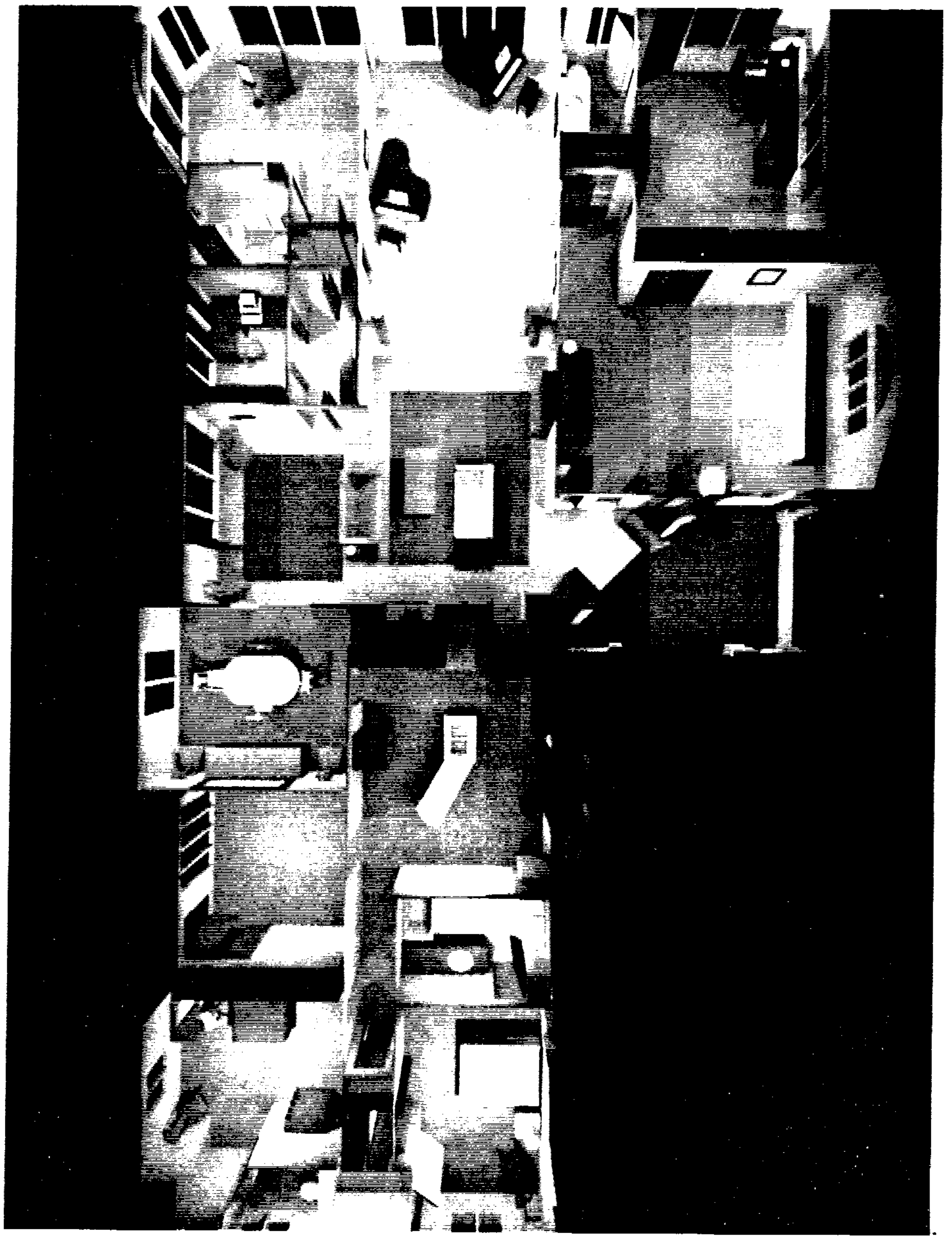
[10] D. Khorramabadi. A walk through the planned CS building. Master's thesis, 1991.

[11] M. Segal. Using tolerances to guarantee valid polyhedral modeling results. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):105–114, August 1990.

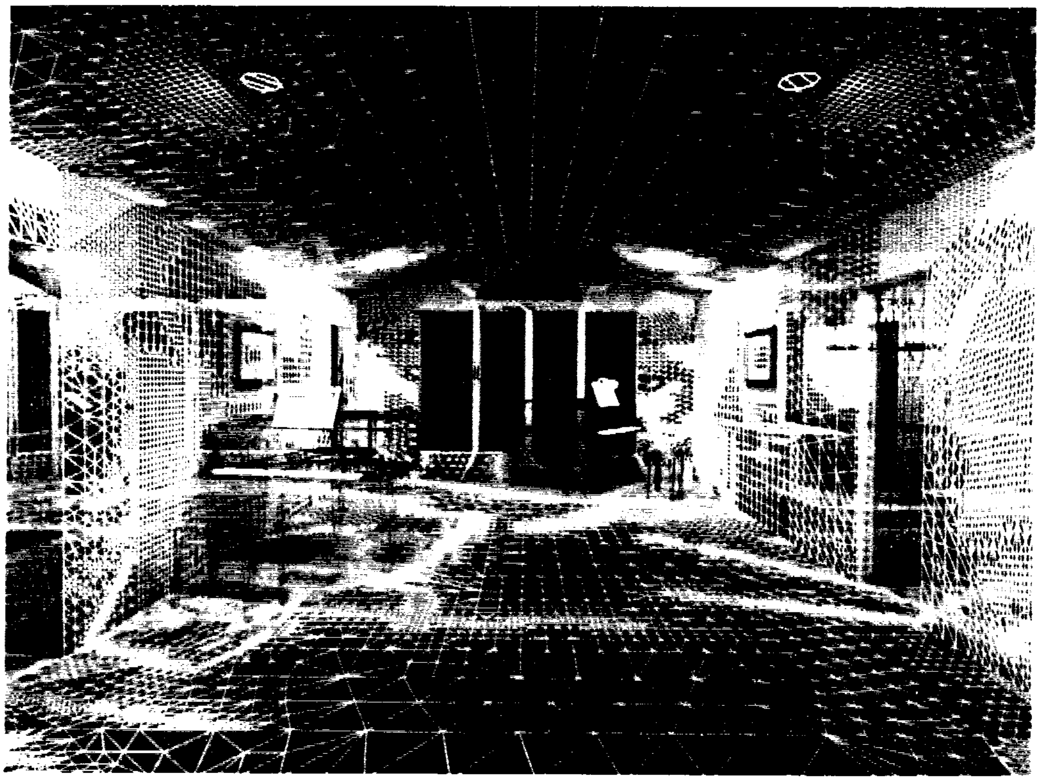
[12] M. Segal and C.H. Séquin . Partitioning polyhedral objects into non-intersecting parts. *IEEE Computer Graphics and Applications*, 8(1):53–67, January 1988.

[13] C.H. Séquin and K.P. Smith. Introduction to the Berkeley UniGrafix tools (version 3.0). Technical Report 606, Computer Science Department, UC Berkeley, 1991.

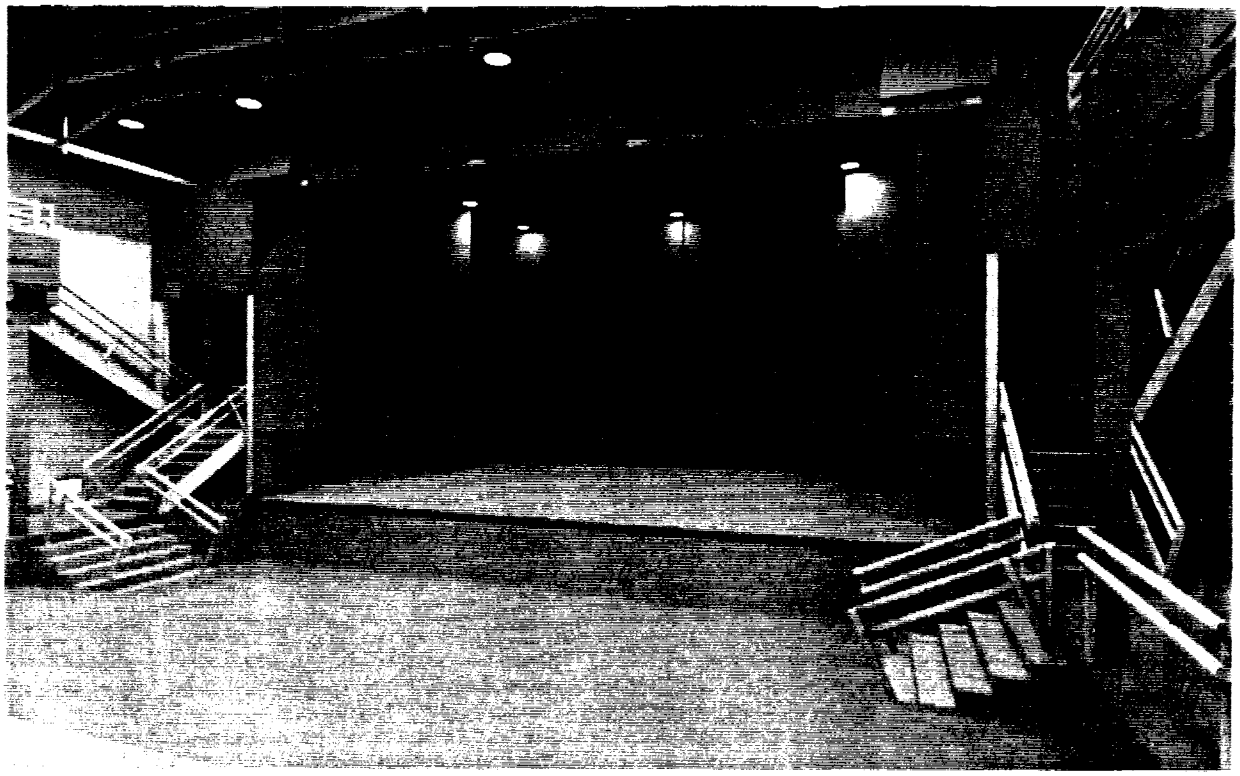
[14] J.R. Wallace, K.E. Elmquist, and E.A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics (Proc. SIGGRAPH '89)*, 23(3):315–324, July 1989.



Color Plate 1



Color Plates 2a and 2b



Color Plates 3a and 3b