Radiosity: A Study in Photorealistic Rendering

by

Christopher De Angelis

Undergraduate HonorsThesis
Computer Science Department – Boston College
Submitted May 1, 1998

Introduction

For thousands of years, man has had a fascination with visually depicting the world in which he lives. From cave paintings to the startlingly real paintings of the High Renaissance, there is no question that this hobby has been preserved and continually reinterpreted throughout the history of mankind. Now we are at a point where man is experimenting in depicting not just with the tools of traditional artistry but with the aid of the computer, and we are finding some fascinating results.

Between all the methods of computer graphics - which is essentially a marriage of mathematics and computer hardware - one of the most compelling yet perhaps most overlooked areas is that of radiosity. I find it compelling because of its amazing ability to generate images that look nearly true to life. Although you don't hear much about radiosity unless you read the technical journals of computer graphics, such as the Computer Graphics publication of the ACM SIGGRAPH group, there has been a fair amount of research and development in the academic circle. This being the case, my patient advisor, William Ames, encouraged me to start the process of understanding just what radiosity is and how it works by using the original sources rather than latter reformulations. Although I was often frustrated by the progress of my work, I believe this ended up being a solid approach for understanding the method and then implementing a radiosity renderer of my own. This paper serves to summarize my findings and efforts with a fair amount of detail and to give some insights as to why radiosity is fascinating but has not generated all that much in the way of widespread development efforts.

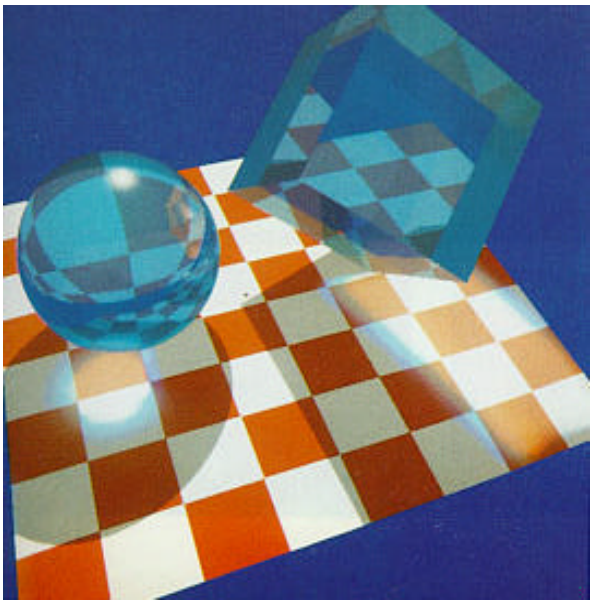
I. Radiosity: What is it?

1.1: General Description and Usefulness

The first thing to realize about radiosity is that it is not an algorithm for rendering *interactive* 3-D graphics like those in modern computer games. The ever-present tradeoff of accuracy vs. speed still prohibits the use of realistic graphics in games on today's hardware - at least in real-time computations. Instead of interactive applications, the camp of rendering that radiosity does belong to is that of *photo-realism*, that is, the quest to generate images on the computer that are very similar to photographs.

Within the category of photorealistic rendering, the two main approaches are ray tracing and radiosity, ray tracing being the more common one. (It is valuable to talk about ray tracing here since the discussion will provide a frame of reference for the pros and cons of radiosity as a method in the photorealism camp.) Images generated by ray tracing are often used in entertainment and advertising industries. For example, ray tracing is the algorithm that Pixar employed to generate the highly successful Disney film "Toy Story" - the world's first feature length film to be generated entirely by a computer graphics rendering algorithm. Outside the commercial realm, freely distributable versions of ray tracers are readily available to computer graphics enthusiasts, such as Persistence of Vision (POV-RAY). Ray tracing tends to produce images that look rather glossy and polished - often nearly too polished. This is because ray tracing works by modeling the *objeccts* of the scene and is especially

concerned with scene geometry. More specifically, the nature of ray tracing involves casting simulated rays of light from a single viewpoint in 3-dimensional space through every pixel location on the screen and is dependent on information regarding object intersections before doing any image rendering. This process results in excellent representations of specular reflections and refraction effects (a specular light comes from a particular direction and is reflected sharply towards a particular direction; specular light causes a bright spot on the surface it shines upon, referred to as the specular highlight. One example would be the shiny spot on a balloon directly in the path of a light source). However, due to the fact that ray tracing renders scenes based on the definite location of the observer, it is what is known as a *view-dependent* algorithm. The consequence of being a view-dependent algorithm is that in order to be viewed from a different vantage-point, the entire scene must be re-rendered. In addition, in order to better simulate the effect of a particular object contributing its illumination towards all other objects in the scene, a directionless ambient lighting term must be arbitrarily added into the scene. This consequence makes ray tracing what is called a *local illumination* algorithm, that is, one where the light travels directly from the source to the surface being illuminated - in other words, direct illumination.

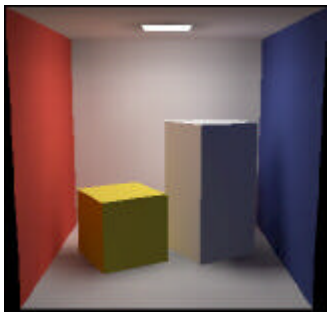


(image 1 - a ray traced image showing specular reflection & refraction - by J. Arvo)

In contrast, the goal of radiosity is not to model the objects of the environment, but rather the *lighting* of the environment. As we will see in the detailed description, it spends a significant amount of time dealing with issues of scene geometry, but the purpose of this is to make many minute light calculations which combine to describe the entire scene. The information gathered in this step - referred to as the form factor determination - is computed entirely independent of surface properties of each entity or patch in the scene. The result of this form factor computation is a description of the amount of light from a particular patch reaching every other patch in the scene. Because this object-to-object lighting information is a key part of the rendering process here, radiosity is categorized as a *global illumination* algorithm; that is, the interaction of light amongst all objects in the scene is considered for total illumination in the scene, eliminating the need for adding an arbitrary, ambient lighting term. Since the

information from the form factor computation is then used in combination with the surface property data to produce an image of the scene without any other light source information, the radiosity method generates images that accurately model diffuse reflections, color bleeding effects, and realistic shadows (diffuse lighting comes from a particular direction but is reflected evenly off a surface, so although you observe brighter highlights when the light is pointed directly at a surface rather than hitting the surface at an angle, the light gets evenly distributed in all directions; fluorescent lighting is a good example of diffuse lighting). It turns out that these are the types of effects we observe frequently in every day living. In addition, since radiosity models its scenes using information inherent to the scene geometry and surface properties alone, it is a *view-independent* algorithm, meaning that that the vantage-point of the observer is irrelevant to the rendering process. The consequence here is that after the costly computation of the form factors is completed once, the viewer's vantage-point can be changed to any location pretty quickly. This presents the possibility of a nearly real-time walkthrough of a scene that has been computed via radiosity. So as in the case of ray-tracing, you cannot produce an animation of objects in the scene without recomputing the entire scene image from scratch, but you can use the advantages of view-independence to create a pretty fast walkthrough of a static scene using a typical home computer of the 1990's.

One industry that is using radiosity to create walkthroughs is that of architectural and interior design. The radiosity method is proving itself useful in applications of virtual tours of buildings designed but not yet built by architectural and interior design firms; this would be advantageous for a customer who is not quite sure about committing to a particular job or is having trouble deciding between several options. An example of a still done by the architectural firm Hellmuth, Obata and Kassabaum, Inc., is shown below in figure 3. So in the sense that you can demonstrate what a particular designed or imagined environment might look like, radiosity can be used as a simulation tool. Though in the sense that it does not allow you to dynamically change the scene, it isn't a true tool for simulation. But until computers become powerful enough to entirely recompute scenes via radiosity in real-time, this is the price of such accuracy. Although some video games claim to use radiosity in their graphics engines, what they must really mean is that they use some ideas borrowed from radiosity rendering theory as no interactive game could possibly produce scenes generated by radiosity in real-time with hardware such as it is at the time of this writing. But one day this too will change.



(image 2 - an image generated by Ian Ashdown's radiosity program "HELIOS", demonstrating diffuse reflections & color bleeding - notice these effects on the walls and the objects in the center of the scene, respectively. This image was also generated using anti-aliasing.)

(image 3 - a radiosity-generated image of the New York Hospital modelled by the architectural firm Hellmuth, Obata, and Kassabaum, Inc.)

1.2: The Original Radiosity Algorithm

An exhaustive explanation of the code of a radiosity based renderer would not be helpful without understanding the details of the theory by itself. In fact, a detailed, line-by-line description of a radiosity implementation would be too tedious to read. My intent, therefore, is to give a summary of the principles behind the method of radiosity and then discuss non-trivial details of my implementation and implementations in general.

Radiosity was first introduced to the computer graphics community by a group of researchers at Cornell University in 1984. It is based on principles of radiation heat transfer useful in calculating heat exchanges between surfaces. These principles are basically concerned with calculating the total rate at which energy leaves a surface (*the quantity of radiosity itself*) - in terms of energy per unit time and per unit area - based on the properties of energy emission and reflectivity particular to a surface. To put it more succinctly, the radiosity of a surface is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. This is an interesting idea to apply to computer generated images as the human eye senses intensities of light; the intensity of light can be approximated well by summing the light emitted plus the light reflected. The observer only perceives a total energy flux and is unable to distinguish between the two components of light energy. But this is not quite all there is to computing radiosity. In order to calculate the total quantity of light reflected from a Lambertian (diffuse) surface, the surface reflectance must be multiplied by the quantity of incident light energy arriving at a surface from the other surfaces in the environment. Now the quantity of radiosity for a particular patch `(j)` can be formulated as shown below in figure 1:

$$B_j = E_j + \rho_j H_j$$

(figure 1 - the general formulation of radiosity by Goral, et. al.)

where `B` denotes the radiosity, `E` denotes the energy emission, rho (`p`) denotes the surface reflectance, and `H` denotes the incident light energy arriving at a surface from the other surfaces in the environment. Since `H` for a particular patch j is dependent on

the geometry of all other patches in the scene, it must be computed by essentially summing the fraction of energy leaving a patch `j` and arriving at patch `i` for *all* such patches `i` in the scene. This quantity of the fraction of energy leaving surface `j` and arriving at surface `i` is known as the *form factor*. But since the form factors are purely dependent on shape, size, position and orientation of the surfaces in the scene (the scene geometry), it is perhaps better to describe a form factor as a quantity describing the relative amount of energy that can be "seen" between a pair of patches. Naturally, if elements of the scene block the path of energy transfer between patches `i` and `j` within the scene, there will be no contribution of light energy from patch `i` onto patch `j` (note: the original paper by Goral, Terrance, Greenberg and Bataille is written assuming unoccluded environments, but the radiosity method is easily extended, as I have done, for handling complex environments where there is occlusion of patches by other patches). Now the radiosity equation can be expressed in its entirety as follows, with `F` denoting the form factor between patches `i` and `j`:

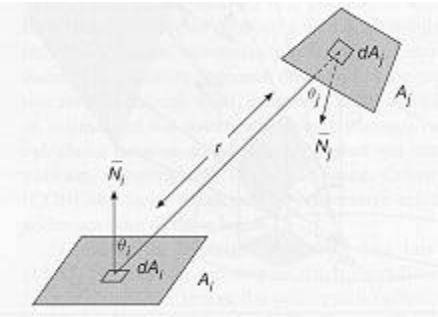
$$B_j = E_j + \rho_j \sum_{i=1}^{N} B_i F_{ij} \_ for \_ j = 1, N$$

(figure 2 - the full radiosity equation - Goral, et. al.)

The `B` values (radiosities) computed from this equation are essentially the intensities or brightnesses of each patch to be rendered in the scene. But all you have is a brightness - that is, there is no color information. So in order to display images of color instead of varying shades of gray, the radiosity values must be solved with different emissivity and reflectivity values for each of the three color bands used in computer graphics: red, blue and green. This is actually of little consequence in terms of computing time since the form factors themselves - which demand the vast majority of computing time - are not dependent on the surface properties. Finally, in order to gain the benefits of this light intensity information that was worked so hard for, interpolated, or, smooth shading must be performed between coplanar patches so that the scene doesn't just look like a bunch of patches of slightly different, uniform colors.

Before getting into the messy business of computing form factors - again, where the bulk of the time in a radiosity implementation is spent - first consider the assumptions that go along the above method. This method assumes that every surface in the scene is a convex polygon that can be subdivided into a mesh of elements, or, patches. As we will see later, a surprisingly significant part of a radiosity implementation is concerned with the meshing of surfaces in the scene so that the radiosity computations can be performed. Perhaps the most important assumption of all with radiosity in terms of the final result is the fact that all surfaces (and therefore all patches) are assumed to be ideal diffuse reflectors. Again, this means that they reflect light equally in all directions regardless of the angle of the incoming light rays. So as to not miss the forest for the trees, all of this means that radiosity is good for modeling large structures in many scenes encountered in everyday experiences, such as floors, ceilings, walls, furniture, etc., but it does not account for any specular reflection or refraction of light as seen in glass objects like soda bottles.

Now we turn to the tedium of the matter, the computation of the form factors. While I will leave the minutiae of the mathematical derivation to the authors of the original technical papers, a basic explanation of how the form factors are computed is in order. Recall that a form factor of a particular patch `i` in the scene is the fraction of radiant energy leaving patch `i` and arriving at all other patches `j` in the scene (I have only reversed the `i` and `j`). Realize that the form factor between a pair of patches might be written in array notation as `formfacts[i][j]`, but the form factor for patch i against all other patches `j` in the scene is a really summation of `j` numbers. Since this must be computed for all patches in the scene and not just one, there are really `i*j` form factors to be computed. So if there are a total of `n` patches in the scene, `n*n` computations must be performed in this step (and thus making this an $O(n^2)$ problem, but performance issues will be discussed later). The amount of energy transferred between two patches `i` and `j`, though, is not just a matter of the distance and angles forming a ray between patches `i` and `j` - it is highly dependent on the areas of the patches in question. Not only that, but since initially a patch within the scene is likely to be of a size manageable for human beings, it is not appropriate for accurate form factor computation as it would yield too coarse a brightness estimate for rendering an area of that size, and there could easily be a patch only partially occluding a pair of patches, resulting in further inaccuracies. So the problem of computing a form factor must be broken down in both patches in question by using smaller or *differential areas* and adding the sum over many steps. Figure 3 succinctly illustrates an iteration of the form factor computation between two patches:



(figure 3 - an illustration of the form factor computation)

For patches `i` and `j`, `N` denotes the surface normal of the patch, the angle theta denotes the angle between the surface normal and the line `r` between the points on each patch, `A` denotes the enitre area of the particular patch, and `dA` is the differential area of the patch. Since the form factors are being computed using a sum between many differential areas of the patch and this process is repeated for every patch in the scene, the approximation of the form factors is actually an equation involving a double area integral:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} dA_j * dA_i$$

(figure 4 - the double area integral form factor equation – Goral, et al.)

From this equation, it is clear that the angles between the surface normal and the line `r` are important in the form factor computation. Being the distance between the two differential patches, `r` is easily computed after doing tedious operations on the original patch coordinates, but it is not clear how these angles are computed from a first glance. Apparently the authors of the original paper assumed it was trivial to describe how to derive these angles, which I am computing using some vector algebra based on the surface normal and the vector between the two patches (direction depending on which patch is being referred to). Specifically:

```
                surface normal (N) • i-j vector (V)
cos theta =   ------—---------------------------------
              / length of N |  *  | length of V |
• = dot product
* = regular multiplication
```

A more efficient method of computing form factors than the equation shown in figure 4 can be realized without greatly changing the conceptual model of the computation. Although I originally tried using this form of the equation I was unsuccessful and not in a position to debug that implementation very well as I am not familiar with the math involved (see the implementation section for details). The math involved is an application of Stokes' theorem to transform the area integrals into contour integrals, which can apparently be computed in less time than the area integrals:

$$F_{ij} = \frac{1}{2\pi A_i} \oint_{C_i} \oint_{C_j} [\ln(r) dx_i dx_j + \ln(r) dy_i dy_j + \ln(r) dz_i dz_j]$$

(figure 5 - the contour integral form of the form factor equation – Goral, et.al.)

Finally, there are a few properties of form factors which prove to be useful in the implementation:

1. There is a reciprocity relationship for radiosity distributions which are diffuse and uniform over each surface:

   $A_i * F_{ij} = A_j * F_{ji}$

   This means that knowing F_{ij} also determines F_{ji}, saving significant, but linear computing time.

2. In a closed environment of `n` surfaces, there will be total conservation of energy, so all of the energy leaving a surface must be accounted for. One can check whether or not the implementation of a closed scene sums nearly to unity as it would if the computer didn't have to approximate the integral:

   $$\sum_{j=1}^{N} F_{ij} = 1 \_\_ for \_ i = 1, N$$

3. For a convex surface (one that does not see itself):

$$F_{ii} = 0$$

Once the form factors have been computed, a matrix which determines the intensities of each patch is formed and solved. Using the same notation as in figure 2 (but with lower case letters in some cases), the matrix for solving the radiosity equation for `n` patches is:

$$\begin{bmatrix} 1-\rho_1 F_{1,1} & -\rho_1 F_{1,2} & \cdots & -\rho_1 F_{1,N} \\ -\rho_2 F_{2,1} & 1-\rho_2 F_{2,2} & \cdots & -\rho_2 F_{2,N} \\ \vdots & & & \vdots \\ -\rho_N F_{N,1} & -\rho_N F_{N,2} & \cdots & 1-\rho_N F_{N,N} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}$$

(figure 6 – solution matrix for the radiosity equation)

The matrix can be solved using any standard method. (I solved it using Gaussian elimination with pivoting.) At this point, the general radiosity values of each have been found; for color images, this matrix is solved three times, each time using separate reflectivity and emissivity information for each of the three color bands (red, green and blue). Before rendering, the uniform, patchwise values are sent through an interpolating function that will smooth the values across patches on the same 2-dimensional plane, producing the desired diffuse and color bleeding effects. Once the scene is rendered, it can be displayed from any point of view without having to recompute the image. This is a tremendous plus for radiosity, and one that I have implemented in my program.

II. My Radiosity Implementation

2.1: Tools and Environments of Construction

My radiosity implementation was created using the C++ language and the OpenGL graphics library. C++ was not strictly needed as its object-oriented features were not employed in the program (e.g., a linked list of patches is constructed using pointers and the `struct` type of C), but it was helpful in defining multidimensional arrays of unknown bounds until run-time. While this can be done in C as well, the memory allocation style of C++ makes it clearer as to what exactly is going on in this process. The data maintained with the assistance of the C++ memory allocation style are the form factors and the matrix that is constructed and solved at the end of the computation to determine the radiosity values.

As to the graphics library, my original reason for using OpenGL had less to do with its rich features than the fact that I wanted the program to run with the maximum number of colors available. More specifically, the benefits of radiosity are apparent when interpolated (smooth) shading is being performed, which is a process that results in calculating small but noticeable differences in colors across coplanar patches. Since

I wanted to be able to fully see the fruits of this labor, I realized long ago that I would need the ability to display the rendered scene in a color depth of 24 bits per pixel (bpp), that is, the palette would have to contain a total of 16,777,216 color entries. This is the maximum number available on most displays and probably many more than the human eye is capable of distinguishing between. This being the case, I thought about what platform it would be practical to develop the program on, along with which graphics library to use. Here at Boston College, I was unsure of the capabilities of the video hardware in the Digital Alpha machines available to Computer Science majors, so I didn't consider the Alphas as a viable option. As much as I didn't want to buy anything I didn't already have, I figured that using my own machine with Linux might not work too well because I can never manage to get X Windows to display in 24 bpp color depth. So I decided I would be best off working in an environment where I was sure of what I could get in terms of video power, which meant using a Microsoft Windows operating system for the platform. I wound up using the Microsoft Visual C++ 5.0 development tools, which come with the libraries and header files for OpenGL (I ran this on Windows NT Workstation 4.0). Although I was at first hesitant to develop under Windows knowing that it takes about 100 lines of straight C code to produce a Window that says "Hello World", my task was made much easier thanks to the Auxiliary (AUX) library that comes with the Microsoft implementation of OpenGL. The AUX library, among other things, took care of all the tedium of making calls to the operating system to open a window - this may not sound like much, but under Windows it is. It also came in handy for doing keyboard input; the OpenGL specification makes no provision for input since it strives to be a platform independent library (the use of keyboard input will be discussed in section 2.6, "Walkthrough Ability"). These and other functions performed by AUXLib and the OpenGL calls allowed me to concentrate on the problem of radiosity itself rather than having to worry about the details of graphics programming under the operating system being used. And since OpenGL is so widely available, this program should run under any operating system with an ANSI C++ compiler and an OpenGL or compatible library. The Digital UNIX server owned by the Computer Science department at Boston College now has Mesa - an OpenGL compatible library - available for general use. Although the X Windows display, in most cases, does not show all the colors being used by the program and dithering gets performed, it is nice to see that simply by changing a couple of the header files, the program compiles and runs without a hitch.

The source code is about 1800 lines, or 50 kilobytes, and the executable file size on both Windows and UNIX machines is roughly 170 kilobytes. The only libraries linked besides the OpenGL libraries are `stdio`, `stdlib`, and `math`.

2.2: General Program Flow

The program must first read in all patch information from a scene file (see Appendix A for a description of the scene file) and construct a linked list of patches. The initialize routine performs file parsing, construction of a linked list of patches, subdivision of patches into smaller pieces (discussed in section 2.4, "Area Based Patch Subdivison") and allocation of arrays for storing the form factors, solution matrix, and radiosity values. Patch areas are also calculated by the initialization routine to reduce effort needed to prepare the scene file. Upon completing the initialize routine, the lengthy routine of computing the form factors is called (aspects of which are discussed

in section 2.3, "Form Factor Computation"). Once this lengthy computation is complete, the matrix is prepared and solved three times for the three color values for each patch, the radiosities are normalized so that all values are fractions of the largest radiosity value, and radiosity values are assigned to each patch structure and at the vertices (discussed in section 2.5, "Computation of Radiosity Values at Vertices for Smooth Shading"). This may sound like an awful lot of work, but it is actually much less demanding than the form factor computation.

None of the above is performed by OpenGL. OpenGL does, however, play an important role in this program by handling the tasks of performing the perspective projection transformations so as to get a better sense of depth, the viewpoint definition, and performing smooth shading on the patches to be rendered once the vertex radiosities have been computed. The automation of the smooth shading calculation in particular saves a significant effort in doing this manually. I consider myself fortunate to have done this project at a time when these "low-level" foundations of computer graphics are provided for in the graphics library, allowing more attention to be paid to the rendering algorithm itself.

2.3: Form Factor Computation

The first method of form factor computation I tried was the contour integral form as cited by Goral, et. al. My understanding is that this form of the equation is more efficient as the contour integral is an "edge integral" and can be computed without having to step through differential areas as with the area integral. But not being familiar with contour integrals beforehand, I didn't really know what I was doing. Even after getting some notion of how this equation should work from Professor Howard Straubing and reviewing the pseudocode of the computation in the original radiosity paper, I was still unable to get the computation to run successfully. The computation involves natural logarithms, and I was getting negative numbers from these logs when `r < 1` (see figure 5). This shouldn't have happened. After spending much effort trying to see what was going wrong using the DBX debugger on UNIX (which seemed appropriate since these were simply calculations for which I needed no graphical output), I finally gave up and decided to use the originally derived, double area integral form of the equation. Things went a lot better once I started using this equation; I got it working within a few weeks instead of spending two months debugging this phase unfruitfully. This area integral form is less efficient, but just as accurate as the contour integral form. Moreover, there is a problem using the contour integral form for which there is no obvious solution. When using this form of the equation, one must evaluate `ln(r),` where r is the distance between points on two patches, but when r goes to zero, there is some question as to how to deal with negative infinity. The paper by Goral, et. al. states that this occurs when two patches being used in the computation share an edge, and the integral was evaluated analytically in this case to both avoid this problem and imporve the accuracy of the integration. An analytical solution to the integral would take an additional amount of time to develop, and since I am not familiar with contour integral math to begin with, I would have to have someone else develop this solution completely independently.

No matter which of these two forms is used to compute the form factors, though, there is an important element to realize here. Both equations are double integrals, and

ordinarily they are solved iteratively rather than analytically. Solving them iteratively with the computer means that some kind of approximation will be done instead of getting a definitively accurate answer. So the bottom line is that the accuracy of the form factor computation depends on how finely divided the steps are between differential areas on each patch. The larger the distance between differential areas, the more coarse an estimate of energy coupling between two patches results. Likewise, the smaller the distance, the more accurate the resulting estimate. Naturally, there is a great difference in speed here as well, illustrating one of many recurring instances of the efficiency-accuracy tradeoff in problems of engineering and computer science. Statistics demonstrating this principle will be given in section 2.8, "Performance Issues".

The original paper by Goral, et. al. assumes that the scene being modelled has no occluded surfaces. Subsequent papers and books on radiosity say that not only does the original formulation not account for occluded surfaces but that the contour integral formulation cannot be extended for occlusion. However, I have been able to incorporate a routine into my implementation of the form factors computed by area integrals that checks for occluded surfaces and works with a fair degree of success. The routine I have included in my program adds a factor of `N` to the complexity to the run-time behavior and is based on ray tracing techniques for testing whether or not the line between the two points on each of the differential patches intersects with any other surface in the scene. But this tests only whether or not the line intersects with the infinite plane - not the actual patch. So if there is an intersection found with the infinite plane based on a particular patch's geometry, a test is then done to see the line actually intersects with the finite patch in question. For an arbitrary polygon, this is difficult to do - at least not without a knowledge of what the geometry of the patch is and branching to the appropriate code fragment for that polygon. In my case, though, since I am assuming only convex quadrilaterals and have actually developed scenes using only quadrilaterals that are rectangles, the intersection test is not very difficult. Actually, for non-quadrilaterals, an infinite plane intersection wouldn't do at all, but it can save some time for scenes with many quadrilateral patches. But there is an important subtelty here. In using the double area integral version of the form factor equation, an occluding patch is not truly occluding. It can act as *translucent* rather than opaque because light received by one side of the flat quadrilateral can be partially absorbed *and* partially reflected back into the environment on its other side! So in order to have a truly occluding patch between a pair of patches in my program, two patches must be stacked adjacent to each other and separated only by a small amount relative to the dimensions of the scene.

2.4: Area Based Patch Subdivision

An important piece of information for the routine that computes the form factors is the area of a particular patch. Originally, I was computing the areas of the patches in my scene while sketching the scene with vertices labelled on paper and encoding them into the scene file for input. But I realized that this was entirely too tedious if I wanted to try preparing more complex scenes or modifying existing scenes. The user should not have to do this calculation and put it into the scene file. So I had the program break the convex quadrilaterals into two triangles, compute the area of each triangle, and then add the two areas to determine a single patch area. This is a nice feature to have,

and I suspect most other radiosity implementations do some form of patch area calculation, provided the program is able to determine enough information about the geometry of the patch (if more than one patch geometry is being handled - not in my case though). But it still meant that every patch in the scene had to be calculated by hand. This is not such a great thing if one wants to be able to define more complex scenes without a lot of hassle. In addition, the benefits of radiosity can't be truly appreciated unless the patch size is sufficiently small because a few large patches would result in coarse images.

I solved this problem of having to define each and every patch manually by extending the usefulness of the patch area calculation. Since I had this funciton working, I decided to use it to figure out the size of a particular patch after it has been completely inputted and compared it to a threshold for the maximum allowable patch area. If the patch is less than or equal to this area, it remains in the linked list and the program continues creating the linked list of patches. However, if the patch area is larger than the threshold, the program figures out how many pieces the patch needs to be broken into, and that number of patches is created automatically using the vertex information from the patch read in. These patches are inserted into the linked list and the patch that was read in originally is deleted after this process is complete. This is another feature that makes a radiosity renderer a lot more useable by someone who wants to use it with a minimum amount of effort. It turns out that this is not just an enhancement that most programs implement, but that a lot of research in radiosity actually goes into the issue of patch subdivision. The problem of patch subdivision, or meshing, represents a significant part of the research done in radiosity techniques, even though it does not directly affect the radiosity equation itself. But its importance comes into play in trying to produce truly photorealistic images since it is desireable to have both small patches for accuracy and to reduce scene preparation effort simultaneously.

2.5: Computation of Radiosity Values at Vertices for Smooth Shading

Once the matrix for computing radiosity values has been solved (see figure 6), the patchwise radiosity values are obtained and the scene can be rendered. But if each quadrilateral patch is simply drawn with a different color, you wind up with just that - a scene that looks like a bunch of differently colored quadrilaterals. Despite the differences that may be observed across a number of patches looked at together, the effects of diffuse reflections, color bleeding, and shadows cannot be observed. This is because simply drawing the patches of different radiosities is done with "flat" shading, where a patch has a uniform color. What really needs to be done in order to see the benefits of the radiosity computation is rendering the patches using interpolated or "smooth" shading. In order to do this, the routine that performs the smooth shading must know the colors of each vertex of the polygon to be shaded and drawn. But the computation as originally described only produces radiosity values for patches, not for the vertices of each patch. So the problem of producing a photorealistic image with radiosity becomes a matter of being able to find the radiosity values of each vertex of a patch.

Of significant importance in solving this problem as I have done is an additional step in the portion of the initialization routine that subdivides a patch into smaller pieces

according to the maximum area threshold. With each patch node, there are four pointers for a patch's immediate neighbors sharing edges, i.e., a pointer to a patch located to the left, right, upward or downward of the patch in question. After a large patch has been completely subdivided into its smaller pieces, a second pass through all the subpatches is performed to determine whether or not a particular patch has these neighbors, and if it does, the pointer is assigned to the appropriate patch. If a patch does not need to be subdivided, then the pointers are simply set to null as the smooth shading is dependent on computing color values at the vertices using coplanar patches only. That is, if a patch is not broken down into smaller patches, it will not be coplanar with any other patch in the way I have designed my program and scene file format, so there is no need to set pointers to neighboring patches.

Much later in the program, after radiosity values for the patches have been computed and those values are explicitly assigned to the patches, a routine is called to determine set the radiosity values at the vertices of the quadrilateral patches. This routine must loop through all patches in the scene and do lookups for the patch colors of the three adjacent patches at all four vertices and for each of the three color bands (red, green and blue). This means there are 36 lookups done for each patch. For each patch vertex, the radiosity value for each neighboring patch is taken into account for a weighted average of the patch colors to find the specific "vertex radiosity"; again, this process is repeated three times at each vertex to get all required color information. If the neighboring patch in a particular direction does not exist, it is not taken account into the weighted average. All of this sounds like a lot of work, but it is not all that significant in terms of the overall run-time of the program. Once this is completed, OpenGL can render a series of smooth shaded quadrilaterals making up the scene after being told the colors and coordinates of the four patch vertices. Programming this required a lot of thought and effort once the radiosities were being computed correctly, but it was well worth it to see the images the program started producing. See Appendix B for color plates of flat and smooth shaded images of the same scene.

2.6: Walkthrough Ability

As mentioned earlier, one of the more interesting and useful outcomes of radiosity is that once the scene has been computed, the view position and view angle can be easily changed without having to recompute the whole scene. Only a change in the view matrix is computed before the patches are re-rendered to the scene in their new orientation. This allows the ability to navigate through the scene once it has been rendered. If double buffering is implemented, as I have done very easily with OpenGL, the walkthrough can be pretty smooth. OpenGL also takes care of the view matrix transformations via a function call. This function requires the new coordinates of the eye position, center position and the up vector (which remains the same). In my implementation, I have the AUX library set up to listen for keypresses corresponding to forward, backward, left, right, upward and backward motion. Upon a keypress, the eye coordinates are quickly recalculated using some basic trigonometry and then the OpenGL functions are called to do the view transformation and re-rendering of the scene. Of course, the more patches, the slower the redraw time will be, but navigating a scene can occur pretty quickly on most machines of today if not in "real-time".

2.7: Gamma Correction

Often in radiosity rendering, the image is computed and displayed quite accurately but pretty dimly. I suspect this has something to do with the fact that humans cannot easily anticipate the exact effects of surface emissivity and reflectivity on every other patch in the scene. That is, since radiosity so elaborately models the interaction light between diffuse surfaces, it is very much different from setting an object's inherent color and modeling it with point light sources and ambient lighting terms (as with ray tracing or approximations such as Phong shading) - a scenario that humans have a much better handle on for whatever reason. Because of all this, radiosity renderers may have a setting for gamma correction, which is a method for brightening all colors in a rendered image by a power of the gamma factor. The curve of an even (linear) distribution of colors applied to a gamma function results in a logarithmic curve, showing that the colors that are already bright become brighter by a small amount while the colors in the lower ranges are brought to about mid-range intensities. This is especially nice as what one may ordinarily perceive as uniformly black for some area can be brightened to show some subtle differences computed by the radiosity algorithm.

The formula I am using for the gamma correction is straightforward:

```
color = color ^ (1 / gamma_factor)
```

where this is applied three times for the red, green, and blue colors of each of the vertices of each patch to be smooth shaded.

2.8: Performance Issues

The double area integral equation for computing the form factors as seen in figure 4 represents a complexity of $O(n^2)$ as measuring the amount of energy transferred from a particular patch to all other patches - for every patch in the scene - requires n^2 computations. But the consequence of approximating this double area integral with the computer means that there is another factor k^2 of complexity for each of these n^2 computations. The reason is that even with a particular pair of patches for which the form factor is being computed, the area of both patches must be broken into some number of pieces (specified by an input paramter in the scene file) as the differential areas of each patch must be computed on in order to approximate this equation. So the complexity of the form factor computation then looks like $O(n^2 * k^2)$. Applying property 1 as detailed in section 1.2 divides this problem in half, but the overall complexity remains the same. With the addition of occlusion testing, the complexity factor becomes a whopping $O(n^3 * k^2)$. Since this is such a large factor, it is no wonder why people have been developing faster methods of solving the problem of radiosity. But any such attempts will almost undoubtedly lose a factor of accuracy since they all sacrifice some accuracy for speed. Pros and cons of the double contour area integral equation (shown in figure 5) have already been discussed, but as to the runtime behavior, it runs at $O(n^3 * k)$. The n^3 factor is exactly the same, but the power of k is reduced by one as the contour integral will only compute differential patch areas by stepping along the edges of the subpatches rather than stepping through a complete grid of a differential patch.

So just how bad is this form factor computation? Since I have separated the process of computing form factors from the rest of the program via a command line argument (with the exception of constructing the linked list of patches from the scene file), I have taken measurements of entire runs rather than a fancier method such as profiling runs of the program. This is possible because I have the portion that computes the form factors write the results to a file while the other portion just reads them in from the file. In running on a scene with 196 patches after subdivisions and with an integral step factor of 2 (where 1 means no step at all), the form factors took approximately 2 minutes and 15 seconds to compute; the time for the other portion of code to load the form factors, solve the matrix three times, compute vertex colors and display the scene was approximately 15 seconds. These measurements were taken on an Intel Pentium based machine running at 100 MHz. Based on these measurements, the time taken by the program as a whole to compute the form factors takes 90% of the total run-time. The parameters controlling the speed-accuracy tradeoff in this run were on the low to mid end of the spectrum. I have tried running the program after setting the parameters so that more accurate results could be obtained and have found that the form factor computation takes anywhere between 90 to 99 percent of the total run-time. Again, because it is such a crucial and time consuming step, much effort has been placed in coming up with more efficient ways to compute the form factors, but all such attempts are susceptible to quality loss.

In terms of memory usage, the most memory consumption is coming from the 2-D arrays of the form factors and the solution matrix. In the scene with 196 patches, this means that for a 2-D array of type double, assuming that double equates to a 64-bit data type, the memory required is:

```
196 * 196 * 64 bits / 8 bits per byte = 307328 bytes
```

So for 2 arrays of this size, 614 K of storage is required here. A rough estimate of the storage needed for a single patch in the linked list of patches is 150 bytes, so in our example this would require a total of 29,400 bytes to store. The storage required for each array of radiosity values is 1568 bytes, so for the 3 colors that would be 4704 bytes. So the only real critical elements for global data storage in terms of memory use are the two 2-D arrays, whose size is $O(N^2)$ where N is the number of patches.

III. Enhancements and Research Areas

3.1: Handling Other Surface Geometries

Ideally, an algorithm that strives for photorealism should not be limited to handling quadrilateral surfaces only. In ray tracing, one is limited to displaying objects for which methods of finding the point of intersection of a ray with the general equation of the object are known. With radiosity, the limitation is essentially the ability to subdivide the object into quadrilateral pieces or being able to compute the integrals of the form factor equation in a way other than the dual-axis/inner-outer `for` loop manner. In other words, computing form factors of arbitrary polygons is non-trivial. One is perhaps better off devising ways of computing the form factor equation with several other basic, convex polygons - such as triangles - and then attempting to subdivide an arbitrary polygon into a combination of the other polygons. Just as troublesome is the

problem of modelling inherently 3-dimensional objects, especially objects with curvature. It would be nice to be able to render spheres in a radiosity program. Perhaps one might borrow an algorithm from Phong or Gouraud shading to subdivide a sphere into a series of quadrilaterals. But based on pictures of scenes and their captions I have seen, spherical objects seem to be modelled using global illumination algorithms that are radiosity-derived but not strictly radiosity algorithms. Unfortunately, an investigation of such an algorithm requires a thorough understanding of radiosity to begin with, and since I spent so much time developing my radiosity implementation, I was unable to delve into these other areas in a purely conceptual level. A brief description of some of these methods will be discussed in section 3.5.

In the development of scenes for use with my radiosity program, I specified the geometry of cubes and rectangular prisms several times. In so doing, I found myself doing a lot of work to model a cube, which is really just 6 rectangles put together. It's simply tedious to do the manual work of determining the coordinates of each of face of the cube and orienting them correctly in the scene file. It seems that by sticking with the original radiosity method, one might continue by tackling the problem of automatically dividing a cube definition into six quadrilaterals for each of the face. This would probably not be too difficult. One way to do it might be to specify an object of type "cube" in the scene file, provide the vertices for one face of the cube and the three dimensions only. Then the program could be modified so that when it encounters an object of type "cube", it knows to construct the other faces using the information about the dimensions of the cube.

3.2: Imported Scene Files

Since there are already many types of file formats for specifying the geometry of objects within a scene, it only makes sense to consider the possibility of extending the program's usefulness by being able to parse these files so that they can be used for computations by the radiosity renderer. I considered doing this with my program for two different file types: the DXF and VRML formats. Although the DXF format was supposedly simple according to some sources, it seems rather complicated to me, and I chose not to spend my time coming up with a general DXF file parser. Besides, the issue of parsing the file by itself is not the same as being able to interpret the parsed information for use with radiosity, which is the issue I'm really after! I also considered parsing VRML files for several reasons. Not only is VRML (slowly) becoming more popular, but it's a much simpler format and there are programs available for converting the thousands or millions of DXF files out there to VRML format. It seems like I should be able to create a VRML file parser that takes and uses information about quadrilaterals without too much hassle, but I decided not to pursue it as most VRML files I've seen don't use the pure coordinate specification abilities alone. Most VRML files incorporate primitives such as spheres, cones, and cubes. So unless or until my program can subdivide all the primitives in the file format, I don't think it makes that much sense to try and partially support this other file type.

3.3: Modellers

An alternative to importing other file types would be to create my own modeller for the radiosity renderer. One of the first questions that may come to mind with the

issue of a custom modeller is, "There are so many modellers out there already - why go and create another one?" Although some file formats and modellers that generate their own file types may have notions of associating color properties with objects (such as diffuse, ambient, and specular in the case of VRML), most likely no modeller or file format has information specifically about the emissivity and reflectance properties inherent to a surface. Besides the ability to select object types and modify them on the screen with the mouse, a modeller designed for use with a radiosity renderer should also have some way of representing the surface properties, as well as the ability to store them internally and write them to a file. This could be done perhaps by just selecting a single color using an RGB or other color picker that gives a sense of relative brightness of a color. This inherent object color could be translated into the three reflectivity components, and then an optional emissivity might be chosen for one or multiple color bands and represented in the modeller by some kind of "glow" effect that gives a rough idea of the emission values of the object. Though the modeller could not give an accurate, if any, sense of the color blending or bleeding effects that would occur in the process of rendering via radiosity, one might at least get some sense of what the scene will look like before processing by the radiosity algorithm. I think a modeller designed to work in this fashion would be useful for radiosity not only by eliminating or reducing the tedium of manually specifying vertex coodinates, but by incorporating a sense of color information that corresponds with how radiosity works. I'm surprised that none of my reading or searching on the Internet has produced any talk of such a modeller existing for the radiosity programs that actually are being developed. It might have to do with the other radiosity-derived methods for global illumination, which currently seem more dominant, having color or brightness information treated more similarly to what existing modellers produce.

3.4: Distributed Computations

Since there is so much research and development going into methods and practices for distributed computing at the time of this writing, it only makes sense to consider how the run-time performance of radiosity might benefit by distributing the computations across multiple machines. It turns out that with the exception of making use of the property that the area of a patch `i` multiplied by the form factor from patch `i` to `j` is equivalent to patch `j`'s area multiplied by the form factor from patch `j` to `i` (i.e., $A_i * F_{ij} = A_j * F_{ji}$), the form factor computation lends itself to being computed across multiple machines pretty nicely. While more sophisticated methods of distributing the tasks performed by radiosity programs are being explored at the top R&D centers, it is quite conceivable to implement a system that will distribute the form factor computations across multiple average-powered computers of today. Only a procedure for splitting the task across some number of machines, recombining the results, and executing the whole procedure using socket programming is required.

I discussed and planned this with Chris Viens, fellow Boston College A&S Class of 1998 student, who wrote some code to distribute tasks among known machines on the same network as part of his senior year thesis. However, his system had the requirement that the code to run on each machine be written as a Java class. His code also had the convention of transmitting all data across network sockets as strings. I decided that it might be a fun experiment to try and port the initialiation and form factor routines of my program to Java and then see if we could combine our efforts to do

something really neat. Unfortunately, this effort had several problems on my side of things. First of all, Chris had to work with the machines and development tools available to him in the Computer Science Department of Boston College in the Spring 1998 semester, which meant that he was limited to using the 1.0 version of the Java language. This version has only cumbersome methods for doing the type of file parsing I needed to do, and since I had to use the same version of Java, I spent a fair amount of time having to figure out and debug the file parsing routine. Once this part was working successfully, I continued with the rest of the port and finally got it to compile and run. Except that it ran excruciatingly slowly, even for Java (simply because of the number of machine instructions that had to be interpreted from the Java bytecodes, I suppose). In a run with 118 patches in the scene, the Java program that constructed the linked list of patches and performed form factor computations took 11 minutes to compute the first 50 figures from the very first `i` patch to all the `j` patches. Based on this performance, I determined it would take nearly 26 minutes for one complete iteration of the form factor code to finish; this means computing the whole scene of 118 patches would take 3,063.28 minutes, or a whopping 51 hours. Because of the horrible run-time performance of this Java program, I decided to try it using a Just-In-Time compiler (JIT), which proved to be much faster once I got it to compile my code. Computation of the form factors for the same scene and accuracy parameters I used took less than 30 minutes - a vast improvement. However, the numbers were coming out completely wrong. Realizing that it would take me entirely too long to debug the Java version of the computations, make the remaining enhancements to the program and write this paper in the time remaining, I had to stop the distributed computing effort and turn to the more essential matters at hand.

3.5: Radiosity-Derived Methods

Because of the slow run-time performance of a radiosity renderer that is based on the original work in the field, much effort has been spent on developing ways of computing or estimating form factors more efficiently. It's safe to say that most of these other methods are designed to give up some degree of accuracy in order to gain speed, which is presumably more desireable for many applications if not in general. As mentioned earlier, while I cannot reasonably give a thorough overview of the more popular methods that have come into development and are currently being developed, I should at least mention what's out there. One algorithm that appears to use empirical ideas about global illumination and projection of graphics is called the hemi-cube algorithm. The hemi-cube algorithm is said to be pretty widely used, but has some significant resolution problems. Hence, much research and deveopment has gone into improving the accuracy of this algorithm. The hierarchical radiosity algorithm seems to be more concerned with the problem of surface meshing than form factor computation and is a popular topic among the technical journals of computer graphics. One particularly interesting method for giving the user a high degree of direct control of the accuracy vs. speed tradeoff is called progressive refinement. The goal of progressive refinement is for the user to specify a "convergence value" that the program iteratively works towards, displaying images that start out as coarse and get refined with the more iterations the program makes. An analogy here would perhaps be Newton's Method for approximating a square root. Other methods exist for global illumination but get farther away from the principles that are truly identified with radiosity.

Conclusion

Producing photorealistic images on the computer is not a challenge to sneeze at. Even with the power of today's home computers that was once only available in supercomputers by the likes of IBM and Sun, we still face the classic tradeoff of accuracy vs. speed. Perhaps this tradeoff will always exist with radiosity and photorealism in general, but one of the more intriguing aspects of radiosity is that the same principles that have been used for years in other fields of science and technology have been finding their way into the problem of producing very real looking graphics on the computer. They are making valuable contributions in solving this problem.

Given the complexity and performance issues in implementing radiosity as compared to other methods for generating photorealistic images, it isn't hard to see why it hasn't caught on like wildfire amongst computer enthusiasts. However, its ability to emulate real-world environments demands attention. It is encouraging that a lot of research and development is going towards radiosity and combined radiosity and ray tracing methods and in the academic circle. Perhaps it is only a matter of time before enough advances are made in this pursuit before radiosity methods or their derivatives are used in other industries or simulations besides the fields of architecture and design. Whatever the case, the important part here is that the emergence of radiosity - which has slowly been gaining popularity in the 1990's - provides an effective solution to the problem of creating realistic environments (though not necessarily the definitive one). Just because implementing the solution is difficult or requires a lot of effort does not eliminate its potential, as I believe I have demonstrated through my research, program, and this paper.

Bibliography

Ashdown, Ian. Radiosity: A Programmer's Perspective. (New York: John Wiley & Sons, Inc., 1994.)

Goral, Cindy M., Torrance, Kenneth E., Greenberg, Donald P., and Battaile, Bennett. "Modeling the Interaction of Light Between Diffuse Surfaces". ACM Computer Graphics (SIGGRAPH Proceedings, 1984). (New York: ACM Press, 1984) Vol. 18, No. 3.

Sillion, Francois X. and Pueche, Claude. Radiosity & Global Illumination. (San Francisco: Morgan Kaufmann Publishers, Inc., 1994)

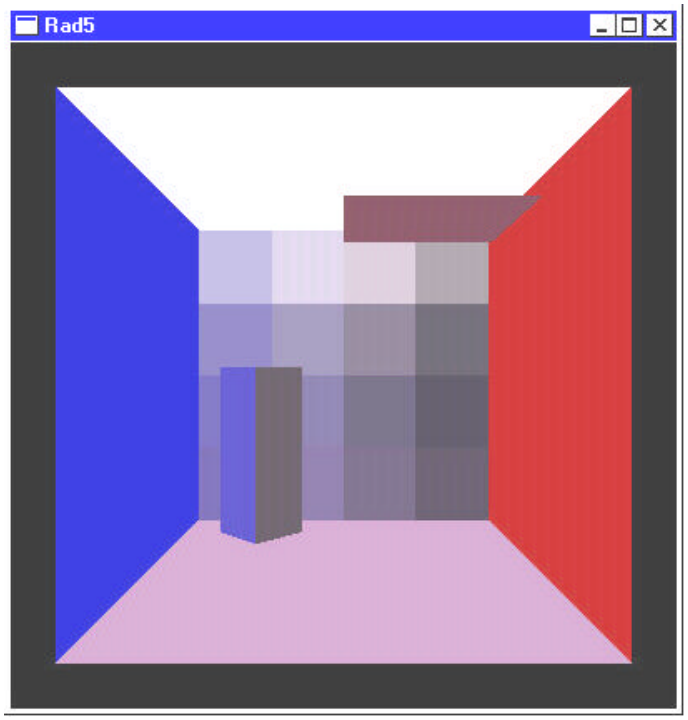
Watt, Alan and Watt, Mark. Advanced Animation and Rendering Techniques: Theory and Practice. (New York: ACM Press, Addison-Wesley Publishing Company, 1992).

Appendix A – Scene File Format

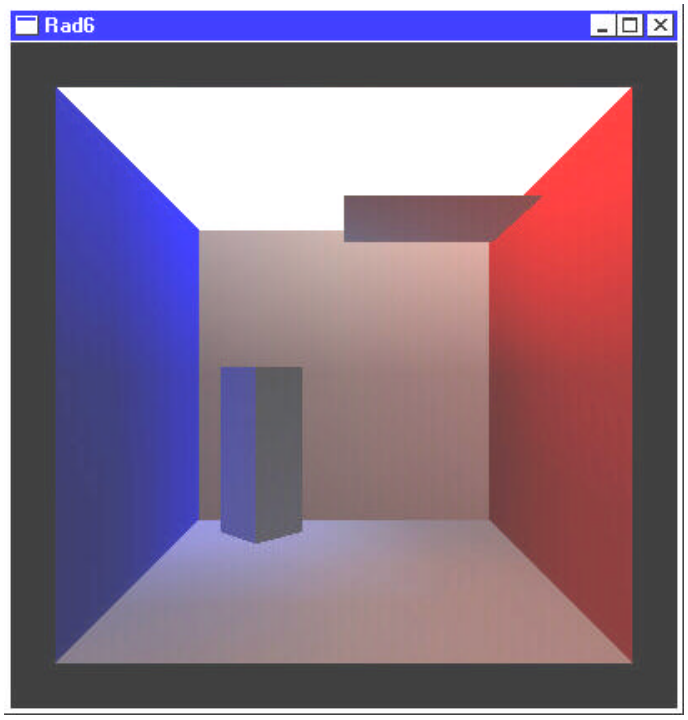
The input file has the following form:

```
MAX_PATCH_AREA (integer) //maximum patch area – meshing threshold
INTEGRAL_STEPS (integer) //# of steps along ea. axis of diff. patches
SCENEBEGIN               //specifies beginning of scene description
PATCHBEGIN               //specifies beginning of patch definition
#PATCH_NAME              //comment area – used to store patch name
VERTICES (integer)       //# of vertices in the patch
Er (double)              //Emission of red light
Eg (double)              //Emission of green light
Eb (double)              //Emission of blue light
Rr (double)              //Reflectivity of red light
Rg (double)              //Reflectivity of green light
Rb (double)              //Reflectivity of blue light
VERTEX0 (double) (double) (double)   //coordinates of vertex0 (x,y,z)
VERTEX1 (double) (double) (double)   //coordinates of vertex1 (x,y,z)
VERTEX2 (double) (double) (double)   //coordinates of vertex2 (x,y,z)
VERTEX3 (double) (double) (double)   //coordinates of vertex3 (x,y,z)
PATCHEND                 //specifies end of patch definition
PATCHBEGIN
…
…
PATCHEND
SCENEEND                 //specifies end of scene description
```

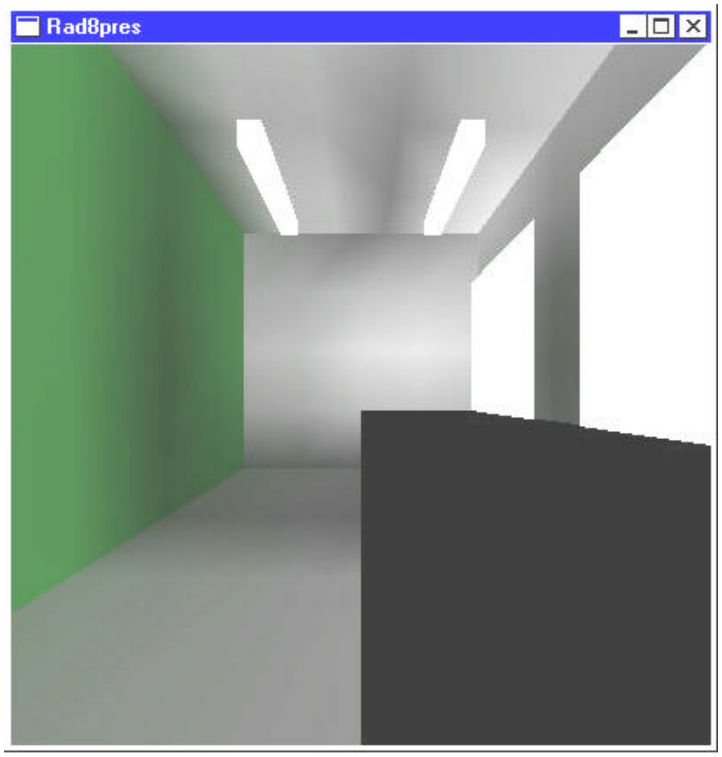
Appendix B – Color Plates



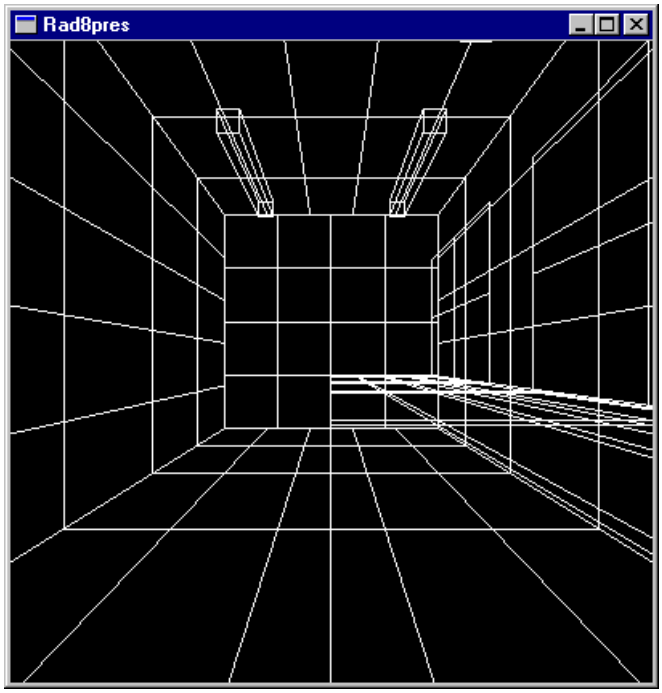
(color plate 1 – a test scene from my radiosity program – without smooth shading - computed correctly)



(color plate 2 – the same test scene computed by my program, rendered with smooth shading)



(color plate 3 – another scene computed by my program; the black quadrilateral was added for experimenting with occlusion)



(color plate 4 – the same scene from color plate 3 drawn in wireframe mode, showing the actual patches comprising the scene)

Appendix C – Program Listing

(attached)