# Assignment P4 : RADIOSITY

Computer Graphics 2

Due 25 Apr 1996

*Final version, 24 April 96*[1]

## Overview

For this assignment you will implement a radiosity program that reads scene files and displays pictures in an X Window. The algorithm we'll use is progressive radiosity with substructuring, as described in Foley's book, section 16.13.4. I'll describe the minimum requirements to get 100% below; if you want extra credit then you can go beyond this.

We're providing code to do much of the support work: reading in scene files, using ray tracing and bounding volumes to test visibility between points (from assignment P2), code to do z-buffer rendering into arrays in memory, and code to put up a picture with X windows. Support code is in *classdir*`/pub/src/p4`. You'll write code to subdivide polygons into elements, to shoot light from polygons (called "patches" in Foley) to elements (called "subpatches" in Foley), and to display intermediate results. In a progressive radiosity algorithm, it's convenient to display an approximate solution after each shooting step.

## Scene

The program will be limited to polygonal scenes where each polygon is either a diffuse reflector or diffuse emitter. Polygons in the scene files will be restricted to parallelograms, to simplify mesh generation (subdivision). These scene files will thus contain no specular or transmissive materials, and no spheres. Also, instead of (point) light sources, we use emissive polygons as light sources. This way, everything in the scene is polygons. If `oi` is an object of type `POLYGON` then the emission from the polygon is the color $\mathbf{e}_i =$ `(mi->kemission*mi->color.r, mi->kemission*mi->color.g, mi->kemission*mi->color.b)` where `mi = oi->material`. The reflectance of polygon `oi` is the color $\rho_i =$ `(mi->kdiffrefl*mi->color.r, mi->kdiffrefl*mi->color.g, mi->kdiffrefl*mi->color.b)`

Use the `scene_read` subroutine to read in the scene file. Call the supplied routine `precompute()` to precompute bounding volumes, which are used in the visibility tests later.

## Mesh Generation

Your main program should read command line argument(s) that allow you to set a maximum edge length `lenmax`. The recommended mesh generation technique is that you subdivide each parallelogram in the scene into a grid of congruent sub-parallelogram-shaped elements, each edge of which is smaller than `lenmax`.

You'll need to create the data structures for a grid of elements on a polygon. Let $nu$ and $nv$ be the number of elements in the $u$ and $v$ parameter directions, respectively. Note that the number of elements on a polygon is $nu \times nv$, while the number of element vertices of polygon $i$ is $n_i = (nu + 1) \times (nv + 1)$. The minimal information that you'll want to save with each polygon is $nu$ and $nv$, an unshot radiosity $\Delta\mathbf{b}_i$ (an RGB color) for the polygon, and a radiosity vector $\mathbf{b}_k$ (an RGB color) for each element **vertex** $k$ of this polygon. The positions of the element vertices and pointers to the element vertices making up each element could be saved, or could be computed on the fly.

I suggest you make the "outside" side of the polygon (the side from which the vertices are in counterclockwise order, and toward which the normal points) be emissive or reflective with the formulas given above,

[1] The only changes since 11 April were corrections of some file name suffixes from `.ppm` to `.tiff`.

and make the other side black. Otherwise you would need to shade both sides of the polygon independently, storing twice as many radiosity values.

## Shooting

We'll take the pseudocode on page 802 of Foley as our starting point, but modify it to get the best results we can with a simple algorithm.

Part of the shooting step involves computing form factors. Since the final picture will be displayed with Gouraud shading, it's most convenient to compute radiosities at the vertices of the elements rather than at the centers.

The form factor between shooting polygon $i$ and receiving element vertex $l$ could be calculated by tracing a single ray between the center of $i$ and point $l$, but this would lead to blocky shadows. The picture `pub/pix/p4/rtable_cheap.tiff` was computed as described above, and its shadows look terrible. The picture `pub/pix/p4/rtable_fine.tiff` was computed with the form factor formula described below, with the same mesh, and it looks much better.

I recommend the following formula for $\Delta\mathbf{radiosity}$ to plug into the algorithm on page 802[2]:

$$\Delta\mathbf{radiosity} = \rho_j \Delta\mathbf{b}_i \sum_{k=1}^{n_i} \frac{\cos\theta_i \cos\theta_j}{1 + \pi n_i r^2 / A_i} v$$

where

**bold** denotes an RGB color,
$k$ is the index of an element vertex on shooting polygon $i$,
$l$ is the index of an element vertex of receiving polygon $j$,
$\rho_j$ is the diffuse reflectance (an RGB color) of polygon $j$,
$n_i$ is the number of element vertices on polygon $i$,
and the following variables all relate to the vector between point $k$ and point $l$:
$v$ is the visibility of line segment $kl$ (0 if occluded, 1 if not),
it can be computed by calling `visible()` in `rayvis.c`,
$r$ is the length of segment $kl$,
$\theta_i$ is the angle between polygon $i$'s normal and line segment $kl$,
$\theta_j$ is the angle between polygon $j$'s normal and line segment $lk$.
You can compute the cosines using dot products, as described in lecture. If either cosine is negative, then one or both polygons are backfacing with respect to this ray, so treat this ray as if it were occluded.

Compared to Foley's pseudocode, the differences are that we're computing radiosities to element vertices $l$, not to elements $s$, and I'm being explicit about which form factor formula to use. To update the unshot radiosity of polygon $j$, we'll associate equal areas $A_l = A_j/n_j$ with each element vertex $l$ (this is one way to do it, but not the only way). That is, instead of Foley's formula: $\Delta\mathbf{b}_j \mathrel{+}= \Delta\mathbf{radiosity} A_s/A_j$, we'll use the formula: $\Delta\mathbf{b}_j \mathrel{+}= \Delta\mathbf{radiosity}/n_j$.

[2]This form factor formula comes from the paper "A Ray Tracing Algorithm for Progressive Radiosity", by John Wallace, Kells Elmquist, and Eric Haines, *SIGGRAPH '89 Proceedings*, pp. 315-324, and the book *Radiosity and Reealistic Image Synthesis* by Cohen and Wallace. Wallace's formula $F_{il} = A_l/n_i \sum_{k=1}^{n_i} \frac{\cos\theta_i \cos\theta_j}{A_i/n_i + \pi r^2} v$ implies that $F_{li} = (A_i/A_l)F_{il} = \sum_{k=1}^{n_i} \frac{\cos\theta_i \cos\theta_j}{1 + \pi n_i r^2/A_i} v$.

The pseudocode for each shooting step is then:

select polygon $i$ with greatest $\|\Delta\mathbf{b}_i\| A_i$
for each polygon $j$
for each element vertex $l$ of polygon $j$
$\Delta\mathbf{radiosity} = \rho_j \Delta\mathbf{b}_i \sum_{k=1}^{n_i} \frac{\cos\theta_i \cos\theta_j}{1+\pi n_i r^2/A_i} v$
$\mathbf{b}_l \mathrel{+}= \Delta\mathbf{radiosity}$
$\Delta\mathbf{b}_j \mathrel{+}= \Delta\mathbf{radiosity}/n_j$
$\Delta\mathbf{b}_i = 0$

Your program can stop shooting when the unshot radiosity goes below a small fraction of the initial unshot radiosity, where the fraction is .001, say. Implementing the ambient light approximation described in the book is optional. Adaptive subdivision is not required, but extra credit.

## Display

Call the routine `xwindow_setup()` before anything else, to create an empty X window. You can call the `zbuf` routines to initialize a zbuffer, render Gouraud-shaded polygons into it[3], and generate a `Pic` data structure containing an array of RGB pixels. This `Pic` can then be passed to `xpic_show` to display it in an X window.

See the source file `viewsc.c` for a template program that you can compile with `main.c` (run `make viewsc`) that shows how to call the `zbuf` routines. Hitting the '`q`' key on this program will quit out, and hitting the '`w`' key will prompt for a filename (in the window from which you ran it), and write a TIFF picture file with that name. If you keep the options in `main.c`, you'll be able to see smoother, less dithered pictures if you use the '`-gray`' or '`-gray -newcm`' options. This is useful for debugging on an 8 bit display.

Nice interaction and/or use of hardware-assisted z-buffer rendering for form factor calculation are also extra credit.

## What to Submit

In your `p4` directory (not a subdirectory):

Submit a nice looking picture of the scene file `pub/scene/rtable.sc`. Call it `rtable.tiff`. Also create and render a new scene containing multiple light sources and at least one table and chair. You can write these picture files using the '`w`' key, as mentioned above. For your final pictures, make the mesh fine enough (make the elements small enough) that the grid artifacts become unobjectionable, and let enough shooting steps run that your pictures show some color bleeding and other interreflection effects.[4]

Leave your source code and working executable (for either SGI, Sun, or LINUX) in your directory as well, but please delete or move away `.o` files and miscellaneous files.

Write a `README` file that explains what each file is. Since we're providing so many source files, tell us which files were written or nontrivially modified by you. For each picture file, tell us what command you ran, how many shooting steps took place, how many minutes it took, and what type of machine you ran on. Specifically, tell us how to run your program to recreate `rtable.tiff`. Also tell us what problems you encountered or special features you added.

In addition to turning in the above by midnight at the end of 25 April, we'll have you do demos in the Wean 5th floor cluster or Doherty 2300 on Tuesday, 30 April, during the class period.

[3] The polygon scan converter and clipper used here came from "Generic Convex Polygon Scan Conversion and Clipping", Paul S. Heckbert, *Graphics Gems*, Andrew Glassner, ed., Academic Press, 1990, pp. 84-86, 667-680. Also available by anonymous FTP from `princeton.edu:/pub/Graphics/GraphicsGems`.

[4] If you get an artifact where the corners of the room look dark, try moving in the element vertices that are along the edges of the polygon by a small amount. That is, shrink the polygon a bit.