Parallel Radiosity

A.N. Yzelman

February 23, 2007

Contents

Preface

Radiosity has been around since 1984, and is one of the most used algorithms for global illumination regarding perfectly diffuse surfaces. However, it is also quite a heavy algorithm for any computer; on sufficiently detailed surfaces, rendering may take days or longer.

To reduce running times, parallelisation seems a natural choice. Many problems need to be addressed before any such algorithm can be properly designed. Also, to achieve good performance, an in-depth discussion of parallel distributions is required.

Experiments have been performed as well, to show that the final algorithm presented here indeed works properly. Altogether, thinking of and programming a radiosity renderer had its difficulties, but in the end a working and nicely performing algorithm was successfully created.

This paper will first take one through the basic theory of radiosity, and of progressive radiosity. Then it will first present a sequential algorithm. Furthermore, preparations are made for a parallel algorithm; problems such as parallel occlusion detection, patch distribution, load balancing, transition to matrix-vector operations, and parallel matrix element distributions are addressed before moving on to the final algorithm.

Finally, experiment results will be given together with following conclusions. Also, suggestions for future work will be given.

Hopefully this document will prove useful to others as well as myself. Working on this subject certainly has increased my understanding of radiosity itself as well as parallel distributions. As always, feedback is welcome and may be preferably given by email.

Albert-Jan Yzelman
(ajy777 ”at” gmail.com)

Chapter 1

Sequential Algorithm

In this first part we will present the necessary theory for building a radiosity algorithm. This includes a small introduction to the physics of radiosity, deriving a linear system, effectively solving this system, as well as the computational representation of a scene. From these basics, a sequential radiosity algorithm will be built. It is this algorithm that we will try to efficiently transform into a parallel version.

1.1 The Basic Radiosity method

Radiosity is a way to model light interaction between surfaces and light sources. It is based solely on the idea that light in fact is energy, and this energy is transmitted from light sources to all visable surfaces. Then, in turn, energy is transmitted by those surfaces to other surfaces via reflection.

In this sense, a light source is in principle nothing more than a surface initially radiating energy, whereas the other surfaces only reflect incoming energy. This emission of energy is presumed to be equally distributed in all directions. This is also called diffuse lighting.

1.1.1 The Form Factor

If we consider a scene in which light sources and surfaces are placed, our model defines a relation between all those surfaces, since energy is transmitted between each such pair of surfaces.

The amount of energy transmitted is dependent on the relative positioning of the two points. The greater the angle between them, the less energy is transmitted. The factor which defines this relation given the positions of the two points, is called the *Form Factor*.

Definition Point-to-Point Form Factor

The factor with which energy from some point x is transmitted to an other point y is denoted by the Form Factor:

$$f(x,y) = \frac{\cos\theta_x \cos\theta_y}{\pi r^2} V(x,y) \tag{1.1}$$

where $V(x,y)$ denotes the *visability* function:

$$V(i,j) = \begin{cases} 1, & \text{if the surfaces i and j are in a direct line of sight of each other} \\ 0, & \text{otherwise} \end{cases} \tag{1.2}$$

The points regarded here are of course of infinitesimal size; and a surface consists of infinitely many points. Hence if we want to calculate the form factor between two surfaces, we have to integrate the point-to-point form factors over both surfaces.

Definition Exact Surface-to-Surface Form Factor
The exact Form Factor from *surface i* to surface j is given by integrating over the area A_i of i:

$$F_{ij} = \frac{1}{A_j} \int_{A_i} f(x,y) dx dy \tag{1.3}$$

where A_j is the area of surface j.

With regards to developing an algorithm for radiosity, such a multi-dimensional integral is unworkable since an integral is nothing short of an infinite summation. Of course an infinite summation could be avoided if an exact solution of such integrals existed; but this is not always the case, and even if there exists such a solution, computationally it is likely that it would be more efficient to approximate the integral instead of evaluating the exact integral. Therefore we will work with the following approximation instead:

Definition Instead of working with the exact integral (1.3) we will instead use the approximating *Oriented-Disc Form Factor* [4]:

$$F_{ij} = A_j \frac{\cos\theta_i \cos\theta_j}{\pi r^2 + A_j} V(i,j) \tag{1.4}$$

The form factor between surfaces i and j is usually denoted by F_{ij}, meaning the factor with which i transmits its energy to j. Note that F_{ij} does not have to be equal to F_{ji}, for example when i and j are of different size.

1.1.2 General Equation

Apart from the form factor, we need other variables to describe the radiosity model more precisely. As said before, radiosity is based on the idea that surfaces can reflect energy. Therefore, every surface has a reflectance property ρ which we assume is constant over the surface. This property is best described as a vector in $\mathbb{R}^3$; every dimension represents a different base colour (red, green, blue). This property defines the colour of surfaces; some colours are reflected better than others. In physics, this is explained by the fact that every colour has a different energy frequency, and some frequencies are better absorbed by a given surface than others.

What we also need to realise is that energy from one surface cannot reach a second surface if the line-of-sight between these two surfaces is occluded by other surfaces. Therefore, we define the function V for visibility, which takes value either one or zero. For simplicity, we assumed that the factor V is included in the form factor.

Let us furthermore define a variable $B = (B_R, B_G, B_B)^T$ for each surface, defining the brightness (or radiosity) of that surface, per distinct RGB-colour. This is the value we are interested in and need to calculate for all surfaces. We also have a variable $E = (E_R, E_G, E_B)^T$, defining the initial energy output. This will be nonzero if the surface in question is a light source. Let S denote the set of all present surfaces and let S have n surfaces. We can then formulate the radiosity relation in a general equation:

Definition The radiosity equation for all surfaces $i \in S$ is given by:

$$B_i = E_i + \rho_i \sum_{j \in S} B_j F_{ji} \tag{1.5}$$

or, in words: the brightness of a surface is equal to the emissivity of that surface, plus the amount of energy which is received from all other surfaces, that this surface will reflect.

1.1.3 Matrix Radiosity

All variables in the equation (1.5) are known, except for the ones we are interested in, namely the B_i. The problem of calculating the B_i-vectors is in essence the same as solving three independent linear systems. We first write the radiosity equation slightly differently:

$$E_i = B_i - \rho_i \sum_{j \in S} F_{ji} B_j$$

Then:

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{21} & \cdots & -\rho_1 F_{n1} \\ -\rho_2 F_{12} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{1n} & -\rho_n F_{2n} & \cdots & 1-\rho_n F_{nn} \end{pmatrix} \begin{pmatrix} B_{1R} & B_{1G} & B_{1B} \\ B_{2R} & B_{2G} & B_{2B} \\ \vdots & \vdots & \vdots \\ B_{nR} & B_{nG} & B_{nB} \end{pmatrix} = \begin{pmatrix} E_{1R} & E_{1G} & E_{1B} \\ E_{2R} & E_{2G} & E_{2B} \\ \vdots & \vdots & \vdots \\ E_{nR} & E_{nG} & E_{nB} \end{pmatrix}$$

We can rewrite this more compactly to:

$$M \begin{pmatrix} \vdots & \vdots & \vdots \\ \mathbf{B_R} & \mathbf{B_G} & \mathbf{B_B} \\ \vdots & \vdots & \vdots \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \mathbf{E_R} & \mathbf{E_G} & \mathbf{E_B} \\ \vdots & \vdots & \vdots \end{pmatrix}$$

and hence we may derive the three linear systems $M\mathbf{B_R} = \mathbf{E_R}$, $M\mathbf{B_G} = \mathbf{E_G}$ and $M\mathbf{B_B} = \mathbf{E_B}$. M is an n by n matrix, and the bold faced vectors are n by 1. Note that in an implementation, there is no need to explicitly solve three different linear systems; with the use of vector-classes one can implement an elementwise multiplication and subdivision between vectors, so that the program will solve three systems in one pass. This may be a little tricky though (it requires the use of distance norms on vectors, which in effect can cause a few more iterations than strictly necessary), and will be subject to analysis later on.
Also note that a trivial opportunity for parallelism presents itself here; by using one processor for each colour, there will never be too many iterations done by any processor. This parallelism would be expected to cut the normal running time in somewhat less than three. Note that the three different linear systems to solve are similar in the sense that they all use the same system matrix M. Therefore, from now on we will consider only one such linear system:

$$MB = E \tag{1.6}$$

where B and E are $n \times 1$ vectors representing brightness and initial energy emission.

1.2 Progressive radiosity

Progressive refinement is an iterative strategy for solving the basic radiosity system without having to store all matrix entries. The basic matrix radiosity method requires $O(N^2)$ storage. For complex scenes, this storage requirement is excessive. Progressive radiosity effectively reduces storage requirements from $O(N^2)$ to $O(N)$. It has been proven that progressive refinement is equivalent to Southwell relaxation (an iterative method for solving linear systems) applied to the matrix radiosity method [3].

It works by selecting as a first solution for B the vector E. Recall that B is the brightness vector; the solution of the radiosity system. E was the initial emission vector.

As the name implies, this iterative method is used for refining a current solution B^i of the linear system. Refinement is done by first choosing a surface j with the largest amount of residual energy, i.e. energy that hasn't been used for refinement yet. Then, for each other surface with index $i \in [1, n]$ $(i \neq j)$, the factor $B_j \rho_i F_{ji}$ is added to the current value of B_i. This process is repeated until the result is satisfactorily close to the exact solution.

So next to a vector storing the brightness of all surfaces, we need a way of knowing how much energy is already used for refinement, or rather, how much energy has not been. Therefore, next to a vector B, we define a vector U. This vector stores for each surface the amount of not yet transmitted energy (the total energy received and not transmitted, refinement after refinement; also called *residual* energy). Before the first refinement step, no energy has been transmitted at all, and as such, initially:

$$B^0 = U^0 = E \tag{1.7}$$

Note that the surface with the highest amount of residual energy at refinement step k is now easily given by $\arg\max\{U_i^k\}$, where $i \in \{1, ..., n\}$. The superscript numbers stand for the number of refinements done.

Concluding, after setting the initial B and U vectors, the progressive refinement algorithm can be expressed in the following steps, which are to be repeated until the solution has converged satisfactorily:

Algorithm 1 The progressive radiosity algorithm

```
while ||U|| ≥ some threshold value do
  Choose a shooter surface j which has the highest residual energy
  for all i ∈ S, i ≠ j do
    Set B_i to B_i + ρ_i F_ji U_j
    Set U_i to U_i + ρ_i F_ji U_j
  end for
  Set U_j = 0
end while
```

1.2.1 Matrix-vector formulation

A progressive refinement of B^l, for all $l \geq 0$, with shooting surface $j = j_l$, may be written as:

$$B^{l+1} = B^l + U_j^l \begin{pmatrix} \rho_1 F_{j1} \\ \vdots \\ \rho_{j-1} F_{j(j-1)} \\ 0 \\ \rho_{j+1} F_{j(j+1)} \\ \vdots \\ \rho_n F_{jn} \end{pmatrix} \tag{1.8}$$

A formula for U^{l+1} may be written as:

$$U^{l+1} = \mathbf{A}(j) U^l \tag{1.9}$$

With the matrix $\mathbf{A}$ defined on a shooter j as follows:

$$\mathbf{A}(j) = \begin{pmatrix} 1 & 0 & \cdots & 0 & \rho_1 F_{j1} & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & \rho_2 F_{j2} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \rho_{j-1} F_{j(j-1)} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \rho_{j+1} F_{j(j+1)} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \rho_n F_{jn} & 0 & \cdots & 1 \end{pmatrix} \tag{1.10}$$

Note that equation (1.8) can be rewritten using (1.10) to:

$$B^{l+1} = B^l + U_j^l A(j) e_j \tag{1.11}$$

where e_j is the jth unit vector. Combining (1.11) and (1.9) gives us:

$$B^{l+1} = B^l + (A(j_{l-1}) U^{l-1})_j A(j) e_j$$

which can easily be simplified to

$$B^{l+1} = B^l + (e_j^T A(j_{l-1})) U^{l-1}) A(j) e_j \tag{1.12}$$

Taking into account the above, we can rewrite one progressive refinement iteration itself as a matrix-vector formulation, as in algorithm 2

Algorithm 2 Algorithm to calculate the new U^{l+1} and B^{l+1} by overwriting the old U^l and B^l.

Set $j =$arg max U
for all $i \in S, i \neq j$ **do**
  $\Delta = U_j A(j) e_j$
  $U = U + \Delta$
  $B = B + \Delta$
end for
$U_j = 0$

1.3 Hierarchical scene description

We will now introduce a method of scene storage, which goes beyond simply storing individual surfaces. Note that for simplicity we allow only square surfaces in a scene. Of course, other types of surfaces can also made to work with a radiosity renderer.

1.3.1 Description

Defining each square separately becomes very ineffective when we want detailed images; many squares must be defined for each surface to obtain a fine grid upon which to apply the radiosity method.

Therefore, we propose to use a type of hierarchical scene description. This means we will only define surfaces and the number of patches in which it is divided.

In the experimental renderer written for this paper, we only support square surfaces. Each such surface is then divided into n predefined patches. These patches are also of a square shape, and each patch has the same area as any other patch on the same surface.

As a consequence, each square surface k will be divided into a number of $n_k = 4^{d_k}$ equally-sized square patches; each square can only be subdivided into 4 other squares, if we demand the sub-squares to be equally sized. The radiosity algorithm will now work on these patches instead of individual surfaces. The total number of patches equals $n = \sum_{k=1}^{\tilde{n}} n_k$, where $\tilde{n}$ is the number of surfaces. The depth d_k may differ for different surfaces, although in the later experiments we have chosen to let $d_k = d$, for all surfaces k.

1.3.2 Relation to quad-trees

In effect, this scheme is very similar to representing a surface with a full quad-tree of depth d. If we do not require the quad-tree to be full, certain areas of the square can be subdivided into more elements than other areas. This is useful if only parts of a surface need high resolution rendering. There exist methods for determining while rendering if a certain area of a surface should be further refined for good results. This method is called adaptive subdivision, see for example [4].

1.3.3 Storage

For each square surface, we store the total number of patches, the location of the mid-point of the square in the scene space, the size of one side of the square, the normal vector of the square and finally, the direction where the square is pointed to. See figure 1.1.

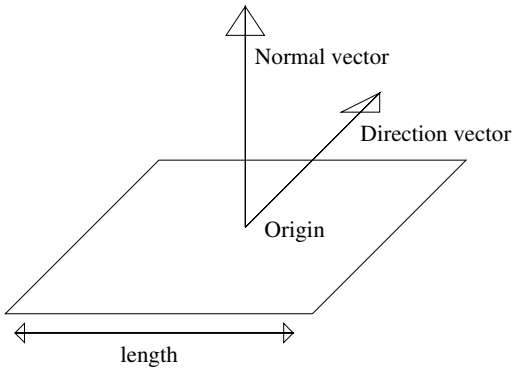


Figure 1.1: Geometric properties of a square surface

The patch numbering is as in figures 1.2 and 1.3. There the numbering is given for the case of $d = 1$ and $d = 2$, respectively. Numbering proceeds in a similar way for higher patch resolutions. For each patch we store the radiosity and residual vector values. Other values do not have to be explicitly stored; individual patch locations can be derived from the patch number and data on the surface origin and orientation.

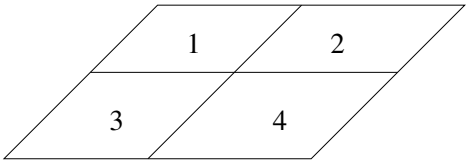


Figure 1.2: Patch numbering on a square divided into 4 patches

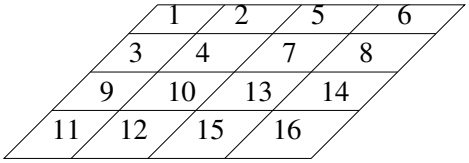


Figure 1.3: Patch numbering on a square divided into 16 patches

1.3.4 Test scene

For this paper, a parallel radiosity renderer was programmed. The test scene it rendered consists of a cube (consisting of six squares) of a 10 x 10 x 10 size. The sides of the cube have different colours. The standard position for the camera in this scene will be the centre of the cube. The camera will point directly to the top square. As a light source we added a small square surface, hovering a little below the top square.

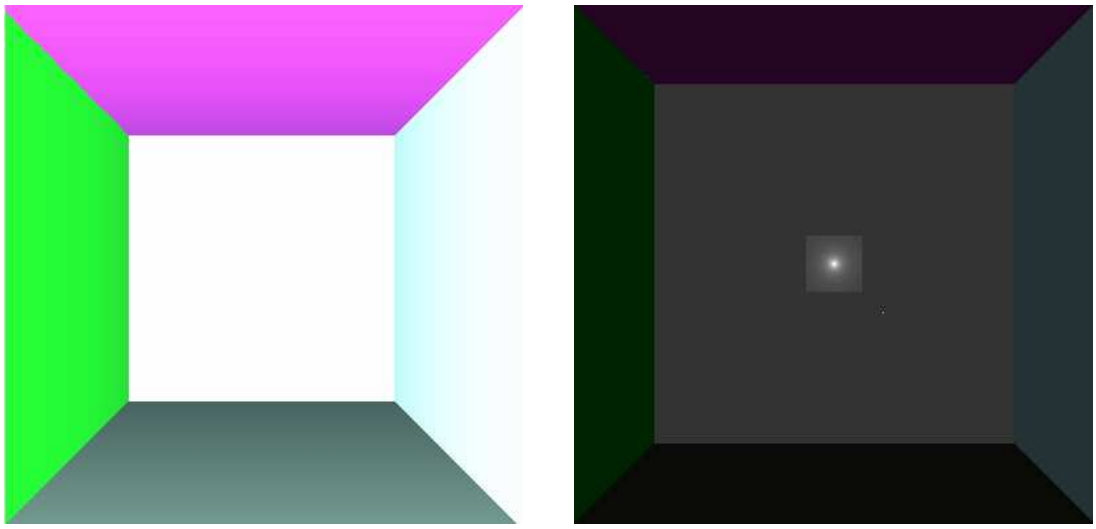
A picture of this scene is given at figure 1.4(a). Note that this picture simply displays the geometry and the colours of the surfaces; no amount of (global) illumination was attempted. A ray-traced rendering of this scene, which does do illumination, results in the picture seen at figure 1.4(b). Since ray-tracing does not support surface light sources, a point light source was added at the middle of the supposedly light-emitting square. Ambient lighting is also present.

Figure 1.4(b) does not display a very realistic image; colours of the sides are not mingled with each other, and the square behind the light source remains unrealistically dark. A radiosity renderer will give a much more realistic image, as will be shown later on. The camera position in these images will be maintained in the entire paper so that any two images may be compared correctly.

1.3.5 Differences between ray-tracing and radiosity

We will now briefly explain why radiosity would render more realistic images. This is because of several limitations in ray-tracing (RT). RT works with point-light sources; a light source can thus be only defined as a infinitesimal point which somehow emits light. This alone will irrevocably cause very sharp, and thus unrealistic shadows.

Ray tracing is also incapable of reflection in the diffuse 'radiosity' way. Reflections in RT will result in mirror-like reflections, instead of a diffuse spreading of light energy. There are methods for making individual surfaces look diffuse, but there still is no inter-surface energy transmission.



(a) Scene rendered by returning the square colour.

(b) Ray-traced image of the base scene.

Figure 1.4: Base scene, viewing from inside the cube. This is the standard viewpoint.

Another difference is the use of ambient lighting in RT; every surface is lighted by a preset amount of ambient light. This is actually RTs approximation of indirect (or global) lighting. This makes it clear that ray-tracing and radiosity are two very different ways of rendering; it is also common to first use radiosity rendering to calculate global light distribution, and then afterwards use other types of renderers to render objects in the scene.

Another difference between ray-tracing and radiosity is camera-dependency. With ray-tracing, a camera position and view-point must be defined; from that viewpoint, the scene is projected and lighting is calculated. With radiosity however, energy is calculated for every patch. Those results are of course viewpoint-independent.

However, we still like to generate pictures of the output. Therefore, a stripped-down ray tracer is used. This renderer uses a camera location and a direction to project the scene on the camera viewport; like a painter draws the scene he is looking at on his board (therefore, this rendering algorithm is also called the painter's algorithm). The ray-tracer draws patches with the colour defined by the radiosity renderer, taking only into account the viewing angle; the sharper the angle, the less energy is percepted.

1.4 Hierarchical occlusion detection

One of the most costly operations done in progressive radiosity, is to determine the visability constant of a form factor between two patches. Since we do not want to explicitly store anything of the form factors, the visability constants must be recalculated any time we need a form factor.

To speed up occlusion detection we could use BSP trees[1] (or perhaps Z-buffering) as is standard in many graphics applications. However, to apply these methods on the level of patches is wasteful since we are using a hierarchical representation of our scene. It is more efficient to note that any patch has a direct line of sight to any other patch, if and only if no other surface occludes this line sight. So, instead of searching for occluding *patches*, we can search for occluding *surfaces*.

This will simplify occlusion detection a lot. For this paper, no further optimisations were done with respect to occlusion detection; for small scenes, detection already went sufficiently fast. Also, extending BSP-trees or Z-buffers to a parallel setting might not be straightforward, and thus beyond the scope of this paper. See [6] for a parallel algorithm which does use a Z-buffer.

1.4.1 Occlusion detection algorithm

To do the actual occlusion detection between two patches, say T_1 and T_2, we need their origins in the scene space (o_1 and o_2), and their normal vectors (n_1 and n_2). Also, we need the origins of all *surfaces* in the scene ($so_1, ..., so_n$) and their normal vectors ($sn_1, ..., sn_n$).

Before we derive anything, let us first shift the origin of the scene to the origin of T_1; then the location of the origin of T_2 is at $v = o_2 - o_1$. The normal vectors of course stay unchanged. Let us now consider a surface i with origin so_i and normal vector sn_i. Since we shifted our coordinate system, note that in further calculations we have to consider $so_i - o_1$. If the surface i intersects the vector v, then the line of sight from T_1 to T_2 is occluded.

If this is the case, then certainly there exists a point p on v which lies on the plane defined by sn_i and so_i while p indeed is contained in the surface i. The plane in question here consist of all points $x \in \mathbb{R}^3$ for which $sn_i \cdot x = sn_i \cdot so_i$ holds. A point on the line v is any point $x \in \mathbb{R}^3$ for which $x = dv, d \in \mathbb{R}^3$. Occlusion occurs if for the point $p = dv$ we have that $0 \leq d \leq 1$.

Concluding, we have the following two equations:

$$sn_i \cdot p = sn_i \cdot (so_i - o_1) \tag{1.13}$$

$$p = dv \tag{1.14}$$

with

$$v = o_2 - o_1 \tag{1.15}$$

Combining equation (1.13) and (1.14) we obtain:

$$sn_i \cdot dv = d(sn_i \cdot v) = sn_i \cdot (so_i - o_1) \tag{1.16}$$

and so,

[1] BSP tree in this context refers to a datastructure derived from *binary space partitioning*; hence not to be confused with *Bulk Synchronous Parallel*.

$$d = \frac{sn_i \cdot (so_i - o_1)}{sn_i \cdot v} \tag{1.17}$$

This procedure is independent of our surfaces being of square shape; every two-dimensional surface lies on a plane in a three-dimensional space. The only shape-dependent procedure here is checking if $p = dv$ lies within the surface i. For a square shape, we can either transform p to a two-dimensional coordinate $\tilde{p}$ on the plane spanned by i, and check if $|\tilde{p}_x| \leq \frac{l}{2}$ and $|\tilde{p}_y| \leq \frac{l}{2}$ where l is the square length.

A different method would be to check if the point p is within a cube C, where C is the smallest box in $\mathbb{R}^3$ which contains the patch i. This works correctly, because if a point in C does not lie on the surface i, this point also does not lie on the plane spanned by i; while p definitely lies on that plane. If the maximum coordinates of C are readily available, this method may be preferable to the previously described procedure. Note that this method works for square surfaces, but may not work for other shapes[2].

Algorithm 3 presents the occlusion detection algorithm for surfaces with an arbitrary shape. Note that if we normalise the vector v before entering the loop, d becomes the distance between the surface i and T_1. This may be useful in some applications. Note however that when v is normalised, we have to check if $d \in (0, ||o_2 - o_1||)$ instead of $(0, 1)$. Finally, note that the only patches we need the exact data of are T_1 and T_2; we do not need data on any other patch. This will be exploited further when we design the parallel algorithm.

Algorithm 3 Generic occlusion detection algorithm.

```
v = o_2 − o_1
for all i in the set of all available surfaces in the current scene do
  d = ((so_i − o_1) · (sn_i)) / (sn_i · v)
  if d ∈ (0, 1) then
    p = dv
    if p is contained in the surface of patch i then
      RETURN 1 (The line of sight is occluded; exit algorithm)
    end if
  end if
end for
RETURN 0 (The line of sight is not occluded)
```

[2]For example, it would work for rectangles, but not for triangles.

1.5 The sequential progressive radiosity algorithm

We now have all the ingredients to put together a progressive radiosity algorithm. Suppose we have a scene description consisting of various surfaces and their properties, including the number of patches each surface consists of. Patches which do not emit any light have their radiosity and residual vectors initialised to 0. Patches which do emit light have both their radiosity and residual values set to a predefined level. The final sequential algorithm is presented in algorithm 5. This algorithm of course requires to calculate the form factors; an algorithm doing this is outlined in algorithm 4. In turn, this algorithm refers back to the occlusion detection Algorithm 3.

Note that to determine which patch has the highest residual vector, we must decide which norm we use. We could settle for the standard euclidean norm, but taking a square root is a costly operation, and is best avoided. Also, taking the square is not necessary for determining the vector with maximum norm; therefore, in the implementation, we will determine the maximum of an array of vectors v_i by calculating the maximum of $||v_i||^2$.

Algorithm 4 Pseudocode for calculating F_{ji}

if Patch i is visible from patch j (not occluded) **then**
  Obtain A_j, the area of patch j
  Obtain o_i and n_i, the origin and normal vector of patch i in $\mathbb{R}^3$
  Obtain o_j and n_j, the origin and normal vector of patch j in $\mathbb{R}^3$
  $v = o_j - o_i$
  $d = v \cdot v \quad (= ||v||^2)$
  $v = v/d$
  $c_1 = n_i \cdot v$, the first cosine
  $c_2 = n_j \cdot v$, the second cosine
  $c = c_1 c_2$
  RETURN $\frac{cA_j}{d\pi + A_j}$
else
  RETURN 0
end if

Algorithm 5 The final sequential progressive refinement algorithm

while Stop conditions have not been met **do**
  $M = 0$
  for all patches i in the entire scene **do**
    $m = ||U_i||^2$
    if $m > M$ **then**
      $m = M$
      $s = i$
    end if
  end for
  for all patches $p \neq s$ **do**
    Calculate F_{sp}
    $\Delta = \rho_p F_{sp} U_s$
    $U_p = U_p + \Delta$
    $B_p = B_p + \Delta$
  end for
  $U_s = 0$
end while

1.5.1 Stop conditions

The experimental program as it is implemented now checks for three different stop conditions. The first is a total residual threshold; if the sum of total residuals is lower than this threshold, the program will exit. The second is a maximum number of iterations; this comes in handy when one does not want to wait a long time for the algorithm to finish.

The third condition is a maximum residual threshold. If the maximum of all residual vectors is less than this threshold, the algorithm will exit. Note that the patch with the maximum residual would have become the shooter of the next iteration. Thus this third condition actually enforces that if the next shooter would be a very weak shooter, the algorithm can exit.

Chapter 2

Parallelisation

2.1 Parallel radiosity methods

Now, we shall explore options for parallelisation of the progressive radiosity algorithm. Then we shall explore two different ways of parallelisation.

2.1.1 Parallelisation

Globally, there are two ways of parallelisation. We will investigate a method called P-shooting, keeping communication optimisation as well as scalability in mind. Scalability in this case means that the memory usage ($O(n)$ in the original sequential algorithm) will drop as more processors are used for solving. In effect, when we have P processors, we would like a memory usage of $O(n/P)$ at each processor.

P-Shooting

This first method does not use parallel iterations. Instead, if we have P processors, this method will do P iterations concurrently. This is also investigated in [2]. Needless to say, this method modifies the original progressive algorithm; the order of the patches with the greatest amount of energy may be disturbed. However, if we take care to let every processor work with one of the P patches with the greatest amount of unprocessed energy, convergence speed will not suffer all that much.

In this setting, each processor needs to calculate the form factor between the patch it is processing, and all other patches in the scene. If we define the form factor matrix F to be:

$$F = \begin{pmatrix} F_{11} & F_{12} & \dots & F_{1n} \\ F_{21} & F_{22} & \dots & F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ F_{n1} & F_{n2} & \dots & F_{nn} \end{pmatrix}$$

and the diagonal reflection matrix R to be:

$$R = \begin{pmatrix} \rho_1 & 0 & \cdots & 0 \\ 0 & \rho_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \rho_n \end{pmatrix}$$

We assume the shooting patches are the ones with index numbers $i_0 < i_1 < \ldots < i_{P-1}$. Then, one parallel iteration is mathematically equivalent to the sequential algorithm 6.

Algorithm 6 Pseudocode for one parallel iteration

Phase 0
Find the patches with the globally largest unprocessed radiosity, $A_{i_0}, \ldots, A_{i_{P-1}}$.

Phase 1 Calculate

$$\Delta = (F * R)^T \left[\begin{pmatrix} 0 \\ \vdots \\ 0 \\ U_{i_0} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ U_{i_1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \cdots + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ U_{i_{P-1}} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \right]$$

$$(\Delta = (F * R)^T (U_{i_0} e_{i_0} + U_{i_1} e_{i_1} + \cdots + U_{i_{P-1}} e_{i_{P-1}}))$$

Phase 2

$$U = U + \Delta$$

Phase 3

$$B = B + \Delta$$

Phase 4
For all i_j we set $U_{i_j} = 0$.

Note that in algorithm 6 Δ is obtained as a series of matrix-vector multiplications; P-shooting may thus be considered as an extension of a parallel matrix-vector multiplication algorithm. This becomes more clear if we write $A = (F * R)^T$ and $x = (U_{i_0} e_{i_0} + U_{i_1} e_{i_1} + \cdots + U_{i_{P-1}} e_{i_{P-1}}))$; then $\Delta = Ax$.

For solving the axpys and matrix-vector calculations from algorithm 6 in parallel, we propose to use already established algorithms, such as those found in [1]. In particular, the Cartesian distribution at p.179-p.186 looks very usable.

The only thing we need to determine is whether or not we will explicitly store the F matrix. The whole matrix is of size N^2. However, if we distribute it over P processors in a Cartesian way, the memory requirement of each processor will become N^2/P. This would have been interesting if P was of the same order as N, but alas, this is definitely not the case for most scenes.

So, direct storage of the whole matrix is still out of the question. Instead, we will let every processor requesting access to F calculate the form factors it wants itself. To do this with the least amount of communication possible, we want one of the processors storing entries at the ith column of F to hold the ith patch as well. For the same reasons, we want one of the processors holding entries at the ith row to hold the ith patch as well. These demands are conflicting if each patch should be stored only once. For now, we will opt to distribute the ith patch simply on the processor which holds the ith diagonal in the F matrix. This is called a cyclic or diagonal distribution.

The standard parallel matrix-vector algorithm must of-course be modified for on-the-fly calculation of the form factors. Regardless, we now have ensured ourselves of a scalable solution

($O(N/P)$) storage at each processor. Also, by using a Cartesian distribution, communication is also optimised. However, because of the on-the-fly calculation of form factors, communication will increase. By what amount this is, will be investigated later on. There is also the matter of occlusion detection; this will be addressed in the following section.

Parallel iterations

Another method for parallel computation of progressive radiosity is to actively make each iteration parallel, without modifying the original algorithm as we did in the previous method. We can do this by simple distributing the patches over each processor. Then, at each iteration, when calculating the Δ-vector, we do this locally. The reason why this is harder to implement than it now sounds, is because of occlusion detection. For that to succeed, the entire scene must be tested. However, there is a way to prevent that. In the current implementation of the radiosity renderer, we used a kind of hierarchal data distribution. All patches must come from surfaces. Thus, if we make known to each processor all surfaces, but not all patches, we still can do occlusion detection perfectly. This approach is also used in P-shooting.

The storage at each processor will then become of order $O(n + N/P)$, with n the number of surfaces in the scene. This is not fully scalable, but may still be acceptable when n is much smaller than N/P. This may for example be the case when each surface is thoroughly refined. If we compare this method with P-shooting, we actually see that the only difference is that we either have P simultaneously shooting patches, or just one. Shooting more than once at the same time will however almost surely be faster, so from now on we will investigate the P-shooter algorithm.

2.1.2 Parallel model

For the experimental implementation of the final P-shooter algorithm, we need a programming model to ease programming and make the algorithm portable over many architectures. The current program makes use of the *Bulk Synchronous Parallel (BSP)* model. This model makes it possible to divide algorithms into supersteps. Between supersteps, communication is possible. Different supersteps are separated by a *synchronisation* (or *sync*, for short) of processors.

For a more in-depth introduction to BSP, see [1] or [7].

2.2 Parallel scene distribution

We have already determined that because of occlusion detection, it is best to store all information about surfaces at each processor. In contrast, we distribute the patches over all processors. Generally, we can choose from either a tiled or scattered distribution ([2], fig. 3).

With a tiled distribution, patches close to each other are assigned to the same processor. A scattered method distributes in the opposite way; patches close to each other are assigned to as many different processors as possible.

For our applications, the scattered distribution seems to be the best choice. The tiled distribution would result in distributions in which many patches stored at a given processor are on the same surface. However, there is little to no interaction between these patches which are so close to each other.

As a consequence, any elected shooter from any processor has no interaction with a lot of its other locally stored patches. Also, if a certain surface is occluded from any elected shooter (from any processor), the processor holding the patches in that occluded surface has a much lighter computation load than the other processors. Therefore, with respect to load balance, the scattered distribution is preferred.

Figures 2.2(a), 2.2(b), 2.2(c) and 2.2(d) detail the scene distribution amongst processors in the current parallel radiosity renderer. We used here 4 processors and refined each square surface to have 4 patches.

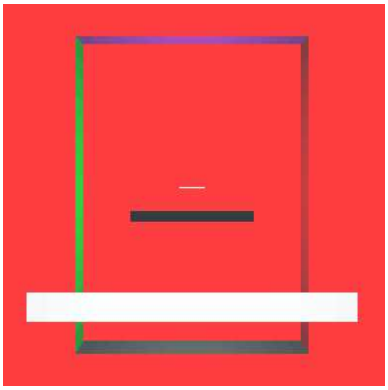


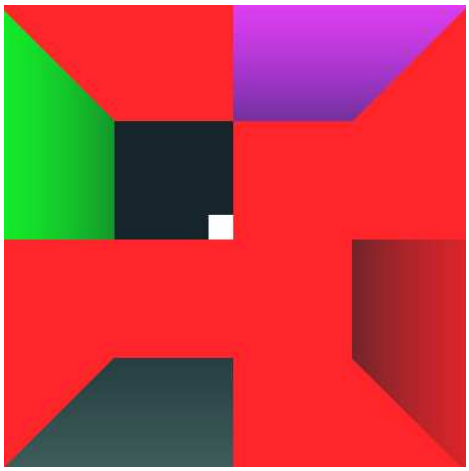
Figure 2.1: Patches local to processor 14; non-perfect scattering. See also figure 2.2.

2.2.1 Non-perfect scattering

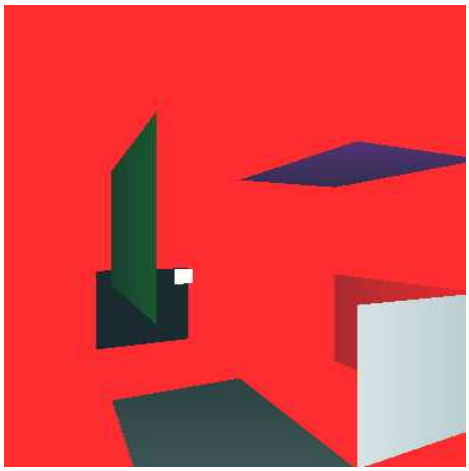
While investigating the patch distribution in the experimental renderer, an unlucky bug was discovered. The patch numbering we use together with a diagonally cycled patch distribution results in a kind of row-scattering. See figure 2.1, obtained by a 16 processor run on a scene where each square was refined to $4^4 = 256$ patches.

As can be seen, while the patches are indeed scattered over all surfaces, they are not *completely* scattered over a single surface. However, this should not prove to cause a large load imbalance over most surfaces; it's rarely seen that any row of patches on a surface is occluded while other rows are not.

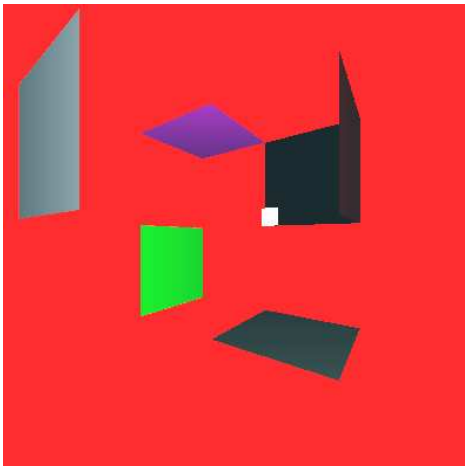
Because this bug proved to be quite tricky to fix, and its negative influence on load-balancing is expected to be insignificant, it was still present in the experimental software due to time considerations.



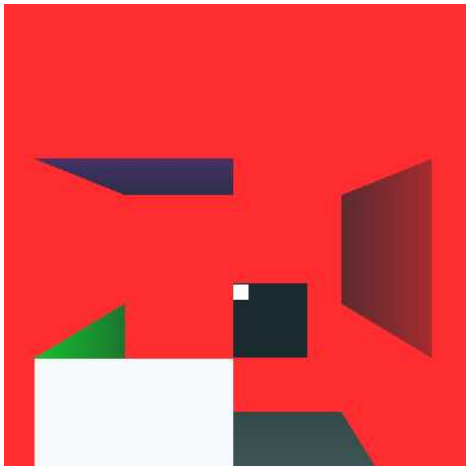
(a) Patches drawn by processor 1. Viewed from the standard viewpoint.



(b) Patches drawn by processor 2. Viewed from a top-left viewpoint.



(c) Patches drawn by processor 3. Viewed from a bottom-right viewpoint.



(d) Patches drawn by processor 4. Viewed from a top-mid viewpoint.

Figure 2.2: The standard scene distribution among four processors with 4 patches per square surface.

2.3 Parallelisation using cyclic distribution

Instead of using a Cartesian distribution, let us first look at the P-shooter algorithm using a cyclic distribution. By using this distribution, we let the ith patch of each surface to be stored by the processor with the number i mod P (with P the total number of available processors).

Also, in this distribution, the matrix entry (i, j) of the parallel radiosity matrix F will be calculated by the processor which also stores i. This irrevocably results in the situation in which each processor will need the data stored at all other processors. To meet this need, we will need $P - 1$ communication steps, in which non-local patch information is rotated through all processors. Algorithm 7 details a parallel algorithm using a cyclic distribution. Note that this algorithm also works if we do not have a hierarchical scene structure; all patches visit all processors, and as such it is possible to do occlusion detection perfectly.

2.3.1 Memory usage

The memory usage for this distribution is of the order $O(\tilde{n} + 2 * n/P)$, where $\tilde{n}$ is the number of surfaces, n the total number of patches and P the number of processors. n/P is multiplied by a factor 2 here because we need to work with a temporary buffer.

2.3.2 Communication

The communication steps counted in number of transmitted patches for a progressive refinement procedure using this distribution, is of order $g \cdot O(P + P \cdot n/P)$ per iteration. g represents a time constant a given supercomputer requires for communication of a single patch. We first have to transmit information on the P different shooters to all processors. After that we have to cycle through all local data of other processors.

Algorithm 7 Pseudocode for a parallel algorithm through a cyclic distribution

1: Determine the patch s with the highest residual available locally
2: Send all other processors data on the local shooter, patch s
3: *Synchronise*
4: **while** Stop conditions have not been met **do**
5: **for all** locally stored patches p **do**
6: Shoot energy from the local shooter s to p *(See algorithm 3, 4 and 5 for details.)*
7: **end for**
8: Copy the data on local patches to an outgoing buffer
9: **for** $i = 1$ to $P - 1$ **do**
10: Send the data in the outgoing buffer to the incoming buffer of the next processor
11: *Synchronise*
12: **for all** patches p in the incoming buffer **do**
13: Shoot energy from the local shooter s to p
14: **end for**
15: Move the patches from the incoming buffer to the outgoing buffer
16: **end for**
17: Send the data in the outgoing buffer to the incoming buffer of the next processor
18: *Synchronise*
19: Replace the local patch data with the data in the incoming buffer
20: Set the residual energy vector of the local shooter U_s to zero
21: Determine the patch s with the highest residual available locally
22: Send all other processors data on the local shooter, patch s
23: **end while**

Short notes:

Lines 1 *to* 4, 21 *and* 22; as we will see, it is useful to have at hand information about the elected shooters at each processor for determining the stop condition. Therefore, every time the stop condition is checked at line 4, every processor elects its shooter and communicates its info to all other processors.

Line 10; in a cyclic distribution, processors have a predefined order $p_0 < p_1 < \cdots < p_{P-1}$. The next processor with respect to any processor p_i is the processor with number $p_{(i+1)\%P}$.

Lines 17 *to* 19; note that at this point, the patches which reside in the incoming buffer at any processor, are the copies of the locally stored patches created at line 8. These copies have received energy from all elected shooters at the P different processors. Hence the locally stored patches contain obsolete data and are overwritten with the patches in the incoming buffer.

2.4 The Cartesian distribution

2.4.1 Cartesian distribution for parallel matrix-vector multiplication

As mentioned before, in regular (non-sparse) matrix-vector multiplication a Cartesian distribution is recommended ([1], chapter 4.4). We have also just seen that a progressive refinement iteration can be viewed as a matrix-vector multiplication, followed by some vector additions. Therefore, if we want to apply a Cartesian distribution to our current problem, it seems prudent to first review its basics regarding matrix-vector multiplication.

In parallel matrix-vector multiplications $Ax = b$, the matrix entries are divided over all processors, as are the entries of the input and output vectors. By a Cartesian distribution, the input (x) and output (b) vectors are distributed cyclically over all processors, i.e., the ith entry of one of those vectors is mapped to the $(i \bmod P)$th processor.

The Cartesian distribution partitions the matrix into rectangular areas and assigns those to separate processors. One processor may have more than one area assigned to it, but any area is of course linked to only one processor. Before we can define a Cartesian distribution, we must introduce a two-dimensional processor addressing scheme. If the number of processors P equals $q \cdot r$, such a scheme may be the following:

$$P(i,j) = P(i + j * q), 0 \leq i < q, 0 \leq j < r$$

We can now define how areas of the matrix are mapped to the processors. In the most abstract form, we can state that that the (i,j)th entry of the matrix M, M_{ij} for short, will be mapped to $P(\phi_0(i), \phi_1(j))$. Typically, one searches for the best choices for ϕ_0 and ϕ_1 to find a good load balance over all processors. However, this searching only is useful in the case of sparse matrices. Since all the matrix entries will have to be calculated if we need its value, we can state that in our case M is a dense matrix.

For dense matrices [1] recommends choosing $q = r = \sqrt{P}$ with $\phi_0(i) = (i \bmod \sqrt{P})$ and $\phi_1(j) = (j \bmod P)$ div $\sqrt{P}$ for optimal performance.

2.4.2 Parallel matrix-vector multiplication algorithm outline

By cyclic distribution, a processor with number $P(a,b)$ will only locally store the entries A_{ij} if $a = \phi_0(i) = i \bmod \sqrt{P}$ and $b = \phi_1(j) = (j \bmod P)$ div $\sqrt{P}$. Hence each processor has a unique submatrix $\hat{A}$ built up from the elements from A at row indexes I_i and column indexes I_j.Obviously, if we calculate $\hat{b} = \hat{A}x(I_j)$ at all processors we will obtain b by combining the different $\hat{b}$ while taking into account the different row indexes I_i. For calculating $\hat{A}x(I_j)$, we need all values from the input vector x at indexes I_j. Since x is cyclically distributed over all processors, communication is needed. This preliminary step is called the *fan-out*. Multiplying $\hat{A}x(I_j)$ can then be done locally. After multiplication we need to write the results to the final output vector; each processor needs to access b at the indexes I_i. This also results in communication, and this step is called the *fan-in*. A parallel matrix vector multiplication algorithm is given in algorithm 8, based on [1, Algorithm 4.5].

2.4.3 Cartesian distribution for parallel progressive refinement

We propose to keep using the original diagonal Cartesian distribution which is most effective for dense matrices. However, since we need to recalculate the matrix entries at each processor, an extra step is required.

Furthermore, we know that the multiplication vector has only P non-zero entries; we use P-Shooting. Therefore, at each iteration, we propose to store the corresponding P patches at each

Algorithm 8 A parallel matrix-vector multiplication algorithm

1: **for all** $j \in I_j$ **do**
2: Get x_j from the processor storing that element (*fan-out*)
3: **end for**
4: *Synchronise*
5: Calculate $\hat{b} = \hat{A}x(I_j)$
6: **for all** $i \in I_i$ **do**
7: Put the element of $\hat{b}$ corresponding to the row i of A to the processor responsible for storing b_i (*fan-in*)
8: **end for**
9: *Synchronise*
10: Take the sum of all received $\hat{b}$ values representing the same row, and put them in the final b vector

processor, bringing the total amount of storage to the order of $O(n/P + \tilde{n} + P)$, which is in most cases still equal to $O(n/P)$, given enough scene refinement.

Now, to calculate all entries in F^T, we only need to cyclically communicate the patches that each processor needs individually. For example, if we want to calculate the matrix entry F_{ij}, we require the ith patch data as well as the jth patch data. Let $A_0, \cdots, A_{P-1}$ denote the indexes of the shooting patches. Then only the local matrix entries F_{ij} for which $i \in \{A_0, ..., A_{P-1}\}, j \in \{1, 2, \ldots, n\}$ are necessary to perform the multiplication. Although the number of needed matrix entries has dropped from n^2 to nP, we still need all patches available in the scene since $i \in \{1, 2, \ldots, n\}$.

By Cartesian distribution, each individual column and each individual row of F is distributed over $\sqrt{P}$ processors, while each individual patch is distributed cyclically. Thus if we want to calculate Δ_i we have to multiply the ith row of A with the input vector x, $\sqrt{P}$ processors have to perform local multiplications and add up their results. This is the same situation as was presented in the cyclically distributed case, although there the rows were divided between P different processors. Nevertheless, we can use the same patch-rotating scheme as in algorithm 7 to calculate Δ_i in the Cartesian distributed case.

The fan-in superstep in algorithm 8 has to be applied in our case as well; as a result of our matrix distribution, processors will need exactly $\sqrt{P}$ non-local shooters. In contrast, the fan-out superstep is no longer necessary. Since we did not store our matrix explicitly, we needed to explicitly communicate patch information to calculate matrix entries on the fly. We did this by using the algorithm we constructed when using the cyclic distribution. This algorithm already stores the output values in the rotating patches; no results are stored locally. Hence, after $\sqrt{P}$ rotations, we automatically obtain the fully updated local patches.

So, as a result of the chosen Cartesian distribution, our algorithm needs an extra $\sqrt{P}$ communication steps in addition to the basic parallel matrix-vector multiplication algorithm. The fan-out superstep, however, is no longer necessary. Note the improvement over the cyclic distribution which required $O(P)$ communication steps.

The final Cartesian method is presented in algorithm 9, and the experimental program is based on that algorithm. Some points in the algorithm are based on algorithm 7 and algorithm 8. Remember the observation that any row i in the multiplication matrix A is stored by $\sqrt{P}$ processors $Q_i = \{q_1, q_2, \ldots, q_{\sqrt{P}}\}$. For any fixed processor q, Q_i q are the *row-neighbours* of processor q. If a processor stores elements from A on a set of rows I, then $Q_i = Q_j, \forall i, j \in I$; the row-neighbours of any processor are always the same for any row from A it stores elements of. This is because of the modulo operation in the chosen ϕ_0.

Also note that for this algorithm, processors will not have had received information about all patches in the scene. Therefore, it is required that for occlusion detection to function properly, we

have to have a structure like the hierarchical scene description.

Algorithm 9 A parallel algorithm through Cartesian distribution

```
1: Determine the patch s with the highest residual available locally
2: Send all other processors data on the local shooter, patch s
3: Synchronise
4: while Stop conditions have not been met do
5:    for all row-neighbouring processor i do
6:       Send s to processor i (fan-out)
7:    end for
8:    Synchronise
9:    for all locally stored patch p do
10:      Shoot energy from s to p (local shooting with locally present shooter)
11:      for all received shooter ŝ patches from other processors do
12:         Shoot energy from ŝ to p (local shooting with fan-inned shooters)
13:      end for
14:   end for
15:   for i = 1 to √P − 1 do
16:      Send the data in the outgoing buffer to the incoming buffer of the next row-neighbouring
         processor (Cycling of local patches among row-neighbouring processors enables implicit fan-out
         during multiplication)
17:      Synchronise
18:      for all patches p in the incoming buffer do
19:         Shoot energy from the local shooter s to p (shooting with locally present shooter on non-
            locally stored patches; implicit fan-out)
20:         for all received shooter ŝ patches from other processors do
21:            Shoot energy from ŝ to p (shooting with fan-inned shooters on non-locally stored patches;
               implicit fan-out)
22:         end for
23:      end for
24:      Move the patches from the incoming buffer to the outgoing buffer
25:   end for
26:   Send the data in the outgoing buffer to the incoming buffer of the next row-neighbouring
      processor
27:   Synchronise
28:   Replace the local patch data with the data in the incoming buffer
29:   Set the residual energy vector of the local shooter U_s to zero
30:   Determine the patch s with the highest residual available locally
31:   Send all other processors data on the local shooter, patch s
32: end while
```

Memory usage

The memory usage of this algorithm is of the same order as for the algorithm derived from the cyclic distribution; $O(\tilde{n} + 2n/P)$. More precisely, we also need to store $\sqrt{P}$ shooting patches at each processor instead of just 1, but the memory requirements for this pale with respect to $\tilde{n}$ and n/P.

Communication

The communication volume is of order $g \cdot O(P + \sqrt{P}(n/P))$ per iteration. The only and most important difference with the cyclic algorithm is the reduction in the number of synchronising steps; from P to $\sqrt{P}$.

2.5 An alternative algorithm using additional storage

If we step back we see that the extra communication is required because we have assigned to some processors entries F_{ij} while i is not locally stored. However, if we force i to be stored at all processors that have dealings with that patch, this communication could be avoided.

This redundant storage increases the required memory per processor to an order of $O(\tilde{n} + 2(n/\sqrt{P}))$, and it completely eliminates all extra communication that was needed for the previously described Cartesian distribution. However, note that this scheme requires a new kind of communication to take place; when patches are stored at multiple locations, they always have to be kept in sync. Hence if a processor updates a given patch, this update has to be propagated to all other processors also storing this patch. This was not an issue with the Cartesian distribution.

Because we do not store patch information explicitly, updating patch information translates to updating the radiosity vector. Hence the extra communication is of order $g \cdot O(nP/(\sqrt{P} - 1))$. This actually is worse than the communication required by the cyclic distribution. Thus we see that storing patch data multiple times actually leads to a worse performing algorithm. We have not experimentally researched this algorithm.

Chapter 3

Experiments

3.1 Parameters

A few parameters have to be set before an experiment can take place. We will now consider each parameter and clarify what purpose it serves. Then we will proceed with experiment descriptions, and after that the experiment results.

3.1.1 Number of processors

By our design choices, we have two active restrictions on the possible processor counts. Because we have chosen a Cartesian distribution using $q = r = \sqrt{P}$, the processor counts are limited to square numbers.

The second restriction is due to scalability. If n is the total number of patches, we wanted each processor to store n/P patches. We also determined that the patch subdivisions of each square resulted in $n_i = 4^{d_i}$ patches on any square i, with d_i an positive integer.

This means that if we have $\tilde{n}$ square surfaces, $n = \sum_{i=1}^{\tilde{n}} n_i$ and thus follows that $n = 4 * \sum_{i=1}^{\tilde{n}} 4^{d_i - 1}$; n always is a multiple of 4. Therefore, the second restriction should be that P mod $4 = 0$. We can then guarantee that each processor has the same amount of patches.

Program experiments have been done for $P = 1, 4$, and 16.

3.1.2 Refinement level

The refinement level prescribes how many times a square in the scene is refined. Each refinement level splits existing squares into 4 smaller squares. The refinement level is in essence the value d_i from the previous formula $n_i = 4^{d_i}$. The program demands that $d_i = d_j$ for all $0 \leq i, j \leq n$; all surfaces are divided in an equal amount of patches.

3.1.3 Total residual stop value (TRes)

When the total residual value of all vectors is below the value defined here, the algorithm will cease iterating. This value essentially represents the heuristic that if the amount of residual energy is low enough, the change it would bring to the scene if all that energy is distributed correctly, would be indiscernible.

3.1.4 Single residual stop value (SRes)

When the maximum value of all residual values drops below the value defined here, the algorithm will also stop iterating. The heuristic here is that if the highest residual is low, the changes

it will infer on the total residual will be even less. Therefore, continuing to iterate will lead to slow convergence and over-the-top running times.

It can be that while making one iteration does not make a lot of difference to the final scene, it can be true that performing many iterations would have made a lot of difference to the final scene. Therefore, care must be taken in choosing an appropriate value here.

3.1.5 Maximum number of iterations

Most of the time, it is hard to predict the running time of any iterative algorithm on any previously uninvestigated settings. What however does remain rather constant, is the time it takes to do a single iteration. So if a time upper bound is necessary, it is only necessary to determine iteration speed on the current settings. Then, a maximum number of iterations is easily inferred and this variable can be set accordingly.

This comes particularly in handy on parallel machines which need upper time bounds on any incoming batch jobs.

3.2 Experiments

For doing parallel experiments, two machines were available. The first (named Grit) is the home computer of the Institute of Mathematics, at the University of Utrecht. It is not quite state of the art, and was mostly used as a testing environment. Nonetheless, some small experiments have been performed using this computer. Grit only has two processors however. It could simulate a BSP environment of four processors, but that seriously hits performance. To be complete, those experiments have also been included here (number 9, 10, 11 and 12).

The second computer available was a SGI Altix 3700 named Aster, consisting of 416 1.3GHz Intel Itanium 2 processors. On this computer, the main experiments have been performed. Below is a table of selected experiments and their results:

Experiment	P	Ref. level	TRes	SRes	Iterations done	Time taken (s)
10 (Grit)	1	2	1	0.1	62	1.07779
9 (Grit)	4	2	1	0.1	16	12.6196
11 (Grit)	1	3	1	0.1	259	15.8189
12 (Grit)	4	3	1	0.1	50	49.5227
15	1	4	10	1	256	9.1857
16	4	4	10	1	64	1.85162
18	4	5	500	0.5	256	25.5067
17	16	5	500	0.5	64	7.5179
–	1	6	500	1	takes too long	takes too long
21	4	6	500	1	1024	392.049
22	4	5	100	0.05	885	92.4416
23	16	5	100	0.05	258	28.8044
24	4	6	100	0.05	3967	1639.14
25	16	6	100	0.05	1133	479.941
new_ex_1	16	7	100	0.01	5998	11505.3

3.3 Selected output

Using 4 processors, refinement level 5, Tres 500 and SRes 0.5 we obtain:

```
...
(256) Total residual: 30440.2 Difference: 29996.7
```

```
      Greatest Residual: 7500 Difference 0
STOP Total residual: 443.54 Difference: 29996.7
     Greatest Residual: 0.39875 Difference 7499.6
...
```

Using 16 processors, refinement level 5, Tres 500 and SRes 0.5 we obtain:

```
...
(64) Total residual: 120505 Difference: 119984
     Greatest Residual: 7500 Difference 0
STOP Total residual: 520.549 Difference: 119984
     Greatest Residual: 0.468616 Difference 7499.53
...
```

Using 4 processors, refinement level 5, Tres 500 and SRes 0.5 we obtain:

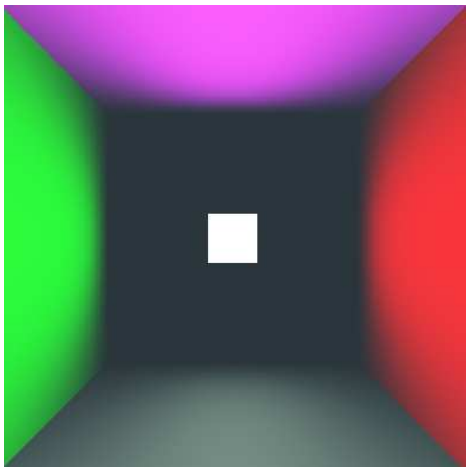
```
...
(1024) Total residual: 31770.7 Difference: 29996.7
       Greatest Residual: 7500 Difference 0
STOP Total residual: 1773.99 Difference: 29996.7
     Greatest Residual: 0.399353 Difference 7499.6
...
```

Using 4 processors, refinement level 6, Tres 500 and SRes 0.5 we obtain:

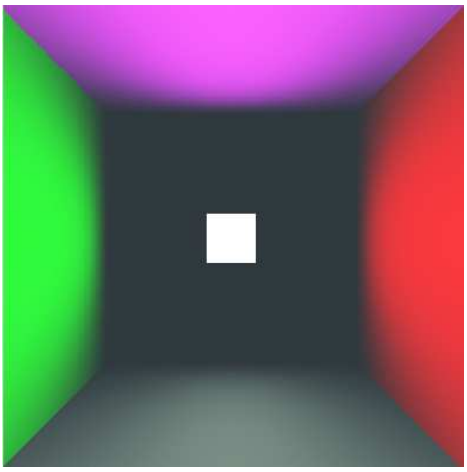
```
...
(3967) Total residual: 323.904 Difference: 0.156016
       Greatest Residual: 0.0500152 Difference: 0.00010892
STOP Total residual: 323.758 Difference: 0.145569
     Greatest Residual: 0.0499937 Difference 2.14943e-05
...
```

3.4 Selected result images

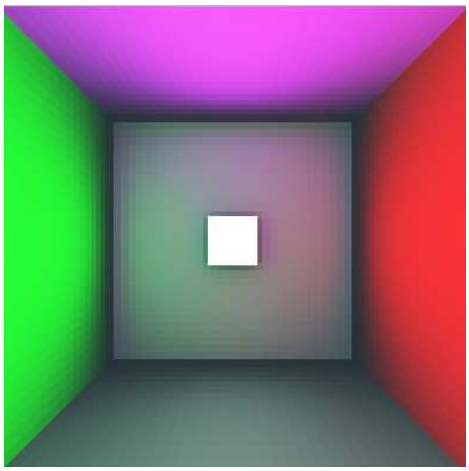
There are three selected result pictures rendered from the standard viewpoint. These can be seen in figure 3.1.



(a) Resulting scene from experiment 17, which used 16 processors



(b) Resulting scene from experiment 18, which used 4 processors



(c) Resulting high-res scene from experiment 23

Figure 3.1: Selected result pictures

Chapter 4

Conclusions

Conclusions can be drawn regarding three globally different aspects of the program. At first there is of course the running time of the algorithm; we did strive for the parallel algorithm to be as fast as possible compared to the sequential algorithm.

The number of iterations done per experiment is an issue also. It testifies to the effectiveness of the P-shooter algorithm in general; how much more effective or ineffective is a single iteration step compared to a normal (sequential) progressive refinement step?

Lastly, a point of discussion is of course the rendered images the algorithm produced. The quality should be coherent with the experiment parameters, and no oddities should be present. Also, a parallelly rendered scheme should not differ from a sequentially rendered scene.

4.1 Running times

If we look at the running times for the experiments ran on Grit, we see horrible results for the 4-processor experiments. This is however due to the fact that Grit actually is a computer with 2 actual processors, as mentioned before. The time values thus do not hold much relevant information for our experiments.

The experiments performed on Aster did in fact take place on a real P-processor environment. The time values here thus do hold important information, and we see instantly that experiments ran using multiple processors finished earlier as well; this is always nice to see confirmed after one is done programming a parallel algorithm.

For experiments 15 and 16, we see that an increase from 1 processor to 4 processors gives a speedup of approximately 4.96. With experiments 17 and 18, a speedup of 3.39 is witnessed for an increase from $P = 4$ to $P = 16$. For experiments 22 and 23 we have the factor 3.21, and experiments 24 and 25 show a speedup of 3.42. From these data we can conclude that rather independently of stop conditions or refinement levels, the speedup remains near 3.4. It must be noted that the speedup > 4 witnessed for experiments 15 and 16 is not due to an algorithmic effect. This superlinear speedup can only have been attained due to cache effects; each processor needs to store less patches which may cause faster calculations if they can fit more easily into cache memory instead of main memory.

The use of more and more processors seems to slow down the algorithm a lot. The communication volume is of order $(\sqrt{P} - 1)(n \text{ div } P)$, which is still much smaller than $(P - 1)(n \text{ div } P)$. The cyclic distribution was not experimented with; so we cannot compare the two methods.

Refinement level 7 is the largest experiment our software is tested for. Remember that in the matrix radiosity sense this corresponds to iteratively solving a dense linear system consisting of $O((7 \cdot 4^7)^2)$ not-explicitly stored elements; our experimental software did this in less than 3.5

hours (*new_exp_1* in the experiments table in the previous chapter). This may be considered a satisfactory result, at the time of writing.

4.2 Iteration steps

Although the experiments done on Grit took a lot more time due to the simulation of 4 processors, the iteration steps done for an experiment are equal to the number of steps it would have taken if the experiment had run on Aster. So for considering the results on iterations, we can take into account the results from Grit.

We would expect the number of iterations multiplied by the number of processors used, to increase with larger processor numbers. This is because the order of shooting in the original progressive refinement algorithm is disturbed. As a result, the *P*-shooter algorithm may elect patches to shoot their energy, while other patches may have been better candidates, if it only had already received the energy from another elected shooter.

In the following table, the number of iterations is multiplied by the number of participating processors. It shows if the hypothesis previously made holds true:

Experiment	Processors (P)	Iterations done (I)	$P \cdot I$
10	1	62	62
9	4	16	64
11	1	259	259
12	4	50	200
18	4	256	1024
17	16	64	1024
22	4	885	3540
23	16	258	4128
24	4	3967	15868
25	16	1133	18128

In general, the table confirms the hypothesis. The total number of iterations done on all processors while using multiple processors, does indeed result in an increase compared to the experiments using a single processor.

There is one exception for experiments 11 and 12; the number of iteration steps actually became less. Apparently, *P*-shooting caused to meet the stop conditions a lot sooner than the sequential algorithm; however, it must be noted that the results from experiment 11 are better in the sense of residuals than those of 12. It is not possible for *P*-shooting to come with better results than the sequential algorithm in less total iterations.

The third section of the table displays an oddity; the total number of iterations are all equal, no matter the amount of participating processors. This is because of the experiment settings (namely the maximum single residual stop condition); the next section will explain this fully.

The fourth section displays the same expected results. The 4-processor algorithm however shows a total iteration amount which is much smaller than the 16-processor iteration count. It thus seems that the larger the P in the *P*-shooter algorithm, the less effective the algorithm becomes.

4.3 Result pictures

Not only running times and iteration counts are important in these experiments. The result pictures had to look good too. Selected in the previous chapter were three pictures to show that

indeed the renderer functions adequately.

One of the first questions to be answered is if a scene rendered with 4 processors indeed looks similar to the same scene rendered with 16 processors. The final colour versions do not have to, and eventually did not, have to be the same. This is because of the difference between normal progressive refinement and P-shooting; it disturbs the normal order of shooting, and also, even if the order remained the same, the residual vectors would differ because the shooting now is simultaneous.

If we look at the figures 11 and 12, no difference is to be found with the naked eye. This is also true for the many other experiments done. We therefore can conclude that the use of multiple processors does not introduce errors in the resulting pictures.

We do however see a big difference between the figures 3.1(a) and 3.1(b) on one hand, and figure 3.1(c) on the other. The first two pictures showed a realistic lightfall from the source to the side squares. This manner of light distribution is for example very similar to a light source near a wall.

What is *not* realistic, though, is the black 'roof' in the picture. The light source is of course quite visible, but the square behind it is not. In realistic situations, the back surface would have been illuminated too, due to reflection via the other surfaces.

What went wrong here, is the value set for the Maximum single residual stop condition (SRes). For this experiment, it was set to 0.5. However, if we look at the iteration progress, we can conclude 0.5 is by far not a good value. Let us take a look at selected output from that experiment:

```
...
(256) Total residual: 30440.2 Difference: 29996.7
      Greatest Residual: 7500 Difference 0
STOP Total residual: 443.54 Difference: 29996.7
     Greatest Residual: 0.39875 Difference 7499.6
...
```

Output snippet from experiment number 18*, as also seen in the previous section.*

What happened here is that the renderer just finished shooting all the residual energy from the light emitting patches at iteration number 255. After that, the algorithm must choose a shooter from the patches who received energy from that emitting patch. This is in fact the process that should light the background.

However, the patch with the most energetic residual has a value of approximately 0.39875; which is less than 0.5. This causes the algorithm to stop, way too soon for our tastes. Note also the rather large amount of total not-processed residual energy; also an indication the current solution was not near the final solution yet.

The solution here then is of course to choose a lower value for SRes. Experimentation showed that 0.05 already gave good results. Figure 3.1(c) shows a scene rendered with SRes $= 0.05$ (and $P = 16$). We see there that the background is well-lit, and also, among the four sides one can see colours mingling better as well. This particularly shows on the bottom side; its right side colours slightly red and the left side slightly green.

Note also the shadow directly behind the light emitting surface; this is also realistic because much reflecting energy is occluded by the light source. Also, because the light source hovers slightly above the back surface, a large shadow is cast to the lower sides; the patches there have no direct contact with the light source, and the distance or angle (depending on the other patch) are too unfavourable for good reflectance of energy from other surfaces.

4.4 Future work

4.4.1 Incorporate more features

To prevent unnecessary levels of detail, adaptive subdivision could be added to the algorithm. However, this will cause the number of patches to increase while the algorithm is running. Thus mechanisms should be in place to do patch distribution while the program is running, while maintaining optimal load balance.

Also, for speedups in occlusion detection, using binairy space partitioning or a Z-buffer is recommended.

4.4.2 Experiment with other parallel algorithms or distributions

It is always worth comparing different algorithms with each other. In general, it is expected that for example the double storage algorithm would work a lot faster, at the cost of a larger memory usage. How much faster such an algorithm would be seems to be a very interesting question.

Also worth investigating may be the Parallel Iterations algorithm. Perhaps trading the convergence speedup (that *P*-shooting gave) for a better shooter selection sequence proves worthwile.

Bibliography

[1] Rob H. Bisseling, Parallel Scientific Computation - A structured approach using BSP and MPI, first edition, Oxford University Press (2004).

[2] Cevdet Aykanat, Tolga K. Çapin, Bülent Özgüç, A parallel progressive radiosity algorithm based on patch data circulation, Comput. & Graphics, Vol. 20, No. 2 (1996).

[3] Steven Gortler, Michael F. Cohen, and Philipp Slusallek (1993), Radiosity and Relaxation Methods

[4] Greg Coombe and Mark Harris (2005), Global Illumination Using Progressive Refinement Radiosity, Article taken from ch. 39 of "GPU Gems 2, Addison-Wesley"

[5] Paul Nettle (visited 2006), Radiosity in English II: Form Factor Calculation, http://www.gamedev.net/reference/articles/article653.asp

[6] Christophe Renaud, François Rouselle, Fast massively parallel progressive radiosity on the MP-1, Parallel Computing 23 (1997)

[7] BSP Worldwide (July 2006), http://www.bsp-worldwide.org

Appendix A

File descriptions

The experimental radiosity renderer takes as input a scene file (by default this is *./scene.txt*, and gives as output a viewer file *./renderedScene.txt*. Below, the build up of both files is defined.

A.1 Input scene file

The current version of the radiosity renderer only works with square surfaces. Therefore, an input file should simply define a series of squares. Each individual square must have an origin vector, a normal vector, a direction vector, a reflectance (colour) vector, and some initial residual vector (for light sources). Thus, a valid input scene file is defined to be file containing any number of repetition of the following code:

```
1: square {
2:     origin < 0 , 0 , 5 >
3:     normal < 0 , 0 , -1 >
4:     direction < 1 , 0 , 0 >
5:     reflectance < 0.7 , 0.7 , 0.7 >
6:     residual < 0 , 0 , 0 >
7:     length 10
8: }
```

A.2 Output viewer file

As opposed to the radiosity renderer, the viewer application has to render images from a certain viewpoint. Hence it requires a defined camera position and target, along with data regarding the size of the picture it should render, and any optional image quality parameters such as *Anti Aliasing* or *Gamma* parameters. The basic syntax remains similar to that of the input file. A small example output file would be the following:

```
1: width 400
2: height 400
3: gamma 1.0
4:
5: camera {
6:     origin < 0 , 0 , 4.9 >
7:     viewUp < 1 , 0 , 0 >
8:     target < 0 , 0 , -1 >
9:     sampleroot 1
```

```
10:     }
11:
12: square {
13:     origin < 0 , 0 , 5 >
14:     normal < 0 , 0 , -1 >
15:     direction < 1 , 0 , 0 >
16:     reflectance < 0.7 , 0.7 , 0.7 >
17:     residual < 0 , 0 , 0 >
18:     length 10
19:     material {
20:         colour < 0 , 1 , 0 >
21:     }
22: }
```

A.3 Combining multiple output files

In the previous example the output file described a single square with a green colour. When rendering in parallel, all processors calculate colour values for their own uniquely stored squares. These processors will write their output to separate files, which can later be simply concatenated to produce the complete rendered scene.

After concatenation, the viewer preferences, colour information[1] and camera properties are added to the front of the rendered scene to obtain a valid viewer scene file[2].

A.4 Experimental scene file

The following is a listing of the scene file used in the experiments (*scene.txt*):

```
1: square {
2:     origin < 0.0 , 0.0 , -4.5 >
3:     normal < 0 , 0 , 1 >
4:     direction < 1 , 0 , 0 >
5:     reflectance < 0.7 , 0.1 , 0.7 >
6:     residual < 50 , 50 , 50 >
7:     length 2
8: }
9:
10: square {
11:     origin < 0.0 , 0.0 , -5.0 >
12:     normal < 0 , 0 , 1 >
13:     direction < 1 , 0 , 0 >
14:     reflectance < 1 , 1 , 1 >
15:     residual < 0 , 0 , 0 >
16:     length 10
17: }
18:
19: square {
20:     origin < 5.0 , 0.0 , 0.0 >
21:     normal < -1 , 0 , 0 >
22:     direction < 0 , 0 , 1 >
23:     reflectance < 0.2 , 0.2 , 0.17 >
24:     residual < 0 , 0 , 0 >
25:     length 10
26: }
27:
28: square {
29:     origin < -5.0 , 0.0 , 0.0 >
30:     normal < 1 , 0 , 0 >
31:     direction < 0 , 0 , 1 >
32:     reflectance < 0.7 , 0.1 , 0.7 >
33:     residual < 0.0 , 0.0 , 0.0 >
34:     length 10
35: }
36:
```

[1] See appendix B

[2] This procedure can easily be accomplished by redirecting streams. For example, on any *NIX: *cat prefs >> cat colours.txt >> cat cameraloc.txt >> cat renderedScene1.txt >> ... >> cat renderedSceneP.txt >> finalOutput.txt*, where *prefs*, *colours.txt* and *cameraloc.txt* are textfiles containing settings for the viewer, and the *renderedScene*i*.txt*, $1 \leq i \leq P$ are the individual outputs of the processors the parallel algorithm was running on.

```
37: square {
38:     origin < 0 , 5 , 0 >
39:     normal < 0 , -1 , 0 >
40:     direction < 0 , 0 , 1 >
41:     reflectance < 0.7 , 0 , 0 >
42:     residual < 0 , 0 , 0 >
43:     length 10
44: }
45:
46: square {
47:     origin < 0 , -5 , 0 >
48:     normal < 0 , 1 , 0 >
49:     direction < 0 , 0 , 1 >
50:     reflectance < 0 , 0.7 , 0 >
51:     residual < 0 , 0 , 0 >
52:     length 10
53: }
54:
55: square {
56:     origin < 0 , 0 , 5 >
57:     normal < 0 , 0 , -1 >
58:     direction < 1 , 0 , 0 >
59:     reflectance < 0.7 , 0.7 , 0.7 >
60:     residual < 0 , 0 , 0 >
61:     length 10
62: }
```

Appendix B

Dealing with colour values in radiosity

In most graphics applications, we use different strengths of the three primary colours (red, green, blue) to generate all possible colours. Usually, we let the strength of one such colour be given by an integer between 0 and 255, where 0 means no strength, and 255 full strength.

However, as we have seen, radiosity will assign to surfaces amounts of energy dependent on reflection and the initial energy of light sources; there is no guarantee the final energy values for any colour is in $[0, 255]$. Hence, if we want to generate images, we need to map the energy levels to a specific value between 0 and 255. There are several ways to achieve this. The different methods are visualised in figure B.1

B.1 Linear

Suppose we have somehow obtained the maximum energy level M of all surfaces in the rendered scene. Surely, for all other energy levels l we have that $0 \leq l \leq M$. Thus let us define the linear mapping $f_L : \mathbb{R}([0, M]) \to \mathbb{R}([0, 255])$ by

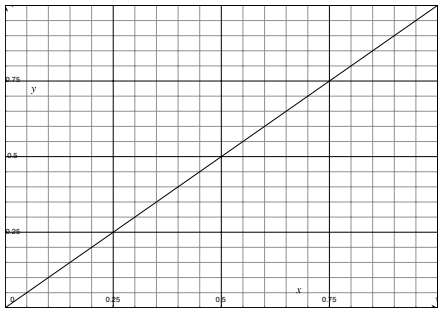
$$f_L(l) = 255\frac{l}{M} \tag{B.1}$$

B.2 Affine

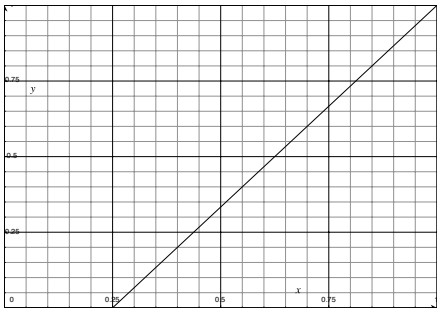
Due to contrast considerations, we may want to map the minimum energy level m of all surfaces in the scene to zero. This leads to the following map:

$$f_A(l) = \begin{cases} 255\frac{l-m}{M-m}, & \text{if } l \in [m, M] \\ 0, & \text{if } l < m \end{cases} \tag{B.2}$$

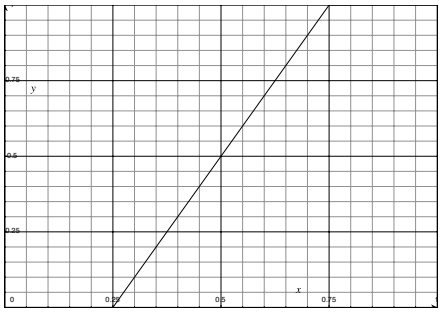
since the step size $\frac{1}{M-m} \geq \frac{1}{M}$, we see that contrast indeed has improved. Of course, manually setting values for m and M may lead to even more pleasing results; however, such optimisations are a different subject entirely.



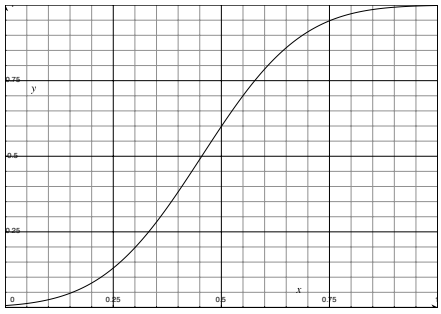
(a) Linear mapping. The final colour value is linearly dependent on the energy level between $(0, M)$



(b) Affine mapping. The final colour value is linearly dependent on the energy level between (m, M). m is at $\frac{1}{4}M$.



(c) Manual mapping. The final colour value is linearly dependent on the energy level between some manually set $(\hat{m}, \hat{M})$



(d) Normal mapping. The final colour value is statistically dependent on the energy level between $(0, M)$.

Figure B.1: Various schemes for mapping colour values. The values on the x-axis represent a fraction of M, thus 1 maps to M and $\frac{1}{2}$ to $\frac{M}{2}$. The values on the y-axis represent fractions of the final colour value; 1 maps to 255 and $\frac{1}{4}$ to $\frac{255}{4}$, etcetera.

B.3 Normal

The third and last method we will discuss is of a more statistical nature. Suppose we want to differentiate between energy levels in the mid-range more than energy levels near 0 (or even m) and M. Then we may want to map according to a function which looks like the cumulative distribution function of the normal distribution, see figure B.1(d). If we define

$$f_N(l) = \Phi(\frac{l-\mu}{\sigma}) \tag{B.3}$$

we obtain a suitable mapping function if we take for μ the average energy level in the scene, and σ the standard deviation.

B.4 Conclusion

Colour mapping is actually a large subject in computer graphics, and what is presented here are only simple heuristics; for example, the linear mapping presented here is a simple variant

on lowering colour brightness. The affine and manual mapping are in turn similar to contrast enlarging manipulations. Also, gamma-correction is a colour mapping of this type with mapping function $f_\gamma(l) = l^\gamma$.

During experiments, it became evident that the normal mapping f_N worked sufficiently well. Also, all figures previously presented are rendered using f_N as mapping function.