Fuzzy Rendering for High-Quality Image Generation

Andrew James Harman Willmott

A thesis submitted in partial fulfilment of the requirements for

the degree of Master of Science in Computer Science,

University of Auckland, 1993.

Abstract

This thesis is concerned with the field of high-quality image rendering. It describes research into a system for rendering a scene so that the picture produced is accurate to some specified limit. We refer to the system as “Fuzzy Rendering,” because it deals with ranges of metrics, rather than precise values.

An overview of the concepts behind the system is given, and previous research on it is summarised. The problems that have to be solved in order to produce such a renderer are pointed out.

A method for finding the amount of light transferred between two surfaces is described. Several methods are proposed and tested for finding the variation in this light over an area of a surface.

The design of a limited fuzzy-rendering system is presented, as well as some aspects of the implementation of that design. The performance of this renderer is then analysed. Finally, the achievements of the research are summarised, and directions for future research suggested.

Acknowledgments

I’d especially like to thank the following people for their support and assistance over the course of this thesis.

My supervisor, Richard Lobb, for proposing an interesting topic, and for his many helpful suggestions along the way. Without his guidance this thesis would not have been possible.

My family, who I’ve seen so little of in the past few weeks, for their support over the years. Also, my grandparents for their constant encouragement and interest.

All the graduate students, for making this year such an enjoyable one, but especially to my officemates, and my fellow thesis students.

Thanks are due also to Clare West for her proof-reading, and Edouard Poor for checking chapter one for me.

To the staff at Auckland University, for making my time at this university so pleasant, and teaching me a few things along the way.

Finally, to the technical staff, who helped when things went wrong, and who thoughtfully taught me how to juggle during the final preparation of this manuscript.

Preface

A few months ago a spoof of the annual SIGGraph[1] conference appeared on the Usenet network. It included the following supposed seminar:

> "State-of-the-Art in Illumination Models" by Don Greenberg.
>
> Dr. Greenberg will review illumination models that for two decades have maintained the Law of Constant Rendering Time, which states that the time needed to render a high-quality image shall be one full day, regardless of the speed of the hardware.
>
> Just a few years ago, ray tracing a surface would take all day. However, that is no longer true, and so more complex illumination models are needed. According to the new treatise by Greenberg, Torrance, Sparrow, and Cook entitled "Wait, It's Not That Simple", current research considers each diffusely reflecting surface patch to be an irregular assembly of microfacets.
>
> The microfacets must be ray traced to get reflection coefficients. If this doesn't take long enough, then each microfacet itself can be considered as an assembly of smaller facets. This subdivide-and-publish paradigm should insure that illumination methods will defeat the hardware for years to come.

It does seem that computer scientists have no difficulty in keeping the latest generations of computer workstations busy. This is in spite of "Joy's Law,"[2] which states that such workstations double in speed every year, and has remarkably held true ever since it was stated.

In Computer Graphics, this excess computing power has largely been absorbed in the never-ending pursuit of realism, which is the goal of the field of "high-quality image rendering." Over the last ten years, the main issue in this field has been the problem of modelling the flow of light through a scene, so that the scene can be shaded in a realistic way. The first such "global-illumination" models made quite strong assumptions about how surfaces reflect light. Recently more advanced models have been developed that allow arbitrary surface-reflection, with varying degrees of success.

We are now reaching the stage where it is possible to simulate quite accurately the flow of light through a scene. Thus it makes sense to start asking how good the images generated are. A traditional complaint about this field of Computer Graphics has been the vagueness of the goal of realism. Over much of the past twenty years the quality of an image has largely been measured by whether or not it "looks right."

[1]Special Interest Group in Graphics.
[2]Formulated by Bill Joy of Sun microsystems.

The purpose of the Fuzzy Rendering system presented in this thesis is to produce a renderer that is capable of handling scenes with arbitrary surface-reflections, and which actively tracks the error in its results, in order to estimate the accuracy of the picture it is producing. At any point in time, the error in different parts of the renderer’s model of the scene is used as an indication of where it should work to improve that model. The hope is that this will result in a renderer that can produce an image to a specified level of accuracy, and that will minimise the work done to produce that image.

The thesis can be split up into a number of sections:

Overviews

Chapter one summarises the current state of the field. A short history of the early work done in Computer Graphics is given, and we then look at those algorithms that are important in the area of high-quality image rendering. Chapter two looks at the fuzzy rendering system. The concepts involved and the issues in its development are presented.

Investigation Into Light Flow Between Surfaces

Chapters three and four look at the most important of these issues, the calculation of how light flows between surfaces. A framework for calculating such light flows is presented, and several methods for finding bounds on the variation in light across part of a surface are developed.

Design and Implementation of the Renderer.

In chapters five and six we look at the renderer that was built during the course of this research. Its design is presented, as well as several techniques that were used to monitor its operation.

Testing the Renderer

In chapter seven a number of different aspects of the renderer are analysed, including its performance.

Conclusions and future work.

In chapter eight the achievements and implications of the thesis are presented. Possible future areas of research into the fuzzy rendering system are commented on.

Contents

CHAPTER 1

An Introduction to High Quality Image Rendering

1.1 Introduction

High-quality image rendering is concerned with accurately reproducing real-life images by computer simulation. The way objects reflect light, their shapes, their positions, where they are being viewed from — all of these things must be considered when attempting to meet this goal. The Computer Graphics field itself encompasses the display of both two-dimensional and three-dimensional images. In this thesis, we restrict ourselves to the latter.

We start by taking a look at the uses of Computer Graphics, and also some of the standard terminology of the subject. An overview of the formative years of the field is given, and we then survey the algorithms that have been developed for high-quality image rendering.

1.1.1 Uses of Computer Graphics

Computer Graphics has a number of important uses:

- We can see things that couldn't otherwise be observed. One of the biggest applications of this is Scientific Visualisation, where data gathered from experiments can be displayed in ways that help its interpretation. The medical field has also used the technology in order to display the data from tools such as CAT scans.
- We can see things that don't exist yet. This is useful in areas such as computer-aided design and drafting. An architect's building plan, designs for cars, aircraft, even microchips, can all be first displayed on a screen to help in the design process.

- We can precisely control the scene we are rendering. This can be especially useful in the entertainment/advertising industries, where computer animations allow manipulations of objects and “actors” that would be impossible with real camera-work.

In many ways Computer Graphics is a dichotomy because it has both scientific and artistic applications. On the scientific side, the aim is usually to model a possible situation accurately, so that decisions can be made about it. The emphasis is on the rigorous simulation of physical phenomena, and the presentation of the results of this simulation.

On the artistic side, it is interesting to compare the current progress of Computer Graphics with that of painting or sculpture. The first objective in any artistic field has always been realism. It is usually only once this has been achieved that artists start to use the medium as a tool for the expression of ideas. In painting realism was followed by impressionism, and later surrealism, both of which started to concentrate more on the subject matter, rather than aiming for a precise reproduction of nature. Once computer science has also achieved this goal of realism, it too can move beyond mere reproduction.

1.1.2 Basic Concepts

The starting point in Computer Graphics is the *scene*, which is a mathematical description of some three dimensional world. Usually this description is built up in a hierarchical manner from a number of *primitives* , such as spheres, cubes or polgons. For instance, a chair might be constructed from a stool and a back, with the stool in turn being constructed from four legs and a seat, as in figure 1.1.

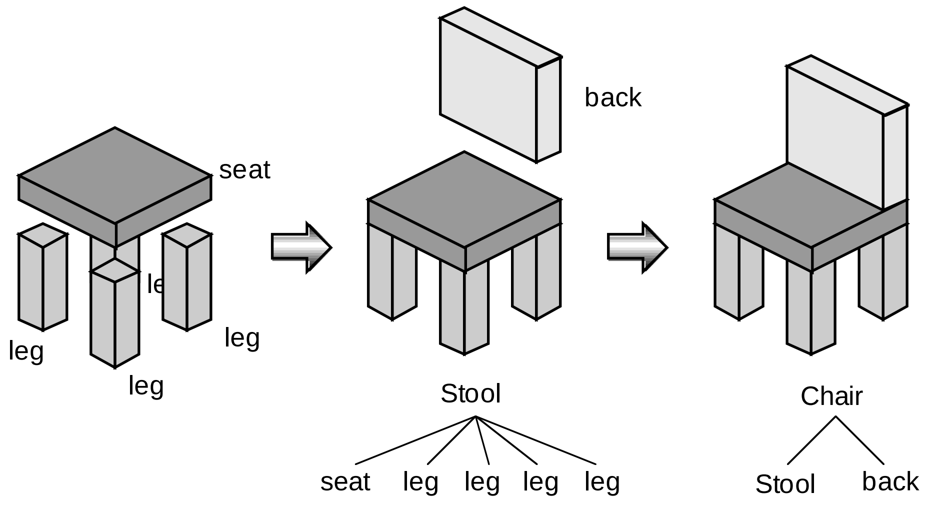


Figure 1.1 Scene construction

Once we have such a scene, we try to *render* it, which involves calculating the two-dimensional picture that we would see if we viewed the scene from a certain point in space, commonly called the *viewpoint*. Generally we *transform* the scene so that the viewpoint lies at the origin of a coordinate system, with the x and y axes lying horizontally and vertically, and the z-axis measuring how far away objects are, as in figure 1.2[1].

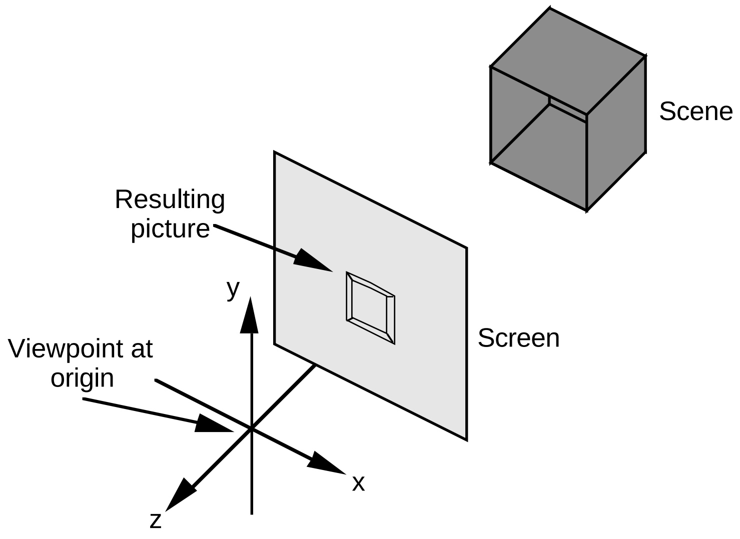


Figure 1.2 Viewing a scene

For obvious reasons we refer to the act of converting the three-dimensional scene into a two-dimensional picture as *projection*. Once this has been done, we then have to decide how to *shade* the objects in the picture to get the desired image.

1.2 A Brief History of Computer Graphics

As in many fields, much of the history of Computer Graphics has been a case of using previous results as building blocks in order to construct more ever more complex work. In turn it is the algorithms of today that dictate what can be done tomorrow. At the same time, the research has often been driven by the hardware available.

There are three major areas of research in Computer Graphics:

- Determining how a particular scene object should look on screen. (Rendering.)
- Exploring different ways of representing the surfaces of scene objects. (Surface modelling.)
- Deciding how to shade surfaces so they look “realistic”. (Illumination.)

All overlap to some degree , but can be seen as independent aspects of the field.

[1]The picture shown is for a right-handed coordinate system. Left-handed coordinate systems, where the z-axis points the other way, are also common.

1.2.1 The First Steps

The earliest work on rendering scenes (in the early 1960s) concentrated on the act of projecting a three-dimensional scene onto the two-dimensional computer screen. These renderings were 'wire-frame', in that objects were represented as polyhedrons, the edges of which were drawn on the screen. (See figure 1.4A.) Much of the projection problem for line-drawings had already been covered in the architectural and art fields, with the development of orthographic and perspective projections. For computer work, these were soon formalised as linear and non-linear vector transformations, and today they can be found in most introductory texts on the subject.

While such wire-frame images were perfectly acceptable for the large CAD/CAM systems that were developed in the mid-60s, they didn't present a particularly "realistic" image of solid objects, because you could see through them. This soon led to work on *hidden-line removal*, which involved identifying those line segments obscured by other parts of the scene, and removing them from the set of lines to be drawn. The earliest paper on this topic was published in 1963 by Roberts [Robe63], and presented a multi-staged approach that, after eliminating as many obvious cases as possible, clipped each line against every polygon in the scene that might obscure it.

1.2.2 Solid Scenes

During the sixties, the predominant type of *display device* was the *vector* display, which stored a list of lines, and displayed them on the screen. The late sixties and early seventies saw the popularisation of inexpensive *raster* displays, based on television technology. In such displays, pictures are made up of a rectangular grid of dots called *pixels*. A separate colour or intensity value for each of these pixels is stored in a *frame-buffer*, and this buffer is redisplayed on the screen every sixtieth of a second or so, as with television pictures. This kind of display is capable of showing any kind of primitive once an algorithm for identifying which pixels it covers has been produced, and the field of computer graphics soon began to move away from the line-drawings that had dominated it until then.

By 1965 Bresenham had developed a classic algorithm for *scan-converting* lines into a frame-buffer [Bres65]. It was given this name because the algorithm stepped through the horizontal *scan-lines* of the image one by one, keeping track of where the line crossed the current scan-line, and updating that position every time the next scan-line was crossed, by the use of incremental techniques. A similar algorithm was developed for polygons. It used the same techniques to keep track of the edges of the polygon that crossed the current scan-line, and at each new line filled in the spans between these edges.

The renderings of solid polyhedrons that these surfaces produced had an equivalent to the hidden-line removal problem, which was rather confusingly referred to as either the *hidden-surface* or the *visible-surface* problem. The nature of raster displays, however, led to a much broader array of possible algorithms than for the hidden-line problem.

1.2.3 The Visible Surface Problem

We can state the visible-surface problem as the task of finding, given some viewpoint and a description of a scene, which surfaces in a scene are visible to the eye. This was and still is a rich area of research. Algorithms to solve the problem can be separated into two groups; *image-space* algorithms, which produce a frame-buffer of the results, and *object-space* algorithms, which produce a list of the surface primitives visible [Suth74].

The simplest visible-surface algorithms are known as painter's algorithms, and rely on the observation that if the polygons are sorted correctly, drawing them in order from the fartherest to the nearest will result in the correct picture. The classic painter's algorithm was developed in 1972 by Newell, Newell and Sancha [Newe72]. They first sorted the the polygons by their z coordinates, and then resolved any ambiguities that were left over due to polygons overlapping in z. They also observed that it was possible for the above observation to fall down in certain situations where two polygons closely overlapped each other. In such cases the problem was resolved by splitting one of the polygons in half so that a correct ordering was possible.

Earlier, Schumaker had developed a method for determining a drawing order for polygons by the use of partitioning planes [Schu69]. Ten years later this work was extended to produce the binary-space partitioning (BSP) tree, a data structure that could guarantee a perfect depth ordering [Fuch80]. A particularly impressive feature of the BSP tree is that the tree only has to be built once; it can then be used to generate viewing orders for any given viewpoint.

In 1974[1] Catmull developed the image-space depth-buffer algorithm as part of his doctoral work on displaying curved surfaces[2]. This proved has to be easily the most popular visible-surface method that has been developed, mainly because of its simplicity, and its potential for hardware implementation[3]. The algorithm requires the creation of a second frame-buffer for the picture, one which stores a depth for the pixel rather than an intensity value. Whenever we come to draw a new polygon over such a picture, the depth of each pixel it covers is checked against the depth of the polygon at that pixel, and is replaced only if the polygon's depth is smaller. As a consequence of this, polygons can be rendered to the screen in any order, and the correct image will still be produced.

In the late sixties/early seventies a number of object-space methods were developed by, among others, Warnock [Warn69], Appel [Appe67], Galimberti [Gali69], and Loutrel [Lout70]. While we do not have space to go into all of these algorithms, Warnock's was probably the most influential in terms of the elegance of his approach. He posed the problem recursively, so that for any rectangular area of the image the following algorithm was applied:

---

[1]The depth-buffer method was preceeded by several algorithms similar in spirit developed over the years 1967-70, by Romney, Bouknight, and Watkins; see [Sund74] for details.

[2]In this PhD Catmull produced a remarkable number of new concepts. As well as the depth-buffer, he developed a method for displaying bi-cubic patches, by subdivididing such patches until they only covered a single pixel. He also developed techniques for mapping two dimensional pictures onto such surfaces, establishing the foundations of texture mapping.

[3]Such hardware implementations are available on most high-end graphics work-stations these days.

1. If the area is simple enough, draw it. The area is defined as being simple if it contains only one polygon, or if there is one polygon that completely covers the area, and is in front of all the other polygons that intersect that area. In both cases we only have to draw that polygon to achieve the correct result.
2. If the area isn't simple, subdivide it into four smaller rectangles, and go back to 1 for each of them.

This divide-and-conquer approach has since been applied to many other problems in Computer Graphics.

Finally, the definitive paper on the visible-surface problem, in which ten such algorithms were surveyed, was published in 1974 by Schumaker, Sutherland and Sproull [Suth74]. The authors observed that all the algorithms rely on sorting in x and y and z; they simply differ in the order in which they sort the scene, and in the method of sorting. For instance, the depth-buffer method sorts in x and y, and then in z, using a bucket sort for y and z, and a bubble-sort for x.

1.2.5 The Introduction of Illumination

These early rendering methods all assumed that objects were equally lit; each surface was assigned a particular colour, and the entire surface shaded with that colour. This approach has since been labelled as a "self-luminous surface" shading scheme, in an attempt to put it in context with more complex shading schemes that came later, but certainly there was never any consideration of lighting models at the time. Soon, however, researchers started to attempt to reproduce the effects of illumination with various shading schemes, and this has since become an important area of research.

We can define the illumination problem as that of determining the colour at any point of a surface, taking into account where the viewer is, where any light sources in the scene are, and the reflection properties of the surface. Traditionally, this area has seen a new model of the way surfaces reflect light developed, paired with a new rendering technique for effectively displaying that model.

Before we look at developments in this area, we should look at some of the basics of illumination. A glance at any photograph will reveal that illuminated surfaces tend to have a number of common features:

- Shadow boundaries. For distant or very small light sources these are sharply delineated. For large area light sources they tend to be smeared out.
- Diffuse reflection. Many dull surfaces scatter light evenly back into space, rather than in a particular direction. This type of surface is referred to as diffuse, or Lambertian, and looks equally bright from any viewing angle. Diffuse reflection tends to change slowly across a surface.
- Specular reflection. This occurs with shiny, mirror-like surfaces. It is characterised by specular highlights — small spots of bright light where light sources are reflected off the surface

towards the eye. Specular reflection is characterised by fast-varying changes in intensity across a surface.

- Caustics. These aren't common, but we mention them here because they turn out to be important in more recent work in the subject, for reasons that will be explained later. A caustic occurs when light is scattered off a reflective surface and onto a diffuse one, resulting in a (usually blurry) image of the light source on that surface. A common example is the reflection of bright sunlight off a bathroom mirror, resulting in a dappled patch of light on a wall or the floor.

1.2.6 Flat Shading

Early on in the development of solid-shading renderers, a number of methods were employed to give some kind of variation in shading to scenes; without this, any object tended to look like an amorphous blob. One obvious way to do this was simply pick a different colour for each adjacent polygon in an object, but another method that proved popular was depth-cueing [Warn69], which gave more realistic pictures. In this method, a polygon's brightness is scaled inversely by its distance from the viewer, so that objects get darker the further away they are. The result is scenes that look as though they have been lit from the front. (See figure 1.4B, for example.)

The next step was the introduction of light sources and diffuse-reflection model. If the light source of a scene was an infinite distance away, the light reflected by any polygon in the scene was assumed to be proportional to the cosine of the angle between its normal, and the direction in which the light source lay. This relation, known as Lambert's law, ensures that the polygon is at its brightest when it is directly facing the light source, and becomes dimmer as it is turned away from it. To stop the polygon from becoming completely dark when it faces away from the light source, a fixed amount of *ambient* light is usually added to the intensity of each polygon.

Bouknight [Bouk70], was one of the first to use such a model. He simulated curved objects such as spheres by representing them with a *polygonal mesh*[1], and then *flat-shaded* each polygon with the colour calculated for it as above. The results he obtained were similar to figure 1.4C.

1.2.7 Gouraud shading

One thing that became obvious from Bouknight's work was that flat-shading polygons didn't give as good results as one might have thought — the discontinuity in shading between adjacent polygons is quite noticeable, even when the polygons are small. The reason behind this is a phenomenon known

[1]In this history we have ignored the development of more complex surface representations, as this thesis is more concerned with rendering and illumination issues. Parallel to the work described, however, research was being done on surfaces such as bicubic patches (general curved surfaces) and quadrics (ellipsoids, cones and hyperbloids), and how to render these surfaces. The most common rendering technique is to produce a polygonal-mesh representation for the surface, partly because the manipulation and rendering of polygons is so well understood, and partly because of current workstation hardware, which tends to be oriented towards drawing large numbers of polygons.

as Mach banding, first noted by Edward Mach in the 1860’s. It turns out that the brain actually preprocesses images it sees in order to make it easier pick out the edges in such an image.

Figure 1.3 provides an example of this. There are a number of vertical bands, each one of which is a fixed intensity step up from the previous one. The eye sees these bands as being “scalloped” because of this effect; on the left-hand side of a band the intensity appears lighter than it is, and on the right-hand side, darker.

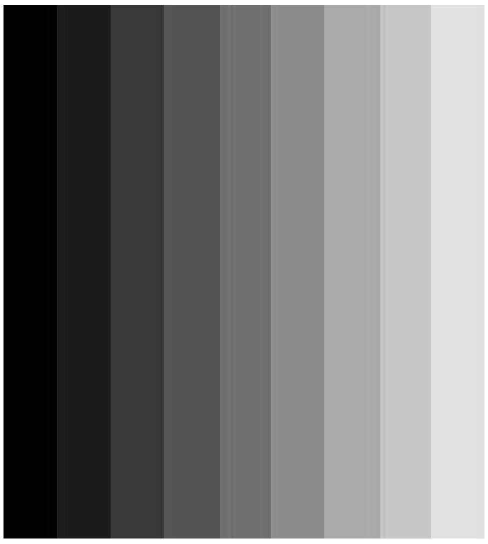


Figure 1.3 Mach Banding

Gouraud overcame this phenomenon by evaluating the lighting model at the vertices of a polygonal mesh, and then interpolating the resulting colours across each polygon. This gave colour shading across the polygons that was continuous (to the first order, anyway), resulting in more realistic views of objects, as in figure 1.4D.

The greatest advantage of Gouraud shading was in the rendering of polygonal approximations of curved surfaces, as in the sphere and cylinder of figure 1.4. The surface normals to these points, which is what the lighting model relies on, could be replaced with pseudo-normals taken from the original curved surface. As adjacent points would thus have the same pseudo-normal, the resulting image gave the impression of a smooth surface, as in figure 1.4E. Although they looked smooth, the true nature of these objects could be recognised by looking at their silhouettes, which were still polygonal.

1.2.8 Phong shading

At this stage (around 1975), the emphasis was beginning to shift away from the visible surface problem to modelling the way light interacted with surfaces. This led to further work on reflection functions, in an attempt to add specular reflection. Bui-Tong Phong [Phon75] formulated an empirical model which produced realistic-looking highlights, by using the cosine function raised to a power to provide a suitable “bump” in the reflection model of the surface. The argument of the cosine function was the difference between the viewing angle of the surface and the angle of reflection of the light source involved. This guaranteed that specular highlights appeared in the direction of reflection.

Unfortunately, Phong found that Gouraud shading tended to smear these highlights over the polygon they occurred on. He solved this by developing the Phong shading method, which, rather than interpolating vertex colours across a polygon, interpolated the surface normal. The full illumination model was then applied at each pixel, giving much more accurate, albeit expensive, results.

A Phong-shaded scene can be seen in figure 1.4F. Of particular interest is the sharpness of the two highlights on the sphere, compared to the smeared versions in the Gouraud-shaded scene of figure 1.4E.

1.2.9 Shadows

Parallel to the development of the above rendering techniques, a number of algorithms were developed to generate shadows for polygon-based scenes with point light sources. All of these algorithms work by using one of two methods. In the first, the scene is viewed from the light source, and a visible-surface algorithm used to calculate those parts of the scene that the light source can "see". These parts are then tagged as lit, and when the time comes to render them, this is taken into account.

Two such algorithms appeared around 1978. One was produced by Williams [Will78], and used a pair of depth-buffers. By the application of an appropriate mapping between the viewer's and the light source's depth buffer, one could easily find whether a pixel was in shadow. The second was published by Atherton, Weiler and Greenberg, and utilised an object-space method for hidden-surface removal [Athe78].

The second method of shadow generation projects each polygon onto the rest of the scene, treating the light source as the origin. This is usually done by creating shadow polygons, extending from each edge of a normal polygon away from the light source. The intersections between these shadow polygons and other polygons in the scene determine the boundaries between light and dark areas. Bouknight produced one of the earliest shadow-generation algorithms in 1970 by incorporating such an approach into a scan-line algorithm [Bouk70b]. Later, Crow produced an object-space version of the method using BSP trees [Crow77].

An algorithm that was published even before Bouknight's was that of Arthur Appel [Appe68]. While at the time it appeared to be just another possible shadowing algorithm, and had the drawback of being much more expensive than most others in terms of computation time, it eventually formed the foundation of the most popular and powerful method for high-quality image rendering to be produced over the last fifteen years. We shall meet it again later in this chapter.

1.2.10 Further Developments in Illumination Models

While some degree of realism was achieved by the Phong model, there was no variation in detail over the surface; all objects appeared to be made from featureless plastic. An early way of adding some variety to scenes was the use of surface-detail polygons. These were added to parent polygons in order to represent things like windows or a door on a house. They could be ignored during the visible-surface determination process, because they lay within the parent polygon. When this parent was rendered, they were in turn drawn over the top of it.

In 1974 Edwin Catmull produced several images of curved surfaces which had had two-dimensional pictures wrapped onto them [Catm75]. He achieved this by using a normal raster-image picture as a *texture-map*. The parametric coordinates of each pixel on the surface were used to index into the contents of the map to find their appropriate colour. Jim Blinn later extended this work [Blin76], and also developed a similar technique for perturbing surface normals in order to make surfaces appear bumpy. Later, in the eighties, the concept of “solid” texture-mapping was developed by Ken Perlin [Perl85] in order produce surfaces that looked as though they had been carved from some material, such as wood or marble.

A number of researchers were not content with the plastic-looking effects provided by the Phong model itself. This led to a fair amount of tweaking of the model in order to get results that “looked” more realistic. It also led to interest in physically-based reflection models, i.e. ones based on studies of how real surfaces reflect light. In 1967, two physicists, Torrance and Sparrow, had published a paper on modelling the reflective properties of surfaces [Torr67]. This identified several non-obvious reflection effects. For instance, at low viewing angles a surface’s reflectivity was predicted to increase, and the dominant direction of reflection to be at an angle slightly below that normally expected. The later effect is often referred to as the “off-specular peak,” and arises from considering the Fresnel equation for light reflected by a dielectric interface.

The reflectivity effect predicted by the model is due to geometrical effects, and can often be observed on wax floors or other shiny surfaces. When viewed at a grazing angle the floor is almost mirror-like, whereas from above it appears normal. Torrance and Sparrow’s work was brought into the Computer Graphics field by Blinn [Blin76], and later Cook, who coauthored a paper with Torrance [Cook82], in which a more sophisticated refelction model was put forward[1].

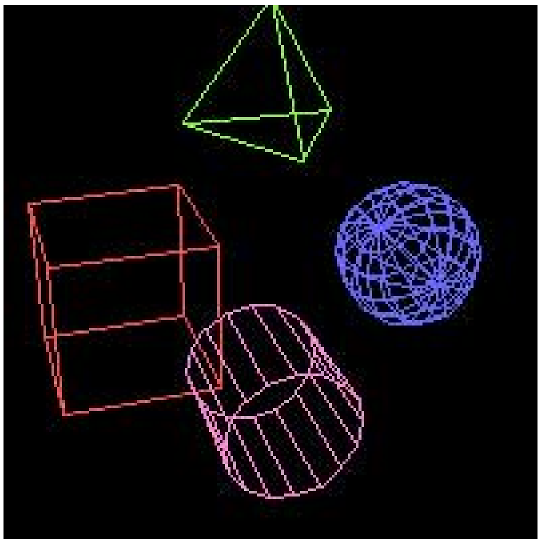
1.2.11 Summary

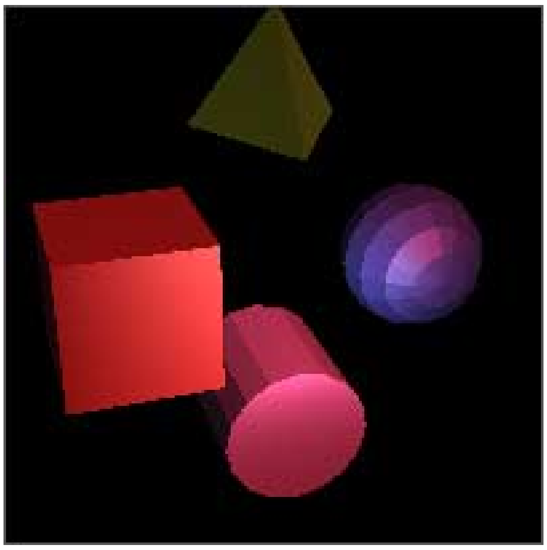
It is from this basis that today’s high-quality image rendering techniques have evolved. Up until the late 1970s, rendering focused on solving shape-rendering problems, and then applying the correct shading afterwards. Even the most complicated shading methods only tried to model the light that was reflected from point light sources; inter-surface reflection was completely ignored.

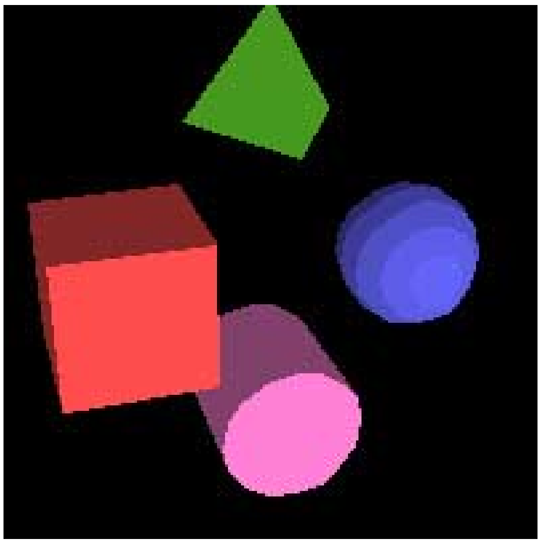
[1]In particular, this model distinguishes between the characteristics of non-metallic and metallic surfaces.

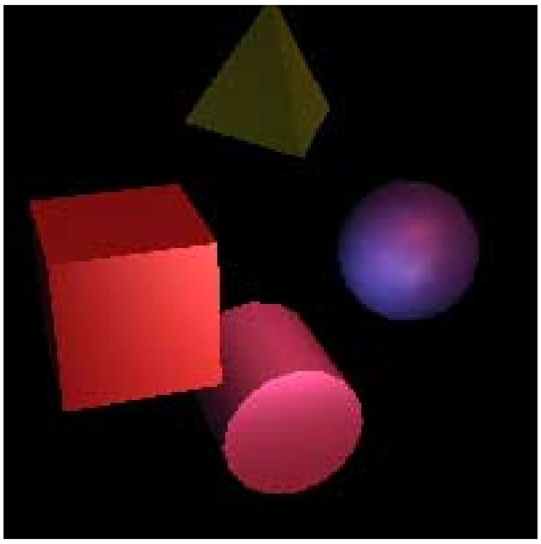
As work on illumination models has progressed, however, attention has turned to *global-illumination* algorithms. These algorithms attempt to completely model the way light moves around a scene, by including the contribution made by indirect illumination, whereby light may bounce off many surfaces in the scene before it reaches the eye.

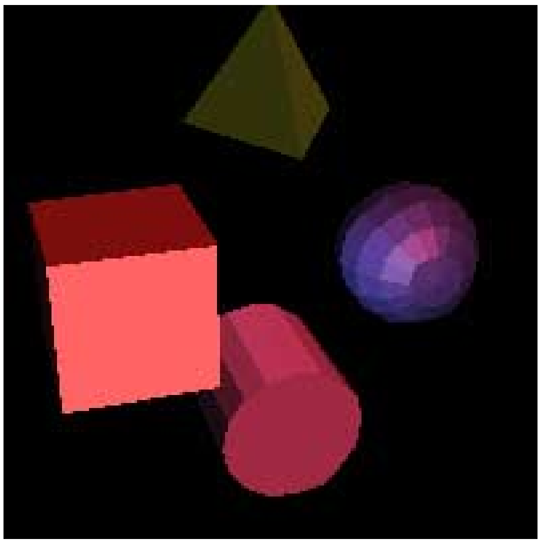
Two such algorithms have been developed, both of which employ their own approximations, and have corresponding strengths and weaknesses. From Appel's shadow algorithm came *ray-tracing*. From heat-mechanics work done in physics in the 1960's came *radiosity*. In the next two sections we take a look at these important techniques.


A Wire-frame rendering.


D Gouraud shaded.


B Solid, depth-shaded.


E Gouraud with curved surfaces.


C Solid with illumination model.

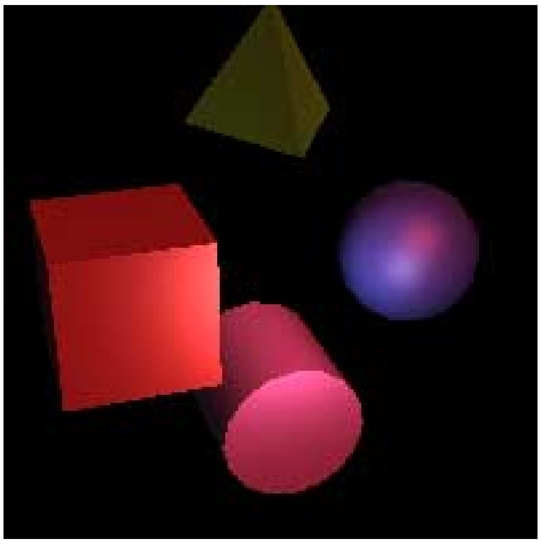

F Phong shaded.

Figure 1.4 Methods for rendering scenes

1.3 Ray Tracing

Much of the success of ray-tracing can be attributed to its simplicity. Conceptually, the algorithm involves shooting out rays of light in all directions from the eye, in order to discover what it sees. These rays bounce through the environment, eventually finding their way out into space or towards a light source. Of course, light actually travels the other way round, but the idea is still valid. Ray-tracing is the most integrated of all rendering techniques in Computer Graphics; the one algorithm handles shading, shadows, surface-rendering and visible-surface determination.

1.3.1 The Development of Ray Tracing

In 1968 Appel, who was working on the visible-surface problem, introduced a new method for determining whether surfaces were in shadow. This involved casting a “ray” from the surface point under consideration towards the light source in the scene. This line through space could be tested algebraically for intersection with any scene object. If such an intersection occured between the light source and the surface, the surface was held to be in shadow.

Goldstein and Nagel [Gold71] extended the use of this ray-casting to incorporate the visible-surface determination problem. For every pixel on the screen, a *view* ray was cast that passed through the eyepoint and the pixel. This ray was then tested against every object in the scene for any possible intersection. The intersection that was closest to the screen was taken as the surface that was seen through that pixel, and Appel’s method was then applied to determine how to colour it.

Ten years later, Whitted [Whit80] generalised Goldstein and Nagel’s method to account for specular reflection and refraction, thereby producing the *recursive ray-tracing* algorithm that is common today. When a view ray intersects with a surface, shadow rays are cast towards each light source, and reflection and refraction rays are cast in the appropriate directions, as in figure 1.5.

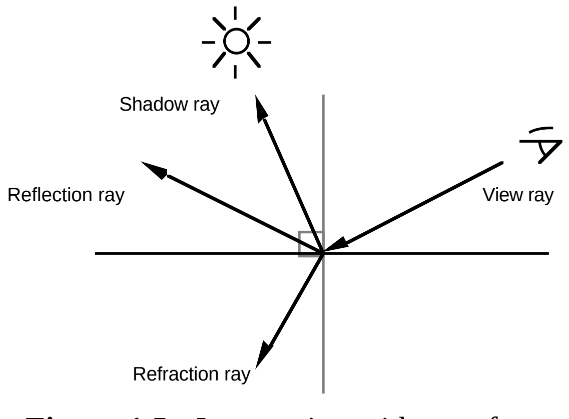


Figure 1.5 Intersection with a surface

The shadow rays are used to calculate the contribution of the light sources to the colour “seen” by the view ray, and the other rays the to calculate specular reflection or refraction of light from other surfaces. These latter two rays are recursive, in that they are treated like new view rays, capable of in turn spawning more reflection and refraction rays. In this way multiple reflections and refractions can be handled. An example is shown in figure 1.6.

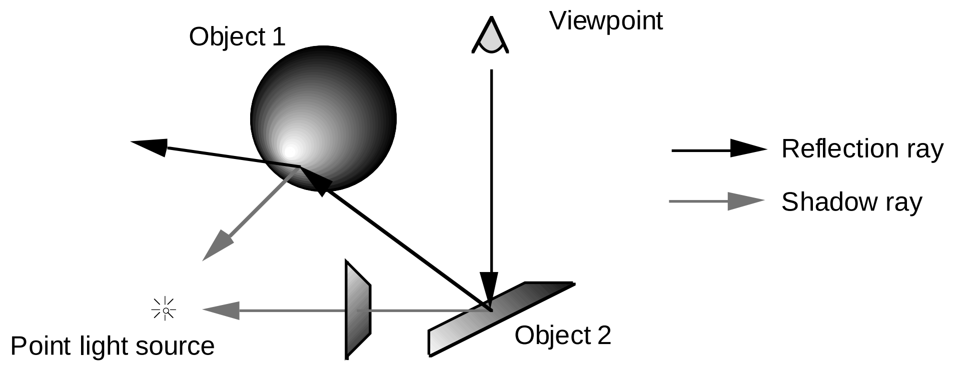


Figure 1.6 An example ray-cast

All of this is horrifically expensive in terms of computation time. Casting a single ray involves checking every object in the scene for possible intersections, and at least one ray has to be cast for every pixel on the screen, resulting in the order of one hundred thousand to a million rays for typical scenes. However, the great advantage of this technique is its algebraic nature. This means that quite complex surfaces can be tested for intersection, including spheres, cones, cylinders, and many higher-order surfaces such as Bezier patches. In fact, a sphere turns out to be one of the easiest surfaces to test for intersection, hence its domination of early ray-traced scenes. Ironically, one of the hardest shapes to test for intersection is the polygon.

An example of a ray-traced figure can be seen in figure 1.7. The most notable features are the complex shadows cast by the metallic fractal sphere, and the multiple reflections on its surface.

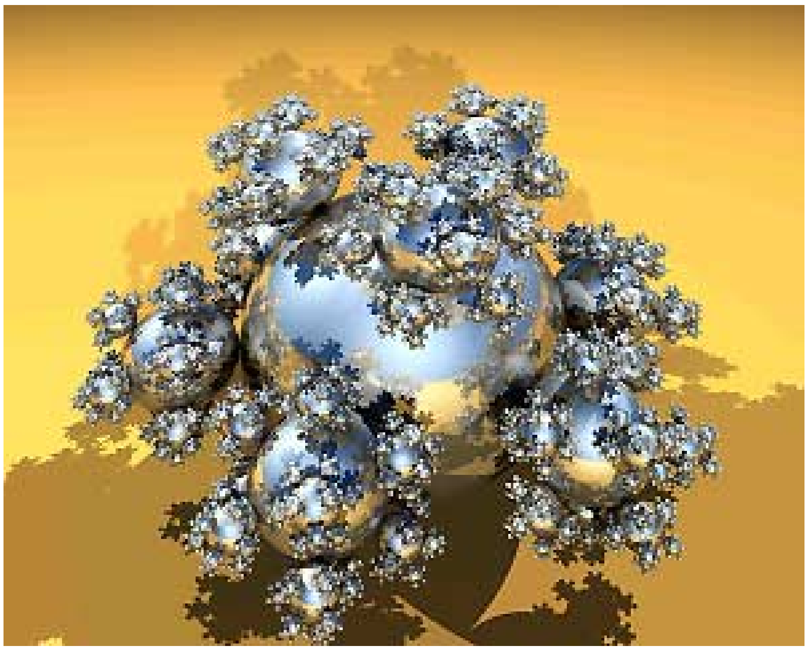


Figure 1.7 A ray-traced picture

In summary, the features of ray-tracing are:

- Ability to show reflections. Mirror-like surfaces are handled particularly well.
- Transparent objects are modelled due to the refraction rays. Rendering glass spheres was particularly popular in early ray-traced images.
- High-order surfaces can be handled easily; indeed if an intersection test can be provided for it, any geometric surface can be rendered.
- Shadows are provided as an integral part of the algorithm.

1.3.2 Discussion of Ray Tracing

Ray tracing is known as a *view-dependent* rendering method. Because of the way light rays are cast out from the eyepoint, and then further only towards areas of interest, illumination calculations aren't performed for those surfaces of the scene that are never seen. However, this does mean that whenever the viewing point changes in relation to the scene, all of the illumination calculations must be repeated.

Because of the recursive nature of ray-tracing, every reflection/refraction ray cast can in turn spawn other rays. To avoid this process continuing on unnecessarily, a bound on the depth of the*ray tree* is usually introduced, in order to limit the number of consecutive "bounces" of light. Hall also introduced an adaptive limit, which kept track of each ray's contribution to the original pixel, and didn't cast the ray if this contribution fell below some lower bound [Hall83].

Whitted's original research indicated that 75-95% of the algorithm's time was taken up with intersection tests. Since that time, speeding up these tests has become a large area of research. All of the methods proposed for this task work by exploiting "coherence" in the scene, namely the continuity in things such as object surfaces. As an example of this, the probability of two adjacent rays hitting the same surface is fairly high. The most successful schemes for exploiting this coherence work by building data-structures that roughly sort the positions of the objects in the scene. For any given ray they use these structures to eliminate all the objects that it definitely won't hit, leaving only a small number of "possibly-hit" objects to be tested. Glassner [Glas84] uses an octree for this purpose, while Fujimara [Fuji85] uses a simpler grid-like structure which, while not representing empty space as efficiently, is easier to traverse.

1.3.3 The Drawbacks of Ray-tracing

One of ray-tracing's biggest drawbacks is that, because it involves casting infinitely-thin rays through a grid, it can misrepresent the finer details of a scene. Obviously, very small objects can be completely missed by the rays, but another problem is that regular patterns of objects can suffer from what is called *aliasing*. Because the rays are themselves cast in a regular pattern, the two patterns can interfere with each other, producing a somewhat misleading effect on screen. A common example of this occurs in movies or on television, when the spoked wheels of some vehicle appear to be revolving the wrong way. This occurs because the rate at which the wheel is turning is of the order of the frame rate of the movie. Indeed, if these rates matched exactly, the wheel would appear stationary!

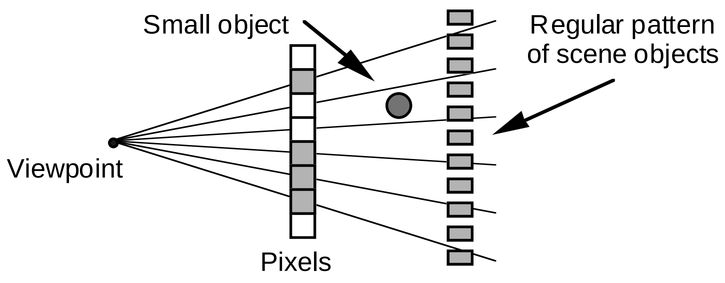


Figure 1.8 Aliasing problems

The obvious solution to these problems is to cast more rays for each pixel, in order to get a finer coverage of the scene, and then average their results to find a single intensity value for the pixel. This is known as *supersampling.* The mathematical Fourier-series can be used to build a theoretical model which explains the effects of sampling a continous function (such as a scene). It can be used to show that straight averaging of the super-sampled rays does not give optimal results, but because of its simplicity this approach is still popular.

In terms of illumination, the major drawback of the ray-tracing algorithm is its assumption that only specular transfer is important. This becomes apparent when rendering scenes where low-light conditions apply, or large areas of shadow exist. In such situations the viewed colour of a surface depends on the diffuse reflection of light from other surfaces, rather than specular reflection, or direct illumination from light sources.

These problems can also manifest themselves in brightly lit scenes, however. Ray-tracing does not account for any caustics in the scene, because even if very bright light is reflected on to a surface by a mirror, it will not in turn undergo diffuse reflection towards the eye. Only light directly from light sources is treated in this manner.

More subtle problems also exist, especially with refractive objects. For instance, a glass sphere is capable of acting like a lens, and focusing a spot of light onto a surface. This is missed by a standard ray-tracer because it casts its shadow ray straight towards the light source. As the light reaches the surface by being refracted through the sphere, and therefore follows a curved path, the ray-tracer is unable to trace it.

1.4 Radiosity

Radiosity is somewhat of an anomaly in the graphics field in that it has its roots not in the work mentioned in section 1.2, but in work done in Physics in the 1960s on radiative heat transfer[1]. It arose out of an attempt to treat the illumination problem strictly through theory, rather than by using empirical methods[2] to acheive scenes that "look" realistic.

1.4.1 The Development of Radiosity

In 1984 Goral et al. [Gora84] published the radiosity algorithm[3] in 1984. The main concept is that the surfaces of all the objects in the scene are evenly divided into a polygonal mesh of small polygons, called *patches*. The radiosity method then simulates the flow of light between each of these patches, distributing the energy from those that are emitting light throughout the rest of the scene. This is done in a *view-independent* step; the calculation of patch intensities is done once, and the resulting scene can then viewed from any position without any recalculation.

Each patch is assumed to reflect light in a diffuse manner, and thus all reflection is non-directional. No matter where you view a patch from, it has the same apparent intensity. This means that the light emitted by any patch can be completely captured by its *radiosity* (B), which is the energy per unit time per unit area leaving the patch.

The fraction of the energy emitted by one patch (j) that is intercepted by another (i) is called the *form factor* (F_{ji}) between those two patches. The fraction of the incoming light that is reflected by a patch is given by its reflectivity (ρ_i); it may also be emitting its own radiosity (E_i). We can therefore write a patch's total radiosity as

$$\begin{aligned} B_i &= \textit{Emitted Radiosity} + \textit{Reflected Radiosity}, \\ &= E_i + \frac{1}{A_i}\sum \textit{Light reflected from other patches}, \\ &= E_i + \frac{\rho_i}{A_i}\sum_{j=1}^{n} F_{ji}\left(B_j A_j\right). \end{aligned} \tag{1.1}$$

This relationship can be written in vector form for all the patches in the scene, so that

$$\tilde{B} = \tilde{E} + \mathbf{M}\tilde{B}, \tag{1.2}$$

and rearranging this gives us the system of linear equations,

$$(\mathbf{I} - \mathbf{M})\tilde{B} = \tilde{E}, \tag{1.3}$$

[1]Most radiosity papers reference [Sieg81] as a good summary of the field.

[2]Although ray-tracing did not evolve this way, it has since been given a theoretical basis by Kajiya [Kaji86].

[3]Much the same technique was developed independently by Nishita and Nakamae [Nish85].

which can be solved by any of several well-known methods, such as Gaussian elimination, or more commonly, Gauss-Siedel iteration. This gives us the radiosities of all the patches, which we then use to render the scene.

An example of a radiosity-rendered scene can be seen in figure 1.9. Because diffuse lighting changes very slowly over a surface, the coarse mesh of figure 1.9 can produce very realistic results when it is Gouraud-shaded, as in figure 1.10

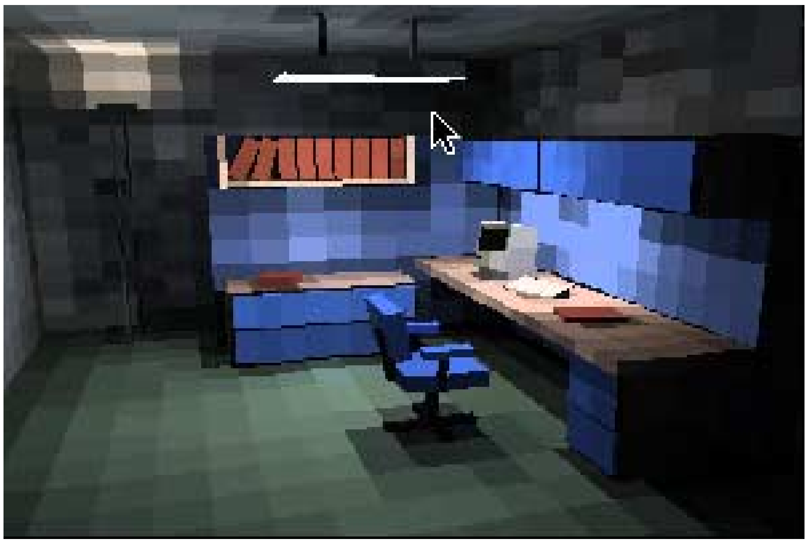


Figure 1.9 Radiosity Rendering with Flat Shading

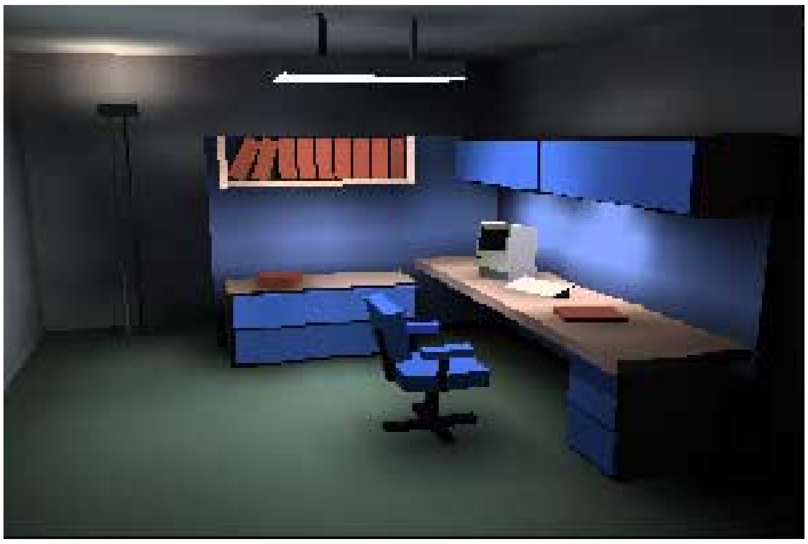


Figure 1.10 Radiosity Rendering with Gouraud Shading

1.4.2 Discussion of Radiosity

The calculation of the form-factors between patches has proved an interesting research problem. The original paper used numerical evaluation of contour integrals to calculate the form factors for a scene, a rather inflexible approach that didn't take into account the possibility of obscuring surfaces. In 1985 Cohen and Greenberg [Cohe85] adapted the depth-buffer algorithm to produce the *hemicube* algorithm, which calculates approximate form factors efficiently, taking into account hidden surfaces.

The hemicube algorithm works by surrounding a patch with the upper half of a cube, as in figure 1.11. Each side of the hemicube is taken to be an imaginary screen, with the view-point at the centre of the patch. The depth buffer algorithm can be used to establish which patches cover which pixels on these “screens”, delta form-factors for each of the pixels having been pre-calculated by analytical means. The form factor for a particular patch is then obtained by summing the delta factors of every pixel it covers.

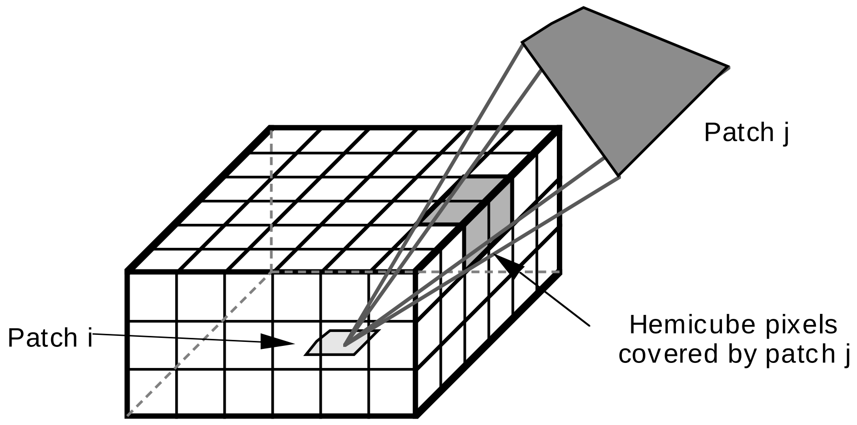


Figure 1.11 The hemicube in operation

The hemicube method has proved very popular because it can take advantage of the high-speed hardware depth buffers that many graphics workstations have. However, Baum et al. [Baum89] have pointed out that it is prone to error, especially when calculating the form-factors of adjacent surfaces. They propose a way of rectifying this by checking for cases where such errors occur, and supplementing the hemicube results with an analytically-calculated form factors for these cases.

One of the major disadvantages of the radiosity algorithm is that often a large amount of space is required to store the matrix of equation 1.3. For instance, a scene with only one thousand patches will have a corresponding matrix of one million elements. A secondary disadvantage is that it is only once the set of linear equations associated with this matrix has been solved that the scene can be displayed.

These disadvantages were addressed in 1988, when Cohen, Chen, Wallace and Greenberg published a paper on the idea of *progressive refinement* [Cohe88]. They developed an iterative method for solving the radiosity equation which only needed access to one hemicube’s worth of form-factors at each iteration, thus greatly reducing the storage cost of the algorithm. Moreover, on every such iteration all of the radiosity estimates were updated, in contrast to the matrix solution, which only found the radiosity of a single patch at a time. This meant that the current solution for the scene could be redisplayed after each iteration, progressively getting better and better as time went on. This is generally known as the *shooting* method of solution, as it involves picking a patch in the scene and “shooting” its current energy throughout the rest of the environment. In contrast, the original method is called *gathering*, as in each iteration one patch “gathers in” the light from the environment in order to work out its own radiosity.

A common problem with radiosity is that in some places, such as shadow boundaries, the polygonal mesh needs to be reasonably fine to capture amount of detail involved, whereas at others it can be quite coarse. Cohen, Greenberg, Immel and Brock [Cohe86] solved this problem by checking the gradient of the radiosity over each patch; patches with too high a gradient were subdivided into a number of smaller ones. Thus in areas where the radiosity changed rapidly, the mesh became finer in order to capture it better.

Because the mesh has such an effect on the final result, there is currently a great deal of emphasis on automatic mesh-generating techniques, such as [Baum91] and [Lisc92]. The latter of these calculates where discontinuities in illumination will fall (such as shadow boundaries) and ensures that the borders of the mesh lie along those discontinuities, so that they don't become "smeared". Prior to these algorithms, meshes had to be hand-generated by the user.

Just recently, some efforts have been made to produce view-dependent radiosity methods so that when rendering very large scenes, only the parts that are currently seen by the user have their radiosities calculated [Smit92].

1.4.3 Drawbacks of Radiosity

Radiosity handles diffuse environments such as dimly-lit indoor scenes very well, and many of its early problems have been solved. However, it has a number of remaining drawbacks:

- It assumes diffuse reflection. This rules out specular highlights, and reflective surfaces. Many surface-detail techniques, such as bump-mapping and texture mapping, aren't possible.
- The effectiveness of the algorithm is heavily dependent on the way surfaces are split into patches. In particular, if a sudden change in illumination (such as a shadow edge) falls across a patch, it gets smeared out over that patch. This is another reason why sharply-lit scenes are not handled well by most radiosity renderers, although the work of [Lisc92] mentioned above appears to have gone some way towards solving the problem.
- Aliasing problems are still apparent, because a point-sampling method (the hemicube) is used to find form factors. This can manifest itself in small patches that appear unlit; because they are smaller than pixel size on the light source's hemicube, their effective form factor is zero, and they receive none of its light.

We now take a look at some of the work that has been done on extending radiosity and ray-tracing in order to overcome their drawbacks.

1.5 Extended Ray Tracing

Since the development of ray-tracing, several schemes have been put forward for extending ray-tracing in order to address problems such as aliasing and lack or diffuse reflection. Some of these are presented in this section.

1.5.1 Backward Ray Tracing

The indirect illumination that a standard ray-tracer misses could be accounted for by running a ray-tracer backwards, i.e., tracing rays from the light sources to the eye. This approach soon runs into problems, however, because an enormous number of rays would need to be cast from a light source in order to produce even one or two that eventually hit the eye.

Arvo has overcome this to some extent by using rays cast from the light sources to deposit “energy” on diffusely-reflecting surfaces [Arvo86]. A normal ray-tracing pass is then used to render the scene. Whenever a ray intersects with such a surface, it adds its energy to the usual direct-light and specular-reflection components of illumination.

1.5.2 Distributed Ray Tracing

In 1984 Cook et al. [Cook84] introduced stochastic sampling to ray-tracing, resulting in an elegant algorithm capable of modelling most illumination effects, including diffuse reflection. Instead of casting a single ray to determine the light reflected from a surface, it samples the surface’s reflectance function by distributing a number of rays over it, as in figure 1.12. This idea of using rays to sample a function or area is also used to model other effects, such as area light sources, motion blur and blurred reflection.

The major drawback of the algorithm is the huge number of rays it requires. When a surface’s reflection function is highly specular, the number of rays needed to properly sample the function is not great, but when a diffuse reflectance function must be evaluted, the number of rays cast is often large. Also, the algorithm still uses infinitely-thin rays, and thus can suffer from aliasing effects.

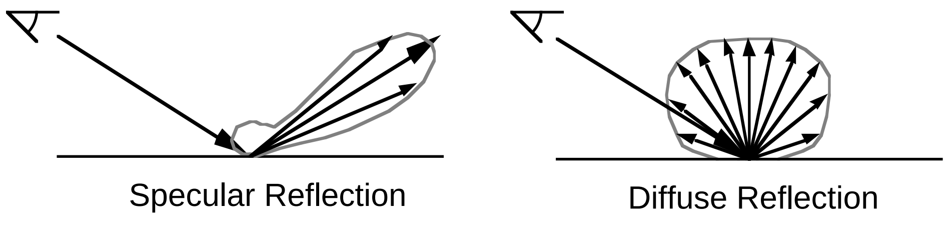


Figure 1.12 Stochastic Sampling

1.5.3 Finite-Width Ray Tracing

Amanitides [Aman84] developed the idea of cone tracing, which replaces the usual infinitely-thin ray with a cone. This cone is assigned an angular spread wide enough to encompass the pixel it is fired through. Reflection and refraction of a cone-ray involves calculating the new angular spread of the cone as well as its new direction. Blurry specular surfaces can be simulated by broadening these angles; in particular quite realistic pictures of unpolished metal surfaces can be produced.

Because any intersection of a cone with an object involves an area, the *fractional blockage* of the cone by that object is generated, rather than a single hit/didn't hit decision. Thus a single ray may hit several objects. Also, area light sources are modelled by using spheres; the shadow rays cast towards a light source are generated so that their base encloses its cross-section.

Similar schemes which give rays a notion of "thickness" have been developed by Heckbert and Hanrahan (*beam tracing*) [Heck84], and Shinya, Takahashi and Naito (*pencil tracing*) [Shin87].

1.5.4 Other Schemes

Ward introduced a scheme whereby the diffuse component of reflection was calculated at a slower rate than for specular reflection [Ward88]. This scheme takes advantage of the relatively slow-changing nature of the diffuse component of reflection, by reusing diffuse samples for many nearby specular samples. Wherever the diffuse component is needed, the algorithm checks for any samples that have been made nearby. If the current point is within a 'sphere of influence' of one or more of these points, a weighted average of them is taken. If not, a new sample is taken. The scheme has been implemented in the *Radiance* ray-tracer, which has proved particularly popular in architectural circles.

1.6 Specular Radiosity

At the SIGGraph '86 conference Immel [Imme86] presented a scheme which extended radiosity to make it capable of specular reflection. Instead of storing a single radiosity value for each patch, the radiosity in a particular direction is stored for a number of such directions. Thus for each patch we have the directional distribution of radiated light stored, and we can represent specular reflection.

Immel uses the hemicube as the basis of this technique. Each pixel in the hemicube represents a different direction, and all hemicubes are oriented with respect to a global coordinate system, to allow easy matching of the emitting and incident directions of different patches. The resulting hypermatrix of directional radiosities is solved by taking advantage of its sparseness.

One of the major visual drawbacks of the scheme is that its patch-based nature severly restricts the sharpness of any reflection effects. Because of this, reflective surfaces rendered with specular radiosity often appear dappled, like an unpolished floor.

The standard radiosity solution involves solving an n x n system of linear equations, where n is the number of patches in the scene. Unfortunately, if D is the number of directional radiosities stored for each patch in the specular radiosity method, this becomes a system of nD x nD equations! Although the method is a good example of how a complete solution to the illumination problem might operate, it has been largely unused since its proposal because of its storage and time costs. However, with the rapid pace of computer development in recent years, this could change[1].

1.7 Ray Tracing and Radiosity Mixes

Wallace, Cohen and Greenberg were the first to attempt to combine the ray-tracing and radiosity techniques in order to produce a hybrid scheme [Wall87]. This certainly seemed an attractive approach. Most early illumination models added together the specular and diffuse components of reflection to get a composite reflection model, so why not combine a specular and a diffuse illumination method, in order to get a new method capable of handling composite reflection?

Unfortunately, the interdependence of illumination means that the models cannot be combined by simply summing their results. To overcome this, Wallace et al. formulated four mechanisms of transfer, those being the different combinations of diffuse and specular reflection. The specular-to-diffuse and diffuse-to-diffuse transfer was handled by a modified radiosity algorithm, which used the work of Rushmeier [Rush86] to allow the radiosity pass to take account of mirror-like reflection. This radiosity pass was then followed by a ray-tracing pass, which incorporated its results in order to model diffuse-to-specular and specular-to-specular transfer, and produce the final picture.

In the last few years a couple of methods have extended this approach, using specular-radiosity methods for the first pass, and then applying ray-tracing in order to overcome the coarseness of detail caused by the first pass's reliance on patches. Shao et al. [Shao88] use a method where a form-factor includes the effects of specular reflection, and is therefore dependent on the light reflected by the source patch, as well as the relative positions of two patches. These form-factors are first approximated, and then successively refined by *procedural iteration*. In contrast, Sillion et al. follow Immel's technique more closely [Sill91]. Significantly, though, they use spherical harmonics (the spherical equivalent of the Fourier series) to represent directional radiosity distributions with a reasonably small number of coefficients.

All these schemes have in common the idea of using some sort of specular radiosity method[2] to roughly calculate the distribution of illumination through the scene, and then compensating for the lack of detail due to the polygonal mesh by overlaying the results with a ray-tracing pass.

[1]It should be noted, too, that Immelman's algorithm is very much a brute-force algorithm; there are many optimisations that could be made. The two specular radiosity hybrid methods mentioned in the next section have implemented some useful techniques for this, such as the use of spherical harmonics to represent directional radiosities, often by storing only a small number of coefficients.

[2]Rushmeier's technique only implements a limited form of specular radiosity, in that it is possible to have "mirror" patches. The other two algorithms use a more general form based on Immel's work.

While these “mixed” algorithms may seem the natural progression from ray-tracing and radiosity, in that they attempt to combine two techniques with opposing assumptions, it is our opinion that those techniques have not proved to mesh particularly well. Certainly, the growing complexity of the mixed algorithms, compared to the original methods, tends to suggest that it might be better to start from scratch in order to develop a truly general global-illumination algorithm.

CHAPTER 2

An Introduction to Fuzzy Rendering

2.1 Introduction

The main purpose of this thesis has been to continue work on a new global-illumination scheme that is being developed at Auckland University under the guidance of Dr. Richard Lobb. In this chapter we present the motives behind this development, and the fundamental ideas of the scheme. We then examine these ideas in greater detail.

The second part of the chapter looks at the development of the scheme. Previous work done is presented, and several important issues that must be addressed in such a scheme are discussed. Finally, we discuss the goals that were set for this thesis, and lay out the work that had to be done in order to achieve those goals.

2.2 Motivation

The two major global-illumination algorithms solve the global illumination problem by making simplifications to it. The ray-tracing algorithm assumes that the only possible inter-surface reflection is specular, whereas the radiosity algorithm assumes all light transfer is diffuse. Each algorithm has a restricted domain of scenes which it can render realistically.

To some extent this problem has been addressed by distributed ray-tracing and specular radiosity. However, with both these methods the final picture that they produce is heavily dependent on how they are set up. With distributed ray-tracing, the number of rays the renderer casts at each surface intersection affects the quality of the final picture, and with specular radiosity the fineness of the patch grid is important.

Unfortunately, the precise effect of these parameters on the quality of the resulting picture is often obscure. Obviously increasing the total number of rays cast or making the patch grid finer will produce a better picture. However, such changes also result in the renderer having to do a lot more work to produce its picture.

The quality of a picture can be improved without this extra work by distributing the work that the renderer does more carefully. Places in the scene where the illumination of objects varies slowly don't need a lot of work; a small number of samples, or a coarse patch grid, will suffice to capture the light flow. In contrast, places where the illumination does vary sharply, such as shadow boundaries or specular highlights, will need a large number of samples or a fine grid in order to capture the light flow to a similar degree of accuracy.

Adaptive techniques that try to distribute the work done by a renderer so that the accuracy of the picture is reasonably constant are an ongoing area of research. An adaptive algorithm starts off with a coarse solution to a particular problem, and then gradually refines that solution into an acceptable one, at each stage seeking to direct the work done to where it is most needed. The patch substructuring technique mentioned in section 1.4.2 is a case in point. None of these techniques attempt to measure the effect of any work done on the accuracy of their solutions in any formal way. Instead they use heuristics (rules of thumb) to guide them. For instance, in radiosity the difference between the calculated illumination for two adjacent patches is measured; if this exceeds some threshold, then the fineness of the grid at that point is increased. Such heuristics can result in small details being missed if the initial solution is too coarse.

If we take a step back from these problems, we realise that ideally we would like to be able to give a global-illumination algorithm a figure for the accuracy of the picture it is to produce. The algorithm would then start from a coarse solution to the problem, and improve it adaptively until the accuracy of the solution reached the required level. At all times it would keep track of the accuracy of its current solution, and from this information be able to identify the best part of the scene to work on next in order improve this accuracy. Most importantly, it should be able to guarantee that the accuracy of its final solution is equal to or better than the required figure. Our aim has been to develop such a system.

To put this in context, current schemes for high-quality image rendering work by solving exactly an approximation of the global illumination problem. What we have set out to do is develop a system that will solve the exact problem, to some degree of approximation. This means that the renderer must be able to handle arbitrary reflection models, like the distributed ray-tracing and specular radiosity methods.

These and other factors have resulted in a number of goals being formulated which have formed the basis of development work on the rendering system, namely that the renderer should:

- Be able to generate a picture to a specified degree of accuracy.
- Be able to handle surfaces with arbitrary surface reflectance models.

- Be suitable for parallel execution. (Any global illumination method involves a lot of work; parallelisation is one way of reducing the time taken to do this work.)
- Avoid point sampling. As has been pointed out in Chapter 1, this is a drawback common to all current rendering methods.
- Not do any more work than is necessary to produce the final picture. If part of the scene is has no effect on the picture seen by the viewer, there is no point in calculating its illumination. By the same token, if a surface is illuminated by a bright light and a dim light, and the bright light completely masks the effect of the dim light, most of the renderer's effort should be put into calculating the illumination of the surface due to the bright light. A distant object will also require less work than a similar object that lies much closer to the viewer.

To meet these goals an object-oriented, error-bounded rendering method has been developed. We refer to this method as *Fuzzy Rendering*, as a recurring theme of the method is the treatment of quantities such as position or light flow as being uncertain to some degree. The next section outlines the way it operates.

2.3 Fuzzy Rendering

There are three major concepts that underpin the Fuzzy Rendering method. The first is the use of bounded values, or *ranges*, in place of discrete values to represent the metrics used in the illumination process. Any such metric is guaranteed to lie within the range that has been calculated for it. The primary consequence of this is that the renderer can keep track of the error involved in its calculations, and hence the error in its solution. Calculations with ranges involve *range arithmetic.*

The second concept is the use of an object-oriented framework in which the various surfaces in the scene are represented by *patch objects.* Such objects are regarded as intelligent entities, which attempt to negotiate with other patch objects in order to determine the illumination that they receive, and hence emit. The end result is a community of objects each of which works on a part of the illumination problem, passing messages back and forth to the other objects in order to solve that part.

There must be some means of driving such a system in order for it to produce the answer that we want. In a completely view-independent system this is straight-forward; each object is responsible for determining how its associated surface transmits light towards any point in space. For any non-trivial specular scene, however, this requires a large amount of work. For instance, if we consider a scene containing a mirror, we realise that what we see in the mirror is completely dependent on where we are viewing it from[1]. Calculating what the mirror looks like from all possible viewpoints would

[1]Radiosity methods can afford to calculate a view-independent solution to the illumination problem because diffuse surfaces are themselves view-independent. Nevertheless, a view-dependent radiosity algorithm has been proposed in [Smit92] to avoid calculating the illumination of surfaces that are hidden from the viewer.

therefore be prohibitively expensive. We can reduce the amount of work required by a large amount by requiring a view-dependent solution, as in ray tracing.

To generate a such a solution we introduce our third concept, that of the *eyeball object*, which has the responsibility of producing a picture of the scene as seen from a particular point in space. To do this, it communicates with the patch objects of the scene in order to calculate this picture, ensuring that it is accurate to some user-specified figure. The patch objects then only have to determine the light they emit towards the eyeball, or towards other objects that reflect light towards the eyeball.

In order to explain the system more fully, we take a closer look at the concepts behind ranges and range arithmetic, and then present the full details of the patch and eyeball objects.

2.3.1 Using Ranges

An important advantage of using ranges in a rendering system is the elimination of point sampling. In global illumination work we are usually trying to determine a value that is representative of a function over some small area or interval. Point sampling occurs when we calculate this value by taking a sample of the function, usually from the centre of the interval in question. See figure 2.1 for an example.

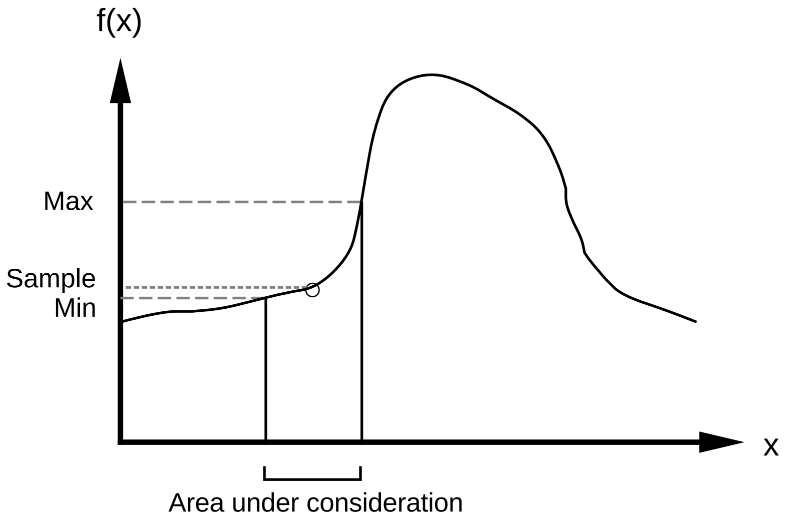


Figure 2.1 Ranges vs. Sampling.

As mentioned previously, the danger is that the sample taken may not be representative of the function over the interval, and unfortunately we have no way of knowing whether this is indeed the case. The sample in figure 2.1 is unrepresentative, as the function climbs quite steeply to its right.

Calculating a range for the interval in question requires more work, but, as well as providing a value for the function, it also gives an indication of the error in that value. We take the centre of the range, that is (min + max)/2, as our *estimate* of the function over the interval, and (max-min)/2 as the *error* of that estimate. It is then a simple matter to check that the error of the estimate is small enough for our needs. For instance, if we wish to ensure that over the interval the function always lies within 5% of our estimate, we need to ensure that (error/estimate) < 0.05.

If a range is not accurate enough for our requirements, we can subdivide the interval to produce two smaller intervals with correspondingly smaller errors. Figure 2.2 shows how this process can be applied to our original interval. The original interval has been subdivided once, so that the left-hand subdivision now meets some accuracy requirement. The right-hand subdivision was not accurate enough, and thus was in turn subdivided to produce two smaller subdivisions that do meet the accuracy requirement.

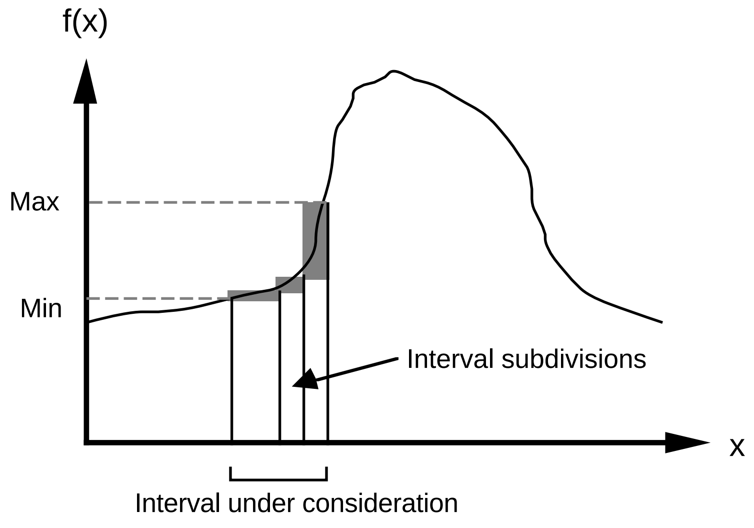


Figure 2.2 Interval Subdivisions

As a final note, we write ranges using the notation [minimum, maximum]. It is our practice to use lower case for a variable, and upper case for the range of that variable. We have also found it useful to use the convention that for a range A, A.top denotes its maximum, and A.bottom denotes its minimum.

2.3.2 Range Arithmetic

To manipulate ranges in a similar way to normal numbers, we use the concept of range arithmetic. Given some function of one or more variables, we can define a similar function that takes a number of ranges as arguments, and returns an output range. To find the range of numbers produced when applying a function f(x,y) to two ranges X and Y, for instance, we must find the minimum and maximum possible values of f when x is in the range X and y is in the range Y. More formally,

$$f(X,Y) = \left[\min\left(f(x,y)\right),\ \max\left(f(x,y)\right)\right],\ x \in X,\ y \in Y.$$

Range addition is the simplest example of range arithmetic. If we have two ranges A and B, then

$$A + B \quad = \quad [A.bottom + B.bottom, A.top + B.top].$$

Range multiplication, on the other hand, is a little more complicated due to the way the signs of two numbers interact when they are multiplied. When multiplying two ranges [a,b] and [c,d], there are actually four candidates for the bounds of the resulting range, namely ac, ad, bc and bd. The largest and smallest of these numbers give us our bounds.

This is a common theme of range arithmetic. Usually analysis of a function involves calculating several candidates for the minimum and maximum, which can then be sorted to find the true range of the function. To see why this is so we observe that finding f(X,Y) is equivalent to finding the minima and maxima of f(x,y) over the rectangle defined by

X.bottom <= x <= X.top,

Y.bottom <= y <= Y.top.

Over such a rectangle, the only possible candidates for minima and maxima are

- The four corners of the rectangle.
- Any local minimum or maximum that occurs along one of the edges of the rectangle. (For instance if f(x,Y.top) had a maximum that fell within X, this would be a candidate for the maximum of f(X,Y).)
- Any local minimum or maximum of f(x,y) that lies within the rectangle.

If we define f(x,y) = xy, the last two cases never occur, so that the only candidates for the minimum and maximum of the multiplication of the ranges X and Y are the four corner values of the rectangle.

Finding the range version of a function of one variable is even simpler than this; the only candidates for the bounds of f(A) are f(A.bottom), f(A.top), and any local maxima or minima of f that occurs within the range of A. Thus the possible candidates for the bounds of sin(X) are:

(i)	sin(X.top)	(iii)	1, if $\pi/2$ lies within X
(ii)	sin(X.bottom)	(iv)	-1, if $3\pi/2$ lies within X

A number of operations on ranges were defined in the course of implementing the renderer. The code for these can be found in the *URange* unit in appendix B.

The idea of using ranges and their application to adaptive problem solving has been formalised in [Snyd92], which was published towards the end of this thesis. The paper introduces the concept of *Interval Arithmetic*, and points out its application to several common graphics problems.

2.3.3 Role of a Patch Object

As has been stated previously, our approach to solving the global illumination problem involves setting up a system of patch objects, each of which represents a surface. These objects then communicate amongst themselves in order to determine the light flow in the scene.

The primary aim of a patch object is to determine the light that it emits towards other objects in the scene. To do this, it must in turn know what light it is receiving from the rest of the scene, assuming that it does indeed reflect light. (Light sources are a typical exception to this; in Computer Graphics they are usually assigned a reflectivity of zero, as any reflected light would be washed out by the light emitted by the light source.) Thus it determines what light is being emitted towards it by asking questions of the other objects, and then calculates how much of this light it reflects, and in what directions. If it is a light source, it adds in the light that it emits into the scene itself.

A patch object does not attempt to answer a question about the light it is emitting precisely. To do so would require the global illumination problem to already have been solved, which would make this rendering scheme redundant! Instead, it tries to determine the way a typical point on its surface emits light in different directions. Put another way, we can think of the patch object as trying to determine the light emitted by some fuzzy point, in the sense that the point is known to lie on the patch, but its exact location on that patch is unknown. Thus the patch's task is the calculation of the range of light emitted by the points on its surface for any direction.

Figure 2.3 provides a two-dimensional example of this. The way light is emitted at the points A, B and C is shown. The bounded distribution that the patch produces for its answer must enclose each of these distributions, as well as any other distributions on the patch.

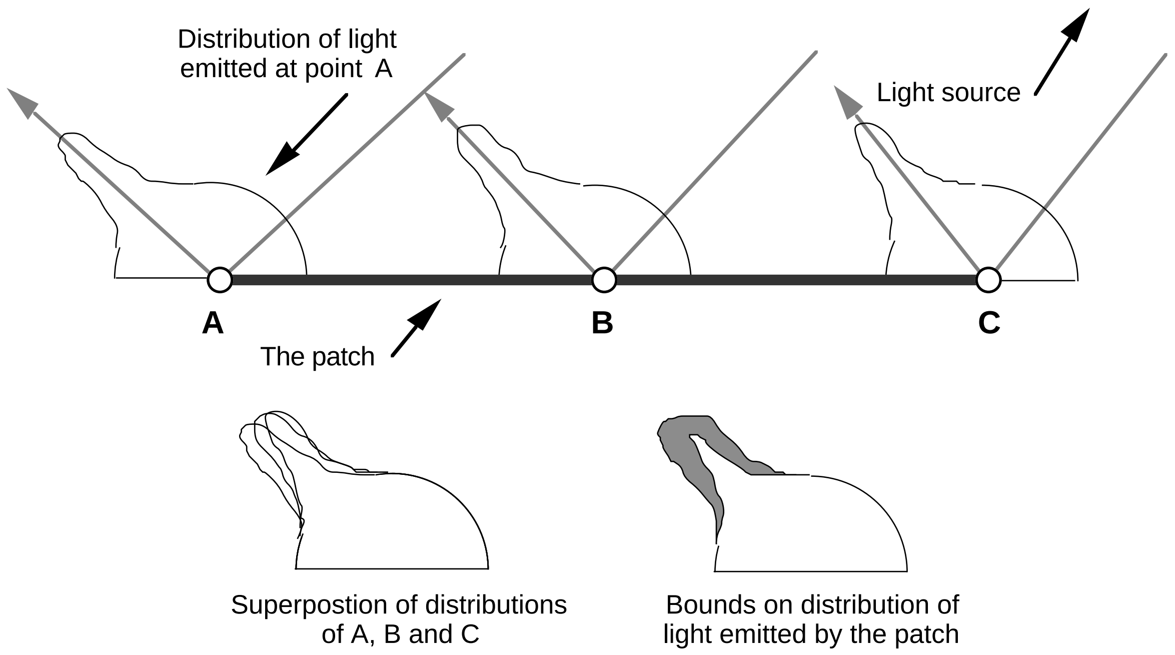

Distribution of light emitted at point A
Light source
A
B
C
The patch
Superpostion of distributions of A, B and C
Bounds on distribution of light emitted by the patch

Figure 2.3 The Range of Light Emitted by a Patch.

Because the answer that a patch object gives is a bounded one, we have a measure of its accuracy, as in section 2.3.1. This accuracy depends on two factors: the variation in reflection properties over the patch, and the accuracy of the answers of the other objects that contribute light to the patch. The accuracy of the other patches' answers is a matter for them to deal with, but if the patch object has too large a variation in the way it reflects light to give an answer that is accurate enough for its questioner, it must be able to improve this.

Any two points on a patch tend to reflect light differently because of their slightly different positions in space, and also possible variations in their surface normals or reflectivity. Due to continuity, the closer together that these two points are, the less the variation in their reflection properties. This suggests that if we subdivide the surface of a patch object to produce a number of smaller patch objects, the variation in reflection of its child subdivisions will be reduced, and these children will be able to provide more accurate answers than their parent.

A point that should be emphasised is that a patch object only needs to worry about that light that it emits towards those patch objects that are questioning it. The light that it emits into open space or towards uninterested patch objects can be ignored. (A good example of an uninterested patch object is one whose surface cannot receive light from the given object, either because it is facing the other way, or because there is some intermediate surface between the two.)

The next section examines some of the implications of communication between a system of patch objects.

2.3.4 A System of Patch Objects

To keep track of all the other patch objects that contribute light towards it, a patch object must keep a list of references to these patch objects. We can think of such references as providing links between the patch objects which illustrate the paths of communication between them. Following these links through a scene allows us to trace the possible paths of light flow between patches. Figure 2.4 below provides an example.

When we come to render a scene, we start off with a fairly coarse set of patch objects, in the sense that each object represents quite a large area of surface. As the rendering process proceeds, certain patch objects will find themselves unable to provide an answer that is accurate enough to satisfy their questioner, and will hence have to subdivide. When this happens, the links between patches must be recalculated, as in figure 2.4

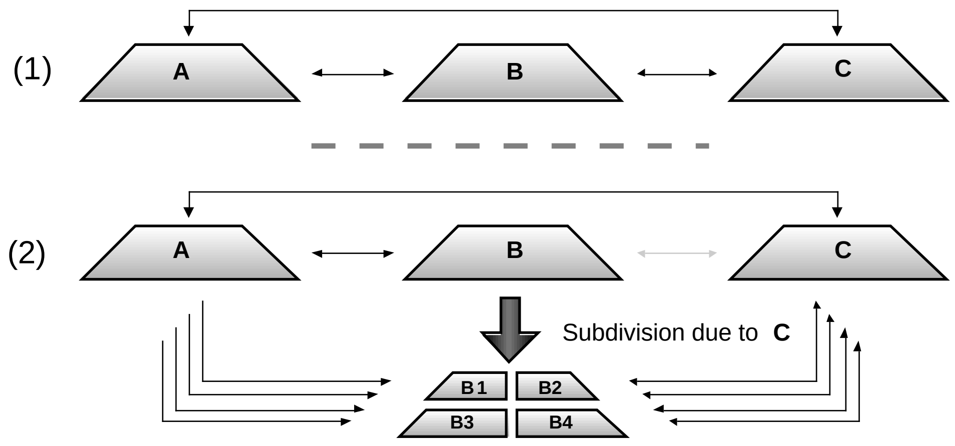


Figure 2.4 Patch Subdivision: Patch B Subdivides.

This subdivision process carries on until every patch is small enough to allow the calculation of an accurate enough solution to the global illumination problem. Once this point has been reached, the light emitted by light sources continues to filter through the patch objects until eventually the system settles into a steady state, this being the solution state.

A key point here is that subdivision doesn't affect uninvolved patches; they continue to communicate with top level of the object until it cannot answer their questions. For instance, in spite of patch B's subdivision in the second half of figure 2.4, patch A continues to carry on a dialogue with patch B, rather than B's children, although it must supply answers to those children.

If this were not the case, patch A would have to replace its reference to B with references to its four subdivisions. While this is not a problem in the example, in a scene with thousands of patch objects, the resulting proliferation of references would be horrendous. We therefore try to reduce this complexity by ensuring that a patch object only deals with the subdivisions of another patch object if it needs the extra accuracy that they provide.

As can be imagined, even a simple system of patch objects soon becomes highly convoluted after a number of subdivisions. Each original patch object lies at the root of a tree of all of its subdivisions, and every node of such a tree has links to all the other nodes it is presently communicating with. The resulting system of patches and links is affectionately dubbed the 'view forest'.

Another point about such a system is that communications loops are possible. The problem is illustrated in figure 2.5, where patch A is asking patch B to calculate the light that it emits towards it. In order to get an answer that is accurate enough for patch A, patch B may in turn decide to ask patch A to calculate its emitted light towards B. The loop starts if A again questions B; we become trapped in a 'B asks A asks B asks A ...' pattern of communication. A useful physical analogy here is that of two mirrors that are facing one another. Inevitably light trapped between the mirrors tends to bounce back and forth between them for a long time.

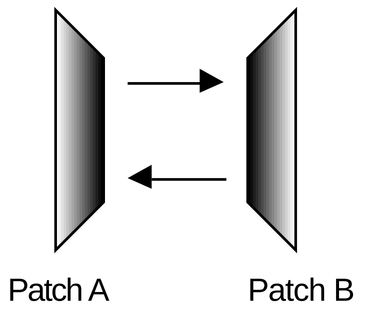


Figure 2.5 Looping between two patches.

Some kind of strategy must be put in place to deal with this problem in any implementation of fuzzy rendering. Ray-tracing handles such problems by placing a limit on the allowable number of reflections between objects. A different method was used in our implementation of fuzzy rendering; such looping situations were explicitly checked for and avoided, as explained in chapter 5.

2.3.5 Control of the Renderer

As mentioned previously, the eyeball object controls the renderer. Its purpose is to initiate and subsequently control the dialogue between patch objects in the scene, in order to produce the picture that it 'sees'. The eyeball inherits from the patch object, in the sense that it shares many of its common features, the most important of these being the ability to ask other patch objects about the light that they are emitting. However, there are some major differences:

- The eyeball object's associated surface has an elemental area.[1]
- The aim of an eyeball object is to capture the directional distribution of light that is incident on it, rather than the light that it emits. Therefore it doesn't have to deal with answering questions; it is not visible to other patch objects.
- It has different idea of accuracy. Most patch objects are trying to calculate answers for other patch objects, and are only interested in the straight-forward accuracy of light flows. An eyeball object on the other hand is trying to calculate an answer for the human eye. It needs to produce a picture that is *perceived* to be accurate to some percentage figure. To determine the relationship between this perceived accuracy and normal accuracy, we must consider the behaviour of the human eye. Chapter 5 develops this concept further.

The basic algorithm of the eyeball is as follows:

```
Start with a list L of references to each patch object in the scene.
repeat
  Ask a patch object for the light it emits towards the eyeball.
  (This patch object may have to consult other patches in a recursive process.)
  if the patch cannot supply an accurate enough answer (and thus subdivides and
        returns references to its children) then
    Delete the reference to the patch object in L
    Add the references to its children to L
until the perceived accuracy of all patch objects referred to by L is
  sufficiently accurate.
```

The next patch to be questioned can be selected by a method such as choosing the one with the worst perceived error. Once the entire process has been completed, the eyeball can then render its final list of patch objects to the screen using conventional techniques.

2.3.6 Summary

In summary, the major features of the scheme are:

- Surfaces are represented by patch objects that are responsible for determining the light that they emit into the scene, to the best of their ability.

[1]This is the case for the traditional "pinhole camera" model which is assumed by most global illumination algorithms. Making the area finite would allow a finite-aperture camera to be modelled, but not much thought has been put into this.

- They are capable of answering the question (from any other such object), “how much light do you shine towards me?” They give their answers as ranges: [bottom, top] pairs.
- If the answer provided by a patch is not accurate enough for other patches due to its size, it can subdivide itself into smaller parts.

2.4 Previous Work Done

The first work done on this proposal was by Guyon Roche for his Masters Thesis [Roch91]. He designed and implemented a two-dimensional version of the fuzzy rendering algorithm in object-oriented C. The restriction to two dimensions was made because of the simplifications it produced at an experimental stage of development. The goal was understanding, rather than a working three-dimensional renderer.

The renderer that was developed successfully produced bounded results to within a specified accuracy for simple scenes, usually consisting of a linear light source and one or two diffuse-reflecting lines. (A line being the two-dimensional equivalent of a three-dimensional surface.) Results were also produced for a scenes containing a specularly-reflecting line and a light source. The implementation of specular reflection was incomplete, however, in that it assumed a point light source.

Due to lack of time, a number of issues were not addressed. The rendering algorithm developed also had a couple of drawbacks, although the ideas of object-communication and subdivision worked very well. In particular, Guyon noted that:

- The hidden surface problem had been ignored, although some thought had been given to the issues involved.
- His renderer used an inefficient method of subdivision control, which sometimes caused subdivision of scene objects where it was not necessary.
- Question loops were prevented by a simple depth limit, which proved to have several drawbacks.
- A number of the calculations would have to be reworked substantially for a three-dimensional implementation.

2.5 Current Research Issues

At the start of this thesis, there were a number of issues that had to be addressed before a fully-fledged three dimensional renderer would be feasible. There are also a number of other issues that are important in understanding the algorithm. In this section we list these issues and examine the challenge they present.

2.5.1 Determining Light Transfer Between Patches

The first problem is that of calculating the light flow between patches. We need to determine the light reflected by a *receiving* patch in terms of the light emitted towards it from a single *source* patch. For any point on the receiving patch, this process can be summarised by the equation

$$R(x) = P(S,x)V(S,x)\rho(x),$$

where

S is the source patch,
x is a point on the receiving patch,
R(x) is the light reflected at x.
P is a function that determines the light emitted by S that is received at x,
V is a function determining the visibility of E from x, and
ρ is the reflectivity function of the surface at x.[1]

Once this is known, we then need to find bounds for R(x) over the receiving patch. It should be noted that to simplify the problem, bounds can be found for P, V and ρ separately, and then combined using range arithmetic. This then allows us to attack three subproblems , namely:

- Determining bounds on P(S,x) over the patch. This involves first finding an analytic formula for P, which involves considering the physics of the situation, such as the orientation of the source patch and its distance from the point in question. This function then has to be bounded over the receiving patch.
- Determining bounds on ρ over the patch. This problem is not so difficult, as usually in computer graphics surfaces are assumed to have constant reflectance functions. For surfaces where this is not true, such as texture-mapped surfaces, we must be able to produce bounds on the reflectance function over any area of the surface.
- Determining bounds on V. We refer to this as the ‘Fuzzy visible-surface problem’; it is covered in more detail below.

2.5.2 The Fuzzy Visible-Surface Problem

The visible-surface problem for a point in space has been studied intensively, and is currently well understood. Solving a similar problem for a surface, however, involves consideration of how the visibility of objects from different points on the surface varies, and has had little research effort put into it.

[1]It should be noted that, although it is convenient to think of P and ρ as scalar-valued functions, they in fact represent the directional distribution of light.

Figure 2.7 shows a simple example of the problem in two dimensions. There is a single light source and a single occluder, resulting in a pattern of light and shadow being cast on the receiving patch.

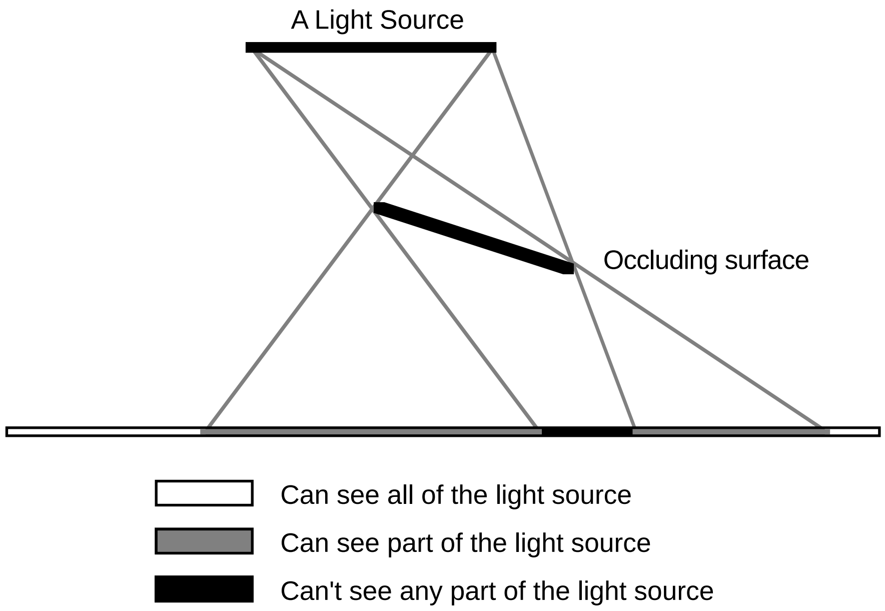

A Light Source
Occluding surface
Can see all of the light source
Can see part of the light source
Can't see any part of the light source

Figure 2.7 The Fuzzy Visible-Surface problem.

A possible solution to the fuzzy visible surface problem is to use a Warnock-type view subdivision algorithm. With any view region, a list of those patches that are visible would be stored. The accuracy of any view region could then be improved by subdividing it. For each of the new regions produced, references to those patches that either didn't intersect it, or were completely covered by some other patch, could be removed.

2.5.3 Control of the Algorithm

In Guyon's renderer, required accuracy was distributed among patch objects by the eyeball, and the patch objects were then required to produce answers to that accuracy. Problems with this arose because accuracy can be traded off between patch objects. For instance, for two patch objects that are equally important in determining the accuracy of the picture, the obvious solution is to ask them both to produce an answer accurate to the same percentage, say 10%. However, one of these objects may find it much easier to calculate an accurate answer. Thus it might be possible to ask that patch object for an answer accurate to 5%, while the other only has to work to get an accuracy of 15%, producing a solution of the same accuracy, but with less work involved.

In order to overcome this problem, it is best that a control algorithm be progressive, in that it improves the accuracy of its current answer by small steps as it moves towards a solution of the required accuracy. The emphasis is thus on asking patch objects to improve their current answer, rather than having them produce an answer accurate to a certain figure. If the renderer is implemented in this way, then it should be possible to produce heuristics capable of predicting the best patch to ask next for improvement, given the aim of having the greatest increase in accuracy for the least amount of work.

Another point is that such an algorithm should also take into account the behaviour of the human visual system when considering accuracy issues. For instance, an important point is that the eye registers intensity on a log scale, and thus the dimmer sections of a picture will require a smaller absolute error than the brighter sections.

Finally, there are two desirable goals for such an algorithm. Firstly, the algorithm must terminate. Secondly, the algorithm should maintain a reasonable spread of accuracy over its picture at any stage, rather than just working on one part of it at a time.

2.6 Goals of this Thesis

The major goal of this thesis was to design and implement a three-dimensional version of the renderer, which could then be used as a testbed in order to investigate the behaviour of the algorithm. Before this could be done, an investigation into the fuzzy-rendering technique was needed, in order to establish a more formal model of how it worked. The results of that investigation have been presented in this chapter, and established that 3D rendering would be a fairly complex problem. In order to simplify it to the point where it could be attempted in a Masters thesis, several assumptions were made with respect to the surfaces in the scene. These were:

- Polygonal, flat surfaces. All objects in the scene were assumed to be polyhedrons or polygons.
- No obscuring surfaces. The visible-surface problem was ignored.
- Lambertian Surfaces. All objects were assumed to only reflect light in a diffuse manner.

The first assumption was made in order to eliminate many of the problems that could arise with curved surfaces, such as self-reflection. It was felt that flat, polygonal surfaces would present a reasonably challenging exercise, the results of which could eventually be extended to curved surfaces. The second assumption was made because for simple scenes the visible-surface problem could safely be ignored while still producing valid results. It seemed to be sensible to postpone its consideration until the renderer had proved feasible.

The last assumption removed the problem of considering directional distributions of light. As a lambertian surface reflects light uniformly into space, we can measure the light it emits by a single scalar, rather than having to give a directional distribution for the light emitted, and thereby calculations are simplified enormously. Given the parallels between normal radiosity and specular radiosity presented in chapter 1, it was hoped that any results could be extended to a specular renderer.

Another goal was to establish the dynamics of the rendering process, namely what controls convergence, and how the system behaves over time. Of interest here are factors such as how the number of patch objects required varies with the accuracy of the picture, and how the accuracy of the picture increases over time.

It was also hoped that the role of object-oriented concepts in the method could be better established. Such concepts were useful in the development of the method, but it was suspected that they were not fundamental to it. One possible avenue of research was to explore the method's relationship to linear algorithms such as specular radiosity.

In summary, however, the major goal set was try to solve the problem of correctly rendering a scene containing just two or three square "patches" with trivial reflectance characteristics.

CHAPTER 3

Light Flows and Patch Factors

3.1 Introduction

Chapter 2 established the need for finding bounds on the amount of light passing between a source patch and a receiving patch. In this chapter, we develop a method for achieving this. Firstly we give an introduction to the physics behind the transfer of light in order to establish a framework for what follows. The concepts presented are then used to develop a method for calculating the distribution of light that a diffuse, polygonal source patch induces on another surface. Some examples of such light distributions are given.

A scheme for finding approximate bounds on such a distribution over a rectangular patch is then introduced and explained. The results this method produces are profiled against theoretical results using a number of tests, and conclusions about its viability drawn.

3.2 A Mathematical Basis for Light Transfer

Being a young science, Computer Graphics uses many rather ad hoc and empirical methods. Surface-reflectance modelling is a case in point; it is only in recent years that research was done into physically-based surface reflection models, and these are still not in widespread use. However, as the emphasis in high-quality image rendering is now shifting towards work with a physical basis, a proper understanding of the theory is necessary[1]. The field seems to have settled on a reasonably standard approach to the physics of light flow over the last few years; typical is the treatment given in [Hall89]. This section provides a brief overview of this approach, and the concepts established are then used to calculate the transfer of light between patches.

[1]This is due to the ever-increasing power of computers making such work feasible, and can be viewed amusingly as a variant of Parkinson's law — 'work expands to fill the time available.'

3.2.1 The Physics of Light

Light is often represented in Physics as a wave, quantified by a phase and an amplitude at any point and time in space. In Computer Graphics the phase information is usually neglected, as it is not important except when considering more specialised effects such as interference or diffraction. We consider only the square of amplitude, which is proportional to the energy of the wave. We measure the “amount” of light by **Light Flux**, often denoted as Φ, this being defined as energy per unit time, which is measured in Joules per second, i.e., Watts.

We also assume that a scene is made up of a collection of surfaces. When considering the light flow in such a scene, we need to understand the way such surfaces emit and reflect light[1]. We start by considering how light is emitted by point light sources, and then develop the equivalent model for surfaces.

3.2.2 Point Light Sources

When considering the light being emitted by a point in space, we need some way of quantifying the distribution of the light flux it emits over different directions. In two dimensions we would handle this by measuring the light flux with respect to the size of the angle it was emitted into, which we measure in radians. The three-dimensional equivalent of this is the **solid angle**, which is measured in steradians; figure 3.1 illustrates the concept for an area A and a point P. To find the solid angle that A subtends, we would project it onto a unit sphere centred at P. The solid angle is defined to be the area of this projection. A consequence of this is that any surface that completely surrounds P will subtend a solid angle of 4π – the area of the unit sphere.

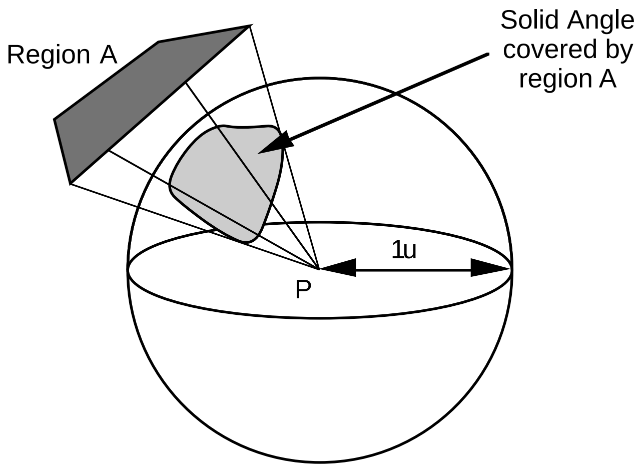


Figure 3.1 Solid angles.

[1]In the following discussion we assume that these surfaces are opaque, although transparent surfaces could be handled in a similar manner.

The light flux emitted from a point light source in a certain direction is called the **radiant intensity**[1] of that point, and is measured in Watts per steradian. If we have a point light source emitting a constant light flux Φ, and this is radiated evenly in all directions, then its radiant intensity in any direction will be $\Phi/4\pi$.

3.2.3 Light Flux and Surfaces

The surface equivalent of a point light source is the **elemental surface**, which has an associated elemental area and normal, as in figure 3.2. The elemental surface can be used to establish the properties of real surfaces by integration.

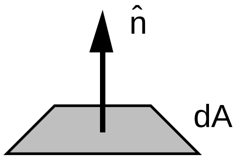


Figure 3.2 An elemental surface.

The elemental-surface equivalent of the light flux emitted by a point light source is the **light flux density**, which is defined as the light flux per unit area of the surface. If this light flux is emitted by the surface, we call it **radiosity** (B), and if is incident on the surface it is termed **irradiance** (E). For instance, if we have a point light source P and an elemental surface dA, as in figure 3.3, then the light flux incident on the surface will be

(3.1) $$d\Phi = I_P d\omega = I_P \frac{Cos(\theta)dA}{r^2},$$

where I_p is the radiant intensity of the point source. Thus the irradiance of the surface is

(3.2) $$E = \frac{d\Phi}{dA} = \frac{I_p Cos(\theta)}{r^2} .$$

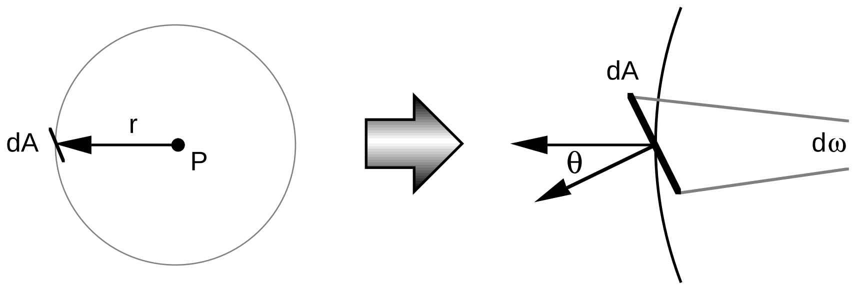


Figure 3.3 Light flux density

The surface equivalent of radiant intensity is **directed light flux density**, also known as **radiance**. It can be derived by treating an elemental surface much like a point light source, with one modification. We have to take account of the fact that from directly above the surface we see it in its entirety, whereas as we move to a side-on view its apparent size becomes less and less, until from

[1]The “radiant” is helpful in distinguishing it from other concepts of intensity.

directly side-on it is no longer visible. (Any surface is, by definition, infinitely thin.) In order to do this, we take the light flux emitted relative to the *projected* or *foreshortened* area of the surface, rather than its actual area.

Thus we define the radiance of an elemental surface as the light flux per unit solid angle, per unit projected area. For an elemental surface of radiance I, as in figure 3.4, the light flux emitted over the solid angle shown will be

(3.3) $$\Phi = I \; Cos(\theta) dA \; d\omega .$$

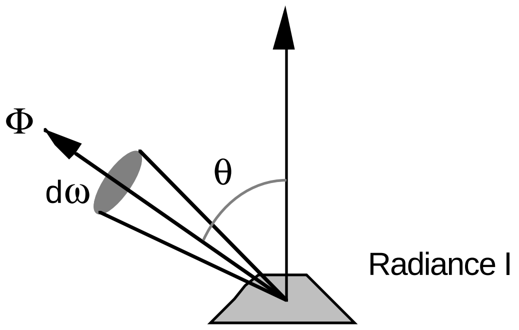


Figure 3.4 Directed light flux density.

We usually label radiance as "I" because it represents the intensity of the surface. Obviously "intensity" can be quite confusing, because it can be taken to refer to a number of different concepts. In Computer Graphics we generally use it to refer to the brightness of an object, and in different situations radiant intensity, radiosity and radiance can all be a measure of this, depending on what the object is, and how it emits light. In Physics intensity also has a number of conflicting definitions, so at the beginning of any discussion that uses the term, it is best to define what type of intensity is being talked about.

3.2.4 Bidirectional Reflectivity

When light flux is reflected from an elemental surface, the resulting outgoing light flux is distributed over different directions. Different types of surface display different reflection characteristics; light reflected from a shiny surface is strongly directional, whereas a matte surface scatters reflected light over all directions. In any case, the amount of outgoing light flux in any direction depends on both the incoming and outgoing directions of the reflected light, and hence it is bidirectional.

This behaviour is captured by the bidirectional reflectance distribution function (BRDF), which maps incoming irradiance at the elemental surface to the outgoing radiance. It is defined by the equation

(3.4) $$I = k_0 \, \rho(D_i, D_o) E,$$

where ρ is the BRDF, and D_i and D_o denote the incoming and outgoing directions of light. Such directions are represented either by unit vectors, or pairs of polar coordinates (θ, ϕ), taken with respect to the surface's coordinate system. Lambertian reflection is the simplest form of bidirectional reflectivity, as the BRDF is a constant for Lambertian surfaces. Thus it can be represented by a scalar ρ, which is usually termed the *reflectivity* or *reflectance* of the surface, as in section 1.4.1.

Many common surfaces are isotropic, meaning that their BRDF is invariant under rotation of the surface in its plane. (No matter what direction you look at the surface from, its reflection characteristics remain the same.) For such surfaces the only parameters needed by the BRDF are the incoming and outgoing direction vectors, and the surface normal. Surfaces that don't display such characteristics are called anisotropic[1].

3.2.5 Finding k_0

For convenience, we would like to find a value for k_0 of equation 3.4 so that ρ=1 defines a perfectly-reflecting, Lambertian surface. If we integrate the radiance of the elemental surface over its covering hemisphere, we will get the radiosity of that surface. Assuming that the surface is not itself emitting any light, and that it is a perfect reflector, we can equate its radiosity with its irradiance, giving

$$\int_H I\, Cos(\theta)\ d\Omega \ = \ E. \tag{3.5}$$

For a polar coordinate system, $d\Omega = Sin(\theta)d\theta d\phi$,[2] so the equation becomes

$$E \ = \ I\int_0^{2\pi}\int_0^{\pi/2} Cos(\theta)Sin(\theta)\ d\theta\ d\phi. \tag{3.6}$$

Integrating, and substituting equation 3.4 gives us a value for k_0,

$$k_0 = \frac{1}{\pi}. \tag{3.7}$$

We can use a similar technique to find a normalisation condition for any perfectly-reflecting BRDF, namely that

$$\int_H \rho(\theta_i, \theta_o)Cos(\theta_o)\partial\omega = \pi. \tag{3.8}$$

3.2.6 Energy Transfer Between Patches

As an exercise, we can see how these concepts work when it comes to determining the transfer of energy between two elemental patches, as in figure 3.5.

1 Most Computer Graphics work involves isotropic surfaces, although [Kaji85] has investigated anisotropic textures.
2 Assuming θ is measured from the vertical axis.

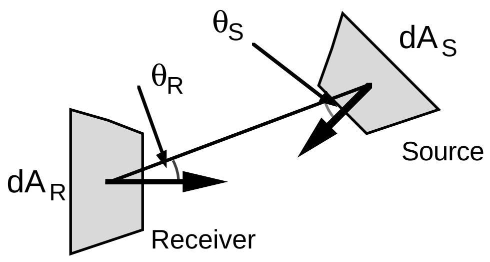


Figure 3.5 Two elemental patches

We assume that the source patch has a radiance of I_S in the direction of the receiving patch. If we equate the light flux emitted towards the receiving patch with that it receives, we find that

(3.9) $$E_R dA_R = I_S \, Cos(\theta_S) dA_S \, d\omega_R,$$

where $d\omega_R$ is the solid angle that it subtends when viewed from the source. This can be written in terms of dA_R by using the relationship $r^2 d\omega = dA$, so that

(3.10) $$E_R dA_R = I_S \, dA_S Cos(\theta_S) \, \frac{dA_R Cos(\theta_R)}{r^2},$$

and thus

(3.11) $$E_R = I_S \frac{Cos(\theta_S) Cos(\theta_R)}{r^2} dA_S.$$

When both patches are diffuse, we can use this to find the relationship between the radiosity of the source patch and the irradiance of the receiving patch. This irradiance can then be used to calculate the radiosity of the receiving patch, as per equation 1.1. For diffuse patches, $I = B/\pi$, so that

(3.12) $$E_R = B_S \frac{Cos(\theta_S) Cos(\theta_R)}{\pi r^2} dA_S.$$

This equation is commonly used to calculate the form factors between patches in the radiosity rendering method.

3.2.7 Summary

A summary of the quantities we must deal with is presented in the following table.

Quantity	**Represents**	**Units**
Light Flux	Energy wave passing through space.	Watts
Intensity (Radiant Intensity)	Light radiated by a point source.	Watts per steradian.
Light Flux Density (Irradiance/Radiosity)	Light incident on/emitted by a surface. (Non directional)	Watts per unit area.
Directed Light Flux Density (Radiance)	Light emitted by a surface. (Directional)	Watts per steradian per unit projected area.

3.3 Derivation of the Patch Factor Equation

As explained in Chapter 2, we need to determine bounds on the light flow from one patch to another, assuming that the source patch has lambertian surface. To find the irradiance at a single point on the receiving patch involves integrating equation 3.12 over the emitting patch. The source's radiosity B_s can be moved outside the integral, which when evaluated will produce a ratio between it and the corresponding irradiance. We refer to this ratio as the *patch factor* from the source to the point, in order to avoid confusion with the form factor of radiosity, which is the area-weighted integral of the patch factor over the receiving patch.

In this section we develop a method for calculating a patch factor. We then look at ways of finding the range of values the patch factor takes over the receiving patch.

3.3.1 The Viewing Circle

At this point it is useful to introduce a mechanism to help in the calculation of the light flux incident at a point on a surface. As has been pointed out by [Sieg81], the patch factor from any surface to a view point can be found by projecting that patch onto a sphere about the point, and then dropping this projection down onto the surface the point lies on. (Briefly, the normalisation converts the area of the patch into a solid angle, and the projection takes care of the foreshortened area part of equation 3.9.) We refer to this process as projection onto the view circle, as it maps the space around the point onto a unit circle. (See figure 3.6.) The patch factor is then the ratio of the view-circle area of the patch to the view-circle's total area (which is π).

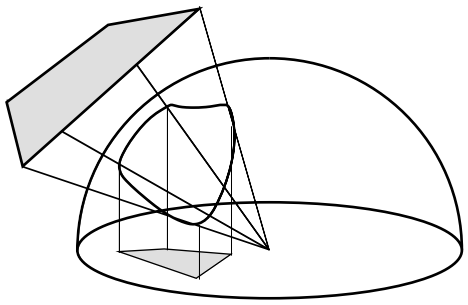


Figure 3.6 Projecting onto the view circle.

The projection of a point from world space coordinates into the view-circle coordinates of a particular view point is simplified if we first transform the scene so that the x-y plane is normal to the surface that the view point lies on, and the view point lies at the origin. The projection then involves normalising the vector from the origin to the point in question, and dropping the z coordinate, so that

$$[x,y,z] \rightarrow \frac{[x,y]}{\|[x,y,z]\|} = \frac{[x,y]}{\left(x^2+y^2+z^2\right)^{1/2}}. \tag{3.13}$$

3.3.2 View Circle Areas

The conventional method for calculating the area of a patch that has been mapped onto the view circle is to use the hemicube algorithm. Unfortunately this algorithm suffers from point-sampling artefacts, as well as geometrical errors [Baum89], and is not really appropriate for our ideas.

To replace it we have developed a technique for calculating such areas by using the concept of an edge area[1]. This is defined as the area bounded by an edge and the origin of the coordinate system, giving a triangular shaped wedge. As is shown in figure 3.7, the area of an arbitrary polygon can be calculated by adding or subtracting the various edge areas together.

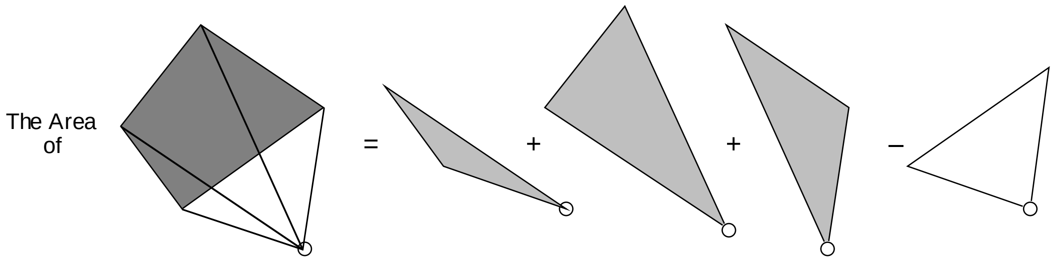


Figure 3.7 Summing edge areas to find a polygonal area (in 2D)

To find out whether we should add or subtract a particular edge's area, we regard the polygon as being made up of directed edges, in either a clockwise or anti-clockwise fashion. If the origin lies on the same side of the edge as the inside of the polygon, then the edge's area is added to the total. If not, it is subtracted. Of course, after projection into the view circle these straight edges become curved, but the same idea still applies to finding the area they enclose.

In summary, we can write the patch factor from a polygonal patch to a point as

$$PatchFactor = \frac{1}{\pi}\sum_{i=1}^{n} EdgeArea(i), \tag{3.14}$$

which we call the patch-factor equation (PFE).

3.3.3 Finding Edge Areas

This leaves us with the task of finding the edge area of a line segment in space that has been mapped onto the view circle. Such a line segment can be represented parametrically by

$$\tilde{c} = \tilde{p} + t\tilde{v} \qquad ; \; t \in [0,1]. \tag{3.15}$$

Figure 3.8 illustrates the situation.

[1]What follows was derived independently from the analysis in [Hott67] for the field of radiative heat transfer.

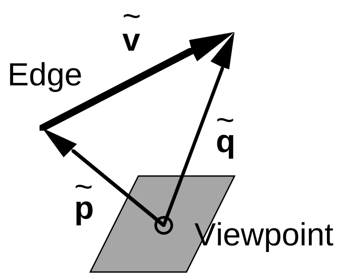


Figure 3.8 An edge and a viewpoint.

Transforming this into view-circle coordinates gives a 2D curve defined by

$$\tilde{c}' = \frac{(p_x, p_y) + t(v_x, v_y)}{\|\tilde{p} + t\tilde{v}\|}. \tag{3.16}$$

We need to find the area bounded by this curve and the origin. In polar terms this will be

$$Edge \;\; Area = \int_{\theta_1}^{\theta_2} \tfrac{1}{2} r^2 d\theta. \tag{3.17}$$

We then relate r and θ to the curve defined by c'. From equation 3.16 a bit of algebra gives us

$$r^2 = \frac{(p_x + tv_x)^2 + (p_y + tv_y)^2}{\left(\|\tilde{p}\|^2 + 2t(\tilde{p}.\tilde{v}) + t^2\|\tilde{v}\|^2\right)}, \tag{3.18}$$

and

$$Tan(\theta) \;=\; \frac{p_y + tv_y}{p_x + tv_x}. \tag{3.19}$$

If we differentiate equation 3.19 using the chain rule we get

$$Tan'(\theta)d\theta \;=\; \frac{v_y p_x - v_x p_y}{(p_x + tv_x)^2} dt, \tag{3.20}$$

and substituting a trigonometric identity

$$Tan'(\theta) \;=\; 1 + Tan^2(\theta), \tag{3.21}$$

as well as equation 3.19 gives

$$d\theta \;=\; \frac{v_y p_x - v_x p_y}{(p_x + tv_x)^2 + (p_y + tv_y)^2} dt. \tag{3.22}$$

Putting this together with equation 3.18, we can restate the edge-area integral in terms of our original line segment, so that

$$Edge \;\; Area \;=\; \int_0^1 \frac{v_y p_x - v_x p_y}{2\left(\|\tilde{p}\|^2 + 2t(\tilde{p}.\tilde{v}) + t^2\|\tilde{v}\|^2\right)} dt \tag{3.23}$$

It turns out that the bottom line of this equation always has a determinant less than zero, so it cannot be factorised. Thus we can use the integration formula for the arctangent function to give us our final formula

$$(3.24)\qquad Edge\ \ Area\ = \frac{v_y p_x - v_x p_y}{2\left(\|\tilde{p}\|^2\|\tilde{v}\|^2 - (\tilde{p}.\tilde{v})^2\right)^{\frac{1}{2}}} ArcTan\left(\frac{\left(\|\tilde{p}\|^2\|\tilde{v}\|^2 - (\tilde{p}.\tilde{v})^2\right)^{\frac{1}{2}}}{\|\tilde{p}\|^2 + (\tilde{p}.\tilde{v})}\right),$$

which is not as complicated as we might have expected.[1]

3.3.4 Different Interpretations of the Edge-Area Equation

A couple of substitutions make this edge-area equation more intuitively obvious. If we substitute $q = p + v$, then we can rewrite it as

$$(3.25)\qquad \begin{aligned} Edge\ \ Area\ &= \frac{e_z.(p\times q)}{2\|p\times q\|} Arctan\left(\frac{\|p\times q\|}{p.q}\right), \\ &= \frac{e_z.(p\times q)}{2\|p\times q\|}\theta_{pq}, \end{aligned}$$

where θ_{pq} is the angle between the vectors p and q, as in figure 3.9.[2] The cross product of p and q turns out to be twice the area of the triangle formed by the edge and the viewpoint. The $e_z.(p \times q)$ factor is twice the area of this triangle after it has been projected onto the x-y plane. Thus a further rewrite of the equation gives us

$$(3.26)\qquad Edge\ \ Area\ = \frac{P_{pq}}{2A_{pq}}\theta_{pq},$$

where A_{pq} is the area of the edge triangle, P_{pq} is its projected area, and θ_{pq} is as before.

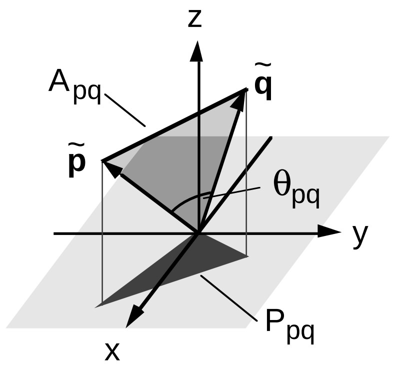


Figure 3.9 Alternate views of the edge formula.

[1]This process of working out edge areas, and then summing such areas to find an area integral is akin to applying Stoke's theorem to convert a surface integral into a contour integral.

[2]The vector e is traditionally taken to be the vector [1,1,1], so e_z is [0,0,1].

The third and last way of looking at the edge area is to realise that the ratio of the areas A_{pq} and P_{pq} is actually the cosine of the angle between the triangle's normal and e_z. If we call this angle ϕ, then we have

$$\text{Edge Area} = \tfrac{1}{2} Cos(\phi_{pq})\theta_{pq} . \quad (3.27)$$

3.4 Some Example Patch Distributions

The PFE was used to produce plots of the patch factor for a couple of example source-receiver patch pairs. Figure 3.10 illustrates a cube similar to the scene used in the seminal radiosity paper [Gora84].

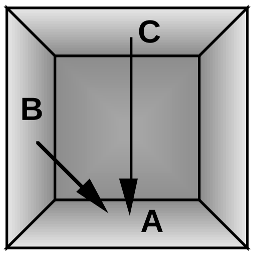


Figure 3.10 A test cube

There are two possible orientations of source and receiving patch for this cube; figure 3.11 displays the results for both. The left-hand graph shows the patch factor distribution over patch A when an overhead patch (patch C) is acting as a light source. The right-hand graph shows this distribution for the case when an abutting patch (patch B, for example) is the source.

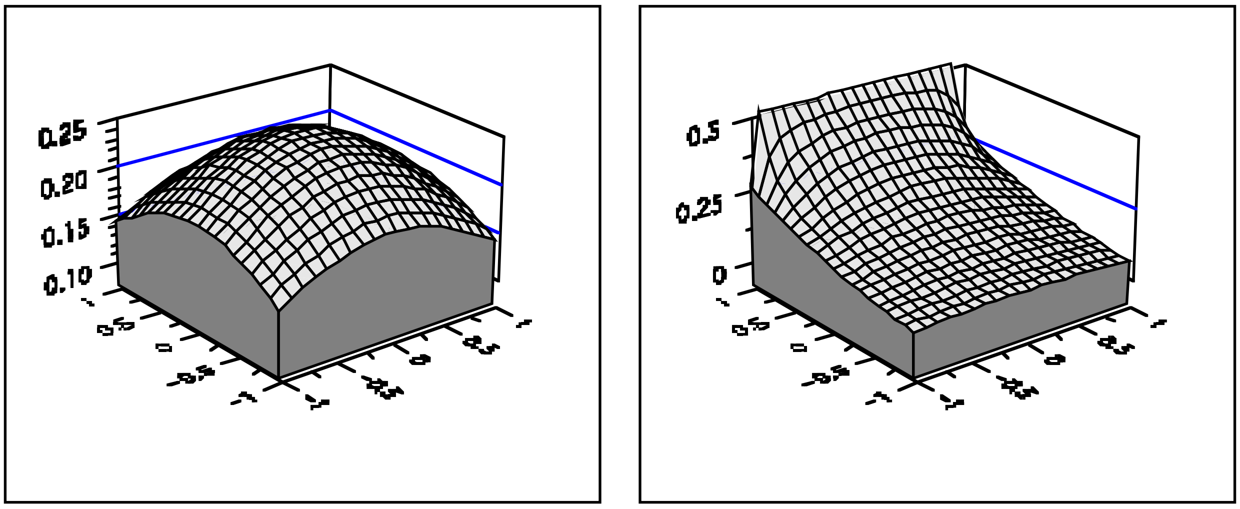


Figure 3.11 Possible Patch Factor Distributions

The first graph displays what we would expect; a symmetrical distribution that has a peak in the centre (where the patch is directly overhead), and falls off to either side.

The second graph shows the patch factor dropping down from a maximum of 0.5 along the common edge of the two patches to almost nothing on the opposite side of the receiving patch. An interesting point to note is that there is a discontinuity in the distribution at either end of the common edge; at these two points the function drops straight down to 0.25. This discontinuity is caused by the abrupt edge of the emitting patch. If other such patches are placed adjacent to it, combining their density factor distributions causes the ‘corner’ values to sum to 0.5, resulting in a continuous distribution.

3.5 An Analytic Bounding of the PFE

Now that we have an expression for the incoming irradiance at some point on a receiving patch due to a source patch, we need to find bounds for this function over the patch. As these bounds are a guide to whether further subdivision of a patch is necessary, we make the observation that the bounds need not be accurate. They must enclose the accurate bounds, or subdivision may fail to take place when it is needed. If the bounds are ‘looser’ than the actual bounds, however, the consequences are unnecessary subdivision, and thus more work for the renderer, but ultimate accuracy is not compromised. As a result it was decided to take the approach of finding rough bounds with reasonably simple techniques, and then concentrate on improving the accuracy of these bounds only if it proved necessary.

It was decided that the easiest first approach would be to individually bound the edge terms of the PFE, and then sum them using range arithmetic. While this has the drawback that it isn’t as accurate as bounding the entire sum, it was postulated that as the size of the patch decreased, the bounds produced would approach the actual bounds. The next few sections develop a method for producing such edge term bounds. We refer to this as the *simple* patch-factor bounds (PFB) method.

3.5.1 Bounding an Edge Term

The edge term equation of section 3.2.4 is written in terms of three major quantities involving *p* and *q*; the dot product, the length of the cross product (A)[1], and the z-component of the cross-product (P). We reproduce it here for reference.

$$\text{Edge Area} = \frac{e_z.(p \times q)}{2\|p \times q\|} \operatorname{Arctan}\left(\frac{\|p \times q\|}{p.q}\right) = \frac{P}{2A}\theta. \tag{3.28}$$

To bound this edge term over a patch, we need to consider how it varies as we move the viewpoint, and thus *p* and *q*. Figure 3.12 illustrates the situation for a rectangular patch.

[1]In fact the P and A of equation 3.28 are equal to half these quantities, but as we are looking at their ratio, we ignore this for convenience.

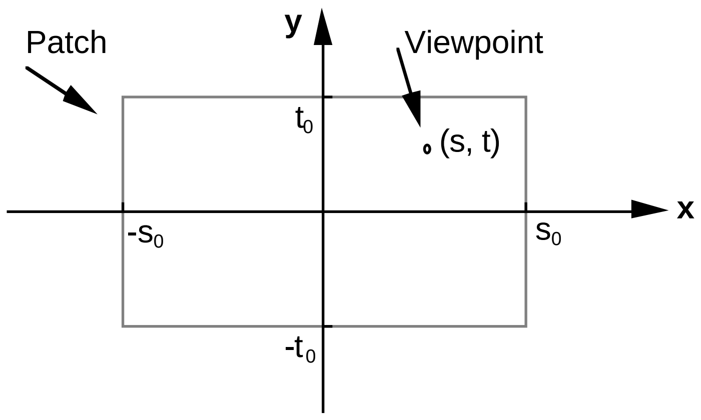


Figure 3.12 Coordinate system for a patch

To do this we perform the substitution

$$\begin{aligned} \tilde{p}' &= \tilde{p}' + s\tilde{e}_x + t\tilde{e}_y, \\ \tilde{q}' &= \tilde{p}' + s\tilde{e}_x + t\tilde{e}_y. \end{aligned} \tag{3.29}$$

For the cross product term, this gives us[1]

$$\begin{aligned} \|p' \times q'\|^2 = \ & \|p \times q\|^2 + s^2\left(a^2 - a_x^2\right) + t^2\left(a^2 - a_y^2\right) \\ + \ & 2s\left(q_x\left(p.q - p^2\right) + p_x\left(p.q - q^2\right)\right) \\ + \ & 2t\left(q_y\left(p.q - p^2\right) + p_y\left(p.q - q^2\right)\right) \\ - \ & 2st\left(a_x a_y\right), \end{aligned} \tag{3.30}$$

where the vector a is defined as $a = q - p$. For the dot product

$$p'.q' = p.q + s\left(p_x + q_x\right) + t\left(p_y + q_y\right) + s^2 + t^2, \tag{3.31}$$

and the projected cross product is

$$\left(p' \times q'\right).e_z = \left(p \times q\right).e_z + sa_y - ta_x. \tag{3.32}$$

Substituting these back into equation 3.25 gives a rather horrendous result which is difficult to handle. While differentiating it is possible, solving for local minima or maxima analytically appears to be intractable.

To get around this, we again observe that we aren't after perfect bounds. The approach taken is to instead find bounds for the three quantities mentioned above, and use range arithmetic to get a range for the overall term.

[1]For a derivation of this, see Appendix A.

3.5.2 Component Bounds

The bounds for the projected area term (P) are easy to calculate, it being a linear equation. We find that over the rectangular area defined by $|s|<s_0$ and $|t|<t_0$, it is bounded by

$$\left[(p\times q).e_z + s_0 Abs\left(a_y\right) + t_0 Abs\left(a_x\right),\ (p\times q).e_z - s_0 Abs\left(a_y\right) - t_0 Abs\left(a_x\right)\right] \tag{3.33}$$

The bounds on the other two terms are calculated by using a generic function for finding the bounds on a quadratic in two variables over a rectangular area. This function checks for minima and maxima within the bounds of s and t that we're interested in, and if either of these don't exist it looks for minima and maxima along the edges of these bounds. (See appendix A.)

The only problem in combining the individual terms into a range for the edge term is in bounding the arctan function. The naive approach would be to individually bound the cross product and dot product terms, and then use range division and range arctan functions on them. However, this ignores the way the A and θ terms relate to each other. For small values of θ, the arctan(θ) term tends to θ, and the length-of-the-cross-product terms in the equation cancel out. Even if we had large variations in these terms individually, the resulting range of θ/A would be small.

Luckily, it can be shown that for a composite function z of the form

$$z = \frac{1}{f} ArcTan\left(\frac{f}{g}\right), \tag{3.34}$$

the partial derivatives dz/dg and dz/df are always negative. (See appendix A for details.) Thus z decreases with both f and g, and if we have [bottom, top] ranges for f and g, the bounds on zare

$$\left[\frac{1}{f.top} Arctan\left(\frac{f.top}{g.top}\right),\ \frac{1}{f.bottom} Arctan\left(\frac{f.bottom}{g.bottom}\right)\right]. \tag{3.35}$$

As θ/A is equal to z when f is $\|p'xq'\|$ and g is $p'.q'$, as in equation 3.28, we have our bounds for θ/A.

3.5.3 The Abutting Patches Problem

As it stands, this procedure fails when the vertex of an edge lies on the receiving patch. In such a situation θ/A tends to infinity as the view point approaches the vertex (θ remains finite while A tends to zero), while at the same time P tends to zero. In a scene this case most commonly occurs when two patches share an edge (i.e., abut one another). The shared edge and the edges leading off from it cause problems when viewed from either patch.

To solve this problem we look at the limit of the P/A ratio as we approach such a vertex. It can be shown that this limit is

$$R_{Max} = \sqrt{1 - \frac{a_z^2}{\|a\|^2}}, \tag{3.36}$$

and that R_{max} is an upper bound for $|P/A|$.[1] Given this, a quick way to eliminate the singularity in the bounds is to clip them to $\pm R_{max}\theta_{max}$, as these are guaranteed limits for the bounds. By a method similar to that used in the previous section, it can be shown that the minimum and maximum of arctan(F/G), where F and G are both ranges, is given by[2]:

Bottom of range of g	**Maximum of Arctan(F/G)**	**Top of range of g**	**Minimum of Arctan(F/G)**
0	$\pi/2$	0	$\pi/2$
negative	Arctan(F.bottom/G.bottom)	negative	Arctan(F.top / G.top)
positive	Arctan(F.top / G.bottom)	positive	Arctan(F.bottom / G.top)

Thus we can find a maximum for theta. This approach proved very successful in eliminating the singularities. It can be extended by observing that for an edge with both vertices in the plane of the receiving patch, A is the same as P, and hence the edge term is given by $\theta/2$. For such a case we use the table above to find bounds for the edge term directly.

3.6 Testing the Simple PFB

Once this simple PFB method had been produced, the next stage was to evaluate its performance. A number of tests were proposed with the objective of demonstrating that the simple bounds did eventually converge to the actual bounds as the size of the receiving patch decreased, and also that they always enclosed the actual bounds. The need to find the actual bounds restricted the tests to reasonably simple geometries.

3.6.1 Defining the Tests

Two tests were defined. In the first, the vertical test, two equally-sized square patches were placed a distance s apart, and s was slowly increased. (See figure 3.13. The shaded patch is the source patch, and the other the receiving patch.) This was supposed to simulate the transfer of light between patches on two opposing surfaces as those patches were subdivided. (The relative size of such patches would remain the same, but their relative distance would increase.) As the distance between the two patches is increased, we expect the variation in the patch factor across the receiving patch to decrease, as the solid angle it subtends at the source is also decreasing.

To find the actual range of light flux for such a situation, we note from figure 3.11 that the maximum patch factor in such a case occurs at the centre of the receiving patch, and the minimum at the corners of the patch. Therefore sampling at a corner and at the centre of the patch gives us our bounds.

[1]See Appendix A.

[2]Assuming that f is positive definite (>=0) which is true here.

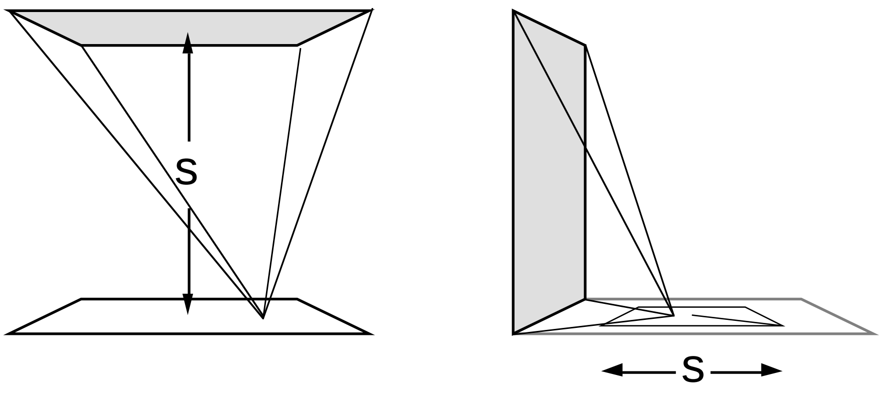


Figure 3.13 Vertical and abutting patch factor tests

The second test, also displayed in Figure 3.13, involves a vertical patch originally sitting on the left hand edge of a horizontal patch. The size and position of the vertical patch are then kept constant while the size of the horizontal patch is decreased. As its size drops to zero, we would expect the simple bounds and the actual bounds to converge to a single value, that being the patch factor in the centre of the horizontal patch.

Again, from figure 3.11 we observe that in such a situation the patch factor at the centre of the left-hand edge of the receiving patch will always be the maximum. Similarly, the minimum occurs at the right-hand corners of the patch.

3.6.2 Test Results

Data for the vertical test was gathered for the range [1,6] at intervals of 0.2 units, and the results can be seen in figure 3.14. It is noticeable that the simple bounds start off being fairly inaccurate, and then converge to the actual bounds as the distance increases. The simple bounds do enclose the actual bounds at all points.

The abutting test data was gathered for the range [0,1] at intervals of 0.02 units, and is graphed in figure 3.15. Both sets of bounds start from the fairly wide variation of the patch factor across the initial patch, and converge towards a central value of 0.1901 as the size of the receiving patch decreases.

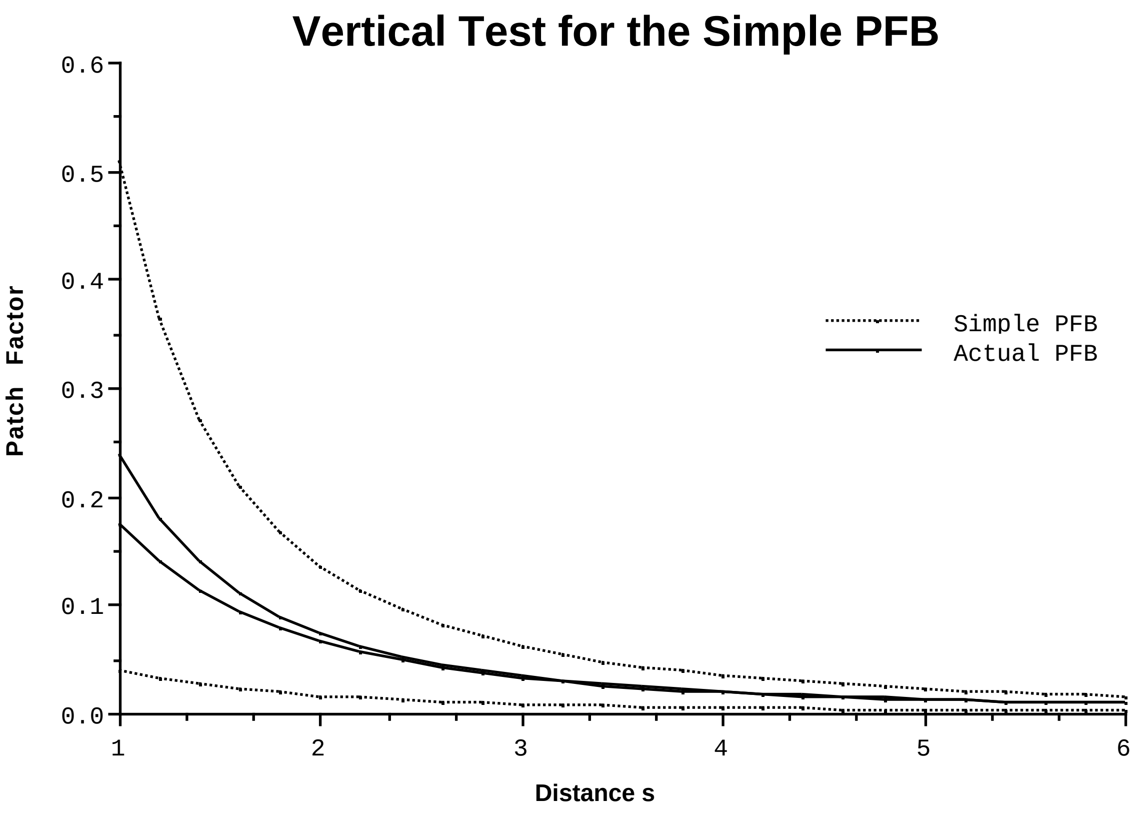


Figure 3.14 Results of the vertical test

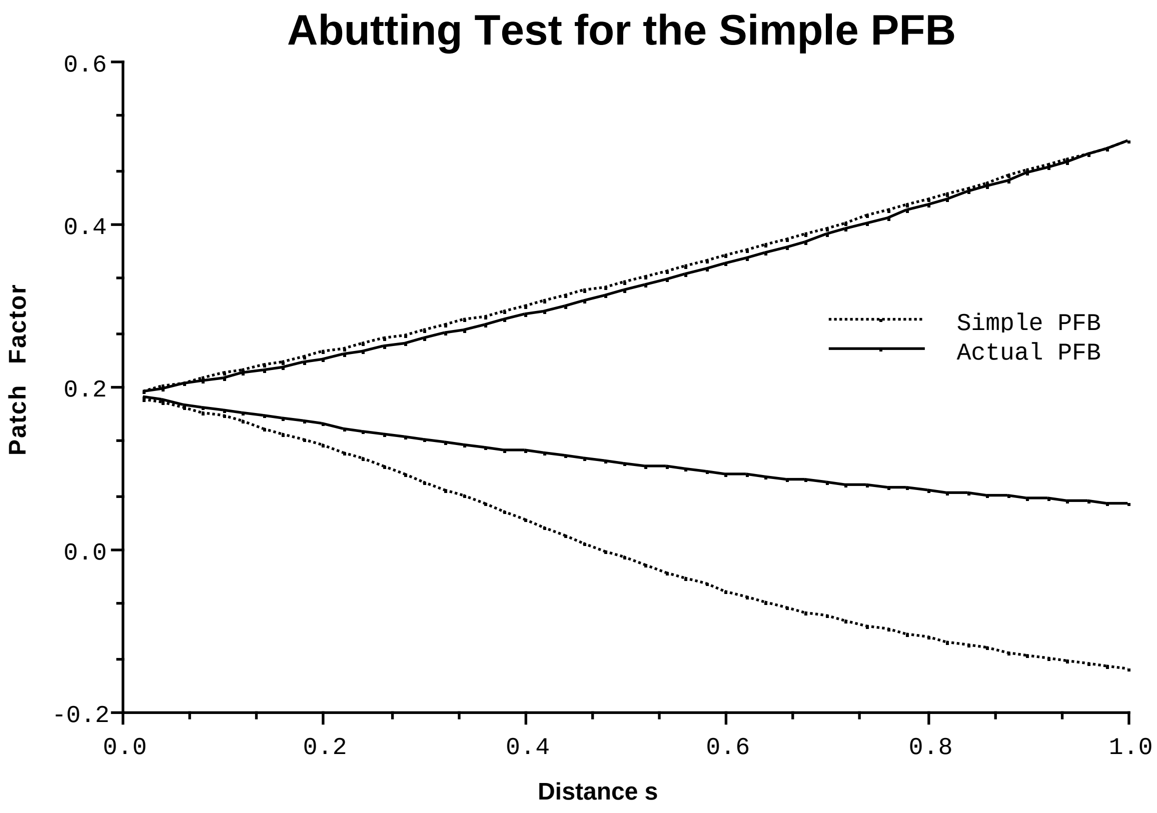


Figure 3.15 Results of the abutting test

3.6.3 Conclusions

At first the simple PFB seemed to work fairly well. However, tests with a preliminary version of the renderer[1] ran into problems towards the end of the rendering process, when much more patch subdivision than seemed necessary occurred. At this point large amounts of subdivision were taking place with almost no consequent increase in accuracy.

The problem is illustrated in figure 3.16, which shows the amount of subdivision necessary to reach a solution accurate to 5% as compared to 10%.

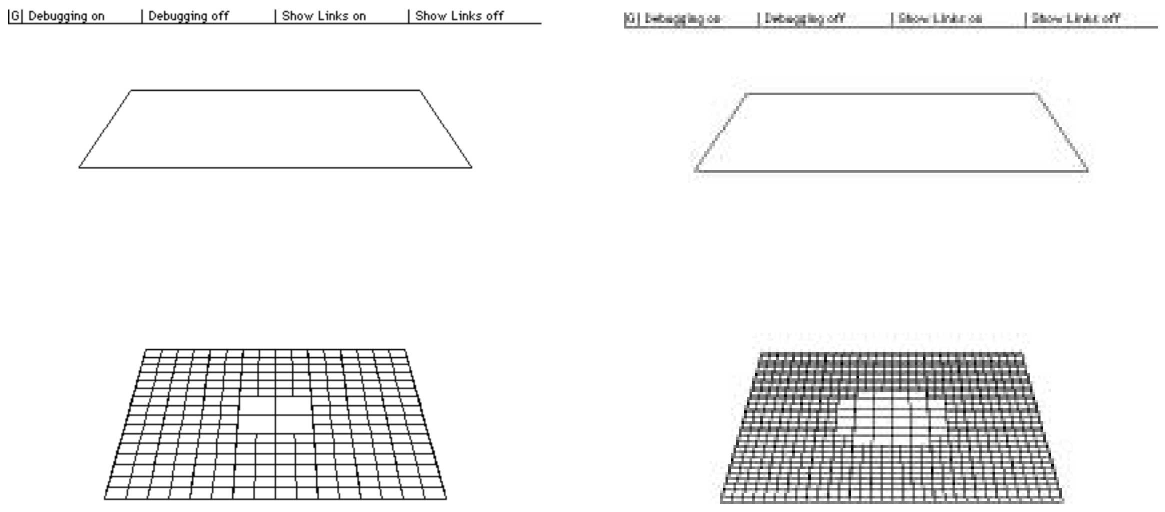


Figure 3.16 Patch subdivision: 10% (left) and 5% (right).

Investigation showed that at high levels of subdivision the simple bounds weren't anywhere near as tight as the actual bounds. This situation is illustrated by figure 3.17, which shows a close-up of the vertical-test data for the range [9,11].

It was concluded that the coherence lost in bounding each edge's area separately was the main cause of this problem. Further tests with the preliminary renderer showed that the problem was worst when the source patch was directly above the receiving patch. In such cases, as the viewpoint moves across the receiving patch, one edge's area term tends to be increasing at almost the exact rate that its opposing edge's term is decreasing. Thus although the variation in the total patch factor is very small, the individual edge terms vary by quite an amount, and bounding them before they are summed to give the patch factor gives far worse bounds than are actually the case.

The final conclusion made was that, as the renderer was unable to complete an adequate amount of the rendering process with the resources available[2], it would be beneficial to spend time developing a more optimised version of the simple PFB.

---

[1]The renderer was developed in parallel with the work done on patch factors, although they are presented separately here. Chapters five and six contain details of the design and implementation of the renderer.

[2]Memory was proving a problem at this time, the renderer having only 4Mb available to it. Later on, this was increased to 17Mb.

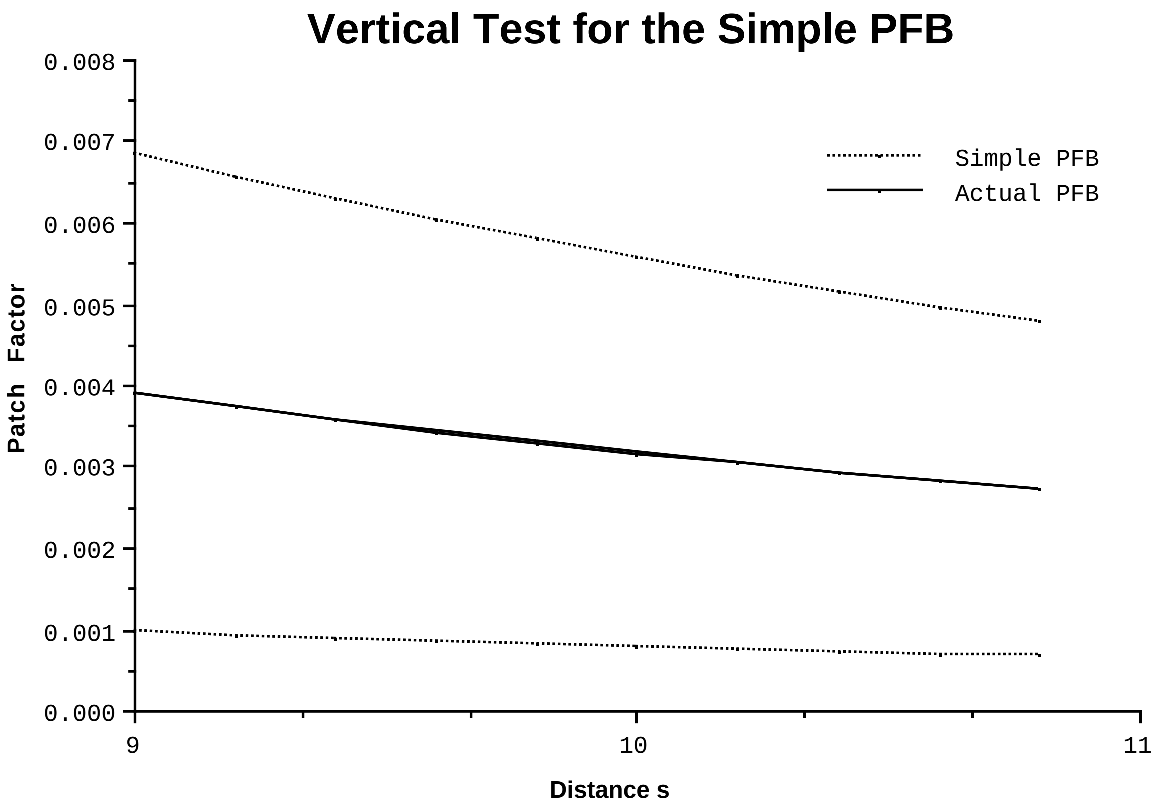


Figure 3.17 A close-up of the vertical test

CHAPTER 4

Towards Better PFB Methods

4.1 Introduction

In Chapter 3 we established what patch factor bounds are, and described the first attempt at producing such bounds, which resulted in the simple PFB method. A major drawback to this method was pointed out; in the first part of this chapter we present a way of testing any PFB method to determine whether this drawback is present.

Having established this, we present two more accurate methods for finding patch factor bounds. Firstly, we develop another analytical method, which produces much tighter bounds than the simple PFB by bounding the edge-term sum, rather than the individual terms. We then present a numerical method for determining these bounds, and profile the cost of this method.

4.2 A Better Method for Comparison

As the size of the receiving patch is decreased, or it is moved farther from the emitting patch, the variation in patch factor across a patch also decreases[1]. If we have some method for estimating bounds on the patch factor, we would like those bounds to at least decrease at the same rate. Indeed, it would be preferable if, as the patch got smaller, our bounds became closer to the actual ones. If neither of these situations is the case then, as the size of a patch gets smaller, our estimate of the patch factor bounds will become worse, making solution of the problem much more expensive than it should be, and possibly preventing it altogether.

In order to test whether this happens, we define the *relative excess* of some estimated bounds E over the corresponding actual bounds A as

$$\text{(4.1)} \qquad Relative\ \ Excess \equiv \frac{E.top - E.bottom}{A.top - A.bottom} - 1.$$

[1]Assuming that there are no obscuring surfaces between the two which may cast shadows.

The relative excess is a measure of the inaccuracy of the estimated bounds when compared with the actual bounds. A relative excess of zero represents the greatest possible accuracy; a negative value would suggest that our estimated bounds are more accurate than the accurate bounds, and thus that the method used to produce them requires substantial reworking! We would like the relative excess to remain constant or decrease as the actual variation in the patch factor decrease.

For the two patch-factor tests of the preceding chapter, we plot the relative excess of the simple PFB, rather than the bounds it produces. The results are shown in figures 4.1 and 4.2.

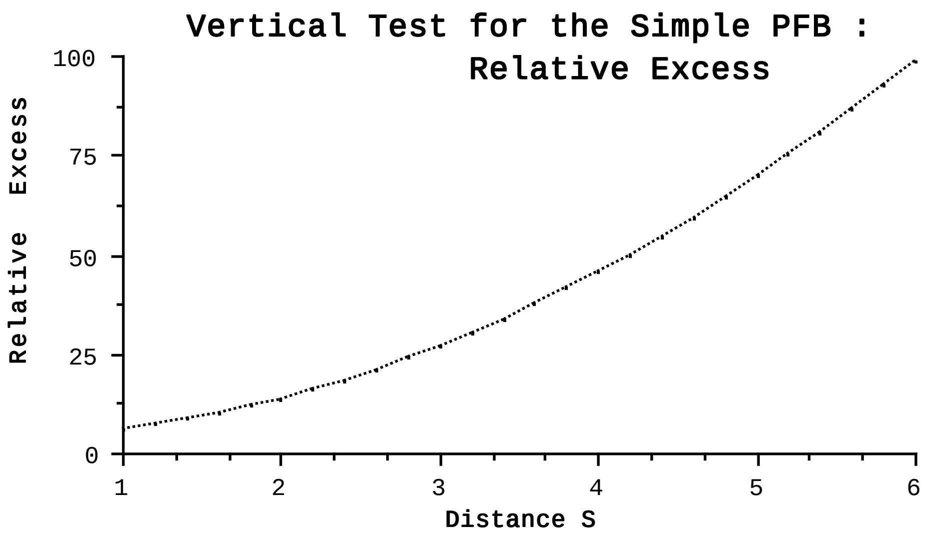


Figure 4.1 Relative Excess for the Vertical Test of the Simple PFB

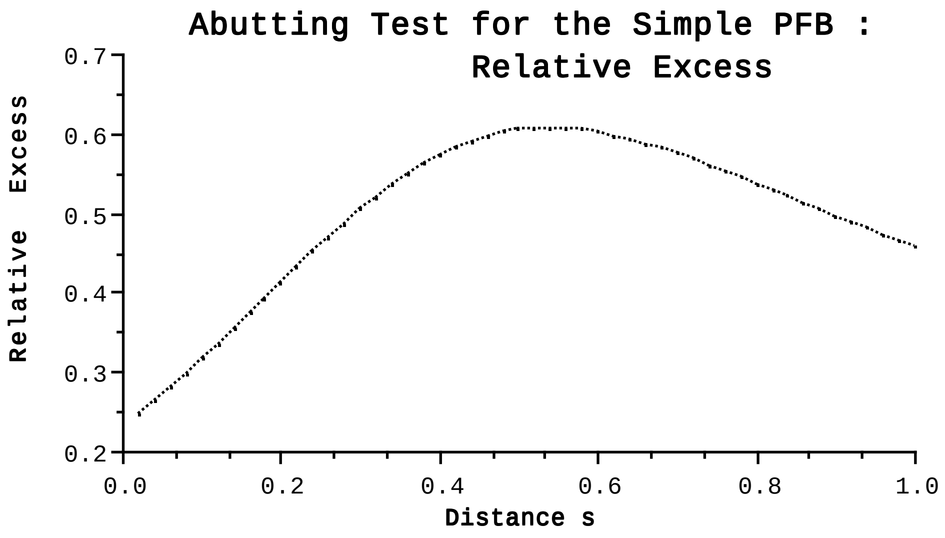


Figure 4.2 Relative Excess for the Abutting Test of the Simple PFB

For the vertical test, the graph confirms that as the patches get farther away, the simple PFB method gets worse in its estimate of the actual bounds. The abutting test, on the other hand, shows acceptable results in that, after an initial rise, the relative excess starts to tail off as s decreases.

4.3 The Linear PFB Method

To overcome this drawback in the Simple PFB, we needed to develop a new method that traded off the variation between edge terms, rather than treating them as independent entities. This trade-off had to be particularly effective when the patch was overhead.

A number of attempts were made to extract common factors from the edge terms, so that the sum of the edge terms could be bounded, rather than the individual terms. This proved to be fruitless, largely due to the arctangent function in the edge term equation. The series expansion for the arctangent function was tried, but its substitution did not solve the problem. Eventually, it was decided that we should proceed with a simpler approach that had suggested itself earlier in the piece, rather than continue searching for common factors in the complete edge term. This approach is presented below.

4.3.1 Ideas behind the Linear PFB

Investigations had revealed that when the source patch was close to being directly above the receiving patch, most of the variation in the edge terms was due to the P part of equation 3.28, which we reproduce below for reference. By contrast, the θ/A part had relatively little variation in such situations, which suggested that better bounds might be had by treating the θ/A part of each edge term as a constant, and looking to extract a common factor from the P parts. This proved rather successful.

$$Edge\ \ Area = \frac{P}{2A}\theta . \tag{4.2}$$

The P term varies linearly with position on the receiving patch. (See equation 3.32.) Thus, the edge term can be regarded as the product of the linear function for P, and the range for θ/A, which gives us a set of *linear* bounds for it. Figure 4.3 provides a two-dimensional example of this.

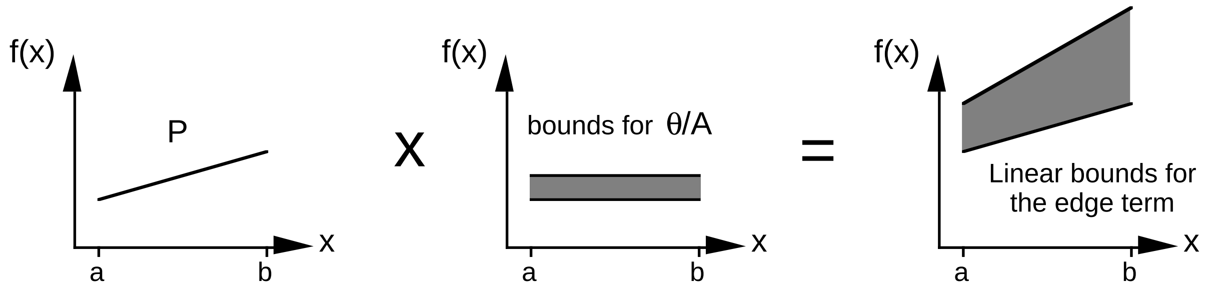


Figure 4.3 Linear Edge Term Bounds

We can then sum these linear bounds to get a linear bound for the patch factor. The advantage of this approach over the simple PFB is that when we add two linear bounds together, we trade-off any opposing slopes. Consider figure 4.4, which shows the linear bounds for an increasing function and a decreasing function being added together; the result is more tightly bounded than the equivalent simply-bounded answer.

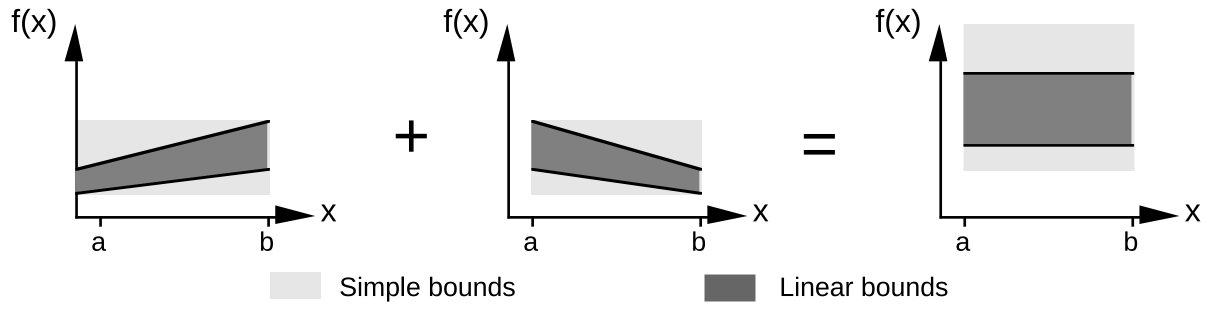


Figure 4.4 Adding linear bounds

We call the patch-factor bounds produced by this method the linear patch-factor bounds. Their derivation is presented in more detail below.

4.3.2 Derivation of the Linear PFB

We start with equation 4.2, and substitute for P using equation 3.32, giving

$$\begin{aligned} Edge\ \ Area &= \left((p \times q).e_z + sa_y - ta_x\right)\left(\frac{\theta}{A}\right) \\ &= \left(m_0 + sm_1 + tm_2\right)\left(\frac{\theta}{A}\right). \end{aligned} \tag{4.3}$$

The patch factor is then the sum of the patch's edge-area terms, so that

$$\begin{aligned} Patch\ \ Factor &= \sum_{i=1}^{n}\left(m_{i0} + sm_{i1} + tm_{i2}\right)\left(\frac{\theta_i}{A_i}\right) \\ &= K_0 + sK_1 + tK_2\ , \\ where\ \ K_j &= \sum_{i=1}^{n} m_{ij}\left(\frac{\theta_i}{A_i}\right). \end{aligned} \tag{4.4}$$

As the distance between the source and receiver patches becomes large, the θ/A term of each edge tends to vary very little[1], and thus the variation in the K_i terms is also small. In such cases the patch factor is bounded tightly by the linear equation above.

To find the K_j terms we use range addition on the $m_{ij}(\theta_i/A_i)$ terms. We then have a linear equation for the patch factor, where each coefficient is a range rather than a constant. We then bound this linear equation in a similar fashion to equation 3.33, so that

$$\begin{aligned} Linear\ \ PFB &= \left[K_0 - KError,\ \ K_0 + KError\right], \\ where\ \ KError &= s_0 abs(K_1) + t_0 abs(K_2). \end{aligned} \tag{4.5}$$

[1]At large distances A can be approximated by the area of a segment, namely $r^2\theta$, and thus θ/A tends to $1/r^2$. If the size of the receiving patch is small compared to the distance between the patches, the variation in r across its surface is not large.

4.3.3 The Abutting Patches Problem II

In section 3.5.3 it was pointed out that the simple PFB suffered from singularities where patches abutted, and that clipping the bounds produced to $\pm R_{max}\theta_{max}$ would fix that problem. Exactly the same problem affects the Linear PFB, in that when calculating the range of θ/A, a singularity will occur when the A factor of the edge area term goes to zero.

This poses a particular problem for the linear PFB because we have made the assumption that we can separate out the P and θ/A terms. However, for any edge we can *escape* from this assumption, because linear bounds are a special case of ordinary bounds. Linear bounds may be represented by A + sB + tC, where A, B and C are ranges; if B and C are set to the range [0,0], then these linear bounds are equivalent to the simple bounds A.

To cater for the singularity problem, then, when calculating the range for θ/A, we check whether clipping is necessary, as in section 3.5.3. If it isn't, we proceed as before. If it is, we instead calculate simple bounds for the edge term P(θ/A), and add these to the overall total. Thus the algorithm for calculating linear patch-factor bounds is[1]:

```
for each edge of the source do
  Calculate ranges for θ/A and P, and the coefficients $m_i$
  if P.(θ/A) requires clipping then
    Clip to find simple bounds S for the edge
    $K_1 = K_1 + E$
  else
    for i = 1 to 3 do $K_i = K_i + p_i(θ/A)$

Find bounds for the linear equation $K_1 + sK_2 + tK_3$
```

4.4 Results for the Linear PFB

4.4.1 Comparison with the Simple PFB

The vertical test that was used to test the simple PFB in section 3.5.3 was applied to the linear PFB. The results are shown in figure 4.5. Obviously, in comparison with the test for the Simple PFB in figure 3.12, the bounds produced are a lot tighter. The graph of the corresponding relative excess of the bounds shown in figure 4.6 shows that these bounds become more accurate as s increases, in contrast to figure 4.1.

[1]An implementation of this can be found in the UPatchFactorRanges listing of Appendix B.

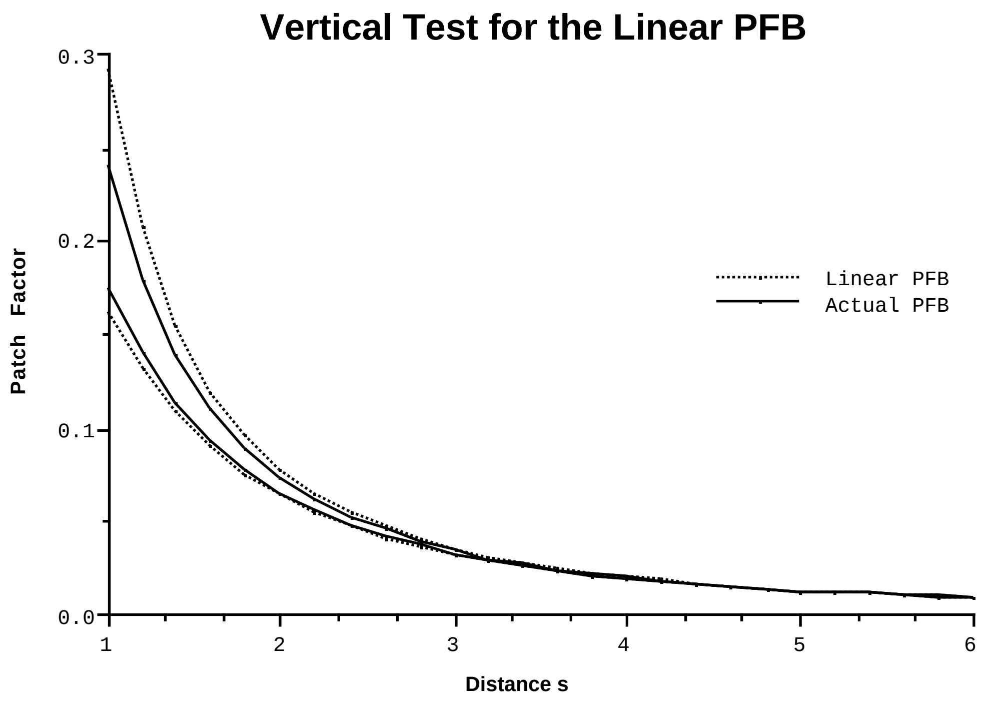


Figure 4.5 Applying the Vertical Test to the Linear Patch Factor Bounds

The abutting test for the Linear PFB was also performed, and found to give identical results to the same test for the Simple PFB. This occurs because three of the edges in such a situation require clipping, and therefore are bounded simply. The sole linearly-bounded edge term has no other term to trade-off against, and so we get the same results as for the simple patch factor bounds.

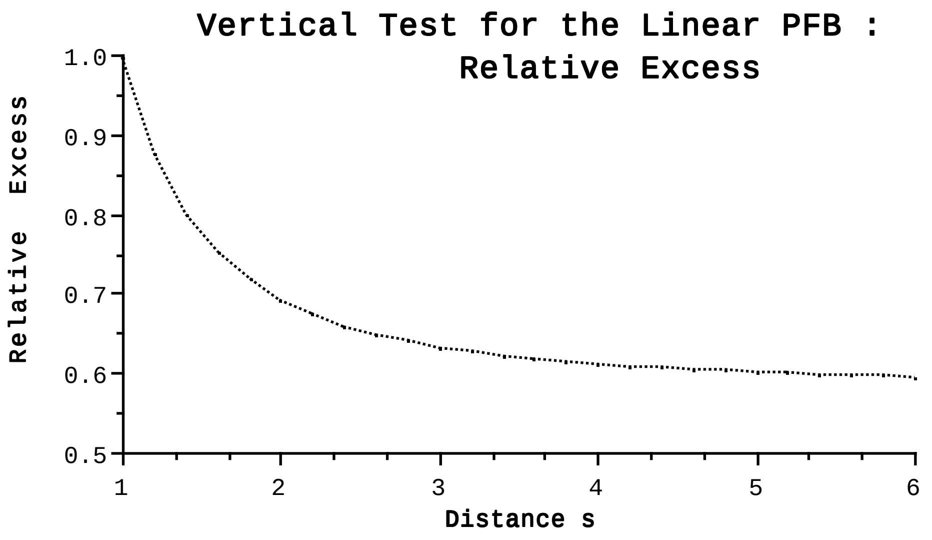


Figure 4.6 Relative Excess for Linear Patch Factor Bounds

4.4.2 A Flaw in the Linear PFB

As a final check on the linear PFB, its results were tested for a number of distant and small source patches at various angles, these being thought most likely to cause problems. Although the linear PFB performed well for most of these patches, it was noticed that patches that were low down on the horizon, and not directly facing the receiving patch, tended to have a high relative excess. Further testing showed that as such patches were moved further away, the relative excess increased, which suggested that a weakness in the linear PFB had been found.

To investigate this problem, a third patch-factor test was developed, called the horizontal test. As in the vertical test, the test starts off with two equally-sized square patches placed a unit apart. Instead of then moving the upper patch vertically away from the lower patch, however, we slide it along horizontally a distance s, as in figure 4.7.

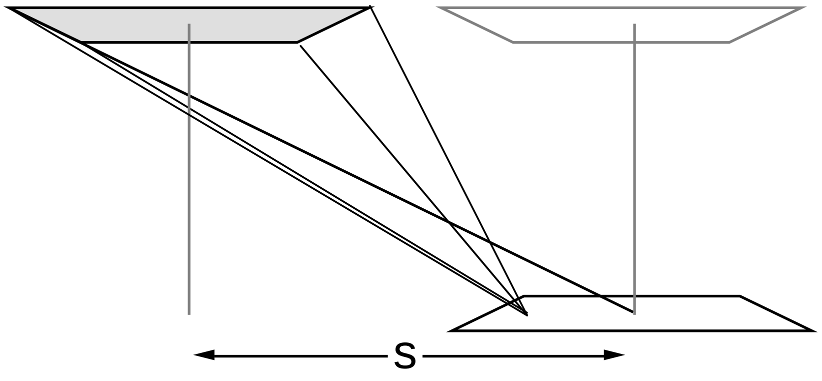


Figure 4.7 The Horizontal PFB Test

Calculating the actual bounds for this test is slightly more complicated than for the previous two. We observe that the maximum patch factor from the upper patch to any point in the plane of the lower patch will always occur at the point directly below its centre, when it is directly overhead. If this point falls inside the lower patch, then we have our maximum. If it falls outside, however, the maximum will occur somewhere along the patch's boundary. In the case of figure 4.7, symmetry will ensure that the maximum falls in the centre of the left-hand edge of the patch. Similarly, the minimum always occurs at both of the right-hand corners of the patch.

Results for the horizontal test were gathered over the range [0,20] at intervals of 0.4 units, and are graphed in figure 4.8. The problem is more clearly illustrated when we look at the relative excess of the method, shown in figure 4.9. As with the vertical test for the simple PFB, the graph shows that as s increases, our estimate of the PFB gets worse, which is undesirable.

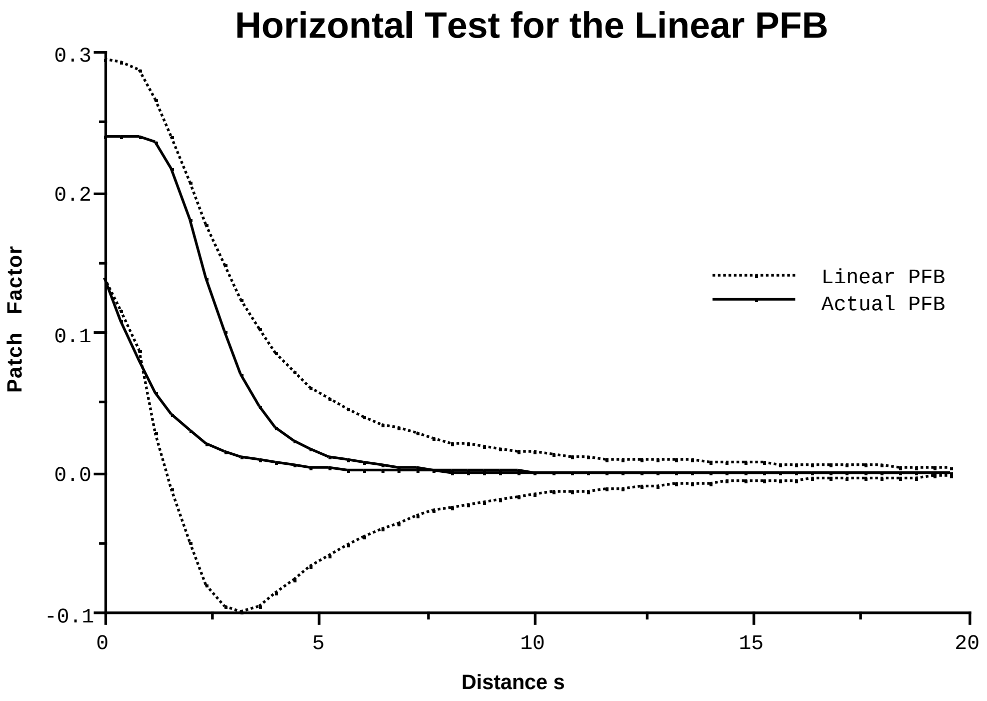


Figure 4.8 Horizontal test for the linear PFB

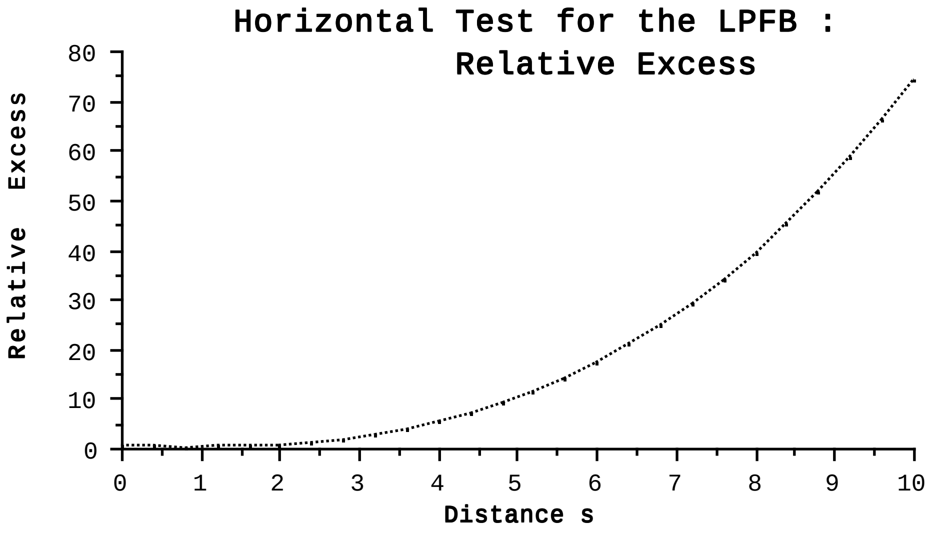


Figure 4.9 Relative Excess for the Horizontal Test

4.4.3 Conclusions

The Linear PFB method certainly proved to be a vast improvement over the simple PFB. It gave bounds that were much closer to the actual bounds, especially as inter-patch distances increased. It has the flaw commented on above, but this was not significant for the scenes we were rendering.

The linear PFB was based around the assumption that θ/A was relatively constant for each edge. For those far-off, close-to-the-horizon edges where it produces non-optimal results, it turns out that it is

P/A, rather than θ/A, that tends towards being constant, approaching the R_{max} of section 3.5.2[1]. To solve this flaw we could try to develop a bounding method that assumed that P/A was constant, and traded off the theta term of the various edges. This method and the linear PFB could then be mixed to produce a hybrid method that would cover both cases well. Unfortunately, theta is not nearly as amenable to analysis as P.

To improve on the linear PFB itself we would need to find a better way of bounding the sum of edge factors. A fair amount of work was put into trying to develop such methods; techniques such as expanding the arctan series and limiting the sum to three or four well-defined edges were tried[2], but without much success. By this time, a good understanding of the way the sum of edge factors worked had been reached, but the major conclusion of this understanding was that there were no "easy outs" in analytical bounding.

A major problem at this point was that we were still uncertain as to exactly how important having accurate bounds for patch factors was to the rendering process. We didn't want to spend too much time developing ever-more-complicated analytical bounding methods if any gains in accuracy weren't going to translate into better performance by the renderer. Balanced against this was the suspicion that accurate bounds were important. In preliminary tests with the renderer, we were still getting nowhere near the accuracy figures we were hoping for, even with simple one light-source, one-patch scenes.

In light of these factors, we decided that, having covered one end of the accuracy-vs-cost spectrum with relatively inexpensive but also inaccurate bounds, we would try to cover the other extreme, by developing a method to calculate patch factor bounds to a high degree of accuracy. Such a method would use numerical search techniques to find the patch factor bounds, and, while it might take many iterations to find the bounds, we could be confident that they would lie within a specified amount of the true bounds. The development of this numerical PFB method is detailed in the rest of this chapter.

4.5 A Numerical PFB Method

In section 3.4 the distribution of the patch factor over a receiving patch was plotted for a couple of source-receiver configurations. One of the reasons for attempting to produce a numerical solution for the PFB was that such plots always seemed to reveal a smooth, well-behaved function, no matter what strange configuration of patches was tried. Indeed, observation of a large number of such plots led us to make several assumptions about patch-factor distributions that made searching for bounds on them a much simpler task. These assumptions are detailed in the next section.

[1]If the distance of the edge from the plane of the patch is small compared to its distance from the viewpoint, then its projected area onto that plane, P, will approach the actual area A.

[2]For instance, constraining the source patch to being a square. It was felt that some of the difficulty we were having with the patch-factor sum was due to the fact that it was an arbitrary sum, which allowed any possible polygon as a light source. For reasons explained in 4.5.1, we believed that constraining the source to be a convex polygon might make bounding easier.

Another reason was that, although a numerical solution might initially require a lot of work, that work could be reused when patches subdivided. If we know the point where the maximum occurs in a patch, for instance, then when we subdivide that patch we automatically know the maximum of the subdivision that contains that point.

Finally, it was reasoned that even if the numerical PFB proved prohibitively expensive in terms of computation time, it could be used in the renderer to indicate how much better the system would perform with accurate bounds, in order to gauge the effect that improving the accuracy of any other bounding methods could have.

4.5.1 Assumptions Made

Our fundamental assumption is that the patch-factor distribution has a single maximum, and contains no *valleys*. This is equivalent to requiring any cross-section of the distribution to have a shape similar to that of the function in figure 4.10, which has a single maximum, and tails off towards zero on either side of that maximum.

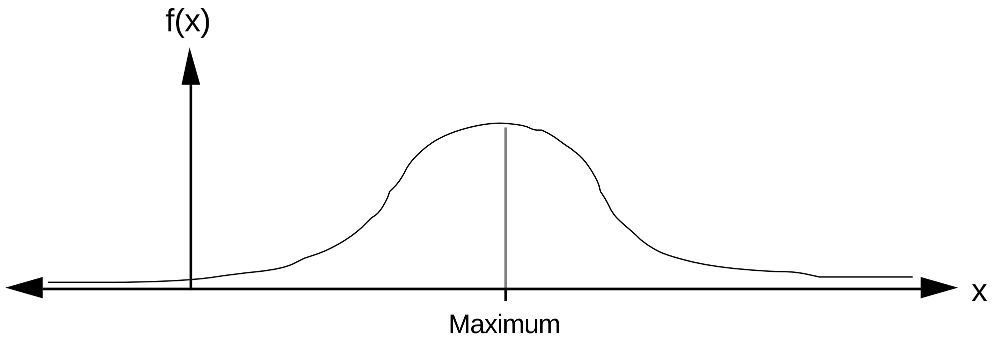


Figure 4.10 Assumed Cross-Sectional Distribution

The two most important results of this assumption are that:

- As the function only has one maximum, we can use a numerical method for finding local maxima, without having to worry about whether any maximum found is actually the global maximum.
- The minimum of the function over a convex polygonal region will always occur at one of its vertices. (If this was not the case, it would violate our single cross-sectional maximum assumption.)

We believe that this assumption holds for any planar, convex source, although it has not been possible to construct a convincing proof in the time available. A number of test cases were produced, and contour plots of the resulting patch factor distributions were used to check whether the assumption had been violated; in all of these cases the assumption held. The test case that was held to be most likely to disprove the assumption is shown in figure 4.11. It was felt that a patch that had one edge that was close to the receiving patch, yet small in width, and had the opposing edge farther from the ground, yet much wider, might result in a valley in the patch factor distribution somewhere between these edges.

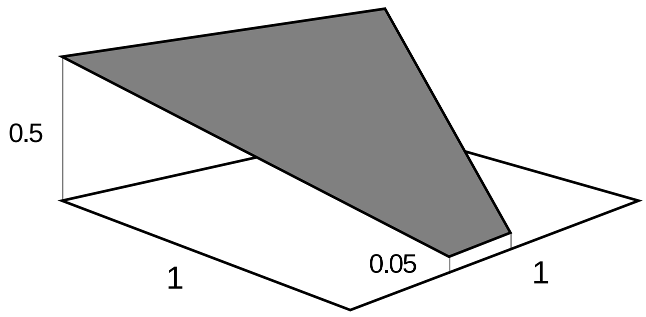


Figure 4.11 A test patch configuration

A plot of the patch factor distribution for this configuration, as well as a contour plot, is shown in figure 4.12. As can be seen, all of the contour curves in the plot are convex, and thus any line drawn through the plot cannot contain a local minimum.

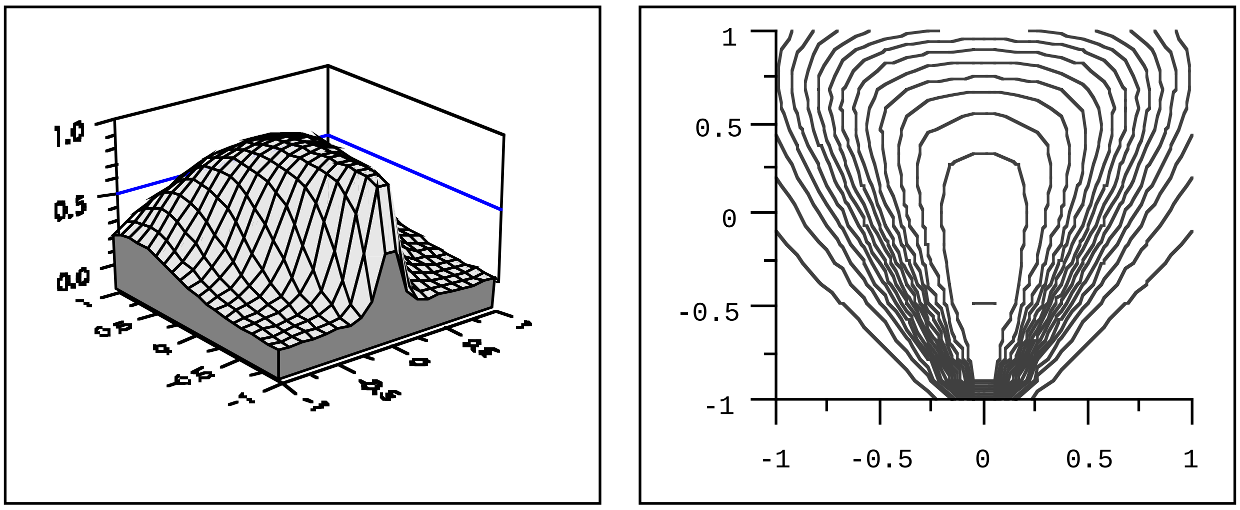


Figure 4.12 Surface and contour patch-factor plots for figure 4.11

As an example of a source patch that does violate our assumptions, consider the concave patch shown in figure 4.13.

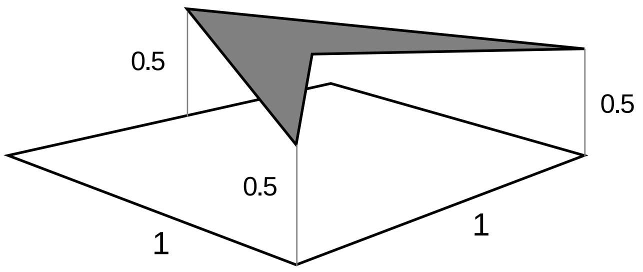


Figure 4.13 A concave source patch

The patch-factor distribution for this configuration is plotted in figure 4.14. We can see that the surface plot has valley running from the maximum though the middle of its front edge. Also, the contour lines of the contour plot are concave, and hence it is possible to draw a line through the plot that contains a local minimum.

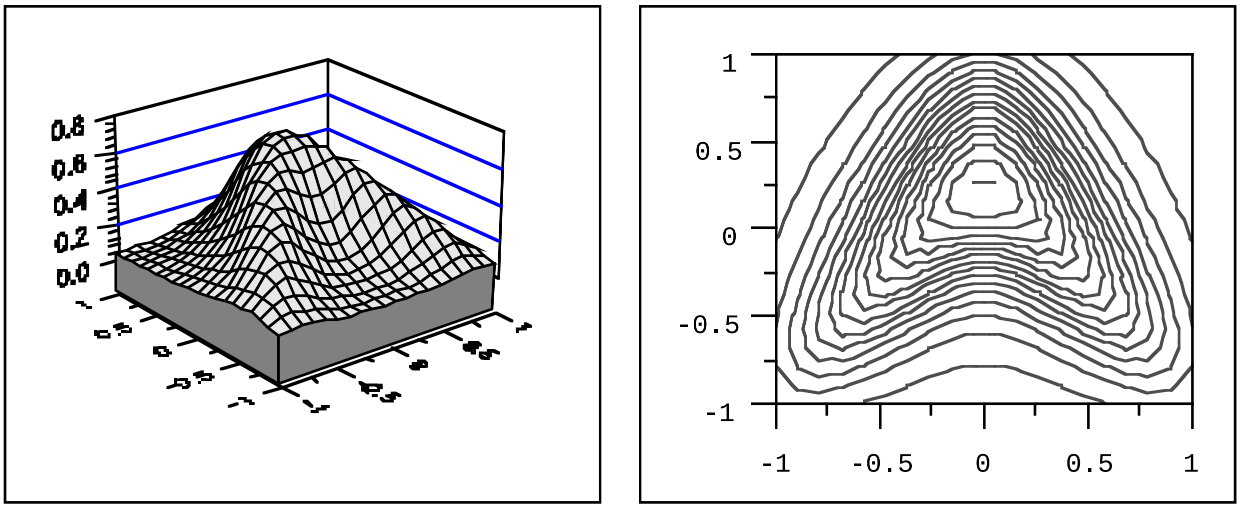


Figure 4.14 Surface and contour patch-factor plots for figure 4.13.

Having made these assumptions, we proceeded to construct an algorithm for finding the maximum patch factor over a patch by searching for it numerically.

4.5.2 A Hill-climbing Algorithm for Maximum Search

We consider again the patch coordinate system shown in figure 3.12. If we find the derivative of the patch factor P(s,t) with respect to s and t, then, at any point in the plane of the patch, the vector [dP/ds, dP/dt] will point in the direction of greatest increase of the patch factor. Thus if we take a small step in this direction, we should be moving towards the maximum of the function. (This is much like climbing a hill; if we want to get to the top quickly, we always climb in the direction of the steepest ascent.) If we keep repeating this process, we should eventually reach the maximum of the function, as in figure 4.15.

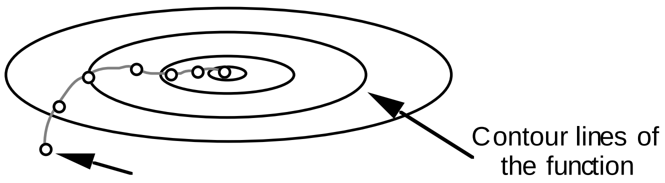


Figure 4.15 An example of hill-climbing

In the development of any algorithm to carry out this hill climbing, there are three important considerations:

- The size of the step to take at each iteration. The smaller the step size, the greater the number of steps that are required to reach the maximum. If the step size is too large, however, we will continually overshoot the maximum, bouncing from side to side without getting much closer to it, in a process known as hem-stitching. Ideally, we would like the step size to start off being large, and decrease as we get closer to the summit of our ‘hill’.

- Deciding whether any new position generated should be accepted. If we step too far in the direction of the gradient, we may end up stepping over the maximum, and arrive at a point that is lower than the point we started from. In such a case we can reject this new position, reduce the step size, and try again.
- Deciding when we are close enough to the maximum to stop our search. The obvious indicator for this is the size of the gradient, as this tends to zero as we reach the maximum. However, the gradient can also be close to zero at large distances from the hill we are climbing. Consider the far left and far right of the function shown in figure 4.10, for instance. We must therefore check that the gradient is decreasing between steps before terminating our search, in order to ensure that it is the maximum of the function that we are close to, rather than the minimum.

Bearing all of this in mind, the hill-climbing algorithm we have constructed is as follows:[1]

```
new_position = position + t*gradient      { Move a step in the direction of the }
                                                { gradient }
evaluate new_f and new_gradient

if (new_f < f) then                       { If new point is not acceptable }
  t = t / 2                               { Decrease step size and try again }
else
  if gradient has reversed direction then       { If we've passed over the maximum }
    t = t / 2                                   { Decrease step size for next time }
  else
    t = t * 2                                   { Otherwise, increase step size }
    if curve is decreasing and error < epsilon then   { And check for termination }
      we're finished: exit
  f = new_f; gradient = new_gradient; position = new_position
```

The variable t represents the current step size, and is initialised to some fraction of the patch size. If the new position gets rejected, this step size is decreased, and we start over. If it is accepted, we alter the step size as necessary, check for termination, and then move on to the new position.

If at some point the gradient reverses direction, we assume that we have passed over the maximum, and decrease the step size. This is done in order to avoid the hem-stitching problem mentioned above. If oscillations set in and we continue to pass over the maximum with each step, the step size will be halved with each iteration, until it is small enough for such oscillations to stop. At this point we proceed towards the maximum from one side.

If the gradient hasn’t reversed direction, we increase the step size, so that if it was initially too small for fast convergence, it will eventually reach a reasonable level after a number of iterations. We originally planned to include a delay count, so that this increase only took place after a fixed number

[1]Much of this algorithm is adapted from the *modified secant method* presented in [McCo82].

of iterations, the reasoning being that we would not want to increase the step size if it was already large enough for reasonable convergence. However, this delay was found to be unnecessary, probably because as we move closer to the maximum, the length of the gradient decreases, and thus any increase in step size is only compensating for this. Also, the sole consequence of having a step size that is too large is that we end up jumping over the maximum, in which case the step size will be cut back to what it was before.

To test whether the curve is decreasing, we check whether the current point is lower than the tangent plane at the new position we have generated, as in figure 4.16. If this is the case, we assume we are on a part of the function where the gradient is decreasing.

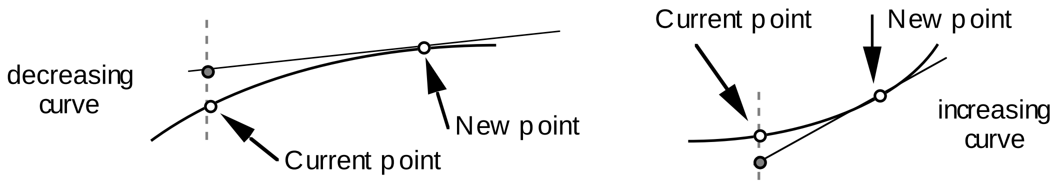


Figure 4.16 Checking whether the function is decreasing.

If the curve actually is decreasing, we can be sure that the tangent plane at the new position will also pass above the maximum of the function. We can thus use this tangent plane to get an upper bound on the maximum. For a square patch of width *w*, for instance, the farthest possible distance we could travel from the new point before running off the edge of the patch is the length of one of its diagonals[1], i.e. $w\sqrt{2}$. Thus an upper bound for the tangent plane over the patch, and hence the maximum of the function, is:

$$\begin{aligned} Bound &= new\_f + error, \ where \\ error &= w\sqrt{2}\ \|new\_gradient\|. \end{aligned} \tag{4.6}$$

Moreover, because we know that the maximum lies in the range [*new_f*, *new_f* + *error*], we can use this range as a measure of how close we are to the maximum. In the current implementation of the maximum-searching algorithm, we require the relative error of this range to be less than a certain fraction of the width of the patch, so that for smaller patches we get more accurate answers. Thus the termination test for the algorithm is

$$\frac{\sqrt{2}\ \|new\_gradient\|}{new\_f} < k, \tag{4.7}$$

where *k* is a small number. If the curve is decreasing, and the above test is true, then the upper bound for the maximum, *new_f* + *error,* is returned.

[1]We could do better than this by considering the position of the new point and calculating exactly the greatest distance from it to the border of the patch. However, the improvement this exact answer would give almost certainly doesn't warrant the work involved.

4.5.3 Clipping and the Seed Point

The above description glosses over a few points. For a start, the maximum of the patch factor distribution may fall outside the patch we are interested in. In such a case, the maximum patch factor on the receiving patch will lie somewhere along one of its borders. To handle such a possibility, we do the following:

- If at any time a step takes us outside an edge of the patch, we clip the point generated back to that edge.
- If the point is on one of the edges of the patch boundary, and a component of the gradient points over that edge, then we clip that component to zero. This prevents any steps taken from that point from going over the edge, and also stops that component from figuring in the termination test.

These two actions ensure that if the maximum lies outside the patch, we will end up searching along the edge closest to it, in order to find the largest patch factor that belongs to the patch.

Another consideration is the choice of the *seed* point from which to start the hill-climbing algorithm. We recall that the minimum patch factor over the patch is guaranteed to fall at one of its corners. We can also use the value of the patch factor function at these corners to determine the seed point, as follows:

- Sample the patch factor function at each corner of the patch.
- Take the lowest sample as our minimum for the PFB
- Take the corner which had the highest sample as the seed point.
- If two corners both have the maximum sample value, we take the point halfway between them as the seed point.
- If all corners have the same value, we take the centre of the patch as the seed point.
- Call the hill-climbing algorithm with the seed point to find the maximum for the PFB.

4.5.4 Calculating the Derivative

Calculating the derivative of the edge term function was avoided for a long time because of its apparent difficulty. Not only is the edge equation a reasonably complex function of *s* and *t*, but any differentiation of such a function would seemingly produce to an even more complex function. In the end, however, it turned out that if several parts of the equation were encapsulated as functions, the differentiation was quite straightforward, and the result comprehensible(!).

In the spirit of section 3.5.2, we represent the edge-term equation as a composite function of the form

(4.8) $$E = \frac{e}{f} Arctan\left(\frac{f}{g}\right).$$

Matching with equation 3.28 gives

$$\begin{aligned} e &= e_z.(p \times q), \\ f &= \|p \times q\|, \\ g &= p.q. \end{aligned}$$

Differentiating this composite function gives the following:

(4.9) $$\begin{aligned} \mathrm{E}' &= \frac{\mathrm{e}}{2\pi \mathrm{f}} \mathrm{Arc\,tan}\left(\frac{\mathrm{f}}{\mathrm{g}}\right)' \\ &= \left[\frac{\mathrm{e}'}{\mathrm{e}} - \frac{\mathrm{f}'}{\mathrm{f}}\right] \frac{\mathrm{e}}{2\pi \mathrm{f}} \mathrm{Arc\,tan}\left(\frac{\mathrm{f}}{\mathrm{g}}\right) + \frac{\mathrm{e}}{2\pi}\left[\frac{\mathrm{g}\frac{\mathrm{f}'}{\mathrm{f}} - \mathrm{g}'}{\mathrm{f}^2 + \mathrm{g}^2}\right] \\ &= \mathrm{d}_0 \mathrm{E} + \mathrm{d}_1 \end{aligned}$$

From the relations in 3.5.1, we differentiate with respect to *s* and *t* to find *e'*, *f'* and *g'* at (0,0):

$$\frac{de}{ds} = a_y \qquad \frac{de}{dt} = -a_x$$

$$f\frac{df}{ds} = \left(q_x\left(p.q - p^2\right) + p_x\left(p.q - q^2\right)\right) \qquad f\frac{df}{dt} = \left(q_y\left(p.q - p^2\right) + p_y\left(p.q - q^2\right)\right)$$

$$\frac{dg}{ds} = p_x + q_x \qquad \frac{dg}{dt} = p_y + q_y$$

If we have calculated the patch factor at some point, we will already have had to calculate*E*, *e*,*f* and *g* for each edge. We can combine these with the expressions for e', f' and g' above to in turn work out d and d for each edge, and thus *dE/ds* and *dE/dt*. Summing *dE/ds* for all edges will give the overall derivative for the patch factor *dP/ds*, and likewise *dE/dt* gives *dP/dt*[1].

It is interesting to note that the extra work required to obtain the derivative from the patch factor is not that great, and indeed doesn’t even require another arctan evaluation. Also, the quantities above are identical to those required in the evaluation of the simple PFB for a patch. As a rough guide, calculating the patch factor and its gradient at a point involves about half as much work as calculating the simple PFB.

4.6 Results for the Numerical PFB

The tests used to test the simple and linear PFB have an unfortunate drawback in the case of the numerical PFB, in that the seed-point-picking procedure outlined above will always pick the

[1]The UEdgeRanges unit listed in Appendix B demonstrates an implementation of this.

maximum correctly, so that no search is needed. To overcome this, this procedure was temporarily disabled, and the hill-climbing algorithm always started from a corner of the patch. All of the tests were run, mainly to confirm that the algorithm was robust, which proved the case. Graphing the results would be pointless, as to the eye the bounds produced by the numerical PFB are identical to the actual bounds. It should be noted, though, that in all cases the relative excess of the numerical PFB was below 0.1%.

4.6.1 Cost of Evaluation

One factor that does need investigation, however, is the cost of finding the bounds numerically for different source-receiver patch configurations. This cost was measured by counting the number of calls to the routine that finds the patch factor and its gradient at a point, as this is the routine that does most of the work. The vertical, abutting and horizontal tests were then run to profile the cost of the numerical PFB.

The cost of the numerical PFB for the abutting test is shown in figure 4.17

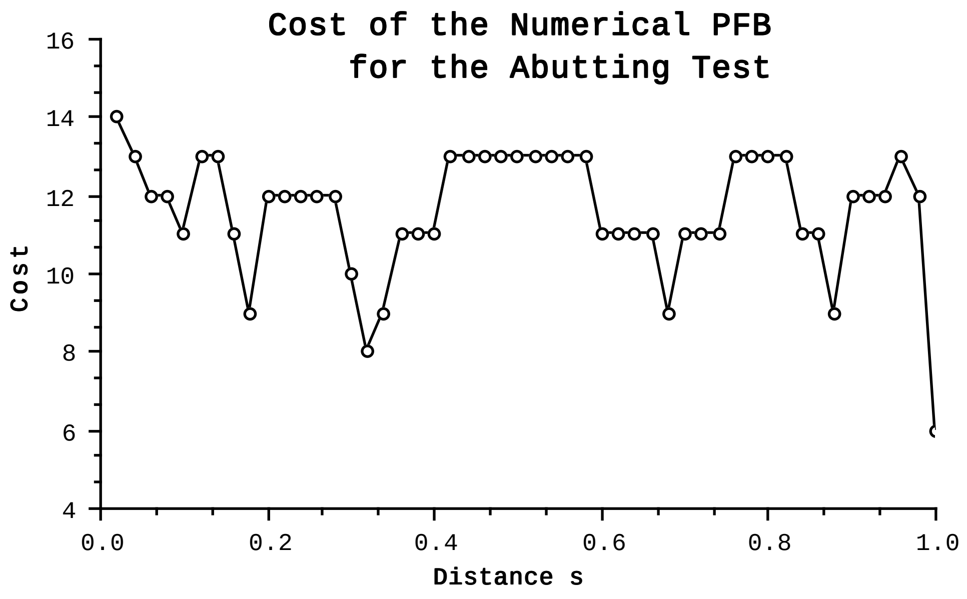


Figure 4.17 Cost of evaluation for the Abutting test

The notable feature here is that the cost is largely static for different values of s. As s decreases, the distance that algorithm has to cover to find the maximum also decreases, and hence we would expect the number of iterations required to find the maximum should also decrease. However, we recall that we made the accuracy required of the hill-climbing algorithm dependent on the size of the patch, so as s decreases, more accurate answers are required. These two factors tend to cancel out, resulting in the graph of figure 4.17.

The horizontal and vertical cost graphs turned out to be very similar, with the cost of the numerical PFB increasing slowly as s increased. The horizontal cost graph is shown in figure 4.18.

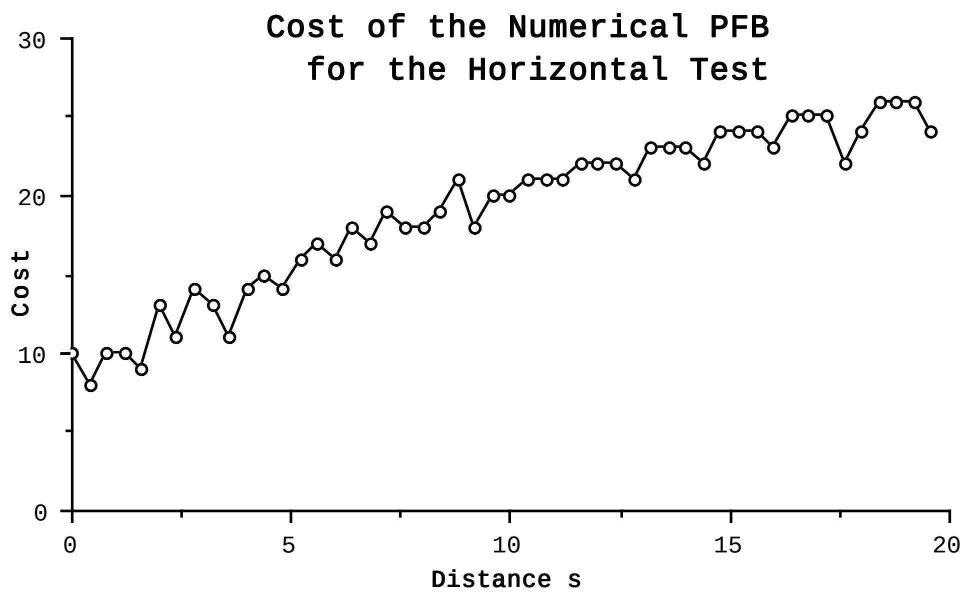


Figure 4.18 Cost of evaluation for the Horizontal test

The cost increase occurs because as the distance between the source and receiving patches increases, the height of the patch factor distribution decreases, and hence the gradient at any point also tends to decrease. As a result of this, the step size required to quickly reach the maximum is larger, and it takes a while for the algorithm to increase the step size to a size large enough for reasonable convergence.

4.6.2 Conclusions

The major conclusion about the numerical PFB method is that it has proved more successful than was perhaps expected. In all the test cases we have tried it has proved robust, contrary to earlier fears that for some situations the hill-climbing algorithm might take a long time to converge. A reasonable metric for indicating the accuracy of the upper bound has been formulated, which has been successful in ensuring a high degree of accuracy in the values returned for the maximum. Perhaps most importantly, the cost of evaluation hasn’t proved prohibitive, being around twenty iterations for most of the source-receiver patch combinations that have been tried.

The current hill-climbing algorithm could be improved in several ways to provide even faster convergence. The most obvious of way of doing this is to find a method for calculating the second derivatives (i.e., the Hessian) of the patch factor, which would allow us to use a variation of the Newton-Raphson search. Another possibility is to generate these second derivatives by using difference methods on the first derivatives of the patch factor. Such approaches could well be unnecessary, however, as with a decent coherence scheme, the improvement in performance they deliver may well be negligible.

Having investigated the issues of patch factor bounds, and developed several methods for calculating these bounds, we turn our attention to the design of the renderer, which is covered in the next chapter of this thesis.

CHAPTER 5

The Design of the Renderer

5.1 Introduction

In this chapter we present the design of the renderer that has been constructed during the course of this thesis. We start by looking at the design of the patch object, and the components that allow it to keep track of the other patches of the scene, and calculate the light it emits towards those patches.

We then look at the eyeball object, and consider how it differs from the patch object. The issues involved in the eyeball's perception of error are established, and a way of modelling this error is presented.

Finally, we introduce the world object, which provides a mechanism for initialising the renderer.

5.2 The Patch Object

5.2.1 Overview

Our first task in the design of the fuzzy renderer was to produce an outline for a generic patch object; one that could be used for any kind of fuzzy rendering, and wasn't dependent on any assumptions. We could then expand on this outline to produce a full design, taking into account any assumptions that were going to be made about patch objects.

The fields and methods of this generic patch object are shown on the next page[1].

```
CPatchObject = object
  ContributingPatches : list of CContributingPatchInfo;

  Position      : CPositionalInfo;          { Internal Information about this patch }
  Reflectance   : CReflectanceInfo;
```

[1]In the pseudo-code presented in this chapter and the actual code of the renderer we have followed the convention of prefixing abstractions by a capital letter, in order to indicate the type of abstraction. The only prefix used in the pseudo-code is 'C' for class; see Appendix B for the others.

```
      InternalLight : CLightDistribution;
      CurrentAnswer : CLightDistributionRange;

      Children : list of CPatchObject;        { Subdivisions of this patch }

      procedure Subdivide;                    { Methods of the patch }
      procedure RecalculateAnswer;
      procedure FindLargestErrorContributor;
      function  PatchFactor(sourcePatch : CPatchObject) : CRange;

      function  Answer(askingPatch : CPatchObject;
                       var subdivisions : list of CPatchObject) : CAnswer;
    end;
```

All the methods or fields of the patch object are private[1], apart from the *Position* field (which other patches need to access in order to know where this patch is), and the *Answer* method. A short summary of the purpose of each field or method follows:

- The *Reflectance* field contains information about how the patch reflects light, and the *InternalLight* field contains information about the light that the patch emits itself.
- The *ContributingPatches* list is used by the object to keep track of all the other patches in the scene which contribute light to it. Each entry in the list contains information known about a particular patch, such as the last answer it gave, and the bounds on the patch factor from that patch to this patch object.
- The *RecalculateAnswer* method uses this information, as well as the *Reflectance* and *InternalLight* of the patch, to calculate the *CurrentAnswer* of the patch.
- The *FindLargestErrorContributor* method uses the list to decide what the largest source of error in the current answer is. It may decide that it is the answer of one of the contributing patches, or it may decide that the patch's size is the greatest factor, and thus that it should subdivide to get a better answer.
- The *PatchFactor* method calculates the patch-factor bounds from any source patch to this patch object, taking into account any obscuring surfaces between the two.
- The *Subdivide* method creates a number of subdivisions of the patch object, and stores a list of references to them in the *Children* field. (This field contains an empty list if the patch hasn't had to subdivide yet.)
- Finally, the *Answer* method, when called by another patch, uses *FindLargestErrorContributor* to determine how it should try to improve the current answer. If the patch's size is deemed to be the largest cause of error, the *Subdivide* method is called, and the children that it produces are returned to the caller via the subdivisions parameter. Otherwise, the *Answer* method of the offending patch is called to improve its answer. Once this has been done, the information in the patch's entry is updated, and the *RecalculateAnswer* method is called; the new and improved *CurrentAnswer* is then returned to the caller.

[1]I.e., cannot be accessed except from within the object's methods.

The following sections expand on the above outline.

5.2.2 Internal Patch Information

For the “test bed” version of the renderer, we assume that patches are polygons, the position of the patch can be represented by a list of the world-space coordinates for each the patch’s vertices.

We also assume diffuse surfaces, and monochromatic light. Thus to represent the distribution of light emitted by a patch we only need to store a single value, its radiosity, as diffuse emission is non-directional. For similar reasons, we don’t have to store the reflectance properties of the patch in a directional way. A single reflectance value, representing the fraction of the irradiance on the patch that is radiated back into space, will be sufficient.

Bearing this in mind, we can make the following definitions, where TReal is the type of a real number:

```
CRange = record
    top, bottom : CReal;
  end;

CPositionalInfo = list of CCoordinate;
CReflectance = CReal;
CLightDistribution = CReal;
CLightDistributionRange = CRange;
```

While these definitions may seem trivial, they do help keep clear what would need to be changed in the patch object if more complex types of patch reflectance were to be implemented.

5.2.3 The Contributing Patches List

The design of the contributing-patch list is straightforward. Each entry in the list contains the following fields:

```
CContributingPatchInfo = record
    Patch : CPatchReference;
    ItsPatchFactor : CRange;
    ItsLastAnswer  : CRange;

    ItsRadiosityContribution : CRange;
    AnswerError : CReal;
    PatchFactorError : CReal;

    AnswerNotImproving : boolean;
  end;
```

The first field holds a reference to a contributing patch, and the next two are used to cache the last answer we got from that patch, and the bounds on the patch factor that we associate with it. The *ItsRadiosityContribution* field contains the contribution that it makes to the *CurrentAnswer* of the patch object, and can be calculated as:

```
RadiosityContribution = Reflectance x ItsLastAnswer x ItsPatchFactor;
```

Given this, the *RecalculateAnswer* method is simply:

```
CurrentAnswer = Range(InternalLight, InternalLight);
for each contributing patch do
  CurrentAnswer = CurrentAnswer + RadiosityContribution;
```

To help keep track of the causes of error in this patch's contribution to the answer, we work out how much of the error in the *RadiosityContribution* is due to the contributing patch's answer, and how much is due to the patch factor from that patch. This can be done as follows:

```
AnswerError = Reflectance x RangeError(ItsLastAnswer) x RangeEstimate(PatchFactor);
PatchFactorError = Reflectance x RangeEstimate(ItsLastAnswer)
                                   x RangeError(PatchFactor);
```

The RangeError function returns the difference between the two bounds (top-bottom), and the RangeEstimate function the average of the two bounds. For convenience, we introduce a new method *UpdateContributingPatchInfo*, which will recalculate these fields and the *RadiosityContribution* field whenever *ItsLastAnswer* or *ItsPatchFactor* are changed.

Finally, the *AnswerNotImproving* flag is used for loop prevention, and is discussed in section 5.2.3.

5.2.4 Subdivision

When the subdivide method is called, it follows the algorithm:

```
procedure Subdivide;

Create Subdivisions;    { Apart from having different positions and sizes, these }
                               { subdivisions are just clones of the parent patch }
Children = list of subdivisions;
for each child do
  with ContributingPatches do
    for each contributing patch record do
      Recalculate ItsPatchFactor;
      UpdateContributingPatchInfo;
  RecalculateAnswer;
end;
```

Each subdivision inherits its parent's contributing-patch list, as this represents all the information that has been built up by its parent about the light incident on it. We keep the answers from the contributing patches as they are still applicable to the subdivision. However, we do recalculate the patch factors from those patches, in order to take advantage of the increased accuracy we can get due to the smaller size of the subdivision. We also recalculate its answer so that it incorporates this increased accuracy.

5.2.5 The Patch Factor Method

As we assume that we can ignore the visible-surface problem, the algorithm for the *PatchFactor* method becomes largely trivial. All we need to do is transform the position of the source patch into the coordinate system of the viewing patch, and then call one of the simple, linear or numerical PFB functions to calculate the patch factor bounds. Thus we have:

```
function PatchFactor(sourcePatch : CPatchObject) : CRange;

  localPosition = ApplyTransform(ViewTransform(self.Position), sourcePatch.Position);
  bounds = LinearPFB(localPosition);

  return bounds;
end;
```

5.2.6 Finding the Largest Error Contributor

We can find the patch that most effects the error in the current answer by traversing the contributing patches list to find the entry with the largest *AnswerError*. At the same time, we can calculate the error in the answer due to the size of the patch object by adding up the*PatchFactorError* fields.

We then need to decide which of these two sources of error is the largest. We cannot compare them directly, because the action resulting from either choice will involve different amounts of work, and have a different potential for improving the answer. Indeed, as subdivision typically involves the recalculation of a large number of patch factors, we tend to favour asking another patch over subdivision unless it is absolutely necessary.

Instead of using a direct comparison, then, we use the weighted comparison

$$\text{ProblemIsOurSize} = (\text{TotalPatchFactorError} > \text{alpha} \times \text{LargestContributorError})$$

to make our decision. The alpha constant is our weighting factor; the larger we make it, the more we favour a question over subdivision, and vice-versa. Once the renderer is implemented, we can try to find an optimum balance between asking questions and subdivision by experimenting with different values of alpha.

The algorithm for finding the largest source of error runs as follows:

```
function FindLargestErrorContributor(var ProblemIsOurSize : boolean;
                                              var LargestPatchEntry : CIndex);

  if Children list isn't empty then
    ProblemIsOurSize = true;

  else
    AnswerError = 0;
    TotalPatchFactorError = 0;

{ Find the patch that contributes the greatest error to the radiosity estimate, and }
{ also find the total amount of error that the patch-factor errors contribute. }
    with ContributingPatches do
      for i = 1 to NumberOfContributingPatchRecords do
        with ContributingPatch[i] do
             TotalPatchFactorError = TotalPatchFactorError + PatchFactorError;
             if AnswerError > MaximumAnswerError then
                    MaximumAnswerError = AnswerError;
                    LargestPatchEntry = i;

    ProblemIsOurSize = TotalPatchFactorError > alpha * MaximumAnswerError
end;
```

5.2.7 The Answer Method

The algorithm for answering a question is:

```
  function  Answer(askingPatch : CPatchObject;
                     var subdivisions : list of CPatchObject) : CAnswer;

  FindLargestErrorContributor(ProblemIsOurSize, LargestPatchEntry);

  if ProblemIsOurSize then
    Subdivide;
    subdivisions = children;

  else
    if we're already asking a question then
      return CurrentAnswer;
    else
      Ask largest error contributor for a better answer;
      if the patch subdivides and returns its children to us then
        Add entries for these children to the ContributingPatch list;
        Remove the original patch's entry;
        RecalculateAnswer;
        return CurrentAnswer;
      else
        UpdateContributingPatchInfo(LargestPatchEntry);
        RecalculateAnswer;
        return CurrentAnswer;
end;
```

This algorithm typically results in a sequence of questions, originating from the eyeball, which terminates when a patch object decides to subdivide itself. We refer to such sequences as "lines of enquiry", and it often useful to write them down in a manner similar to:

```
Eyeball -> A -> B -> C -> D : D subdivides.
```

Where A -> B indicates that patch A is asking patch B a question. It should be noted that once any such line of enquiry has terminated, there is a corresponding flow of information from the end of the line back to the eyeball, as each patch object in turn returns its new answer.

An important point about the algorithm is that it only tries to improve the current answer if the patch object isn’t already involved in the current line of enquiry. We do this to avoid looping, which will occur if the patch object continues to ask questions, resulting in a sequence like:

```
Eyeball -> A -> B -> C -> A -> B -> C -> A -> B -> C -> A ...      ( Etc. )
```

The loop is guaranteed to repeat because when any patch in the loop is asked to improve, it won’t have changed since the last time it was questioned, and therefore will make the same decision as to which patch it will question next.

If we follow the answer algorithm and halt the above enquiry when it reaches patch A for the second time, then when the patches update their contributing patch lists, they will change the criteria for deciding what to do when they’re next asked a question. The most probable course of events is something like:

```
Eyeball -> A -> B -> C -> A : A already asking.   (Enquiry 1)
Eyeball -> A -> B -> C -> A : A already asking.   (Enquiry 2)
Eyeball -> A -> B -> C -> A : A subdivides.       (And so on...)
```

Another problem we have is that it is possible to get into a situation in which the same line of enquiry is traced out again and again fruitlessly. If in the second enquiry above, A’s answer was exactly the same as its answer in enquiry 1, we would end up repeating the same enquiry over and over again, as none of the information in the contributing patch lists of the objects would change from the previous enquiry. Although it seems unlikely, experience has shown that this can happen one or twice early on in the rendering process.

To prevent this situation from occurring, we use the *AnswerNotImproving* flag present in the entries of the contributing patch list. If a contributing patch doesn’t return an answer that is more accurate than before, we set this flag. When the *FindLargestErrorContributor* method is searching for the patch with the largest *AnswerError*, it ignores any entries with this flag set, although it will clear the flags so that those entries are considered the next time it is called.

5.2.8 Improvements to the Answer Algorithm

When the renderer was implemented, it proved to work reasonably well. Observation of the algorithm suggested a number of possible modifications that could be made to improve performance. The most important of these is described in this section.

The modification is based around the observation that when a patch is asked to improve its last answer by another patch, it may not need to do any work to produce that improvement. Consider the following sequence of enquiries:

```
1. Eyeball -> A -> B -> ...
2. Eyeball -> C -> D -> B -> ...
3. Eyeball -> C -> A -> B -> ...
```

When A asks B to improve its answer in enquiry 3, it is only expecting a better answer than it got in enquiry 1. The question posed by D in enquiry 2, however, will have already forced B to improve its answer from the one it gave in that first enquiry, and thus B's current answer will be enough to satisfy A. Indeed, B's current answer may already be as accurate as it ever needs to be, in which case any further effort put into improving it would be wasted.

Our solution to this is to add an improvement counter to the patch object. Every time the patch object does some work to improve its answer this counter is incremented, and its current value returned along with the new answer. Any questioning patch can then store this value alongside the corresponding *ItsLastAnswer* field. The next time it asks the patch object for a better answer, it passes in the last counter value it received, so that the questioned object can check whether any further improvement is really necessary. To achieve this, we modify the Answer method as follows:

```
function Answer(askingPatch, theSubdivision, var ImprovementCount);

  if ImprovementCount < CurrentImprovementCount then
    ImprovementCount = CurrentImprovementCount;
    return current answer

  else
    { ... as before ... }
    CurrentImprovementCount = CurrentImprovementCount + 1;
    ImprovementCount = CurrentImprovementCount;
end;
```

For the simple scenes with which we were working, this modification reduced the number of subdivisions required to achieve a given figure of accuracy by 3-5%. We would expect that with more complex scenes, the savings would be correspondingly greater.

5.3 The Eyeball

As noted in chapter 2, the eyeball object shares many operations with the patch object, and so we let it inherit from the patch object, overriding any methods as necessary. We recall that it differs from the patch object in a number of ways:

- Its criteria for deciding how accurate its answer is differs from those of a patch object; it is trying to produce a picture that is perceived to be accurate to some degree. We must find how this picture accuracy can be related to the answer ranges given by the patch objects.

- It isn't actually part of the scene, and therefore doesn't have to answer any questions, so there is no need for it to subdivide.
- The eyeball has an elemental surface area. A consequence of this is that we can calculate patch factors to it exactly.

Before we present the design of the eyeball object, we consider how the perceived error in the picture produced by the eyeball is related to the answers it gets from the patches in the scene.

5.3.1 Displaying Patch Objects

When designing the eyeball object for the test-bed renderer, we assumed that it would be producing a picture for a raster display[1]. Thus the output of the eyeball would be a grid of pixel intensities which, when shown on a monitor, would produce an image of the scene. This allowed us to define the accuracy of a picture somewhat more formally; for picture to be accurate to x%, every pixel in the picture must lie within x% of its actual value. To be able to check that this is so, we must have a way of determining what value should be assigned to a pixel, given the current answers of all the patch objects in the scene.

When we display a particular patch, we must determine what intensity we should assign every pixel that patch crosses. For diffuse surfaces it can be shown that this intensity is the same for all the pixels, and is directly proportional to the radiosity of the patch. (This is a common exercise in the illumination section of many graphics courses and textbooks; see [Fole90] pp. 723, for example.) Hence to display such a patch we would flat-shade it with its radiosity value, scaled by some convenient factor[2].

The upshot of this is that for a picture to be accurate to x%, every patch object that can be seen by the eyeball must be perceived to have an error less than x% in its answer. Thus a fundamental difference between the eyeball and a normal patch object is that we don't need to take into account the patch factors between the scene patch objects and the eyeball.

We now take a brief look at the two components that lie between the intensity we compute for a particular pixel and the image of that pixel seen by the viewer; the computer monitor and the human eye.

5.3.2 The Human Visual System

There is a large body of experimental evidence that suggests that the perceived brightness of an object is a logarithmic function of its actual intensity[3]. If we have a row of pixels, each of which has an

[1]An eyeball that produced an object-space list of 2D-surfaces could be constructed but, as we shall see, this would affect a number of design decisions.

[2]For instance, we can choose this factor so that the maximum patch radiosity in the scene will be mapped to the maximum possible pixel intensity.

[3][Gonz87] presents a summary of the human visual system.

intensity double that of the previous pixel, we will in fact perceive those intensities to be equally spaced; figure 5.1 illustrates this relationship.

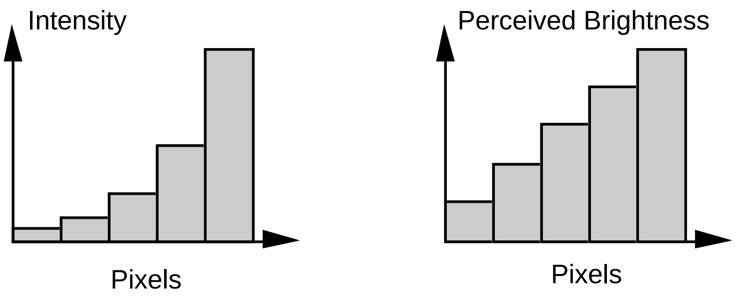


Figure 5.1 Perception of Intensity

The human visual system can cope with an enormous range of intensities, covering some ten orders of magnitude. However, it cannot handle such a wide range of intensities simultaneously. Instead, it adapts to the current *background* level of light, with a much smaller range of discrimination around that level. Below this range, all intensities appear black, and likewise above it all intensities appear white. The upper level of the range tends not to have too much meaning, however, as intensities much higher than this maximum would increase the background level of light. The eye would then adapt to this higher level, shifting the former maximum upwards.

Experimentation has shown that black and white limits of the range tend to lie at 1.5 log units above and below the background level of intensity. Within this range there is a region of about 2.2 log units over which the logarithmic relationship mentioned above holds with reasonable accuracy.

5.3.3 Monitor Characteristics

Ideally, a monitor would reproduce the intensities it is given to display exactly. In practice the intensity displayed for a particular pixel (I) is a non-linear function of the pixel's actual value (P). For most monitors, this function can be approximated by the equation

$$I = k_0 P^{\gamma} + I_{Min} \tag{5.1}$$,

where γ is a constant, typically in the range 1 to 3, and I_{min} is the minimum level of light the monitor can display. There is also a maximum intensity, I_{max}, beyond which the monitor cannot be driven. The ratio between these minimum and maximum levels is called the *contrast ratio* (ε) of the monitor, and for high-quality monitors tends to be around 1:50. Background light reflecting off the monitor decreases this ratio, so for a normal working environment we generally estimate ε as being somewhere in the range 1:20 to 1:40

We can use this information to choose an appropriate value for a pixel given the intensity I_{req} that we require it to have. We assume that $I_{req} = 1$ corresponds to the maximum displayable intensity, and gets mapped to the maximum possible pixel value P_{max}. This results in the equation

(5.2)
$$P = P_{max}\left[\frac{\left(I_{req} - \varepsilon\right)}{(1-\varepsilon)}\right]^{\frac{1}{\gamma}},$$

where we clip the resulting value for P to [0, P_{max}] if necessary.

5.3.4 Our Model

The model we have chosen for perceived intensity is a simple one. We make the following assumptions:

- The perceived intensity is some multiple of the log of the actual intensity.
- The display system is ideal. We assume gamma is 1, mainly because many graphics work-stations these days have built-in gamma correction. The renderer was developed on a Macintosh computer which had a display card capable of such correction.
- The display's limits are the significant ones, rather than those of the eye. This is a fairly safe assumption, given that the range of intensities that the eye can perceive simultaneously is far larger than the range of intensities that can be produced by a computer monitor.
- The minimum and maximum thresholds are cut-off thresholds; we don't make any attempt to model a smooth transition between the curve and the minimum and maximum intensities.

Figure 5.2 (overleaf) shows a graph of the model.

To find the perceived error in a range of absolute intensities, we can write:[1]

(5.3)
$$\begin{aligned} PerceivedError &= Ln(top) - Ln(bottom) \\ &= Ln(top / bottom). \end{aligned}$$

and thus the perceived error in the radiosity of a patch can be calculated as:

```
function PerceivedError (RadiosityEstimate : CRange);

  with RadiosityEstimate do
    PinNumber(epsilon, 1, top);          { Clip top to [epsilon, 1] }
    PinNumber(epsilon, 1, bottom);       { Clip bottom to [epsilon, 1] }

  { Compute perceived error as a number in the range [0,1] }
    return -ln(top/bottom) / ln(epsilon);
end;
```

[1]The log function is expensive to compute. As we are only interested in whether a particular patch answer has more error than some other answer, we can instead use the relative error of answers as a method of comparison, as it can be shown that

PerceivedError(A) > PerceivedError(B) <=> RelativeError(A) > RelativeError(B).

(The relative error of a range is (top-bottom)/(top+bottom).)

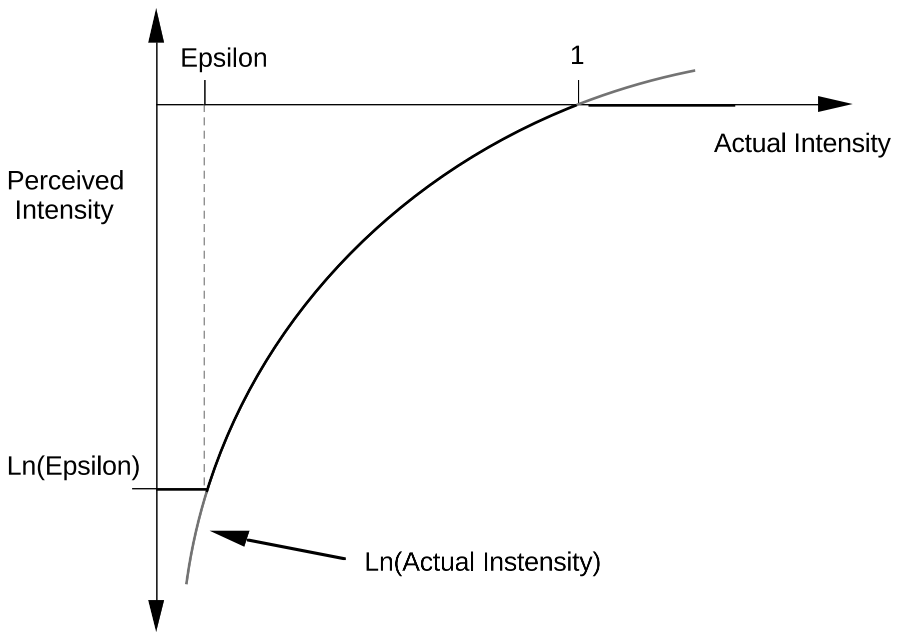


Figure 5.2 Perceived Intensity Model

5.3.5 The Snap Method

To actually carry out the rendering of the scene, we introduce a new method,*Snap*:

```
procedure Snap (requiredAccuracy : CReal);

  repeat
    FindLargestErrorContributor(ProblemIsOurSize, LargestPatchEntry);

    with LargestPatchEntry do
      if PerceivedError(ItsRadiosity) * ItsPatchFactor < requiredAccuracy then
        finished = true;

    Ask LargestPatchEntry.Patch for a better answer;
    if the patch subdivides and returns its children to us then
      Add entries for these children to the ContributingPatch list;
      Remove the original patch’s entry;
    else
      UpdateContributingPatchInfo(LargestPatchEntry);

  until finished;

  DisplayPatches(ContributingPatchList);
```

It operates much like a simplified version of the Answer method, except that it doesn’t try to calculate any answer; it just iterates until all its contributing patches have a perceived error less than that required by the user. At this point it terminates, and displays the scene, using conventional polygon-rendering techniques.

The *Snap* method relies on the *FindLargestErrorContributor* method to use perceived error as its criterion for selecting the *LargestPatchEntry*, and also not to tell the eyeball that it should subdivide. Thus we override the patch object's version of the method with:

```
{ ... }

    itsPerceivedError := PerceivedError(ItsLastAnswer) * ItsPatchFactor;
    if itsPerceivedError > MaximumAnswerError then
             MaximumAnswerError = itsPerceivedError;
             LargestPatchEntry = i;

{ ... }

IsOurSize := false;
```

The surprising thing in both of these methods is that we incorporate a patch's patch factor when calculating its perceived error, especially considering that we have already explained why it should be ignored! The next section explains why this is done.

5.3.6 Below the Pixel Level

Up until now we have assumed that a pixel's intensity is proportional to the radiosity of the patch that covers it. However, we haven't considered what happens if a pixel is only partially covered by a patch. In such a case we must weight the radiosity with the fraction of the pixel that is covered by the patch.[1] If more than one patch covers the pixel, we sum their individual contributions.

To see why we must take this into account, we consider what happens in the case of a discontinuity in radiosity across a patch. Normally patches will subdivide until the variation in their radiosity is accurate enough for the eyeball. However, in the case of severe discontinuities, it is possible for subdivision to continue for a long time, producing subdivisions well beneath pixel size. In the case of first-order discontinuities, as in figure 5.3, this subdivision could continue forever without the accuracy of the eyeball ever improving.

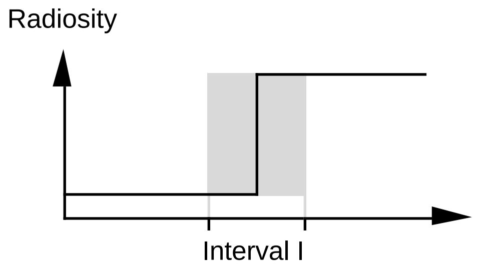


Figure 5.3 A first-order discontinuity.

We can stop this infinite subdivision by taking pixel areas into account. As soon as the size of a patch drops below the size of a pixel, we start weighting its radiosity, and hence its error, with the

[1]This is assuming a simple box-filter anti-aliasing scheme. More complex schemes such as cone-filtering would need an alternative weighting system.

fractional area of the patch. This ensures that the error will decrease with subdivision, no matter how sharp any discontinuity is. We can carry this out by using the patch factor method, overriding it with:

```
ActualPatchFactor = inherited PatchFactor(thePatch);
if ActualPatchFactor > PixelArea then
  PatchFactor = 1
else
  PatchFactor = ActualPatchFactor/PixelArea;
```

Whenever we calculate the perceived error of a patch, then, we multiply the perceived error of its answer by this patch factor in order to get the actual error of the pixel.

5.4 A New World Object

5.4.1 Discussion

Before we can use the eyeball and patch objects to build a rendering system, we must address the problems involved in starting the rendering system. We begin with a collection of patch objects that describe the scene we need to render, and we must determine the initial state of each of these objects, i.e., their initial answer. Here we have a dilemma, because determining the initial answer of any patch would require knowledge of the initial answers of all the other patches. Indeed, we have the classic chicken and the egg paradox.

To solve this problem, we introduce the world object. The world object can be thought of as a “master” patch object that encapsulates the properties of all the patches. Thus the answer method of the world will return bounds on the maximum and minimum radiosity of any of the scene patches. These bounds will fairly rough ones, as the world will have only the reflectances and the internal radiosities of the patches from which to calculate them.

At the start of the rendering process, each patch object has in its contributing patches list a single entry that refers to the world object. It can then use the answer of the world object to calculate its initial answer without having to interact with the other patches in the scene.

The first time a patch object asks the world to improve its answer, it performs a pseudo-subdivision, and returns all of the scene patches to that patch. The patch can then proceed to interrogate these other patches in order to improve its own answer, and rendering can proceed as normal. Indeed, we can also apply this method to the eyeball, so that we start the rendering system with just the world object contributing light to the eyeball object. The inter-object dialogue that follows would look something like:

```
Eyeball -> World : Subdivides and returns its children, the scene patches.
Eyeball -> P1 -> World : Subdivides and returns its children.
Eyeball -> P2 -> World : Subdivides and returns its children.
Eyeball -> P2 -> P1 -> P3 etc.
```

5.4.2 Design of the World Object

We start with the problem of calculating approximate bounds for the world's answer. If we assume that all patch objects in the scene have reflectances less than one, then the patch with the largest radiosity must be a light source. It needn't be the light source with the largest internal radiosity, however; if that light source had a low reflectivity, and there was another light source with a slightly smaller internal radiosity but a large reflectivity, it would be possible for the latter light to have the largest total radiosity. It is possible to construct an algorithm to find the "brightest" light source under these conditions, but we simplify things enormously if we assume that all light sources in the scene don't reflect any light. In this case, the largest radiosity will definitely be that of the brightest light source.

A lower bound for the radiosities in the scene is far easier to find; we simply take zero as our minimum, reasoning that it is quite possible for a particular patch in the scene to receive no light at all from any of the other patches.

When we are setting up the entry for the world in one of the scene patches, we must also produce its associated patch factor. We cannot use the object's *PatchFactor* method to do this, because the world has no definite position. What we currently do is assume that, in the worst case, the brightest patch completely encloses the object and thus the patch factor from it is one. For the lower bound, we assume that the patch can see no other patches at all, and so its patch factor to the world will be the range [0, 1].

We must also make sure that the entry is set up so that the patch object's first course of action is to ask the world to improve its answer; it would be disastrous if instead the patch decided to subdivide itself. We can do this by setting the *PatchFactorError* field to zero, and the *AnswerError* field to one.

The world object can be defined as inheriting from the patch object as follows:

```
CWorldObject = object (CPatchObject)

  procedure SetScenePatches (theScenePatches: CList);
  function  Answer(questioningObject : CPatchObject;
                   var subdivisions : list of CPatchObject) : CAnswer;
  override;
end;
```

When creating the world object, we must give it a list of the patches in the scene, so that it can work out its own answer. It can then store this list in its *Children* field, so that later it can return it to any enquiring patch. To force the world to always subdivide and return its children, we override the *Answer* method:

```
function Answer(questioningObject : CPatchObject;
                   var subdivisions : list of CPatchObject) : CAnswer;

  subdivisions = children;
end;
```

One problem that we have is that, when setting up a patch object's world entry, we need to temporarily override its mechanisms for calculating patch factors and error information. This is handled by making the world object itself responsible for creating this entry, and adding it to the patch's contributing patches list. The setting up of these entries and the calculation of the world's answer takes place when it is passed the list of scene patches, in the *SetScenePatches* method:

```
procedure SetScenePatches (theScenePatches: CList);

   Children = theScenePatches;

   MaximumRadiosity = largest of child[i].InternalLight;

   CurrentAnswer = Range(0, MaximumRadiosity);

   for each scene patch do
     Create contributing patch entry;
     Add entry to the patch's contributing patch list;
     with entry do
        ItsPatchFactor = Range(0,1);
        ItsLastAnswer = CurrentAnswer;
        ItsFluxError = 0;
        ItsAnswerError = 1;
end;
```

5.5 Summary

This chapter has introduced our design for a three-dimensional fuzzy renderer, which makes several assumptions as to surface type and visibility. In spite of these assumptions, the outline of a patch object produced early in the chapter should be applicable to more general versions of the renderer.

In developing the final design from this outline, a number of points had to be considered. These included how a patch object should maintain information about other patches, and how it should make various decisions, such as whether to subdivide, or which other patches to question. We also had to decide how to avoid looping in inter-patch communications.

Of course, the specification of a fuzzy renderer also calls for an eyeball object. The main task in the design of this object was finding a definition for our previously vague notion of "perceived accuracy". We also had to introduce the notion of a world object, in order to provide proper initialisation for the system.

An implementation of this design was developed during the course of this thesis. Chapter 6 discusses some of its more important features, and chapter 7 presents the results that were gathered using the renderer.

CHAPTER 6

Implementation Notes

6.1 Introduction

In this chapter we take a brief look at how the renderer was implemented, and in particular at some of its features. The available options for displaying a rendered scene and monitoring the renderer's operation are also presented.

6.2 The Renderer

6.2.1 Technical Specifications

The current version of the fuzzy renderer runs on high-end colour-capable Macintosh computers. The particular machine we were using, a IIcx, had a 24-bit colour card and a floating point coprocessor. The renderer was written in Object Pascal, and made use of the Tridee library [Lobb92]. (This is a class-based rendering toolkit which provides basic components for transforming and viewing scenes.) It consists of approximately equal amounts of rendering code, and monitoring code.

Object Pascal was chosen as the implementation language for a number of reasons. The renderer was first and foremost a prototype, and thus we wanted to use a reasonably high-level language, as it was envisioned that the code would go through many changes before the final version. This certainly proved to be the case. It also helped that the object extensions to Pascal were clean and elegant, and that there was a list class readily available, always useful in graphics work. The two most important reasons, though, were the availability of Tridee, which was written in this language, and the extremely good debugging environment that the Think Pascal compiler offered.

6.2.2 Memory Issues

Currently the limiting factor for the renderer is memory. The machine used for the rendering has 20Mb of memory installed, of which 17Mb is available to the renderer after the overhead of the debugger and the operating system. Even with this amount of memory, we found that for some scenes we couldn't reach our accuracy goals with the number of patches available.[1]

A typical five-patch scene tends to run out of memory after generating 3000 child patches, and therefore each patch is taking up around 5.5 Kb of memory. Each patch object takes up around half a Kb for itself, plus another Kb for every four contributing patches it keeps track of. Thus on average each patch is keeping track of twenty other patches. Part of the reason for these sizes is the use of extended precision (twelve-byte) arithmetic for all floating-point numbers, both for its extra accuracy, and also to take full advantage of the machine's floating-point processor[2].

In some ways the memory constraint proved an advantage. It forced us to take a closer look at the strategies of the rendering algorithm, in order to reduce the number of patches needed to reach a reasonable level of accuracy. In particular, the work on the numerical bounding method of chapter 4 was largely due to these memory constraints.

6.3 The Tridee Library

The Tridee class library contains a number of tools necessary to construct various types of renderer, such as Phong or Gouraud-shading polygonal renderers, or a BSP-tree renderer.

Hierarchical scenes can be constructed from various *Scene Objects* available from a *Library*, and various properties such as *Surfaces* can be attached to these objects. Such a scene can then be fed into a particular *Camera*, which transforms it into objects suitable for display, usually by passing it through a chain of *Pipeline Stages*, and then passes these objects on to a *Display*, which draws them on the screen.

For the fuzzy renderer, we use the polygon-based *Face* scene object to represent patches. Each patch object creates its own face and attaches a surface of the appropriate colour to it. When the time comes to display the scene being rendered, all of these faces can be gathered together and sent off to the appropriate camera.

More detail on how the Tridee system works can be found in [Lobb92].

[1]Virtual memory was considered, and indeed used earlier when we only had 8Mb of memory installed on the machine. Thrashing proved to be a problem, however, and the limited amount of swap-space available has meant that we haven't been able to use it with out current setup.

[2]By using the same floating-point format as the FPU, we avoid conversions between formats, which can cause excessive slow-downs because of a bug in the machine's software.

6.4 Forms of Display

The renderer displays the scene incrementally, redrawing each patch whenever it returns a better answer to the eyeball[1]. When the renderer is first started, the scene is displayed completely flat-shaded. However, as the various patches subdivide and are updated on the screen, it quickly starts looking more realistic.

The camera that the renderer sends the faces to can be changed, in order to vary the style of display. We currently have three different methods for displaying a patch; these are detailed below.

6.4.1 Pseudo-Colour Display

The most commonly-used form of display for the renderer is the pseudo-colour display. All patches are displayed as flat-shaded polygons with the current estimate of their grey-scale intensity. Colour is then used to indicate the error in that intensity. An example of the results it produces can be seen in figure 6.1, which shows a large area light source over a single diffuse patch, which has a reflectance of 0.3. For further information about the pseudo-colour aspect of the display, see 6.5.1.

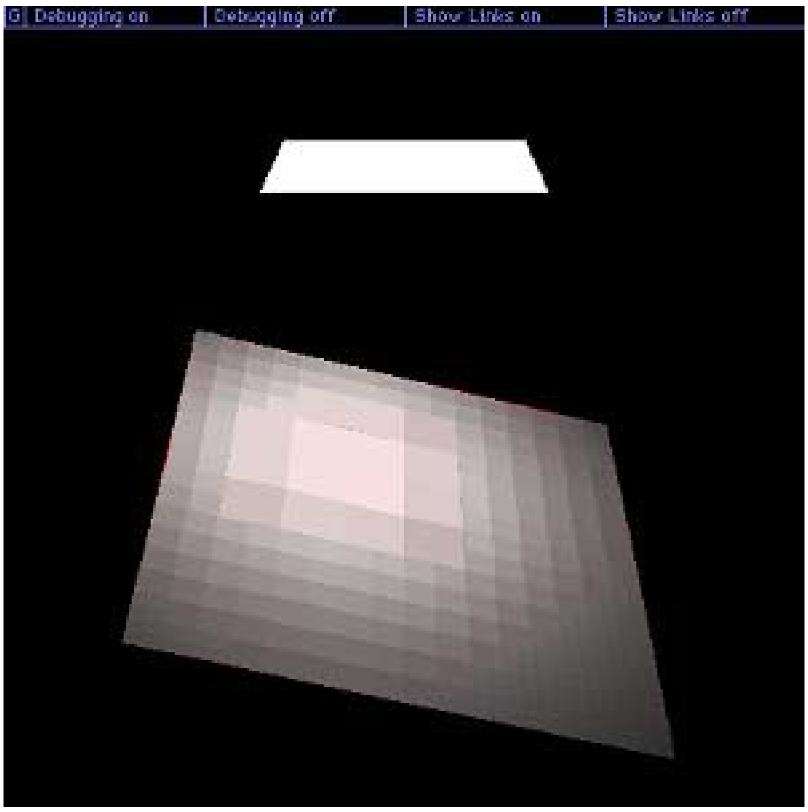


Figure 6.1 Pseudo-Colour Display

[1]The implementation of the renderer refers to the eyeball object as a “pinhole camera”, for historical reasons.

6.4.2 Vector Display

We found that we were sometimes more concerned with the pattern of subdivision than the intensities of the various patches. In such cases we just wanted to see the outlines of the current set of patches. The vector display can be used to produce such a picture, essentially by ignoring most of the information about the faces that get passed to it. In the example shown in figure 6.2, we can see that the area of the patch where the least subdivision has taken place is directly below the light, with more subdivision occurring towards the edges. This occurs mainly because the gradient of the radiosity distribution over the patch is lower towards the maximum[1].

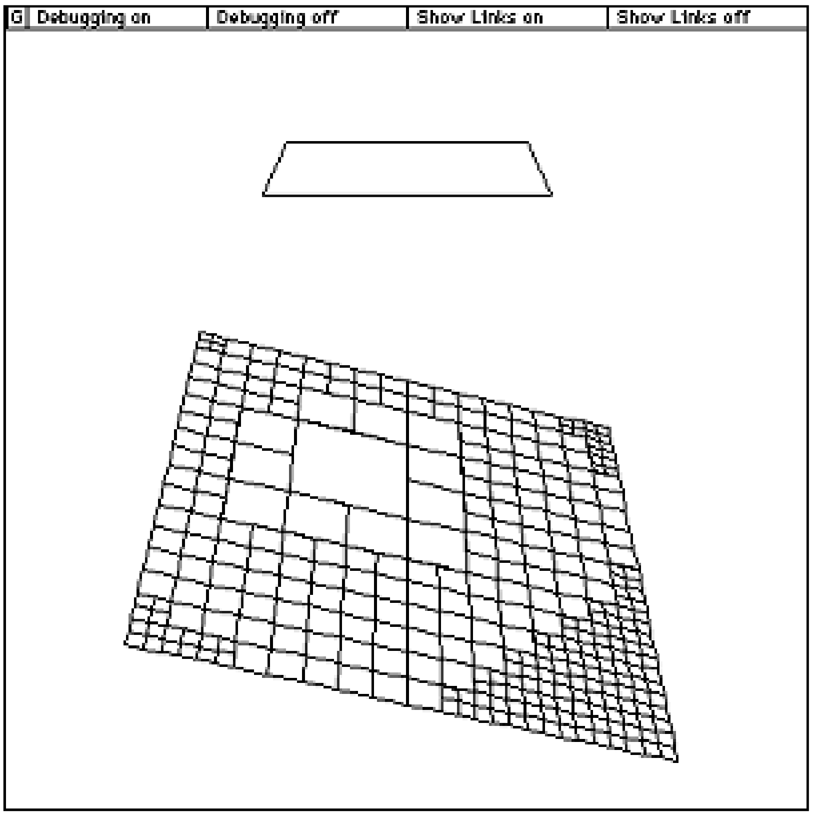


Figure 6.2 Vector Display

6.4.3 Fuzzy Display

The third display method is an experimental one, and is responsible for the name we have given our technique[2]. When displaying a patch, we pick each pixel value randomly from the range of intensities we have calculated for it. This results in a truly fuzzy patch; if it has a large error, it will look heavily speckled, whereas more accurate patches seem only lightly dappled, or even perfectly smooth. (See figure 6.3.)

[1]It also occurs because of our model of perceived accuracy; the error in the radiosity of the brighter areas of the patch is not regarded as being as significant as the error in the dimmer parts of the patch.

[2]Prior to this thesis it went under such snappy titles as “Rendering with collaborating scene objects”. While the new name has a lower information content, it does have the advantage that it is easier to say.

There are some drawbacks to this approach; in particular the high-frequency component of the noise tends to irritate the eye unduly. However, it does have the important advantage that it will still work when real colour is introduced to the renderer, unlike the pseudo-colour display.

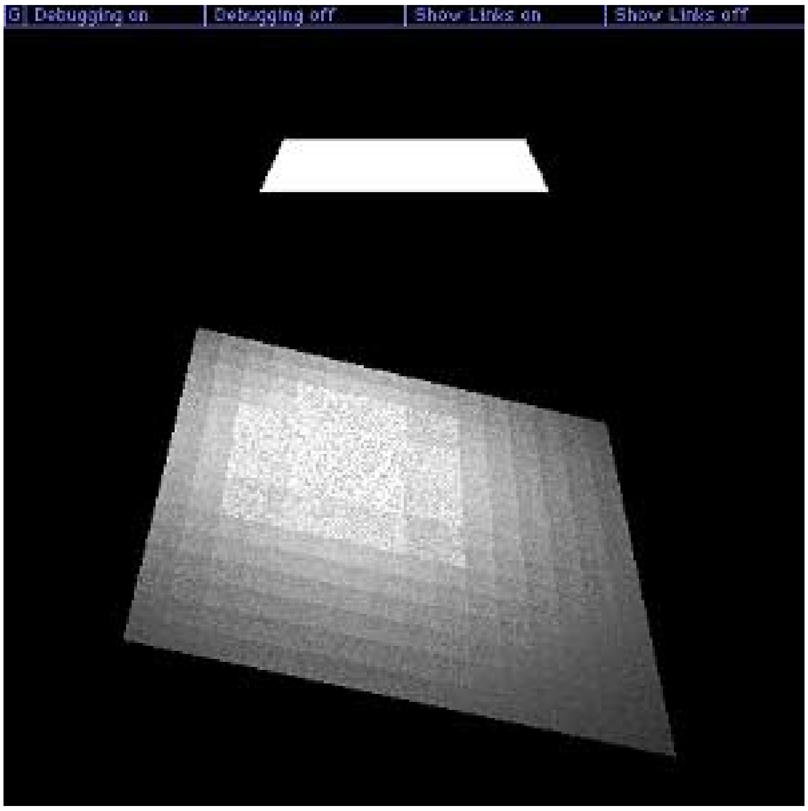


Figure 6.3 Fuzzy Display

It was hoped that a Gouraud-shaded display could be produced, as obviously this would result in more presentable images. (See the differences between figures 1.9 and 1.10, for instance.) A scheme for producing a Gouraud-shadable mesh from the patch-objects has been developed, but lack of time has prevented its implementation.

6.5 Debugging and Monitoring Facilities

Especially in the early stages of this thesis, one of the biggest problems with the rendering system was understanding how its separate parts were interacting. Towards the end of a run the system usually consists of thousands of patch objects, each storing its own information on how the rest of the scene affects it. This makes comprehension of the current state of the system difficult, and consequently understanding why subdivision is taking place or failing to take place at a certain point can be hard. More information about the system is needed than a simple display of the surfaces in the scene can provide, especially in order to tune parts of the renderer or identify problems with it. To this end a number of monitoring techniques were implemented to give some insight into the way the algorithm is functioning.

6.5.1 Pseudo-Colour Display

When displaying a patch, pseudo-colour is used to indicate its properties. The hue-saturation-value colour model is used, such that:

- Saturation indicates error. Thus if a patch has an exact result for the light it is reflecting towards us, it will appear some shade of grey. The greater the error in its answer, the greater the amount of colour it has.
- Hue can be used to indicate subdivision depth. Patches are assigned a different hue depending on how many 'reflections' they are relying on to determine their answers.
- Value (that is, the intensity of the colour) indicates the current estimate of the answer. We take this as being the middle of the bounds on the answer.

From these cues it is possible to see what error in the scene remains, where it is located, and how effectively that error is being reduced.

6.5.2 Debugging Information

The simplest form of monitoring is to turn on the debugger, which prints out information on the current activities of the renderer. This includes inter-object enquiries, and any decisions a patch might make about how to improve its answer. The answers of the patches are also displayed as they are updated.

Each patch is assigned a name-tag to help keep track of it. Whenever a patch is subdivided, its name is passed on to each of its children, suffixed by a letter that indicates the position of the child. For instance, the tag “Patch AAC” would indicate that the patch is the lower-right subdivision of the upper-left subdivision of the original “Patch A”. Reproduced on the next page is an extract from a typical rendering session with the debugger turned on.

One of the major drawbacks of using the debugger is that the renderer slows to a crawl while it dumps all of this information to the screen. Indeed, the amount of detail it gives is often simply overkill; we may be only interested in one particular sequence of queries that occurs half way through a rendering run.

For this reason the debugging mode can be turned on and off during the rendering process, by clicking on the appropriate buttons at the top of the picture window. (These buttons can be seen in most of the pictures produced by the renderer in this chapter.)

```
Camera: Working on Patch DA,
which has error 65.09
=================
Patch DA : decided to ask Patch A
(last answer was : 0.0954 - 1.1203.)
Patch A : decided to ask Patch C
(last answer was : 0.0000 - 2.7000.)
Patch C : decided to subdivide.
  Updating CP list in Patch A
  Patch A : New ER: 0.0954 - 0.8081
  PF excess:1.6676
  Patch DA : New ER: 0.0488 - 0.4917
  PF excess:1.3114
  Camera : new ER: 17.7698 - 47.9052

Camera: Working on Patch BB,
which has error 65.08
=================
Patch BB : decided to ask Patch A
(last answer was : 0.0954 - 1.1203.)
Patch A : decided to ask Patch D
(last answer was : 0.0000 - 2.7000.)
Patch D : decided to subdivide.
  Updating CP list in Patch A
  Patch A : New ER: 0.0954 - 0.4951
  PF excess:1.7617
  Patch BB : New ER: 0.0488 - 0.4440
  PF excess:1.2621
  Camera : new ER: 17.7698 - 47.8114

Camera: Working on Patch DBDDB,
which has error 64.87
=================
Patch DBDDB : decided to ask Patch C
(last answer was : 0.0016 - 0.8945.)
Patch C : decided to subdivide.
  Updating CP list in Patch DBDDB
  Patch DBDDB : New ER: 0.0030 - 0.2523
  Camera : new ER: 17.7707 - 47.7901

Camera: Working on Patch DC,
which has error 64.68
=================
Patch DC : decided to ask Patch A
(last answer was : 0.0954 - 1.4134.)
Patch A : has improved since last query
Returning current answer.
  Patch DC : New ER: 0.0488 - 0.3923
  PF excess:1.0016
  Camera : new ER: 17.7707 - 47.6523

Camera: Working on Patch DBDDD,
which has error 64.65
=================
Patch DBDDD : decided to ask Patch C
(last answer was : 0.0016 - 0.8945.)
Patch C : decided to subdivide.
  Updating CP list in Patch DBDDD
  Patch DBDDD : New ER: 0.0031 - 0.2501
  Camera : new ER: 17.7716 - 47.6310

Camera: Working on Patch DBBB,
which has error 64.51
=================
Patch DBBB : decided to ask Patch BC
(last answer was : 0.0017 - 0.5318.)
Patch BC : decided to subdivide.
  Updating CP list in Patch DBBB
  Patch DBBB : New ER: 0.0017 - 0.2626
  Camera : new ER: 17.7716 - 47.6236
Picture accurate to: 64%

Camera: Working on Patch BA,
which has error 64.27
=================
Patch BA : decided to ask Patch A
(last answer was : 0.0954 - 1.1203.)
Patch A : decided to subdivide.
  Patch AA : New ER: 0.0966 - 0.4711
  PF excess:1.6222
  Patch AB : New ER: 0.0966 - 0.4043
  PF excess:1.3123
  Patch AC : New ER: 0.0966 - 0.4743
  PF excess:1.6222
  Patch AD : New ER: 0.0966 - 0.4054
  PF excess:1.3123
  Updating CP list in Patch BA
  Patch BA : New ER: 0.0468 - 0.4708
  PF excess:1.5994
  Camera : new ER: 17.7696 - 47.5724

Camera: Working on Patch CBBD,
which has error 64.24
=================
Patch CBBD : decided to subdivide.
  Patch CBBDA : New ER: 0.0599 - 0.2297
  Patch CBBDB : New ER: 0.0026 - 0.1649
  Patch CBBDC : New ER: 0.0772 - 0.2673
  Patch CBBDD : New ER: 0.0027 - 0.1898
  Updating CP list in Pinhole Camera
  Camera : new ER: 17.9096 - 48.1567
Warning: Error has increased in Camera
Memory left: 9400k
...
```

6.5.3 Showing Links

In a similar way it is possible to turn *link animation* on and off while the renderer is running. When the animation has been turned on, every time a patch asks another patch a question a link is drawn between the two on the screen. The subdivision of patches is also shown. Thus while the renderer is running it is possible to see the lines of enquiry between patches being traced out as they occur. This provides a very useful tool for ensuring that the pattern of these enquiries is what we expected.

Figure 6.4 shows an example of link animation. The patch at the lower-right of the back wall has asked a patch on the floor for its answer, and that patch has in turn asked the entire side wall for its answer. At this point the line of enquiry has terminated, with the side wall subdividing, as shown by the lines of subdivision drawn on the patch.

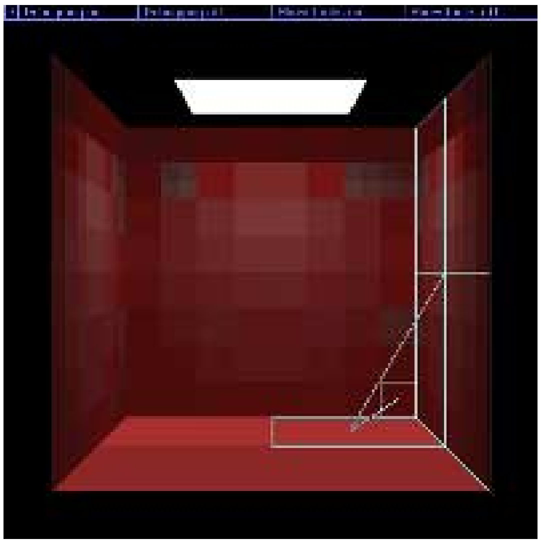


Figure 6.4 Link Animation

6.5.4 Stop-Frame Photography and Logging

The result of the rendering process is a picture that evolves over time, from an initially flat-coloured scene to a the final, fully-illuminated answer. While watching the renderer in action leads to some insight into this process, the rendering tends to take place too slowly for all but the most patient observer. Moreover, once the rendering is finished, the process cannot be reviewed without starting it over again.

To alleviate this problem, stop-frame photography is used. The renderer can be set up so that periodically the current picture of the scene is saved, as well as a number of statistics on the subdivision process at that point in time. At the end of a run, a Quicktime movie can be made from all of these pictures. This movie can then be stepped through frame-by-frame, or played from start to end within a few seconds. The sped-up animation of the subdivision process can give an insight into the way the method is operating. In particular, one can see whether subdivision is occurring in the correct place, and whether the balance between subdivision and solution steps is correct.

6.5.5 The Gopher

At any time during the rendering process, rendering can be interrupted, and we enter the Gopher, a monitor program which lets the user examine the current state of the scene. Information about a particular patch can be displayed, including the other patches it depends upon in order to calculate its own answers, as well as the patch-factor ranges from those patches. It is also possible to render the view from such a patch. (See figure 6.5.) By doing this we get to see how the patches it depends on look to it, and which of those patches have had to subdivide most heavily in response to its questions. If we use the pseudo-colour display, we can also quickly get an idea of which patches it needs better answers from.

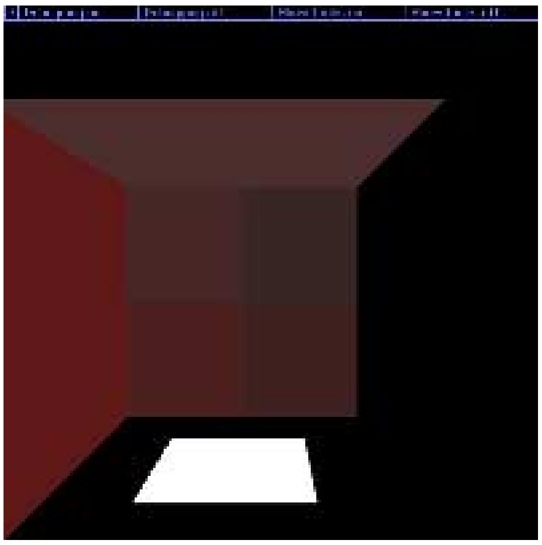


Figure 6.5 Viewing from a patch

The gopher can be used to examine the information that the eyeball holds, including the bounds it currently has on its answer, as well as information on the overall levels of subdivision in the scene. This information includes the total number of times any patch returned its subdivisions, as well as the number of times actual subdivision took place.

Reproduced below is a typical gopher session.

```
Picture accurate to: 56%
Memory left: 9100k
Picture accurate to: 55%
Picture accurate to: 54%

(Entered Gopher... )

[V]iew, [I]nfo, [P]atch Info,
E[x]it, [T]erminate, [H]alt,
Swap [C]amera, Swap [D]isplay
Your choice: i
Patches by level:
Level 1: 238 patches.
Level 2: 175 patches.
Level 3: 7 patches.

Total subdivisions: 139
real subdivisions: 80

Acc. of worst patch: [0.0030,0.1646].
Rel. Acc. : 51.08%
Number of worst patch: 134

[V]iew, [I]nfo, [P]atch Info,
E[x]it, [T]erminate, [H]alt,
Swap [C]amera, Swap [D]isplay
Your choice: v
Patch to view: (0 - 242) : 134
Displaying Patch CBDDDD
Building scene list
Finished!

[V]iew, [I]nfo, [P]atch Info,
E[x]it, [T]erminate, [H]alt,
Swap [C]amera, Swap [D]isplay
Your choice: p
Patch to get info for: (0 - 242) : 134
Printing Patch CBDDDD

Current Radiosity Estimate:
[0.0030,0.1646]
From patch Patch A:
Radiosity Contribution: [0.0020,0.0319]
Patch Factor: [0.0710,0.0753]
Last Flux Answer: [0.0954,1.4134]
Errors: Patch: 0.0010, Flux: 0.0289
Impr. Level: 0
Actual Impr. Level: 1

From patch Patch B:
Radiosity Contribution: [0.0001,0.0294]
Patch Factor: [0.1183,0.1313]
Last Flux Answer: [0.0016,0.7463]
Errors: Patch: 0.0015, Flux: 0.0279
Impr. Level: 2
Actual Impr. Level: 2

From patch Patch E:
Radiosity Contribution: [0.0000,0.0718]
Patch Factor: [0.0000,0.0266]
Last Flux Answer: [9.0000,9.0000]
Errors: Patch: 0.0718, Flux: 0.0000
Impr. Level: 0
Actual Impr. Level: 0

From patch Patch DA:
Radiosity Contribution: [0.0004,0.0060]
Patch Factor: [0.0297,0.0320]
Last Flux Answer: [0.0488,0.6270]
Errors: Patch: 0.0002, Flux: 0.0054
Impr. Level: 0
Actual Impr. Level: 3

From patch Patch DB:
Radiosity Contribution: [0.0000,0.0109]
Patch Factor: [0.0583,0.0613]
Last Flux Answer: [0.0017,0.5928]
Errors: Patch: 0.0003, Flux: 0.0106
Impr. Level: 0
Actual Impr. Level: 0

From patch Patch DC:
Radiosity Contribution: [0.0004,0.0052]
Patch Factor: [0.0303,0.0326]
Last Flux Answer: [0.0488,0.5300]
Errors: Patch: 0.0002, Flux: 0.0045
Impr. Level: 0
Actual Impr. Level: 1

From patch Patch DD:
Radiosity Contribution: [0.0000,0.0093]
Patch Factor: [0.0599,0.0627]
Last Flux Answer: [0.0017,0.4959]
Errors: Patch: 0.0002, Flux: 0.0091
Impr. Level: 0
Actual Impr. Level: 0

[V]iew, [I]nfo, [P]atch Info,
E[x]it, [T]erminate, [H]alt,
Swap [C]amera, Swap [D]isplay
Your choice:...
```

Chapter 7

Results

7.1 Introduction

In this chapter we present a number of findings that came from analysing the data produced by the logging system of the renderer. First of all, we look at a few of the images that have been produced, and establish a convenient scene for testing purposes. This scene was used to gather a number of statistics about the renderer; the following sections present some of the implications of these statistics.

7.2 Images Generated by the Renderer

Figure 7.1 shows the result of running the renderer on a couple of simple, one-patch scenes. Both pictures were rendered until accurate to 5%. The left-hand scene took 3870 patches for the error to drop to this level, and the right-hand one 2650 patches.

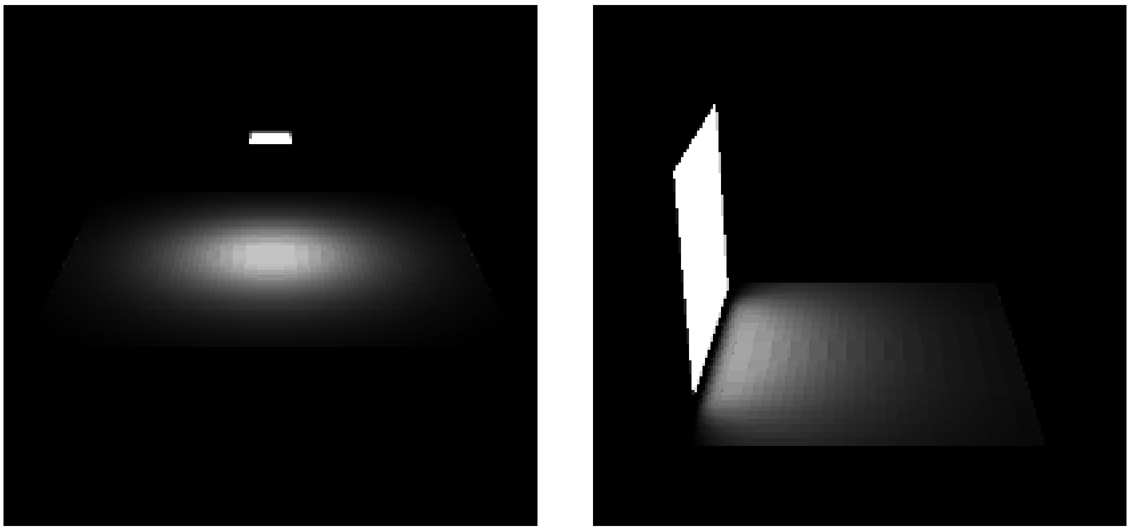


Figure 7.1 Single patch scenes

Several such scenes were rendered to check that the subdivision process was working reasonably well. However, such scenes only show direct illumination; we need at least three patches in a scene (including the light source) to see how the renderer handles light reflecting between patches.

Figure 7.2 shows such a scene. A small light-source patch is reflecting light off a larger, angled patch, onto the ground patch. Although the reflecting patch has undergone substantial subdivision in response to the queries of the ground patch, we don’t see its subdivisions because the eyeball is looking at the back of the reflecting patch.

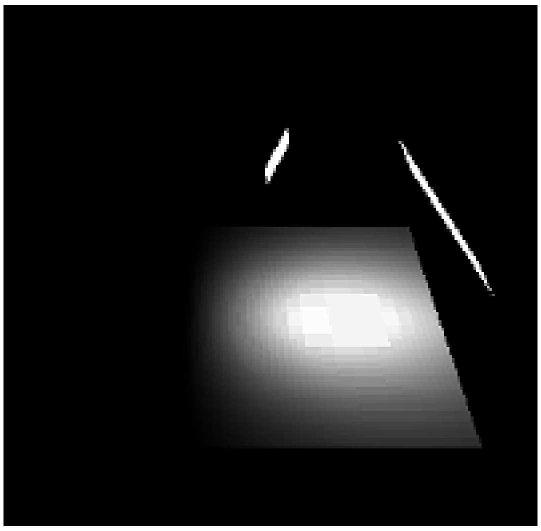


Figure 7.2 Reflection Example

7.3 The Cube Scene

There was a need for some kind of standard scene that we could use to test the renderer. It had to have a reasonable number of reflecting patches, so as to test multiple reflections, and yet be simple enough so that our memory constraints weren’t tried too badly. The scene that we chose for this task was the cube scene. Relevant points are:

- The scene is a cube with the top and front faces removed.
- The top face of the cube has been replaced with a light-source, which is a quarter-sized patch centred in the middle of the ceiling.
- The floor and walls of the cube all have a reflectivity of 0.3. The light source has a reflectivity of 0, and emits light at 9 radiosity units. One such radiosity unit is equivalent to the maximum intensity displayable on the screen.

It should also be noted that for all of the renderings mentioned in this chapter, excepting section 7.6, the alpha constant of section 5.2.6 was set at 4. Previous experience with the renderer had led us to believe that this gave good results[1].

Figure 7.3 shows a sequence of six pictures extracted from an animation of the rendering of the cube scene. It starts off with completely red walls, indicating the large amount of error in their radiosity estimates. As subdivision and questioning improves the scene in the following frames, the red colour disappears as the estimates become more accurate. The numerical PFB was used for this particular animation; the final picture has an error of 23.2%.

[1]This number was largely arrived at through observations of how well a scene “seemed” to be converging for different values of alpha. Section 7.6 investigates its effects in a more rigorous fashion.

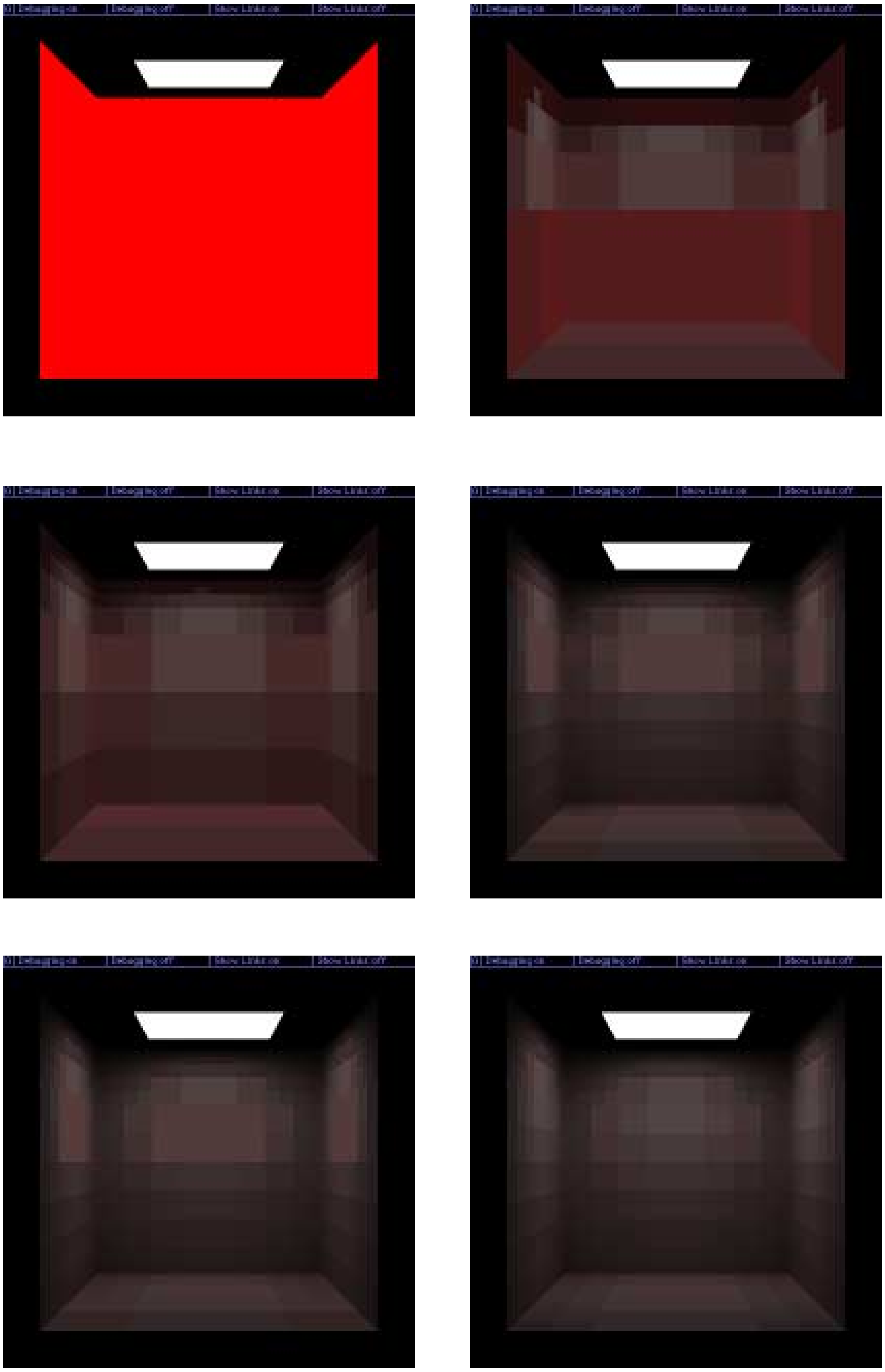


Figure 7.3 Rendering the cube scene.

7.3 Profiling the PFB Methods

Our first task in investigating the renderer was to compare the results obtained from using the three different PFB methods we had developed. Statistics for the cube scene were gathered for all three methods, and the results can be seen in figures 7.4 and 7.5.

The first graph plots error vs. subdivisions. The number of subdivisions indicates how many times a patch has subdivided since the start of rendering, and hence is an indicator of the total number of patches in the scene, as every time such a subdivision occurs four new patches are created. It is also an indicator of the amount of memory used by the algorithm, as patch objects are the major consumer of space. Thus the graph gives us an indication of the "storage efficiency" of the algorithm; the smaller the number of subdivisions required to reach a certain level of accuracy, the smaller its storage requirements.

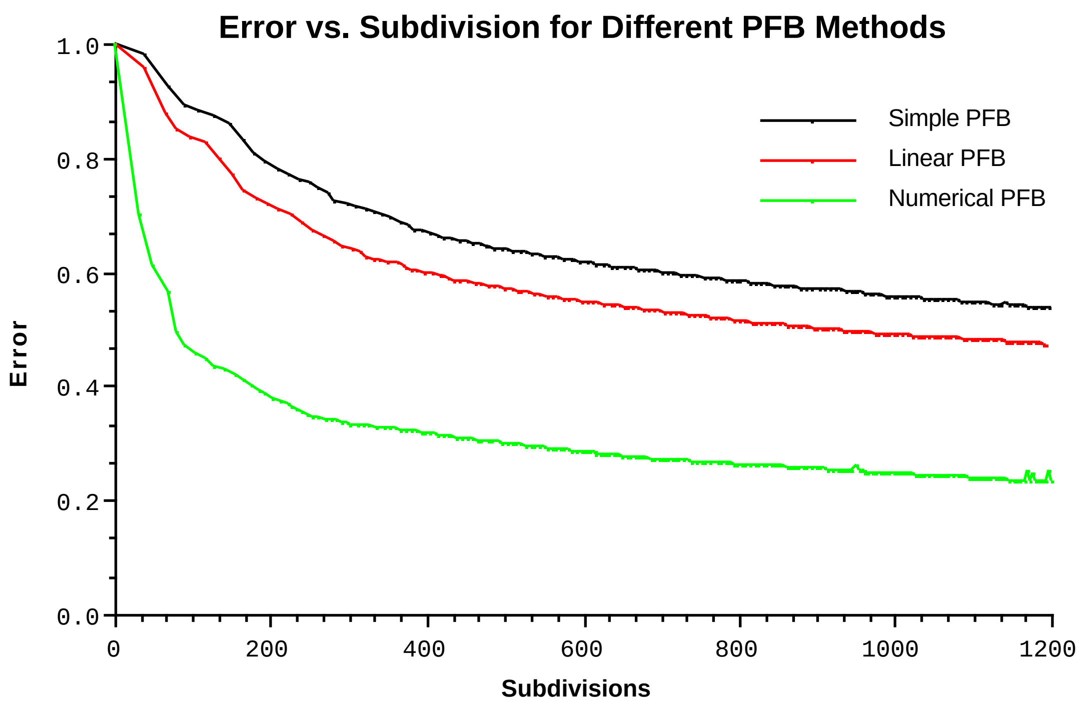


Figure 7.4 Storage efficiency of the various PFB methods

From the graph we can see that the numerical PFB outperforms the other two PFB methods by a considerable amount, and that the linear PFB outperforms the simple PFB. It is noticeable that the performance difference comes mainly in the first 400 patches or so; after that the error decreases at much the same rate for all methods.

The second graph plots error against time, showing much the same results as the first graph. While we expected the extra precision of the numerical PFB to greatly decrease the number of patches needed for a given level of accuracy, we did not expect a similar performance gap for running time. It was thought that the numerical PFB would take longer to calculate its estimates than the analytical methods, which appears not to be the case for this graph. It must be noted, however, that the configuration of the cube scene does mean that the seed-picking algorithm of section 4.5.3 often picks a starting point close to the maximum. This could be the reason for its better-than-expected performance.

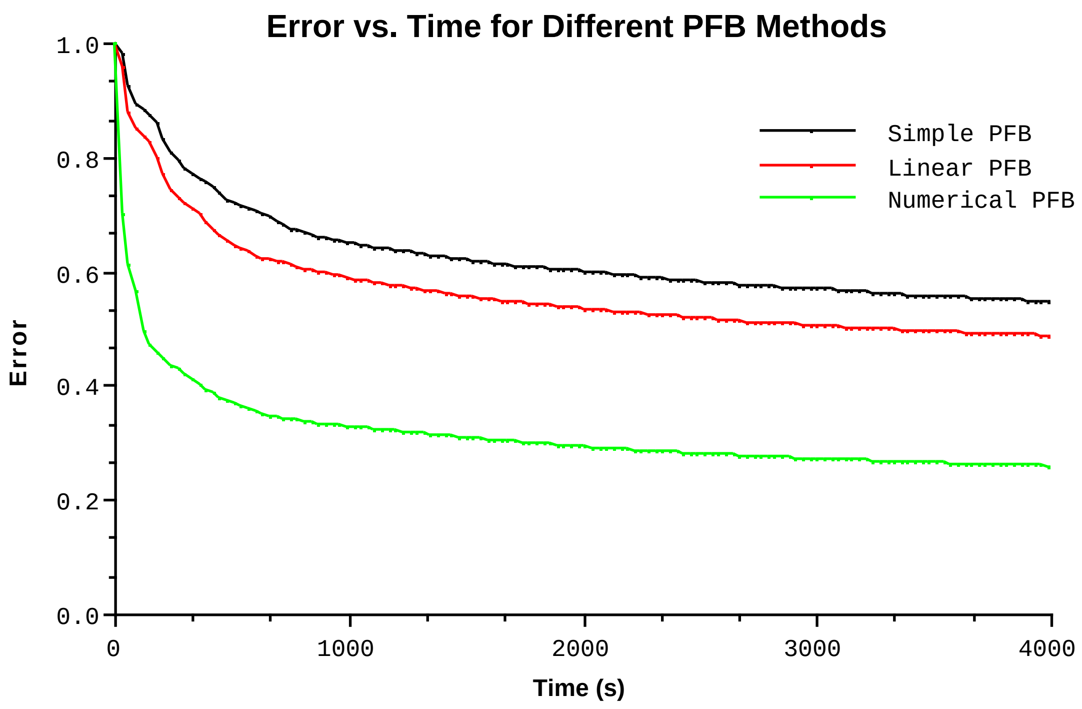


Figure 7.5 Time efficiency of the various PFB methods

7.4 Profiling the Renderer

Given the much better levels of error the numerical PFB gives us for the cube scene, it is the obvious choice of method for profiling the performance of the rendering algorithm itself. The data gathered from the numerical PFB run was further analysed, first to find the form of the relationships between error and time and subdivisions, and then to investigate the pattern of subdivision during rendering.

7.4.1 Error Relationships

The connection between error and the number of subdivisions in the scene was found reasonably easily. A log-log graph of the relationship was plotted, as in figure 7.6, and it was found that the data settled into a straight line after a few hundred patches, indicating a power-law relationship. The graph shows a line fitted to the data by the least-squares process, and gives the function indicated by this line. We can invert this to find that

$$\text{subdivisions} = 6.37 / (\text{error}^{3.61})$$

It is interesting to note that, according to this relationship, getting a result accurate to 5% would require some 300,000 patches. We would certainly hope that this is not the case! Given the huge improvements gained by the use of the numerical PFB over the other PFB methods, we suspect that further tuning of the algorithm should reduce this figure somewhat. A similar analysis for the simple patch factor method, for instance, shows that about 140 million patches would be needed to reach a 5% level of error. We also feel that there are a number of savings that can be made in the storage of patch factors themselves. Section 8.3 discusses some possible ways of achieving both of these goals.

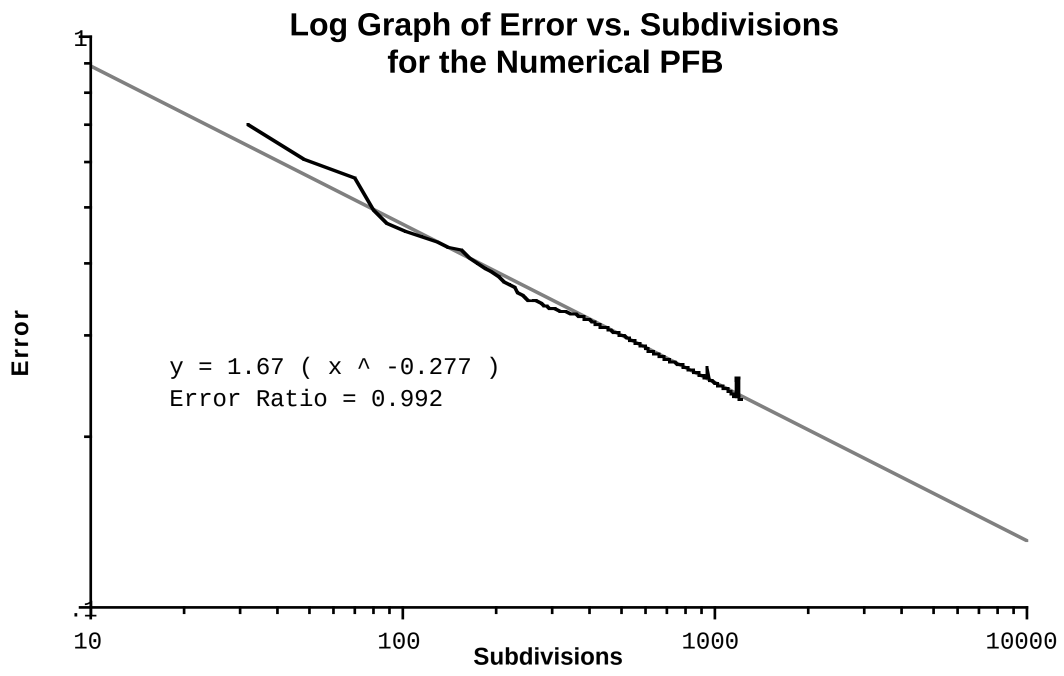


Figure 7.6 Relating error to storage

The graph in figure 7.7 shows a similar result for the relationship between error and time elapsed. Again, a power-law form of relationship is indicated, and inverting it gives us

$$\text{time} = 2.67 / (\text{error}^{5.38}) \text{ seconds.}$$

Using this relationship, we find that it would take 300-odd days to reach a 5% level of accuracy!

Both of these relations point to a method for comparing the efficiencies of different renderers with respect to space and time. If they are all given identical scenes to render, and the error of the picture produced is profiled over time or per unit of storage, the orders of the resulting power-law relations can be compared. The smaller the exponent of the error term, the better the renderer. (Assuming that the relation is written in a form similar to those above.)

Obviously the exponents in the time and space relations are affected by both the scene's complexity and the limitations of the renderer involved. It would be useful if we could find an analytical lower bound for the renderer's contribution, assuming some kind of "ideal" renderer. Not only would this give us some insight into the rendering process, but we would then have a metric for the complexity of a scene.

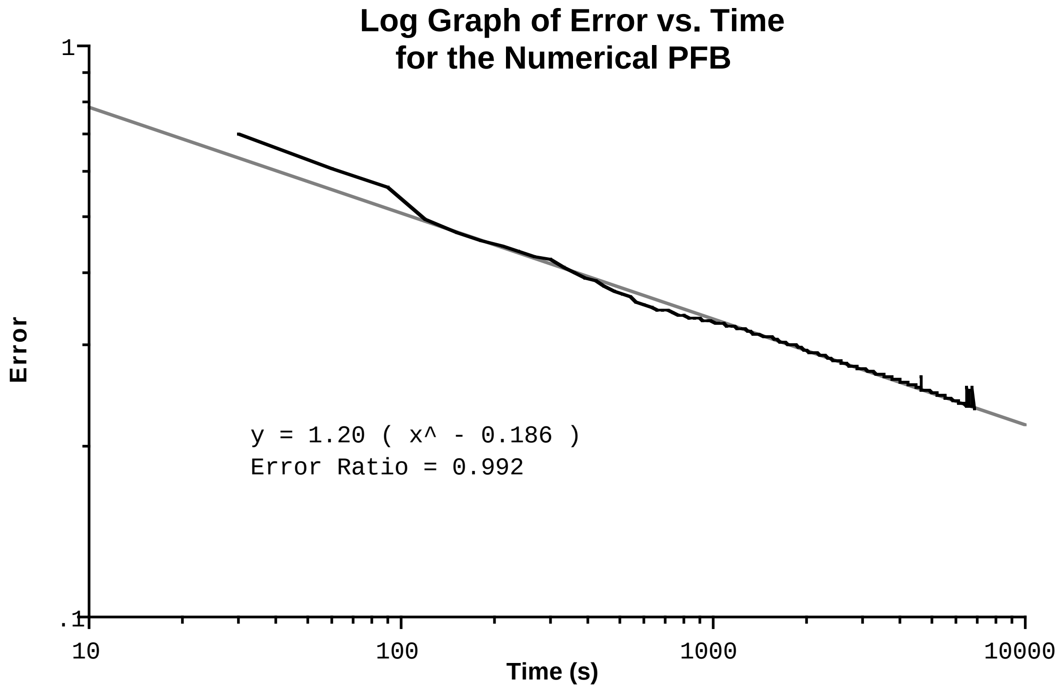


Figure 7.7 Relating error to running time

This could also help in establishing the order of the running time of the algorithm. Normally we give a running time in terms of the size of the input, this being a good indicator of the amount of work that the algorithm will have to do. Thus the temptation in patch-based rendering systems is to measure the performance of a renderer with respect to the number of patches in a scene. Unfortunately the idea of a patch is ill-defined; it doesn't sit well with the concept of curved surfaces, and is very weakly related to scene complexity. For instance, for any given scene we could increase the number of patches in it by repeatedly subdividing one of the patches already there, and this would not increase the complexity of the light flowing through the scene at all. On the other hand, the placing of one patch directly in front of a light source would drastically affect the scene.

If we had some way of calculating a scene's complexity, we could use this instead as an indicator of the size of the input to the rendering algorithm, allowing a more rigorous approach to the measurement of the efficiency of renderers. Such a concept would also be useful in establishing a formal theory of rendering.

7.4.2 Investigating Subdivision

There are several subdivision issues that are worthy of investigation. The first is the relationship between the actual number of subdivisions that take place, and the number of times an object decides to subdivide and pass back its children. This is important because it is an indicator of the usefulness of reusable subdivision. Because a patch always subdivides in a set way in the fuzzy rendering system, those same subdivisions can be passed back to several questioners. If we tried to optimise the subdivision for individual questioners, they would each need a different set of subdivisions, and we would lose this reusability.

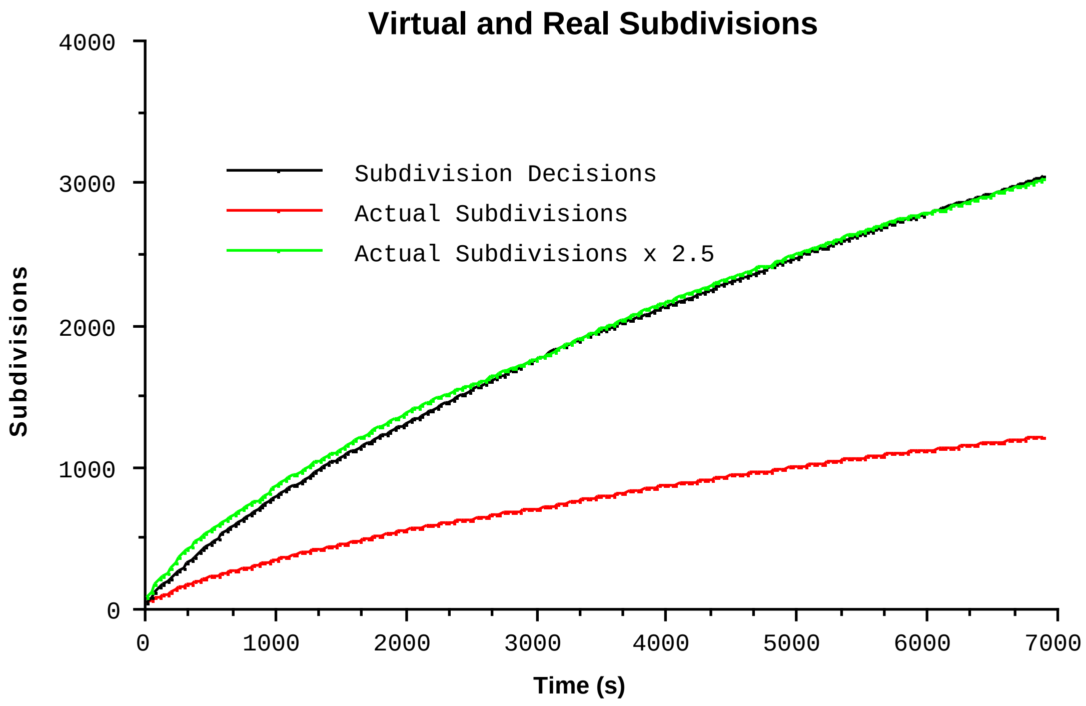


Figure 7.8 Effectiveness of reusable subdivision

The graph in figure 7.8 shows that the two quantities have a fairly predictable relationship. Their ratio starts at around 1.2 and quickly climbs towards around 2.5, where it appears to stabilise. Thus, as a rule-of-thumb, for every subdivision that occurs we save another one and a half further subdivisions[1]. In the future we can use this figure as a rough guide when considering whether a particular non-reusable subdivision scheme could be justified.

The second issue we look at is that of how subdivision is stratified. We can tag subdivisions by how far down a line of enquiry they occur. A level one subdivision would occur as soon as the eyeball questioned a patch. A level two subdivision would involve the eyeball asking one patch, which then asks another before subdivision takes place.

Figure 7.9 shows a graph of the subdivisions occuring at different levels in the cube scene. We see that most subdivision takes place at level 2, and comparatively few subdivisions at level 3. One subdivision at level 4 was recorded, but it has not been reproduced on the graph for reasons of clarity. We can use such a graph as an indication of the amount of reflection in the scene; if we increased the reflectance of the patches, for instance, the distribution of subdivisions would be spread out over another level or two.

[1]It should be mentioned that this figure is specific to the cube scene. We expect that the number of subdivisions saved would increase for more complex and reflective scenes.

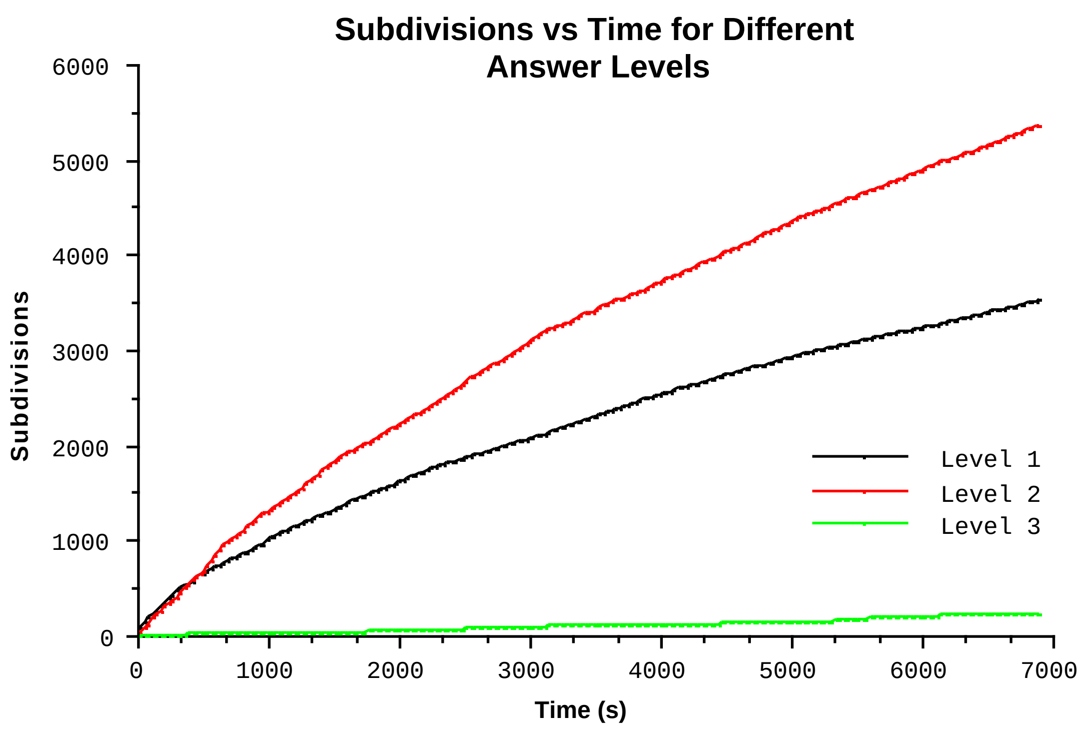


Figure 7.9 Varying the reflectivity of the cube scene

7.5 Convergence for Different Reflectances

An important question is how the reflectance of the scene effects error and rendering time. To investigate this, the cube scene was rendered with a reflectance (rho) of 0.1, 0.2, 0.3, 0.4 and 0.5. Figure 7.10 shows the results.

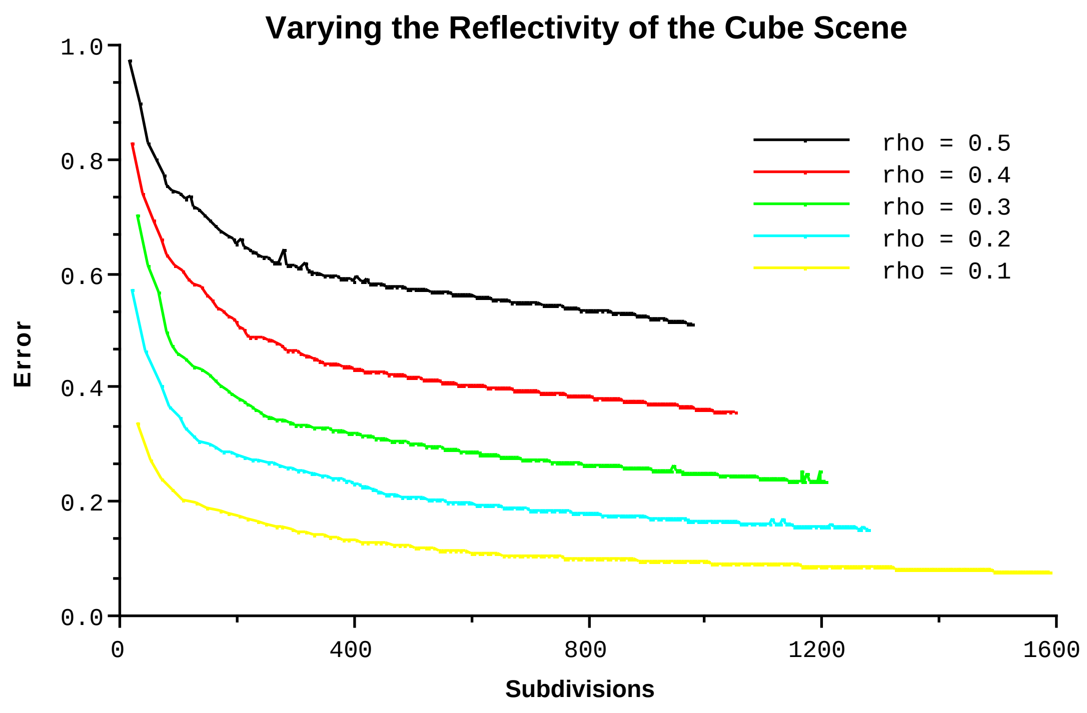


Figure 7.10 Subdivisions by reflectance

Obviously we expect that the amount of subdivisions necessary to reach a given level of accuracy will increase for more reflective scenes. In such scenes light bounces between the walls for a longer time before it peters out, and thus the average length of a line of enquiry will increase, and consequently the depth of subdivision. This turns out to be the case; when the walls have a reflectance of 0.1, a 20% error level is reached fairly quickly, whereas when this reflectance rises to 0.5, a 50% level of error cannot be reached before memory runs out.

An interesting feature of the graph is that the amount of subdivisions each run managed before running out of memory decreased as the reflectance of the scene was increased. We hypothesise that at higher reflectances patches had to keep track of a larger number of other subdivisions, increasing the average size of the contributing patches list, and thus the average size of a patch object.

7.6 Optimising Alpha

We mentioned in chapter 5 that the alpha constant of the *FindLargestErrorContributor* algorithm should be experimented with in order to find an optimum value for it. In this section we attempt to find such an optimum value.

The cube scene was plotted for a number of different values of alpha. The graph in figure 7.11 shows the effect that these values had on the error versus subdivisions trade-off. We see that as alpha is increased, the performance of the algorithm gets better and better, until above alpha = 4 it seems to reach an optimal level. These results suggest that we should try to maximise alpha, at least until it is over five; at this point any further increase is largely ineffective.

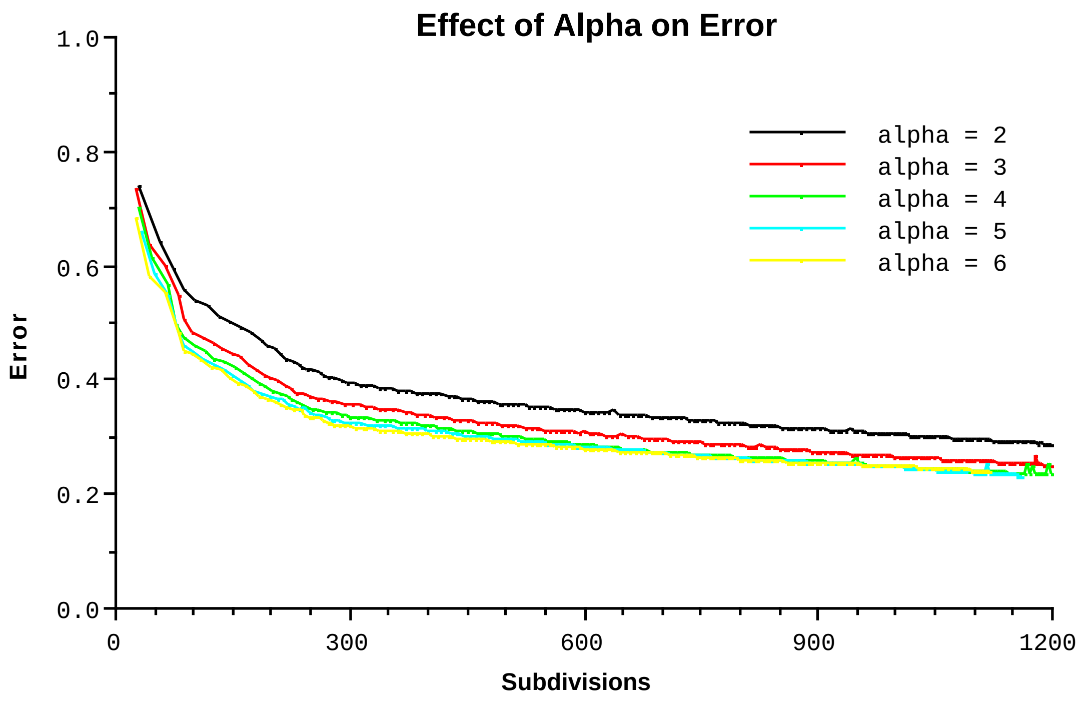


Figure 7.11 Effect of alpha on error

We then look at the effect of alpha on how the number of subdivisions increases with time. We might expect that as alpha is increased, and therefore lessens the likelihood of patches subdividing, the rate at which subdivisions are produced will drop off. The graph in figure 7.12 confirms that this is so.

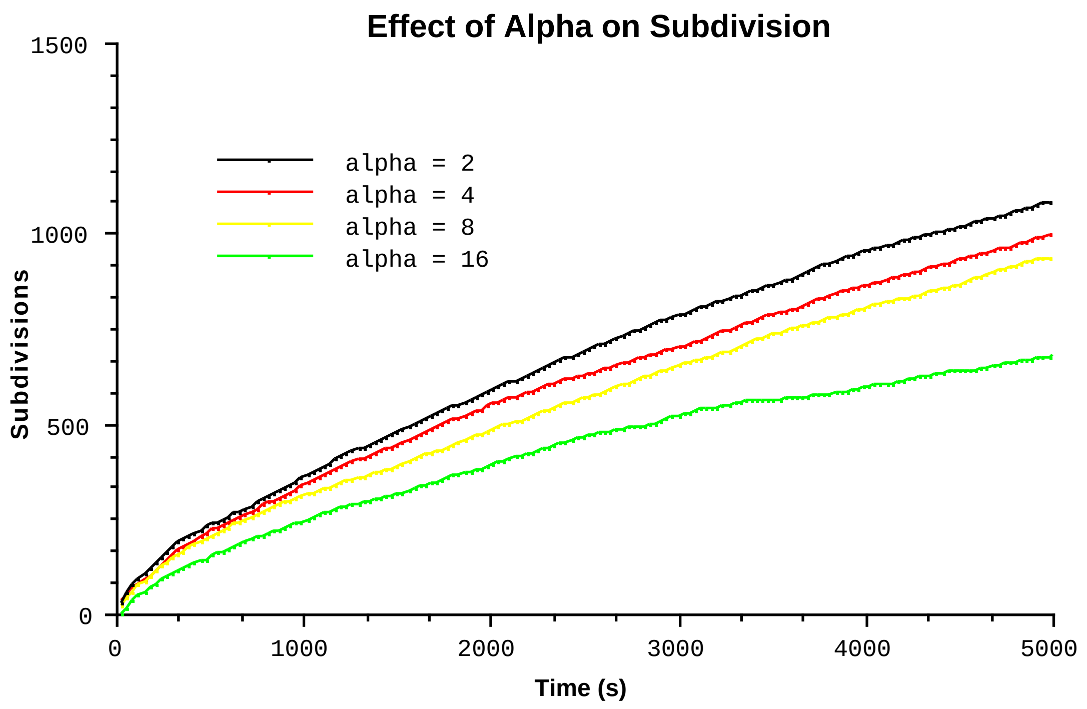


Figure 7.12 Effect of alpha on subdivision

All other things being equal, we would like the rate of subdivision to be as high as possible, as the error level of a scene increases with the number of subdivisions. Thus on the evidence of this graph we would like to minimise alpha. Taking the results of the first graph into account, we see that an alpha value of around five or six would be optimal for this particular scene. Certainly our earlier “guestimate” of four was not too far off the mark.

7.7 Conclusions

A fair amount of thought was put into understanding why the numerical PFB so outperforms the other two PFB methods. We reached the conclusion that it was partly because towards the end of any line of enquiry, there is usually a fairly large patch; certainly early on this will almost always be one of the original scene patches. Any extra error in the estimate of the PFB for this patch will filter back down the line of enquiry, to the eyeball. This tends to badly affect the PFB techniques that have large errors in their estimated bounds for large patches, but improve their estimates as the size of a patch decreases, as with the linear and simple PFB methods.

The problem could be partially alleviated by the use of a parent-update mechanism. When the children of a patch that has subdivided get better answers, they could pass these answers back up to their parent, which would then combine them to update its own answer. That way, any work done by the children to improve themselves would also be passed on to their parents, improving the error of the initial large patches.

We have two ways to proceed:

(1) Use the numerical PFB. This gives accurate answers at the beginning, so that an update later on it competes well with time for simple PFB.

(2) Use the linear/simple PFB with the update mechanism. Such a combination might be able to compete somewhat better with (1).

Our current sympathies lie with method (1) because of its conceptual simplicity, and because it would be easier to extend to surfaces other than rectangular patches. At some stage, though, the update mechanism should be implemented in order to find how great an effect it has.

One surprising aspect of the images produced by the renderer is how good they look, even when they have an error of 50% or so. Indeed, Gouraud-shading the images would produce even better results. Possible reasons for this are:

- Our error estimate is an upper bound; it simply means that the picture is guaranteed to be accurate to at least that level. The majority of the picture will in fact have much less error than this.

- The eye is not that good a judge of the actual error in the illumination of a scene. If an accurate picture and a picture accurate to 10% were shown side by side, it would be easy to pick the differences. However, if the 10% picture is viewed by itself, most people would probably reach the conclusion that it was already perfectly accurate.

Certainly it would be interesting to see if this observation still holds for specular scenes.

Chapter 8

Conclusions

8.1 Introduction

In previous chapters we have presented the research that has been done into a number of aspects of fuzzy-rendering. In this chapter we consider the foundations of that research and the goals that were set for this thesis. The achievements of the thesis are presented, and we take a brief look at some of the implications. Finally, we look at possible future areas of research, commenting on what needs to be done in these areas.

8.2 Achievements

The major achievement of this thesis has been the development of a three-dimensional renderer capable of calculating the illumination of diffuse scenes to within specified error bounds, using a methodology we have named “fuzzy rendering”. The production of this renderer has involved research into a number of areas, including the transfer of light between surfaces, and the relationship between these error bounds and the scene itself.

The starting point of the research was the work done by Dr. Richard Lobb and Guyon Roche on a new rendering system. This work used object-oriented concepts to develop the idea of a system of “intelligent” scene objects, which would negotiate with each other in order to determine the illumination of the scene. An important part of this scheme was the use of absolute error bounds on all metrics in the illumination calculations. It was envisaged that any scene object would be able to guarantee that the illumination answers it gave would lie within such bounds. Guyon implemented a two-dimensional prototype of the scheme in his Masters thesis.

At the beginning of this thesis, a number of goals were set, subject to time constraints:

- Investigate the three-dimensional implications of the algorithm.

- Design and implement a (limited) three-dimensional version of the renderer, making any assumptions necessary for this be feasible in a Masters thesis. A tentative goal of rendering just a few patches with trivial reflectance characteristics was set.
- Establish the dynamics of the rendering process, namely find what controls convergence, and how the system behaves over time.
- Investigate the role of object-oriented concepts in the method. See whether it would be better implemented under a non object-oriented paradigm.

As things have turned out, we have mainly concentrated on the first three of these goals. A number of new techniques have been developed, and some of the previous work done has been extended. Specifically:

- The concepts and implications of the fuzzy rendering method have been more fully explored than before, and a solid frame-work for the technique has been established. A new scheme for controlling the interaction between scene objects has been developed.
- The notion of a "patch factor" which measures the transfer of light between a diffuse surface and another surface has been developed. This is related to the form-factor of the radiosity algorithm, but applies to a specific point on the receiving surface. A methodology for calculating this patch factor has been developed, and a formula for polygonal surfaces presented.
- Research has been done into bounding the light flux incident at a point on a surface over some area of that surface. Both analytical and numerical search techniques for solving this problem have been developed.
- A link between the perceived error in a picture and the actual error in its intensities has been established.

A renderer been designed using the results of this research. It assumes polygonal, diffuse patches, and ignores the hidden surface problem. The renderer was implemented, along with a number of mechanisms for investigating the system. Its performance has been investigated, and relationships found between factors such as the subdivision of patch objects and error in a picture.

The most important aspect of the research work being done on fuzzy rendering is that it is leading towards a truly general, adaptive global-illumination algorithm. Whereas methods developed so far can produce quite realistic results, they make simplifications to the global-illumination problem, and often have to be hand-tuned in order to produce better results. The technique also has implications for a number of scientific endeavours, where the guaranteed accuracy of a picture might be important.

In spirit, the method has the most in common with radiosity and specular radiosity algorithms, in that it tries to establish a model of light flow in the scene, and then refine that model until it is good enough. However it differs in its approach to the subdivision of the patch mesh, and the fact that this subdivision and the solution of the patch system takes place at the same time. It should also be noted that it is a completely object-space algorithm, and as such will avoid the aliasing that can cause problems for conventional renderers.

8.3 Directions for Further Research

8.3.1 The Fuzzy Visible Surface Problem

The problem of introducing obscuring surfaces to the fuzzy rendering technique was discussed in chapter 2. As was pointed out there, we are confident that if a method for classifying the visibility of other patches from any point on a patch can be found, it can be used to integrate the idea of hidden surfaces into the renderer. Such a method would classify patches according to whether they were:

- Always obscured.
- Sometimes obscured
- Never obscured.

This problem has much in common with that of determining the penumbra and anti-penumbra of the shadows cast by an area light source. Several papers on this problem have been published recently ([Tell92] and [Chin92], for instance) and it seems likely that the ideas they have introduced could prove most useful in this area.

8.3.2 Specularity

Conceptually, introducing the ability to handle general reflectance functions to the renderer is simple; we just replace the radiosity value stored as a patch's answer with a distribution of light. However there are several issues that would have to be addressed. Firstly, a means of representing such a distribution would have to be decided on. The spherical-harmonics idea of [Sill91] looks to be the most promising approach, although we would use circular harmonics instead to fit in with our idea of the view circle.

Secondly, a mechanism would have to be established to calculate the contribution that one patch's emitted light distribution makes to the light emitted by another patch. The work done on specular radiosity seems to indicate that much of this can be achieved by extending the concepts we have developed for diffuse patches over an array of different directions. This would require the introduction of "distributed" patch factors and reflectivities, and methods for calculating these distributions. Certainly this area should prove a challenging one.

8.3.3 Linear Bounds

One of the problems of the current renderer is the number of patches it needs in order to produce a reasonably accurate picture. One of the reasons for this is that the simple bounds used for most of the metrics of a patch don’t handle rapid changes in those metrics well. This problem could be alleviated by using higher-order bounds, so that fewer patches are needed to represent a given distribution of radiosity.

This could be achieved by extending the linear-type bounds introduced with the linear PFB method to all error bounds kept by the renderer. Conceptually, instead of bounding scalar quantities over a patch with a single range, we would instead be bounding them with planes. An advantage of this scheme is that it could be integrated particularly well with Gouraud shading; obviously it would be desirable to improve on the flat-shading currently used in the renderer.

8.3.4 Structure Trimming

Obviously one of the greatest drawbacks of the fuzzy-rendering scheme is the amount of memory it requires for even reasonably simple scenes. In this it shares a lot in common with specular radiosity schemes, in that the current state of the scene is stored, and slowly updated as a solution is approached. Ray-tracing is at the other extreme. It only ever stores information about the current ray being traced. Once that trace is over, all the storage concerned is thrown away. While this is a great advantage storage-wise, it is a disadvantage in terms of speed, as any new ray cast is unable to use any information gathered by previous rays. Most methods for speeding up ray-tracing trade off some space for better time.

There are a number of solutions to our space problem. The first of these is to trim the patch-object “tree” belonging to any original scene patch. As the rendering progresses, queries from other patch objects tend to move downwards from this patch to its subdivisions, and then on to the children of those subdivisions. Thus as soon as the patch at the top of the tree is no longer of interest to the other patches, it can be deleted to free up space.

A similar idea could be applied to the lists of references to other patches that each patch must keep. Those references to patches that have no effect (or perhaps a minimal one) on the answer of a particular patch can obviously be trimmed.

Finally, if space still proves to be a problem, some kind of least-used-object disposal system could be used. When the system runs out of memory, patch objects that are unlikely to be asked further questions could be trimmed away. However, if they were needed again, they would have to be regenerated in some manner; it is not immediately clear how this could be handled.

8.3.5 Moving Away From Polygons.

An issue that has gone largely unconsidered is how to apply the light-transfer techniques developed for polygonal surfaces to non-polygonal ones. One of the biggest problems would be handling the effects of curved surfaces. There seems no reason why our method for calculating patch factors can't be extended to handle these, but their non-planar nature is not really suited to our current methods for finding patch-factor bounds. From our experiences with finding these bounds analytically, we would expect that producing similar bounds for curved surfaces would be challenging. For this reason the Numerical PFB seems best suited to such a task.

8.3.7 Parallelisation

The object-oriented nature of the fuzzy rendering algorithm makes its parallelisation a slightly easier task. This is because the message-passing model of object-oriented systems lends itself to the distribution of objects over several processors. Indeed, such distributed object-oriented systems are an active area of research, and it seems likely that some of their results could be applied to the patch-object system, without having to change the top-level design of the renderer.

The major avenue of investigation in developing such a distributed system would be how objects handle the implications of working asynchronously. To benefit from parallelism, a patch object would have to find something to do while the patch it has questioned thinks about its answer. Also, the current model of the system works by considering one line of enquiry at a time. In a parallel system we would want to extend this so that a number of such enquiries are going on at any one time. Thus an object could pose several questions to other objects in order to improve its answer, dealing with the answers as they came back.

8.4 Summary

The various parts of the fuzzy-rendering algorithm have proved a most stimulating area of research. They have required coming to grips with several areas of Computer Graphics, and initially a great deal of reading was needed in order to bring myself up to date with the current research in the field of high-quality image rendering. Certainly my understanding of the field has improved vastly over the last year.

When this thesis began, it was still largely unknown whether the technique would be feasible in three dimensions; at the end of it we have demonstrated that it is. Hopefully the work done has established a basic framework for fuzzy rendering which can be built on in the future, in order to produce a general, error-driven rendering system.

Appendix A

More on Bounds

In this appendix we provide proofs for some of the statements that were made in Chapter 3. These mainly concern the finding of analytical bounds for the patch factor equation. We also discuss how the procedure for bounding a quadratic function of two variables works.

A.1 Proofs

A.1.1 The Area Term

While deriving equation 3.30 is not unduly difficult, it is rather tedious. We wish to find ||p' x q'||^2, that is,

(A.1) $$\left\| \left(p + se_x + te_y\right) \times \left(q + se_x + te_y\right) \right\|^2 .$$

We substitute $u = se_x + te_y$, giving

(A.2) $$\begin{aligned} \|(p+u) \times (q+u)\|^2 &= \|(p\times q) + u\times(q-p)\|^2 \\ &= \|p\times q\|^2 + \|u\times a\|^2 + 2(p\times q).\left(u\times(q-p)\right) \end{aligned}$$

For the last term in this equation, we use the cross-product rules,

(A.3) $$\begin{aligned} (A\times B).C &= A.(B\times C), \quad and \\ A\times(B\times C) &= (C\cdot A)B - (B\cdot A)C, \end{aligned}$$

to show that

(A.4) $$\begin{aligned} (p\times q).\left(u\times(q-p)\right) &= u.\left((q-p)\times(p\times q)\right), \\ &= u.\left(q\left(p^2 - p.q\right) + p\left(q^2 - p.q\right)\right), \\ &= s\left(q_x\left(p^2 - p.q\right) + p_x\left(q^2 - p.q\right)\right) + \\ &\quad\; t\left(q_y\left(p^2 - p.q\right) + p_y\left(q^2 - p.q\right)\right). \end{aligned}$$

For the second term, we find that

$$\begin{aligned}\|u\times(q-p)\| &= u^2a^2-(u.a)^2,\\ &= \left(s^2+t^2\right)a^2-\left(sa_x+ta_y\right),\\ &= s^2\left(a^2-a_x{}^2\right)+t^2\left(a^2-a_x{}^2\right)-2st\left(a_xa_y\right),\end{aligned} \tag{A.5}$$

Putting the both of these results together with the first term of equation A.2 gives us equation 3.30.

A.1.2 Bounding θ/A

In section 3.5.2, it was stated that for the function

$$z=\frac{1}{f}Arctan\left(\frac{f}{g}\right), \tag{A.6}$$

both *dz/df* and *dz/dg* were never positive. Because we are using this function as a template for equation 3.28, we can make the following assumptions:

Arctan(x) >= 0.[1]

f >= 0.

Evaluating *dz/dg* gives us

$$\frac{dz}{dg} = \frac{-1}{f^2+g^2}. \tag{A.7}$$

which is obviously always negative. For *dz/df* we have

$$\frac{dz}{df} = \frac{-1}{f^2}Arc\tan\left(\frac{f}{g}\right) + \frac{1}{fg}\frac{g^2}{f^2+g^2} . \tag{A.8}$$

[1]In our work we assume that the arctan function returns a value from 0 to π, although many mathematical texts and computer languages instead assume that its range is [-π/2, π/2]. The two are forms are generally interchangeable, although care must be taken to keep track of which is being used.

If g =< 0 both of the terms of A.8 are also =< 0, and thus so is *dz/df*. The proof for g > 0 is a little more complex. It is equivalent to putting *y = f²(dz/df)*,[1] and proving that *y =< 0*. If we also substitute *x = f/g* we have

(A.9)
$$\begin{aligned} y &= -arctan(x) + \frac{x}{1+x^2}, \quad thus \\ \frac{dy}{dx} &= -\frac{1}{1+x^2} + \frac{1-x^2}{\left(1+x^2\right)^2}, \\ &= -\frac{2x^2}{\left(1+x^2\right)^2}, \end{aligned}$$

The gradient of y is always =< 0, and as y(0) = 0, we have that y =< 0 for x > 0, which will be the case if g > 0.

A.1.3 Finding R_{max}

In section 3.5.3 it was stated that equation 3.36 gave a limit for P/A as the viewpoint moved towards one of the vertices of the edge. It was also stated that this was an upper bound for |P/A|. We now prove both these claims.

We can pick any coordinate system we like for the plane of the viewing surface (the x-y plane in figure 3.9). For convenience we pick the one shown in figure A.1, where the edge is aligned with the y-axis, and has the vertex which is closest to the surface directly over the origin of the x-y plane.

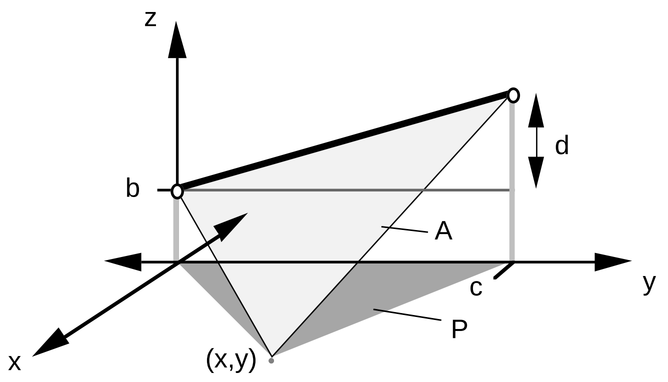


Figure A.1 Considering an edge

We then derive the formula for P and A. Finding the projected area is straightforward:

(A.10)
$$P = \tfrac{1}{2}ax.$$

[1]This will not be true if f = 0, but in such a case dz/df = 0 anyway.

For the actual area, we have

(A.11)
$$\begin{aligned} A &= \tfrac{1}{2}\left\|[x,y,-b]\times[0,c,d]\right\|, \\ &= \tfrac{1}{2}\left\|[yd-bc,-xd,xc]\right\|, \\ &= \tfrac{1}{2}\left(y^2d^2+x^2d^2+x^2c^2+b^2c^2-2ybcd\right)^{\frac{1}{2}}. \end{aligned}$$

We start by finding a bound for (P/A). First we look at the variation of (P/A) with respect to y. As only A depends on y, we can find the maximum for |P/A| by minimising A, which we do as follows:

(A.12)
$$\begin{aligned} \frac{dA^2}{dy} &= yd^2-bcd = 0 \quad for \ a \ minimum, \ thus \\ y &= \frac{bc}{d}. \end{aligned}$$

Substituting this back into A^2 gives

(A.13)
$$\begin{aligned} A^2_{Max} &= \tfrac{1}{2}x^2\left(c^2+d^2\right), \quad and \ thus \\ \left(\frac{P}{A}\right)_{Min} &= \frac{c}{\left(c^2+d^2\right)^{\frac{1}{2}}} = \sqrt{1-\frac{c^2}{c^2+d^2}}. \end{aligned}$$

Relating this back to the *a* vector of equation 3.36, we find that c^2 is $a_x^2 + a_y^2$ (the squared length of the x-y projection of *a*), and d is a_z, so that $c^2+d^2 = a^2$.

Now we consider the case where a vertex is touching the x-y plane, and we are moving the viewpoint towards that vertex. In such a case b = 0, and we want to find the limit of (P/A) as x and y -> 0. We have

(A.14)
$$\begin{aligned} \frac{P}{A} &= \frac{xa}{\left(y^2d^2+x^2d^2+x^2c^2\right)^{\frac{1}{2}}}, \\ &= \frac{a}{\left(\frac{y^2}{x^2}d^2+d^2+c^2\right)^{\frac{1}{2}}} \end{aligned}$$

The value of y/x depends on the direction we are approaching the vertex from. To maximise |P/A| we assume we are approaching along the line y = 0, so that y/x = 0, and we have the same result as A.13.

A.1.4 Bounding Theta

We write theta as a composite function z:

(A.15)
$$z = Arctan\left(\frac{f}{g}\right)$$

We then look at z’s partial derivatives. Firstly,

(A.16) $$\frac{dz}{dg} = \frac{-f}{f^2+g^2},$$

which is always negative. We also have

(A.17) $$\frac{dz}{df} = \frac{g}{f^2+g^2}.$$

Thus the sign of dz/df will depend on that of g. To find the maximum of arctan(F/G) for two ranges F and G, we observe that because of A.16, G.bottom will always give us the largest arctan value. We then look at the sign of G.bottom to see whether z will be monotonically increasing or decreasing with respect to f for that value of g. If it is positive, we choose F.top, otherwise we choose F.bottom. Finding the minimum can be approached in much the same way. These results are summarised in the table in section 3.5.3.

A.2 Bounding a Quadratic in Two Variables

The *MixedQuadRange* procedure of the *URange* unit in appendix B implements a method for finding bounds for a quadratic function (in two variables) over a square patch. In this section we outline how it carries this out.

Such a “mixed” quadratic has the form,

(A.18) $$z = a + bs + ct + dst + es^2 + ft^2.$$

There are a couple of features of this function that make it easier to handle. Firstly, it has only one minimum, as we shall see[1], and extends up towards infinity as we move away from that minimum, much like a one-variable quadratic. Secondly, it is separable; along any line in s and t, z is a standard quadratic. Thus any cross-section of the function will produce another function with only one minimum, and, as in section 4.5.1, we can conclude that the maximum of the function over a convex patch will always fall at one of its vertices.

The *MixedQuadRange* procedure in turn uses the *QuadRange* procedure, which finds the bounds of a simple quadratic over a specified range. This procedure works by evaluating the value of the quadratic at either end of this range, as well as finding where the local minimum of the quadratic lies. If it lies outside the range, the endpoint with the smallest value is taken to be the minimum, and the endpoint with the largest value the maximum. If it lies inside the range, then it is the minimum, rather than one of the endpoints.

The *MixedQuadRange* procedure works in much the same way. As each of its edges are lines in s and t, and the function along them is a quadratic, it can call the *QuadRange* procedure to get bounds for

[1]The *MixedQuadRange* function assumes that e and f are positive, which is true for all cases we need it for.

each of them. These can then be merged to find overall bounds for all of the edges. The procedure then finds where the local minimum of the mixed quadratic lies; if it is outside the patch, it returns the edge bounds, otherwise, it is taken to be the minimum, and the top of the edge bounds the maximum.

We now consider how to find the local minima. The minimum for a quadratic is easy enough:

$$\begin{aligned} z &= a + bx + cx^2, \\ \frac{dz}{dx} &= b + 2cx = 0, \quad \textit{and thus} \\ x &= \frac{-b}{2c}. \end{aligned} \tag{A.19}$$

Finding the local minimum of the mixed quadratic involves firstly finding minima for both s and t:

$$\begin{aligned} \frac{dz}{ds} &= b + dt + 2es = 0, \qquad (1) \\ \frac{dz}{dt} &= c + ds + 2ft = 0, \qquad (2) \end{aligned} \tag{A.20}$$

We can then solve (1) and (2) simultaneously to find its coordinates:

$$s = \frac{2bf - dc}{d^2 - 4fe}, \qquad t = \frac{2ce - db}{d^2 - 4fe}. \tag{A.21}$$

Appendix B

Code

B.1 Source Code for the Fuzzy Renderer

The following pages contain the Pascal units that make up the fuzzy renderer. These units can be divided into three groups:

[1]	**The Renderer's Code**	**(See chapter 5)**
	MTestbed	
	UPinholeCamera	(The "eyeball" object)
	UWorld	
	UDiffusePatch	(Together, these units implement...)
	UPatch	(... the patch object)
	UGlobals	(Handles much of the monitoring code)
[2]	**PFB Functions Code**	**(See chapters 3 and 4)**
	UNumericalPFB	
	UPatchFactorBounds	(Contains the simple and linear PFB functions)
	UEdgeRanges	(Contains edge area function)
	URanges	(Contains operations for the *Range* type)
[3]	**Display Code**	**(See chapter 6)**
	UQDAnimationDisplay	(Handles stop-frame animation)
	UQDVectorDisplay	(Provides the vector display)
	UFuzzyFace	(The remaining units implement the fuzzy display)
	UFuzzySurface	
	UFuzzyShadedPoint	

References

Those references marked with a ° are not in the possession of the author, but are referred to in survey books or papers, such as [Suth74], [Fole91] or [Hall89].

[Aman84]° J. Amanatides, “Ray Tracing with Cones”, Computer Graphics, Vol. 18, No. 3, July 1984, pp. 129-135.

[Appe67]° A. Appel, “The Notion of Quantitative Invisibility and the Machine Rendering of Solids”, Proceedings of the ACM National Conference, Thompson Books, Washington DC, 1967, pp. 387-393.

[Appe68]° A. Appel, “Some Techniques for Shading Machine Renderings of Solids”, Proceedings of the Spring Joint Computer Conference, 1968, pp. 37-45.

[Arvo86]° J. Arvo, “Backward Ray Tracing”, Developments in Ray Tracing, Course Notes 12 for SIGGraph ‘86, Dallas, Texas, August 1986.

[Athe78]° P. Atherton, K. Weiler, D. Greenberg, “Polygon Shadow Generation”, Computer Graphics, Vol. 12, No. 3, August 1978, pp. 275-281.

[Baum89] D. Baum, H. Rushmeier, J. Winget, “Improving Radiosity Solutions Through the Use of Analytically-Determined Form Factors”, Computer Graphics, Vol. 23, No. 3, July 1989, pp. 325-334.

[Baum91] D. Baum, S. Mann, K. Smith, J. Winget, “Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions”, Computer Graphics, Vol. 25, No. 4, July 1991, pp. 51-59.

[Blin76]° J. Blinn, M. Newell, “Texture and Reflection in Computer Generated Images”, Communications of the ACM, Vol. 19, No. 10, October 1976, pp. 542-547.

[Blin77] J. Blinn, “Models of Light Reflection for Computer Synthesized Pictures”, Computer Graphics, Vol. 11, No. 2, 1977, pp. 316-322.

[Bouk70b]°	W. Bouknight, K. Kelley, “An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources”, Proceedings of the Spring Joint Computer Conference, AFIPS Press, Montvale, New Jersey, 1970, pp. 1-10.
[Bouk70]°	W. Bouknight, “A Procedure for the Generation of Three-Dimensional Half-Toned Computer Graphics Presentations”, Communications of the ACM, Vol. 13, No. 9, September 1970, 527-536.
[Bres65]°	J. Bresenham, “Algorithm for Computer Control of a Digital Plotter”, IBM Systems Journal, Vol. 4, No. 1, 1965, pp. 25-30.
[Catm75]	E. Catmull, “Computer Display of Curved Surfaces”, Proc. IEEE Conf. on Computer Graphics, Pattern Recognition and Data Structures, May 1975, pp. 309-315.
[Chen91]	E. Chen, H. Rushmeier, G. Miller, D. Turner, “A Progressive Multi-Pass method for Global Illumination”, Computer Graphics, Vol. 25, No. 4, July 1991, pp. 165-174.
[Chin92]	N. Chin, S. Feiner, “Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees”, Proceedings, 1992 Symposium on Interactive Computer Graphics, Cambridge, MA, March 1992, pp 21-29.
[Cohe85]	M. Cohen, D. Greenberg, “The Hemicube: A Radiosity Solution for Complex Environments”, Computer Graphics, Vol. 19, No. 3, July 1985, pp. 31-40.
[Cohe86]	M. Cohen, D. Greenberg, D. Immel, P. Brock, “An Efficient Radiosity Approach for Realistic Image Synthesis”, IEEE Computer Graphics and Applications, Vol. 6, No. 3, March 1986, pp. 26-35.
[Cohe88]	M. Cohen, S. Chen, J. Wallace, D. Greenberg, “A Progressive Refinement Approach to Fast Radiosity Image Generation”, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 75-83.
[Cook82]°	R. Cook, K. Torrance, “A Reflectance Model for Computer Graphics”, Association for Computing Machinery, Transactions on Graphics, Vol. 1, No. 1, January 1982, pp. 7-24.
[Cook84]	R.L. Cook, T. Porter, L. Carpenter, “Distributed Ray Tracing”, Computer Graphics, Vol. 18, No. 3, July 1984, pp. 137-145.
[Crow77]°	F. Crow, “Shadow Algorithms for Computer Graphics”, Computer Graphics, Vol. 11, No. 2, 1977, pp. 242-247.

[Fole90] J. Foley, A. van Dam, S. Feiner, J. Hughes, “Computer Graphics, Principles and Practice”, second edition, Addison-Wesley, 1990.

[Fuch80]° H. Fuchs, Z. Kedem, B. Naylor, “On Visible Surface Generation by A Priori Tree Structures”, Computer Graphics Vol. 14, No. 3, July 1980, pp. 124-133.

[Fuji85]° K. Fujimura, T. Kunii, “A Hierarchical Space Indexing Method”, Computer Graphics: Visual Technology and Art, Proceedings of Computer Graphics Tokyo ‘85 Conference, Springer-Verlag, 1985, 21-34.

[Gali69]° R. Galimberti, U. Montanari, “An Algorithm for Hidden Line Elimination”, Communications of the ACM, Vol. 12, No. 4, April 1969, pp. 206-211.

[Glas84]° A. Glassner, “Space Subdivision for Fast Ray Tracing”, Computer Graphics, Vol. 20, No. 4, 1986, pp. 297-306.

[Gold71]° R. Goldstein, R. Nagel, “3-D Visual Simulation”, Simulation, Vol. 16, No. 1, January 1971, pp. 25-31.

[Gonz87] R. Gonzalez, P. Wintz, “Digital Image Processing”, second edition, Addison-Wesley, 1987.

[Gora84]° C. Goral, K. Torrance, D. Greenberg, B. Battaile, “Modelling the Interaction of Light between Diffuse Surfaces”, Computer Graphics, Vol. 18, No. 3, July 1984, pp. 213-222.

[Hall83]° R. Hall, D. Greenberg, “A Testbed for Realistic Image Synthesis”, IEEE Computer Graphics and Applications, Vol. 3, No. 8, November 1983, pp. 10-20.

[Hall88] R. Hall, “Illumination and Color in Computer Generated Imagery”, 1st edition, New York: Springer-Verlag, 1988.

[Hanr91] P. Hanrahan, D. Salzman, L. Aupperle, “A Rapid Hierarchical Radiosity Algorithm”, Computer Graphics, Vol. 25, No. 4, July 1991, pp. 197-206.

[Heck84]° P. Heckbert, P. Hanrahan, “Beam Tracing Polygonal Objects”, Computer Graphics, Vol. 18, No. 3, July 1984, pp. 119-127.

[Hott67]° H. Hottel, A. Sarofim, “Radiative Heat Transfer”, McGraw-Hill, New York, 1967.

[Imme86] D. Immel, M. Cohen, D. Greenberg, “A Radiosity Method For Non-Diffuse Environments”, Computer Graphics, Vol. 20, No. 4, August 1986, pp. 133-141.

[Kaji85]° J. Kajiya, “Anisotropic Reflection Models”, Computer Graphics, Vol. 19, No. 3, July 1985, pp. 15-21.

[Kaji86] J. Kajiya, “The Rendering Equation”, Computer Graphics, Vol. 20, No. 4, August 1986, pp. 143-150.

[Lisc92] D. Lischinski, F. Tampieri, D. Greenberg, “Discontinuity Meshing for Accurate Radiosity”, IEEE Computer Graphics and Applications, Vol. 12, No. 6, November 1992, pp. 25-39.

[Lout70]° P. Loutrel, “A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra”, IEEE Trans. on Computers, Vol. 19, No. 3, March 1970, 205-213.

[McCo82] G. McCormick, “Nonlinear Programming”, first edition, Wiley-Interscience, 1983.

[Newe72]° M. Newell, R. Newell, T. Sancha, “A Solution to the Hidden Surface Problem”, Proceedings of the ACM National Conference, 1972, pp. 443-450.

[Nish85]° T. Nishita, E. Nakamae, “Continous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection”, Computer Graphics, Vol. 19, No. 3, July 1985, pp. 23-30.

[Perl85] K. Perlin, “An Image Synthesizer”, Computer Graphics, Vol. 19, No. 3, July 1985, pp. 287-296.

[Phon75]° B. Phong, “Illumination for Computer Generated Pictures”, Communications of the ACM, Vol. 18, No. 8, 1975, pp. 311-317.

[Robe63]° L. Roberts, “Machine Perception of Three Dimensional Solids”, Lincoln Laboratory, TR 315, MIT, Cambridge, MA, May 1963.

[Roch91] G. Roche, “Image Rendering Without Point Sampling”, MSc. Thesis, Computer Science Department, University of Auckland, Auckland, 1992.

[Rush86]° H. Rushmeier, “Extending the Radiosity Method to Transmitting and Specularly Reflecting Surfaces”, M.S. Thesis, Mechanical Engineering Department, Cornell University, Ithaca, New York, 1986.

[Schu69]° R. Schumaker, B. Brand, M. Gilliland, W. Sharp, “Study for Applying Computer Generated Images to Visual Simulation”, TR 69-14, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, Texas, September 1969.

[Shao88] M-Z. Shao, Q-S. Peng, Y-D. Liang, “A New Radiosity Approach By Procedural Refinements For Realistic Image Synthesis”, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 93-101.

[Shin87]° M. Shinya, T. Takahashi, S. Naito, “Principles and Applications of Pencil Tracing”, Computer Graphics, Vol. 21, No. 4, July 1987, pp. 45-54.

[Sieg81]° R. Siegel, J. Howell, “Thermal Radiation Heat Transfer”, second edition, Hemisphere, Washington, DC, 1981.

[Sill89] F. Silion, C. Puech, “A General Two-Pass Method Integrating Specular and Diffuse Reflection”, Computer Graphics, Vol. 23, No. 3, July 1989, pp. 335-344.

[Sill91] F. Sillion, J. Arvo, S. Westin, D. Greenberg, “A Global Illumination Solution for General Reflectance Distributions”, Computer Graphics, Vol. 25, No. 4, July 91, pp. 187-196.

[Smit92] B. Smits, J. Arvo, D. Salesin, “An Importance-Driven Radiosity Algorithm”, Computer Graphics, Vol. 26, No. 2, July 1992, pp. 273-282

[Snyd92] J. Snyder, “Interval Analysis For Computer Graphics”, Computer Graphics, Vol. 26, No. 2, July 1992, pp. 121-129.

[Suth74] I. Sutherland, R.F. Sproull, R.A. Schumacker, “A Characterisation of Ten Hidden-Surface Algorithms”, ACM Computing Surveys, Vol. 6, No. 1, March 1974, pp. 1-55.

[Tell92] S. Teller, “Computing the Antipenumbra of an Area Light Source”, Computer Graphics, Vol. 26, No. 2, July 1992, pp. 139-148.

[Torr67]° K. Torrance, E. Sparrow, “Theory for Off-Specular Reflection from Roughened Surfaces”, J. Opt. Soc. Am., Vol. 57, No. 9, September 1967, pp. 1105-1114.

[Wall87] J.Wallace, M.Cohen, D. Greenberg, “A Two-Pass Solution to The Rendering Equation: A Synthesis Of Ray Tracing And Radiosity Methods”, Computer Graphics, Vol. 21, No. 4, July 1987, pp. 311-319.

[Wall89] J. Wallace, K. Elmquist, E. Haines, “A Ray-Tracing Algorithm For Progressive Radiosity”, Computer Graphics, Vol. 23, No. 3, July 1989, 315-324

[Ward88] G. Ward, F. Rubinstein, R. Clear, “A Ray-Tracing Solution for Diffuse Interreflection”, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 85-92.

[Warn69]°	J. Warnock, "A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures", TR4-15, Computer Science Department, University of Utah, Salt Lake City, Utah, June 1969.
[Whit80]°	T. Whitted, "An Improved Illumination Model for Shaded Display", Communications of the ACM, Vol. 23, No. 6, June 1980, pp. 343-349.
[Will78]°	L. Williams, "Casting Curved Shadows on Curved Surfaces", Computer Graphics, Vol. 12, No. 3, August 1978, pp. 270-274.

A Special Treat

For all those who stuck it out until the end, here’s the Cookie Monster song...

Now what starts with the letter C?
Cookie starts with C... Let’s think of other things that start with C!
Ah... oh... who cares!

[starts singing]

C is for cookie, that's good enough for me...
C is for cookie, that's good enough for me...
C is for cookie, that's good enough for meeee...
Oh, cookie, cookie, cookie starts with C.

[repeat with monsters humming in the background]

[voice - with humming in background]

You know what? A round cookie with a bite out of it looks like the letter C.
A round donut with one bite out of it also looks like a C, but it is not as good as a cookie.
You know what? A round cookie with a bite out of it looks like the letter C.
A round donut with one bite out of it also looks like a C, but it is not as good as a cookie.
Oh, and the moon sometimes looks like a C, but you can't eat that.

[singing]

C is for cookie, that's good enough for me, Yeah...
C is for cookie, that's good enough for me...
C is for cookie, that's good enough for meeee...
Oh, cookie, cookie, cookie starts with C. Yeah!
Cookie, cookie, cookie starts with C. Oh Boy!
Cookie, Cookie Cookie starts with C! Arraagh yum yum yum!

[Finis]