Computer Generation of Photorealistic Images using Ray Tracing

A Thesis by: Derek O' Reilly B.Sc.
Supervisor: Dr. M. Scott Ph.D.

Submitted to
SCHOOL OF COMPUTER APPLICATIONS
DUBLIN CITY UNIVERSITY
for the degree of
Master of Science
July 1991

Declaration No portion of this work has been submitted in support of an application for another degree or qualification in Dublin City University or any other University or Institute of Learning.

Abstract

Computer generation of *photorealistic* images has been *the* target of computer graphics designers almost since the birth of the computer. Of all methods tried, *Ray tracing* has proven to be the best at generating computer images that exhibit all the optical features found in a real life photograph.

Ray tracing is the subject matter of this thesis. We discuss the various issues involved in the design of a ray tracing system that will model and render complex scenes and we implement such a ray tracing system.

Contents

The contents of this thesis are divided into six chapters. Chapter one describes the problems of generating photorealistic images and how ray tracing can be used to solve these problems.

Chapter two gives an overview of ray tracing. Here we describe what rays are, how they travel and how they interact within a modelled scene. We describe some aliasing features of computer monitors that reduce the photorealistic quality of a ray traced scene and we discuss some methods for dealing with the aliasing.

We need to model a scene before we can ray trace it. Modelling is the subject matter of chapter three.

How ray tracing is used to *render* a modelled scene, so as to produce a photorealistic image on a computer monitor, is the topic for discussion in chapter four.

In chapter five, we discuss the implementation of a real ray tracer. This ray tracer, *PRIME*[1], was developed by the author of this thesis. *PRIME* is based on the contents of the previous chapters.

In chapter six we discuss some conclusions from the research, and we describe some possible enhancements and extensions that can be made to *PRIME*.

---

[1] *PRIME* is *P*hoto*R*ealistic *I*mage *M*odelling *E*nvironment.

Acknowledgement

To my mother, for the chance to pursue these studies, and to Dr. Michael Scott, for his advice and guidance along the way.

CONTENTS

CHAPTER ONE Photorealistic Graphics 1

1.1 MOTIVATION 1

1.2 WHAT IS COMPUTER GRAPHICS 1

1.3 PHOTOREALISTIC IMAGES 1
- 1.3.1 The Difficulties of Modelling 2
- 1.3.2 The Difficulties Of Rendering 2

1.4 RAY TRACING 2

CHAPTER TWO An Overview of Ray Tracing 3

2.1 Introduction 3

2.2 TRACING RAYS 3
- 2.2.1 Forward Ray Tracing 3
- 2.2.2 Backward Ray Tracing 5
- 2.2.3 Pixels and Rays 5

2.3 RAY TYPES 5
- 2.3.1 Reflected and Transmitted Rays 6
- 2.3.2 Shadow and Illumination Rays 7

2.4 RECURSIVE MODEL 7

2.5 ALIASING 9
- 2.5.1 Spatial Aliasing 10
- 2.5.2 Temporal Aliasing 10
- 2.5.3 Anti-Aliasing 13

2.6 SUPERSAMPLING 14
2.6.1 Adaptive Supersampling 14
2.6.2 Stochastic Supersampling 15
2.6.3 Statistical Supersampling 16
2.6.4 Beam Tracing 16
2.6.5 Cone Tracing 17

CHAPTER THREE Modelling the Real World 18

3.1 Introduction 18

3.2 GEOMETRICAL TRANSFORMATIONS 18
3.2.1 2D Transformations 18
3.2.2 Homogeneous Coordinates 23
3.3.3 Composition of 2D Transformations 24
3.3.4 3D Transformations 26
3.3.5 Coordinate Systems 27
3.3.6 Inverse Transform Matrix 28

3.4 SURFACE MODELLING 29
3.4.1 Polygonal Surfaces 29
3.4.2 Parametric Surfaces 31
3.4.3 Fractal Surfaces 34

3.5 SOLID MODELLING 36
3.5.1 Polyhedra 36
3.5.2 Quadrics 37
3.5.3 Other Forms Of Solid Modelling 40

3.6 CONSTRUCTIVE SOLID GEOMETRY 41

3.7 SPEEDING THINGS UP 44
3.7.1 Bounding Volumes 45
3.7.2 Spatial Subdivision 47

3.7.2.1 Uniform Spacial Subdivision 47
3.7.2.2 Nonuniform Spacial Subdivision 48
3.7.3 Potential Pitfalls In Spacial Subdivision 49
3.7.4 Ray Directional Techniques 50
3.7.4.1 The Light Buffer 50
3.7.4.2 Ray Coherence 51
3.7.4.3 Ray Classification 52
3.7.4.4 Exploiting Coherence 52

CHAPTER FOUR Rendering in Ray Tracing 53

4.1 Introduction 53

4.2 COLOUR 53
4.2.1 The Wave Model of Light 54
4.2.2 The Particle Model Of Light 55

4.3 COMPUTER REPRESENTATION OF COLOUR 55
4.3.1 CIE Chromaticity diagram 55
4.3.2 Colour Monitors 57
4.3.3 The RGB Model 59
4.3.4 Colour Palettes and Lookup Tables 60

4.4 COLOUR QUANTIZATION 60
4.4.1 Getting over the 256 colour limit 61
4.4.2 Popularity Algorithm 61
4.4.3 Median-cut Algorithm 62
4.4.4 Octree quantization algorithm 63

4.5 COMPUTER COLOUR AS RAYS 64
4.5.1 Three rays in one 64

4.6 LIGHT TRANSPORTATION MODES 64
4.6.1 Surface Normals 64

4.6.2 Specular Reflection 65
4.6.3 Specular Transmission 67
4.6.4 Diffuse Reflection 70
4.6.5 Diffuse Transmission 71
4.6.6 Light Sources 71

4.7 SURFACE CHARACTERISTICS 72
4.7.1 Surface Texture Maps 72
4.7.2 Surface Roughness Maps 73
4.7.3 The Fresnel Function, *F* 75
4.7.4 Reflectance Coefficients 77
4.7.5 Transmission Coefficients 77
4.7.6 Transmissitivity 78

4.8 THE HALL RENDERING MODEL 78

4.9 OTHER RENDERING MODELS 80

CHAPTER FIVE The *PRIME* System 81

5.1 Introduction 81

5.2 DEVELOPMENT ENVIRONMENT 81

5.3 MODELLING 82
5.3.1 Primitives in *PRIME* 82
5.3.2 Objects In PRIME 84
5.3.3 Copying Objects 86

5.4 INTERSECTION ROUTINES IN PRIME 86
5.4.1 Mathematical Definition of a Ray 86
5.4.2 Ray/Primitive Intersection 87
5.4.2.1 Ray/Sphere Intersection 87
5.4.2.2 Ray/Cylinder Intersection 89

5.4.2.3 Ray/Infinite Cylinder Intersection 89
5.4.2.4 Ray/Circular Plane Intersection 90
5.4.2.5 Ray/Cone Intersection 92
5.4.2.6 Ray/Cube Intersection 93
5.4.2.7 Ray/Pyramid Intersection 93
5.4.2.8 Ray/Polygon Intersection 94
5.5.1 Inverse Mapping of a Sphere 96
5.5.1.1 Inverse Mapping for a Cylinder 97
5.5.1.2 Inverse Mapping for a Circle 99
5.5.1.3 Inverse Mapping for a Cone 100
5.5.1.4 Inverse Mapping for a Cube 101
5.5.1.5 Inverse Mapping for a Polygon 102
5.5.1.5.1 Inverse Mapping for a Quadrilateral 102
5.5.1.5.2 Inverse Mapping for a Triangle 105

5.8 BOUNDING VOLUMES 105

5.9 PROBLEMS ENCOUNTERED 106
5.9.1 Numerical Precision 106
5.9.2 Memory 106

5.10 COLOUR PLATES 107

CHAPTER SIX Conclusions And Future Work 109

6.1 CONCLUSIONS 109

6.2 FUTURE WORK 109
6.2.1 Speed Efficiency 109
6.2.1.1 Spacial Subdivision 109
6.2.1.2 File Management 110
6.2.2 Enhancing the Rendered Image 110
6.2.2.1 Extra Modelling Primitives 110
6.2.2.2 Texture Maps 110

6.2.2.3 Surface Modelling 111

6.3 CURRENT AREAS OF RESEARCH 111
6.3.1 Parallel Machines 111
6.3.2 Radiosity 111

Bibliography

Colour Plates

Appendix A: Program Listings

Photorealistic Graphics

1.1 MOTIVATION

Graphics is perhaps the most rewarding area of computer science. This is hardly surprising, considering that the predominant human sense is sight. There is a measure of fascination in creating the appearance of a solid object on a computer monitor, solely by the execution of a list of computer instructions. It is this fascination that has lead me to pursue my research; a study of computer generated *photorealistic* images and their implementation using ray tracing techniques.

1.2 WHAT IS COMPUTER GRAPHICS

Any *computer generated* graphical image that is represented on a computer monitor is called *computer graphics*. A related area, *image-processing* also outputs graphical images onto a computer monitor. The difference between the two fields is that computer graphics systems generate their own images, while those of image processing systems are captured by a camera, or some other image grabbing device.

Most computer graphics represents simple two dimensional (2D) images. Examples of 2D graphics are statistical pie and bar charts, icons used in windows applications, and the space craft and monsters used in the famous arcade game "Space Invaders".

More complicated, three dimensional (3D) images are used in *computer aided design* (CAD), flight simulators and other games, advertising, logos, and movies. Unlike the simple 2D graphics, 3D graphics must represent 3D objects on a 2D computer monitor, much in the same way that a camera represents a 3D scene with a 2D photograph.

1.3 PHOTOREALISTIC IMAGES

In order for a user to properly perceive the 3D image, the image must include shading, hidden surface removal and other photorealistic characteristics.

Computer generation of photorealistic graphics can be separated into two distinct parts; *modelling* and *rendering*. Modelling involves creating objects, moving them around to arrange a scene, defining how each object will look in the scene, and defining how the lighting and camera will look in the scene. Rendering involves making a realistic image out of the modelled scene by applying surface characteristics to the surfaces of the objects in the scene. Both modelling and rendering are difficult operations in a computer environment.

1.3.1 The Difficulties of Modelling

We can outline the difficulties of modelling by comparing the task to that of a sculptress. Both are creating a 3D scene. However, the sculptress has a greater use of both hands and eyes.

The sculptress can manipulate the modelled scene with her hands and 3D tools. She works with 3D objects in a familiar 3D world. The computer modeller's task is much more difficult. She has to manipulate 3D objects in the 2D world of a computer monitor. At best, she will have the use of a mouse for manipulating objects.

The sculptress has the benefit of depth perception and depth perspective - both eyes work together to provide a sense of depth. Also, the sculptress can easily move to provide a different view of the scene. The computer modeller is restricted to a single fixed 2D frame of the 3D scene at any given time.

1.3.2 The Difficulties Of Rendering

Rendering begins at the end of the modelling process, with a description of how objects are arranged in a scene, the materials they are supposedly made of, the lights that fall upon them and the placement of the camera. Rendering ends with a finished image on a 2D computer monitor.

In the real world, light interacts in many ways with objects. For example, light bends when it enters water, it is reflected off shiny surfaces, while being absorbed by matt ones. Light cast on one side on an object can produce shadows on the surface of objects that lie on the other side of the object. When rendering, objects can be hidden behind other objects. We cannot see the complete surface of a 3D object at once, unless we add mirrors to our scene. If we do add mirrors, then we have the equally difficult task of rendering the light reflection effects that they create. To generate photorealistic images, we must also take into account camera lens aperture and numerous other optical phenomena.

1.4 RAY TRACING

Clearly, computer generation of photorealistic images is a non-trivial task. Of all the 3D rendering approaches known, one method stands out as producing most accurately photorealistic images. The method is *Ray Tracing* and is the subject matter of this thesis. The first computer generated image to be passed as a photograph, that developed by Porter [PORT84], was developed using ray tracing techniques. Since then, ray tracing techniques have been developed to incorporate even more complex visual characteristics.

An Overview of Ray Tracing

2.1 Introduction

Ray tracing was first suggested by Appel in 1968 [APPE68]. It was later used by Goldstein and Nagel [NAGE71] as a solution to the hidden surface problem. However, it was not until the late 1970's that it was implemented by Kay & Greenberg [GREE79] and by Whitted [WHIT80] to render complete images. Since then, ray tracing has proven to be the most popular graphics technique for rendering photorealistic scenes. The visual attributes of each pixel in a viewport are determined by tracing a ray from a viewing position, via the pixel, into the world coordinate system. At its simplest, the pixel takes the colour of whichever object is struck first by the ray. Further tracing of rays that are reflected or transmitted at the ray's intersection point with an object allows ray tracing to be used to create a large variety of optical effects.

In this chapter we describe the basic concept of the ray tracing algorithm. We then expand on the problems that aliasing can cause when ray tracing, and discuss some supersampling methods that can be employed to minimise the aliasing effects.

2.2 TRACING RAYS

In the real world light travels as *photons* of energy. These photons travel along straight paths called *rays*. When a photon strikes our eye, the energy of the photon is transferred to the receptor cells on our retina. The retina perceives colour as a measurement of the energy of the photon. Different energy levels give different colours.

2.2.1 Forward Ray Tracing

Ray Tracing techniques mimic very well the way light interacts with the real world. In the real world light is emitted from light sources. It then travels as light rays in an infinite number of directions away form the light source. The vast majority of the rays will never be visible to a viewer looking into the scene through a view plane. Those rays that will ultimately be visible to the viewer must first be reflected and transmitted throughout the scene

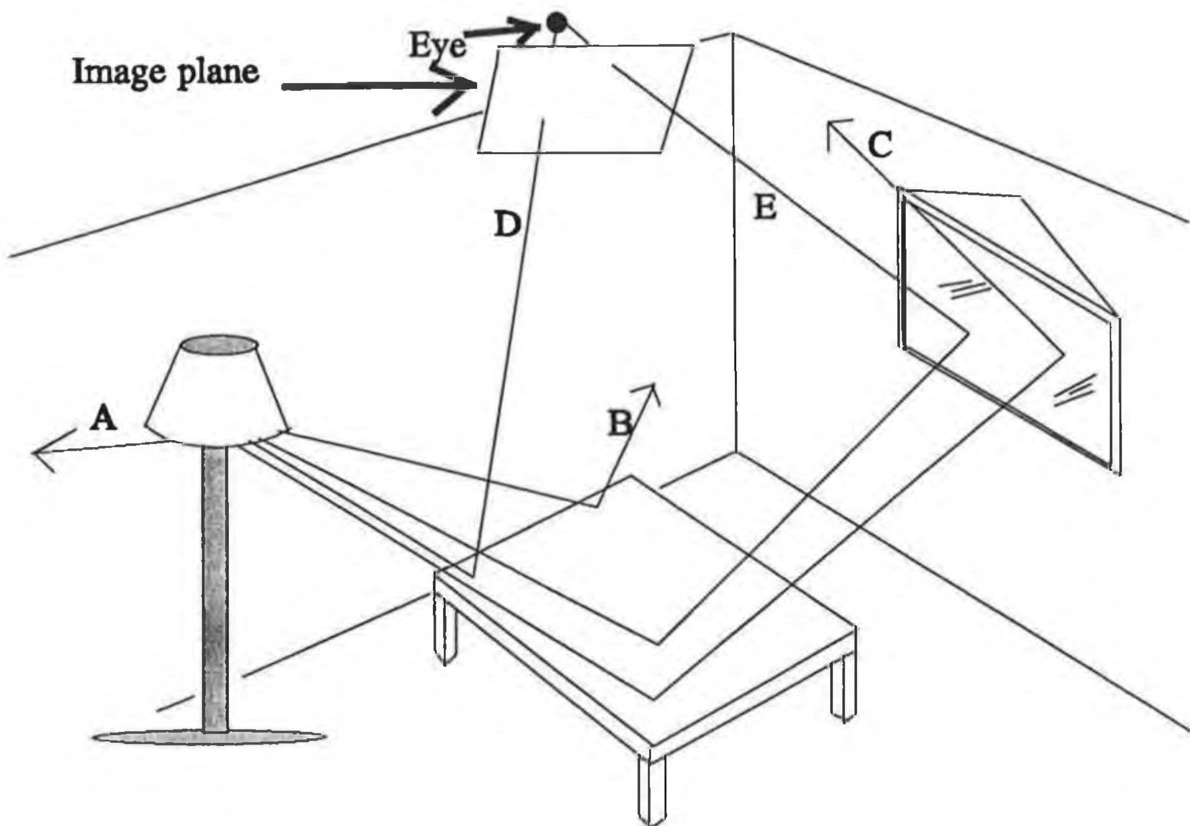


Fig. 2.1 The paths taken by some of the light rays in a scene.

before finally being projected through the viewing plane and into the eye. Let us take *Fig 2.1*[1] as an example. Light ray **A** travels away from the light source. Upon striking the wall it is absorbed. Ray **B** strikes the table, and is reflected. It then strikes the wall and is absorbed. Ray **C** is reflected off the table and the mirror before it too strikes the wall. Ray **D** is reflected off the table and through the view plane into the eye. Ray **E** is reflected off the table, then off the mirror and through the view plane into the eye. By following the paths taken by these five rays we have been *forward ray tracing*. Rays **A**, **B**, and **C** represent a sample of the vast majority of rays that will never reach the eye. In reality, there would be an infinity of such rays. Rays **D** and **E**, on the other hand, are rays that do reach the eye. These are the only rays that we are interested in.

---

[1] Strictly speaking, not all the light will be absorbed when a ray strikes the wall. However, for simplicity of discussion we shall assume that it does. The rendering models discussed in chapter 5 take into account the full complexities of ray surface intersections.

2.2.2 Backward Ray Tracing

Forward Ray Tracing is too inefficient a technique to implement on a computer, as only a tiny percentage of emitted rays ever reach the view plane. Fortunately, it is possible to trace the path taken back along a ray from the eye, through the view plane, and around the scene, until it finally arrives at the light source from which it originated. Knowing the path that a ray takes and the objects that it intersects makes it possible to calculate the ray's colour. This method of tracing rays is called *backward ray tracing*. Backward ray tracing guarantees that only those rays which we are interested in are traced; rays **D** and **E** in *Fig 2.1*. This is the method used by all ray tracing systems. Because backward ray tracing is the only ray tracing method used, it is usually referred to simply as *ray tracing*. All future references to *ray tracing* in this thesis will be taken to mean backward ray tracing.

Associated with backward ray tracing is some reverse terminology. When we talk about a reflected or transmitted ray, what we really refer to is the ray that caused the reflection or transmission. The direction of these rays is also reversed, as is shown in *Fig 2.2*.

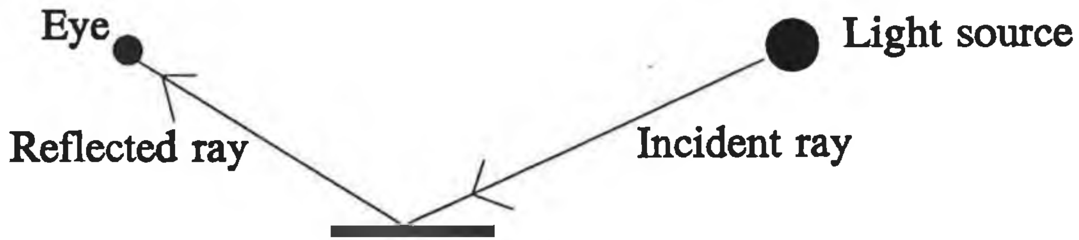
2.2.3 Pixels and Rays

In a computer model an origin, a viewport, and a world coordinate system will take the place of the eye, the view plane and the scene respectively. To generate a perspective rendering of the scene one ray is projected from the origin through each pixel in the viewport. For parallel rendering one ray is shot through each pixel in a perpendicular direction to the view plane.

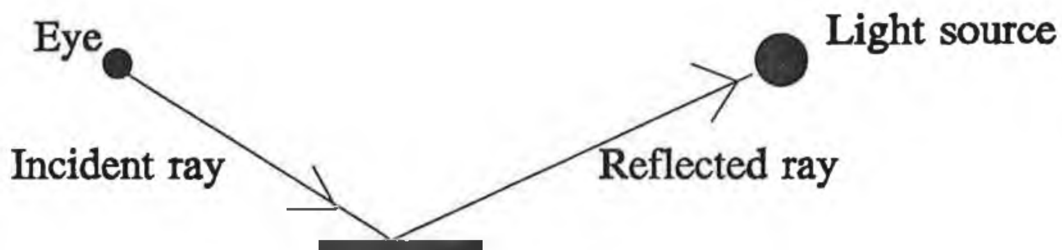
2.3 RAY TYPES

Rays can be divided into four classes: *pixel rays* or *eye rays* which carry light directly to the eye through a pixel on the monitor; *reflection rays* which carry light reflected by an object; *transmission rays*[2] or *transparency rays* which carry light passing through an object; and finally *illumination rays* or *shadow rays* which carry light from a light source directly to an object's surface. A further type of ray is an *incident ray*. Any ray that strikes a surface is an incident ray with respect to that surface.

[2] *Transmission rays* are refered to in some literature as *refraction rays*. Both are the same thing.



Forward Ray Tracing



Backward Ray Tracing

Fig. 2.2 Comparison between *Forward* and *Backward* ray tracing.

Although rays are divided into different classes, they are all mathematically similar. The classifications are made only as an aid to discussion. The mathematical details of rays is described in *Section 5.4.1*.

2.3.1 Reflected and Transmitted Rays

When a ray intersects a point on a surface it causes a new ray to be reflected away from the surface. If the surface is not opaque then a second ray is transmitted through the surface. An exact mathematical derivation for finding the direction of such reflected and transmitted rays is described in *Section 4.6*. For now it is sufficient to know that these rays exist.

2.3.2 Shadow and Illumination Rays

When a ray intersects a point on the surface of an object it is necessary to find out which lights in the world coordinate system are cast upon the point. The intensity of each light cast on the point will affect its colouring. In order to find out which lights do reach the point a *shadow ray* is sent out from the point in the direction of each of the lights in turn. If any opaque object is positioned between the point and the light source along the path of the shadow ray, then the point is in the shadow of the object with respect to this light source. If no objects lie along the shadow ray's path, then the point is illuminated by the light source. The ray then becomes an *illumination ray*. Looking at *Fig 2.3*, we see that ray **E** intersects

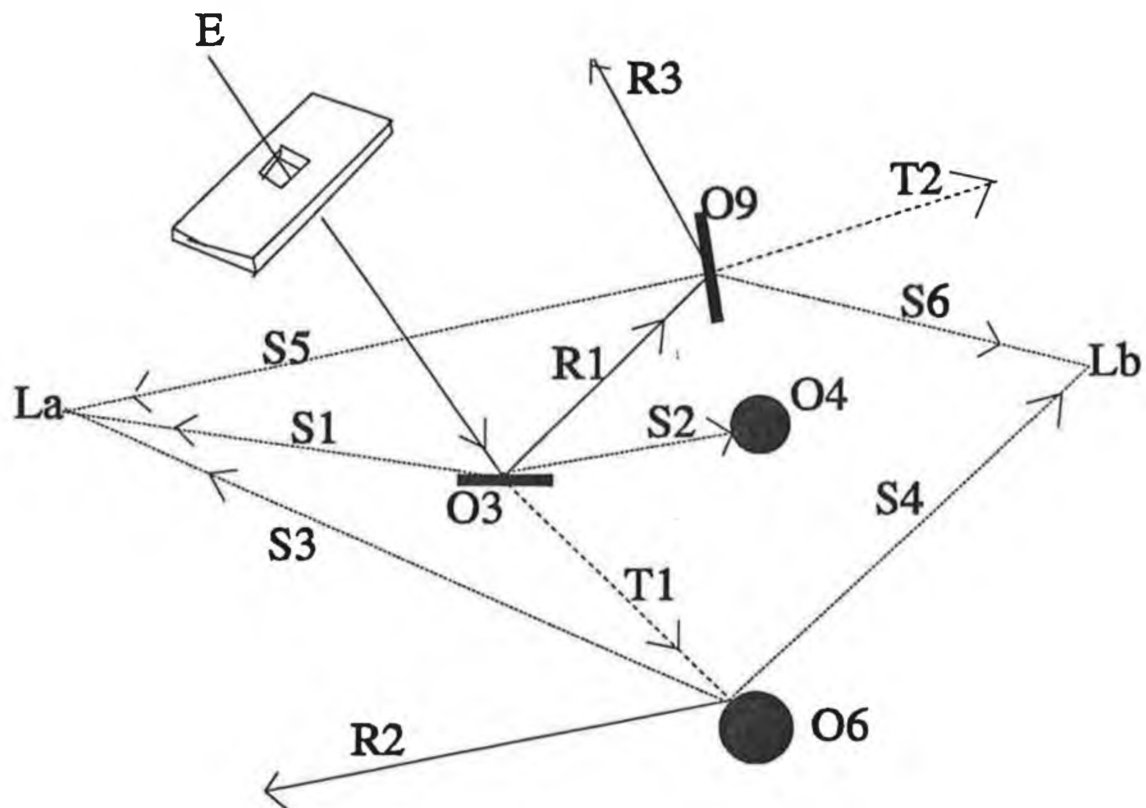


Fig. 2.3 An eye ray **E** shot through a scene.

object **O3**. It then generates two shadow rays, **S1** and **S2**. As object **O4** lies in the path of shadow ray **S2** we say the intersection point is in the shadow of **O4** with respect to light source **Lb**. No objects lie in the path of shadow ray **S1**, so it becomes an illumination ray. This will contribute to the final colour of light leaving the intersection point back along **E**.

2.4 RECURSIVE MODEL

If an eye ray strikes an object's surface then the ray becomes an incident ray with respect to that surface. The interaction of the ray and the surface will cause a reflected ray to be generated. Depending on the surface characteristics, a transmitted ray may also be generated. These new rays will themselves be cast in the same manner as the eye ray. Upon striking an object's surface the new ray will itself become incident to the surface and again a new level of reflected and transmitted rays will be generated. This leads to a recursive model for ray tracing. The recursive model is shown in schematic form as a *ray tree*. *Fig 2.3* shows a ray traced scene. It's ray tree is shown in *Fig 2.4*.

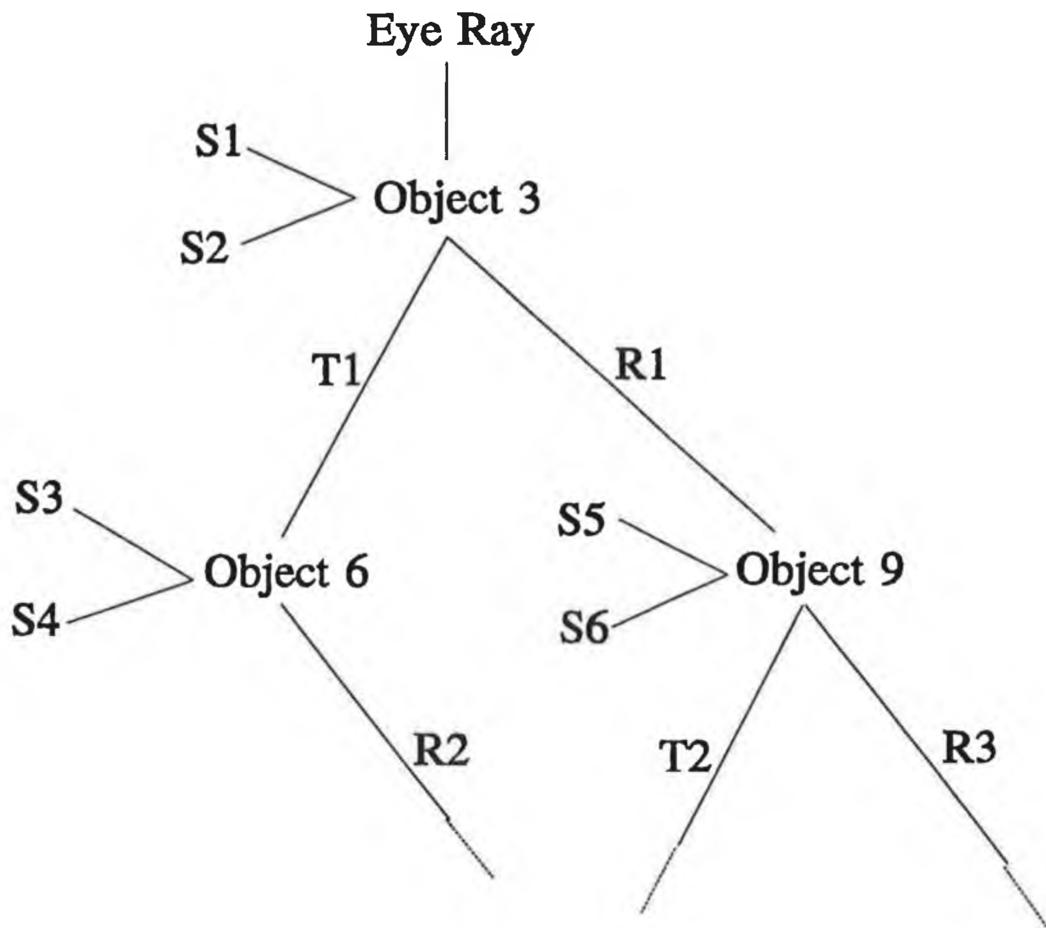


Fig. 2.4 A *ray tree* in schematic form.

We must now ask the question "At what stage does the recursion stop?". The normal procedure is to stop tracing if either a ray goes out of the world coordinate system or its

contribution to the final colour of the ray tree becomes too small. When a ray leaves the world coordinate system it can be assigned a predefined background colour and traced no further. Working out the contribution of any ray to the final colour of a ray tree is more difficult to decide. The further down its position in the ray tree, the less the contribution of any ray to the final colour. An example should help verify this. Suppose ray **E** was the only ray affecting a pixel (i.e. it spawned no reflected or transmitted rays), then we would take its colour as being the final colour to arrive at the top of the ray tree. However, it does spawn both a reflected and transmitted ray. Their individual contributions to the final colour must be less than that of **E** since **E** is formed by combining both together with the shadow ray **S1**[3]. Now **R1**, the reflected ray that helped form colour **E**, is itself formed by combining **R3** and **T2**, along with the shadow rays **S5** and **S6**. Therefore, both **R2** and **T2** must contribute less to the final colour than does **R1**. When building a ray tree, it is usual to set a contribution threshold, below which further tracing of rays stops. This technique is known as *adaptive tree-depth control.*

2.5 ALIASING

A major problem when synthesizing an image on a digital computer is that a computer monitor cannot represent a continuous (*analog*) signal. Through our eyes, or by using a camera, we can see an analog picture. Every line, curve, and tiny object in the frame is represented exactly. When using a computer to simulate this image it is impossible to generate an exact photo replication. This is because the computer is restricted to using a finite number of pixels to represent the analog signal. You may argue that by using a computer monitor with a higher resolution it must be possible to overcome this problem. This is only partially true. No matter how high a resolution monitor is used the effects of aliasing are bound to creep into any computer generation of photorealistic images. Some of the aliasing problems are discussed in more detail in the following sections. Further reading on the problems of aliasing can be found in [CROW77].

---

[3] Exactly how these rays are combined is described in detail in chapter 5.

2.5.1 Spatial Aliasing

Aliasing caused as a result of the uniform nature of the pixel grid is known as *spatial aliasing*. *Fig 2.5* shows a quadrilateral displayed at a variety of monitor resolutions. The smooth edges of the original quadrilateral are approximated by the jagged edges of the

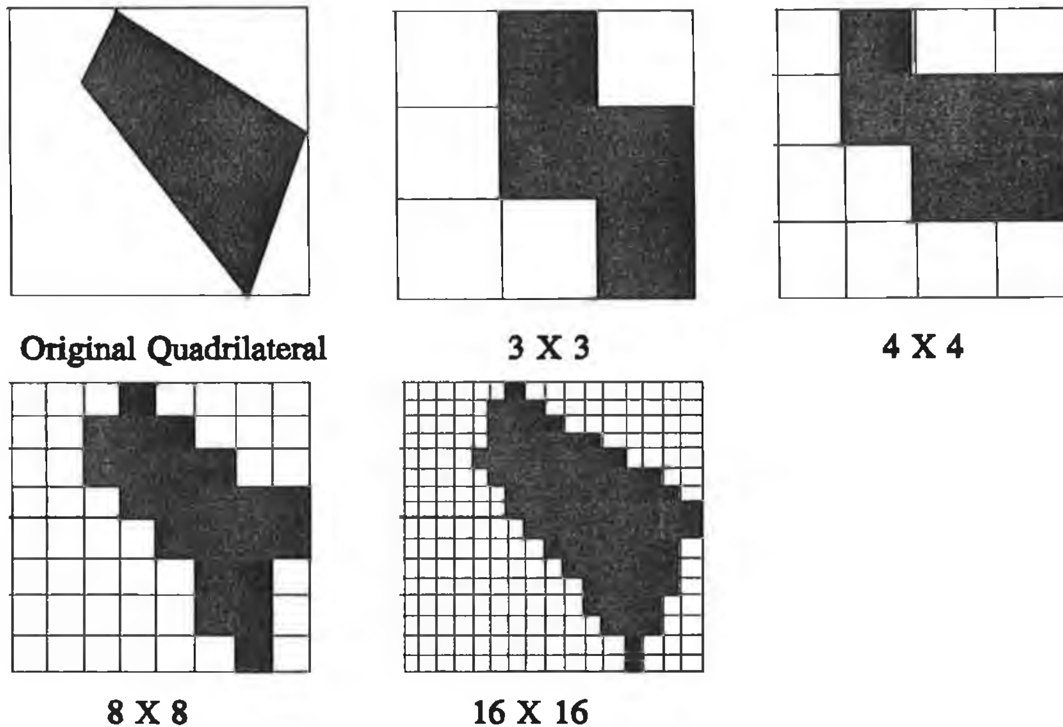


Fig. 2.5 The effects of spatial aliasing at different screen resolutions.

monitor grid. These jagged edges are known as *the jaggies*. As the resolution of the monitor is increased, the effect of the jaggies will diminish. However, the jaggies will never completely disappear, they will only get smaller. If you have a very high resolution monitor, it may appear that there is no spatial aliasing. However, by projecting the same image onto a hugh cinema screen the jaggies will be magnified and will thus be clearly visible.

A second effect of spacial aliasing is that very small objects, or large objects sufficiently far away, may be hidden from the rays shot through the pixels. This is shown in

Fig 2.6. You might think that if an object is that small, then it doesn't really matter whether or not it is displayed at all. Unfortunately, that is very far from the truth, as will be seen by looking at temporal aliasing.

2.5.2 Temporal Aliasing

The word *temporal* comes from the latin *tempus*, meaning time. *Temporal aliasing* is aliasing produced when using computer graphics in animation. An animation is nothing more than many still frames shown in sequence. You might think that if each still frame was very good, then the animation would also be very good. This is not the case.

You may have noticed on television what happens as a wagon wheel accelerates from a stationary position. It initially appears to rotate in the direction of the cart's motion, as expected. However, it then appears to stop moving, and then rotates backwards! Why is this so? A film normally consists of a *sampling rate* of between 24 and 30 frames per second (i.e. between 24 and 30 frames are shown in sequence per second). When the wheel is rotating at a speed less than the sample rate, a camera can correctly sample the image. As the wheel speeds up and goes faster than the sample rate, then it may appear to go backwards. Take *Fig 2.7* as an example. This shows a wheel, with one spoke coloured black, sampled at eight frames per second. In the top row the wheel is rotating clockwise at one revolution per second and is correctly sampled. In the centre row, the wheel is rotating at four revolutions per

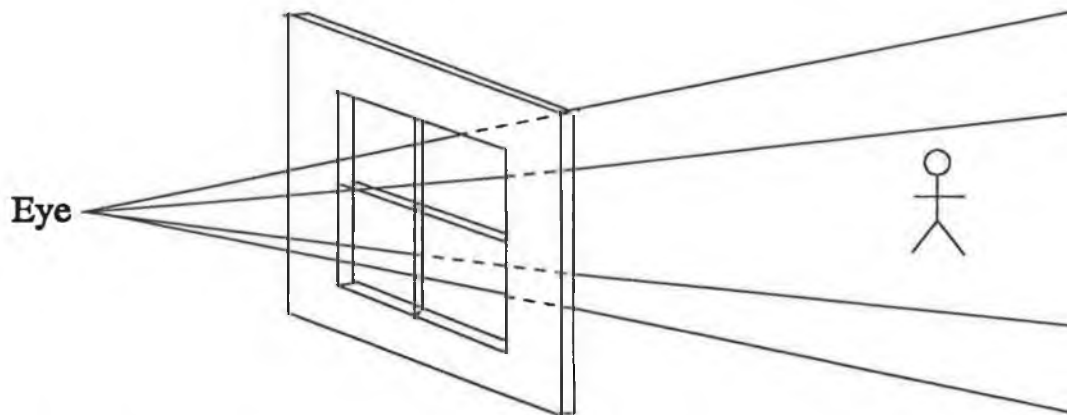


Fig. 2.6 A small object is missed by the rays shoot through these four adjoining pixels.

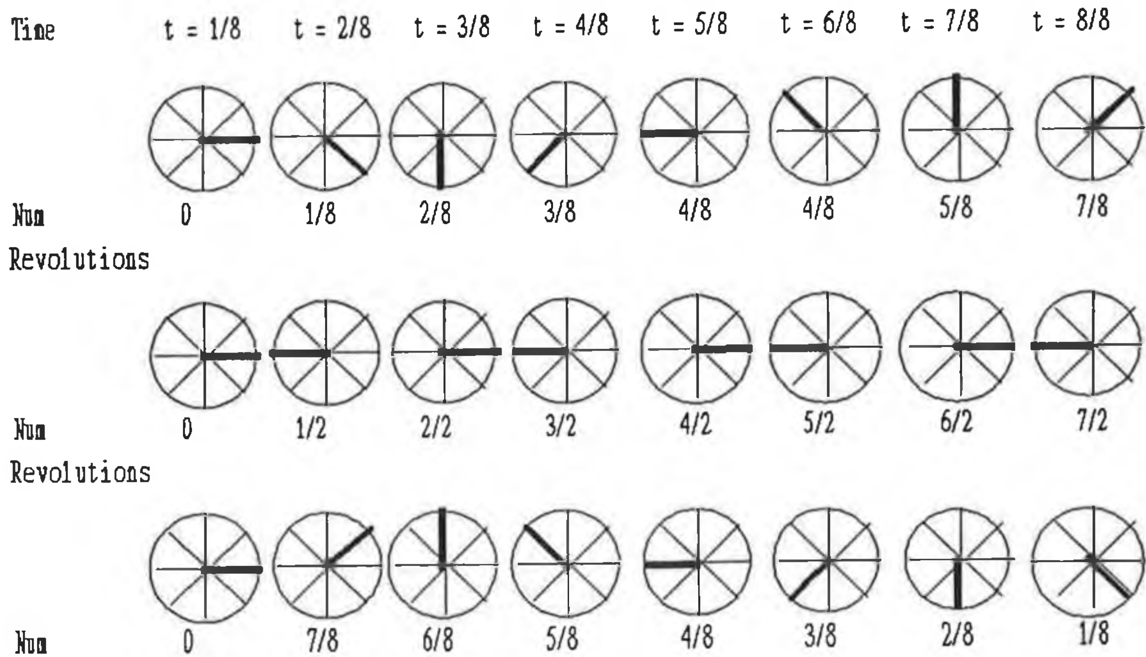


Fig. 2.7 A spinning wheel sampled at a constant rate of eight frames per second.

second. After sampling, we cannot tell in which direction the wheel is moving. In the bottom row the wheel is rotating at seven revolutions per second. However, it actually appears to be rotating anticlockwise at one revolution per second.

A second case of temporal aliasing highlights the problem of disappearing objects discussed in the above section on *spatial aliasing*. Now that this object is moving, it can cause more problems. As it moves across the monitor, an object may be hidden over several frames only to suddenly 'pop' up at the next frame. After several further frames, this object will again disappear off the monitor. *Fig 2.8* shows a polygon moving up a monitor over time. It pops up the monitor in discrete jumps rather than moving up in a smooth manner. This jerky movement is very disconcerting to the eye.

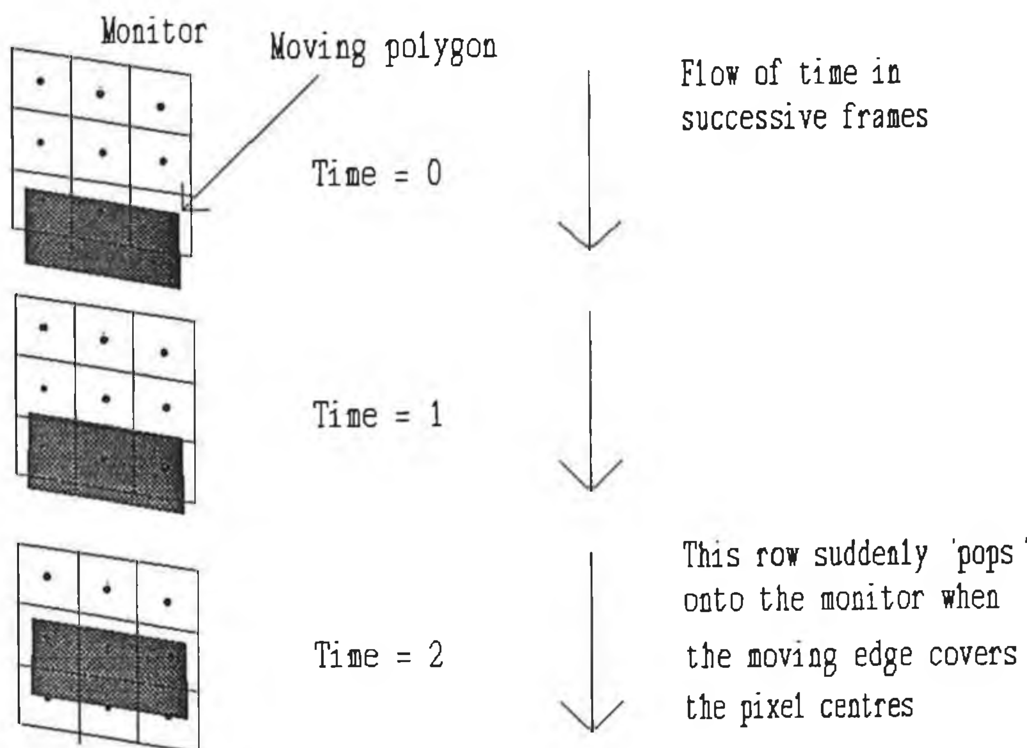


Fig. 2.8 A polygon slowly moving up the screen appears to 'pop'.

2.5.3 Anti-Aliasing

Aliasing effects can always be tracked down to the fundamental nature of digital computers and the point sampling nature of ray tracing. The essential problem is that we are trying to represent continuous phenomena with discrete samples. We will now discuss some of the methods of dealing with aliasing.

2.6 SUPERSAMPLING

The simplest way to counteract the effects of aliasing is to shoot lots of extra rays to generate our monitor image. We can then take the colour of each pixel to be the average colour of all the rays that pass through it. This technique is called *supersampling*. We might send nine rays through each pixel, and let each ray contribute one-ninth to the final colour of the pixel. For example, if six rays shot through a pixel hit a green ball, and the other three hit a blue background, then the final colour of the pixel will be two thirds green, one third blue; a more accurate colour than either pure green or pure blue. Although supersampling can

greatly reduce the effects of aliasing, it can never fully solve them.

The major problem with supersampling is that it is computationally very expensive. If nine rays are sent through each pixel then the total running time of the program is increased nine fold.

2.6.1 Adaptive Supersampling

Adaptive supersampling offers an attempt at reducing the computational overhead associated with supersampling. Rather than firing off some fixed number of rays through every pixel, we will use some intelligence and shoot rays only where they are needed. Whitted [WHIT80] describes such a method for supersampling. One way to start is to shoot five rays through a pixel, one through the centre, and one through each of the pixel's four corners.If all these rays return *similar* colours[4] then it is fair to assume that they have all hit the same object, and therefore we have found the correct colour. If the rays have sufficiently differing colours, then we must subdivide the pixel area into four quarters. We will then fire five rays through each of the four regions. Any set of five rays through a region that return similar colours will be accepted as a correct colour. We will recursively subdivide and shoot new rays through each region where the five rays differ. Because this technique subdivides where the colours change, it adapts to the image in a pixel, and is thus called *adaptive supersampling*.

This approach works fairly well, and is not too slow. Moreover, it is easy to implement. However, the fundamental problem of aliasing remains. No matter how many rays we shoot into a scene, if an object is too small , it may not be visible. We will still have small objects 'popping' across the monitor in animated sequences. The problem with adaptive supersampling is that it uses a fixed, regular grid for sampling. By getting rid of this regularity in the sampling, it is possible to minimise the effects of aliasing.

---

[4] The rays do not need to return *exactly* the same colour. A confidence interval can be set stating just how similar the rays are required to be.

2.6.2 Stochastic Supersampling

If we get rid of the regular sampling grid and replace it with an evenly distributed random grid we can greatly reduce aliasing effects. We will still shoot a regular number of rays through each pixel, but we will ensure that these rays are spread pretty randomly (or *stochastically*) over the whole area of the pixel. An example of this can be seen in *Fig 2.9*.

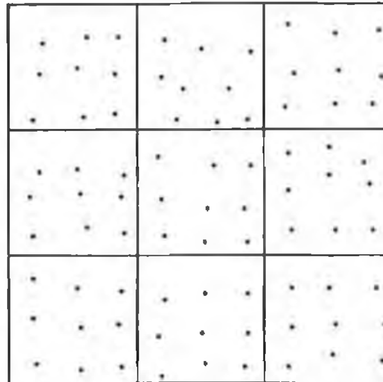


Fig. 2.9 The shooting of evenly distributed rays through pixels.

The particular distribution of rays that we use is important, so stochastic supersampling is sometimes called *distributed ray tracing*.

As a bonus, stochastic ray tracing give a variety of new effects that the discrete tracing algorithms don't handle well, or at all. Stochastic ray tracing allows us to render *motion blur*, *depth of field*, and soft edges on shadows, known as *penumbra regions*.

The bad news is that a new problem is introduced. We now get an average colour at each pixel. So although each pixel is almost the right colour, few are exactly right. We have introduced *noise*. The noise is spread out over the whole monitor like static on a bad television signal. Fortunately, the human visual system can usually filter out this noise.

By using stochastic ray tracing we may still be shooting too many rays through each pixel. As in adaptive supersampling for a regular grid, we need some method to reduce the average number of rays shot through any pixel.

2.6.3 Statistical Supersampling

We can use *statistical supersampling* to reduce the number of rays shot through the average pixel. We start by shooting four randomly distributed rays through a pixel. If the

colours of these rays are sufficiently similar, then stop the sampling. Otherwise shoot another four randomly distributed rays through the pixel and test all eight rays. Testing for similarity can be done by applying various statistical methods. In general, you set a confidence interval for the pixel. Higher confidence intervals will give a more accurate colouring, but as more rays are needed, they will be slower to compute. Supersampling is discussed by Mitchell [MITC87]; by Cook[COOK86], by Cook, Porter and Carpenter [PORT84]; by Lee [LEE85]; by Dippè [DIPP85], by Kajiya [KAJI86], and by Purgathofer [PURG86].

2.6.4 Beam Tracing

Another way of reducing aliasing effects is to use *beam tracing*. Beam tracing gets over the basic point sampling problem of normal ray tracing. Instead of letting the ray represent a point on the pixel we can assign the whole area of the pixel to the ray. The ray then becomes a pyramid, with the apex at the eye and the base defined by the four corners of the pixel. This is shown in *Fig 2.10*. When an intersection is found between such a beam

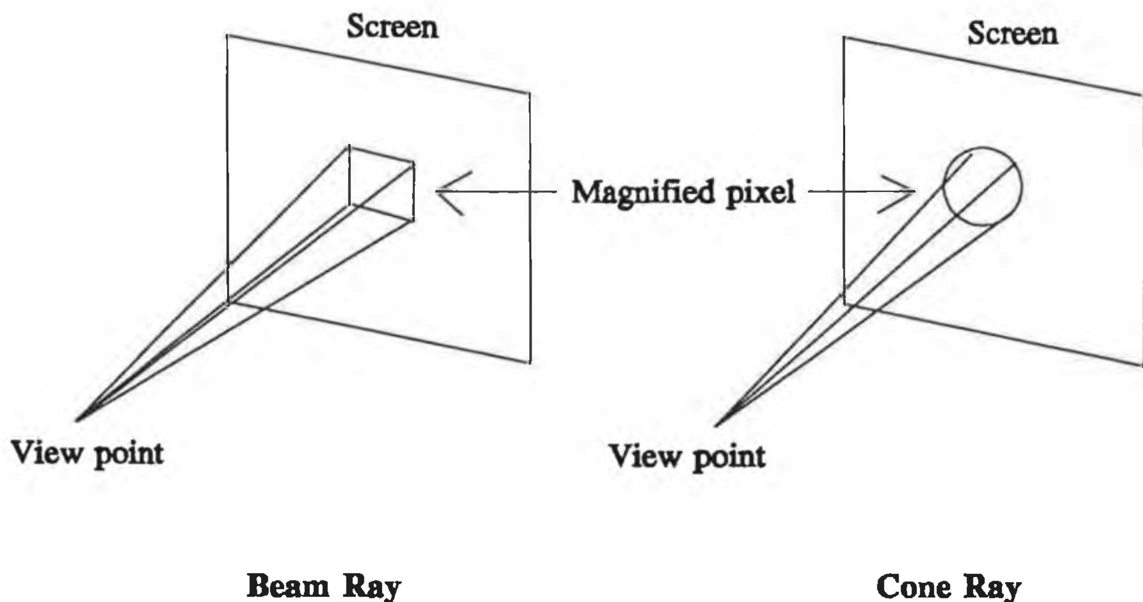


Fig. 2.10 A *beam ray* and a *cone ray*.

ray and an object, the area of intersection is calculated and used as a basis for performing

simple area anti-aliasing.

In beam tracing we need to cast only one ray through each pixel in the viewport. This is an advantage over the supersampling methods. However, the intersection algorithm between a beam ray and an object can be quite complex. Also, as the beam ray is reflected and transmitted off curved surfaces it can become very distorted, furthering the complexity of the intersection algorithm. The geometry of beam tracing is discussed by Dadoun and Kirkpatrick [DADO85], while beam tracing of polygonal objects is discussed by Heckbert and Hanrahan [HECK84].

2.6.5 Cone Tracing

Instead of a beam, we can trace a cone through each pixel. This is also shown in *Fig 2.10*. The advantage of a cone ray is that, when reflected or transmitted, a cone will still represent a good approximation to the incident cone. Cone tracing is discussed by Amanatides [AMAN84].

Like stochastic ray tracing, cone ray tracing can be used to implement various photo effects that cannot be handled using normal ray tracing. These effects include *fuzzy shadows*, and *dull reflections*.

Modelling the Real World

3.1 Introduction

Modelling involves building the scene to be ray traced. This scene could be simple, like a single sphere in space, or it could be a more complex model, like a street, full of buildings, cars and people.

In this chapter we describe the various methods used for describing objects when building computer models. We describe how individual objects are generated within a model, and how these objects can then all be transformed into one scene.

We will conclude this chapter by describing acceleration techniques that may be used when modelling, so as to ensure efficient ray tracing of the scene.

3.2 GEOMETRICAL TRANSFORMATIONS

Every object to be modelled on a computer can be defined in terms of a set of *control points* and a set of operations mapping these control points to form surfaces in 3D space. For example, a cube can be defined by eight control points and an operation describing how these control points represent the six faces of the cube. Even very complex objects, such as spline surfaces are represented as points in 3D with specific operations connecting these control points to form surfaces.

Geometrical transformations deal specifically with objects at the control point level, leaving the description as to how these points are mapped to be discussed in *Section 3.4* and *Section 3.5*. There are three transformations that can be performed on any point in 3D space. These transformations are *translation*, *scaling*, and *rotation*.

The three transformations can also be carried out in 2D space. We shall proceed by developing the structure of the 2D transformation matrix, and we will then use the results to build the 3D transformations matrix.

3.2.1 2D Transformations

2D geometry is carried out on a 2D *xy-plane*, shown in *Fig. 3.1*. Points in the xy-plane can be described in terms of $\mathbf{P} = [x\ y]$, where x and y are unit amounts parallel to the x and

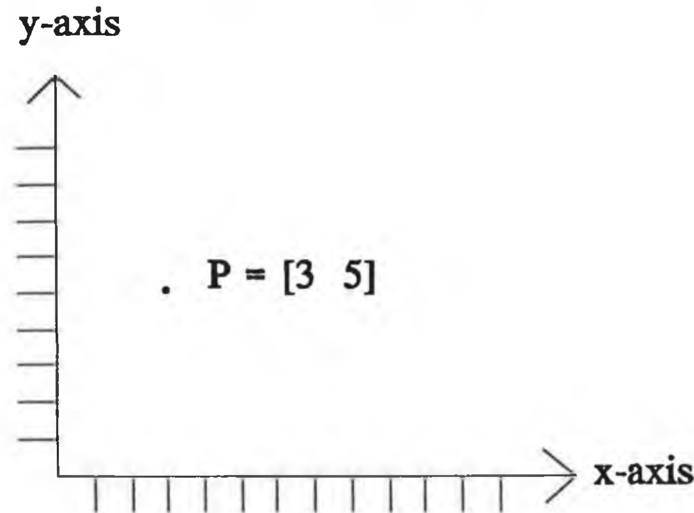


Fig. 3.1 The xy-axis of a 2D coordinate system

y axis respectively. When we represent points in such a way, we are using a *cartesian coordinate system.*

Any point, **P** = [x y], in the xy-plane can be *translated* to a new position by *adding* translation amounts to the coordinates of the point. A translation amount is given in terms of Dx units parallel to the x-axis and Dy units parallel to the y-axis. Dx and Dy may be either positive or negative amounts. We can place Dx and Dy in a translation vector **T** = [Dx Dy]. A translation from **P** = [x y] by an amount **T** = [Dx Dy] to a new point **P'** = [x' y'] can be written:

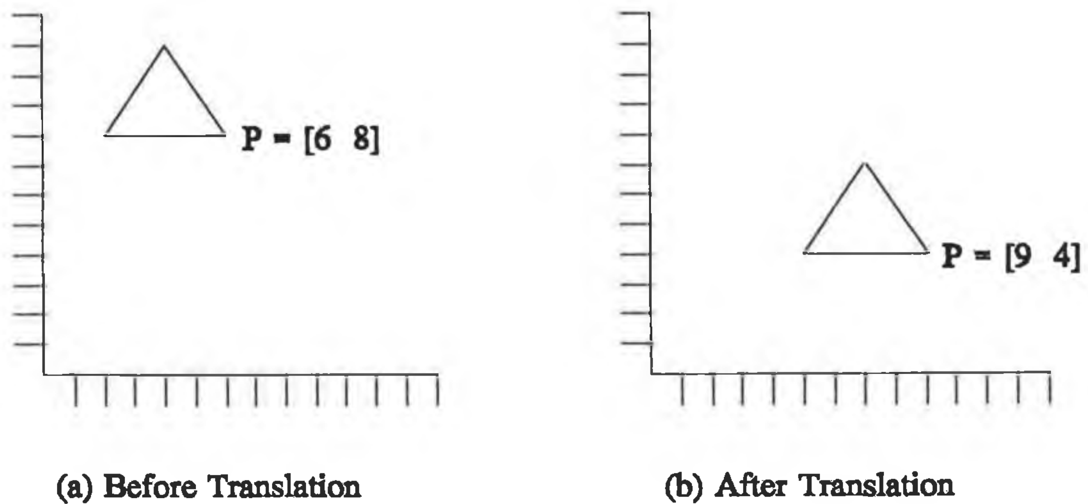
$$x' = x + Dx \qquad\qquad y' = y + Dy$$

or, more concisely:

$$\mathbf{P'} = \mathbf{P} + \mathbf{T}$$

An object can be translated by applying this equation to each of its defining points in turn. *Fig. 3.2a* shows a triangle that has three defining points. A translation **T** = [3 -4] is carried out on each of the three points, thus translating the whole object to the new position shown in *Fig. 3.2b.*

A point can be *scaled* by Sx and Sy units parallel to the x and y axes of the xy-plane. Again, the scaling values can be either positive or negative. Scaling is done by *multiplying*:



Translation T = [3 -4]

Fig. 3.2 Translation in 2D

$$x' = x * Sx, \qquad y' = y * Sy$$

Defining S as: $\begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix}$, we can write, in matrix form:

$$[x' \; y'] = [x \; y] \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix}$$

or

$$P' = P \cdot S$$

The triangle in ***Fig. 3.3a*** is scaled by a factor of 1/2 in the x-axis and 1/4 in the y-axis, resulting in that of ***Fig. 3.3b***. The result of the scaling, however, is not quite what you may have expected. The problem is that the scaling has been done about the origin. If the

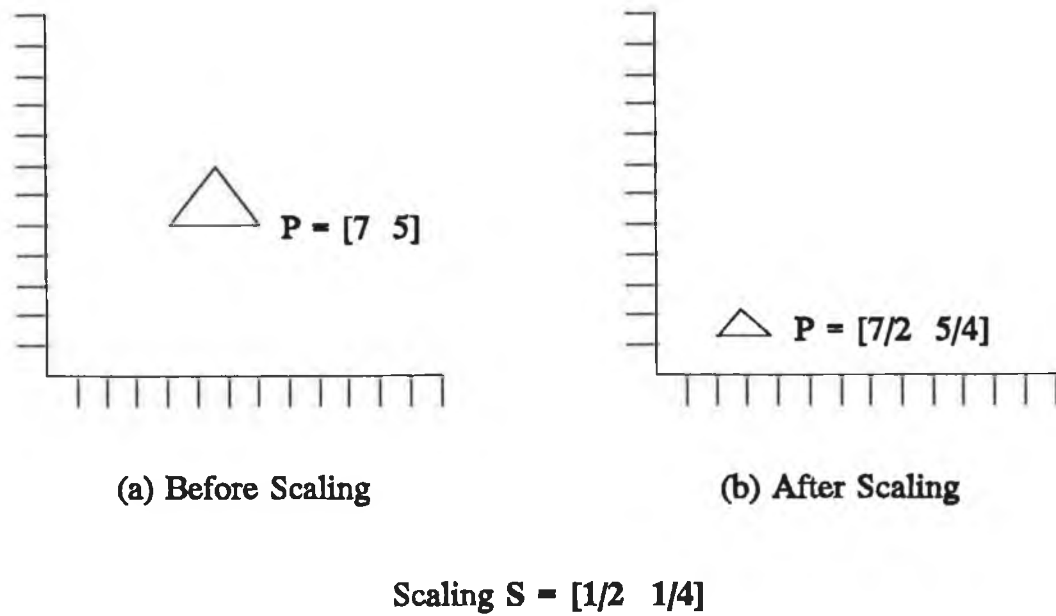


Fig. 3.3 Scaling in 2D

scaling factors were greater than 1, then the object would not only be enlarged, it would also be repositioned further from the origin.

In order to scale an object about a point, **P** (*Fig. 3.4a*), we first translate the object so

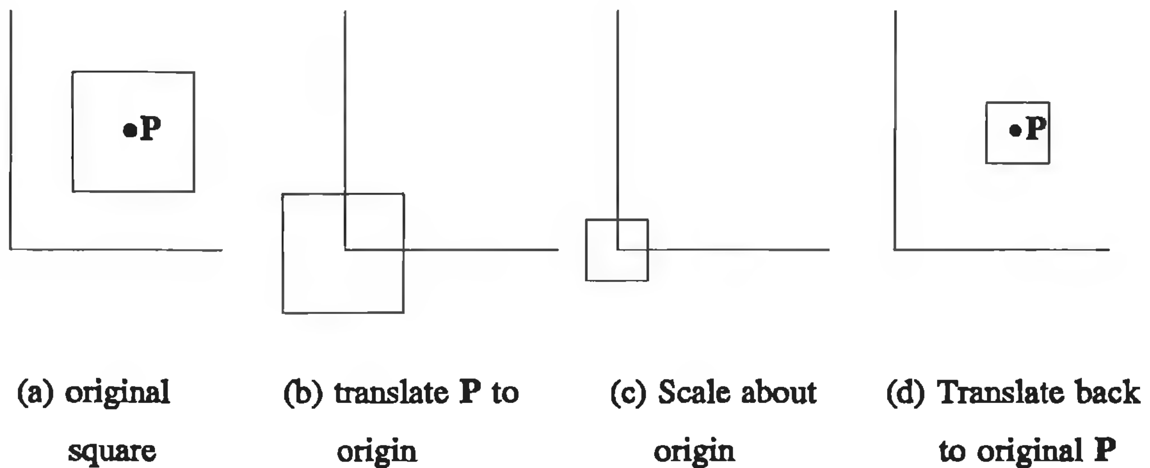


Fig 3.4 Composite translate/scale/translate in 2D

that **P** is at the origin (*Fig. 3.4b*). We then scale the object (*Fig. 3.4c*). Now, as **P** is at the origin, the object is not only scaled about the origin, but also about **P**. After we have scaled the object, we translate it back so that **P** returns to its original position (*Fig. 3.4d*).

Points can be *rotated* about the origin. The rotation is defined as:

$$x' = x * \cos(\theta) - y * \sin(\theta)$$
$$y' = x * \sin(\theta) + y * \cos(\theta)$$

In matrix form, this is:

$$[x' \; y'] = [x \; y] \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

or

$$\mathbf{P'} = \mathbf{P} \cdot \mathbf{R}$$

where **R** represents the rotation matrix defined above.

Fig. 3.5 shows a triangle rotated through an angle of 90^0. As with scaling, the object is rotated about the origin. To rotate about an arbitrary point, we follow the procedure described above for scaling (i.e. we translate to the origin, rotate, and translate back).

The need to translate every object before it is either scaled or rotated is a feature we would rather avoid. It is messy, and computationally expensive. Fortunately, we can overcome this problem by creating *compositions* of 2D transformations.

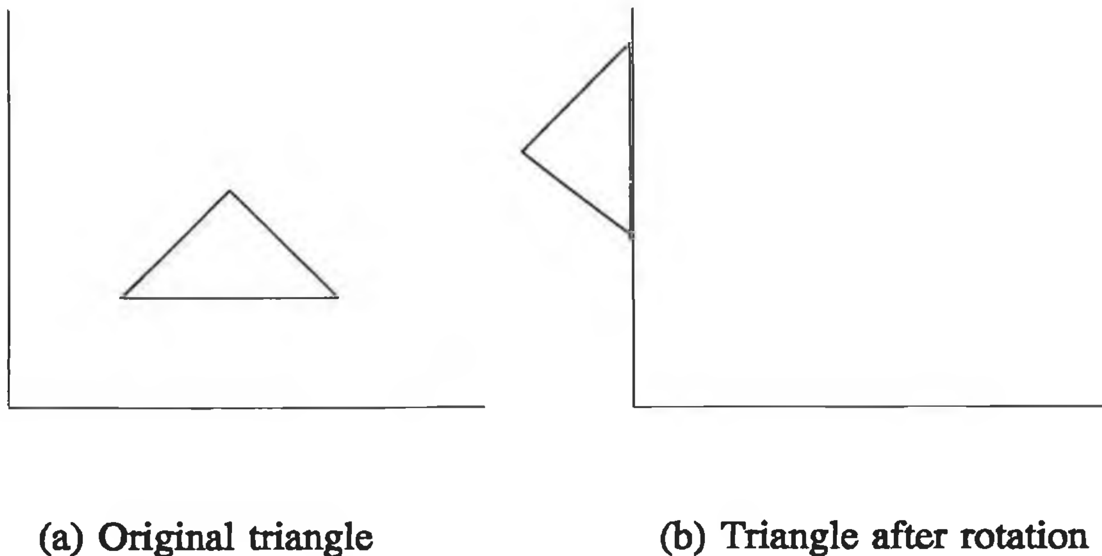


Fig. 3.5 Rotation in 2D

3.2.2 Homogeneous Coordinates

In *Section 3.2.1* we defined translation as an addition of two matrices, while both scaling and rotation were defined as the multiplication of two matrices. We require to be able to treat the three transformations in a *homogeneous*, or consistent, way so that the three transformations can easily be combined together.

Homogeneous coordinates were developed in geometry by Maxwell [MAXW46,MAXW51] and have subsequently been applied in graphics by Roberts [ROBE65] and Blinn [BLIN77b,BLIN78].

We can represent any cartesian point **P** = $[x\ y]$ by an equivalent homogeneous point **P** = $[W * x,\ \ W * y,\ \ W]$, for any scale factor $W \neq 0$.

P = $[x\ y]$	is represented as	**P** = $[W * x\ \ W * y\ \ W]$
P = $[W * x\ \ W * y\ \ W]$	is equal to	**P** = $[x\ y\ W]$

By setting $W = 1$:

	is equal to	**P** = $[x\ y\ 1]$

Therefore:

P = $[x\ y]$ (cartesian) is represented as **P** = $[x\ y\ 1]$ (Homogeneous)

Using homogeneous coordinates, the translation, scaling and rotation transformation matrices are all 3 X 3 matrices. Explicitly, they are defined as:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Dx & Dy & 1 \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where: Dx, Dy, Sx, Sy, and θ have the same meanings as in *Section 3.2.1.*

3.3.3 Composition of 2D Transformations

In order to translate, scale, and rotate points in 2D homogeneous coordinates, we multiply the point by the relevant matrix. Using homogeneous coordinates we can compound translation, scaling and rotation operations into one *transformation* matrix. When we wish to scale or rotate an object about a point, it is no longer necessary to translate the point to and from the origin before and after the actual scaling/rotation takes place. We can combine the three steps of translation, scaling/rotation, translation into one transformation matrix. For example, the composite translate/rotate/translate transformation matrix to rotate an object about some point **P** = $[x\ \ y\ \ 1]$, by an angle of θ, is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x & -y & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{bmatrix}$$

Translate *Rotate* *Translate*

$$= \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ x(1 - \cos(\theta)) + y\sin(\theta) & y(1 - \cos(\theta)) - x\sin(\theta) & 1 \end{bmatrix}$$

Composite Transformation Matrix

Henceforth, when we require to rotate any object by an angle θ, about the point **P**, we need only multiply each of the object's control points by this one transformation matrix, whereas before we required two translations and a rotation per point.

We can combine any number of translations, scalings, and rotations into one transformation matrix. This means that even the most complex transformation of a point can be carried out by multiplying that point by only one matrix.

The computations can be speeded up even further by considering the general layout of any 2D transformation matrix. All 2D transformation matrices have the following layout:

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

The upper 2 X 2 submatrix represents the composite rotation and scale matrix, while the lower 2 X 1 submatrix represents the composite translation matrix. Calculating **P·M** as a point multiplied by a 3 X 3 matrix requires nine multiplications and six adds. However, because the last column of **M** is fixed, we can reduce **P·M** to a total of four multiplications and four additions.

3.3.4 3D Transformations

Points are defined in 3D homogeneous coordinates in the same manner as described for 2D, except that now a *z* coordinate is included in the definition[1]. Therefore:

$$\mathbf{P} = [x\ y\ z\ 1]$$

The following are the general translation and scaling 4 X 4 matrices used in 3D homogeneous coordinate systems:

Translation: $$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Dx & Dy & Dz & 1 \end{bmatrix}$$

Scaling: $$\mathbf{S} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A rotation matrix must be defined for *each* of the three axes that are present in 3D. The rotational matrices given here are for a *right handed* coordinate system. They are:

Rotation about *x-axis*: $$\mathbf{R}x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[1] It is important to note that there are two ways of defining the coordinate system, each way depending on the direction of the +z axis. In the *left handed* coordinate system, the +z axis extends away from the viewer, while in the *right handed* coordinate system, the +z axis extends toward the viewer. In a left handed coordinate system positive rotations are made in a *clockwise* direction and in a right handed coordinate system rotations are in a *counterclockwise* direction.

Rotation about *y-axis*: $\mathbf{R}y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

Rotation about *z-axis*: $\mathbf{R}z(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The general compound transformation matrix (M), comprising translation, scaling and rotation is:

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

where: the upper left 3 X 3 submatrix represents the combined scaling and rotation and the lower left 3 X 1 submatrix gives the aggregate translations.

By using 3D homogeneous coordinates, we can perform any composite transformation on any point in 3D space. By applying the same transformation to all the control points for an object we can efficiently transform the object in 3D space.

3.3.5 Coordinate Systems

The ability to transform whole objects around in 3D space leads to the idea of *coordinate systems*. When generating computer models, we normally use three 3D coordinate systems. These are the *primitive*, *object* and *world* coordinate systems.

Each object that is to be modelled is developed in its own *object coordinate system*. The object is built by the combining of primitives and objects that have been transformed from *primitive coordinate systems* and other object coordinate systems. By using different

transformation matrices, the same object can be transformed to multiple locations in other coordinate systems. These locations can be other object coordinate systems or the world coordinate system.

For example, we can define a car in a *car object coordinate system*. For this car, we need four wheels. We need only define the wheel once in a *wheel primitive coordinate system*, and then use four separate transformation matrices to correctly position the four wheels in the car object coordinate system. We may, if we wish, transform the car object to one or more positions in the *world coordinate system*.

Primitives are at the lowest level of the object tree. They cannot be created by a combination of any other primitives. Primitives are like the nuts and bolts in engineering, or the bricks used to build houses.

The world coordinate system is the last level of transformation that can be done in 3D. It contains the final 3D scene that is being modelled.

The coordinate systems can be represented as an *object tree*, with the world coordinate system as the root, the primitive coordinate systems as the leaf nodes and the various object coordinate systems as the other nodes.

Using the structured approach of an object tree for modelling we may define very complex objects with relative ease. The structured approach also speeds up development time as we need create less object definitions.

3.3.6 Inverse Transform Matrix

Ultimately, the scene in the 3D coordinate system is built from the primitives of the leaf nodes from the object tree. When ray tracing, we must check each ray against every object that is within the scene. Rather than create unique intersection routines for each object in the scene, we instead test the ray for intersection against each of the leaf nodes from the object tree. We then need only describe intersection routines for each of the primitives.

In order to test for intersection we must ensure that the ray and the primitive are in the same coordinate system. The ray is cast in the world coordinate system and a primitive is defined in its own primitive coordinate system. The discussion so far has assumed that we would transform the primitive into the world coordinate system. As both the primitive and the ray will be in the same coordinate system, we can perform our intersection test. This method is correct, but there is a more efficient way.

Instead of transforming the primitive *up* into the world coordinate system, we can transform the ray *down* into the primitive's own coordinate system. As described in *Section 5.4.1*, a ray contains a point and a vector. This point and vector can be transformed down into the primitive's coordinate system by multiplying each by the primitives *inverse transform matrix*. The inverse transform matrix is quite simply the inverse of the transform matrix that would have transformed the primitive into the world coordinate system.

In the world coordinate system we have multiple copies of each primitive transformed into innumerable positions. Rather than writing general ray/primitive intersection routines that are necessary in the world coordinate system, we can make use of the fact that we only need a specific ray tracer for the pre transformed primitive in its own coordinate system. By carefully choosing the way in which we define the primitive we can make significant computational gains. The primitive types defined in *PRIME*[2], and discussed in *Section 5.4* are choosen so as to be effeciently implemented.

In order for a ray/surface intersection routine to be performed, we must be able to detect every point on the surface of a primitive. In *Section 3.2*, we stated that complete surfaces are formed by mapping a set of control points. We will now discuss two methods that are used to describe, or model, whole surfaces in terms of object control points. The operations come under two headings, *surface modelling* and *solid modelling*.

3.4 SURFACE MODELLING

In *surface modelling* we define individual objects by using surfaces. Each surface is defined by using a set of control points and a set of operations connecting the points. There are various methods used in surface modelling. We shall describe the methods of *polygonal*, *parametric*, and *fractal* surface modelling.

3.4.1 Polygonal Surfaces

A *polygon* is a closed plane that consists of three or more vectors, or straight lines, that connect three or more *vertices*, or points, with no sides intersecting. Specific polygons are named according to their number of sides, such as triangle and pentagon. We can

[2] *PRIME* Photo*R*ealistic *I*mage *M*odelling *E*nvironment.

represent any plane surface precisely by using a polygon[3]. Therefore, we can represent any object consisting wholly of plane surfaces by using a number of polygons, as shown in *Fig. 3.6*. Polygonal surfaces are cheaper to ray trace than surfaces created by using other surface modelling methods, and are therefore preferable whenever possible.

Curved surfaces can be modelled by using a number of polygons to approximate the curve. Methods used to approximate curved surfaces as polygonal surfaces are discussed by

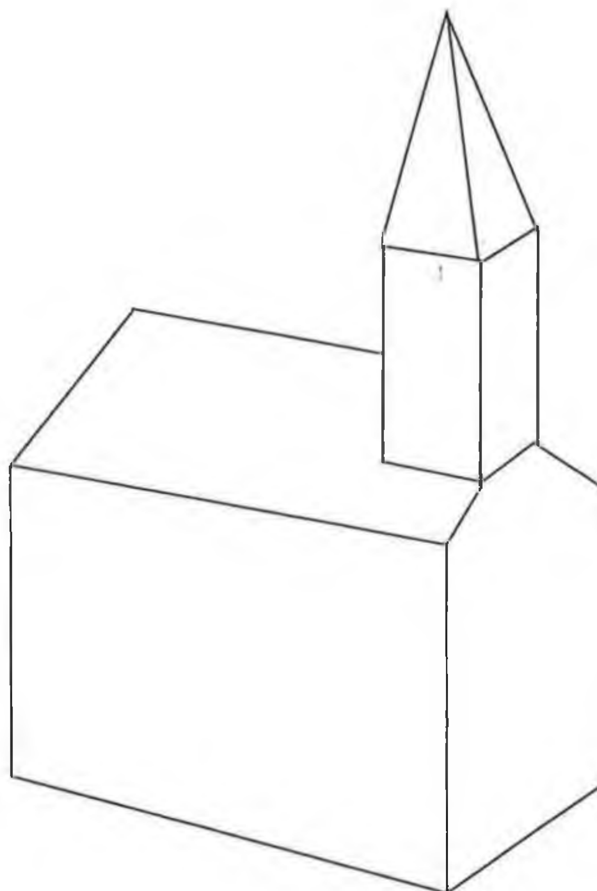


Fig. 3.6 Object constructed using polygons

Ganapathy and Dennehy [GANA82], Synder and Barr [BARR87a], and Von Herzen and Barr [BARR87b]. When generating photorealistic images (ray tracing), it is not always sufficient to merely approximate curved surfaces. Fortunately, we can use *parametric surfaces* to model curved surfaces exactly.

[3] Other regular plane surfaces, such as a circle, are normally included in modelling as being polygons.

3.4.2 Parametric Surfaces

Parametric surfaces are more expensive than polygonals to compute, but parametric surfaces accurately model the curvature of a surface. There are several parametric surface modelling methods available. We shall restrict our discussion to that of the *Beta-Spline*, or *ß-Spline*. Discussions on other parametric surface modelling methods include those by Kajiya [KAJI82], Barr [BARR86], Joy and Bhetanabhotla [JOY86], Plass and Stone [PLAS83], and Toth [TOTH85].

The word *spline* comes from boat building, where planks of wood were formed into shape by bending around pegs hammered into the ground. Originally, it was the pegs that were called splines, though now the word is used to describe the curve itself. As a prelude to ß-spline surfaces, we shall discuss ß-spline curves. These are a 2D equivalent of the 3D ß-spline surface.

When talking about splines, we refer to the control points as *knots*. There are two ways in which ß-splines can be constructed: *interpolation* and *approximation*. By using interpolation the curve passes through each knot. By using approximation, the curve might not necessarily pass through each knot. Approximation is easier to compute, but is not as accurate as interpolation.

Much about ß-spline curves can be described using the simple *Bezier curve*. A typical Bezier curve is shown in *Fig. 3.7*. The Bezier curve is an approximation method, as the generated curve does not pass through all four defining knots. *Fig. 3.8* shows the first six steps of the construction of the Bezier curve from *Fig 3.7*. Each knot is joined by a line segment. A parametric distance, α, is chosen in the range 0 to 1. This gives a new set of knots, one less in number than the original set of knots. Again, these knots are joined by line segments and a new set of knots is marked on the new line segments at a distance of α along each line segment. This recursive process continues until only a single point is left.

All Bezier curves go through both of their endknots and the slope of any Bezier curve is continuous along its entirety. The curve is clamped to its endknots. All the other knots exert a *blend*, or pull, on the curve.

When modelling we usually need to create a precise curve. Using parametric curves, the precision is normally achived through an iterative process. We use an initial set of knots that we know are close to, but not necessarily, the best knots for the curve we desire. By changing the curve locally, we ultimately obtain the precise curve we desire. As shown in *Fig*

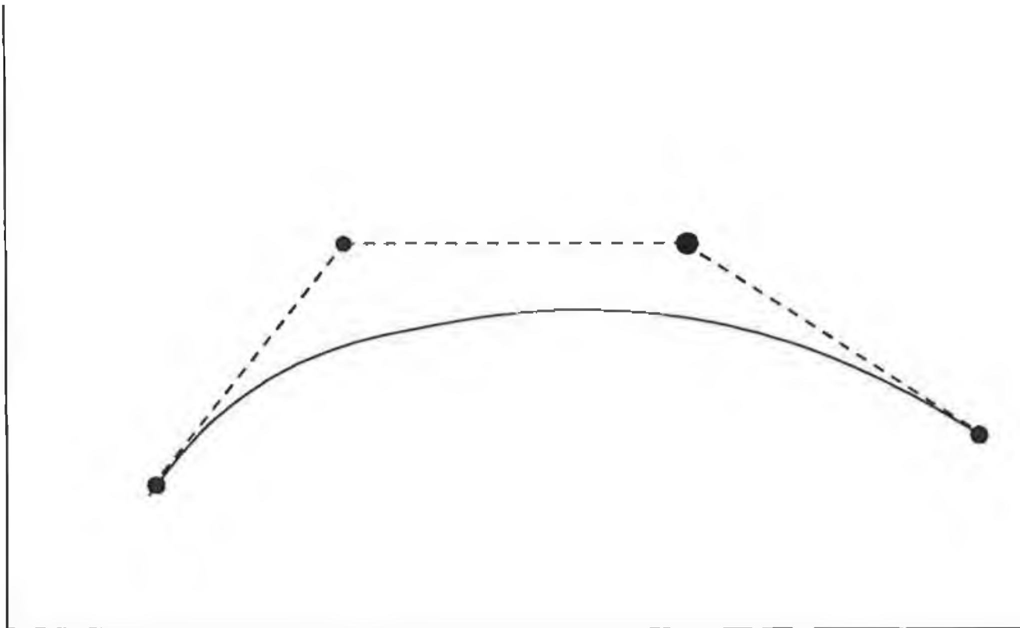


Fig. 3.7 Bezier curve, using four knots

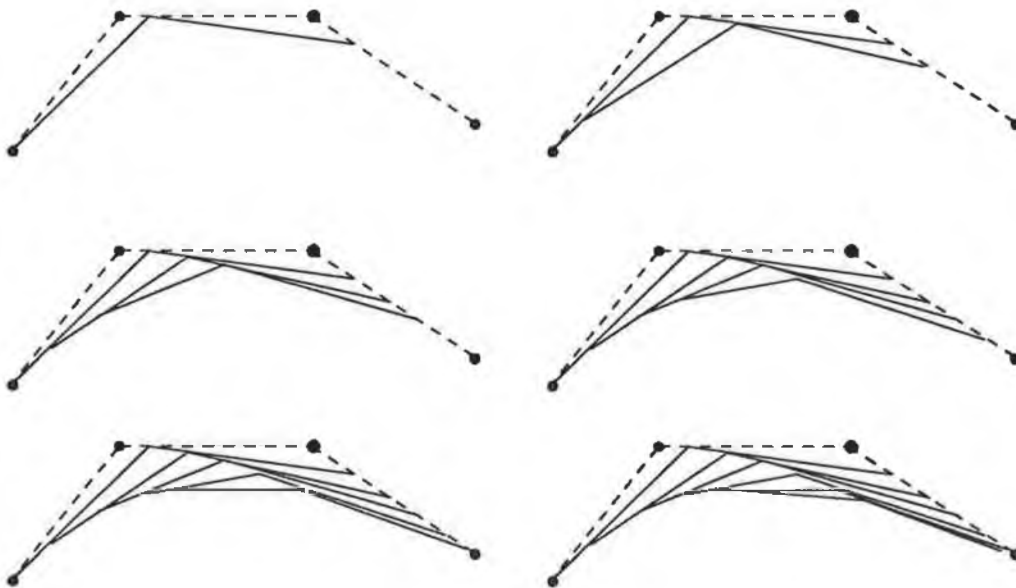


Fig. 3.8 Construction of Bezier curve

3.8, when constructing a Bezier curve, all points that are on the curve are dependent on the position of every one of the original knots. If we need to change a Bezier curve at some

specific locality, we have to reposition all of the knots that define this curve. This is not too bad if we have a Bezier curve generated using only a few knots, but as we define more complex Bezier curves, needing many knots, it becomes infeasible to attempt to change the curve locally.

A second problem when using Bezier curves is that discontinuities arise in a curve when an attempt is made to loop it, as shown in *Fig. 3.9.*

When we wish to have local control and guaranteed continuity of the curve, we use ß-splines.

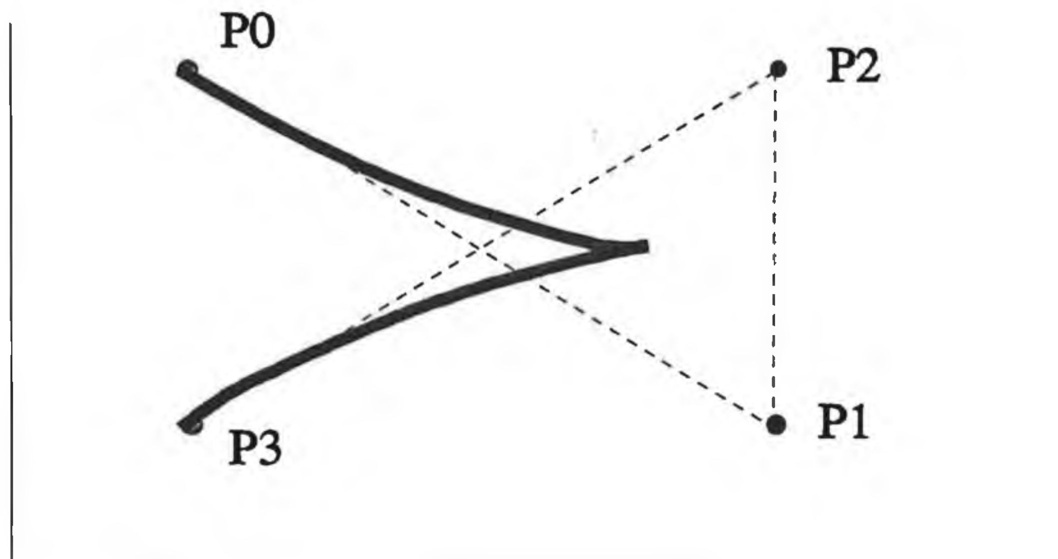


Fig. 3.9 Discontinuities in the Bezier curve

In principle, ß-splines are similar to Bezier curves. The main difference is the choice of blending (or pulling) function. While the Bezier curve takes all knots into account, the ß-spline takes into account only the four most local knots to any point on the curve. No matter how many knots are used to define a curve, we need change only four knots to change the curve at any given locality.

ß-splines allow us to exert exact control over the slope of curve, with the use of *skew* and *tension*. These two attributes are defined for every knot. The skew sharpens the slope of the curve. For high skew values (skew > 1), the slope of the curve is no longer continuous.

The effect of skew does not correspond to any easily described property in design, and so tends to be used very rarely. Skew is shown in *Fig. 3.10*. Tension is used to pull the curve closer to a knot, thus giving local control of the slope of the curve. For high tension values (tension > 100), the resulting curve is closer to being linear. Tension is shown in *Fig. 3.11*.

The concepts of ß-spline curves are readily transferred up to 3D ß-spline surfaces. The

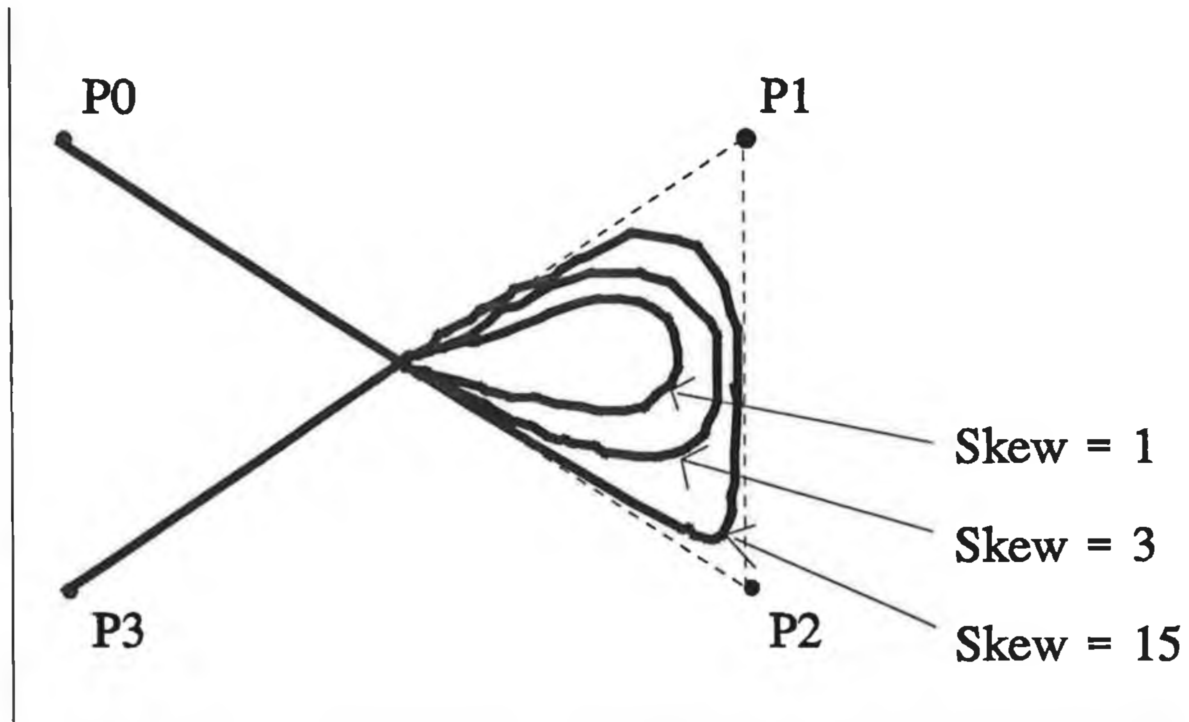


Fig. 3.10 The effects of different *skew* values on a curve.

knots in ß-spline surfaces are stored as a 2D array. ß-spline surfaces retain the desirable property that any change to the surface is kept local, with only the sixteen nearest knots (four by four in the 2D array) being effected. *Colour Plate 1* shows a wire frame of a ß-spline surface.

Detailed mathematical descriptions on the modelling of ß-splines can be found in Barsky's definitive book on the subject [BARS88], and also in papers by Barsky and Beatty [BARS83] and by Schaffner [SCHA81].

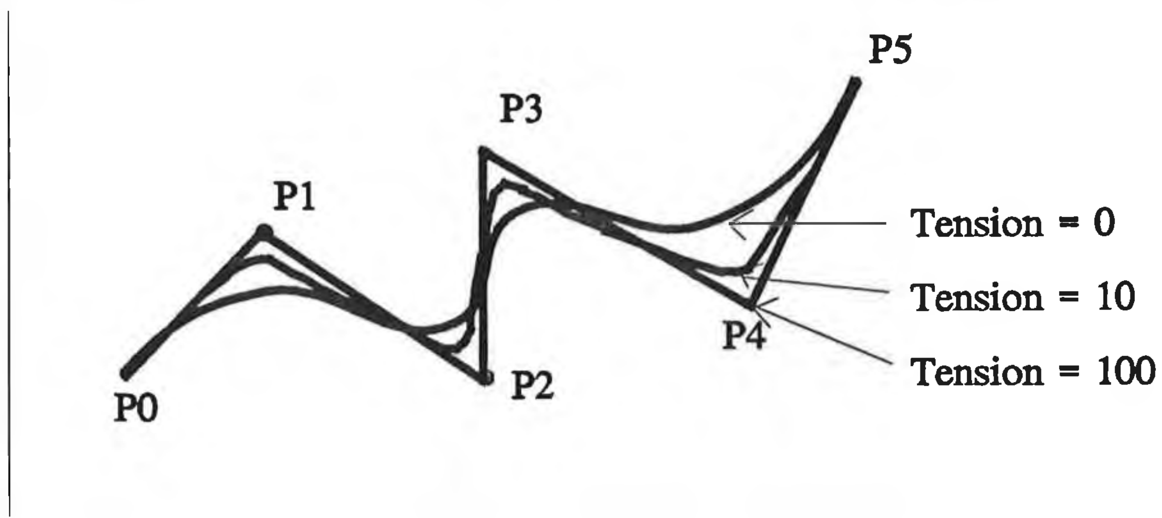


Fig. 3.11 The effects of different *tension* on a curve.

3.4.3 Fractal Surfaces

Fractal surfaces are recursively built, with every level being based on the same mathematical definition. No matter how closely we look at a fractal's surface, we will see the same pattern repeating itself. Fractal surfaces are ideal for the modelling of naturally occurring objects, such as the sea, mountains, and trees.

When designing a fractal model, a set of *variance parameters* must be given, stating the variance allowed by the recursion. The variance parameters ensure that the recursion does not follow some completely chaotic direction. The variance parameters also control the general 'look' of the object. We will take the generation of a tree as an example of a fractal object.

For this model we need five variation parameters. The first parameter needed is the angle between the first and last branches at any level of the tree. We also need a parameter to determine the ratio between the branch angles at consecutive levels, a parameter that states the maximum number of branches that emanate at any level, a parameter that states the ratio between branch lengths at consecutive levels, and finally, a parameter that states the number of levels of the tree. The first four variance parameters will have a random element added. The last variance parameter controls the size of the tree, thus ensuring that it will fit into any given scene. Two sample fractal modelled trees are shown in *Fig. 3.12*.

For fractal objects such as the coastline of a country, we must first create an outline of the object (i.e. draw a rough map) and then tie the recursion around this outline. The

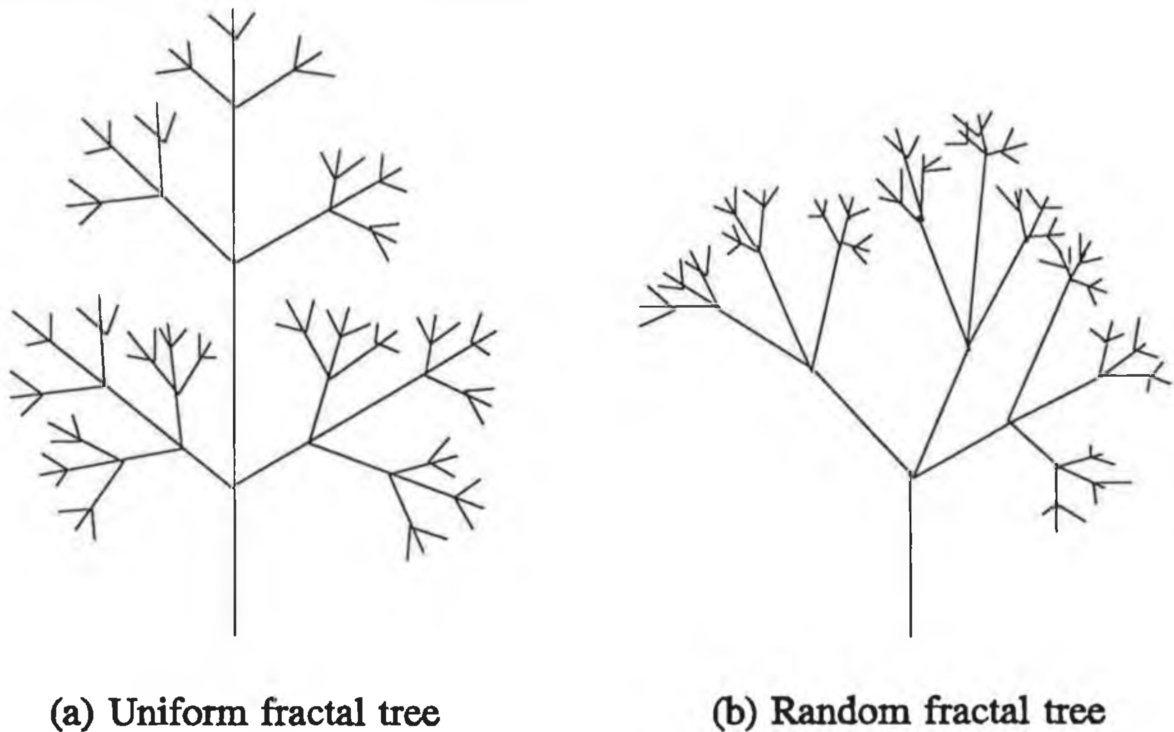


Fig. 3.12 Comparisson of uniformly and nonuniformly distributed fractal modelled trees.

outline can be generated by using polygons. At a local level the recursion is controlled by a set of variance parameters, but at a higher level it is tied into the general outline of the coast. Fractals are studied in detail by Mandelbrot [MAND83].

3.5 SOLID MODELLING

In *solid modelling* we use solid objects, rather than surfaces, to model a scene. Surface and solid modelling are mutually exclusive. The freeform surfaces that can be modelled well by using surface modelling are not good candidates for solid modelling. Solid modelling is best suited to the modelling of non-naturally occurring objects, such as the objects in engineering and architecture.

There are many primitives that can be used within a solid modelling ray tracer. The major prerequisite for any primitive is that it must be a solid. We now discuss solid modelling using *polyhedra* and *quadrics*, and we note other solid modelling methods currently in use.

3.5.1 Polyhedra

Polyhedra are constructed by the joining together of several polygons, so as to form a solid. For a polyhydral object to be a solid, one face of each polygon must point out away from the object and the other face of each polygon must point into the object. The cube and the pyramid are examples of polyhydral objects. Both the cube and the pyramid are used in *PRIME*. Their respective mathematical definitions are given in *Section 5.4.2.6* and *Section 5.4.2.7*.

3.5.2 Quadrics

Quadrics were used in the earliest ray tracers developed, [GOLD71]. Quadrics are ***second degree***, ***implicit*** surfaces. Because they are second degree, no term in a quadric will ever be more than squared. This is significant, as a ray can only ever intersect a quadric twice. This is a very desirable property, as it is the minimum number of intersections that a ray can have with any solid object[4]. Because they are implicit, they are defined by only one point and an equation. Some quadrics are the *sphere*, *cylinder*, and *cone*. In *Section 5.4* we derive the exact ray/quadric intersection routines for each of the above three quadrics. Here, we shall describe the general properties of all quadrics, and how they are defined so as to be easily ray traced.

Sphere

The implicit equation of a sphere is:

$$(X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Y_c)^2 = S_r^2$$

where:

S_c ≡ Sphere's centre ≡ $[X_c \;\; Y_c \;\; Z_c]$

S_r ≡ Sphere's radius

Sphere's surface is the set of points $[X_s \;\; Y_s \;\; Z_s]$.

[4] We exclude the case where a ray emanates from inside an object. This can only occur if a modeller is foolish enough to place the image plane inside the 3D world coordinate system!

The equation of a ray, as per *Section 5.4.1*, is:

$$\mathbf{R}(t) = \mathbf{R}_o + \mathbf{R}_d t$$

where:

$\mathbf{R}_o \equiv [X_o \ \ Y_o \ \ Z_o]$

$\mathbf{R}_d \equiv [X_d \ \ Y_d \ \ Z_d]$

t is the rays distance parameter

Substituting the ray equation into the sphere's equation and solving for t gives:

$$(X_o + X_d t - X_c)^2 + (Y_o + Y_d t - Y_c)^2 + (Z_o + Z_d t - Z_c)^2 = S_r^2$$

In terms of t, this simplifies to:

$$t^2(A) + 2t(B) + C = 0$$

where:

$A = X_d^2 + Y_d^2 + Z_d^2$

$B = X_d(X_o - X_c) + Y_d(Y_o - Y_c) + Z_d(Z_o - Z_c)$

$C = (X_o - X_c)^2 + (Y_o - Y_c)^2 + (Z_o - Z_c)^2 - S_r^2$

This is the general form for any ray/quadric intersection equation. It gives t in terms of a quadratic equation. The solution for t, therefore is:

$$t = \frac{-2B \pm \sqrt{((2B)^2 - 4AC)}}{2A}$$

or

$$t = \frac{-B \pm \sqrt{(B^2 - AC)}}{A}$$

By carefully choosing the primitive coordinate system, it is possible to cut down on some ray/quadric intersection time. If we declare the sphere to be the unit sphere, centred at the origin, then we can replace:

$X_o - X_c$ with X_o
$Y_o - Y_c$ with Y_o
$Z_o - Z_c$ with Z_o

Also, as the ray direction vector is normalised:

$$A = X_d^2 + Y_d^2 + Z_d^2 = 1$$

This means we can drop A out of the equation, giving:

$$t = -B \pm \sqrt{(B^2 - C)}$$

where

$B = X_d X_o + Y_d Y_o + Z_d Z_o$
$C = X_o^2 + Y_o^2 + Z_o^2 - 1$

The ray/quadric intersection equations for the cylinder and cone are:

Cylinder

The canonical equation for an infinite cylinder is:

$$X^2 + Y^2 - 1 = 0$$

Substituting the ray equation into this gives:

$$t^2(X_d^2 + Y_d^2) + 2t(X_o X_d + Y_o Y_d) + (X_o^2 + Y_o^2) - 1 = 0$$

Therefore:

$$t = \frac{-B \pm \sqrt{(B^2 - AC)}}{A}$$

Where:

$$A = X_d^2 + Y_d^2$$
$$B = X_oX_d + Y_oY_d$$
$$C = X_o^2 + Y_o^2 - 1$$

Cone

$$A = X_d^2 + Y_d^2 - Z_d^2$$
$$B = X_oX_d + Y_oY_d - Z_oZ_d$$
$$C = X_o^2 + Y_o^2 - Z_o^2$$

We can easily derive other quadrics along the same lines as the solutions shown above. Other quadrics include the ellipsoid, the hyperboloid, and the paraboloid. Quadrics are discussed by Roth [ROTH82], and Bier [BIER83].

3.5.3 Other Forms Of Solid Modelling

A large variety of objects can be generated by using *swept surfaces*. A swept surface is defined by a planar curve that is moved along its normal, so as to form a cylindrical object. A second swept surface allows the radius of the cylinder to change, so forming a cone like object. Cylinder and cone swept surfaces are discussed by Goldstein and Nagel [GOLD71], Wijk [WIJK84], Bier [BIER83], and Kajiya [KAJI83].

Instead of sweeping a surface along it normal, we can sweep the surface freely in space. The background theory to swept surfaces is described by Faux [FAUX79].

Another technique for creating primitives is to use *surfaces of revolution*. Here, a curve is revolved around an axis. A vase is an example of a surface of revolution.

Ray tracing algebraic surfaces, including quadrics, and higher order surfaces is discussed by Hanrahan [HANR83].

Various other methods of modelling (both surface and solid) are outlined by Glassner [GLASS89].

3.6 CONSTRUCTIVE SOLID GEOMETRY

Constructive solid geometry, (*CSG*), is a solid modelling method. Simple primitives, such as polyhedra or quadrics, are transformed in 3D space and combined together to form solid objects.

CSG is the method of choice for a wide range of applications of engineering design. Objects are built through boolean operations on primitive and intermediate solids in CSG. This reflects the way in which many engineering products are actually manufactured.

Descriptions of CSG systems are given by Boyse and Gilchrist [BOYS82], that describes GMSolid, an interactive modeller for the design and analysis of solids; by Brown [BROW82], who gives a technical summary of a system called PADL-2; by Requicha and Voelcker [REQU82], giving a general overview of solid modelling; and by Myers [MYER82], who views the area from an industrial perspective.

CSG objects can be represented by a CSG binary tree. The leaf nodes of a CSG tree consist of primitive solids that have been transformed into the object's coordinate system[5]. All other nodes contain boolean operators. There are three boolean operators allowable in CSG; *union*, *intersection*, and *difference*.

The final object represented by a CSG tree will be in the tree's root node. It's content is found recursively, by replacing each operator node with the result obtained when that operator has been performed on its two child nodes.

The result of performing the *union* operator on two child nodes is that the union node is replaced by everything that was in either of the child nodes.

The result of performing the *difference* operator on two child nodes is that the difference node is replaced by everything that was in the right child node, but not in the left child node.

The result of performing the *intersection* operator on two child nodes is that the intersection node is replaced by everything that was in the right child node and also in the left child node.

[5] It is conceptionally easier to imagine primitives being transformed up into the world coordinate system, even though, as shown in *Section 3.?*, it is the rays in the world coordinate system that are transformed down into the primitive's coordinate systems. The difference, in terms of implementation, is whether to use the primitive's transform matrices or their inverse transform matrices.

When the whole CSG tree has been recursively processed, the result contained in the root node represents the completely built object.

Diagrammatically, union is represented by +, difference by -, and intersection by &. A CSG constructed object is shown in *Fig. 3.13*.

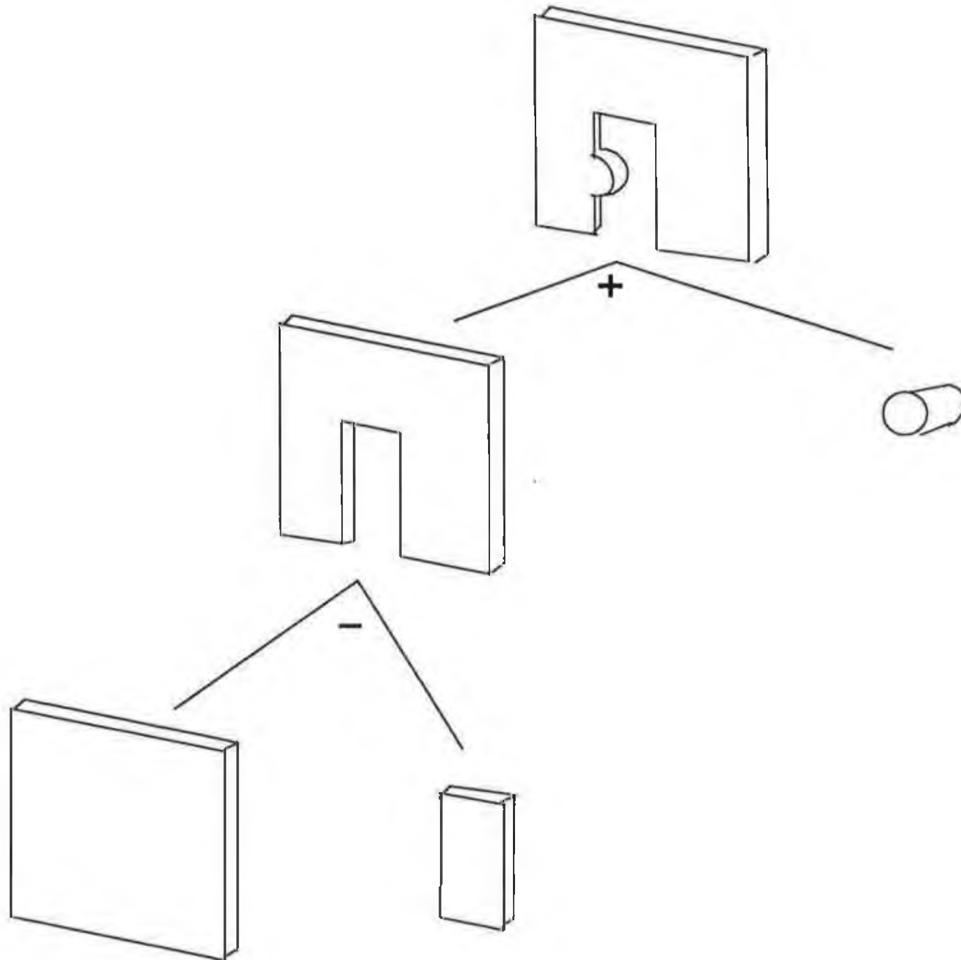


Fig. 3.13 Object constructed using CSG

When ray tracing, it is necessary to classify a ray with respect to the object against which the ray is being intersected. A *Roth diagram* can be used to describe the contents of a CSG tree with respect to a given ray. At any point along the rays path, it is classified as being *outside*, *on*, or *inside* the surface of an object. The ray path through the object can be represented using a Roth diagram, as shown in *Fig. 3.14*. The ray/object intersection points are calculated and a *ray classification interval* is drawn up for each leaf node. The

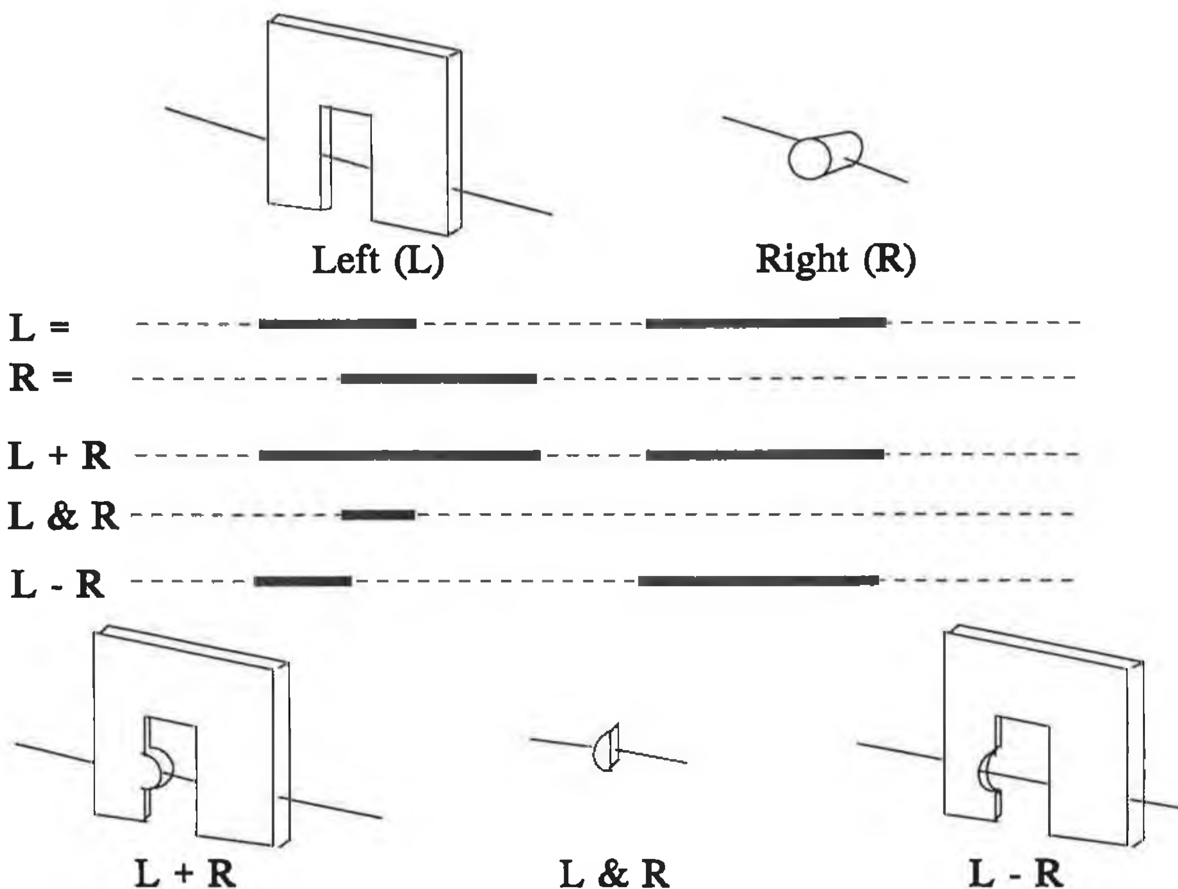


Fig. 3.14 Roth Diagram showing *union*, *intersection* and *difference* operators

classification intervals from two child node's can be combined into one classification interval, depending on the operator type of their parent, as shown in *Fig. 3.14*. The final classification interval replaces the operator at the parent node of the two children. The parent node is then treated as a child of the operator from the next level up the CSG tree. The rules for combining classifications in a CSG tree are shown in *Table 3.1*.

Set operator	*Left*	*Right*	*Composite*
Union	in	in	in
	in	out	in
	out	in	in
	out	out	out
Intersection	in	in	in
	in	out	out
	out	in	out
	out	out	out
Difference	in	in	out
	in	out	in
	out	in	out
	out	out	out

Table 3.1 Rules for combining classifications in a Roth Diagram

Roth also observed some efficiencies that are easily included into any CSG ray tracer. If the operator is intersection or difference, and the left child returns an empty Roth classification interval (i.e. the left child is not intersected by the ray), then it is pointless to process the right child, as the final Roth diagram must be empty.

3.7 SPEEDING THINGS UP

Working out the *ray/surface* intersections points is perhaps the largest computational expense in ray tracing. The number of such intersections to be calculated is directly proportional to the number of primitives in a modelled scene. We may have a scene containing several thousand primitives. Every ray must be checked against each primitive to check if an intersection occurs. Clearly, if we can reduce the number of intersection calculations by even a small percentage, then the computational savings will be great. There are two major methods that can be employed to reduce the number of *ray/surface intersections*. They are, by using *fewer rays* and by obtaining *faster intersections*.

We can shoot fewer rays into the scene by reducing the number of rays shot through each pixel. We have already described such methods in *Section 2.6*, examples being *adaptive tree-depth control*, *statistical optimizations for anti-aliasing*, *beam tracing*, and *cone tracing*.

However, even after these reductions, we must still shoot at least one ray through each pixel. On an average graphics monitor we have a screen resolution of 1024 X 768 pixels, this means we must shoot 786,000 pixel rays. It we have only 100 primitives in our scene, we will need to perform over 78 million ray/primitive intersection tests. If we take into account the

new reflected and transmitted rays that are recursively generated with every ray/surface intersection, we would most likely need to perform over (a conservatively estimated) 700 million ray/primitive intersection tests.

We can perform faster ray/primitive intersection tests, by either implementing faster intersection routines for the various primitive types, or by cutting down on the number of objects that are checked for intersection with each ray. Fast intersection routines are derived for each primitive type implemented in the *PRIME* system. These are described in detail in *Section 5.4*.

Here we shall deal with ways of reducing the number of ray/surface intersection tests needed. There are three methods employed that reduce the number of surfaces checked for intersection with any ray. They are *bounding volumes*, *space subdivision*, and *directional techniques*. A ray tracing system may implement one, two, or all three of these methods in order to obtain maximum efficiency.

3.7.1 Bounding Volumes

A *bounding volume*, also known as an *extent* or an *enclosure*, is a volume that fully encloses an object. The object is only tested for intersection with a ray should that ray first intersect the bounding volume of the object. Although an extra intersection is calculated for those objects that are in the path of a ray, the overall number of intersections is reduced. In a typical scene only a tiny percentage of objects are intersected by any given ray. On the other hand, only one intersection test must be performed for each of the other objects in the scene before they are rejected. When the intersection between a ray/bounding volume fails, we can reject the object contained within the volume. As a typical object will consist of many primitives, the final number of intersection tests performed will be greatly reduced.

Just like an object, a bounding volume can be of any shape. To test a bound for intersection is the same as testing an object for intersection.

There are two factors that need to be taken into account when designing a bounding volume; *tightness of fit* and *cost of intersection of the volume*. Simpler bounding volumes can be tested for intersection more quickly than can complex ones. Because simpler bounds may not tightly bound their enclosed object, there is a higher chance that these will be intersected by a ray that does not intersect the object that is being bounded. Should this happen, the enclosed object must itself be tested. This, of course, defeats the whole purpose of bounding

volumes. On the other hand, it may be more efficient to use a simple bound that is subjected to accepting the odd non good ray, rather than use a complex bound that requires a computationally expensive ray/bound intersection test. Weghorst *et al.* [WEGH84] investigate the tightness of fit versus cost of intersection trade-offs of bounding volumes.

Bounding volumes need not only be used to enclose whole objects. For complex objects, containing many primitives, we can build a *hierarchy* of bounding volumes. Rubin and Whitted [RUBI80] discuss the issue of bounding hierarchies.

Kay and Kajiya [KAY86] describe the use of *convex hulls* as bounding volumes. A convex hull bounding volume can be tailor made to fit any object. Kay and Kajiya use a set of *plane couples* that completely enclose an object to approximate its convex hull. Three of the plane couples must be linearly independent, thus guaranteeing that the object is properly bounded on all sides in 3D space. Each plane couple is defined by taking any two planes

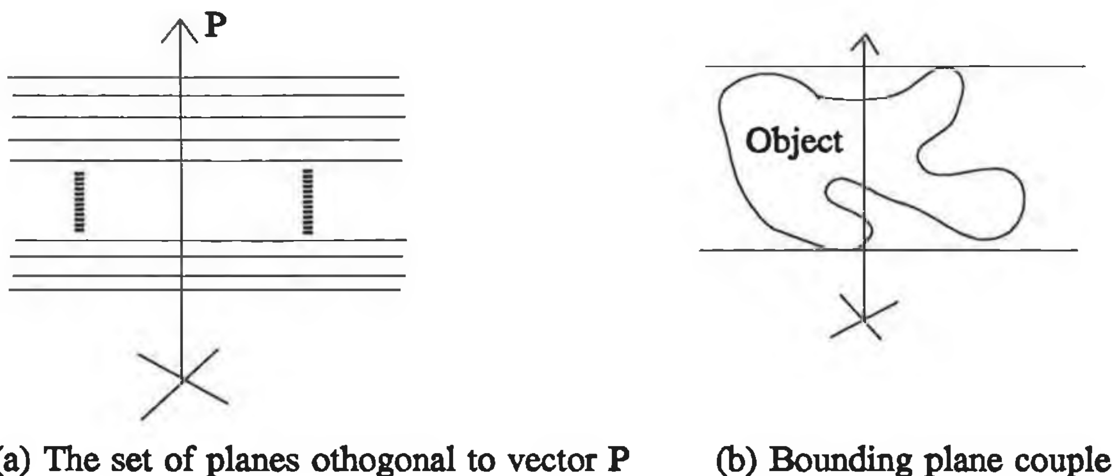


Fig. 3.15 The use of *plane couples* as bounding volumes

from the infinite set of orthogonal planes along any vector in 3D space, as shown in *Fig. 3.15a.* The two planes that best bound an object are chosen as the object's plane couple along that vector, as in *Fig. 3.15b.* By using more and more plane couples, we form a tighter bound around the object. Testing for a bounding volume/ray intersection requires two ray-plane intersections for each plane couple. There is a trade-off between the tightness of fit obtained by using many plane couples versus the extra computational cost of working out the intersections for each new plane couple.

3.7.2 Spatial Subdivision

To implement *spatial subdivision* we must ensure that the coordinate system is a cuboid volume. That is, it must be a cube in 3D space. Spatial subdivision links each object with one or more *voxels*, or sub-volumes, of the original world coordinate volume. As each new object is transformed into the world coordinate system, it is linked with each of the various voxels that contain part of its volume. The voxels act as bounding volumes to the objects contained within them. A ray cannot intersect an object unless it also intersects a voxel containing the object.

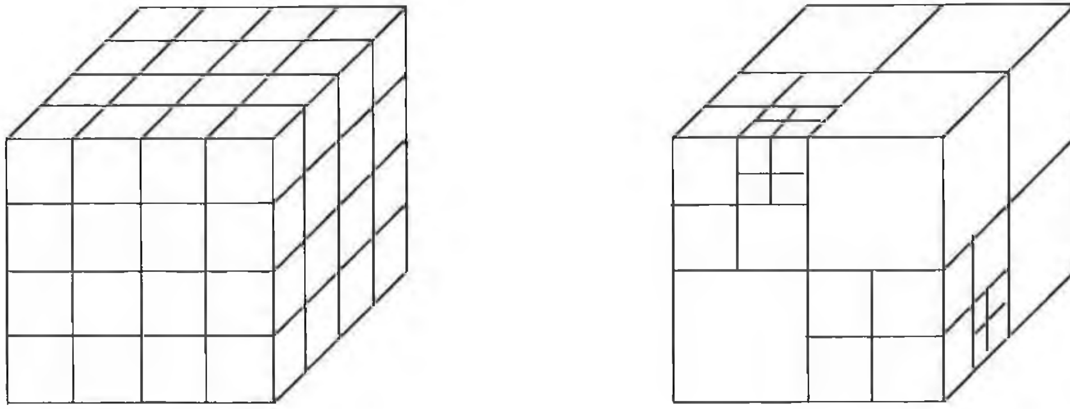
Every ray must travel along a straight path, thus imposing a strict order as to when any voxel is intersected by a ray. Any object intersected within a voxel must be closer to the ray's origin than any object within the voxels that lie further along the rays path. Once we find a ray/object intersection in any voxel, we can safely stop testing voxels further along the ray's path. This will normally lead to a large reduction in the number of objects that need to be tested against a given ray.

We can subdivide the space either by *uniform* or *nonuniform* space subdivision. Both methods prove to be superior to each other under differing circumstances.

3.7.2.1 Uniform Spacial Subdivision

Using uniform space subdivision, we simply divide the world coordinate system into a regular grid of equally sized voxels, as shown in *Fig. 3.16a.* Using this method, no account is taken of the distribution of the objects in the world coordinate space. We often have a situation where many voxels are empty, while a few voxels contain the majority of the objects. In such cases, spacial subdivision is actually detrimental to the efficiency of the ray/surface intersection routine, and nonuniform space subdivision should be used.

When using uniform spacial subdivision, we must decide how many voxels to create. If we have too few voxels, there will be no real division of the space, and so no real computational gains can be expected. If we have too many voxels, the overhead of intersecting many individual voxels may supersede any gains made in ray/object intersection that resulted from using the voxels in the first place. Another problem with using many voxels is that it may lead to memory overload. Glassner [GLAS84] curtailed this problem by storing only those voxels that contained one or more objects.



(a) Uniform spacial subdivision **(b) Nonuniform spacial subdivision**

Fig. 3.16 Dividing a cuboid using uniform and nonuniform spacial subdivision

3.7.2.2 Nonuniform Spacial Subdivision

When objects are nonuniformally distributed in the world coordinate space, we use nonuniform spacial subdivision. In nonuniform spacial subdivision, we set a maximum number of objects that can be associated with any one voxel. Once any voxel exceeds this maximum, it is subdivided into a set of smaller voxels. The usual method of doing the subdivision is to divide the voxel into eight equally sized voxels.

In nonuniform spacial subdivision, the size of the various voxels is not necessarily constant. Each voxel is subdivided independently to all others, with the subdivision process determined by the number of objects penetrating the voxel. Nonuniform spacial subdivision is shown in *Fig. 3.16b.*

However, where the objects are uniformly distributed throughout the world coordinate space, uniform spacial subdivision proves to be more efficient than nonuniform spacial subdivision. This is because it is computationally less expensive to calculate the voxels that lie on the path of a ray for the regular voxels of the uniform spacial subdivision algorithm than it is to calculate them for the nonuniform spacial subdivision. Fujimoto *et al.* [FUJI86] have implemented an efficient algorithm for this purpose.

3.7.3 Potential Pitfalls In Spacial Subdivision

In both uniform and nonuniform spacial subdivision, we may get repeated ray/object intersection testing, as shown in *Fig. 3.17*. This may be caused when an object is included

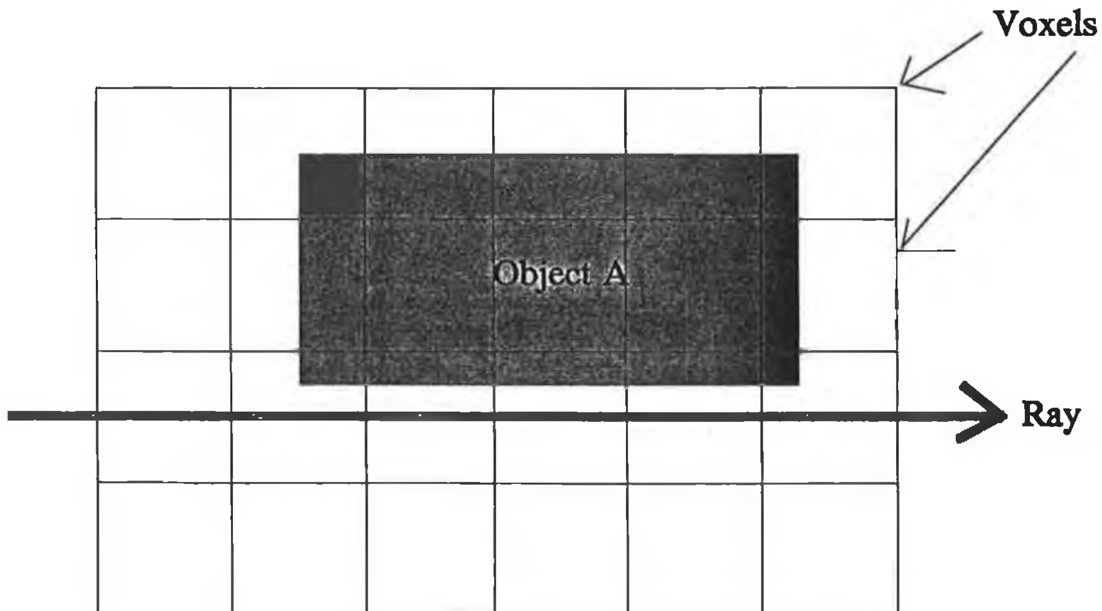


Fig. 3.17 The need for object *mailboxes* when using spacial subdivision

in more than one voxel. **Object A** is tested for intersection against the ray five times. To avoid this, we use a *mailbox*, as described by Arnaldi *et al.* [ARNA87]. Each object is given a mailbox and each ray is given a unique number. When a ray is tested for intersection against an object, the result of the intersection and the ray number are stored in the object's mailbox. Before testing any object for intersection against a ray, the ray numbers contained in its mailbox are compared against the ray's number. If both numbers match, then the result of the intersection can be obtained from the mailbox, without any need to carry out a ray/object intersection.

A second potential problem that must be accounted for is shown in *Fig. 3.18*. In voxel **1**, the ray must be tested against **Object A**. An intersection is detected, as the ray will actually intersect **Object A** later along its path. In order to avoid returning **Object A** as the nearest object intersected by the ray, we accept as valid only those intersections that occur

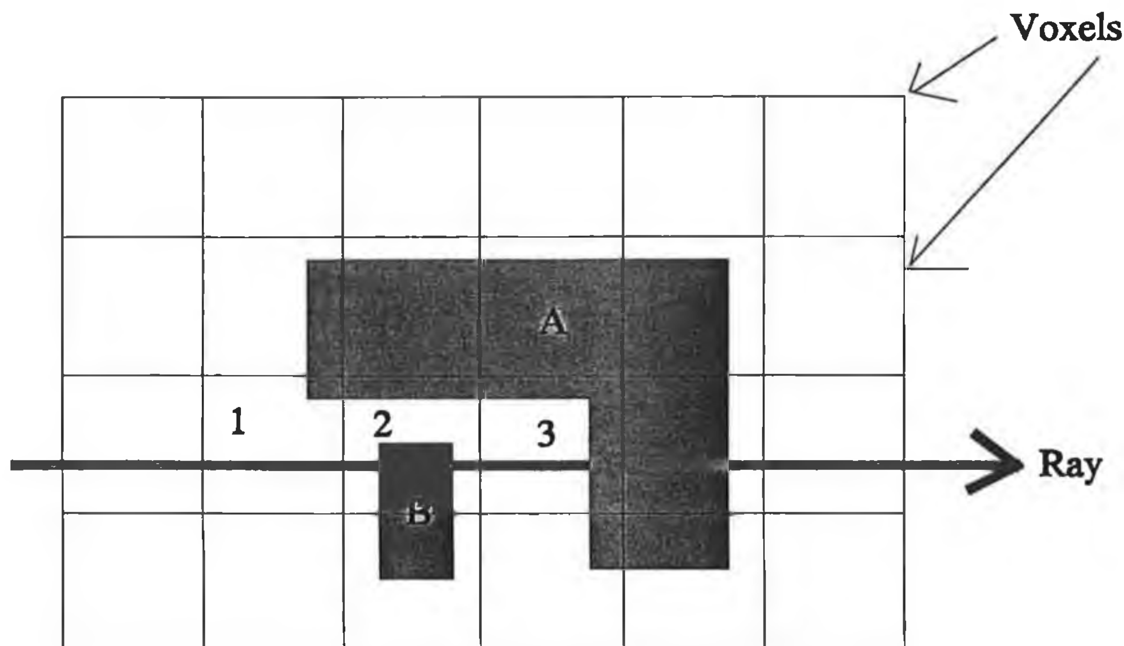


Fig. 3.18

in the current voxel. Under this condition, **Object A** would be rejected as a valid intersection in voxel **1**, as the ray intersects it only in voxel **3**. **Object B** would be a valid intersection in voxel **2**, and would be (correctly) returned as the first object intersected by the ray.

3.7.4 Ray Directional Techniques

Directional techniques explicitly incorporate directional information into the structure of every object. Various operations can then be performed on behalf of many rays at once, instead of just one ray at a time. The cost associated with using directional techniques is that they require large amounts of storage. Three different directional techniques in use are the *light buffer*, the *ray coherence*, and the *ray classification* techniques.

3.7.4.1 The Light Buffer

The *light buffer*, introduced by Haines and Greenberg [HAIN86], is a directional technique that accelerates the calculation of shadows with respect to point light sources.[6] One of the facts utilized by this algorithm is that points can be determined to be in shadow

[6] Point light sources are discussed in *Section 4.6.6*.

without finding the nearest object. Any opaque occluding object will suffice to confirm that a point is in shadow with respect to some light source. The search for an occluding object is narrowed down by making use of the direction of the light source to the point in question.

3.7.4.2 Ray Coherence

Ohta and Maekawa [OHTA87] introduced the *ray coherence theorem* in 1987. It is a mathematical tool for placing a bound on the directions of rays that originate at one object and then strike another. In its simplest form, the ray coherence formula applies to objects that are bounded by non-intersecting spheres.

As shown in *Fig. 3.19*, any ray originating in sphere, *S1*, and terminating in sphere

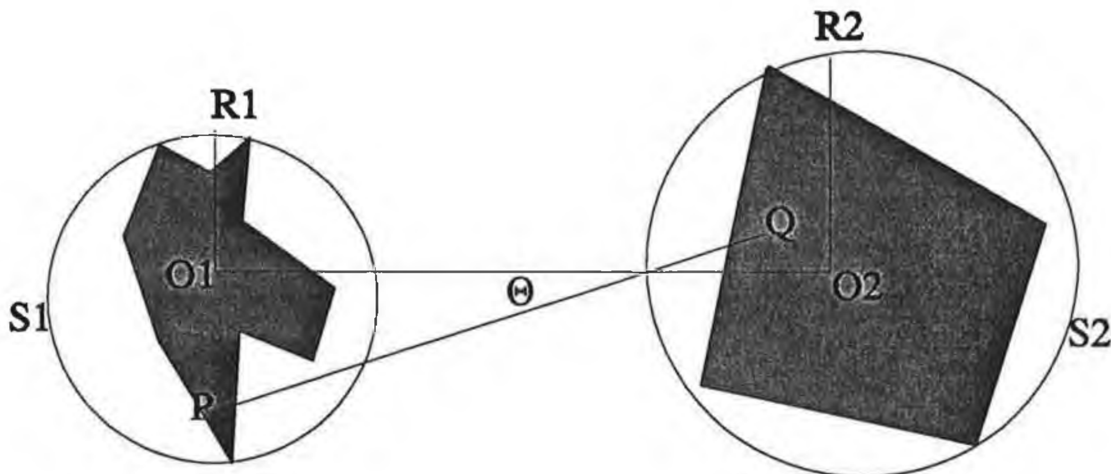


Fig. 3.19 Exploiting *ray coherence*

S2 defines an acute angle θ with the line that goes through the sphere centres. θ can be bound in terms of the sphere radii and the distance between their centres, giving:

$$\cos(\theta) > \sqrt{\left[1 - \frac{r_1 + r_2}{| O_1 - O_2 |} \right]}$$

This inequality can be used to bound the directions of all rays that originate at one object and intersect another.

3.7.4.3 Ray Classification

The *ray classification* algorithm is described by Arvo and Kirk [ARVO87]. Rays in 3D space have five degrees of freedom, and correspond to the points of R^3 X S^2, where S^2 is the unit sphere. We split the 5D space of rays into small neighbourhoods, encoded as 5D hypercubes. A hypercube represents a collection of rays with similar origins and similar directions, and a list of all objects that are hit by any of the included rays. To test for ray/object intersection, we need test only those objects that are contained in the hypercube that contains the ray.

3.7.4.4 Exploiting Coherence

The ability to exploit the coherence of objects in a scene is discussed by Sutherland *et al.* [SUTH74]. Knowledge of coherence can aid in the development of more efficient ray tracers.

Objects tend to consist of pieces that are connected, smooth, and bounded. Distinct objects tend to be largely disjointed in space. This is known as *object coherence*. Nonuniform spatial subdivision works because of this fact.

Image coherence is basicly object coherence transformed down to the 2D world of the monitor. We have the same degree of connectedness, smoothness, boundedness, and disjointness with the final image on the monitor as we do with objects in the 3D scene.

Frame coherence is basicly image coherence with the effects of time taken into consideration. Two successive frames of an animation are likely to be similar when the sampling time difference between them is small.

Rays with almost the same origin and almost the same direction are likely to trace out similar paths in a scene. This is called *ray coherence*. The beam and cone tracing in *Section 2.6.4* and *Section 2.6.5* respectively work only under the assumption of ray coherence. Both Speer *et al.* [SPEE85] and Hanrahan [HANR86] discuss ray coherence.

Rendering in Ray Tracing

4.1 Introduction

Rendering is about generating the colour effects of a scene that is being synthesised using a computer. We are particularly interested in the way rendering is implemented in ray tracing.

Before describing actual rendering models we will discuss colour; what it is and how it comes to be. We discuss models that give a precise definition of what you and I perceive as colour. These are used to develop the *3D RGB cube*, which is the definition of colour as used with computers. We discuss problems that arise because of the limitations of mid-range computer graphics monitors, and describe methods that minimise these problems.

We discuss the various light and surface characteristics that are needed to formulate a rendering equation. We combine these characteristics in a photorealistic rendering model.

4.2 COLOUR

The human eye is able to distinguish between approximately 128 *hues*. Hue is what we colloquially call colour. Hue is the attribute of colour that enables us to classify the colour as being red, purple, etc.. For each hue, around 20 to 30 different *saturations* may be perceived as a different colour. The saturation is a measure of the purity or depth of a colour. The human eye is also capable of distinguishing between 60 to 100 different brightness, or *intensity*, levels. Therefore the human eye can distinguish between approximately 350,000 different shades of colour.

To the human eye, colour develops from the interaction of light rays with the surface of objects. This is easily demonstrated. If you attempt to look at an object in a dark room that has absolutely no light, you will see no colour! When a light ray strikes a surface it somehow acquires colour from this surface. Therefore colour can travel. It travels as light rays. It is travelling colour (light) that strikes our eyes and gives us the perception of colour.

Ray tracing is all about the light rays that strike our eyes. To understand colour we must first understand how it is absorbed by light rays and how these light rays travel.

Light rays exhibit a dual *wave-particle* nature. These can be described by using either a *wave model* or a *particle model.* Under certain circumstances a light ray behaves as a wave, under others, it behaves as a particle. Neither model fully describes the nature of light.

4.2.1 The Wave Model of Light

The basic particle of any light ray is a *photon*. We can think of a photon as a tiny ball flying through space. As the photon flies through space it behaves as if it is *'vibrating'*. In reality it doesn't actually vibrate; it travels in a straight line. However, much of the mathematics that describe vibrating can be applied to describing light rays. With every photon we associate a *wavelength* (or *frequency*). The wavelength and frequency are related by the equation:

$$\lambda = \frac{c}{f}$$

where λ = wavelength

f = frequency

c = the speed of any light ray (in a vacuum, $c = 3.0 \times 10^8 ms^{-1}$)

The individual wavelengths of photons are what give rise to the perceived colours of everyday objects when struck by light rays. When we refer to the wavelengths of visible light we are actually referring to individual monochromatic colours from the visible light spectrum.

Only light rays with a wavelength of between 380nm (nanometres) and 780nm are

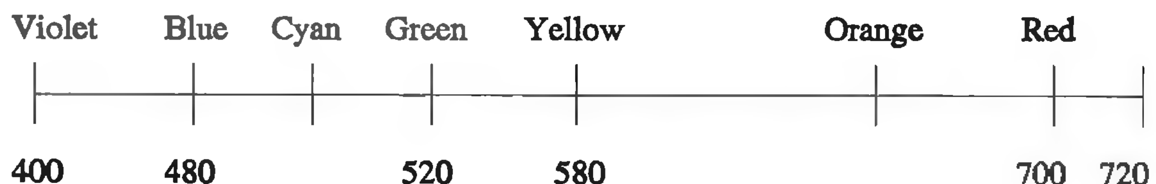


Fig. 4.1 Spectrum of visible light wavelengths.

visible to the human eye. This *visible light spectrum* is shown in *Fig. 4.1*. For example, light rays with a wavelength of 520nm will be coloured green. However, not all colour is directly represented on the spectrum. Only *pure spectral colours* are present. A pure spectral coloured light ray is *monochromatic*, that is, it contains photons of only one wavelength (or colour). Most light rays, however, are not monochromatic (lasers being a notable exception). Light

rays normally contain a *band* (or mixture) of pure spectral colours. All visible non-pure colours can be generated as a band of pure spectral colours with differing intensities.

Using the wave model, a band of photons will travel along a light ray as a single unit. Moravec discusses applications of the wave theory in ray tracing [MORA81].

4.2.2 The Particle Model Of Light

The *particle model* also states that a light ray contains a band of pure colour. However, the difference between this and the wave model is that now the individual photons do not combine into one unit. The individual photons that make up the band travel in parallel, but as separate entities; much in the same way as people on a train are all travelling in the same direction, but they are all still individual people!

When this light ray strikes our eye, the retina is struck not by the light ray as a unit, but by the photons of individual wavelengths that make up the light ray. The individual photons strike the eye in very rapid succession and the retina converts them into one colour.

The particle model is the light model used in ray tracing. When a light ray strikes a surface, the individual wavelengths of the light ray interact separately with the surface.

4.3 COMPUTER REPRESENTATION OF COLOUR

When you and I describe something as being red, are we both perceiving the same colour? If I were to look at the same object through your eyes, perhaps I would perceive it as being pink. Although we both may have different definitions of what each colour is, we each have what we believe to be the correct definition. In order to use colour with a computer, it must be defined explicitly and not left to individuals subjective opinions.

4.3.1 CIE Chromaticity diagram

The *Commission Internationale de L'Eclairage* (CIE) introduced the standard CIE chromaticity diagram in 1931. The CIE diagram gives an exact definition of each visible colour. The CIE diagram defines colours by using an *additive* system. Using the CIE model, any colour may be expressed as an addition of the three *primary colours*. The primary colours are the monochromatic colours red, green and blue. In the CIE diagram the three primary colours may be defined in terms of three *normalised* intensity ratios as:

$$r = \frac{I_{red}}{I_{red} + I_{green} + I_{blue}}$$

$$g = \frac{I_{green}}{I_{red} + I_{green} + I_{blue}}$$

$$b = \frac{I_{blue}}{I_{red} + I_{green} + I_{blue}} = 1 - r - g$$

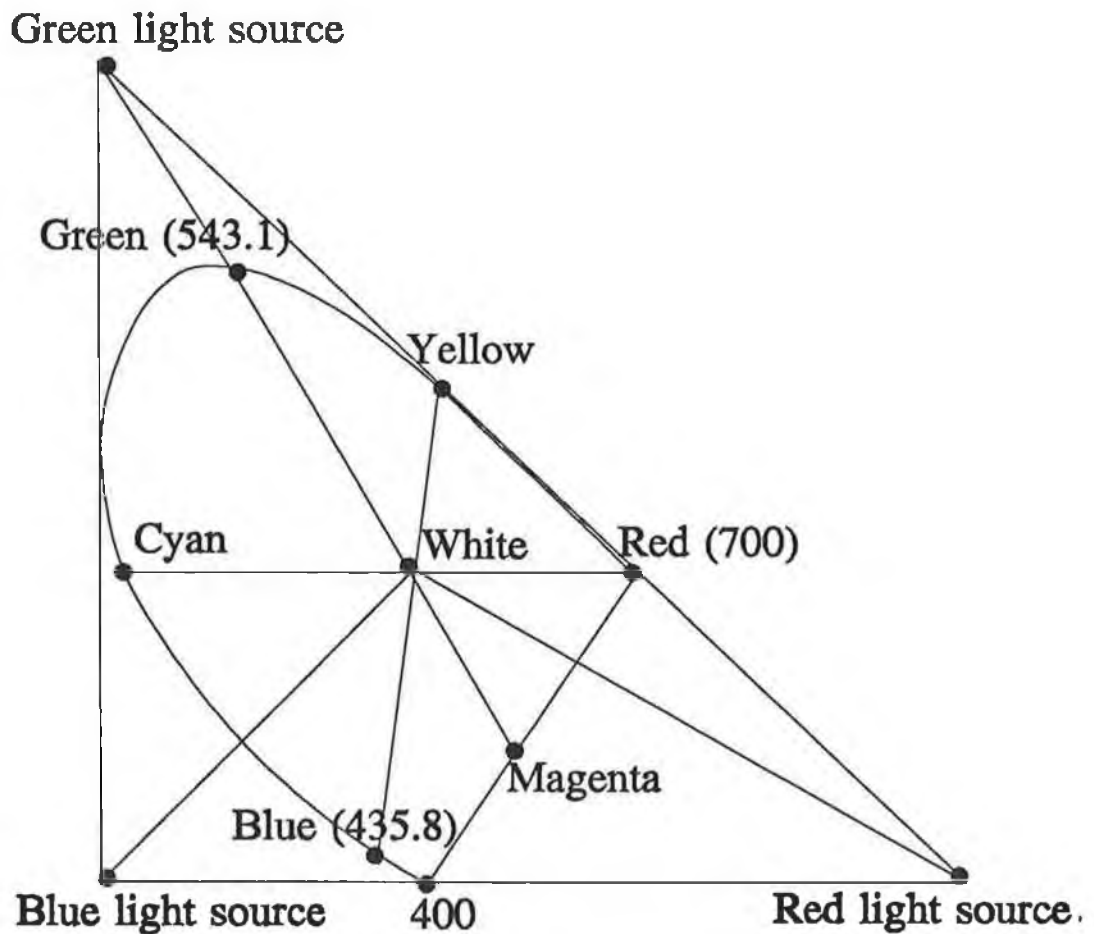


Fig. 4.2 Outline of C.I.E. Chromaticity diagram.

As these three intensities always sum to 1, they form a linear equation; only two of which are independent. This means that the CIE diagram can be represented in 2D. The outline of CIE diagram is shown in ***Fig.** 4.2*. A full colour CIE diagram is shown in [BURG89a]. All the colours in the CIE diagram are enclosed within the triangle defined by the ranges:

$$0 \leq x \leq 1, \qquad 0 \leq y \leq 1, \qquad \text{and } 0 \leq x + y \leq 1$$

The three primary colours are defined on the CIE diagram as

$r = 1$ when: $x = 1$ and $y = 0$

$g = 1$ when: $x = 0$ and $y = 1$

$b = 1$ when: $x = 0$ and $y = 0$

The pure primary colours are defined on the CIE diagram at the following wavelengths:

Red = 700nm

Green = 543.1nm

Blue = 435.8nm

4.3.2 Colour Monitors

Computer generation of colour is based upon *colour mixing*. By combining red, green and blue colouring with differing intensities, we can generate any other visible colour.

When the three primary colours of a colour monitor are mapped onto the CIE diagram, the group of colours that can be displayed on the monitor is shown in ***Fig.** 4.3*. This group is a subset of the CIE set of visible colours. We use *R*, *G*, and *B* to represent the colours red, green, and blue on the monitor. These are not pure red, green and blue as shown on the CIE diagram. Their positions on the CIE diagram in terms of *r*, *g*, and *b* would be as follows:

	r	g	b
R	0.628	0.346	0.026
G	0.268	0.588	0.144
B	0.150	0.070	0.780

We can produce a new colour reference diagram for the colours that are realisable on a computer monitor by mapping the realisable colours from the CIE diagram on to a

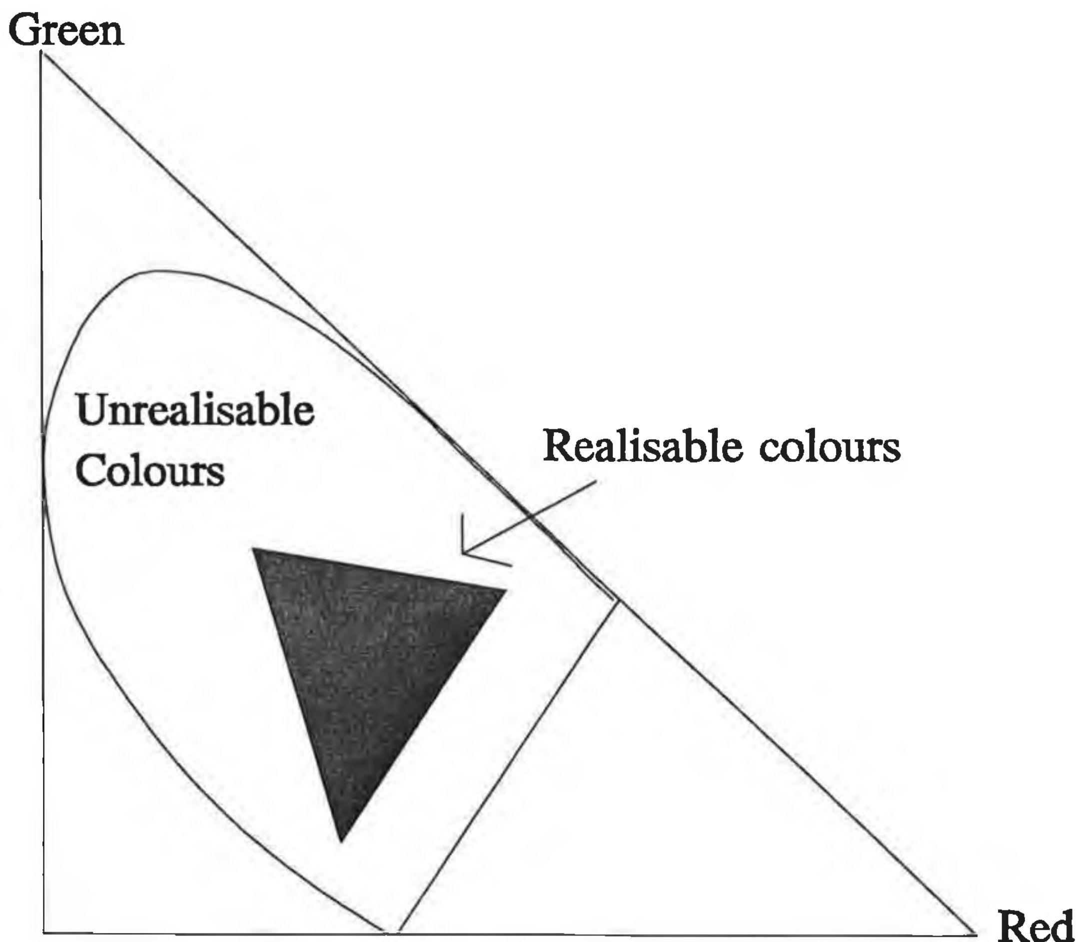


Fig. 4.3 Comparison between CIE diagram colours and colours obtainable on a computer monitor.

monitor red-green-blue diagram. In the monitor red-green-blue diagram, red is normalised along the x-axis and blue along the y-axis. The necessary transformation matrix is as follows:

$$[r,g,b]_{\mathrm{CIE}} = [R,G,B]_{\mathrm{monitor}} \begin{bmatrix} 0.628 & 0.346 & 0.026 \\ 0.268 & 0.588 & 0.144 \\ 0.150 & 0.070 & 0.780 \end{bmatrix}$$

The reverse transformation matrix is found by inverting the matrix.

The problem with both the CIE and the red-green-blue models is that they represent absolute colours. A colour's intensity is not readily distinguishable from its hue.

4.3.3 The RGB Model

In computers, colours are generated as a function of the respective intensities of the

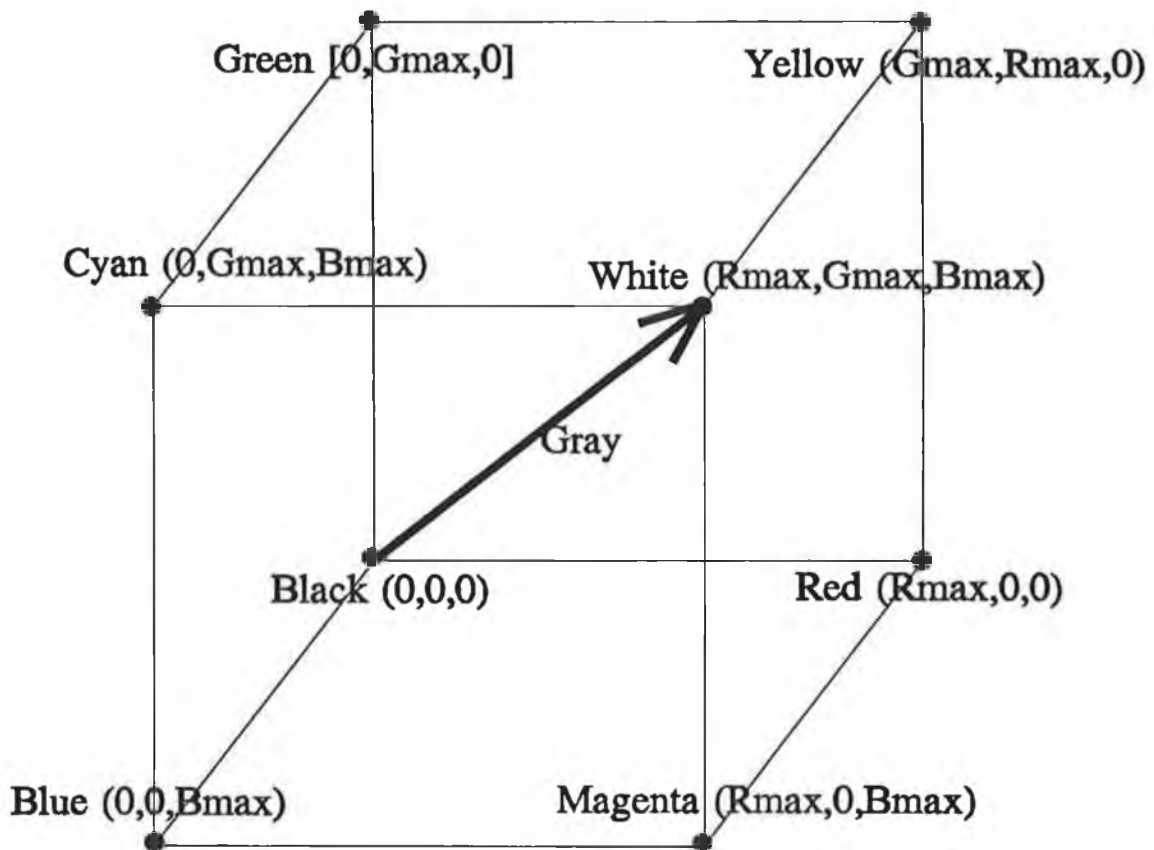


Fig. 4.4 RGB cube.

three primary colours. A way to represent a complete set of colours based on intensities, is to use the 3D *RGB cube*, as shown in *Fig. 4.4*. A full colour RGB cube is found in [BURG89b]. The intensity of each of the three primary colours is normalised. Using the RGB cube, we define any realisable colour in terms of the intensities of red, green and blue.

4.3.4 Colour Palettes and Lookup Tables

The complete range of colours that can be displayed on a colour monitor are known as the *palette* for that monitor. For medium and high range monitors, the intensities of red, green and blue are each stored in one byte of memory, giving a palette of 256 X 256 X 256 (or 16 million) possible colours that can be displayed on the monitor. This represents all the colours in the RGB cube. Many of the 16 million colours are indistinguishable to the human eye, as it can only differentiate between 350,000 colours.

The set of colours from the palette that can be simultaneously displayed on a monitor are kept in a *colour lookup table*. For top of the range graphics monitors, the lookup table contains the entire palette. For medium range graphics monitors, the lookup table is indexed by one byte. It can only hold 256 colours from the palette at any time.

4.4 COLOUR QUANTIZATION

Depending on the graphics requirements, a choice must be made as to which colours from the palette will be placed in the lookup table.

The 256 colours in the lookup table are indexed by an 8-bit byte. The way in which we manipulate these bits will determine what colours are contained in the lookup table. We can, for example, assign 3 bits to each of red and green, and assign 2 bits to blue. This will allow us to have 8 X 8 X 4 intensities of red, green and blue respectively. The choice as to which two colours are given 3 bits and which one colour is given 2 bits is a matter of choice, and depends upon the image being generated.

A good starting place for deciding which colours to place in the lookup table is to *uniformly* step through the palette and pick out 256 uniformly distributed colours. This is a simple algorithm to implement. However, for photorealistic images a maximum of 8 intensities of any one colour is insufficient.

A second method of indexing divides the 8 bits into two fields. By using the two most significant bits to hold specific colours, we have 6 bits (or 64 levels of intensity) for each colour. Alternatively, we could use 3 bits to define colours and 5 bits for intensity. This gives us a choice of 8 colours, each with 32 levels of intensity. There will always be a trade-off between the number of colours we can display, and the number of intensities that we can assign to each. *Colour plate 3* is a good example of the colour/intensity trade off. Its lookup table is described in *Section 5.9*.

4.4.1 Getting over the 256 colour limit

The 256 colour lookup table limit is a feature of the graphics hardware. It is is caused only because the image is being displayed on a monitor. If the image is written to an *image file*[1], instead of being output to a monitor, then every pixel can be kept in its original red, green, blue format. Of course, we may still wish to display the image on a computer monitor, at which time we must map the image down to 256 colours. So, have we actually gained anything by writing the image to an image file? Yes, we have. We can now use the colour information in the image file to help us create a lookup table that is most representative of the colours in the image file. Some methods of processing an image file are now discussed.

4.4.2 Popularity Algorithm

One method that does not require predefining the colour lookup table is the *popularity algorithm*. The rendered image, with all its red, green, blue pixel formats, is saved to an image file rather than been output to a monitor. This image is processed to find the 256 most popular pixel colours. These colours are then used as the entries in the lookup table, and all the colours in the image file are mapped onto their closest representative in the table.

The popularity algorithm is hungry for memory. For an image of 1024 X 768 we need 1024 X 768 X 3 *ints*[2], which is approximately 4.7 Megabytes of memory!

A second problem with the popularity algorithm arises after the entries in the lookup table have been filled. This problem concerns the mapping of the pixels in the image file to their nearest entry in the lookup table. The distance between any two colours could be equated with the 3D distance between points in the RGB cube. However, it is computationally cheaper to simply minimise:

$$|R_1 - R_2| + |G_1 - G_2| + |B_1 - B_2|$$

where: $[R_1, G_1, B_1]$ are intensities for an image pixel

$[R_2, G_2, B_2]$ are intensities in the lookup table

---

[1] An *image file* is just a normal binary file. The usage of the word *image* is for clarity only.

[2] We need one *int* data type to represent each of the three intensities red, green, and blue. Each int occupies two bytes of memory.

The determination of the minimum distance can be reached following an exhaustive search. Alternatively, a zoom in approach can be used to cut down on the number of lookup table entries to be processed. For each pixel in the image this approach works as follows:

Sort the lookup table entries by the red intensity. The lookup table entry with the closest intensity agreement in red to that of the pixel is used as a first approximation to the solution. The distance between this lookup table entry and the pixel is then found and used to eliminate from further processing those table entries whose distance along the red axis is greater than this minimum distance. The reduced table is next sorted by blue intensity. The lookup table entry whose blue intensity is closest to that of the pixel is found. This is compared with the best red. If the blue value is smaller, it replaces the red as the minimum distance. Again the table is reduced by removing from further processing any entries where the distance along the blue axis is greater than the minimum distance. The process is repeated for green, leaving us with only those entries in the lookup table that passed the three minimising tests. To find which of the remaining lookup table entries is the closest to the pixel we conduct an exhaustive search.

The popularity algorithm is very computationally expensive. The closest fit search has to be carried out for every pixel in the image file. Another drawback to the algorithm is that it can miss small, but important areas of colour. For example, a specular highlight of white light in an image without other white pixels will be incorrectly coloured.

4.4.3 Median-cut Algorithm

Another algorithm for assigning values to the lookup table is the *median-cut algorithm*. The median-cut algorithm divides the RGB cube to ensure that each lookup table entry represents the same number of pixels. The division is done by planes parallel to the three axes. The initial division is made along the red axis. A histogram of red values is computed and used to determine the position of the subdividing plane. The image pixels are then split into two sets, one for each of the two subdivided volumes. A count of the number of pixels being placed into each subdivided volume is kept. The algorithm proceeds recursively, with the division of each volume being along its longest side. When 256 volumes have been generated, a table entry is assigned to each volume by means of computing the average pixel value in the volume. Mapping the pixels to the lookup table entries is achieved by comparing each pixel's red, green, and blue intensities with the bounding planes of the 256 volumes.

4.4.4 Octree quantization algorithm

Another algorithm that is similar in principle to the median-cut algorithm is the *octree quantization algorithm*. This method reduces both the computational time and the memory requirements of the median-cut algorithm. The RGB cube is recursively divided into 8 equally sized cubes. Each of these cubes contains some, possibly none, of the image file pixels. The recursive subdivision continues down until every cube at the bottom level contains exactly one or no pixels from the RGB cube. This recursive subdivision of the RGB cube can be represented as an *octree*, with the RGB cube at the root and the cubes containing one or no pixels as the leaf nodes. After removing leaf nodes that contain no pixel, the number of leaf nodes is equal to the number of different pixels in the image. The tree is reduced by an averaging process so as to find the 256 entries to place in the lookup table. The averaging involves taking the nodes one level above the leaf nodes and giving as their colour the average colour of the leaf nodes below them. The tree is then pruned, making into leaf nodes those nodes that were previously one level up from the now pruned leaf nodes. The process is repeated up the levels of the tree until there are exactly 256 leaf nodes left. These colour values are then used as the entries in the lookup table.

In practice we do not build the whole octree, as this would require considerable memory. Instead, the building and the reduction take place in one pass of the image file. The first 256 pixels in the image file are used to form an initial octree. If the tree has 256 leaf nodes, then a reduction is made before adding the next pixel. Which leaf to prune is decided either by using the deepest leaf or the one containing the fewest pixels. Both methods give a slightly differing final image. After a reduction, one or more new pixels are either added as a leaf node, or as part of a leaf's average. New pixels will be added to the tree, either as a leaf node or by inclusion in an average of a reduced node. This process of pruning and adding will be performed until all the pixels in the image file have been processed. In the end there will be exactly 256 leaf nodes. These are used as the entries in the lookup table.

Using either of the two methods for pruning that were stated above will lead to an image of less quality than that of an image produced using the full octree quantization or median-cut algorithms. However, there is very little visual difference in the final image.

4.5 COMPUTER COLOUR AS RAYS

In a computer each colour is represented as a red, green and blue intensity. In order to use light rays to transport colour in a computer model, we must be able to send the combined red, green, blue intensity in the form of a ray.

A first attempt may be to group the information from the three intensities as one piece of information, and to send this along the ray (i.e. use the wave model of light). However, by using this model to describe light rays we cannot implement transmission. When a light ray passes between two media, it is refracted by an amount dependent on its wavelength. This process is described in *Section 4.6.3*. If we are using a single value to represent the three wavelengths of red, green, and blue simultaneously, then there is no single direction that is going to give us correct refraction.

4.5.1 Three rays in one

A simple solution is to send three rays instead of just one (i.e. use the particle model of light). This, in fact, is what is done in ray tracing. For each of the eye rays described in *Section 2.3*, a real ray tracer casts three rays, one for each of the red, green and blue intensities. The red, green, and blue rays are processed *independently* of each other.

4.6 LIGHT TRANSPORTATION MODES

When a light ray strikes a point on a surface, the light ray will undergo changes in colour and direction. The interaction of the light ray and the point will cause the light ray to be *reflected* away from, and *transmitted* into, the surface. The reflection and transmission is broken down into four classes: *specular reflection*, *diffuse reflection*, *specular transmission*, and *diffuse transmission*. The amount of influence of each effect is dependent upon the surface material and the wavelength of the light ray. The wavelength of any ray in a computer is either red, green, or blue.

4.6.1 Surface Normals

In order to discuss the geometry of reflection and transmission of light rays we need to introduce the concept of *surface normals*. A surface normal for any given point on a surface is the vector that indicates the direction perpendicular to the surface at the given point.

The surface normal always points *away* from the surface. For a plane, the surface normal is the same at every point. For a sphere the surface normal for any point is found by following the radius line through that point. In *PRIME*[3], the surface normal for each surface is stored and is returned, along with the intersection point whenever a ray intersects an object.

4.6.2 Specular Reflection

When an incident light ray strikes a hard, flat, shiny surface, it is *specularly reflected*

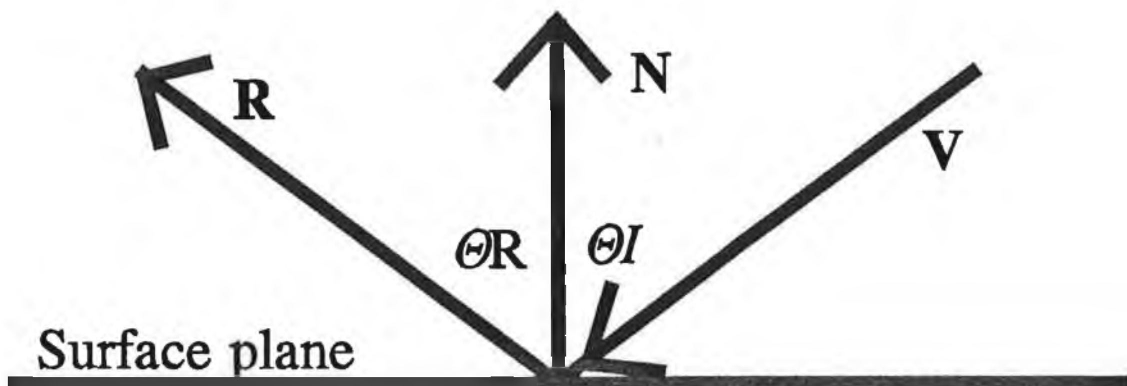


Fig. 4.5 Reflection of light.

away from the surface. Specular reflection is shown in *Fig. 4.5*. The angle between the surface normal (**N**) and the incident light ray (**V**)[4] is called the *angle of incidence*. We denote it as θ_I. The angle between the surface normal and the reflected light ray (**R**) is called the *angle of reflection*, denoted θ_R[5]. We observe two points of detail. Firstly, as **V**, **N** and **R** all lie on the same plane, we express **R** in terms of a linear expression of **V** and **N**. Secondly, the angle of incidence is equal to the angle of reflection (i.e. $\theta_I = \theta_R$).

[3] *PRIME* is *P*hoto*R*ealistic *I*mage *M*odelling *E*nvironment. It is the topic of discussion in *Chapter 5*.

[4] The incident vector is called **V**, so as not to confuse it with light intensity *I*. *I* will be introduced later in this chapter.

[5] θ_I is shown as **ΘI** and θ_R is shown as **ΘR** in *Fig. 4.5* and subsequent figures. This is caused by restrictions in the figure generation software.

We know **N** and **V**, and we wish to find **R**. We use *Heckbert's Method* to derive **R**.

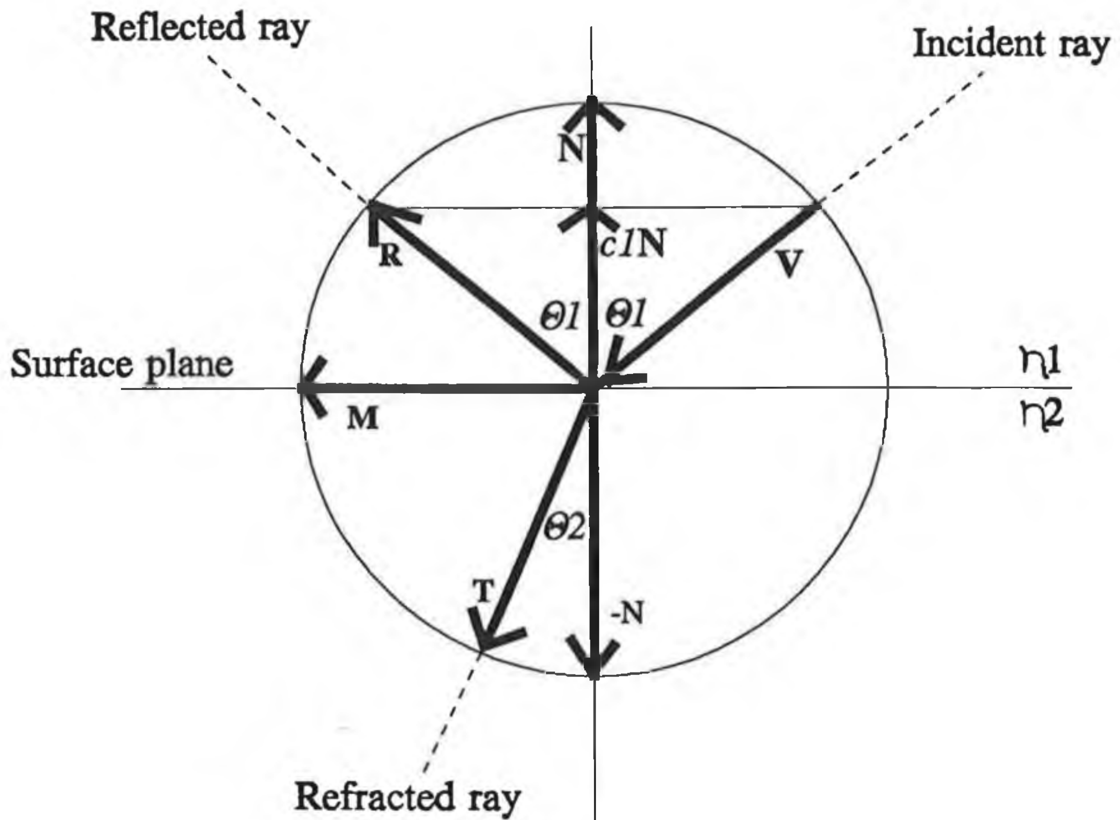


Fig. 4.6 Geometry needed to derive the direction of reflected and transmitted light using *Heckbert's method.*

Using this formula we assume that all vectors are normalised. The geometry of Heckbert's method is shown in *Fig. 4.6.*

$$c_1 = \cos \theta_1 = -\mathbf{V} \cdot \mathbf{N}$$

R is calculated simply by constructing the parallelogram of *Fig. 4.7.*

$$\mathbf{R} = \mathbf{V} + 2c_1\mathbf{N}$$

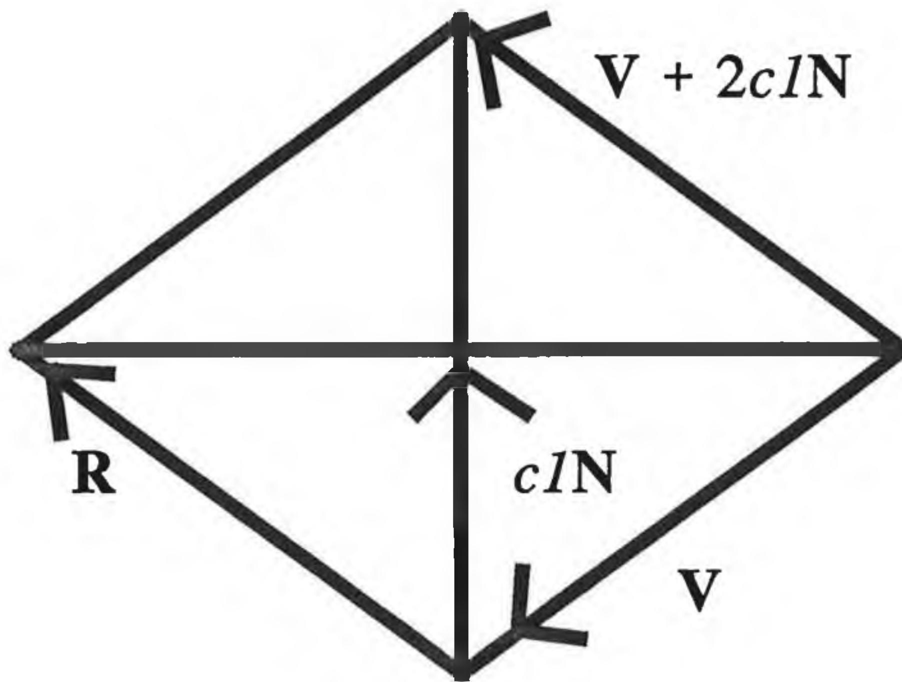


Fig. 4.7 Parallelogram showing composition of reflected light vector.

4.6.3 Specular Transmission

A *specularly transmitted* light ray has the same characteristics as a specularly reflected light ray, except that it is directed *into* the surface. The angle at which the transmitted ray is cast depends upon the density of the medium through which the incident ray travelled and the density of the medium through which the transmitted ray will travel. As a light ray travels from one medium into a more dense medium, it is bent toward the normal. As a light ray travels from a more dense to a less dense medium, it bends away from the normal. Transmission of light rays is shown in *Fig. 4.8*. The angle of incidence and the angle of transmission are related by *Snell's Law*.

Snell's Law states:

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \eta_{21} = \frac{\eta_2}{\eta_1}$$

Where: η_1 is the index of refraction of medium *1* with respect to a vacuum

η_2 is the index of refraction of medium *2* with respect to a vacuum

η_{21} is the index of refraction of medium *2* with respect medium *1*

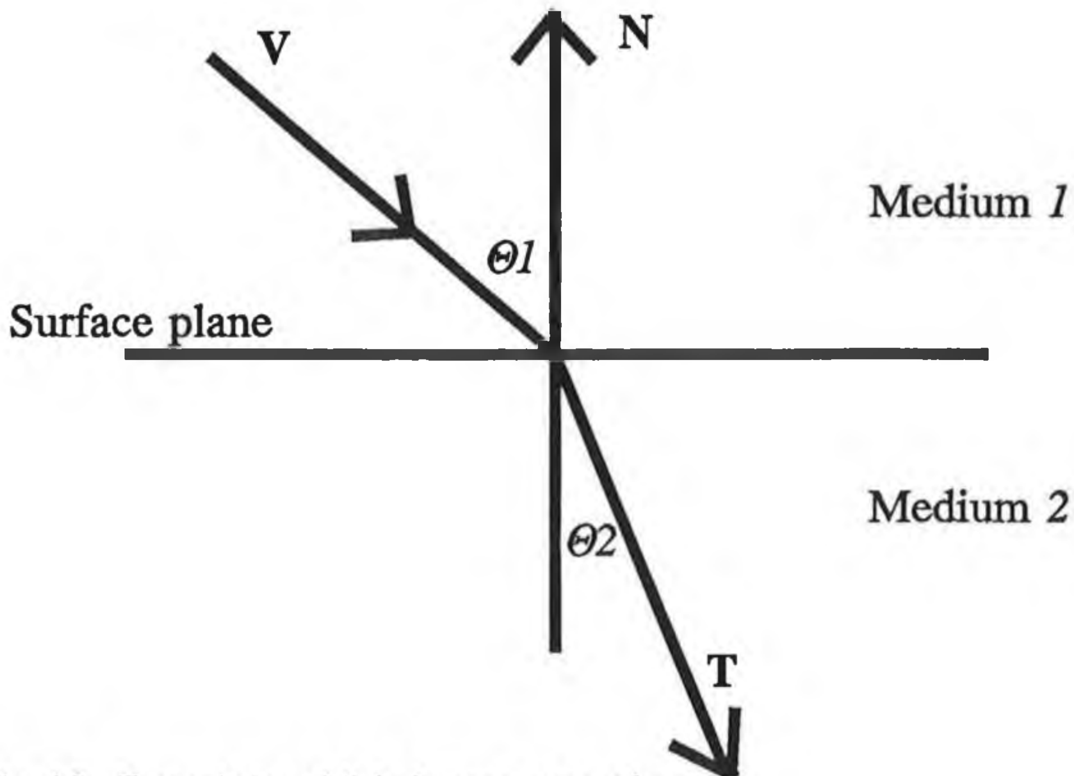


Fig. 4.8 Transmission of light between two surfaces.

The index of refraction is dependent on the wavelength of the incident light ray.

Total internal reflection, shown in *Fig. 4.9*, may occur when a light ray tries to pass from one medium to a less dense medium. If the incident light ray strikes the surface between the two media at any angle greater than the *critical angle* for these two media, then the light ray is reflected back into the more dense medium instead of being transmitted out to the less dense one. The critical angle is reached when the angle of refraction is 90^0.

Heckbert's method is also used to find the direction for any specularly transmitted light ray. If **M** is defined as a unit surface tangent vector in the plane of **V** and **N** ()see *Fig. 4.6*), then the transmitted ray (**T**) is expressed as:

$$\mathbf{T} = \mathbf{M}\sin(\theta_2) - \mathbf{N}\cos(\theta_2)$$

$$\mathbf{M} = \frac{\mathbf{V}_{perp}}{|\mathbf{V}_{perp}|} = \frac{\mathbf{V} + c_1\mathbf{N}}{\sin(\theta_1)}$$

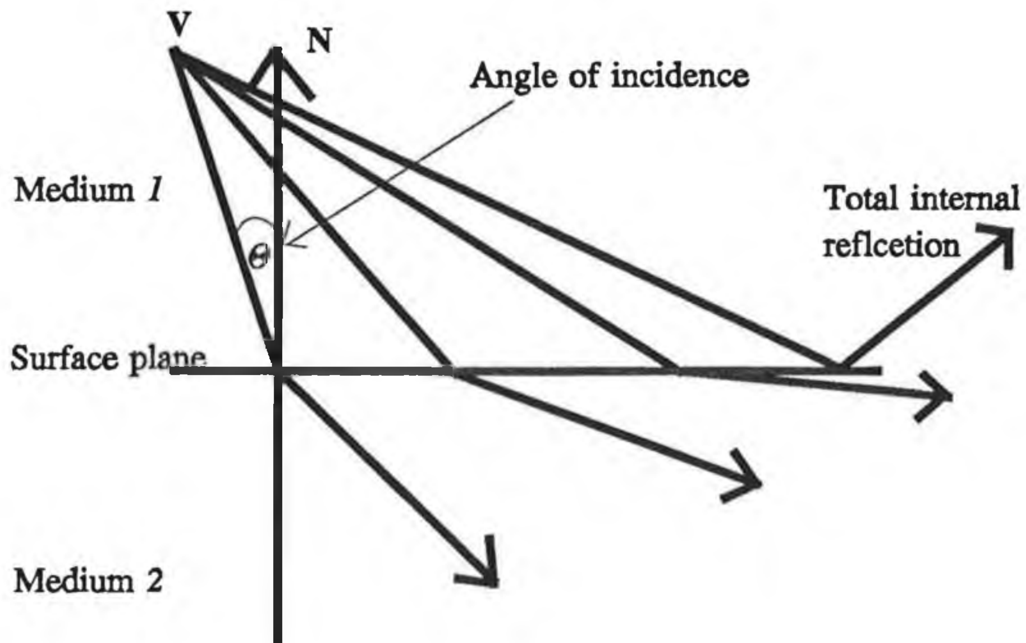


Fig. 4.9 Total internal reflection.

Therefore:

$$\mathbf{T} = \frac{\sin(\theta_2)}{\sin(\theta_1)}(\mathbf{V} + c_1\mathbf{N}) - \cos(\theta_2)\mathbf{N}$$

But by Snell's law, the relative index of refraction η is:

$$\eta = \frac{\sin(\theta_2)}{\sin(\theta_1)} = \frac{\eta_1}{\eta_2} = \frac{1}{\eta}$$

so:

$$\mathbf{T} = \eta\mathbf{V} + (\eta c_1 - c_2)\mathbf{N}$$

where: c_2
$$= \cos(\theta_2)$$
$$= \sqrt{(1 - \sin^2(\theta_2))}$$
$$= \sqrt{(1 - \eta^2\sin^2(\theta_1))}$$
$$= \sqrt{(1 - \eta^2(1 - c_1^2))}$$

The transmission of light rays explain strange visual effects such as why a stick placed in a glass of water appears to bend. As the water is more dense than the air, the light rays travelling to our eye from the stick are bent en route, thus giving the impression that the stick is bent.

4.6.4 Diffuse Reflection

Diffuse reflection is caused when an incident light ray is absorbed upon striking a surface. Light rays are re-radiated away from the surface point as diffusely reflected light rays. The colouring of the diffusely reflected light rays is dependent upon the angle at which the incident light ray strikes the surface and upon the surface characteristics. Furthermore, the re-radiated light rays will travel out in all directions with equal intensity, as shown in *Fig. 4.10a.* The amplitude of the re-radiated light rays is proportional to the angle at which the incident light ray strikes the surface. A greater angle of incidence will lead to a lesser amplitude of the re-radiated light rays. This effect is shown in *Fig. 4.10b.*

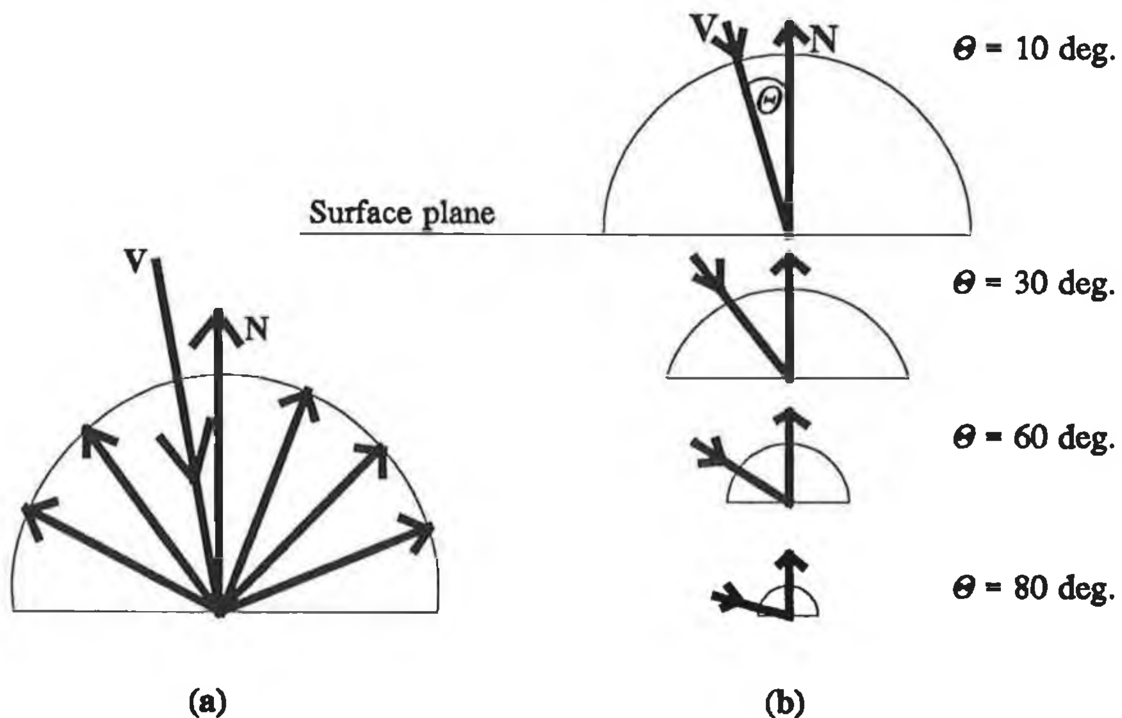


Fig. 4.10 Diffuse reflection.

4.6.5 Diffuse Transmission

The characteristics of *diffusely transmitted* light rays are similar to those of diffusely reflected light rays, except that diffusely transmitted light rays are emitted on the opposite side of a surface to the incident light. Diffuse transmission is shown in *Fig. 4.11.*

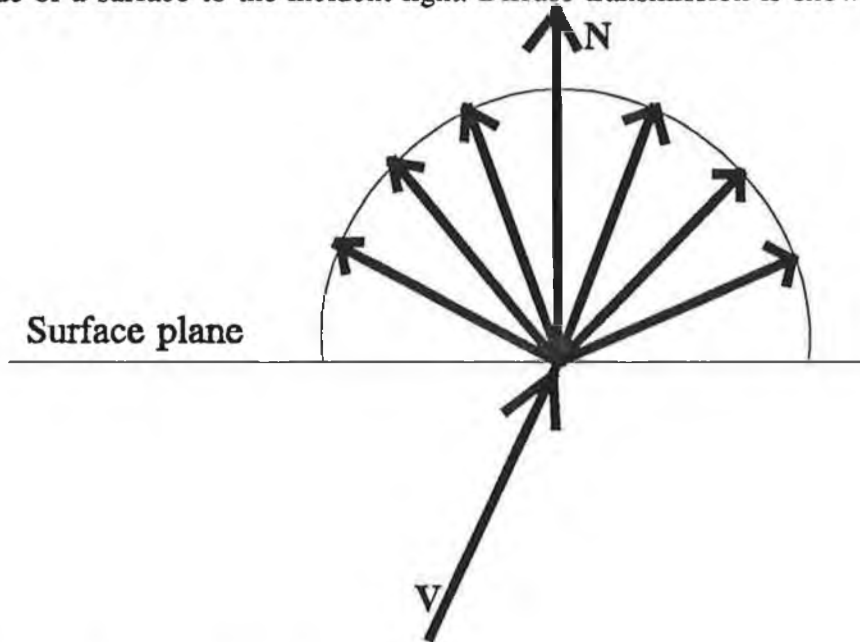


Fig. 4.11 Diffuse Transmission.

4.6.6 Light Sources

A scene will contain one or more light sources of varying colours. These lights can be either *point* or *distributed* sources. A point source is similar to a normal light bulb. We can assume that all of its light rays come from one point. For such a light source we will need to know its origin and its colour. A distributed light source is like a fluorescent light. We model such a light source by using a number of point light sources.

Point light sources will be referenced as an array *0..numLights*. Light rays from each of the light sources may strike a surface point that is being ray traced. We represent individual light sources as L_j. The intensity of individual light sources will be denoted as I_{lj}.

In total, we have four sources of light rays that can influence the colouring of any point being ray traced. These are I_{lj}, I_a, I_{sr}, and I_{st} where:

I_{lj} is the intensity of point light source *j*, where $0 <= j <= numLights$.

I_a is the *ambient light*. This term is used to approximate the effects of diffusely reflected light rays from the various objects in a scene.

I_{sr} is the intensity of the reflected light ray. This tells us how much light is arriving at the surface point along the reflected ray.

I_{st} is the intensity of the transmitted light ray.

4.7 SURFACE CHARACTERISTICS

If you stand back at an angle to the left and look in a mirror you will see objects, such as a chair, that are at an equal angle but to the right of the mirror. You can see the chair because light rays coming from the chair are *specularly reflected* by the mirror into your eyes. A light ray coming from the chair barely interacts with the mirror. The ray strikes the mirror and is reflected away. It retains the colour it had when it struck the mirror; that of the chair. If, instead of a mirror, we were to look at a shiny piece of metal, such as copper, we would still see the chair. However, now the image would contain some copper colouring. The light ray has absorbed some of the copper's colour. If we place a non-shiny surface, such as a piece of paper in the place of the mirror, we see only the colour of the paper. The light ray travelling from the chair was almost fully absorbed by the paper. The colour of the reflected light ray, therefore, is dependent on the surface characteristics. Similarly, all incident light rays interact with a surface before they are reflected and transmitted.

4.7.1 Surface Texture Maps

In order to render an image we need to be able to find the surface colour at any point that is struck by a ray. If the surface is only one colour, then this is used as the colour for any point on the surface. However, in the real world most surfaces are multi-coloured. A globe will have at least green land and blue sea. Wood will have various coloured grains running along it. All of the surfaces in *Colour Plate 2* are examples of multicoloured surfaces. In order to represent any multi-coloured surface we use *texture maps*. A texture map is a two

dimensional representation of the surface colouring. If we 'roll' the surface out until it becomes flat, we get its texture map. Any surface can be mapped onto a texture map using *inverse mapping*. Inverse mapping translates a 3D surface point coordinate onto the 2D texture map. Each surface type that is modelled will need its own inverse mapping routine. The exact mathematics of the texture maps that are used in *PRIME* are described in *Section 5.5*.

Heckbert [HECK86] compares various texture mapping methods. Some specialised texture mapping algorithms are given by Miyata [MIYA90], where he describes a method of generating stone wall patterns, by Smith [SMIT87], who describes a method for mapping warped surfaces, by Maeder [MAED89] who describes a method for describing granular surfaces, and by Peachet [PEAC85] who describes a *3D texture volume* that can be used to map the texture of non-homogeneous materials, such as wood and stone.

4.7.2 Surface Roughness Maps

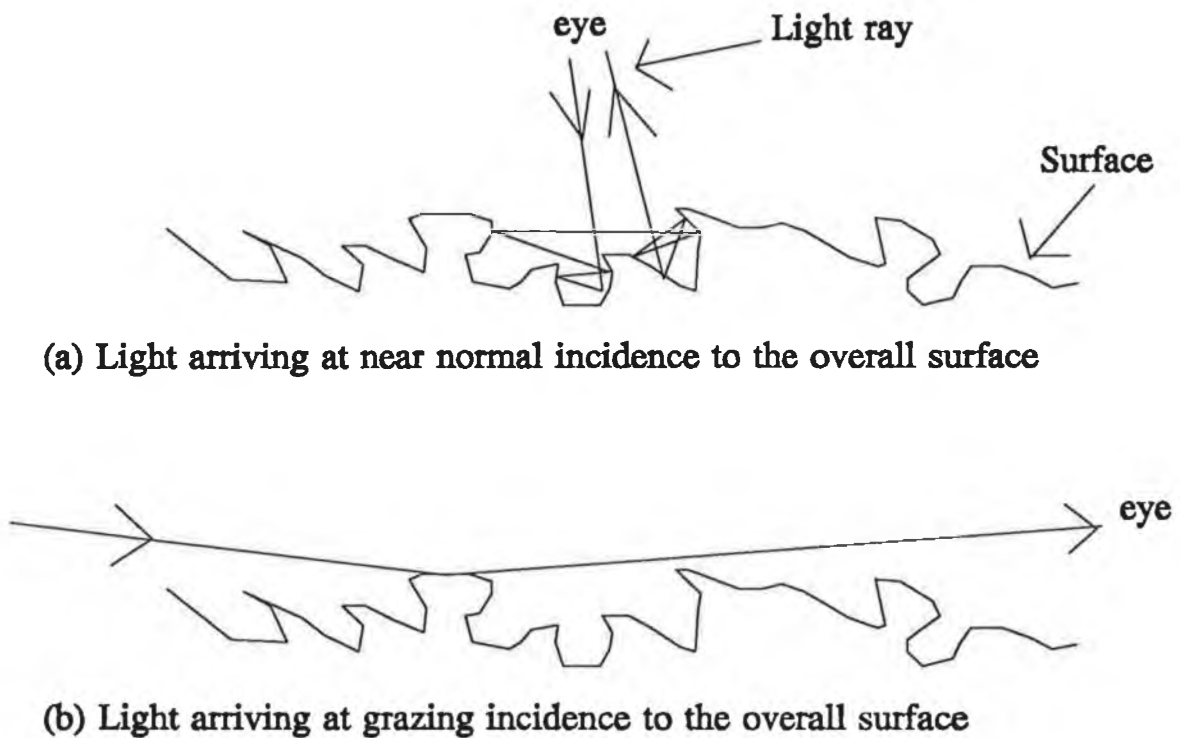
A surface *roughness map* is stored and accessed in the same manner as a texture map. The roughness of a surface will have an effect on the intensity of the specularly reflected and specularly transmitted rays.

We can think of a rough surface as being composed of many very tiny flat surfaces, called *microfacets*. Because each microfacet is flat, light rays will be specularly reflected and specularly transmitted by individual microfacets.

Fig. 4.12a shows a light ray arriving at a rough surface. The incoming light ray is almost normal to the overall surface. The light ray is specularly reflected from one microfacet to another again and again. Each time the light ray strikes a microfacet, it absorbs some colouring from the surface. By the time the light ray finally reaches the eye it has been strongly coloured by the surface. The amount of colouring absorbed by the light ray will depend on the distribution of the microfacets on the surface.

In *Fig. 4.12b* the light ray strikes the surface at an angle almost parallel to the overall surface (perpendicular to the surface normal). This light ray only grazes the surface and has little interaction with the microfacets. When it reaches the eye, its colouring will not have been significantly influenced by the surface.

In general, the colour and intensity of the specularly reflected light ray leaving a given point on a surface will be dependent on the direction of the incoming light ray, the colour of



(a) Light arriving at near normal incidence to the overall surface

(b) Light arriving at grazing incidence to the overall surface

Fig. 4.12 Greatly magnified image of the interaction of a light ray with a rough surface.

the object, and the distribution of the microfacets on the surface.

The microfacet model for specular reflection off rough surfaces is a theoretical model. It is described by Torrance and Rogers [TORR67] and by Cook and Torrance [COOK81]. It is adapted to computer graphics by Blinn in [BLIN77a].

In order to use microfacets in a rendering equation we need two vectors, $\mathbf{H}_j$ and $\mathbf{H}_{j'}$. $\mathbf{H}_j$ is the normal vector for microfacets that specularly reflect the incoming light source ($\mathbf{L}_j$) along $\mathbf{V}$ (the incident ray). $\mathbf{H}_j$ lies exactly in the middle between $\mathbf{L}_j$ and $\mathbf{V}$. Therefore:

$$\mathbf{H}_j = \frac{\mathbf{L}_j + \mathbf{V}_j}{|\mathbf{L}_j + \mathbf{V}_j|}$$

The vector $\mathbf{H}_{j'}$ serves the same purpose for specular transmission.

$$\mathbf{H}_{j'} = \frac{\mathbf{V}_j - \beta \mathbf{L}_j}{\beta - 1} \qquad \text{where } \beta = \frac{\eta_2}{\eta_1}$$

4.7.3 The Fresnel Function, *F*

From the previous section we know that the colour absorbed by a light ray upon striking a surface is dependent upon the angle of incident and wavelength of the ray, and upon the surface characteristics of the point that is hit by the ray. The *Fresnel function* is used in the rendering model to take these factors into account. The Fresnel function is used to describe the interaction between light from each light source and the surface. The equation for F of a given wavelength, given the angle of incidence θ is:

$$F(g,c) = \frac{1}{2}\frac{(g - c)^2}{(g + c)^2}\left[1 + \frac{[c(g + c) - 1]^2}{[c(g - g) - 1]^2}\right]$$

Where: $c = \cos(\theta) \quad = \mathbf{V}\cdot\mathbf{H}_j$

$g^2 = \eta^2 + c^2 - 1$

η = index of refraction at a given wavelength

Only $F(0)$, the value at normal incidence, is directly available. Several reference works such as [PURD70a],[PURD70b] and [PURD70c] contain listings of $F(0)$ for different *material/wavelength* combinations. $F(\theta)$ is not generally listed for all angles of incidence, and so must be derived.

At normal incidence, $\theta = 0$

=> $c = \cos(\theta) = 1$

=> $c^2 = 1.$

We can now solve for g

$$g^2 = \eta^2 + c^2 - 1$$

$$= \eta^2 + 1 - 1$$

Therefore $g = \eta$ *see footnote 6*

---

6 Mathematically $g = \pm\eta$. However, η, the index of refraction, can only be a positive value, so $g = \eta$.

Plugging these values back into the Fresnel function gives:

$$F(0)= \left[\frac{\eta - 1}{\eta + 1} \right]^2$$

We really want η, so we take the square root of both sides and solve:

$$\eta = \frac{1 + \sqrt{(F(0))}}{1 - \sqrt{(F(0))}}$$

Now that we know η, we can easily solve the Fresnel equation at any other angle of incidence.

For every material being rendered we need to build a table of Fresnel values, indexed by the various wavelengths (red, green, and blue) and possible angles of incidence (0^0 .. 90^0) of a light source. Fresnel lookup tables must be generated for both specular reflection and specular transmission.

The Fresnel function for specular reflection is denoted $F_{sr}(\theta)$, and for specular transmission it is $F_{st}(\theta)$. Both $F_{sr}(\theta)$ and $F_{st}(\theta)$ are defined in the range:

$$0 \leq F_{sr}(\theta), F_{st}(\theta) \leq 1$$

where higher values mean a greater colouring of the incident light by the surface.

Because of the principles regarding the conservation of energy:

$$F_{st}(\theta) = 1 - F_{sr}(\theta)$$

Two further terms F_{dr} and F_{dt} also need to be assigned for every *material/ray wavelength* combination. F_{dr} and F_{dt} are not affected by the angle of incidence of the ray.

Again:
$$0 \leq F_{dr}, F_{dt} \leq 1$$
and
$$F_{dt} = 1 - F_{dr}$$

4.7.4 Reflectance Coefficients

k_{dr} is the *diffuse reflectance coefficient.* It is a measure of how much of the reflected light ray is radiated as diffusely reflected light rays. A shiny mirror would have a diffuse reflectance of 0, while a piece of matt cardboard would probably have a diffuse reflectance of higher than 0.9. k_{sr} is the *specular reflectance coefficient.* k_{dr} and k_{sr} are related by:

$$k_{dr} + k_{sr} = 1$$
and
$$0 \leq k_{dr}, k_{sr} \leq 1$$

Associated with k_{sr} is another surface characteristic, n. n is the *specular reflection highlight coefficient.* It is used to exert control of the highlights on a surface. Very shiny surfaces will have a large value for n, generating sharp highlights on the surface. n must be ≥ 0. If n is 1, we get very spread out highlights. As n rises to 10, 20 or even higher, the highlights become sharper. If n is about 100 or larger we get mirror-like surfaces. The parameter n was first developed by Phong [PHON75].

4.7.5 Transmission Coefficients

k_{dt} is the *diffuse transmissive coefficient*, k_{st} is the *specular transmissive coefficient*, and n' is the *specular transmission highlight coefficient.* k_{dt} and k_{st} are related by:

$$k_{dt} + k_{st} = 1$$
and
$$0 \leq k_{dt}, k_{st} \leq 1$$

4.7.6 Transmissitivity

T_r is the transmissitivity per unit length of the medium containing the reflected ray. A light ray travels further through the least dense of two mediums. For example, a light ray travels further through a vacuum than through air, and further through air than through water. It cannot travel at all through a thick steel block. T_t is the transmissitivity per unit length of the medium containing the transmitted ray.

Δ_{sr} and Δ_{st} are the respective distances travelled by the reflected and transmitted rays. These distances are multiplied by T_r and T_t to calculate the percentage of reflected and transmitted light rays that actually arrive at the surface. T_r and T_t are both defined in the range:

$$0 \leq T_r, T_t \leq 1$$

A value of 0 means that there is zero transmissitivity, while a value of 1 means there is full transmissitivity (i.e. the medium is a vacuum).

4.8 THE HALL RENDERING MODEL

Having described each included part, we now tie them together to give a full rendering equation. The *Hall shading model* is a reasonably complex rendering equation that incorporates various effects required to produce photorealistic images. The terms in the model are as laid out in *Table 4.1*.

	Light sources	Other bodies
Specular reflection	$k_{sr}\Sigma_j[I_{lj}F_{sr}(\theta_r)(\mathbf{N}\cdot\mathbf{H}_j)^n]$	$k_{sr}I_{sr}F_{sr}(\theta_r)T_r^{\Delta sr}$
Specular transmission	$k_{st}\Sigma_j[I_{lj}F_{st}(\theta_t)(\mathbf{N}\cdot\mathbf{H}_{j'})^{n'}]$	$k_{st}I_{st}F_{st}(\theta_t)T_t^{\Delta st}$
Diffuse reflection	$k_{dr}\Sigma_j[I_{lj}F_{dr}(\mathbf{N}\cdot\mathbf{L}_j)]$	$k_{dr}I_aF_{dr}$
Diffuse transmission	$k_{dt}\Sigma_j[I_{lj}F_{dt}(\mathbf{N}\cdot\mathbf{L}_{j'})]$	$k_{dt}I_aF_{dt}$

Table 4.1 The terms in the Hall shading model.

By summing the eight terms in the Hall model we will calculate the intensity of the incident light ray.

$$
\begin{aligned}
I = \quad & k_{sr}\Sigma_j[I_{lj}F_{sr}(\theta_r)(\mathbf{N}\cdot\mathbf{H}_j)^n] & + \\
& k_{st}\Sigma_j[I_{lj}F_{st}(\theta_t)(\mathbf{N}\cdot\mathbf{H}_{j'})^{n'}] & + \\
& k_{dr}\Sigma_j[I_{lj}F_{dr}(\mathbf{N}\cdot\mathbf{L}_j)] & + \\
& k_{dt}\Sigma_j[I_{lj}F_{dt}(\mathbf{N}\cdot\mathbf{L}_{j'})] & + \\
& k_{sr}I_{sr}F_{sr}(\theta_r)T_r^{\Delta sr} & + \\
& k_{st}I_{st}F_{st}(\theta_t)T_t^{\Delta st} & + \\
& k_{dr}I_aF_{dr} & + \\
& k_{dt}I_aF_{dt} &
\end{aligned}
$$

For efficiency we move the constant terms out of the summation loops (ie the Σ_j [] loops). That is, the values F_{dr} and F_{dt} are moved out of their respective loops in the third and fourth parts of the equation.

We also move the ambient light terms into the diffuse and transmission curves. This yields an efficient Hall rendering equation of:

$$
\begin{aligned}
I = \quad & k_{sr}\Sigma_j[I_{lj}F_{sr}(\theta_r)(\mathbf{N}\cdot\mathbf{H}_j)^n] & + \\
& k_{st}\Sigma_j[I_{lj}F_{st}(\theta_t)(\mathbf{N}\cdot\mathbf{H}_{j'})^{n'}] & + \\
& k_{dr}F_{dr}I_a\Sigma_j[I_{lj}(\mathbf{N}\cdot\mathbf{L}_j)] & + \\
& k_{dt}F_{dt}I_a\Sigma_j[I_{lj}(\mathbf{N}\cdot\mathbf{L}_{j'})] & + \\
& k_{sr}I_{sr}F_{sr}(\theta_r)T_r^{\Delta sr} & + \\
& k_{st}I_{st}F_{st}(\theta_t)T_t^{\Delta st} &
\end{aligned}
$$

Fig. 2.4 shows a ray tree. A value for I is calculated at every ray/object intersection on the tree. The intensity of the specularly reflected ray (I_{sr}) and specularly transmitted ray (I_{st}) at each intersection point are the respective I values calculated when the reflected/transmitted rays themselves intersect objects one level further down the tree.

The I value at the root of the tree is returned as the colour of the pixel ray.

4.9 OTHER RENDERING MODELS

The Hall rendering equation is a general rendering model. As the quest for greater visual realism continues, various specialised models have been developed. These include those by Nishita and Nakamae [NISH86] for shading objects illuminated by natural sunlight, by Max [MAX86], for dealing with atmospheric illumination, by Inakage [INAK89], also dealing with atmospheric illumination, by Cohen and Greenberg [COHE85] for dealing with the use of radiosity when catering for diffuse reflection in complex environments, by Blinn [BLIN82] for dealing with light in clouds and dusty surfaces, and by Potmesil and Chakravarty [POTM81] for dealing with a lens and aperture camera model for rendering.

The *PRIME* System

5.1 Introduction

PRIME (*P*hoto*R*ealistic *I*mage *M*odelling *E*nvironment) is a ray tracing system developed by the author of this thesis. The code is based on the topics discussed in the previous chapters.

5.2 DEVELOPMENT ENVIRONMENT

PRIME was developed on an *IBM Personal System/2 Model 70* computer (PS/2 model 70). The PS/2 model 70 has an *Intel 80387 processor*, running at 20Mhz clock speed.

The PS/2 used to develop *PRIME* has an *Intel 80386 math coprocessor*. This coprocessor is a must when running a program that has a large number of floating point arithmetic operations, such as *PRIME*.

The PS/2 used for development has 8 megabytes of *extended memory*. One megabyte of this memory is used as a virtual disk, or *vdisk*. Using a vdisk we can store files in RAM memory, rather than on a disk. Every read/write is treated as a memory read/write rather than the much slower disk read/write. Before running *PRIME*, copies of all the files it accesses are copied onto the vdisk. As much file reading/writing is carried out in *PRIME*, the vdisk greatly improves the processing speed.

The rest of the extended memory is used to increase the compile time speed.

As standard, all PS/2 machines come with a VGA device adaptor and monitor. This allows for a colour palette of only 16 colours, which is not enough for photorealistic image generation. Instead of using the VGA, an *IBM 8514/A* device adaptor and an *IBM 8514* monitor were used. This allows for a palette of 256 out of a possible 16 million colours to be represented. The 8514 monitor supports both the VGA and 8514/A adaptor cards, so *PRIME* can operate by using either mode.

PRIME was developed using Borland's *Turbo C*. This environment was chosen over the other C development environments because it provides a full set of low level graphics interface routines for the 8514/A adaptor.

5.3 MODELLING

The program **Model.c** is used to model scenes in the world coordinate system. It contains the code needed to define objects and to transform these objects into the world coordinate system.

5.3.1 Primitives in *PRIME*

Model.c is a *solid modeller*, as described in *Section 3.5*. All objects generated in *PRIME* consist entirely of a set of primitives that are transformed and then combined using a CSG tree. There are five primitive types available in *PRIME*. Three of them, the *unit sphere*, the *unit cylinder*, and the *unit cone* are quadrics. The other two, the *unit cube* and the *unit pyramid*, are polygonal solid objects. All five primitives are shown in *Fig. 5.1*. As described in *Section 3.3.5*, each of the five primitives is defined in its own primitive coordinate system that minimises the processing needed for ray/primitive intersection. The five primitives are defined as:

Unit Sphere The unit sphere has its centre at the origin and a radius of one.

Unit Cylinder The unit cylinder has a height of one and a radius of one. It lies on the $+Z$ axis, ranging from $Z = 0$ to $Z = 1$.

Unit Cone The unit cone has a height of one and a radius of one at its base. It lies on the $+Z$ axis, ranging from $Z = 0$ to $Z = 1$. Its apex is at the origin.

Unit Cube The unit cube has six faces, each of which has one by one dimensions. It lies on the $+Z$ $+Y$ $+Z$ axes.

Unit Pyramid The unit pyramid has a base of one by one lying on the $+X$ $+Z$ axes at $Y = 0$. It has a height of one, with its apex at the point $\mathbf{P} = [0.5 \quad 1.0 \quad 0.5]$.

We must input the following data for each primitive that we have in the scene.

PrimType
t_x t_y t_z
s_x s_y s_z
r_x r_y r_z
pt_x pt_y pt_z
outsideMap1 *insideMap1*
... ...
... ...
outsideMapN *insideMapN*

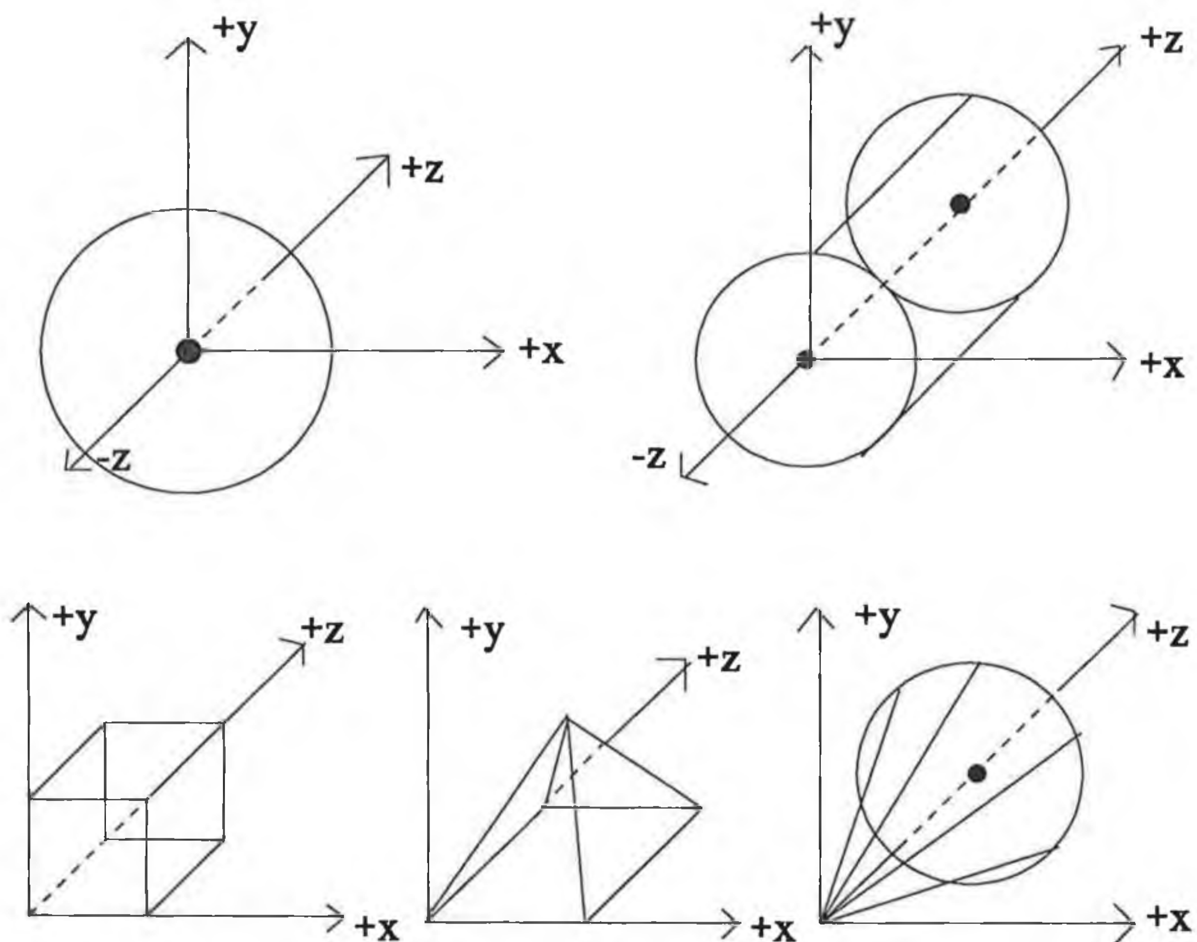


Fig. 5.1 The five primitive types used in *PRIME.*

where: *Primtype* states which of the five types this primitive is.

t, *s*, and *r* represent the translation, scaling and rotation needed to construct the transformation matrix for the primitive.

pt is the point about which we wish to scale and rotate.

Each surface on the primitive has a texture map associated with it. For a sphere there is only one surface, while a cube has six surfaces, each of which may have a separate texture map. We also require a texture map for the inside of every surface. As shown in the bottom left and the right hand objects in *Colour Plate 3*, the inside of a primitive can be of a different colour to the outside.

We must also input the various surface attributes that are needed to properly render each object. The full list of surface attributes we must input is:

kDR	The diffuse reflectance coefficient.
kDT	The transmissive reflectance coefficient.
fDR	The diffuse reflectance curve for object.
tT	The transmissitivity per unit length of object.
kRH	The specular reflection highlight coefficient.
kTH	The specular transmission highlight coefficient.
indx	The refractive index of object with respect to a vacuum.

The meaning of each of these surface characteristics is described in *Section 4.7.*

5.3.2 Objects In PRIME

In *PRIME*, objects are built by using constructive solid geometry. As described in *Section 3.6*, objects are built by combining simple primitives and other objects by the use of *union*, *difference*, and *intersection* operations.

In order to model a scene, we need to input data for the CSG tree of each object in the scene. The necessary data for any object's CSG tree is:

numNodes
operator1 (*leftChild* *rightChild*) **or** *leafNode1*
...
...
operatorN (*leftChild* *rightChild*) **or** *leafNodeN*

where:

numNodes is the number of nodes in the CSG tree.

operatorI is one of the three possible boolean operators; *union*, *intersection*, and *difference*.

leftChild and *rightChild* are the two child nodes to which the operator is to be applied.

leafnodeI is either a primitive or an object.

An example input for a CSG tree could be:

5
+ *1 4*
- *2 3*
B1
B2
C1

Diagramatically, this would produce the CSG tree shown in *Fig. 5.2*. This is the CSG tree of the object constructed in the bottom left section of *Colour Plate 2*.

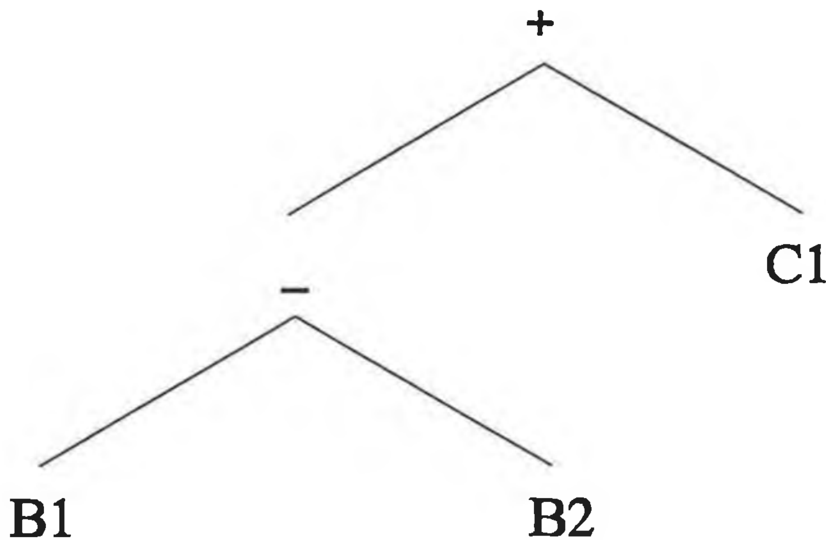


Fig. 5.2

A Roth diagram methodology is used to build up the path a ray takes through an object. A linked list, refereed to as a *tList*, is used to keep an ordered list of intersections that occur between a ray and an object. A *tList* node contains the ray's *t* value at one point of intersection with the object. There is one *tList* node per ray/object intersection, so a *tList* will consist of at least two nodes. *tLists* are implemented by using dynamically allocated memory. By dynamically allocating memory we can ensure that there is no predefined limit as to the number of intersections allowed between a ray and an object. This means that we can build up arbitrarily complex objects. Each of the three operations, *union*, *intersection*, and *difference*, have two *tLists* passed to them and return the one *tList* representing the state of the ray/object intersection after the operation has been performed on the two *TLists*.

In *PRIME*, the building of the CSG tree and the Roth diagram are combined into one

operation. As each new primitive is added to the CSG tree, the Roth diagram is updated so as to reflect the current relation between the tree and the ray.

The code for the CSG tree and Roth methodology, along with the code implementing each of the three operators, is contained in **CSG.c**.

5.3.3 Copying Objects

We can make multiple copies of any object. As with the original objects, copied objects can be transformed within a scene. If, for example, we were modelling a chess board, we would need multiple copies of the same object for each of the sixteen pawns. We would also use multiple copies of objects for the remaining chess board pieces. The code for copying objects is contained in **Obj.c**.

5.4 INTERSECTION ROUTINES IN PRIME

In *PRIME*, intersection is done at the primitive level. The ray/primitive intersections are combined in Roth trees, thus representing ray/object intersections.

In order to implement intersection, we need to give a mathematical definition for a ray.

5.4.1 Mathematical Definition of a Ray

Mathematically, all rays are similar. They are defined in terms of an *origin*, $\mathbf{R}_o$, a unit direction vector, $\mathbf{R}_d$, and a distance parameter t.

$\mathbf{R}_0$ is the point from which all eye rays originate, as described in *Section 2.3*.

$\mathbf{R}_d$ is the direction, in 3D space, in which a given ray is travelling. We need to normalise $\mathbf{R}_d$, otherwise t will be inconsistent when a ray is transformed into different primitive coordinate systems.

By varying the length of t we can generate every possible point on a ray. As t increases positively, we generate points on the ray that lie in front of the origin. These are the only points on the ray we are interested in. A negative value of t represents a point along the ray behind the origin. As the origin represents the eye, as per *Section 2.3*, points behind it cannot be seen. A value of $t = 0$ represents the origin point. This is not included as a valid t value, as it can lead to precision problems.

A ray is defined as:

Set of points on the line $\mathbf{R}(t) = \mathbf{R}_o + \mathbf{R}_d * t$

Where: $\mathbf{R}_o$ = Ray Origin $= [x_o \ y_o \ z_o]$

$\mathbf{R}_d$ = Ray Direction $= [x_d \ y_d \ z_d]$

$x_d^2 + y_d^2 + z_d^2 = 1$ (i.e. $\mathbf{R}_d$ is normalised)

$t > 0$

5.4.2 Ray/Primitive Intersection

In any ray/primitive intersection routine, we need not find the point of intersection between a ray and a primitive, but the t value for this point of intersection. If a ray/primitive intersection returns $t > 0$, then an intersection has occured. In addition to telling us when a ray intersects a primitive, the t value states how far along the ray's path the intersection has taken place.

By substituting t into the ray equation and solving, we obtain the point of intersection.

At most, only one intersected primitive will not be obstructed by other primitives. Of all the intersected primitives for a given ray, the unobstructed primitive is nearest to the ray's origin. The unobstructed primitive will have the smallest t value of all intersected primitives. All other intersected primitives will be obstructed by at least this first primitive, so the ray/primitive intersections for these primitives are invalid.

A separate ray/primitive intersection routine must be written for each of the five primitive types available in *PRIME*. The source code for all five ray/primitive intersection routines is found in **Intersect.c**

5.4.2.1 Ray/Sphere Intersection

The equation of a sphere's surface is:

$$X^2 + Y^2 + Z^2 = 1$$

where:

$X \ Y \ Z$ are points on the surface of the sphere.

Substituting the ray equation into the sphere's equation gives:

$$(X_o + X_d t)^2 + (Y_o + Y_d t)^2 + (Z_o + Z_d t)^2 = 1$$

In terms of t, this simplifies to:

$$t^2(A) + 2t(B) + C = 0$$

where:

$$A = X_d^2 + Y_d^2 + Z_d^2$$
$$B = X_o X_d + Y_o Y_d + Z_o Z_d$$
$$C = X_o^2 + Y_o^2 + Z_o^2 - 1$$

This is a quadratic polynomial in t and can be solved with the quadratic equation. Therefore, the solution for t is:

$$t = \frac{-2B \pm \sqrt{((2B)^2 - 4AC)}}{2A}$$

OR, dividing across by 2:

$$t = \frac{-B \pm \sqrt{(B^2 - AC)}}{A}$$

As the ray direction vector is normalised:

$$A = X_d^2 + Y_d^2 + Z_d^2 = 1$$

This means we can drop A out of the equation, giving:

$$t = -B \pm \sqrt{(B^2 - C)}$$

where:

$$B = X_o X_d + Y_o Y_d + Z_o Z_d$$
$$C = X_o^2 + Y_o^2 + Z_o^2 - 1$$

A further speedup in processing can be obtained by taking into account that if:

$$\sqrt{(B^2 - C)} < 0$$

then the ray cannot intersect the sphere. Should we encounter this case, we need not do any further processing and can return immediately from the routine.

5.4.2.2 Ray/Cylinder Intersection

A unit cylinder is defined in terms of its three surfaces. One surface defines an *infinite cylinder* as a quadric equation. The other two surfaces are *unit circular planes*. They cap the infinite cylinder, making it one unit in length. We need a ray/surface intersection routine for each of the three surfaces.

5.4.2.3 Ray/Infinite Cylinder Intersection

We follow the same path of derivation for this intersection as we did for a sphere in *Section 5.4.2.1*. The equation for an infinite cylinder along the z axis (as per *Fig 5.1*) is:

$$X^2 + Y^2 - 1 = 0$$

Substituting the ray equation into this gives:

$$(X_o + X_d t)^2 + (Y_o + Y_d t)^2 - 1 = 0$$

In terms of t, this is:

$$t^2(X_d^2 + Y_d^2) + 2t(X_o X_d + Y_o Y_d) + (X_o^2 + Y_o^2) - 1 = 0$$

Therefore:

$$t = \frac{-B \quad \sqrt{(B^2 - AC)}}{A}$$

Where:

$$A = X_d^2 + Y_d^2$$
$$B = X_o X_d + Y_o Y_d$$
$$C = X_o^2 + Y_o^2 - 1$$

Again if:

$$\sqrt{(B^2 - C)} < 0$$

Then the ray cannot intersect the infinite cylinder.

5.4.2.4 Ray/Circular Plane Intersection

The two unit circles that cap the infinite cylinder are defined on separate planes. The first is defined as:

$$Z = 0$$

This means that any point whose Z value is 0 is on the plane.
The second unit circle is defined as:

$$Z = 1$$

When testing for a ray/circle intersection, we must test for an intersection between the ray and each of the circles. We shall derive the test for the unit circle on the plane $Z_s = 0$.

If a ray is parallel to the plane on which the unit circle is defined, then the ray cannot intersect the circle.
A ray is parallel to the plane if:

$$Z_d = 0$$

Therefore, any ray where $Z_d = 0$ cannot intersect the circle.

For those rays that pass the above test, we need to find which point along the ray's path that intersects the plane.

Substituting the ray's equation into the plane's equation gives:

$$Z_o + Z_d t = 0$$

Therefore:

$$t = \frac{Z_o}{Z_d}$$

t must be greater than 0, as the plane must be in front of the ray. This gives:

$$t = \frac{Z_o}{Z_d} > 0$$

Again, if this test fails, no intersection can take place, so we stop testing.

We place t back into the ray's equation and solve to give a point **P** = $[x \quad y \quad 0]$. **P** is the point where the ray intersects the plane.

We now test **P** to see if it lies inside the unit circle. The unit circle in the $Z = 0$ plane is defined as:

$$\sqrt{(X^2 + Y^2)} = 1$$

We can drop the $\sqrt{()}$ from the equation, giving:

$$X^2 + Y^2 = 1$$

Therefore, for any values where:

$$X^2 + Y^2 \leq 1$$

The ray intersects the circle.

Testing for the unit circle on the plane Z = 1, is similar to the above.

5.4.2.5 Ray/Cone Intersection

A cone is defined by an infinite cone whose tip is at the origin and a unit circle on the Z = 1 plane.
Intersection of the circle is described in the previous section.

Derivation for the intersection of a ray/infinite cone proceeds along the same lines as for a sphere and infinite cylinder. An infinite cone is defined as:

$$X^2 + Y^2 - Z^2 = 0$$

Substituting the ray equation into this, gives:

$$(X_o + X_d t)^2 + (Y_o + Y_d t)^2 - (Z_o + Z_d t)^2 = 0$$

In terms of t:

$$t^2(X_d^2 + Y_d^2 - Z_d^2) + 2t(X_o X_d + Y_o Y_d - Z_o Z_d) + X_o^2 + Y_o^2 - Z_o^2 = 0$$

Thus giving a quadratic equation in terms of t:

$$t = \frac{-B \pm \sqrt{(B^2 - AC)}}{A}$$

where:

$$A = X_d^2 + Y_d^2 - Z_d^2$$
$$B = X_o X_d + Y_o Y_d - Z_o Z_d$$
$$C = X_o^2 + Y_o^2 - Z_o^2$$

5.4.2.6 Ray/Cube Intersection

A unit cube is defined by the volume enclosed by the six planes. The six planes are:

$$X = 0,\ X = 1,\ Y = 0,\ Y = 1,\ Z = 0,\ \text{and } Z = 1$$

To test for intersection of a unit cube, we must test each of its six faces. We will derive the intersection of a ray and the face $Z = 0$.

We first test to check if the ray is parallel to the plane. If a ray intersects the plane, we then find what the point of intersection is. Both of these tests are the same as for a unit circle on a given plane.
Finally we test the point of ray/plane intersection to ensure it lies inside the square:
The square is bounded by the range:

$$0 \leq X, Y \leq 1$$

Any point that lies inside this range is accepted, while all others are rejected as valid intersections.
Testing each of the other five faces of the cube is similar.

5.4.2.7 Ray/Pyramid Intersection

A unit pyramid is defined in terms of five faces. These are:

The plane:

$$Y = 0,$$

and the four polygons joining this plane to the point **P** = [0.5 1.0 0.5].

We intersect the plane, $Y = 0$, as we would for a cube. We use a ray/polygon intersection routine to test for intersection between the ray and each of the other polygons.

5.4.2.8 Ray/Polygon Intersection

The plane in 3D space that a polygon lies upon can be defined as:

$$AX + BY + CZ + D = 0$$

where:

$$A^2 + B^2 + C^2 = 1$$

The unit normal vector of the plane is defined by:

$$\mathbf{P_n} = [A \quad B \quad C]$$

We substitute the ray equation into this, to give:

$$A(X_o + X_d t) + B(Y_o + Y_d t) + C(Z_o + Z_d t) + D = 0$$

Solving for t:

$$t = \frac{-(AX_o + BY_o + CZ_o + D)}{AX_d + BY_d + CZ_d}$$

If:

$$AX_d + BY_d + CZ_d = 0$$

Then the ray is parallel to the plane containing the polygon, so no intersection occurs.

Otherwise, place t back into the ray equation and solve to get the point of intersection.

We project the polygon onto a 2D plane such that its topology remains unchanged. This can be achieved by removing the X Y Z coordinate whose corresponding plane equation value is of the greatest absolute value. We assign U and V to the remaining two coordinates. The point of intersection is also projected onto the U V 2D plane.

The 2D polygon is translated so that the point of intersection is at the origin. Starting at the origin, we move along the $+U$ axis. We count the number of polygon edges that the $+U$ axis intersects. If there are an odd number of $+U$ axis/polygon edge intersections, then the point lies inside the polygon, otherwise it lies outside the polygon.

5.5 TEXTURE MAPS

In *PRIME*, all texture maps are represented as two dimensional arrays. They are input by the user in the following format:

mapNum
dimX *dimY*
$r_{0,0}$ $r_{0,1}$ ... r_{dimY-1}
$r_{1,0}$
... ...
$r_{dimX-1,0}$ ... $r_{dimX-1,dimY-1}$

$g_{0,0}$ $g_{0,1}$ ... g_{dimY-1}
$g_{1,0}$
... ...
$g_{dimX-1,0}$ ... $g_{dimX-1,dimY-1}$

$b_{0,0}$ $b_{0,1}$ ... b_{dimY-1}
$b_{1,0}$
... ...
$b_{dimX-1,0}$ ... $b_{dimX-1,dimY-1}$

where: *mapNum* identifies the texture map.
dimX and *dimY* are the dimensions of the 2D array containing the texture map.

Texture map details are kept for each of the three primary colours, *r* (red), *g*(green), and *b* (blue).

We need to be able map the surface of each object in a 3D scene onto any 2D texture map. This is done by inverse mapping.

Normalised index values are returned by the inverse mapping procedures in *PRIME*. As they are normalised, the index values will always lie in the range (0..1). As all the inverse mapping procedures return normalised *U*, *V* values, the same texture map array can be used with any primitive type. The normalised index values are denoted *U* and *V*. They correspond to *dimY* and *dimX* respectively.

We need to derive an inverse mapping routine for each primitive type in *PRIME*.

5.5.1 Inverse Mapping of a Sphere

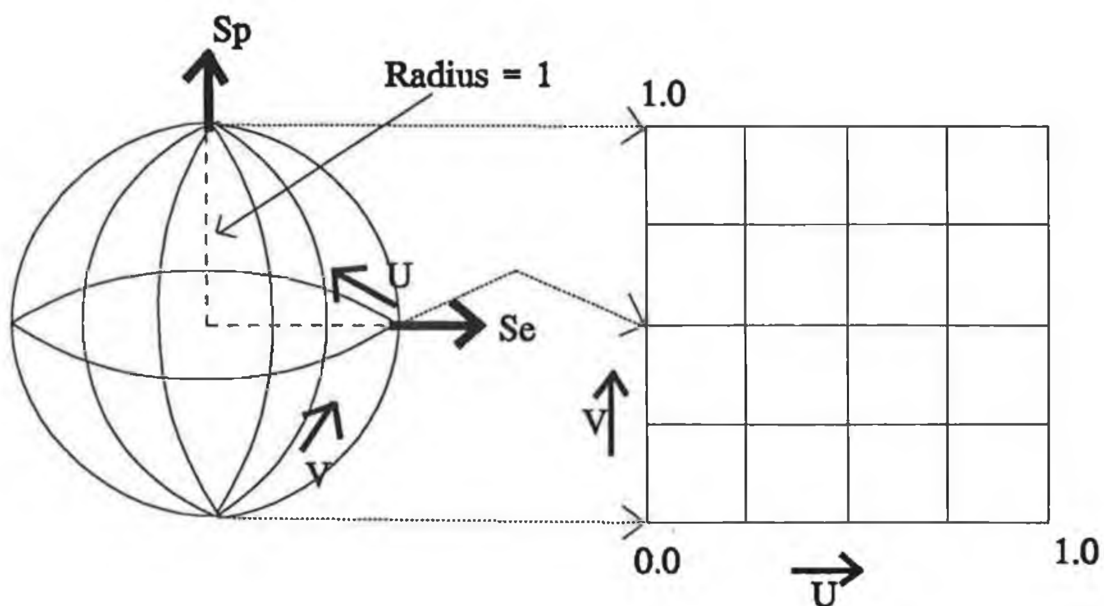


Fig. 5.3 Inverse mapping for a sphere.

We denote the surface normal at the point of intersection on the sphere as $\mathbf{S_n}$. We describe the sphere by the unit vector pointing towards its north pole, $\mathbf{S_p}$, and the unit vector pointing towards its equator, $\mathbf{S_e}$, as shown in *Fig. 5.3*.
As the pole is perpendicular to the equator:

$$\mathbf{S_p} \cdot \mathbf{S_e} = 0$$

At the two poles, the parameter, *U*, is defined to be 0.
U ranges (0..1) starting at the +*X* axis, moving towards the +*Y* axis along the equator.
V ranges (0..1) from the south pole to the north pole.

From these definitions, *V* at the point of intersection is equal to the arc-cosine of the dot product between the intersection's normal and the north pole:

$$\theta = \arccos(-\mathbf{S_n} \cdot \mathbf{S_p})$$

$$V = \frac{\theta}{\pi}$$

If *V* is equal to 0 or 1, then *U* equals 0. Otherwise

$$U = \frac{\arccos((\mathbf{S_e} \cdot \mathbf{Sn}) / \sin(\theta))}{2\pi}$$

If $((\mathbf{S_p} \otimes \mathbf{S_e}) \cdot \mathbf{S_n}) < 0$ (see footnote [1])
Then $U = 1 - U$

[1] ⊗ is the cross product for two vectors.

5.5.1.1 Inverse Mapping for a Cylinder

We need to derive an inverse map for the cylindrical surface and for each of the two unit circular planes.

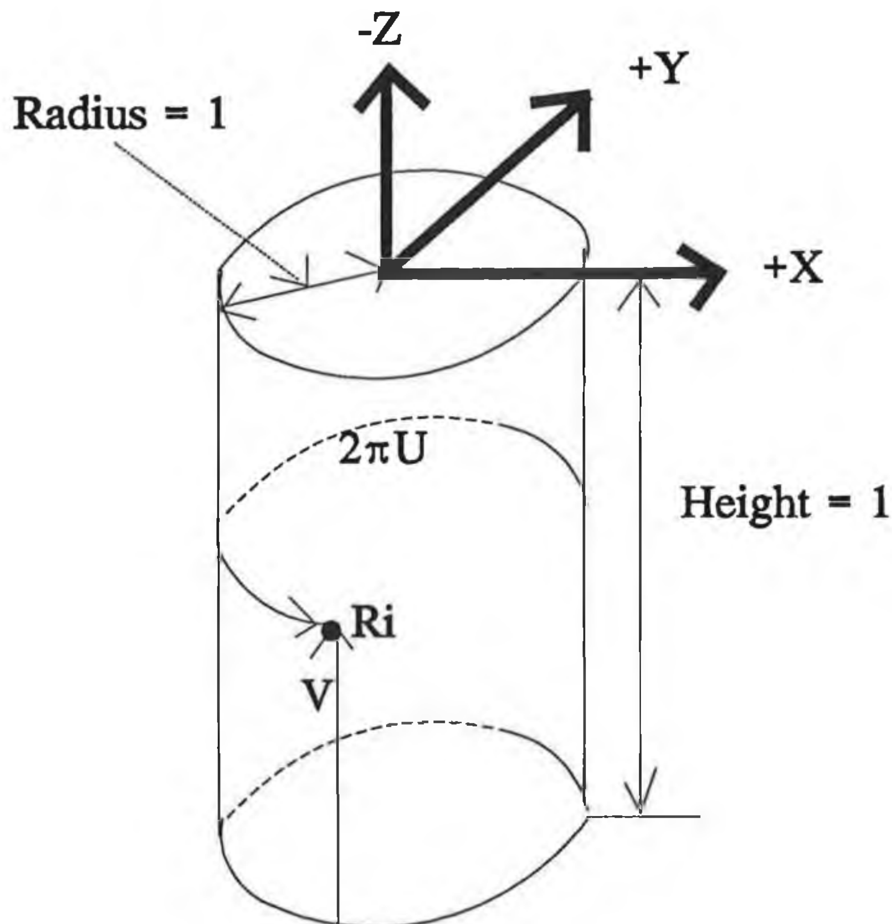


Fig. 5.4 Inverse mapping for a cylinder.

Inverse mapping for the cylindrical surface is shown in *Fig. 5.4*. It is defined as:

U ranges (0..1) starting at the $+X$ axis, moving towards the $+Y$ axis.

V ranges (0..1) starting at $Z = 0$, moving towards $Z = 1$.

Given a point of intersection:

$$\mathbf{P_i} = [X_i \; Y_i \; Z_i]$$

Then

$$V = Z_i$$

$$U = \frac{\arccos(X_i)}{2\pi}$$

If $Y_i < 0$

Then $U = 1 - U$

5.5.1.2 Inverse Mapping for a Circle

We define a circle, lying on the $Z = 0$ plane as:

$$X^2 + Y^2 = 1$$

We also have an intersection point:

$$\mathbf{R}_i = [X_i\ Y_i\ 0]$$

U ranges (0..1) starting at the $+X$ axis, moving towards the $+Y$ axis.

V ranges (0..1) starting at the centre of the circle, moving towards the edge. This mapping is shown in *Fig. 5.5*.

$$V = \sqrt{(X_i^2 + Y_i^2)}$$

$$U = \frac{\arccos(X_i / \sqrt{(X_i^2 + Y_i^2)}}{2\pi}$$

If $Y_i < 0$

Then $U = 1 - U$

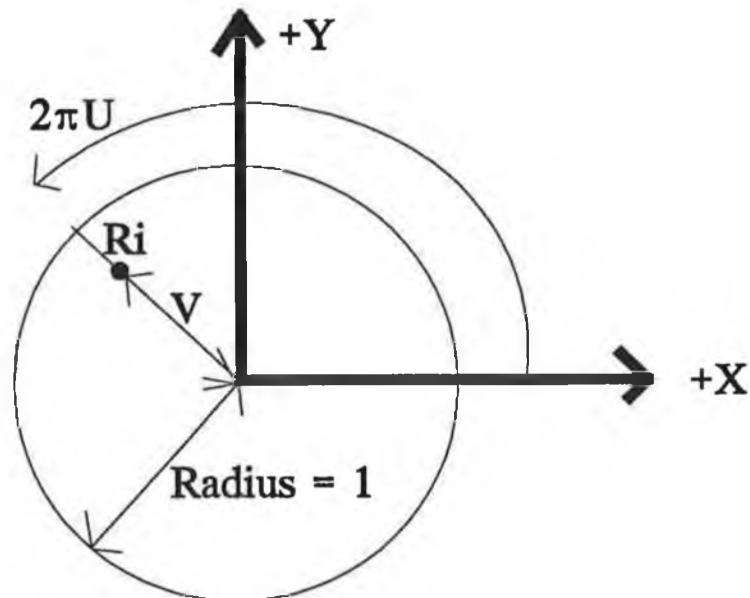


Fig. 5.5 Inverse mapping for a circle.

For the unit circle on the plane $Z = 1$, we must reverse the direction of U, as this circle's outside surface points in the opposite direction to the outside surface of the unit circle on the $Z = 0$ plane.

5.5.1.3 Inverse Mapping for a Cone

We define a cone as:

$$X^2 + Y^2 - Z^2 = 0$$

U ranges (0..1) starting at the $+X$ axis moving towards the $+Y$ axis.

V ranges (0..1) starting at $Z = 0$, moving towards $Z = 1$. This mapping is shown in *Fig. 5.6.*

$$V = Z_i$$

$$U = \frac{\arccos(X_i / Z_i)}{2\pi}$$

If $Y_i < 0$

Then $U = 1 - U$

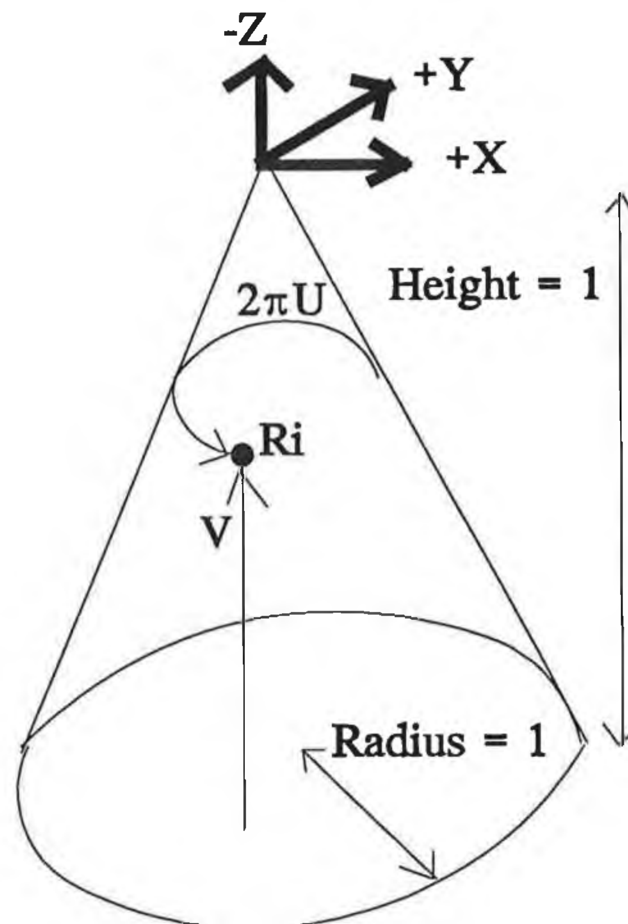


Fig. 5.6 Inverse mapping for a cone.

5.5.1.4 Inverse Mapping for a Cube

Inverse mapping for each of the six faces of a cube is similar. We will derive the inverse map for the face of the cube defined by:

$$Z = 0$$

$$0 \leq X, Y \leq 1$$

U ranges (0..1) starting from $X = 0$, moving towards $X = 1$

V ranges (0..1) starting from $Y = 0$, moving towards $Y = 1$

Given an intersection point:

$$\mathbf{R_i} = [X_i \; Y_i \; Z_i]$$

Then:

$$U = X_i$$

$$V = Y_i$$

5.5.1.5 Inverse Mapping for a Polygon

The complete derivation for this inverse mapping algorithm is complex, so will not be included in this discussion. It is fully developed by Ullner [ULLN83]. We begin by developing a four vertex polygon, or quadrilateral, inverse mapping algorithm.

5.5.1.5.1 Inverse Mapping for a Quadrilateral

We are given an intersection point:

$$\mathbf{R_i} = [X_i \; Y_i \; Z_i]$$

and a polygon defined by the four points:

$$\mathbf{P_i} = [X_i \;\; Y_i \;\; Z_i] \qquad 0 \leq i \leq 3$$

Create a convex quadrilateral, equivalent to the four points of the polygon shown in *Fig. 5.7.*:

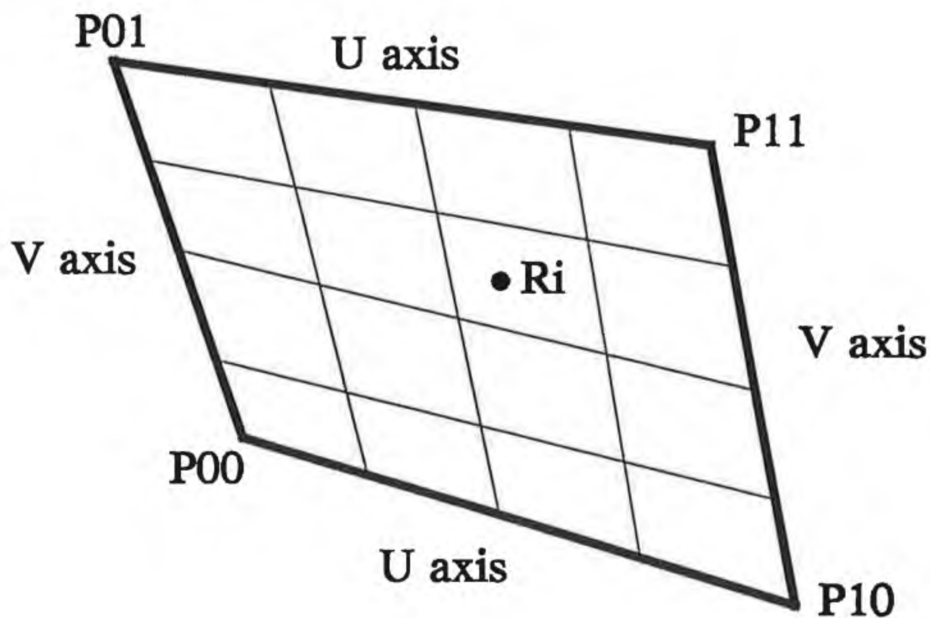


Fig. 5.7 Inverse mapping for a quadrilateral.

The four points of the polygon are defined as:

$$\mathbf{P}_{uv} = [X_{uv} \quad Y_{uv} \quad Z_{uv}]$$

where $U = 0,1$
$V = 0,1$

The surface normal to the plane containing the polygon is denoted $\mathbf{P}_n$.

The plane dependent factors for the algorithm are:

$$\begin{aligned} \mathbf{N}_a &= \mathbf{P}_a \otimes \mathbf{P}_n \\ \mathbf{N}_c &= \mathbf{P}_c \otimes \mathbf{P}_n \\ D_{u0} &= \mathbf{N}_c \cdot \mathbf{P}_d \\ D_{u1} &= \mathbf{N}_a \cdot \mathbf{P}_d + \mathbf{N}_c \cdot \mathbf{P}_b \\ D_{u2} &= \mathbf{N}_d \cdot \mathbf{P}_b \end{aligned}$$

where:

$$\begin{aligned} \mathbf{P}_a &= \mathbf{P}_{00} - \mathbf{P}_{10} + \mathbf{P}_{11} - \mathbf{P}_{01} \\ \mathbf{P}_b &= \mathbf{P}_{10} - \mathbf{P}_{00} \\ \mathbf{P}_c &= \mathbf{P}_{01} - \mathbf{P}_{00} \\ \mathbf{P}_d &= \mathbf{P}_{00} \end{aligned}$$

The basic idea is to define a function for U describing the distance of the perpendicular plane (defined by that U and the quadrilateral's axes) from the coordinate system origin.

$$D(U) = (\mathbf{N}_c + \mathbf{N}_a U) \cdot (\mathbf{P}_d + \mathbf{P}_b U)$$

Given $\mathbf{R}_i$, the distance of the perpendicular plane containing this point is:

$$D_r(U) = (\mathbf{N}_c + \mathbf{N}_a U) \cdot \mathbf{R}_i$$

Setting $D(U)$ equal to $D_r(U)$, solving for U and simplifying, gives a quadratic equation:

$$AU^2 + BU + C = 0$$

where:

$A = D_{u2}$

$B = D_{u1} - (\mathbf{R}_l \cdot \mathbf{N}_a)$

$C = D_{u0} - (\mathbf{R}_l \cdot \mathbf{N}_c)$

If $D_{u2} = 0$

then the U axes are parallel, and the solution is:

$$U = \frac{-C}{B}$$

If $D_{u2} \neq 0$

then the solution is:

$$K_a = D_{ux} + (\mathbf{Q}_{ux} \cdot \mathbf{R}_l)$$
$$K_b = D_{uy} + (\mathbf{Q}_{uy} \cdot \mathbf{R}_l)$$

where:

$$\mathbf{Q}_{ux} = \frac{\mathbf{N}_a}{2D_{u2}}$$

$$D_{ux} = \frac{-D_{u1}}{2D_{u2}}$$

$$\mathbf{Q}_{uy} = \frac{-\mathbf{N}_c}{D_{u2}}$$

$$D_{uy} = \frac{D_{u0}}{D_{u2}}$$

There are two answers:

$$U = K_a \pm \sqrt{(K_a^2 - K_b)}$$

At most one of these values will lie in the range (0..1), so it is the useful one.

A value for V is calculated in a similar way.

5.5.1.5.2 Inverse Mapping for a Triangle

In order to use get the inverse mapping for a polygon with three vertices, we simply

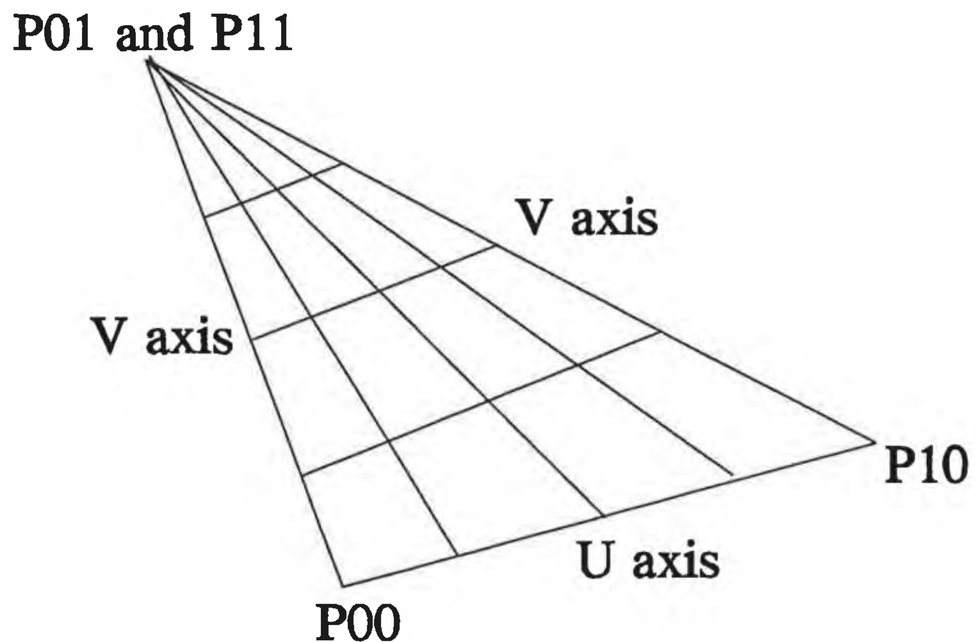


Fig. 5.8 Inverse mapping for a Triangle.

assign both $\mathbf{P_{01}}$ and $\mathbf{P_{11}}$ to the same vertex, as shown in *Fig. 5.8*, and use the quadrilateral inverse mapping just derived.

5.8 BOUNDING VOLUMES

In *PRIME*, any primitive type may be used as a bounding volume. The ray/bounding volume intersection code is similar to that used for ray/primitive intersection, except that ray bounding volume intersection can stop as soon as one intersection is found. Ray/bounding

volume intersection routines for each primitive type are contained in **Bound.c**. Bounding volumes are discussed in *Section 3.7.1*.

5.9 PROBLEMS ENCOUNTERED

Several problems were encountered along the way while developing *PRIME*. Two serious problems involve *numerical precision* and *memory allocation*.

5.9.1 Numerical Precision

Although important throughout the code in *PRIME*, numerical precision is critical is certain areas. For example, when calling the *acos(x)* function, we must insure that x is in the range (-1..1). If x is even slightly outside this range, say by 10E-100 (or less than a billionth of a billionth!), then the program crashes with a domain error.

A second precision critical area is when the union, intersection, and difference operators are being applied to *tLists* while generating a *Roth diagram*. We often need to compare two t values for equality. If the two nodes differ, even by 10E-100, then they are not treated as being equal. This will cause the generation of some very strange looking objects!

In *PRIME*, the routine **RoundOff()** in **MathFunc.c**, rounds off any number that differs from an integer value by less then a threshold minimum amount. The minimun value was obtained by trial and error.

5.9.2 Memory

The *tLists* used to build CSG trees in *PRIME* are connected together as linked lists of dynamically allocated memory. By using linked lists there is no limit to the number of primitives that we can use to construct objects and therefore, we can construct arbitrarily complex objects. A problem with dynamically allocating memory is that it must be freed before program termination. When memory is allocated in *C*, a pointer is returned pointing to the allocated memory block. Memory is freed by passing the pointer to a memory freeing function. If an attempt is made to free a memory block that has not yet been allocated then the potential for disaster arises. The memory freeing function frees the block of memory pointed to by the pointer. This block of memory may be just some available memory, so freeing it and making it available again does no harm. However, the memory may contain

part of the operating system, part of the executable code of *PRIME* that is currently running or some other vital memory. When this block of memory is next accessed by the operating system, by *PRIME*, etc. the computer will crash. This crash will most probably not occur until some time after the memory has been freed. It is therefore almost impossible to track down the culprit pointer.

In *PRIME*, the routines in **Memory.c** are used as a front end to *C*'s memory allocation and freeing routines. Whenever memory is allocated, its address is stored in an array. Before memory is freed, its address is first checked against the table of allocated memory. Should no entry be found, then an error message is issued and the program terminates. It is easy to locate the offending pointer and correct the code accordingly.

An extra function in **Memory.c** is used to compare the number of memory allocations against the number of memory frees. These two figures should be equal. If they are not, then the code must be checked to find out where the unfreed memory allocation is occurring and the program must be changed. This memory check function should be called as the last line of code in a program using dynamic memory allocation.

5.10 COLOUR PLATES

Colour Plate 1 shows a wireframe of a *β-spline* surface, as described in *Section 3.4.2*. For this surface both *skew* and *tension* are set to 1 at every knot. There are 64 control knots contained in an 8 by 8 matrix.

Colour Plate 2 demonstrates *constructive solid geometry*, as described in *Section 3.6*. The top left object is constructed by the union of three primitives. The bottom left object involves both union and difference, while the object to the right involves only difference.

This colour plate also shows how texture maps fit onto various surfaces. Note that the inside of an D.C.U. '91 cube has its own texture map.

Colour Plate 3 is a scene rendered using the *Hall Rendering Model*, that is described in *Section 4.8*. The source code for this shading model is contained in **Hall.c**.

The scene for *Colour Plate 3* consists of a transparent sphere that hovers over a shiny flat chessboard. All the sphere's colouring results from the reflection and transmission of light within the scene.

To allow for a large number of different shades for each colour used, the lookup table for this colour plate is divided into two fields, each four bits in size. One field contains the

red shadings and the other field contains the green shadings. The colour shades are accessed as a 2D array of 16 X 16 dimensions, giving a total of 256 different shades of colour. The array contains all the colour shadings that can be generated by combining red and green. The cost of this method for indexing the lookup table is that we do not include any blue colouring. Because we do not have any blue light component, we cannot generate white light. This is why we use yellow light.

Conclusions And Future Work

6.1 CONCLUSIONS

During the period of this research, ray tracing has increased in popularity. Ray tracing has now left the research laboratory and entered into the *real life* world of commercial computer graphics. Ray tracing has been used in several animated advertisements and for program logo's on television.

The rapid growth of interest in ray tracing is due primarily to the massive increase in capabilities, coupled with a similar reduction in costs of hardware. *PRIME*, the coding part of this research, was developed on a personal desktop computer. Only a few short years ago, development of a system of the size and complexity of *PRIME*, would have been only possible only by using mainframe machines. As the cost/benefit ratio of hardware increases even further, we may expect even greater acceptance of ray tracing as the standard graphical interface for many computing applications.

6.2 FUTURE WORK

There are several improvements which can be made to *PRIME*. These come under two areas; improvements to increase the *speed efficiency* of the system, and improvements to enhance the *rendered image* that appears on a computer monitor.

6.2.1 Speed Efficiency

Here we discuss one extension and one improvement that can be implemented to increase the speed of processing in *PRIME*.

6.2.1.1 Spacial Subdivision

As stated in *Section 3.7, the* major cost in processing time is the testing for ray/object intersections. There are two main ways to cut down on the number of intersection tests that need to be performed, by either using *bounding volumes* or by implementing *spacial subdivision*. In *PRIME* only bounding volumes are utilised. The use of bounding volumes and spacial subdivision are **not** mutually exclusive. Therefore, we can extend *PRIME* to incorporate spacial subdivision. Spacial subdivision is discussed in *Section 3.7.2*.

6.2.1.2 File Management

It was stated in *Section 5.2* that *PRIME* was very heavily file read/write dependent. A vdisk is used to improve file read/writing time. An alternative, that would help remove *PRIME*'s hardware dependency, is the implementation of a more efficient file front end.

Every time a file is opened it can be added to an *open files table*. Whenever a file is closed, it is marked as being closed in the open files table, but is not physically closed. Whenever a file is opened, the open files table is checked for that file name. If the file name is found, then it is simply marked as being opened. If a file is opened that is not found in the open files table, and the open files table is full, we choose one of the files from the table that is marked closed, physically close this file and replace it in the table with the new file. The open files table approach works because of the principles of coherence, as discussed in *Section 3.7.4.4*. The same objects, hence the same files, will be accessed very often over a relatively short period. Once the files are closed, they will remain closed for a relatively long period.

6.2.2 Enhancing the Rendered Image

There are several improvements that can implemented to improve the quality of the rendered image.

6.2.2.1 Extra Modelling Primitives

Adding extra primitives to the five primitive types currently available in *PRIME* is primarily a matter of adding a ray/primitive intersection routine and an inverse mapping routine for each new primitive type added.

The ray/polygon intersection and inverse mapping routines in *PRIME* are written so as to work for any three or four vertex polygon. Therefore, adding new polyhedral solids to *PRIME* is trivial.

6.2.2.2 Texture Maps

As described in *Section 5.5*, texture maps used in *PRIME* are input as simple 2D arrays. For realistic rendering, it would be preferable to use texture maps that are generated by scanning in images using a package such as *AT&T's ScanWare*. The code needed to

convert texture map files generated by using a scanner into *PRIME* texture map formatted files is easily implemented. However, if texture maps are to be generated by using scanned in data, then a colour quantization algorithm, as discussed in *Section 4.4*, would have to be added to *PRIME*. This is because with scanned texture maps, we cannot guarantee that no more than a maximum of *256 predefined* colours will be generated in the final rendered image.

6.2.2.3 Surface Modelling

PRIME is a solid modelling system. An alternative approach, as described in *Section 3.4*, is to use surface modelling. Most surface modelling systems are totally different to solid modelling. However, Carlson [CARL82] describes a surface modelling algorithm that uses the union, intersection, and difference operators normally found only in solid modelling. By using this as a basis, it may be possible to build a system that will deal simultanuously with both solid and surface modelled objects.

6.3 CURRENT AREAS OF RESEARCH

Current research in ray tracing is primarily directed towards speeding up the processing and increasing the variety of visual effects which can be rendered.

6.3.1 Parallel Machines

In *Section 3.7*, we discussed many of the software methods currently being used to increase the speed of ray tracing. Currently, there is also much research into the hardware being used in ray tracing, such as that by Badouel *et al.* [BADO90a,BADO90b], who discuss various ray tracing techniques used with parallel computers.

6.3.2 Radiosity

Much of the current research in ray tracing is within the area of improving the rendering of various visual effects. Some of these areas are listed in *Section 2.6.5*, and *Section 4.9*. Another area of research relating to ray tracing is *radiosity*.

Radiosity is based on principles taken from thermal engineering. Radiosity is applicable to environments composed of diffuse reflectors and transmitters, as discussed in

Section 4.6.4. Radiosity produces the phenomena of ***colour bleeding*** (i.e. variable shading within shadow envelopes), the effect of area light sources and penumbra effects along shadow boundaries. It so happens that diffuse lighting effects are the least effectively rendered effects in ray tracing. By combining radiosity and ray tracing into a single system we can improve the rendered image quality.

Radiosity, as used in conjunction with ray tracing, is discussed by Lang [LANG88].

Bibliography

AMAN84 Amanatides, J., *Ray Tracing With Cones* SIGGRAPH 1984 VOL. 18 #3 JULY PP. 129-135

APPE68 Appel, A., *Some Techniques For Shading Machine Renderings Of Solids*, THOMPSON BOOKS WASHINGTON D.C. 1968 PP. 37-45

ARNA87 Arnalldi B., Priol T., and Bouatouch K., *A New Space Subdivision Method For Ray Tracing CSG Modelled Scenes*, THE VISUAL COMPUTER, SPRINGER-VERLAG 1987 VOL. 3 PP. 98-108

ARVO87 Arvo J. and Kirk D., *Fast Ray Tracing By Ray Classification*, SIGGRAPH 1987 JULY VOL. 21 #4 PP. 55-64

BADO90a Badouel D. and Priol T., *An Effecient Ray Tracing Algorithm On A Distributed Memory Parallel Computer*, INSTITUTE DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALÉATOIRES, INTERNAL PUBLICATION # 506 JANUARY 1990

BADO90b Badouel D., Bouatouch K, and Priol T., *Ray Tracing On Distributed Memory Parallel Computers: Strategies For Distributing Computations And Data*, INSTITUTE DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALÉATOIRES, INTERNAL PUBLICATION # 508 JANUARY 1990

BARR86 Barr A.H., *Ray Tracing Deformed Surfaces*, SIGGRAPH 1986 AUGUST VOL. 20 #4 PP. 287-296

BARR87a Barr A. H. and Snyder J.M., *Ray Tracing Complex Models Containing Surface Tessellations*, SIGGRAPH 1987 JULY VOL. 21 #4 PP. 119-126

BARR87a Barr A. H. and Von Herzen B., *Accurate Triangulations Of Deformed, Intersecting Surfaces*, SIGGRAPH JULY VOL. 21 #4 PP. 103-110

BARS83 Barsky, B.A. and Beatty, J.C., *Local Control Of Bias And Tension In Beta-Splines*, SIGGRAPH 1983 VOL. 17 #3 JULY PP. 193-218

BARS88 Barsky, B.A., *Computer Graphics And Goemetric Modeling Using Beta-Splines*, SPRINGER-VERLAG 1988

BIER83 Bier, E.A., *Solidviews, An Interactive Three-Dimensional Illustrator*, BS and MS THESIS, DEPT. OF EE and CS, MIT MAY 1983

BLIN77a Blinn, J.F., *Models Of Light Reflection For Computer Synthesized Pictures*, COMPUTER GRAPHICS 1977 VOL. 11 #2 PP. 192-198

BLIN77b Blinn, J.F., *A Homogeneous Formulation For Lines In 3-Space*, SIGGRAPH 1977 VOL. 11 #2 PP. 237-241

BLIN78 Blinn, J.F. and Newell, M.E., *Clipping Using Homogeneous Coordinates* SIGGRAPH 1978 VOL. 12 #3 AUGUST PP. 245-251

BLIN82 Blinn, J.F., *Light Reflection Functions For Simulation Of Clouds And Dusty Surfaces*, SIGGRAPH 1982 VOL. 16, #3 JULY PP. 21-29

BOYS82 Boyse J.W. and Gilchrist J.E., *GMsolid: Interactive Modelling For Design And Analysis Of Solids*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 MARCH VOL. 2 #2 PP. 86-97

BROW82 Brown C.M., *PADL-2: A Technical Summary*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 MARCH VOL. 2 #2 PP. 69-84

BURG89a Burger P. and Duncan G., *Interactive Computer Graphics* ADDISON WESLEY 1989 *Color Plate 3*

BURG89b Burger P. and Duncan G., *Interactive Computer Graphics* ADDISON WESLEY 1989 *Color Plate 6*

CARL82 Carlson, W.E., *An Algorithm And Data Structure For 3D Object Synthesis Using Surface Patch Intersections*, SIGGRAPH 1882 VOL. 16 #3 JULY PP. 255-263

COHE85 Cohen, M.F. and Greenberg, D.P., *The Hemi-Cube: A Radiosity Solution For Complex Environments*, SIGGRAPH 1985 VOL. 19 #3 PP. 31-41

COOK81 Cook, R.L. and Torrance, K., *A Reflectance Model For Computer Graphics*, SIGGRAPH 1981 VOL. 15 #3 AUGUST PP. 307-316

COOK86 Cook, R.L., *Stochtastic Sampling In Computer Graphics*, ACM TRANS. GRAPH VOL. 5 #1 1986 JANUARY

COOK88 Cook, R.L., *A Reflectance Model For Realistic Image Synthesis*, MASTERS THESIS CORNELL UNIVERSITY ITHACA NY DECEMBER 1988

CROW77 Crow, F.C., *The Aliasing Problem In Computer-Generated Shaded Images*, COMMUNICATIONS OF THE ACM 1977 VOL. 20 #11 NOVEMBER

DIPP85 Dippè, M.A.Z. and Wold, E.H., *Antialiasing Through Stochastic Sampling* SIGGRAPH 1985 VOL. 19 #3 JULY

DADO85 Dadoun, N. and Kirkpatrick, D.G., *The Geometry Of Beam Tracing*, PROCEEDINGS OF THE SYMPOSINM ON COMPUTATIONAL GEOMETRY 1985 JUNE PP. 55-61

FAUX79 Faux, I.D. and Pratt, M.J., *Computational Geometry For Design And Manufacture*, ELLIS HORWOOD 1979

FUJI86 Fujimoto A., Perrott C.G. and Iwata K,. *Environment For Fast Elaboration Of Constructive Solid Geometry*, ADVANCES IN COMPUTER GRAPHICS (PROCEEDINGS OF COMPUTER GRAPHICS TOKYO 1986) 1986 APRIL PP 20-32

GANA82 Ganapathy S. and Dennehy T.G., *A New General Triangulation Method For Planar Contours*, SIGGRAPH 1982 JULY VOL. 16 #3 PP. 69-75

GLAS84 Glassner, A.S., *Space Subdivision For Fast Ray Tracing*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1984 OCTOBER VOL. 4 #10 PP. 15-22

GLAS89 Glassner, A.S., *An Introduction To Ray Tracing*, ACADEMIC PRESS 1979 PP. 79-119

GOLD71 Goldstein, R.A. and Nagel, R., *3-D Visual Simulation*, SIMULATION 1971 JANUARY VOL. 16 #1 PP. 25-31

GREE79 Gteenburg, D.P. and Kay, D.S., *Transparency For Computer Synthesized Pictures*, SIGGRAPH '79 1979 VOL.13 # 2 AUGUST PP. 158-164

HAIN86 Haines E.A. and Greenberg D.P., *The Light Buffer: A Shadow Testing Accelerator*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1986 SEPTEMBER VOL. 6 #9 PP. 6-16

HANR83 Hanrahan, P., *Ray Tracing Algebraic Surfaces*, SIGGRAPH 1983 JULY VOL. 17 #3 PP. 83-90

HANR86 Hanrahan P., *Using Caching And Breadth-First Search To Speed Up Ray-Tracing*, PROCEEDINGS OF GRAPHICS INTERFACE 1986 MAY PP. 56-61

HECK84 Heckbert, P.S. and Hanrahan, P., *Beam Tracing Polygonal Objects* SIGGRAPH 1984 JULY VOL.18 #3 PP. 119-127

HECK86 Heckbert, P.S., *Survey Of Texture Mapping*, IEEE COMPUTER GRAPH APPLICATION 1986 NOVEMBER VOL. 6 #11 PP. 56-57

INAK89 Inakage, M., *An Illumination Model For Athmospheric Environments*, NEW ADVANCES IN COMPUTER GRAPHICS, PROCEEDINGS OF CG INTERNATIONAL 1989 SPRINGER-VERLAG PP. 533-548

JOY86 Joy K.I. and Bhetanabhotla M.N., *Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques And Ray Coherence*, SIGGRAPH 1986 AUGUST VOL. 20 #4 PP. 279-285

KAJI82 Kajiya, J.T., *Ray Tracing Parametric Patches*, SIGGRAPH 1982 JULY VOL. 16 #3 PP. 245-254

KAJI83 Kajiya, J.T., *New Techniques For Ray Tracing Procedurally Defined Objects*, SIGGRAPH 1983 JULY VOL. 2 #3 PP. 161-181

KAJI86 Kajiya, J.T., *The Rendering Equation*, SIGGRAPH 1986 VOL. 20, #4 PP. 143-150

KAY86 Kay T.L. and Kajiya J.T., *Ray Tracing Complex Scenes*, SIGGRAPH 1986 NOVEMBER VOL. 20 # 4 PP. 269-278

LANG88 Lang L., *Lighting Design; Advances In Ray-Tracing And Radiosity Techniques Improve Lighting Simulation*, COMPUTER GRAPHICS WORLD VOL. 11 #10 1988 OCTOBER PP. 109-114

LEE85 Lee, M.E., Render, R.A. and Uselton, S.P., *Statistically Optimized Sampling For Distributed Ray Tracing* SIGGRAPH 1985 VOL. 19 #3 JULY PP. 61-67

MAED89 Maeder, A.J., *Texture Characterization Using Random Sampling*, NEW ADVANCES IN COMPUTER GRAPHICS, PROCEEDINGS OF CG INTERNATIONAL 1989 SPRINGER-VERLAG PP. 603-612

MAND83 Mandelbrot, B., *The Fractal Geometry Of Nature*, FREEMAN 1983

MAX86 Max, N.L., *Atmospheric Illumination And Shadows*, SIGGRAPH '86 1986 VOL. 20 #4 PP. 117-124

MAXW46 Maxwell, E.A., *Methods Of Plane Projective Geometry Based On The Use Of General Homogeneous Coordinates* CAMBRIDGE UNIVERSITY PRESS 1946

MAXW51 Maxwell, E.A., *General Homogeneous Coordinates In Space Of Three Dimensions* CAMBRIDGE UNIVERSITY PRESS 1951

MITC87 Mitchell, D.P., *Generating Antialiased Images At Low Sampling Densities*, SIGGRAPH 1987 VOL. 21 #4 JULY PP. 65-71

MIYA90 Miyata, K., *A Method Of Generating Stone Wall Patterns*, SIGGRAPH 90 VOL. 24 #4 JULY PP. 387-394

MORA81 Moravec, H.P., *3D Graphics And The Wave Theory*, SIGGRAPH 1981 AUGUST VOL. 15 #3 PP. 289-296

MYER82 Myers W., *An Industrial Perspective On Solid Modelling*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 MARCH VOL. 2 #2 PP. 86-97

NAGE71 Nagel, R. and Goldstein, R.A., *3-D Visual Simulation*, SIMULATION 1971 JANUARY PP. 25-31

NISH86 Nishita, T. and Nakamae, E,. *Continuous Tone Representation Of Three Dimensional Objects Illuminated By Sky Light*, SIGGRAPH 1986 VOL. 20 #4 AUGUST PP. 125-132

OHTA87 Otha M. and Maekawa M., *Ray Coherence Theorm And Constant Time Ray Tracing Algorithm*, COMPUTER GRAPHICS 1987 PP. 303-314

PEAC85 Peachey, D.R., *Solid Texturing Of Complex* Surfaces, SIGGRAPH 85 VOL. 19 #3 NOVEMBER PP. 279-286

PHON75 Phong, Bui-Tuong, *Illumination For Computer Generated Images* COMMUNICATIONS OF THE ACM 1975 VOL. 18 #6 PP. 311-317

PLAS83 Plass M. and Stone M., *Curve-Fitting With Piecewise Parametric Cubics*, SIGGRAPH 1983 JULY VOL. 17 #3 PP. 229-239

PORT84 Porter, T., Cook, R.L. and Carpenter, L., *Distributed Ray Tracing*, SIGGRAPH 1984 VOL. 18 #3 JULY PP. 137-145

POTM81 Potmesil M. and Chakravarty I., *A Lense And Aperture Camera Model For Synthetic Image Generation*, SIGGRAPH 1981 VOL. 15 #3 PP. 297-305

PURD70a Purdue University, *Thermophysical Properties Of Matter, Vol. 7: Thermal Radiative Properties Of Metals*, PLENUM, NY 1970

PURD70b Purdue University, *Thermophysical Properties Of Matter, Vol. 8: Thermal Radiative Properties Of Nonmetallic Solids*, PLENUM, NY 1970

PURD70c Purdue University, *Thermophysical Properties Of Matter, Vol. 9: Thermal Radiative Properties Of Coatings*, PLENUM, NY 1970

PURG86 Purgathofer, W., *A Statistical Method For Adaptive Stochtastic Sampling*, PROCEEDINGS OF EUROGRAPHICS 1986 PP. 145-152

REQU82 Reequicha A. and Voelcker H.B., *Solid Modelling: A Historical Summary And Contemporary Assessment*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 MARCH VOL. 2 #2 PP. 9-24

ROBE65 Roberts, L.G., *Homogeneous Matrix Representations And Manipulations Of N-Dimensional Constructs*, DOCUMENT MS 1405, LINCON LABORATORY, MASSACHUSETTS 1965

ROTH82 Roth, S.D., *Ray Casting For Modeling Solids*, COMPUTER GRAPHICS IMAGE PROCESSING 1982 FEBUARY VOL.18 #2 PP. 109-144

RUBI80 Rubin S. and Whitted T., *A Three-Dimensional Representation For Fast Rendering Of Complex Scenes*, SIGGRAPH 1980 JULY Vol. 14 #3 PP. 110-116

SCHA81 Schaffner, S.C., *Calculation of B-Spline Surfaces Using Digital Filters*, SIGGRAPH 1981 DECEMBER VOL.15 #4 PP. 437-457

SMIT87 Smith, R.A., *Planar 2-Pass Texture Mapping And Warping*, SIGGRAPH 1987 VOL. 21 #4 NOVEMBER PP. 263-272

SPEE85 Speer L.R., DeRose T.D., and Barsky B.A., *A Theoretical And Empirical Analysis Of Coherent Ray Tracing*, COMPUTER GENERATED IMAGES MAY 1986 PP. 11-25

SUTH74 Sutherland I.E., Sproull R.F., and Schumacker R.A., *A Characterization OfTen Hidden-Surface Algorithms*, COMPUTER SURVEY 1974 MARVH VOL. 6 #1 PP. 1-55

TORR67 Torrance, K.E. and Sparrow, E.M., *Theory Of Off-Specular Reflection From Roughened Surfaces* J OPT SOC AM 1967 PP. 1105-1114

TOTH85 Toth D.L., *On Ray Tracing Parametric Surfaces*, SIGGRAPH 1985 JULY VOL. 19 #3 PP. 171-179

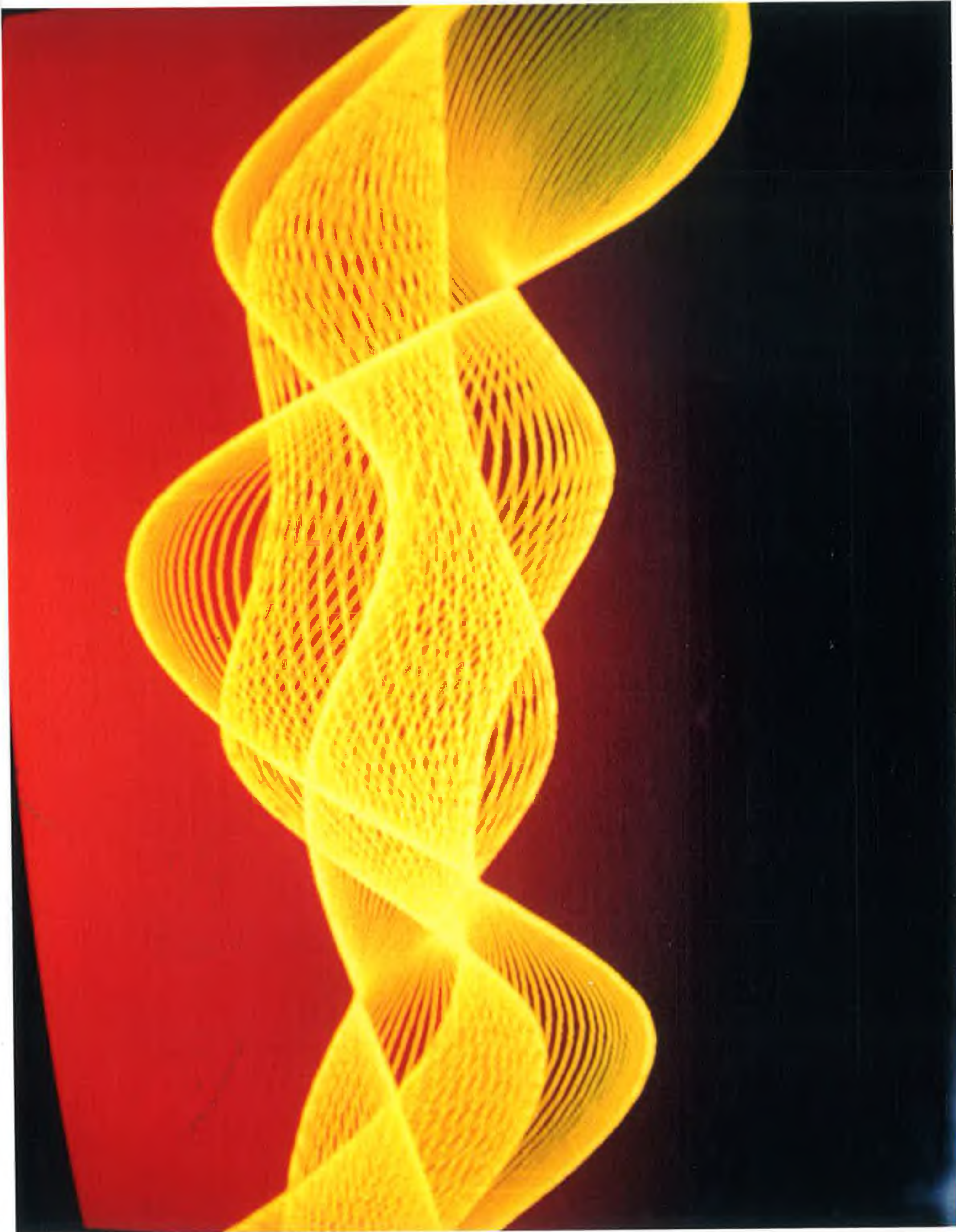
ULLN83 Ullner, M.K., *Parallel Machines For Computer Graphics*, PHD. THESIS, CALIFORNIA INSTITUTE OF TECHNOLOGY, COMPUTER SCIENCE TECHNICAL REPORT 5112 1983

WEGH84 Weghorst H., Hooper G., and Greenberg D., *Improved Computational Methods For Ray Tracing*, SIGGRAPH 1984 JANUARY VOL. 3 #1 PP. 52-69

WHIT80 Whitted, T., *An Improved Illumination Model For Shaded Display*, COMMUNICATIONS OF THE ACM 1980 VOL. 23 #6 JUNE PP. 343-349

WIJK84 Van Wijk, J.J., *Ray Tracing Objects Defined By Sweeping Planar Cubic Splines*, SIGGRAPH 1984 JULY VOL. 3 #3 PP. 223-237

Color Plate 1

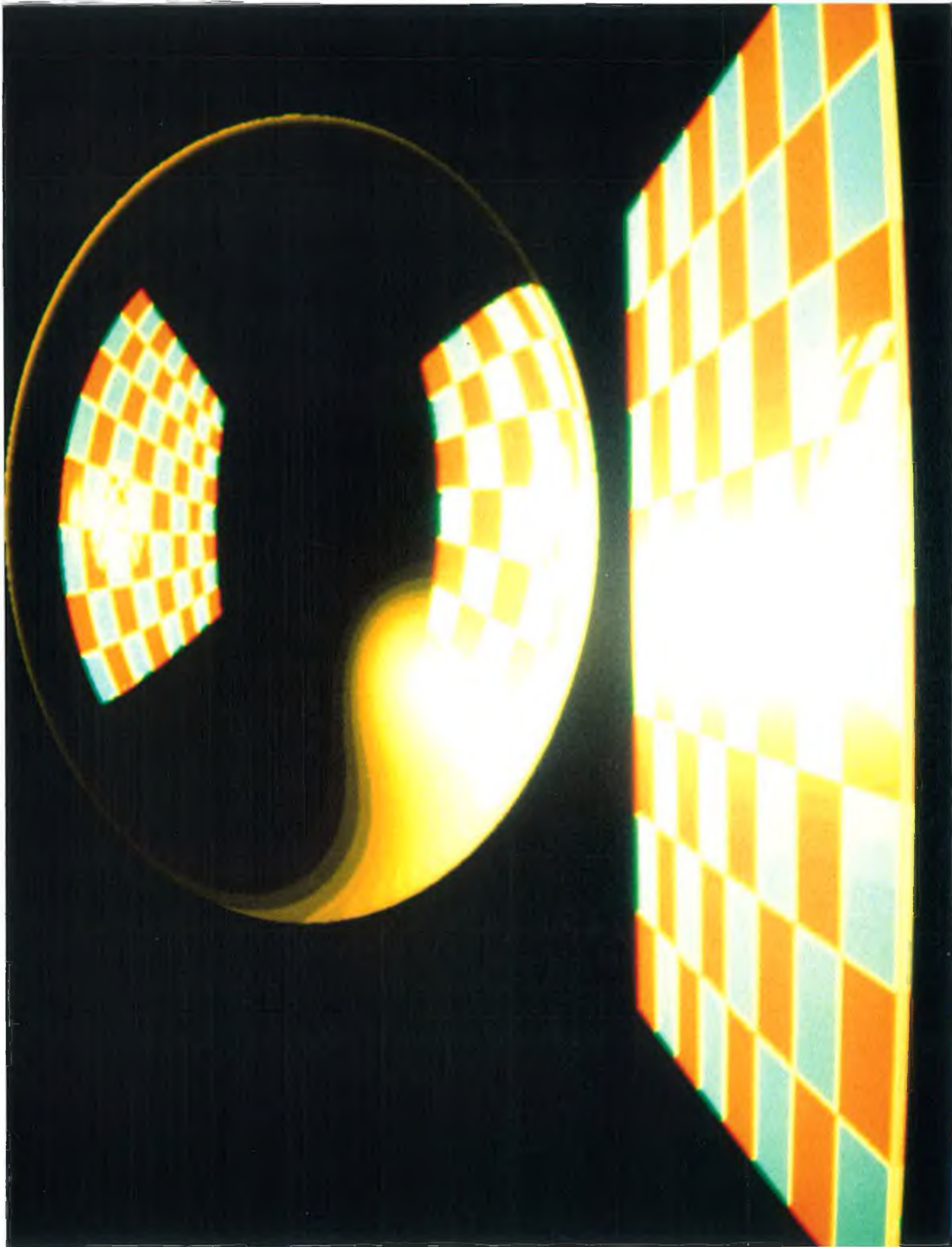


Color Plate 1

Color Plate 2

Color Plate 2

Color Plate 3



Color Plate 3

```
/****************************************************************************/
/*                                                                          */
/* Module: Define.h                                                         */
/*                                                                          */
/****************************************************************************/
#define FALSE 0
#define TRUE !FALSE    /* false is always zero */

#define X 0
#define Y 1
#define Z 2
#define W 3
#define U 0
#define V 1

#define R 0            /* RGB colors. */
#define G 1
#define B 2

#define MaxSurface 6  /* The max number of surfaces in any primitive */
#define Out 0         /* surface outside */
#define In  1         /* surface inside  */
#define BackRound 1   /* Backround color */

typedef double Vector[3];
typedef double Point[3];
typedef double Matrix[4][4];
typedef int    ScrPoint[2];
typedef double Polygon[4][3];
typedef double RGB[3];

#define MaxDepth 5

/****************************************************************************/
/*                                                                          */
/* Module: Global.h                                                         */
/*                                                                          */
/****************************************************************************/
extern Light light[];
extern int   numLights;

/****************************************************************************/
/*                                                                          */
/* Module: Hue.h                                                            */
/*                                                                          */
/****************************************************************************/
typedef struct
          {
           double kDR,    /* Diffuse reflectance coefficient                */
                  kDT,    /*   ''    transmissive    ''                     */
                  fDR[3], /* Diffuse reflectance curve for object           */
                  tT,     /* Transmissitivity per unit length of object     */
                  kRH,    /* specular reflection  highlight coefficient     */
                  kTH,    /*    ''    transmission    ''          ''        */
                  indx;   /* Refractive index of object                     */
          }Hue,*HuePt;


/****************************************************************************/
/*                                                                          */
/* Module: Lights                                                           */
/*                                                                          */
/****************************************************************************/
typedef struct
          {
           Point pt;
           RGB   intensity;     /* rgb intensity */
          }Light,*LightPt;

/****************************************************************************/
/*                                                                          */
/* Module: Map.h                                                            */
/*                                                                          */
/****************************************************************************/
typedef struct
          {
           int uScale,  /* The number of uv entries in the mapping. */
               vScale;
          }MapHeader,*MapHeaderPt;

/****************************************************************************/
/*                                                                          */
/* Module: Obj.h                                                            */
/*                                                                          */
/****************************************************************************/
#include "Hue.h"

typedef struct
          {
           char opCode[4];
           int  l,
                r;
          }Obj,*ObjPt;

typedef struct
          {
           int numEntries,
               boundVol;
           Hue hue;
          }ObjHeader,*ObjHeaderPt;
```

```
/*************************************************************************/
/*                                                                       */
/* Module: Paths.h                                                       */
/*                                                                       */
/*************************************************************************/
static char *GraphFile    = "d:\\c\\thesis\\graph_.scr",
            *BGIFile      = "c:\\bgi",
            *PrimPath     = "d:\\c\\thesis\\prim\\",
            *BoundPath    = "d:\\c\\thesis\\bound\\",
            *PrimFile     = "d:\\c\\thesis\\prim\\PrimFile",
            *PrimTxtPath  = "d:\\c\\thesis\\prim\\A",
            *ObjPath      = "d:\\c\\thesis\\obj\\",
            *ObjCopyPath  = "d:\\c\\thesis\\objCopy\\",
            *ObjTxtPath   = "d:\\c\\thesis\\obj\\O",
            *MapPath      = "d:\\c\\thesis\\map\\",
            *MapFile      = "d:\\c\\thesis\\map\\Map",
            *ObjListFile  = "d:\\c\\thesis\\objList",
            *LightTxtFile = "d:\\c\\thesis\\light\\light.txt";


/*************************************************************************/
/*                                                                       */
/* Module: Prim.h                                                        */
/*                                                                       */
/*************************************************************************/
typedef struct
         {
          char   type;                    /* O,S,C or T */
          Matrix transform,               /* transform matrix */
                 inverse;                 /* inverse of transform */
          int    map[MaxSurface][2],
                 priority;                /* Display higher priority primatives */
          double indx;                    /* refractive index. */
         }Prim, *PrimPt;

/*************************************************************************/
/*                                                                       */
/* Module: Ray.h                                                         */
/*                                                                       */
/*************************************************************************/
typedef struct
         {
          Point  xO;   /* Origin    */
          Vector xD;   /* Direction */
         }Ray, *RayPt;

/*************************************************************************/
/*                                                                       */
/* Module: TList.h                                                       */
/*                                                                       */
/*************************************************************************/
typedef struct A
         {
          double   val;
          int      primNum,
                   surfType;
          Vector   norm;
          struct A *next;
         }TList, *TListPt;
```

```
/****************************************************************************/
/*                                                                          */
/* Module: Bound                                                            */
/*          Various bounding volume functions.                              */
/*                                                                          */
/****************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "paths.h"
#include "define.h"
#include "ray.h"
#include "obj.h"
#include "prim.h"

extern void     *Malloc(size_t size),
                Free(void *block),
                Error(char *msg),
                Inverse(Matrix c,Matrix b),
                ReadPrim(int num,PrimPt prim),
                WritePrim(int num,PrimPt prim),
                Transform(PrimPt prim,Vector s,Vector t,Point pivot,
                        double rX,double rY,double rZ),
                TransformRay(PrimPt prim,RayPt ray,RayPt newRay);
extern double Sqr(double num);
extern FILE     *OpenObj(int num);
extern int      IntersectPoly(int numPoints,double uVPoint[2],
                                double uVPairs[4][2]),
                IntersectRayPlane(RayPt ray,Point plane,Point pt,double *t),
                NewPrimNum(void);

int IntersectObjBound(RayPt ray,int objNum),
     AddBound(ObjPt obj),
     BoundSphere(RayPt ray),
     BoundCube(RayPt ray),
     BoundPyramid(RayPt ray),
     BoundCylinder(RayPt ray),
     BoundCone(RayPt ray);


int IntersectObjBound(ray,objNum)
RayPt ray;
int   objNum;
/****************************************************************************/
/*                                                                          */
/* Test for intersection of an bounding volume.                             */
/* Return 1 if an intersection occures, otherwise 0.                        */
/*                                                                          */
/****************************************************************************/
 {
  FILE         *objFile;
  ObjPt        obj;
  ObjHeaderPt header;
  PrimPt       prim;
  int          intersect = 0;
  RayPt        newRay;
  /**/
  header  = (ObjHeaderPt)Malloc(sizeof(ObjHeader));
  objFile = OpenObj(objNum);
  fread(header,sizeof(ObjHeader),1,objFile);
  if(header->boundVol == -1) /* No bounding volume */
   {
    intersect = 1;
   }
  else
   {
    obj    = (ObjPt)Malloc(sizeof(Obj));
    prim   = (PrimPt)Malloc(sizeof(Prim));
    newRay = (RayPt)Malloc(sizeof(Ray));
    ReadPrim(header->boundVol,prim);
    TransformRay(prim,ray,newRay);
    switch(prim->type)
     {
      case 'S':
       {
        intersect = BoundSphere(newRay);
        break;
       }
      case 'B':
       {
        intersect = BoundCube(newRay);
        break;
       }
      case 'Y':
       {
        intersect = BoundPyramid(newRay);
        break;
       }
      case 'C':
       {
        intersect = BoundCylinder(newRay);
        break;
       }
      case 'N':
       {
        intersect = BoundCone(newRay);
        break;
       }
      default:
       {
        Error("switch case fell to default. Module IntersectPrim. Program Intersect");
        break;
       }
     }
    Free(prim);
    Free(newRay);
    Free(obj);
   }
  Free(header);
  fclose(objFile);
  return(intersect);
 }
```

```
int AddBound(obj)
ObjPt obj;
/*****************************************************************************/
/*                                                                           */
/* Add a new bounding volume to the system.                                  */
/* obj->opcode holds file stem.                                              */
/* obj->l holds number                                                       */
/*                                                                           */
/*****************************************************************************/
 {
  FILE   *boundTextFile;
  PrimPt prim;
  char   tmpStr[40],
         tmpStr1[40];
  Vector s,       /* scale     */
         t;       /* translate */
  Point  pivot;  /* pivot about which we rotate */
  double rX,
         rY,
         rZ;
  int    primNum,
         i,
         j;
  /**/
  prim = (PrimPt)Malloc(sizeof(Prim));
  strcpy(tmpStr,BoundPath);
  strcat(tmpStr,obj->opCode);
  itoa(obj->l,tmpStr1,10);
  strcat(tmpStr,tmpStr1);
  strcat(tmpStr,".txt");
  boundTextFile = fopen(tmpStr,"r");
  fseek(boundTextFile,0L,SEEK_SET);
  fscanf(boundTextFile,"%c",&prim->type);
  fscanf(boundTextFile,"%lf%lf%lf",&s[X],&s[Y],&s[Z]);
  fscanf(boundTextFile,"%lf%lf%lf",&t[X],&t[Y],&t[Z]);
  fscanf(boundTextFile,"%lf%lf%lf",&rX,&rY,&rZ);
  fscanf(boundTextFile,"%lf%lf%lf",&pivot[X],&pivot[Y],&pivot[Z]);
  fclose(boundTextFile);
  for(i = 0;i < 4;i++)          /* Copy Identity matrix into prim->transform */
   {
    for(j = 0;j < 4;j++)
     {
      if(i != j)
       {
        prim->transform[i][j] = 0.0;
       }
      else
       {
        prim->transform[i][j] = 1.0;
       }
     }
   }
  Transform(prim,s,t,pivot,rX,rY,rZ);
  Inverse(prim->transform,prim->inverse);
  primNum = NewPrimNum();
  WritePrim(primNum,prim);

  obj->l = primNum;             /* assign primative number to primative. */
  obj->opCode[0] = 'P';
  Free(prim);
  return(obj->l);
 }


int BoundSphere(ray)
RayPt ray;
/*****************************************************************************/
/*                                                                           */
/* Test for intersection between a ray and a spherical bounding volume.      */
/* Return 1 if an intersection occures, else 0.                              */
/*                                                                           */
/*****************************************************************************/
 {
  double a,
         b,
         c,
         tmp,
         tIn,
         tOut;
  int    intersects;
  /**/
  a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y]) + Sqr(ray->xD[Z]);
  b = ray->xO[X] * ray->xD[X] + ray->xO[Y] * ray->xD[Y] + ray->xO[Z] * ray->xD[Z];
  c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) + Sqr(ray->xO[Z]) - 1.0;
  if((tmp = Sqr(b) - (a * c)) > 0.0)   /* two intersections occur */
   {
    tmp = sqrt(tmp);
    tIn  = (-b + tmp) / a;
    tOut = (-b - tmp) / a;
    if((tIn > 0.0) && (tOut > 0.0))
     {
      intersects = 1;
     }
    else
     {
      intersects = 0;
     }
   }
  else
   {
    intersects = 0;
   }
  return(intersects);
 }
```

```
int BoundCube(ray)
RayPt ray;
/***************************************************************************/
/*                                                                         */
/* Test for intersection between a ray and a cube bounding volume.        */
/* Return 1 if an intersection occures, else 0.                           */
/*                                                                         */
/***************************************************************************/
 {
  double x,
         y,
         z,
         t;
  /**/
  if(ray->xD[X] != 0.0)
   {
    if((t = -ray->xO[X] / ray->xD[X]) > 0.0)
     {
      y = ray->xO[Y] + t * ray->xD[Y];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((y >= 0.0) && (y <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        return(1);
       }
     }
    if((t = (1.0 - ray->xO[X]) / ray->xD[X]) > 0.0)
     {
      y = ray->xO[Y] + t * ray->xD[Y];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((y >= 0.0) && (y <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        return(1);
       }
     }
   }

  if(ray->xD[Y] != 0.0)
   {
    if((t = -ray->xO[Y] / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        return(1);
       }
     }
    if((t = (1.0 - ray->xO[Y]) / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        return(1);
       }
     }
   }


  if(ray->xD[Z] != 0.0)
   {
    if((t = -ray->xO[Z] / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((x >= 0.0) && (x <= 1.0) && (y >= 0.0) && (y <= 1.0))
       {
        return(1);
       }
     }
    if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((x >= 0.0) && (x <= 1.0) && (y >= 0.0) && (y <= 1.0))
       {
        return(1);
       }
     }
   }
  return(0);  /* NO intersection */
 }


int BoundPyramid(ray)
RayPt ray;
/***************************************************************************/
/*                                                                         */
/* Test for intersection between a ray and a pyramid volume.              */
/* Return 1 if an intersection occures, else 0.                           */
/*                                                                         */
/***************************************************************************/
 {
  double t,
         x,
         z;
  Point  plane,
         pt;
  double uVPoint[2],
         uVPoly[4][2];
  /**/

  /* base */
  if(ray->xD[Y] != 0.0)
   {
    if((t = -ray->xO[Y] / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        return(1);
       }
     }
   }
```

```
/* front */
plane[X] = 0.0;  /* definition of plane */
plane[Y] = 0.44721359499958;
plane[Z] = -0.894427190999916;
plane[W] = 0.0;
if(IntersectRayPlane(ray,plane,pt,&t))
 {
  uVPoly[0][0] = 0.5;     /* Drop Z values */
  uVPoly[0][1] = 1.0;
  uVPoly[1][0] = 0.0;
  uVPoly[1][1] = 0.0;
  uVPoly[2][0] = 1.0;
  uVPoly[2][1] = 0.0;
  uVPoint[0] = pt[X];
  uVPoint[1] = pt[Y];
  if(IntersectPoly(3,uVPoint,uVPoly))
   {
    return(1);
   }
 }


/* right */
plane[X] = 0.894427190999916;
plane[Y] = 0.44721359499958;
plane[Z] = 0.0;
plane[W] = -0.894427190999916;
if(IntersectRayPlane(ray,plane,pt,&t))
 {
  uVPoly[0][0] = 1.0;     /* Drop X values */
  uVPoly[0][1] = 0.5;
  uVPoly[1][0] = 0.0;
  uVPoly[1][1] = 0.0;
  uVPoly[2][0] = 0.0;
  uVPoly[2][1] = 1.0;
  uVPoint[0] = pt[Y];
  uVPoint[1] = pt[Z];
  if(IntersectPoly(3,uVPoint,uVPoly))
   {
    return(1);
   }
 }

/* back */
plane[X] = 0.0;  /* definition of plane */
plane[Y] = 0.44721359499958;
plane[Z] = 0.894427190999916;
plane[W] = -0.894427190999916;
if(IntersectRayPlane(ray,plane,pt,&t))
 {
  uVPoly[0][0] = 0.5;     /* Drop Z values */
  uVPoly[0][1] = 1.0;
  uVPoly[1][0] = 1.0;
  uVPoly[1][1] = 0.0;
  uVPoly[2][0] = 0.0;
  uVPoly[2][1] = 0.0;
  uVPoint[0] = pt[X];
  uVPoint[1] = pt[Y];
  if(IntersectPoly(3,uVPoint,uVPoly))
   {
    return(1);
   }
 }

/* left */
plane[X] =  0.894427190999916;
plane[Y] = -0.44721359499958;
plane[Z] =  0.0;
plane[W] =  0.0;
if(IntersectRayPlane(ray,plane,pt,&t))
 {
  uVPoly[0][0] = 1.0;     /* Drop X values */
  uVPoly[0][1] = 0.5;
  uVPoly[1][0] = 0.0;
  uVPoly[1][1] = 1.0;
  uVPoly[2][0] = 0.0;
  uVPoly[2][1] = 0.0;
  uVPoint[0] = pt[Y];
  uVPoint[1] = pt[Z];
  if(IntersectPoly(3,uVPoint,uVPoly))
   {
    return(1);
   }
 }
return(0);
}



int BoundCylinder(ray)
RayPt ray;
/*****************************************************************************/
/*                                                                           */
/* Test for intersection between a ray and a cylindrical volume.            */
/* Return 1 if an intersection occures, else 0.                              */
/*                                                                           */
/*****************************************************************************/
 {
 double x,
        y,
        z,
        t,
        a,
        b,
        c,
        d;
 /**/
 if(ray->xD[Z] != 0.0)
  {
   if((t = -ray->xO[Z] / ray->xD[Z]) > 0.0)
    {
     x = ray->xO[X] + t * ray->xD[X];
     y = ray->xO[Y] + t * ray->xD[Y];
     if((Sqr(x) + Sqr(y)) <= 1.0)
      {
       return(1);
      }
```

```
    }
   if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
    {
     x = ray->xO[X] + t * ray->xD[X];
     y = ray->xO[Y] + t * ray->xD[Y];
     if((Sqr(x) + Sqr(y)) <= 1.0)
      {
       return(1);
      }
    }
  }
 if((a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y])) != 0.0)
  {
   b = ray->xO[X] * ray->xD[X] + ray->xO[Y] * ray->xD[Y];
   c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) - 1.0;
   if((d = Sqr(b) - (a * c)) > 0.0)
    {
     if((t = (-b + (d = sqrt(d))) / a) > 0.0)
      {
       z = ray->xO[Z] + t * ray->xD[Z];
       if((z <= 1.0) && (z >= 0.0))
        {
         return(1);
        }
      }
     if((t = (-b - d) / a) > 0.0)
      {
       z = ray->xO[Z] + t * ray->xD[Z];
       if((z <= 1.0) && (z >= 0.0))
        {
         return(1);
        }
      }
    }
  }
 return(0);
}


int BoundCone(ray)
RayPt ray;
/*************************************************************************/
/*                                                                       */
/* Test for intersection between a ray and a cone volume.               */
/* Return 1 if an intersection occures, else 0.                          */
/*                                                                       */
/*************************************************************************/
 {
  double x,
         y,
         z,
         t,
         a,
         b,
         c,
         d;
  /**/
  if(ray->xD[Z] != 0.0)
   {
    if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((Sqr(x) + Sqr(y)) <= 1.0)
       {
        return(1);
       }
     }
   }
  if((a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y]) - Sqr(ray->xD[Z])) != 0.0)
   {
    b = ray->xO[X] * ray->xD[X] +
        ray->xO[Y] * ray->xD[Y] -
        ray->xO[Z] * ray->xD[Z];
    c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) - Sqr(ray->xO[Z]);
    if((d = Sqr(b) - (a * c)) > 0.0)
     {
      if((t = (-b + (d = sqrt(d))) / a) > 0.0)
       {
        z = ray->xO[Z] + t * ray->xD[Z];
        if((z <= 1.0) && (z >= 0.0))
         {
          return(1);
         }
       }
      if((t = (-b - d) / a) > 0.0)
       {
        z = ray->xO[Z] + t * ray->xD[Z];
        if((z <= 1.0) && (z >= 0.0))
         {
          return(1);
         }
       }
     }
   }
  return(0);
 }
```

```
/*************************************************************************/
/*                                                                       */
/* Module: Coords (Coordinates)                                          */
/*         Various view port and world coordinate setting functions.     */
/*                                                                       */
/*************************************************************************/
#include <graphics.h>
#include "define.h"


int     SetScrCoordsX(double normalisedX),
        SetScrCoordsY(double normalisedY);

void    AssignScrPts(ScrPoint uLScr,ScrPoint lRScr,
                     double xMin,double yMin,double xMax,double yMax),
        AssignWPts(Point wLPt,Point wRPt);
double WorldStepSize(double x0,double x1,int numPixels);


double WLX      = 0.0,      /* (x,y,z) of top left and top right corners of */
       WLY      = 1000.0,   /* world. Because world is a cube, all other    */
       WLZ      = 0.0,      /* corners can be derived from these points.    */
       WRX      = 1000.0,
       WRY      = 1000.0,
       WRZ      = 0.0;


int SetScrCoordsX(normalisedX)
double normalisedX;  /* Normalised position in range (0..1) for x. */
/*************************************************************************/
/*                                                                       */
/* Calculate the screen pixel position of 'x' corresponding to the      */
/* normalised position given as input.                                   */
/*                                                                       */
/*************************************************************************/
 {
  return((int)(normalisedX * (double)(getmaxx() + 1)));
 }


int SetScrCoordsY(normalisedY)
double normalisedY;  /* Normalised position in range (0..1) for y. */
/*************************************************************************/
/*                                                                       */
/* Calculate the screen pixel position of 'y' corresponding to the      */
/* normalised position given as input.                                   */
/*                                                                       */
/*************************************************************************/
 {
  return((int)(normalisedY * (double)(getmaxy() + 1)));
 }

void AssignScrPts(uLScr,lRScr,xMin,yMin,xMax,yMax)
ScrPoint uLScr,
         lRScr;
double   xMin,
         yMin,
         xMax,
         yMax;
/*************************************************************************/
/*                                                                       */
/* Assign the upper left and lower right screen coordinates of the      */
/* screens viewport into the world.                                      */
/*                                                                       */
/*************************************************************************/
 {
  uLScr[X] = SetScrCoordsX(xMin);
  uLScr[Y] = SetScrCoordsY(yMin);
  lRScr[X] = SetScrCoordsX(xMax);
  lRScr[Y] = SetScrCoordsY(yMax);
 }


void AssignWPts(wLPt,wRPt)
Point wLPt,
      wRPt;
/*************************************************************************/
/*                                                                       */
/* Assign the upper left and upper right corner points of the world     */
/* coordinate system.                                                    */
/*                                                                       */
/*************************************************************************/
 {
  wLPt[X] = WLX;
  wLPt[Y] = WLY;
  wLPt[Z] = WLZ;
  wRPt[X] = WRX;
  wRPt[Y] = WRY;
  wRPt[Z] = WRZ;
 }


double WorldStepSize(x0,x1,numPixels)
double x0,
       x1;
int    numPixels;
/*************************************************************************/
/*                                                                       */
/* Given the upper left and upper right coords of the world cube, and the */
/* number of pixels used to represent this distance on the screen,      */
/* calculate the resolution of world units to be steped for each pixel  */
/* step.                                                                 */
/*                                                                       */
/*************************************************************************/
 {
  return((x1 - x0) / (double)(numPixels));
 }
```

```
/*****************************************************************************/
/*                                                                           */
/* Module: CSG (Constructive Solid Geometry)                                 */
/*         Functions needed to implementation recursive CSG tree. These      */
/*         include a control function and functions for each of the three    */
/*         CSG operations; Union Difference and Intersection.                */
/*                                                                           */
/*****************************************************************************/
#include <stdio.h>
#include "define.h"
#include "obj.h"
#include "tList.h"
#include "ray.h"

extern void *Malloc(size_t size),
            Free(void *block),
            Error(char *msg),
            IntersectPrim(RayPt ray,int primNum,TListPt *tList);
            ReadObj(int num,ObjPt obj,FILE *objFile),
            Add(TListPt *pt),
            Kill(TListPt *pt),
            Copy(TListPt *from,TListPt *to),
            AddToTList(TListPt *tList,double t1,double t2,
                       Vector nIn,Vector nOut,int primNum);

extern int  IsPrim(ObjPt obj);


void        CSG(ObjPt obj,TListPt *tList,RayPt ray,FILE *objFile),
            Union(TListPt *tList,TListPt *tList1),
            Intersection(TListPt *tList,TListPt *tList1),
            Difference(TListPt *tList,TListPt *tList1);


extern TListPt tListPos;


void CSG(obj,tList,ray,objFile)
ObjPt   obj;
TListPt *tList;
RayPt   ray;
FILE    *objFile;
/*****************************************************************************/
/*                                                                           */
/* Construct a CSG tree for the real intersections between obj and ray.     */
/* Store the tree in TList.                                                  */
/*                                                                           */
/*****************************************************************************/
 {
  ObjPt   objL,
          objR;
  TListPt newTList = NULL;
  /**/
  while(!IsPrim(obj))
   {
    objL = (ObjPt)Malloc(sizeof(Obj));
    objR = (ObjPt)Malloc(sizeof(Obj));
    ReadObj(obj->l,objL,objFile);
    CSG(objL,tList,ray,objFile);
    if((obj->opCode[0] == '+') || (tList != NULL))
     {
      ReadObj(obj->r,objR,objFile);
      CSG(objR,&newTList,ray,objFile);
      switch(obj->opCode[0])
       {
        case '+' :
         {
          Union(tList,&newTList);
          break;
         }
        case '-' :
         {
          Difference(tList,&newTList);
         }
          break;
        case '&' :
         {
          Intersection(tList,&newTList);
          break;
         }
        default :
         {
          Error("switch case fell to default. Module Roth. Program CSG");
          break;
         }
       }
     }
    Free(objL);
    Free(objR);
    return;
   }
  IntersectPrim(ray,obj->l,tList);
 }



void Union(tList,tList1)
TListPt *tList,
        *tList1;
/*****************************************************************************/
/*                                                                           */
/* Get the tList from the union of tList and tList1. Place the resultant    */
/* tList in tList. tList1 is unchanged.                                      */
/*                                                                           */
/*****************************************************************************/
 {
  TListPt tmpPtDummy = NULL;
  /**/

  tmpPtDummy = (TListPt)Malloc(sizeof(TList));
  tmpPtDummy->next = NULL;
  tListPos = tmpPtDummy;
  while((*tList != NULL) && (*tList1 != NULL))
   {
    if((*tList1)->val < (*tList)->val)
     {
      if((*tList1)->next->val < (*tList)->val)
       {
```

```
          Add(tList1);
          }
         else
          {
           if((*tList1)->next->val > (*tList)->val)
            {
             if((*tList1)->next->val < (*tList)->next->val)
              {
               Copy(&(*tList1)->next,tList);
               Add(tList1);
              }
             else
              {
               if((*tList1)->next->val > (*tList)->next->val)
                {
                 Copy(tList1,tList);
                 Copy(&(*tList)->next,tList1);
                 Add(tList);
                }
               else /* (*tList1)->next->val == (*tList)->next->val */
                {
                 Add(tList1);
                 Kill(tList);
                }
              }
            }
           else  /* (*tList1)->next->val == (*tList)->val */
            {
             Add(tList1);
            }
          }
       }
      else
       {
        if((*tList1)->val > (*tList)->val)
         {
          if((*tList1)->next->val < (*tList)->next->val)
           {
            Add(tList);
            Kill(tList1);
           }
          else
           {
            if((*tList1)->next->val > (*tList)->next->val)
             {
              if((*tList1)->val > (*tList)->next->val)
               {
                Add(tList);
               }
              else
               {
                if((*tList1)->val < (*tList)->next->val)
                 {
                  Copy(&(*tList)->next,tList1);
                  Add(tList);
                 }
                else /* (*tList1)->val == (*tList)->next->val */
                 {
                  Add(tList);
                 }
               }
             }
            else /* (*tList1)->next->val == (*tList)->next->val */
             {
              Add(tList);
              Kill(tList1);
             }
           }
         }
        else  /* (*tList1)->val == (*tList)->val */
         {
          if((*tList1)->next->val < (*tList)->next->val)
           {
            Copy(&(*tList1)->next,tList);
            Add(tList1);
           }
          else
           {
            if((*tList1)->next->val > (*tList)->next->val)
             {
              Copy(&(*tList)->next,tList1);
              Add(tList);
             }
            else /* (*tList1)->next->val == (*tList)->next->val */
             {
              Add(tList);
              Kill(tList1);
             }
           }
         }
       }
    }
   while(*tList != NULL)
    {
     Add(tList);
    }
   while(*tList1 != NULL)
    {
     Add(tList1);
    }
   *tList = tmpPtDummy->next;
   Free(tmpPtDummy);
  }
```

```
void Difference(tList,tList1)
TListPt *tList,
        *tList1;
/***************************************************************************/
/*                                                                         */
/* Get the difference of tList nad tList1. Return this as tList.          */
/* tList1 is deleted.                                                      */
/*                                                                         */
/***************************************************************************/
 {
  int     tmpPrimNum;
  double  tmpVal;
  TListPt tmpPtDummy = NULL;
  /**/
  tmpPtDummy = (TListPt)Malloc(sizeof(TList));
  tmpPtDummy->next = NULL;
  tListPos = tmpPtDummy;

  while((*tList != NULL) && (*tList1 != NULL))
   {
    if((*tList1)->val < (*tList)->val)
     {
      if((*tList1)->next->val < (*tList)->val)
       {
        Kill(tList1);
       }
      else
       {
        if((*tList1)->next->val > (*tList)->val)
         {
          if((*tList1)->next->val < (*tList)->next->val)
           {
            Copy(&(*tList1)->next,tList);
            Kill(tList1);
           }
          else
           {
            if((*tList1)->next->val > (*tList)->next->val)
             {
              Copy(&(*tList)->next,tList1);   /*?*/
              Kill(tList);
             }
            else /* (*tList1)->next->val == (*tList)->next->val */
             {
              Kill(tList);
              Kill(tList1);
             }
           }
         }
        else  /* (*tList1)->next->val == (*tList)->val */
         {
          Kill(tList1);
         }
       }
     }
    else
     {
      if((*tList1)->val > (*tList)->val)
       {
        if((*tList1)->next->val < (*tList)->next->val)
         {
          tmpVal     = (*tList)->val;
          tmpPrimNum = (*tList)->primNum;
          Copy(&(*tList1)->next,tList);
          Copy(tList1,&(*tList1)->next);
          (*tList1)->val = tmpVal;
          (*tList1)->primNum = tmpPrimNum;
          Add(tList1);
         }
        else
         {
          if((*tList1)->next->val > (*tList)->next->val)
           {
            if((*tList1)->val > (*tList)->next->val)
             {
              Add(tList);  /*?*/
             }
            else
             {
              if((*tList1)->val < (*tList)->next->val)
               {
                Copy(tList1,&(*tList)->next);
                Add(tList);
               }
              else /* (*tList1)->val == (*tList)->next->val */
               {
                Add(tList);
               }
             }
           }
          else /* (*tList1)->next->val == (*tList)->next->val */
           {
            Copy(tList1,&(*tList)->next);
            Add(tList);
            Kill(tList1);
           }
         }
       }
      else  /* (*tList1)->val == (*tList)->val */
       {
        if((*tList1)->next->val < (*tList)->next->val)
         {
          Copy(&(*tList1)->next,tList);
          Kill(tList1);
         }
        else
         {
          if((*tList1)->next->val > (*tList)->next->val)
           {
            Copy(&(*tList)->next,tList1);
            Kill(tList);
           }
          else /* (*tList1)->next->val == (*tList)->next->val */
           {
            Kill(tList);
            Kill(tList1);
           }
         }
```

```
        }
      }
    }
  while(*tList != NULL)
   {
    Add(tList);
   }
  while(*tList1 != NULL)
   {
    Kill(tList1);
   }
  *tList = tmpPtDummy->next;
  Free(tmpPtDummy);
 }


void Intersection(tList,tList1)
TListPt *tList,
        *tList1;
/*****************************************************************************/
/*                                                                           */
/* Place the intersection of tList and tList1 in tList. tList1 is deleted.  */
/*                                                                           */
/*****************************************************************************/
 {
  TListPt tmpPtDummy = NULL;
  /**/
  tmpPtDummy = (TListPt)Malloc(sizeof(TList));
  tmpPtDummy->next = NULL;
  tListPos = tmpPtDummy;

  while((*tList != NULL) && (*tList1 != NULL))
   {
    if((*tList1)->val < (*tList)->val)
     {
      if((*tList1)->next->val < (*tList)->val)
       {
        Kill(tList1);
       }
      else
       {
        if((*tList1)->next->val > (*tList)->val)
         {
          if((*tList1)->next->val < (*tList)->next->val)
           {
            Copy(tList,tList1);
            Copy(&(*tList1)->next,tList);
            Add(tList1);
           }
          else
           {
            if((*tList1)->next->val > (*tList)->next->val)
             {
              Copy(&(*tList)->next,tList1);
              Add(tList);
             }
            else /* (*tList1)->next->val == (*tList)->next->val */
             {
              Add(tList);
              Kill(tList1);
             }
           }
         }
        else  /* (*tList1)->next->val == (*tList)->val */
         {
          Kill(tList1);
         }
       }
     }
    else
     {
      if((*tList1)->val > (*tList)->val)
       {
        if((*tList1)->next->val < (*tList)->next->val)
         {
          Copy(&(*tList1)->next,tList);
          Add(tList1);
         }
        else
         {
          if((*tList1)->next->val > (*tList)->next->val)
           {
            if((*tList1)->val > (*tList)->next->val)
             {
              Kill(tList);
             }
            else
             {
              if((*tList1)->val < (*tList)->next->val)
               {
                Copy(tList1,tList);
                Copy(&(*tList)->next,tList1);
                Add(tList);
               }
              else /* (*tList1)->val == (*tList)->next->val */
               {
                Kill(tList);
               }
             }
           }
          else /* (*tList1)->next->val == (*tList)->next->val */
           {
            Add(tList1);
            Kill(tList);
           }
         }
       }
      else  /* (*tList1)->val == (*tList)->val */
       {
        if((*tList1)->next->val < (*tList)->next->val)
         {
          Copy(&(*tList1)->next,tList);
          Add(tList1);
         }
        else
         {
          if((*tList1)->next->val > (*tList)->next->val)
```

```
            {
             Copy(&(*tList)->next,tList1);
             Add(tList);
            }
           else /* (*tList1)->next->val == (*tList)->next->val */
            {
             Add(tList);
             Kill(tList1);
            }
          }
        }
      }
    }
  while(*tList != NULL)
   {
    Kill(tList);
   }
  while(*tList1 != NULL)
   {
    Kill(tList1);
   }
  *tList = tmpPtDummy->next;
  Free(tmpPtDummy);
 }
```

```
/*****************************************************************************/
/*                                                                           */
/* Module: Files                                                             */
/*         Various functions needed to initialise, open and close all        */
/*         global files.                                                     */
/*                                                                           */
/*****************************************************************************/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "define.h"
#include "paths.h"
#include "prim.h"
#include "ray.h"
#include "obj.h"
#include "map.h"

extern FILE *primFile,
            *objListFile;

extern void WriteObj(int num,ObjPt obj,FILE *objFile),
            ReadObj(int num,ObjPt obj,FILE *objFile),
            ReadPrim(int num,PrimPt prim),
            Error(char *msg),
            Inverse(Matrix c,Matrix b),
            Rotate(Matrix m,double rX,double rY,double rZ),
            Scale(Matrix m,Vector s),
            WritePrim(int num,PrimPt prim),
            MatrixCopy(Matrix a,Matrix b),
            Translate(Matrix m,Vector t),
            *Malloc(size_t size),
            Free(void *block);
extern int  CopyPrim(PrimPt srcPrim,int rRRSTFileNum),
            AddBound(ObjPt obj),
            NewPrimNum(void);


void        InitFiles(void),
            OpenFiles(void),
            CloseFiles(void);


void InitFiles(void)
/*****************************************************************************/
/*                                                                           */
/* Initialise global files.                                                  */
/*                                                                           */
/*****************************************************************************/
 {
  FILE *tmpFile;
  tmpFile = fopen(PrimFile,"w+b");
  fclose(tmpFile);
  tmpFile = fopen(ObjListFile,"w+b");
  fclose(tmpFile);
 }


void CloseFiles(void)
/*****************************************************************************/
/*                                                                           */
/* Close all global files.                                                   */
/*                                                                           */
/*****************************************************************************/
 {
  fclose(primFile);
  fclose(objListFile);
 }


void OpenFiles(void)
/*****************************************************************************/
/*                                                                           */
/* Open all global files.                                                    */
/*                                                                           */
/*****************************************************************************/
 {
  primFile = fopen(PrimFile,"r+b");
  objListFile = fopen(ObjListFile,"r+b");
 }
```

```
/***************************************************************************/
/*                                                                         */
/* Module: Global                                                          */
/*         Declare all global variables.                                   */
/*                                                                         */
/***************************************************************************/
#include <stdio.h>
#include "define.h"
#include "light.h"
#include "tList.h"

FILE     *primFile,
         *objListFile;
Light    light[10];
int      numLights;
RGB      iA;
RGB      background;
TListPt tListPos = NULL;
```

```
/***************************************************************************/
/*                                                                         */
/* Module: Graph                                                           */
/*         Various functions for screen graphics.                          */
/*                                                                         */
/***************************************************************************/
#include <graphics.h>
#include <math.h>
#include "define.h"
#include "paths.h"
#include "rgb.h"

int     SetScrCoordsX(double normalisedX),
        SetScrCoordsY(double normalisedY);
int     ColorNum(RGB rgbColor,int graphType);
void    InitGraph(int graphType),
        InitPalette(void),
        CloseGraph(void),
        PutPixel(int x,int y,int color);


void CloseGraph(void)
/***************************************************************************/
/*                                                                         */
/* Close the graphics mode, and restore the crt to normal.                 */
/*                                                                         */
/***************************************************************************/
 {
  closegraph();
  restorecrtmode();
 }


void InitGraph(graphType)
int graphType;  /* graphType == 0 => DETECT  else IBM8514HI */
/***************************************************************************/
/*                                                                         */
/* Initialise the graphics card.                                           */
/*                                                                         */
/***************************************************************************/
 {
  int graphDriver,
      graphMode;
  /**/
  if(graphType == 0)
   {
    graphDriver = DETECT;
   }
  else
   {
    graphDriver = IBM8514;
    graphMode   = IBM8514HI;
   }
  initgraph(&graphDriver,&graphMode,BGIFile);
  if(graphType == 1)
   {
    InitPalette();
   }
 }

void PutPixel(x,y,color)
int x,
    y,
    color;
/***************************************************************************/
/*                                                                         */
/* Color pixel (x,y) on the screen.                                        */
/*                                                                         */
/***************************************************************************/
 {
  putpixel(x,y,color);
 }


void InitPalette(void)
/***************************************************************************/
/*                                                                         */
/* Assign the 256 colors from 256K colors allowed for 'IBM8514' screen.    */
/*                                                                         */
/***************************************************************************/
 {
  int i = 0,
      r,    /* r, g, and b */
      g,
      b;
  /**/
  for(r = 0;r < 256;r += 32)
   {
    for(g = 0;g < 256;g += 32)
     {
      for(b = 0;b < 256;b += 64)
      setrgbpalette(i++,r,g,b);
     }
   }
 setrgbpalette(0,0,0,0);
 setrgbpalette(1,80,178,60);
 setrgbpalette(2,255,0,40);
 setrgbpalette(3,150,239,130);
 setrgbpalette(4,145,25,180);
 setrgbpalette(5,255,255,0);
 setrgbpalette(6,234,209,240);
 setrgbpalette(20,0,100,100);
 setrgbpalette(8,255,255,255);
 setrgbpalette(9,90,95,255);


 setrgbpalette(7,135,135,250);
 }
```

```
int ColorNum(color,graphType)
RGB color;
int graphType;
/***********************************************************************/
/*                                                                     */
/* Return the lookup table color num closest to the given r, g, b     */
/* intensities.                                                        */
/*                                                                     */
/***********************************************************************/
 {
  int r,
      g,
      b;
  if(graphType == 0)  /* VGA Screen */
   {
    r = (int)floor(color[R] * 7);
    g = (int)floor(color[G] * 7);
    b = (int)floor(color[G] * 3);
    r = (r * 32 + g * 4 + b);
    return((r) != 0)?r:1;
   }
  else  /* 8514/A Screen */
   {
    r = (int)floor(color[R] * 7);
    g = (int)floor(color[G] * 7);
    b = (int)floor(color[G] * 3);
    return((r = (r * 32 + g * 4 + b);) != 0)?r:1;
   }
 }
```

```
/**************************************************************************/
/*                                                                        */
/* Module: Hall                                                           */
/*         Various functions needed to compute a Hall shading model.     */
/*                                                                        */
/**************************************************************************/
#include <math.h>
#include "define.h"
#include "light.h"
#include "hue.h"

extern void    MakeVector(Point pt1,Point pt2,Vector v),
               UnitVector(Vector v,Vector unitV);
extern double  VectorDot(Vector v1,Vector v2),
               Cos(double),
               ACos(double x),
               Sqr(double x);

extern int     numLights;
extern Light   light[];
extern RGB     iA;        /* ambient light */

        void   Hj(Vector l,Vector v,Vector hj),
               Hj_(Vector l,Vector v,double indx,Vector hj),
               ProcessColor(HuePt hue,RGB surfColor,RGB iSR,RGB iST,double sT,
                            Point pt,Vector v,Vector n,Vector r,Vector t,
                            RGB color);
        double FSR(double surfColor,double iL,double ang),
               FST(double surfColor,double iL,double ang),
               ProcessWavelength(int lambda,double surfColor,double kDR,
                                 double kDT,double iSR,double iST,double fDR,
                                 double tT,double sT,double kRH,double kTH,
                                 double indx,Point pt,Vector v,Vector n,
                                 Vector r,Vector t);


double ProcessWavelength(lambda,surfColor,kDR,kDT,iSR,iST,fDR,
                         tT,sT,kRH,kTH,indx,pt,v,n,r,t)
int    lambda;  /* Wavelength (ie r, g or b)                              */
double surfColor, /* The intensity of the wavelength at the surface point. */
       kDR,     /* Diffuse reflectance coefficient                        */
       kDT,     /*  ''     transmissive    ''                             */
       iSR,     /* Spectrum of the reflected   ray                        */
       iST,     /*   ''      ''  '' transmitted ray                       */
       fDR,     /* Diffuse reflectance curve for object at wavelength lambda */
       tT,      /* Transmissitivity per unit length of object             */
       sT,      /*   ''         ''       '' transmitted ray               */
       kRH,     /* specular reflection   highlight coefficient            */
       kTH,     /*   ''     transmission    ''         ''                 */
       indx;    /* Refractive index of object                             */
Point  pt;      /* Surface intersection point of incident ray             */
Vector v,       /* Unit incident     vector                               */
       n,       /*  ''  normal          ''                                */
       r,       /*  ''  reflected       ''                                */
       t;       /*  ''  transmitted     ''                                */
/**************************************************************************/
/*                                                                        */
/* Apply "Hall Shading Model" to one wavelength.                          */
/*                                                                        */
/* Return the intensity of light at this wavelength.                      */
/*                                                                        */
/**************************************************************************/
 {
  int    i;
  double a = 0.0,
         b = 0.0,
         c = 0.0,
         d = 0.0,
         intensity,
         kSR,
         kST,
         fDT,
         dotHj,
         dotHj_,
         vD;  /* VectorDot(n,l)                                          */
  Vector hj,  /* vector that perfectly reflects light along incident ray */
         hj_, /*                       transmits                         */
         l;   /* vector from current light source to surface intersection pt */
  /**/
  kSR = 1.0 - kDR;
  kST = 1.0 - kDT;
  fDT = 1.0 - fDR;
  for(i = 0;i < numLights;i++)  /* calculate summations */
   {
    MakeVector(light[i].pt,pt,l);
    UnitVector(l,l);
    Hj(l,v,hj);
    Hj_(l,v,indx,hj_);
    dotHj  = VectorDot(n,hj);
    dotHj_ = VectorDot(n,hj_);
    vD = VectorDot(n,l);
    a += light[i].intensity[lambda] *
         FSR(surfColor,light[i].intensity[lambda],ACos(dotHj)) *
         pow(dotHj,kRH);
    b += light[i].intensity[lambda] *
         FST(surfColor,light[i].intensity[lambda],ACos(dotHj_)) *
         pow(dotHj_,kTH);
    c += light[i].intensity[lambda] * vD;
    d += light[i].intensity[lambda] * -vD;
   }
  return((kSR * a) +
         (kST * b) +
         (kDR * fDR * (iA[lambda] + c)) +
         (kDT * fDT * (iA[lambda] + d)) +
         (kSR * iSR * FSR(surfColor,iA[lambda],ACos(VectorDot(n,r)))) +
         (kST * iST * FST(surfColor,iA[lambda],0.0/*ACos(-VectorDot(n,t)) *
         pow(tT,sT)*/))
        );
 }
```

```
double FSR(surfColor,iL,ang)
double surfColor, /* wavelength                              */
       iL,        /* intensity of light source               */
       ang;       /* angle between light and surface normal */
/***************************************************************************/
/*                                                                         */
/* Get the specular reflection given surfColor, light intensity and ang.  */
/*                                                                         */
/***************************************************************************/
 {
  double tmp,
         f,
         f0,    /* fresnel value at 0  degrees */
         f90,   /*                  90 degrees */
         fAng;  /*                  ang        */
  /**/
  f0   = surfColor;
  f90  = 0.0;
  fAng = Cos(ang) * surfColor;

  if((f90 - f0) == 0.0)
   {
    tmp = 0.0;
   }
  else
   {
    tmp = (fAng - f0) / (f90 - f0);
   }
  if(tmp < 0.0)
   {
    f = iL * f0;
   }
  else
   {
    f = iL * (f0 + ((1.0 - f0) * tmp));
   }
  return(f);
 }


double FST(surfColor,iL,ang)
double surfColor,
       iL,
       ang;
/***************************************************************************/
/*                                                                         */
/* Get the specular transmission given surfColor, light intensity and ang. */
/*                                                                         */
/***************************************************************************/
 {
  return(1.0 - FSR(surfColor,iL,ang));
 }


void Hj(l,v,hj)
Vector l,
       v,
       hj;
/***************************************************************************/
/*                                                                         */
/* Calculate the reflected vector hj. hj is the vector which would        */
/* perfectly reflect the incomming light along the incident ray.          */
/*                                                                         */
/***************************************************************************/
 {
  int    i;
  double tmp;
  /**/
  tmp = sqrt(Sqr(l[X] + v[X]) + Sqr(l[Y] + v[Y]) + Sqr(l[Z] + v[Z]));
  for(i = X;i < W;i++)
   {
    hj[i] = (l[i] + v[i]) / tmp;
   }
  UnitVector(hj,hj);
 }

void Hj_(l,v,indx,hj_)
Vector l,
       v;
double indx;
Vector hj_;
/***************************************************************************/
/*                                                                         */
/* Calculate the transmitted vector hj_. hj_ is the vector which would    */
/* perfectly transmit the incomming light along the incident ray.         */
/*                                                                         */
/***************************************************************************/
 {
  int    i;
  double tmp;
  /**/
  tmp = indx - 1.0;
  for(i = X;i < W;i++)
   {
    hj_[i] = (v[i] + indx * l[i]) / tmp;
   }
  UnitVector(hj_,hj_);
 }
```

```
void ProcessColor(hue,surfColor,iSR,iST,sT,pt,v,n,r,t,color)
HuePt  hue;
RGB    surfColor,
       iSR,     /* Spectrum of the reflected   ray                            */
       iST;     /*    ''     ''  '' transmitted ray                           */
double sT;      /* Distance travelled by transmitted ray inside obj           */
Point  pt;      /* Surface intersection point of incident ray                 */
Vector v,       /* Unit incident     vector                                   */
       n,       /*  ''  normal         ''                                     */
       r,       /*  ''  reflected      ''                                     */
       t;       /*  ''  transmitted    ''                                     */
RGB    color;   /* Color spectrum                                             */
/****************************************************************************/
/*                                                                          */
/* Get the color of an incident ray at its point of intersection with an    */
/* object.                                                                  */
/*                                                                          */
/****************************************************************************/
 {
  int i;
  /**/
  for(i = 0; i < 3;i++)
   {
    color[i] = ProcessWavelength(i,surfColor[i],hue->kDR,hue->kDT,iSR[i],
                                 iST[i],hue->fDR[i],hue->tT,sT,hue->kRH,
                                 hue->kTH,hue->indx,pt,v,n,r,t);
    if(color[i] >= 1.0)
     {
      color[i] = 0.999999;
     }
   }
 }
```

```
/***************************************************************************/
/*                                                                         */
/* Module: Intersect (Intersection)                                        */
/*         Various primitave (ie cube, sphere etc.) intersection functions. */
/*                                                                         */
/***************************************************************************/
#include <string.h>
#include <math.h>
#include <stdio.h>
#include "define.h"
#include "prim.h"
#include "objList.h"
#include "ray.h"
#include "obj.h"
#include "TList.h"

extern void    AddObj(char *objData),
               *Malloc(size_t size),
               TransformRay(PrimPt prim,RayPt ray,RayPt newRay),
               CopyObj(char *srcObj,char *destObj,char *copyTxt),
               Error(char *msg),
               Free(void *block),
               CloseFiles(void),
               VectorCross(Vector a,Vector b,Vector cross),
               TListDel(TListPt *tList),
               AddToTList(TListPt *tList,double t1,double t2,Vector nIn,Vector nOut,int primNum),
               Union(TListPt *tList,TListPt *tList1),
               Difference(TListPt *tList,TListPt *tList1),
               Intersect(TListPt *tList,TListPt *tList1),
               InitFiles(void),      /* For test purposes only */
               OpenFiles(void),
               ReadPrim(int num,PrimPt prim),
               ReadObj(int objNum,ObjPt obj,FILE *objFile),
               AddMap(char *textFileName),
               Inverse(Matrix c,Matrix b),
               MatrixCopy(Matrix a,Matrix b),
               VectorCopy(Vector a,Vector b),
               MatrixMul(Matrix a,Matrix b),
               UnitVector(Vector v,Vector xUnit),
               PointMatrixMul(Point p,Matrix m,Point newP),
               VectorMatrixMul(Vector v,Matrix m,Vector newV),
               CloseObj(FILE *objFile);
extern int     BoundCube(RayPt ray),
               BoundSphere(RayPt ray);
               BoundCylinder(RayPt ray);
               BoundCond(RayPt ray);
               BoundPyramid(RayPt ray);
extern double  Sqr(double x),
               Roundoff(double num),
               VectorDot(Vector a,Vector b),
               PointVectorDot(Point a,Vector b),
               PointDot(Point a,Point b),
               Sin(double num),
               Cos(double num),
               ASin(double num),
               ACos(double num);

extern FILE    *OpenObj(int num);

int            Swap(double t,Vector n,double *tIn,double *tOut,Vector nIn,Vector nOut),
               IntersectLineYAxis(double pt1[2],double pt2[2]),
               IntersectPoly(int numPoints,double uVPoint[2],double uVPairs[4][2]),
               IntersectRayPlane(RayPt ray,Point plane,Point pt,double *t),
               IntersectCube(RayPt ray,double *tIn,double *tOut,Vector nIn,Vector nOut),
               IntersectPyramid(RayPt ray,double *tIn,double *tOut,Vector nIn,Vector nOut),
               IntersectCylinder(RayPt ray,double *tIn,double *tOut,Vector nIn,Vector nOut),
               IntersectCone(RayPt ray,double *tIn,double *tOut,Vector nIn,Vector nOut),
               IntersectSphere(RayPt ray,double *tIn,double *tOut,Vector nIn,Vector nOut);
void           IntersectPrim(RayPt ray,int primNum,TListPt *tList);


int IntersectCylinder(ray,tIn,tOut,nIn,nOut)
RayPt  ray;
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/***************************************************************************/
/*                                                                         */
/* Returns 1 if an intersection between 'ray' and the cylinder prim exists */
/* else returns 0.                                                         */
/* If an intersection does occur then it calculates the two intersection   */
/* points ('in' and 'out') between 'ray' and the cylinder prim type.       */
/*                                                                         */
/* NOTE: This function works in object coordinate system. ie the ray must  */
/*       already be translated into the objects local coordinate system.   */
/*                                                                         */
/*       Also 'in' and 'out' are given in the objects coordinate system.   */
/*                                                                         */
/***************************************************************************/
 {
  int    i;
  double x,
         y,
         z,
         t,
         a,
         b,
         c,
         d;
  Vector n0;
  /**/
  *tIn  = 0.0;
  *tOut = 0.0;
  if(ray->xD[Z] != 0.0)
   {
    if((t = -ray->xO[Z] / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((Sqr(x) + Sqr(y)) <= 1.0)
       {
        n0[X] =  0.0;
        n0[Y] =  0.0;
        n0[Z] = -1.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
```

```
          return(1);
          }
        }
      }
    if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
      {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((Sqr(x) + Sqr(y)) <= 1.0)
        {
        n0[X] =  0.0;
        n0[Y] =  0.0;
        n0[Z] =  1.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
          {
          return(1);
          }
        }
      }
    }
  if((a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y])) != 0.0)
    {
    b = ray->xO[X] * ray->xD[X] + ray->xO[Y] * ray->xD[Y];
    c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) - 1.0;
    if((d = Sqr(b) - (a * c)) > 0.0)
      {
      if((t = (-b + (d = sqrt(d))) / a) > 0.0)
        {
        z = ray->xO[Z] + t * ray->xD[Z];
        if((z <= 1.0) && (z >= 0.0))
          {
          for(i = X;i < Z;i++)
            {
            n0[i] = ray->xO[i] + t * ray->xD[i];
            }
          n0[Z] = 0.0;
          if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
            {
            return(1);
            }
          }
        }
      if((t = (-b - d) / a) > 0.0)
        {
        z = ray->xO[Z] + t * ray->xD[Z];
        if((z <= 1.0) && (z >= 0.0))
          {
          for(i = X;i < Z;i++)
            {
            n0[i] = ray->xO[i] + t * ray->xD[i];
            }
          n0[Z] = 0.0;
          if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
            {
            return(1);
            }
          }
        }
      }
    }
  return(0);
  }


int IntersectCone(ray,tIn,tOut,nIn,nOut)
RayPt  ray;
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/**************************************************************************/
/*                                                                        */
/* Returns 1 if an intersection between 'ray' and the cone prim exists    */
/* else returns 0.                                                        */
/* If an intersection does occur then it calculates the two intersection  */
/* points ('in' and 'out') between 'ray' and the cylinder prim type.      */
/*                                                                        */
/* NOTE: This function works in object coordinate system. ie the ray must */
/*       already be translated into the objects local coordinate system.  */
/*                                                                        */
/*       Also 'in' and 'out' are given in the objects coordinate system.  */
/*                                                                        */
/**************************************************************************/
  {
  Vector n0;
  double x,
         y,
         z,
         t,
         a,
         b,
         c,
         d;
  /**/
  *tIn  = 0.0;
  *tOut = 0.0;
  if(ray->xD[Z] != 0.0)
    {
    if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
      {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((Sqr(x) + Sqr(y)) <= 1.0)
        {
        n0[X] = 0.0;
        n0[Y] = 0.0;
        n0[Z] = 1.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
          {
          return(1);
          }
        }
      }
    }
  if((a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y]) - Sqr(ray->xD[Z])) != 0.0)
    {
    b = ray->xO[X] * ray->xD[X] +
        ray->xO[Y] * ray->xD[Y] -
```

```
        ray->xO[Z] * ray->xD[Z];
     c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) - Sqr(ray->xO[Z]);
     if((d = Sqr(b) - (a * c)) > 0.0)
      {
       if((t = (-b + (d = sqrt(d))) / a) > 0.0)
        {
         z = ray->xO[Z] + t * ray->xD[Z];
         if((z <= 1.0) && (z >= 0.0))
          {
           x = ray->xO[X] + t * ray->xD[X];
           y = ray->xO[Y] + t * ray->xD[Y];
           n0[X] =  x;
           n0[Y] =  y;
           n0[Z] = -z;
           if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
            {
             return(1);
            }
          }
        }
       if((t = (-b - d) / a) > 0.0)
        {
         z = ray->xO[Z] + t * ray->xD[Z];
         if((z <= 1.0) && (z >= 0.0))
          {
           x = ray->xO[X] + t * ray->xD[X];
           y = ray->xO[Y] + t * ray->xD[Y];
           if((z <= 1.0) && (z >= 0.0))
            {
             n0[X] =  x;
             n0[Y] =  y;
             n0[Z] = -z;
             if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
              {
               return(1);
              }
            }
          }
        }
      }
   return(0);
  }


void IntersectPrim(ray,primNum,tList)
RayPt   ray;
int     primNum;
TListPt *tList;
/*****************************************************************************/
/*                                                                           */
/* Returns 1 if an intersection between ray and prim exists, else returns    */
/* 0.                                                                        */
/* If an intersection does exist then the distance along 'ray' where         */
/* 'prim' is intersected is placed in 'tIn' and 'tOut'.                      */
/*                                                                           */
/* If no intersection occurs then in and out are undefined.                  */
/*                                                                           */
/* NOTE: ray is given in Local Coordinates;                                  */
/*       tIn and tOut are returned in World Coordinates.                     */
/*                                                                           */
/*****************************************************************************/
 {
  double tIn,
         tOut;
  int    intersect;
  PrimPt prim;
  RayPt  newRay;
  Vector nIn,
         nOut;
  /**/
  prim   = (PrimPt)Malloc(sizeof(Prim));
  newRay = (RayPt)Malloc(sizeof(Ray));
  ReadPrim(primNum,prim);
  TransformRay(prim,ray,newRay);
  switch(prim->type)
   {
    case 'S':
     {
      intersect = IntersectSphere(newRay,&tIn,&tOut,nIn,nOut);
      break;
     }
    case 'B':
     {
      intersect = IntersectCube(newRay,&tIn,&tOut,nIn,nOut);
      break;
     }
    case 'Y':
     {
      intersect = IntersectPyramid(newRay,&tIn,&tOut,nIn,nOut);
      break;
     }
    case 'C':
     {
      intersect = IntersectCylinder(newRay,&tIn,&tOut,nIn,nOut);
      break;
     }
    case 'N':
     {
      intersect = IntersectCone(newRay,&tIn,&tOut,nIn,nOut);
      break;
     }
    default:
     {
      Error("switch case fell to default. Module IntersectPrim. Program Intersect");
      break;
     }
   }
  if(intersect)
   {
    AddToTList(tList,tIn,tOut,nIn,nOut,primNum);
   }
  Free(prim);
  Free(newRay);
 }
```

```
int IntersectSphere(ray,tIn,tOut,nIn,nOut)
RayPt  ray;     /* Ray in sphere coordinate system */
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/**************************************************************************/
/*                                                                        */
/* Returns 1 if an intersection between 'ray' and the sphere prim exists, */
/* else returns 0.                                                        */
/* If an intersection does occur then it calculates the distance along    */
/* ray of the two intersection points ('tIn' and 'tOut') between 'ray'    */
/* and the sphere prim type.                                              */
/*                                                                        */
/* NOTE: This function works in object coordinate system. ie the ray must */
/*       already be translated into the objects local coordinate system.  */
/*                                                                        */
/*       Also 'tIn' and 'tOut' are given in the objects coordinate system, */
/*       and must later be scaled up to world coordinates.                */
/*                                                                        */
/**************************************************************************/
 {
  double a,
         b,
         c,
         tmp;
  int    intersects,
         i;
  /**/
  a = Sqr(ray->xD[X]) + Sqr(ray->xD[Y]) + Sqr(ray->xD[Z]);
  b = ray->xO[X] * ray->xD[X] + ray->xO[Y] * ray->xD[Y] + ray->xO[Z] * ray->xD[Z];
  c = Sqr(ray->xO[X]) + Sqr(ray->xO[Y]) + Sqr(ray->xO[Z]) - 1.0;
  if((tmp = Sqr(b) - (a * c)) > 0.0)   /* two intersections occur */
   {
    tmp = sqrt(tmp);
    *tIn  = (-b + tmp) / a;
    *tOut = (-b - tmp) / a;
    if((*tIn > 0.0) && (*tOut > 0.0))
     {
      if(*tOut < *tIn)
       {
        tmp   = *tOut;     /* swap tIn and tOut */
        *tOut = *tIn;
        *tIn  = tmp;
       }
      for(i = X;i < W;i++)
       {
        nIn[i] = ray->xO[i] + *tIn * ray->xD[i];
        nOut[i] = -(ray->xO[i] + *tOut * ray->xD[i]);
       }
      intersects = 1;
     }
    else
     {
      intersects = 0;
     }
   }
  else
   {
    intersects = 0;
   }
  return(intersects);
 }


int IntersectCube(ray,tIn,tOut,nIn,nOut)
RayPt ray;
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/**************************************************************************/
/*                                                                        */
/* Returns 1 if an intersection between 'ray' and the cube prim exists,   */
/* else returns 0.                                                        */
/* If an intersection does occur then it calculates the two intersection  */
/* points ('tIn' and 'tOut') between 'ray' and the cube prim type.        */
/*                                                                        */
/* NOTE: This function works in object coordinate system. ie the ray must */
/*       already be translated into the objects local coordinate system.  */
/*                                                                        */
/*       Also 'in' and 'out' are given in the objects coordinate system.  */
/*                                                                        */
/**************************************************************************/
 {
  double x,
         y,
         z,
         t;
  Vector n0;
  /**/
  *tIn  = 0.0;
  *tOut = 0.0;
  if(ray->xD[X] != 0.0)
   {
    if((t = -ray->xO[X] / ray->xD[X]) > 0.0)
     {
      y = ray->xO[Y] + t * ray->xD[Y];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((y >= 0.0) && (y <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        n0[X] = -1.0;
        n0[Y] =  0.0;
        n0[Z] =  0.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
    if((t = (1.0 - ray->xO[X]) / ray->xD[X]) > 0.0)
     {
      y = ray->xO[Y] + t * ray->xD[Y];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((y >= 0.0) && (y <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        n0[X] =  1.0;
```

```
        n0[Y] =  0.0;
        n0[Z] =  0.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
   }

  if(ray->xD[Y] != 0.0)
   {
    if((t = -ray->xO[Y] / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        n0[X] =  0.0;
        n0[Y] = -1.0;
        n0[Z] =  0.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
    if((t = (1.0 - ray->xO[Y]) / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        n0[X] =  0.0;
        n0[Y] =  1.0;
        n0[Z] =  0.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
   }


  if(ray->xD[Z] != 0.0)
   {
    if((t = -ray->xO[Z] / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((x >= 0.0) && (x <= 1.0) && (y >= 0.0) && (y <= 1.0))
       {
        n0[X] =  0.0;
        n0[Y] =  0.0;
        n0[Z] = -1.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
    if((t = (1.0 - ray->xO[Z]) / ray->xD[Z]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      y = ray->xO[Y] + t * ray->xD[Y];
      if((x >= 0.0) && (x <= 1.0) && (y >= 0.0) && (y <= 1.0))
       {
        n0[X] =  0.0;
        n0[Y] =  0.0;
        n0[Z] =  1.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
   }
  return(0);  /* NO intersection */
 }



int Swap(t,n0,tIn,tOut,nIn,nOut)
double t;
Vector n0;
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/*************************************************************************/
/*                                                                       */
/* If a value has not yet been placed in 'tIn' then set up 'tIn' options.  */
/* Return 0, else:                                                       */
/*                                                                       */
/* Place proper values in 'tIn' and 'tOut'. Return 1.                    */
/*                                                                       */
/*************************************************************************/
 {
  Vector n1;
  int    i;
  /**/
  for(i = X;i < W;i++)
   {
    n1[i] = -n0[i];
   }
  if(*tIn == 0.0)
   {
    *tIn  = t;
    VectorCopy(n0,nIn);
    VectorCopy(n1,nOut);
    return(0);
   }
  else
   {
    if(t < *tIn)
     {
```

```
      *tOut = *tIn;
      *tIn  = t;
      VectorCopy(nIn,nOut);
      VectorCopy(n0,nIn);
     }
    else /* t > *tIn */
     {
      *tOut = t;
      VectorCopy(n1,nOut);
     }
    return(1);
   }
 }


int IntersectRayPlane(ray,plane,pt,t)
RayPt  ray;
Point  plane,
       pt;
double *t;
/**************************************************************************/
/*                                                                        */
/* Get the 't' value of the intersection between ray and plane.          */
/* If such an intersection exists, return 1, else return 0.              */
/*                                                                        */
/**************************************************************************/
 {
  double vD,
         vO;
  int    i;
  /**/
  if((vD = Roundoff(PointVectorDot(plane,ray->xD))) == 0.0) /* ray parallel to plane */
   {
    return(0);
   }
  else
   {
    vO = -(Roundoff(PointDot(plane,ray->xO)) + plane[3]);
    if((*t = vO / vD) > 0.0)   /* Find point of intersection */
     {
      for(i = X;i < W;i++)
       {
        pt[i] = ray->xO[i] + *t * ray->xD[i];
       }
      return(1);
     }
    else /* Intersection occurs behind ray */
     {
      return(0);
     }
   }
 }


int IntersectPyramid(ray,tIn,tOut,nIn,nOut)
RayPt ray;
double *tIn,
       *tOut;
Vector nIn,
       nOut;
/**************************************************************************/
/*                                                                        */
/* Returns 1 if an intersection between 'ray' and the pyramid prim exists, */
/* else returns 0.                                                        */
/* If an intersection does occur then it calculates the two intersection  */
/* points ('tIn' and 'tOut') between 'ray' and the pyramid prim type.     */
/*                                                                        */
/* NOTE: This function works in object coordinate system. ie the ray must */
/*       already be translated into the objects local coordinate system.  */
/*                                                                        */
/*       Also 'in' and 'out' are given in the objects coordinate system.  */
/*                                                                        */
/**************************************************************************/
 {
  double t,
         x,
         z;
  Point  plane,
         pt;
  double uVPoint[2],
         uVPoly[4][2];
  Vector n0;
  /**/
  *tIn  = 0.0;
  *tOut = 0.0;


  /* base */
  if(ray->xD[Y] != 0.0)
   {
    if((t = -ray->xO[Y] / ray->xD[Y]) > 0.0)
     {
      x = ray->xO[X] + t * ray->xD[X];
      z = ray->xO[Z] + t * ray->xD[Z];
      if((x >= 0.0) && (x <= 1.0) && (z >= 0.0) && (z <= 1.0))
       {
        n0[X] =  0.0;
        n0[Y] = -1.0;
        n0[Z] =  0.0;
        if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
         {
          return(1);
         }
       }
     }
   }


  /* front */
  plane[X] =  0.0;   /* definition of plane */
  plane[Y] =  0.447213595499958;
  plane[Z] = -0.894427190999916;
  plane[W] =  0.0;
  if(IntersectRayPlane(ray,plane,pt,&t))
   {
    uVPoly[0][U] = 0.5;     /* Drop Z values */
```

```
    uVPoly[0][V] = 1.0;
    uVPoly[1][U] = 0.0;
    uVPoly[1][V] = 0.0;
    uVPoly[2][U] = 1.0;
    uVPoly[2][V] = 0.0;
    uVPoint[U] = pt[X];
    uVPoint[V] = pt[Y];
    if(IntersectPoly(3,uVPoint,uVPoly))
     {
      n0[X] =  0.0;
      n0[Y] =  0.5;
      n0[Z] = -1.0;
      if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
       {
        return(1);
       }
     }
   }

  /* right */
  plane[X] = 0.894427190999916;
  plane[Y] = 0.44721359499958;
  plane[Z] = 0.0;
  plane[W] = -0.894427190999916;
  if(IntersectRayPlane(ray,plane,pt,&t))
   {
    uVPoly[0][U] = 1.0;     /* Drop X values */
    uVPoly[0][V] = 0.5;
    uVPoly[1][U] = 0.0;
    uVPoly[1][V] = 0.0;
    uVPoly[2][U] = 0.0;
    uVPoly[2][V] = 1.0;
    uVPoint[U] = pt[Y];
    uVPoint[V] = pt[Z];
    if(IntersectPoly(3,uVPoint,uVPoly))
     {
      n0[X] = 1.0;
      n0[Y] = 0.5;
      n0[Z] = 0.0;
      if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
       {
        return(1);
       }
     }
   }

  /* back */
  plane[X] = 0.0;  /* definition of plane */
  plane[Y] = 0.44721359499958;
  plane[Z] = 0.894427190999916;
  plane[W] = -0.894427190999916;
  if(IntersectRayPlane(ray,plane,pt,&t))
   {
    uVPoly[0][U] = 0.5;     /* Drop Z values */
    uVPoly[0][V] = 1.0;
    uVPoly[1][U] = 1.0;
    uVPoly[1][V] = 0.0;
    uVPoly[2][U] = 0.0;
    uVPoly[2][V] = 0.0;
    uVPoint[U] = pt[X];
    uVPoint[V] = pt[Y];
    if(IntersectPoly(3,uVPoint,uVPoly))
     {
      n0[X] = 0.0;
      n0[Y] = 0.5;
      n0[Z] = 1.0;
      if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
       {
        return(1);
       }
     }
   }

  /* left */
  plane[X] =  0.894427190999916;
  plane[Y] = -0.44721359499958;
  plane[Z] =  0.0;
  plane[W] =  0.0;
  if(IntersectRayPlane(ray,plane,pt,&t))
   {
    uVPoly[0][U] = 1.0;     /* Drop X values */
    uVPoly[0][V] = 0.5;
    uVPoly[1][U] = 0.0;
    uVPoly[1][V] = 1.0;
    uVPoly[2][U] = 0.0;
    uVPoly[2][V] = 0.0;
    uVPoint[U] = pt[Y];
    uVPoint[V] = pt[Z];
    if(IntersectPoly(3,uVPoint,uVPoly))
     {
      n0[X] = -1.0;
      n0[Y] =  0.5;
      n0[Z] =  0.0;
      if(Swap(t,n0,tIn,tOut,nIn,nOut) == 1)
       {
        return(1);
       }
     }
   }
  return(0);
 }


int IntersectPoly(numPoints,uVPoint,uVPoly)
/*************************************************************************/
/*                                                                       */
/* If uVPoint intersects the polygon defined in uVPoly, then return 1, else */
/* return 0.                                                             */
/*                                                                       */
/*************************************************************************/
int    numPoints;   /* Number of points in the polygon                   */
double uVPoint[2],  /* The intersection point of some ray with a 3D polygon */
       uVPoly[4][2];/* A 2D mapping of the same polygon                  */
 {
  int pt,
      nextPt,
```

```
        numIntersects = 0,
        sH,
        nSH;
  /**/
  for(pt = 0;pt < numPoints;pt++)
   {
    uVPoly[pt][U] -= uVPoint[U];
    uVPoly[pt][V] -= uVPoint[V];
   }
  if(uVPoly[0][V] >= 0.0)
   {
    sH = 1;
   }
  else
   {
    sH = -1;
   }
  for(pt = 0;pt < numPoints;pt++)
   {
    nextPt = (pt + 1) % numPoints;
    if(uVPoly[nextPt][1] >= 0.0)
     {
      nSH = 1;
     }
    else
     {
      nSH = -1;
     }
    if(sH != nSH)
     {
      if((uVPoly[pt][U] >= 0.0) && (uVPoly[nextPt][U] >= 0.0))
       {
        numIntersects++;
       }
      else
       {
        if((uVPoly[pt][U] >= 0.0) || (uVPoly[nextPt][U] >= 0.0))
         {
          numIntersects += IntersectLineYAxis(uVPoly[pt],uVPoly[nextPt]);
         }
       }
     }
    sH = nSH;
   }
  return((int)fmod(numIntersects,2.0));
 }


int IntersectLineYAxis(pt1,pt2)
/*****************************************************************************/
/*                                                                           */
/* If the 2D line defined by pt1..pt2 intersects the +y axis then return 1.  */
/* else return 0.                                                            */
/*                                                                           */
/*****************************************************************************/
double pt1[2],
       pt2[2];
 {
  if(pt2[1] != pt1[1])
   {
    return((pt1[U] - pt1[V] * (pt2[U] - pt1[U]) / (pt2[V] - pt1[V])) > 0.0)?1:0;
   }
  else
   {
    return 0;
   }
 }
```

```
/****************************************************************************/
/*                                                                          */
/* Module: Lighting                                                         */
/*         Functions needed to initiate the various light sources and the   */
/*         background ambiant lighting.                                     */
/*                                                                          */
/****************************************************************************/
#include <stdio.h>
#include "paths.h"
#include "define.h"
#include "light.h"


extern Light light[];
extern int   numLights;
extern RGB   iA;
extern RGB   background;

void         InitLighting(void),
             InitBackground(void);


void InitLighting(void)
/****************************************************************************/
/*                                                                          */
/* Initialise the lighting intersity and position of all the lights in the  */
/* model. Also initialise the ambiant light intensity.                      */
/*                                                                          */
/****************************************************************************/
 {
  FILE *fp;
  int  i;
  /**/
  fp = fopen(LightTxtFile,"r+b");
  fscanf(fp,"%lf%lf%lf",&iA[X],&iA[Y],&iA[Z]);
  fscanf(fp,"%d",&numLights);
  for(i = 0;i < numLights;i++)
   {
    fscanf(fp,"%lf%lf%lf%lf%lf%lf",
        &light[i].pt[X],&light[i].pt[Y],&light[i].pt[Z],
        &light[i].intensity[X],&light[i].intensity[Y],&light[i].intensity[Z]);
   }
 }


void InitBackground()
/****************************************************************************/
/*                                                                          */
/* Initialise the background intensity for r,g and b.                       */
/*                                                                          */
/****************************************************************************/
 {
  background[R] = 0.0;
  background[G] = 0.0;
  background[B] = 0.0;
 }
```

```
/******************************************************************************/
/*                                                                            */
/* Module: MathFunc                                                           */
/*         Various maths functions needed that are not provided by the       */
/*         standard maths library.                                            */
/*                                                                            */
/******************************************************************************/
#include <math.h>


double Sqr(double x),
       Roundoff(double num),
       Sin(double num),
       Cos(double num),
       ASin(double num),
       ACos(double num);


const double Radient = 57.29578;


double Sqr(x)
double x;
/******************************************************************************/
/*                                                                            */
/* Returns x squared.                                                         */
/*                                                                            */
/* NOTE: This function avoids devide by zero error caused by using pow.      */
/*                                                                            */
/******************************************************************************/
 {
  return(x == 0.0)?(0.0):(x * x);
 }


double Roundoff(num)
double num;
/******************************************************************************/
/*                                                                            */
/* Roundoff a real number close to an integer value.                          */
/*                                                                            */
/******************************************************************************/
 {
  double floorNum,
         variance;
  /**/
  if(num > 0.0)
   {
    floorNum = floor(num);
    variance = num - floorNum;
    if(variance > 0.99999999999)
     {
      num = floorNum + 1.0;
     }
    else
     {
      if(variance < 0.00000000001)
       {
        num = floorNum;
       }
     }
   }
  else /* num <= 0.0 */
   {
    if(num < 0.0)
     {
      floorNum = floor(-num);
      variance = -num - floorNum;
      if(variance > 0.999999998)
       {
        num = (floorNum + 1.0) * -1.0;
       }
      else
       {
        if(variance < 0.000000001)
         {
          num = -floorNum;
         }
       }
     }
    else
     {
      num = 0.0;
     }
   }
  return(num);
 }


double Sin(num)
double num;
/******************************************************************************/
/*                                                                            */
/* Return the sin of a number given in degrees.                               */
/*                                                                            */
/******************************************************************************/
 {
  return(sin(num / Radient));
 }


double Cos(num)
double num;
/******************************************************************************/
/*                                                                            */
/* Return the cos of a number given in degrees.                               */
/*                                                                            */
/******************************************************************************/
 {
  return(cos(num / Radient));
 }
```

```
double ASin(num)
double num;
/****************************************************************************/
/*                                                                          */
/* Return the arcsin of a number given in degrees.                          */
/*                                                                          */
/****************************************************************************/
 {
  return(asin(num) * Radient);
 }


double ACos(num)
double num;
/****************************************************************************/
/*                                                                          */
/* Return the arccos of a number given in degrees.                          */
/*                                                                          */
/****************************************************************************/
 {
  return(acos(num) * Radient);
 }
```

```
/**************************************************************************/
/*                                                                        */
/* Module: Matrix                                                         */
/*         Various matrix and vector functions.                           */
/*                                                                        */
/**************************************************************************/
#include <math.h>
#include "define.h"

extern double Sqr(double a);
void          Inverse(Matrix c,Matrix b),
              MatrixCopy(Matrix a,Matrix b),
              VectorCopy(Vector a,Vector b),
              UnitVector(Vector a,Vector n),
              VectorCross(Vector a,Vector b,Vector cross),
              MatrixMul(Matrix a,Matrix b),
              PointMatrixMul(Point p,Matrix m,Point newP),
              VectorMatrixMul(Vector v,Matrix m,Vector newV),
              MakeVector(Point p1,Point p2,Vector v);
double        VectorDot(Vector a,Vector b),
              PointVectorDot(Point a,Vector b),
              PointDot(Point a,Point b),
              Distance(Point pt0,Point pt1);

void Inverse(c,b)
Matrix c,
       b;
/**************************************************************************/
/*                                                                        */
/* Get the inverse matrix of c and place it in b. If no inverse exists,   */
/* then exit the program as an error.                                     */
/*                                                                        */
/**************************************************************************/
 {
  int     i,
          j,
          row,
          found;
  Matrix a;
  double d;
  /**/
  MatrixCopy(c,a);

  for(i = 0;i < 4;i++)
   {
    for(j = 0;j < 4;j++)
     {
      if(i == j)
       {
        b[i][j] = 1.0;
       }
      else
       {
        b[i][j] = 0.0;
       }
     }
   }
  for(row = 0;row < 4;row++)
   {
    if(a[row][row] != 1.0)
     {
      if(a[row][row] != 0.0)
       {
        d = 1.0 / a[row][row];
        for(i = 0;i < 4;i++)
         {
          a[row][i] *= d;
          b[row][i] *= d;
         }
       }
      else
       {
        found = 0;
        i = row + 1;
        while((!found) && (i < 4))
         {
          if(a[i][row] != 0)
           {
            found = 1;
            d = 1.0 / a[i][row];
            for(j = 0;j < 4;j++)
             {
              a[row][j] += d * a[i][j];
              b[row][j] += d * b[i][j];
             }
           }
          i += 1;
         }
       }
     }
    for(i = (row + 1);i < 4;i++)
     {
      if(a[i][row] != 0.0)
       {
        d = -a[i][row];
        for(j = 0;j < 4;j++)
         {
          a[i][j] += d * a[row][j];
          b[i][j] += d * b[row][j];
         }
       }
     }
    for(i = row - 1;i >= 0;i--)
     {
      if(a[i][row] != 0.0)
       {
        d = -a[i][row];
        for(j = 0;j < 4;j++)
         {
          a[i][j] += d * a[row][j];
          b[i][j] += d * b[row][j];
         }
       }
     }
   }
 }
```

```
void VectorCopy(a,b)
Vector a,
       b;
/**************************************************************************/
/*                                                                        */
/* Copy vector a into b.                                                  */
/*                                                                        */
/**************************************************************************/
 {
  int i;
  /**/
  for(i = X;i < W;i++)
   {
    b[i] = a[i];
   }
 }


void MatrixCopy(a,b)
Matrix a,
       b;
/**************************************************************************/
/*                                                                        */
/* Copy matrix a into b.                                                  */
/*                                                                        */
/**************************************************************************/
 {
  int i,
      j;
  /**/
  for(i = 0;i < 4;i++)
   {
    for(j = 0;j < 4;j++)
     {
      b[i][j] = a[i][j];
     }
   }
 }


void MatrixMul(a,b)
/**************************************************************************/
/*                                                                        */
/* Multiply matrix a by b. Store the result in a.                         */
/*                                                                        */
/**************************************************************************/
Matrix a,
       b;
 {
  int    i,
         j,
         k;
  Matrix c;
  /**/
  for(i = 0;i < 4;i++)
   {
    for(j = 0;j < 4;j++)
     {
      c[i][j] = 0.0;
      for(k = 0;k < 4;k++)
       {
        c[i][j] += a[i][k] * b[k][j];
       }
     }
   }
  MatrixCopy(c,a);
 }


void UnitVector(v,xUnit)
Vector v,
       xUnit;
/**************************************************************************/
/*                                                                        */
/* Calculate the unit Vector for v and place it in xUnit.                 */
/*                                                                        */
/**************************************************************************/
 {
  double unitLength;
  int    i;
  /**/
  unitLength = Sqr(v[X]) + Sqr(v[Y]) + Sqr(v[Z]);
  if((unitLength = sqrt(unitLength)) != 0.0)
   {
    for(i = X;i < W;i++)
     {
      xUnit[i] = v[i] / unitLength;
     }
   }
 }


void PointMatrixMul(p,m,newP)
Point  p;
Matrix m;
Point  newP;
/**************************************************************************/
/*                                                                        */
/* Multiply the point 'p' by the matrix 'm' and place the result in       */
/* 'newP'.                                                                */
/*                                                                        */
/**************************************************************************/
 {
  int i,
      j;
  /**/
  for(i = X;i < W;i++)
   {
    newP[i] = m[3][i];
    for(j = X;j < W;j++)
     {
      newP[i] += p[j] * m[j][i];
     }
   }
 }
```

```
void VectorMatrixMul(v,m,newV)
Point  v;
Matrix m;
Point  newV;
/*************************************************************************/
/*                                                                       */
/* Multiply the vector 'v' by the matrix 'm' and place the result in     */
/* 'newV'.                                                               */
/*                                                                       */
/*************************************************************************/
 {
  int i,
      j;
  /**/
  for(i = X;i < W;i++)
   {
    newV[i] = 0.0;
    for(j = X;j < W;j++)
     {
      newV[i] += v[j] * m[j][i];
     }
   }
 }


void MakeVector(p1,p2,v)
Point  p1,
       p2;
Vector v;
/*************************************************************************/
/*                                                                       */
/* Return the direction vector from point p1 to point p2.                */
/*                                                                       */
/*************************************************************************/
 {
  int i;
  /**/
  for(i = X;i < W;i++)
   {
    v[i] = p2[i] - p1[i];
   }
 }



double VectorDot(a,b)
Vector a,
       b;
/*************************************************************************/
/*                                                                       */
/* Return the dot product of vector a.b                                  */
/*                                                                       */
/*************************************************************************/
 {
  int i;
  double dot = 0.0;
  /**/
  for(i = X;i < W;i++)
   {
    dot += a[i] * b[i];
   }
  return(dot);
 }


double PointVectorDot(a,b)
Point  a;
Vector b;
/*************************************************************************/
/*                                                                       */
/* Return the dot product of a.b                                         */
/*                                                                       */
/*************************************************************************/
 {
  int i;
  double dot = 0.0;
  /**/
  for(i = X;i < W;i++)
   {
    dot += a[i] * b[i];
   }
  return(dot);
 }


double PointDot(a,b)
Point  a,
       b;
/*************************************************************************/
/*                                                                       */
/* Return the dot product of a.b                                         */
/*                                                                       */
/*************************************************************************/
 {
  int i;
  double dot = 0.0;
  /**/
  for(i = X;i < W;i++)
   {
    dot += a[i] * b[i];
   }
  return(dot);
 }
```

```
void VectorCross(a,b,cross)
Vector a,
       b,
       cross;
/****************************************************************************/
/*                                                                          */
/* Return the cross product of vector aXb                                   */
/*                                                                          */
/****************************************************************************/
 {
  cross[X] = (a[Y] * b[Z]) - (a[Z] * b[Y]);
  cross[Y] = (a[Z] * b[X]) - (a[X] * b[Z]);
  cross[Z] = (a[X] * b[Y]) - (a[Y] * b[X]);
 }


double Distance(pt0,pt1)
Point pt0,
      pt1;
/****************************************************************************/
/*                                                                          */
/* Get the distance between two world points.                               */
/*                                                                          */
/****************************************************************************/
 {
  return(sqrt(pow((pt1[X] - pt0[X]),2.0) +
              pow((pt1[Y] - pt0[Y]),2.0) +
              pow((pt1[Z] - pt0[Z]),2.0)));
 }
```

```
/******************************************************************************/
/*                                                                            */
/* Module: Memory                                                             */
/*         Malloc() and Free() front-end functions that catch memory          */
/*         allocations bugs.                                                  */
/*                                                                            */
/******************************************************************************/
#include <alloc.h>
#include <stdlib.h>

#define MallocArraySize 500  /* Max number of mallocs at any moment.  */
                             /* Can be increased if not large enough. */

extern void Error(char *msg);

void *Malloc(size_t size),
     Free(void *block),
     FreeAll(void);
int  MemoryEmpty(void);

static unsigned int  numMalloc       = 0,
                     numFree         = 0,
                     mallocArraySize = 0;
static void          *mallocArray[MallocArraySize];


void *Malloc(size)
size_t size;
/******************************************************************************/
/*                                                                            */
/* Same as malloc().                                                          */
/*                                                                            */
/******************************************************************************/
 {
  void          *block;
  unsigned int i      = 0;
  /**/
  block = malloc(size);
  while((mallocArray[i] != NULL) && (i < mallocArraySize))
   {
    i++;
   }
  if(i >= mallocArraySize)
   {
    mallocArraySize++;
   }
  if(i > MallocArraySize)
   {
    Error("Malloc(): Too many mallocs for mallocArray[]");
   }
  mallocArray[i] = block;
  numMalloc++;
  return(block);
 }



void Free(block)
void *block;
/******************************************************************************/
/*                                                                            */
/* Same as free().                                                            */
/*                                                                            */
/******************************************************************************/
 {
  unsigned int i = 0;
  /**/
  while((mallocArray[i] != block) && (i < mallocArraySize))
   {
    i++;
   }
  if(i >= mallocArraySize)
   {
    Error("Free(): Attempt to Free a block currently not Malloced");
   }
  free(block);
  mallocArray[i] = NULL;
  if(i == (mallocArraySize - 1))
   {
    while((mallocArray[mallocArraySize - 1] == NULL) && (mallocArraySize > 0))
     {
      mallocArraySize--;
     }
   }
  numFree++;
 }

int MemoryEmpty(void)
/******************************************************************************/
/*                                                                            */
/* If all Malloc's have been Free'd return 1, else return 0.                  */
/*                                                                            */
/******************************************************************************/
 {
  return((numFree == numMalloc) && (mallocArraySize == 0))?1:0;
 }


void FreeAll(void)
/******************************************************************************/
/*                                                                            */
/* Free all unFreed memory.                                                   */
/*                                                                            */
/******************************************************************************/
 {
  unsigned int i = 0;
  /**/
  for(i = 0;i < mallocArraySize;i++)
   {
    if(mallocArray[i] != NULL)
     {
      Free(mallocArray[i]);
     }
   }
 }
```

```
/****************************************************************************/
/*                                                                          */
/* Module: Model                                                            */
/*          Main module for the modelling of objects in the scene. This     */
/*          calls functions that create the objects from primative types.   */
/*          places these objects in world coordinate space, and assigns to  */
/*          each object its texture and hue information.                    */
/*                                                                          */
/****************************************************************************/
extern void InitFiles(void),
            OpenFiles(void),
            CloseFiles(void),
            AddObj(char *fileName),
            AddToObjList(int num),
            CopyObj(char *srcObj,char *destObj,char *copyFile),
            AddMap(char *mapFile),
            FreeAll(void),
            Error(char *msg),
            Message(char *msg);
extern int  MemoryEmpty(void);


void main(void);


void main(void)
/****************************************************************************/
/*                                                                          */
/* Model the scene to be ray traced.                                        */
/*                                                                          */
/****************************************************************************/
 {
  Message("Initialising Files");
  InitFiles();
  OpenFiles();
  Message("Building model of scene");
  AddObj("d1.txt");
  AddToObjList(1);
  Message("Adding model texture");
  AddMap("Map1.txt");
  AddMap("Map2.txt");
  AddMap("Map9.txt");
  AddMap("Map4.txt");
  AddMap("Map8.txt");
  CloseFiles();
  if(!MemoryEmpty())
   {
    FreeAll();
    Error("main(): Number of Mallocs != Number of Frees");
   }
 }
```

```
/***************************************************************************/
/*                                                                         */
/* Module: Obj (Objects)                                                   */
/*         Various functions needed for objects in the scene.              */
/*                                                                         */
/***************************************************************************/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "define.h"
#include "paths.h"
#include "prim.h"
#include "ray.h"
#include "obj.h"
#include "tList.h"
#include "light.h"
#include "global.h"
#include "map.h"

extern FILE   *primFile,
              *objListFile;

extern void   Error(char *msg),
              *Malloc(size_t size),
              Free(void *block),
              Transform(PrimPt prim,Vector s,Vector t,Point pivot,
                        double rX,double rY,double rZ),
              CSG(ObjPt obj,TListPt *tList,RayPt ray,FILE *objFile);
              TListDel(TListPt *tList),
              VectorCopy(Vector a,Vector b),
              AddPrim(ObjPt obj),
              MatrixCopy(Matrix a,Matrix b),
              MatrixMul(Matrix a,Matrix b),
              PointMatrixMul(Point v,Matrix m,Point newV),
              Inverse(Matrix c,Matrix b),
              Rotate(Matrix m,double rX,double rY,double rZ),
              ReadPrim(int num,PrimPt prim),
              WritePrim(int num,PrimPt prim),
              Scale(Matrix m,Vector s),
              Translate(Matrix m,Vector t);
extern double Roundoff(double num);
extern int    CopyPrim(PrimPt srcPrim,int rRRSTFileNum),
              NewPrimNum(void),
              AddBound(ObjPt obj);
int           GetObj(long *offset);
void          ReadObj(int num,ObjPt obj,FILE *objFile),
              AddObj(char *objData),
              TransferObj(FILE *objFile,double s[3],double t[3],double r[3],double pivot[3]),
              AddToObjList(int num),
              CopyObj(char *srcObj,char *destObj,char *copyTxt),
              WriteObj(int num,ObjPt obj,FILE *objFile),
              CloseObj(FILE *objFile),
              GetHue(int objNum,HuePt hue);
double        IntersectObj(RayPt ray,int objNum,int *primNum,
                           int *surfType,Vector n);

FILE          *OpenObj(int num);



void AddObj(objData)
char *objData;
/***************************************************************************/
/*                                                                         */
/* Add the info in the file objData to the dataBases of the system         */
/*                                                                         */
/* NOTE: info in objData is read in starting at rec 0, ie like an array.   */
/*                                                                         */
/***************************************************************************/
 {
  char          tmpStr[40],
                tmpStr1[40];
  int           i,
                j,
                k;
  char          ch;
  FILE          *textFile    = NULL,
                *objFile     = NULL,
                *srcObjFile  = NULL;
  ObjHeaderPt   header       = NULL,
                srcHeader    = NULL;
  ObjPt         obj          = NULL,
                srcObj       = NULL;
  PrimPt        srcPrim      = NULL;
  /**/
  obj       = (ObjPt)Malloc(sizeof(Obj));
  srcPrim   = (PrimPt)Malloc(sizeof(Prim));
  srcObj    = (ObjPt)Malloc(sizeof(Obj));
  header    = (ObjHeaderPt)Malloc(sizeof(ObjHeader));
  srcHeader = (ObjHeaderPt)Malloc(sizeof(ObjHeader));

  strcpy(tmpStr,ObjPath);
  strcat(tmpStr,objData);

  tmpStr[16] = 'H'; /* Get hue info */
  if((textFile = fopen(tmpStr,"r")) == NULL)
   {
    Error("AddObj(): Obj text file does not exist.");
   }
  fscanf(textFile,"%lf",&header->hue.kDR);
  fscanf(textFile,"%lf",&header->hue.kDT);
  fscanf(textFile,"%lf%lf%lf",&header->hue.fDR[R],
                             &header->hue.fDR[G],
                             &header->hue.fDR[B]);
  fscanf(textFile,"%lf",&header->hue.tT);
  fscanf(textFile,"%lf",&header->hue.kRH);
  fscanf(textFile,"%lf",&header->hue.kTH);
  fscanf(textFile,"%lf",&header->hue.indx);
  fclose(textFile);

  tmpStr[16] = 'D';
  if((textFile = fopen(tmpStr,"r")) == NULL)
   {
    Error("AddObj(): Obj text file does not exist.");
   }
  fscanf(textFile,"%d",&header->numEntries);
  tmpStr[17] = '\0';
```

```
tmpStr[16] = 'O';
i = 1;
while((ch = objData[i]) != '.')
 {
  tmpStr[i   + 17] = '\0';
  tmpStr[i++ + 16] = ch;
 }
objFile = fopen(tmpStr,"w+b");
fclose(objFile);
for(i = 17;i < strlen(tmpStr);i++)
 {
  tmpStr1[i - 17] = tmpStr[i];
 }
objFile = fopen(tmpStr,"r+b");
i = 0;
fwrite(header,sizeof(ObjHeader),1,objFile);
while(i < header->numEntries)
 {
  fscanf(textFile,"%s",obj->opCode);
  switch (obj->opCode[0])
   {
    case '+':
    case '-':
    case '&':
     {
      fscanf(textFile,"%d%d",&obj->l,&obj->r);
      break;
     }
    case 'C':
    case 'B':
    case 'S':
    case 'N':
    case 'T':
    case 'Y':
     {
      fscanf(textFile,"%d",&obj->l);
      obj->r = 0;
      AddPrim(obj);
      break;
     }
    case 'O':
     {
      fscanf(textFile,"%d",&obj->l);
      fscanf(textFile,"%d",&obj->r);
      break;
     }
    default:
     {
      Error("switch case fell to default. Module AddObj. Program obj");
      break;
     }
   }
  WriteObj(i,obj,objFile);
  i++;
 }
fclose(objFile);
objFile = fopen(tmpStr,"r+b");
for(i = 0;i < header->numEntries;i++)
 {
  ReadObj(i,obj,objFile);
  if(strcmp(obj->opCode,"O") == 0)  /* equal */
   {
    strcpy(tmpStr,ObjTxtPath);  /* for Object */
    itoa(obj->l,tmpStr1,10);
    strcat(tmpStr,tmpStr1);
    srcObjFile = fopen(tmpStr,"rb");
    fread(srcHeader,sizeof(ObjHeader),1,srcObjFile);
    ReadObj(0,srcObj,srcObjFile);
    srcObj->l += (header->numEntries - 1);  /* point to EOF */
    srcObj->r += (header->numEntries - 1);
    WriteObj(i,srcObj,objFile);
    k = header->numEntries;
    for(j = 1;j < srcHeader->numEntries;j++)
     {
      ReadObj(j,srcObj,srcObjFile);
      switch(srcObj->opCode[0])
       {
        case '+':
        case '-':
        case '&':
         {
          srcObj->l += (header->numEntries - 1);
          srcObj->r += (header->numEntries - 1);
          break;
         }
        case 'P':
         {
          ReadPrim(srcObj->l,srcPrim);
          srcObj->l = CopyPrim(srcPrim,obj->r);
          break;
         }
        default :
         {
          Error("switch case fell to default. Module AddObj. Program obj");
          break;
         }
       }
      WriteObj(k,srcObj,objFile);
      k++;
     }
    fclose(srcObjFile);
    header->numEntries += (srcHeader->numEntries - 1);
   }
 }
fscanf(textFile,"%s",obj->opCode);
if(strcmp(obj->opCode,"NULL") == 0)
 {
  header->boundVol = -1;
 }
else /* bound vol exists */
 {
  fscanf(textFile,"%d",&obj->l);
  header->boundVol = AddBound(obj);
 }
fseek(objFile,0L,SEEK_SET);
fwrite(header,sizeof(ObjHeader),1,objFile);
fclose(textFile);
fclose(objFile);
```

```
  Free(obj);
  Free(srcPrim);
  Free(srcObj);
  Free(header);
  Free(srcHeader);
 }


void CopyObj(srcObj,destObj,copyTxt)
char *srcObj,
     *destObj,
     *copyTxt;
/************************************************************************/
/*                                                                      */
/* Make a copy of an object.                                            */
/*                                                                      */
/************************************************************************/
 {
  char   tmpStr[80],
         buffer[256];
  double s[3],
         t[3],
         r[3],
         pivot[3];
  FILE   *srcObjFile,
         *destObjFile,
         *copyTxtFile;
  size_t numBytes;
  int    i;
  /**/
  strcpy(tmpStr,ObjPath);
  strcat(tmpStr,srcObj);
  srcObjFile = fopen(tmpStr,"rb");

  strcpy(tmpStr,ObjPath);
  strcat(tmpStr,destObj);
  destObjFile = fopen(tmpStr,"w+b");
  while(feof(srcObjFile) == 0)
   {
    numBytes = fread((void *)buffer,sizeof(char),256,srcObjFile);
    fwrite((void *)buffer,sizeof(char),numBytes,destObjFile);
   }
  fclose(destObjFile);
  fclose(srcObjFile);
  destObjFile = fopen(tmpStr,"r+b");

  strcpy(tmpStr,ObjCopyPath);
  strcat(tmpStr,copyTxt);
  copyTxtFile = fopen(tmpStr,"r");
  fscanf(copyTxtFile,"%lf%lf%lf",&s[X],&s[Y],&s[Z]);
  fscanf(copyTxtFile,"%lf%lf%lf",&t[X],&t[Y],&t[Z]);
  fscanf(copyTxtFile,"%lf%lf%lf",&r[X],&r[Y],&r[Z]);
  fscanf(copyTxtFile,"%lf%lf%lf",&pivot[X],&pivot[Y],&pivot[Z]);
  fclose(copyTxtFile);
  TransferObj(destObjFile,s,t,r,pivot);
  fclose(destObjFile);
 }


void ReadObj(num,obj,objFile)
int    num;
ObjPt obj;
FILE  *objFile;
/************************************************************************/
/*                                                                      */
/* Read field number 'num' from 'objFile' into 'obj'.                   */
/* If no such field exists then it is an error.                         */
/*                                                                      */
/************************************************************************/
  {
   if(fseek(objFile,(long)(num * sizeof(Obj) + sizeof(ObjHeader)),SEEK_SET) != 0) /* bad news */
    {
     Error("ReadObj(): Invalid read attempt");
     exit(0);
    }
   fread(obj,sizeof(Obj),1,objFile);
  }


void WriteObj(num,obj,objFile)
int    num;
ObjPt obj;
FILE  *objFile;
/************************************************************************/
/*                                                                      */
/* Write 'obj' into field number 'num' into 'objFile'.                  */
/* If no such field exists in the file, then it's an error.             */
/*                                                                      */
/************************************************************************/
 {
  if(fseek(objFile,(long)(num * sizeof(Obj) + sizeof(ObjHeader)),SEEK_SET) != 0) /* bad news */
   {
    Error("WriteObj(): Invalid write attempt");
   }
  fwrite(obj,sizeof(Obj),1,objFile);
 }


FILE *OpenObj(num)
int num;
/************************************************************************/
/*                                                                      */
/* Set the open object to being 'num'.                                  */
/*                                                                      */
/************************************************************************/
 {
  char objFileName[40],
       tmpStr[40];
  /**/
  strcpy(objFileName,ObjTxtPath);
  itoa(num,tmpStr,10);
  strcat(objFileName,tmpStr);
  return(fopen(objFileName,"r+b"));
 }
```

```
void CloseObj(objFile)
FILE *objFile;
/*************************************************************************/
/*                                                                       */
/* Close an object file.                                                 */
/*                                                                       */
/*************************************************************************/
 {
  fclose(objFile);
 }


int GetObj(offset)
long *offset;
/*************************************************************************/
/*                                                                       */
/* Return the obj number of the next obj in 'objList'. If no obj exists  */
/* then return -1;                                                       */
/*                                                                       */
/*************************************************************************/
 {
  int num;
  /**/
  fseek(objListFile,*offset,SEEK_SET);
  if(!feof(objListFile))
   {
    fread(&num,sizeof(int),1,objListFile);
    *offset = ftell(objListFile);
    if(!feof(objListFile))
     {
      return(num);
     }
    fseek(objListFile,0L,SEEK_SET);
    *offset = 0L;
    return(-1);
   }
  else
   {
    fseek(objListFile,0L,SEEK_SET);
    *offset = 0L;
    return(-1);
   }
 }


void AddToObjList(objNum)
int objNum;
/*************************************************************************/
/*                                                                       */
/* A the object 'objNum' to the object list.                             */
/*                                                                       */
/*************************************************************************/
 {
  fseek(objListFile,0L,SEEK_END);
  fwrite(&objNum,sizeof(int),1,objListFile);
 }



void TransferObj(objFile,s,t,r,pivot)
FILE   *objFile;
double s[3],
       t[3],
       r[3],
       pivot[3];
/*************************************************************************/
/*                                                                       */
/* Create a new transformation matrix for a copied obj.                  */
/*                                                                       */
/*************************************************************************/
 {
  int          i;
  ObjHeaderPt header;
  ObjPt        obj;
  PrimPt       prim;
  /**/
  header = (ObjHeaderPt)Malloc(sizeof(ObjHeader));
  obj = (ObjPt)Malloc(sizeof(Obj));
  prim = (PrimPt)Malloc(sizeof(Prim));

  fread(header,sizeof(ObjHeader),1,objFile);
  for(i = 0;i < header->numEntries;i++)
   {
    ReadObj(i,obj,objFile);
    if(obj->opCode[0] == 'P')
     {
      ReadPrim(obj->l,prim);
      Transform(prim,s,t,pivot,r[X],r[Y],r[Z]);
      Inverse(prim->transform,prim->inverse);
      obj->l = NewPrimNum();
      WriteObj(i,obj,objFile);
      WritePrim(obj->l,prim);
     }
   }
  if(header->boundVol != -1)
   {
    ReadPrim(header->boundVol,prim);
    Transform(prim,s,t,pivot,r[X],r[Y],r[Z]);
    Inverse(prim->transform,prim->inverse);
    header->boundVol = NewPrimNum();
    WritePrim(header->boundVol,prim);
    fseek(objFile,0L,SEEK_SET);
    fwrite(header,sizeof(ObjHeader),1,objFile);
   }
  Free(prim);
  Free(obj);
  Free(header);
 }
```

```
double IntersectObj(ray,objNum,primNum,surfType,n)
RayPt  ray;
int    objNum,
       *primNum,
       *surfType;
Vector n;
/****************************************************************************/
/*                                                                          */
/* If an intersection between ray and obj exists, this function returns     */
/* the 't' value of the nearest point of intersection between ray and obj   */
/* else returns 0.0                                                         */
/*                                                                          */
/* If an intersection does exist then the color of the primative which      */
/* is the nearest intersection between ray and obj is placed in *primNum,   */
/* and the normal at the point of intersection is in the 'n'.               */
/* If no intersection occurs then *primNum is undefined.                    */
/*                                                                          */
/****************************************************************************/
 {
  TListPt tList    = NULL;
  FILE    *objFile;
  double  t        = 0.0;
  ObjPt   obj;
  /**/
  obj = (ObjPt)Malloc(sizeof(Obj));
  objFile = OpenObj(objNum);
  ReadObj(0,obj,objFile);

  CSG(obj,&tList,ray,objFile);
  if(tList != NULL)
   {
    t = tList->val;
    *primNum = tList->primNum;
    *surfType = tList->surfType;
    VectorCopy(tList->norm,n);
    TListDel(&tList);
   }
  fclose(objFile);
  Free(obj);
  return(t);
 }


void GetHue(objNum,hue)
int   objNum;
HuePt hue;
/****************************************************************************/
/*                                                                          */
/* Get the hue information for objNum.                                      */
/*                                                                          */
/****************************************************************************/
 {
  FILE        *objFile;
  ObjHeaderPt header;
  /**/
  header = (ObjHeaderPt)Malloc(sizeof(ObjHeader));
  objFile = OpenObj(objNum);
  fread(header,sizeof(ObjHeader),1,objFile);
  hue->kDR = header->hue.kDR;
  hue->kDT = header->hue.kDT;
  hue->fDR[R] = header->hue.fDR[R];
  hue->fDR[G] = header->hue.fDR[G];
  hue->fDR[B] = header->hue.fDR[B];
  hue->tT = header->hue.tT;
  hue->kRH = header->hue.kRH;
  hue->kTH = header->hue.kTH;
  hue->indx = header->hue.indx;
  fclose(objFile);
  Free(header);
 }
```

```c
/*************************************************************************/
/*                                                                       */
/* Module: Prim (Primatives)                                             */
/*         Various functions for manipulating primitives.                */
/* NOTE:   Primitives are the basic cubes, spheres, cylinders etc. that are */
/*         used to create objects.                                       */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "define.h"
#include "obj.h"
#include "prim.h"
#include "paths.h"

extern void *Malloc(size_t size),
            Free(void *block),
            Error(char *msg),
            Transform(PrimPt prim,Vector s,Vector t,Point pivot,
                      double rX,double rY,double rZ),
            Rotate(Matrix m,double rX,double rY,double rZ),
            Scale(Matrix m,Vector s),
            Translate(Matrix m,Vector t),
            MatrixCopy(Matrix a,Matrix b),
            MatrixMul(Matrix a,Matrix b),
            Inverse(Matrix c,Matrix b);


int         IsPrim(ObjPt obj),
            CopyPrim(PrimPt srcPrim,int rRRSTFileNum),
            NewPrimNum(void);
void        AddPrim(ObjPt obj),
            ReadPrim(int num,PrimPt prim),
            WritePrim(int num,PrimPt prim);

extern FILE *primTextFile,
            *primFile;


int IsPrim(obj)
/*************************************************************************/
/*                                                                       */
/* If obj holds a primitive leaf (as opposed to being an operator on two */
/* other objs) then return TRUE, else return FALSE.                      */
/*                                                                       */
/*************************************************************************/
ObjPt obj;
 {
  return((obj->opCode[0] == 'P')?TRUE:FALSE);
 }


void AddPrim(obj)
ObjPt obj;
/*************************************************************************/
/*                                                                       */
/* Add a new primative to the primFile. rRRST, map, priority etc. info   */
/* got in Textfile made up of obj->opCode[0], obj->l and .TXT            */
/*                                                                       */
/*************************************************************************/
 {
  FILE   *primTextFile;
  PrimPt prim;
  char   tmpStr[40],
         tmpStr1[40];
  Vector s,       /* scale     */
         t;       /* translate */
  Point  pivot;  /* pivot about which we rotate */
  double rX,
         rY,
         rZ;
  int    primNum,
         i,
         j,
         surface;
  /**/
  prim = (PrimPt)Malloc(sizeof(Prim));
  strcpy(tmpStr,PrimPath);
  strcat(tmpStr,obj->opCode);
  itoa(obj->l,tmpStr1,10);
  strcat(tmpStr,tmpStr1);
  strcat(tmpStr,".txt");
  primTextFile = fopen(tmpStr,"r");
  fseek(primTextFile,0L,SEEK_SET);
  fscanf(primTextFile,"%c",&prim->type);
  fscanf(primTextFile,"%lf%lf%lf",&s[X],&s[Y],&s[Z]);
  fscanf(primTextFile,"%lf%lf%lf",&t[X],&t[Y],&t[Z]);
  fscanf(primTextFile,"%lf%lf%lf",&rX,&rY,&rZ);
  fscanf(primTextFile,"%lf%lf%lf",&pivot[X],&pivot[Y],&pivot[Z]);
  switch(prim->type)
   {
    case 'S':
     {
      surface = 1;
      break;
     }
    case 'B':
     {
      surface = 6;
      break;
     }
    case 'C':
     {
      surface = 3;
      break;
     }
    case 'N':
     {
      surface = 2;
      break;
     }
    case 'Y':
     {
      surface = 5;
      break;
     }
```

```
    default:
      {
       Error("switch case fell to default. Module AddPrim. Program obj");
       break;
      }
    }
  for(i = 0;i < surface;i++)
   {
    fscanf(primTextFile,"%d%d",&prim->map[i][Out],&prim->map[i][In]);
   }
  fscanf(primTextFile,"%d",&prim->priority);
  fclose(primTextFile);
  for(i = 0;i < 4;i++)        /* Copy Identity matrix into prim->transform */
   {
    for(j = 0;j < 4;j++)
     {
      if(i != j)
       {
        prim->transform[i][j] = 0.0;
       }
      else
       {
        prim->transform[i][j] = 1.0;
       }
     }
   }
  Transform(prim,s,t,pivot,rX,rY,rZ);
  Inverse(prim->transform,prim->inverse);
  primNum = NewPrimNum();
  WritePrim(primNum,prim);

  obj->l = primNum;           /* assign primative number to primative. */
  obj->opCode[0] = 'P';
  Free(prim);
 }


int CopyPrim(srcPrim,rRRSTFileNum)
PrimPt srcPrim;
int    rRRSTFileNum;
/****************************************************************************/
/*                                                                          */
/* Get a copy of srcPrim and write it to the primFile                       */
/* manipulate the copy by the info in the file made                         */
/* up of 'A', rRRSTFileNum, .TXT                                            */
/* Return the number of the prim in primFile.                               */
/*                                                                          */
/****************************************************************************/
 {
  PrimPt destPrim;
  char   tmpStr[40],
         tmpStr1[40];
  int    i;
  Vector s,
         t;
  double rX,
         rY,
         rZ;
  Point  pivot;
  FILE   *rRRSTTextFile;
  /**/
  destPrim = (PrimPt)Malloc(sizeof(Prim));
  strcpy(tmpStr,PrimTxtPath);
  itoa(rRRSTFileNum,tmpStr1,10);    /* get the rrrst values */
  strcat(tmpStr,tmpStr1);
  strcat(tmpStr,".txt");
  rRRSTTextFile = fopen(tmpStr,"r+b");
  fscanf(rRRSTTextFile,"%lf%lf%lf",&s[X],&s[Y],&s[Z]);
  fscanf(rRRSTTextFile,"%lf%lf%lf",&t[X],&t[Y],&t[Z]);
  fscanf(rRRSTTextFile,"%lf%lf%lf",&rX,&rY,&rZ);
  fscanf(rRRSTTextFile,"%lf%lf%lf",&pivot[X],&pivot[Y],&pivot[Z]);
  MatrixCopy(srcPrim->inverse,destPrim->transform);
  Scale(destPrim->transform,s);
  Rotate(destPrim->transform,rX,rY,rZ);
  Translate(destPrim->transform,t);
  MatrixMul(destPrim->transform,srcPrim->transform);
  Inverse(destPrim->transform,destPrim->inverse);
  for(i = 0;i < MaxSurface;i++)
   {
    destPrim->map[i][Out] = srcPrim->map[i][Out];
    destPrim->map[i][In] = srcPrim->map[i][In];
   }
  destPrim->priority = srcPrim->priority;
  i = NewPrimNum();
  WritePrim(i,destPrim);
  fclose(rRRSTTextFile);
  Free(destPrim);
  return(i);
 }


int NewPrimNum(void)
/****************************************************************************/
/*                                                                          */
/* Returns the next available number for a new prim. This number is         */
/* always unique for prims written to primFile.                             */
/*                                                                          */
/****************************************************************************/
 {
  div_t result;
  int   fileSize;
  /**/
  fseek(primFile,0L,SEEK_END);
  fileSize = (int)ftell(primFile);
  result = div(fileSize,sizeof(Prim));
  return(result.quot);
 }
```

```
void ReadPrim(num,prim)
int     num;
PrimPt prim;
/****************************************************************************/
/*                                                                          */
/* Read 'prim' number 'num' from database PrimFile.                         */
/* If no such prim exists then it's an error.                               */
/*                                                                          */
/****************************************************************************/
  {
   /**/
   if(fseek(primFile,(long)(num * sizeof(Prim)),SEEK_SET) != 0) /* bad news */
    {
     Error("ReadPrim(): Invalid read attempt");
    }
   fread(prim,sizeof(Prim),1,primFile);
  }


void WritePrim(num,prim)
int     num;
PrimPt prim;
/****************************************************************************/
/*                                                                          */
/* Write 'prim' number 'num' into primFile.                                 */
/* If no such 'num' exists in primFile, then it's an error.                 */
/*                                                                          */
/****************************************************************************/
 {
  if(fseek(primFile,(long)(num * sizeof(Prim)),SEEK_SET) != 0) /* bad news */
   {
    Error("WritePrim(): Invalid write attempt");
   }
  fwrite(prim,sizeof(Prim),1,primFile);
 }
```

```
/*****************************************************************************/
/*                                                                           */
/* Module: Rays                                                              */
/*         Functions needed to generate reflected and transmitted rays.      */
/*         Also the function needed to shot these rays into the world        */
/*         coordinate scene.                                                 */
/*                                                                           */
/*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "define.h"
#include "ray.h"
#include "prim.h"
#include "tList.h"
#include "obj.h"
#include "light.h"
#include "global.h"

extern RGB    background;
extern void   *Malloc(size_t size),
              Free(void *block),
              GetHue(int objNum,HuePt hue),
              ReadObj(int num,ObjPt obj,FILE *objFile),
              VectorCopy(Vector a,Vector b),
              MakeVector(Point p1,Point p2,Vector v),
              ReadPrim(int num,PrimPt prim),
              TransformRay(PrimPt prim,RayPt ray,RayPt newRay),
              VectorMatrixMul(Vector v,Matrix m,Vector newV),
              UnitVector(Vector v,Vector xUnit),
              GetUVCube(Vector n,double *u,double *v,int *surface),
              GetUVPyramid(Vector n,double *u,double *v,int *surface),
              GetUVCylinder(Vector n,double *u,double *v,int *surface),
              GetUVCone(Vector n,double *u,double *v,int *surface),
              GetUVSphere(Vector n,double *u,double *v,int *surface);
              CSG(ObjPt obj,TListPt *tList,RayPt ray,FILE *objFile),
              TListDel(TListPt *tList),
              ProcessColor(HuePt hue,RGB surfColor,RGB colorReflected,
                        RGB colorTransmitted,double sT,Point surfPt,
                        Vector v,Vector n,Vector r,Vector t,RGB color),
              Error(char *msg);
extern FILE   *OpenObj(int num);
extern int    GetObj(long *offset),
              IntersectObjBound(RayPt ray,int objNum),
              ColorNum(RGB rgbColor);
extern double IntersectObj(RayPt ray,int objNum,int *primNum,int *surfType,Vector n),
              Sqr(double x),
              ReadMap(int mapNum,int lambda,double u,double v),
              Roundoff(double x),
              VectorDot(Vector a,Vector b);
void          CastRay(RayPt ray,int depth,RGB rgb),
              ReflectedRay(RayPt ray,double t,Vector n,RayPt newRay),
              TransmittedRay(RayPt ray,int objNum,double t,Vector n,
                        double indx,double *distance,RayPt newRay);


const double Infinity = 1E100;



void CastRay(ray,depth,color)
RayPt ray;
int   depth;
RGB   color;
/*****************************************************************************/
/*                                                                           */
/* Recursively follow a ray backwards from the origin into the world scene. */
/* Return the color of the ray.                                              */
/* Only recurse 'MaxDepth' times.                                            */
/*                                                                           */
/*****************************************************************************/
 {
  Vector     tmpV,
             n,
             unit,
             nearestNorm;
  double     nearest = Infinity,
             u,
             v,
             t       = 0.0,
             sT;
  long       offset  = 0L;
  RayPt      reflectedRay,
             transmittedRay,
             newRay;
  HuePt      hue;
  PrimPt     prim;
  ObjPt      obj;
  int        i,
             colorShadow,
             objNum,
             nearestObj,
             primNum,
             surfType,
             nearestPrim,
             nearestSurfType,
             surface;          /* The intersected surface of a prim */
  Point      surfPt;
  RGB        surfColor,
             colorReflected,
             colorTransmitted;
  /**/
  newRay         = (RayPt)Malloc(sizeof(Ray));
  reflectedRay   = (RayPt)Malloc(sizeof(Ray));
  transmittedRay = (RayPt)Malloc(sizeof(Ray));
  hue            = (HuePt)Malloc(sizeof(Hue));
  obj            = (ObjPt)Malloc(sizeof(Obj));
  prim           = (PrimPt)Malloc(sizeof(Prim));
  UnitVector(ray->xD, ray->xD);
  while((objNum = GetObj(&offset)) != -1)
   {
    if(IntersectObjBound(ray,objNum) == 1)
     {
      if((t = IntersectObj(ray,objNum,&primNum,&surfType,n)) != 0.0)
       {
        if( t < nearest)
```

```
            {
             nearest     = t;
             nearestPrim = primNum;
             nearestObj  = objNum;
             nearestSurfType = surfType;
             UnitVector(n,nearestNorm);
            }
          }
        }
      }
    if(nearest != Infinity)
      {
       /* get color of nearest intersection point */
       ReadPrim(nearestPrim,prim);
       TransformRay(prim,ray,newRay);
       for(i = X;i < W;i++)
         {
          surfPt[i] = ray->xO[i] + nearest * ray->xD[i];
          tmpV[i]   = newRay->xO[i] + nearest * newRay->xD[i];
         }
       switch(prim->type)
         {
          case 'S':
            {
             UnitVector(tmpV,n);
             GetUVSphere(n,&u,&v,&surface);
             break;
            }
          case 'B':
            {
             GetUVCube(tmpV,&u,&v,&surface);
             break;
            }
          case 'Y':
            {
             GetUVPyramid(tmpV,&u,&v,&surface);
             break;
            }
          case 'C':
            {
             GetUVCylinder(tmpV,&u,&v,&surface);
             break;
            }
          case 'N':
            {
             GetUVCone(tmpV,&u,&v,&surface);
             break;
            }
          default:
            {
             Error("switch case fell to default. Module CastRay. Program Intersect");
             break;
            }
         }
       GetHue(nearestObj,hue);
       VectorMatrixMul(nearestNorm,prim->transform,n);
       UnitVector(n,n);
       surfColor[R] = ReadMap(prim->map[surface][nearestSurfType],R,u,v);
       surfColor[G] = ReadMap(prim->map[surface][nearestSurfType],G,u,v);
       surfColor[B] = ReadMap(prim->map[surface][nearestSurfType],B,u,v);
       ReflectedRay(ray,nearest,n,reflectedRay);
       TransmittedRay(ray,nearestObj,nearest,n,hue->indx,&sT,transmittedRay);
       if(depth++ < MaxDepth)
         {                                                  /* recursively follow */
          CastRay(reflectedRay,depth,colorReflected);       /* reflected and      */
          CastRay(transmittedRay,depth,colorTransmitted);   /* transmitted rays   */
         }
       else
         {
          for(i = 0;i < 3;i++)
            {
             colorReflected[i]   = 0.0;
             colorTransmitted[i] = 0.0;
            }
         }
       ProcessColor(hue,surfColor,colorReflected,colorTransmitted,sT,surfPt,
                    ray->xD,n,reflectedRay->xD,transmittedRay->xD,color);
      }
    else
      {
       color[R] = background[R];
       color[G] = background[G];
       color[B] = background[B];
      }
    Free(newRay);
    Free(reflectedRay);
    Free(transmittedRay);
    Free(hue);
    Free(obj);
    Free(prim);
   }


void ReflectedRay(ray,t,n,reflectedRay)
RayPt  ray;
double t;
Vector n;
RayPt  reflectedRay;
/**************************************************************************/
/*                                                                        */
/* Calculate the origin and direction of the reflected ray.              */
/*                                                                        */
/* Apply "Heckbert's Method" for reflected rays.                         */
/*                                                                        */
/**************************************************************************/
  {
   RayPt  tmpRay;
   int    i;
   double twoDot;
   /**/
   tmpRay = (RayPt)Malloc(sizeof(Ray));
   twoDot = VectorDot(ray->xD,n) * 2.0;
   for(i = X;i < W;i++)
     {
      reflectedRay->xO[i] = ray->xO[i] + t * ray->xD[i];
      reflectedRay->xD[i] = ray->xD[i] - (n[i] * twoDot);
     }
```

```
  UnitVector(reflectedRay->xD,reflectedRay->xD);
  for(i = X;i < W;i++)
   {
    reflectedRay->xO[i] += reflectedRay->xD[i];
   }
  Free(tmpRay);
 }



void TransmittedRay(ray,objNum,t,n,indx,distance,transmitRay)
RayPt  ray;
int    objNum;
double t;        /* as in xO + "t" * xD */
Vector n;
double indx,
       *distance; /* The distance travelled by the  */
                  /* ray through the refraction obj */
RayPt  transmitRay;
/**************************************************************************/
/*                                                                        */
/* Calculate the origin and direction of the transmitted ray upon its exit */
/* from the obj through which it has been transmitted. Also find the      */
/* distancence it travels through the obj.                                */
/*                                                                        */
/* NOTE: This function handles Total Internal Reflection.                 */
/*                                                                        */
/*       Apply "Heckbert's Method" for transmitted rays.                  */
/*                                                                        */
/**************************************************************************/
 {
  int     i,
          ok;
  double  c1,
          c2,
          indxInv;
  TListPt tList    = NULL;
  FILE    *objFile;
  ObjPt   obj;
  RayPt   tmpRay;
  /**/
  tmpRay = (RayPt)Malloc(sizeof(Ray));
  obj = (ObjPt)Malloc(sizeof(Obj));
  objFile = OpenObj(objNum);
  ReadObj(0,obj,objFile);
  *distance = 0.0;
  indxInv = 1.0 / indx;

  c1 = -VectorDot(ray->xD,n);
  c2 = sqrt(1.0 - Sqr(indxInv) * (1.0 - Sqr(c1)));
  for(i = X;i < W;i++)
   {
    transmitRay->xD[i] = indxInv * ray->xD[i] + (indxInv * c1 - c2) * n[i];
    transmitRay->xO[i] = ray->xO[i] + t * ray->xD[i] - transmitRay->xD[i];
   }

  ok = FALSE;
  while(!ok)
   {
    CSG(obj,&tList,transmitRay,objFile);
    c1 = VectorDot(transmitRay->xD,tList->next->norm);
    if((c2 = (1.0 - Sqr(indx) * (1.0 - Sqr(c1)))) < 0.0)
     {  /* Total internal reflection occurs */
      for(i = X;i < W;i++)
       {
        transmitRay->xO[i] += tList->next->val * transmitRay->xD[i];
        transmitRay->xD[i] += 2.0 * c1 * -tList->next->norm[i];
        transmitRay->xO[i] -= transmitRay->xD[i];
       }
      *distance += tList->next->val;
      TListDel(&tList);
      tList = NULL;
     }
    else
     {
      c2 = sqrt(c2);
      ok = TRUE;
      *distance += tList->next->val;
     }
   }
  for(i = X;i < W;i++)
   {
    transmitRay->xO[i] += tList->next->val * transmitRay->xD[i];
    transmitRay->xD[i]  = indx * transmitRay->xD[i] +
                          (indx * c1 - c2) * -tList->next->norm[i];
   }
  for(i = X;i < W;i++)
   {
    transmitRay->xO[i] += transmitRay->xD[i];
   }
  TListDel(&tList);
  fclose(objFile);
  Free(tmpRay);
  Free(obj);
 }
```

```
/*************************************************************************/
/*                                                                       */
/* Module: RayTrace                                                      */
/*         Main calling module.                                          */
/*                                                                       */
/*************************************************************************/
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <alloc.h>
#include <math.h>
#include "define.h"
#include "paths.h"
#include "ray.h"


extern int      SetScrCoordsX(double normalisedX),
                SetScrCoordsY(double normalisedY),
                ColorNum(RGB rgbColor,int graphType),
                MemoryEmpty(void);
extern void     CloseFiles(void),
                CastRay(RayPt ray,int depth,RGB color),
                InitLighting(void),
                InitBackground(void),
                Error(char *msg),
                *Malloc(size_t size),
                Free(void *block);
                FreeAll(void),
                MakeVector(Point p1,Point p2,Vector v),
                OpenFiles(void),
                ExitOnEsc(void),
                WaitForEsc(void),
                ClearBuffer(void),
                CloseGraph(void),
                InitGraph(int graphType),
                PutPixel(int x,int y,int color);

extern void     AssignScrPts(ScrPoint uLScr,ScrPoint lRScr,
                        double xMin,double yMin,double xMax,double yMax),
                AssignWPts(Point wLPt,Point wRPt);
extern double Distance(Point pt0,Point pt1),
                WorldStepSize(double x0,double x1,int numPixels);
void            main(void);


double          SourceX =  500.0,       /* ray source x,y,z coords. */
                SourceY =  500.0,
                SourceZ = -1500.0;



void main()
/*************************************************************************/
/*                                                                       */
/* Main function for ray tracing a modelled scene.                       */
/*                                                                       */
/*************************************************************************/
 {
  int         i,
              j,
              k,
              color,
              depth,
              graphType = 1,    /* 0 = DETECT 1 = IBM8514HI */
              numPixAcross,
              numPixDown;
  char        tmpStr[40];
  ScrPoint    uLScr,      /* upper left pixel coords of screen display window */
              lRScr;
  Point       wLPt,       /* The upper left and upper right coner points of */
              wRPt,       /* world coordinate system.                       */
              startPt,
              curPt,
              curAcross, /* The distance traveled across form startPt to curPt */
              curDown,
              wStepAcross,
              wStepDown;
  RGB         rgbColor;
  RayPt       ray;
  FILE        *scrFile;
  /**/
  clrscr();
  InitGraph(graphType);
  OpenFiles();
  ClearBuffer();
  InitBackground();
  InitLighting();
  AssignWPts(wLPt,wRPt);
  AssignScrPts(uLScr,lRScr,0.66,0.0,1.0,1.0);
  ray = (RayPt)Malloc(sizeof(Ray));
  ray->xO[X] = SourceX;
  ray->xO[Y] = SourceY;
  ray->xO[Z] = SourceZ;

  numPixAcross = lRScr[X] - uLScr[X];
  numPixDown   = lRScr[Y] - uLScr[Y];
  for(i = X;i < W;i++)
   {
    wStepAcross[i] = ((wRPt[i] - wLPt[i]) / (double)(numPixAcross));
    wStepDown[i]   = (-wLPt[i] / (double)(numPixDown));
    startPt[i]     = curPt[i]
                   = wLPt[i];
    curAcross[i]   = curDown[i]
                   = 0.0;
   }
  scrFile = fopen(GraphFile,"w+b");
  fwrite(&graphType,sizeof(int),1,scrFile);
  fwrite(&numPixAcross,sizeof(int),1,scrFile);
  fwrite(&numPixDown,sizeof(int),1,scrFile);
  for(i = uLScr[Y];i <= lRScr[Y];i++)
   {
    for(j = uLScr[X];j <= lRScr[X];j++)
     {
      ExitOnEsc();
      for(k = X;k < W;k++)
       {
```

```
        curPt[k] = startPt[k] + curAcross[k] + curDown[k];
        }
      MakeVector(ray->xO,curPt,ray->xD);
      CastRay(ray,0,rgbColor);
      color = ColorNum(rgbColor,graphType);
      fwrite(&color,sizeof(int),1,scrFile);     /* write pixel to a file */
      PutPixel(j,i,color);
      for(k = X;k < W;k++)
        {
        curAcross[k] += wStepAcross[k];
        }
      }
    for(k = X;k < W;k++)
      {
      curDown[k]  += wStepDown[k];
      curAcross[k] = 0.0;
      }
    }
  fclose(scrFile);
  CloseFiles();
  CloseGraph();
  Free(ray);
  if(!MemoryEmpty())
    {
    FreeAll();
    Error("main(): Number of Mallocs != Number of Frees");
    }
  printf("program completed successfully");
  WaitForEsc();
}
```

```
/*****************************************************************************/
/*                                                                           */
/* Module: TextureMp                                                         */
/*         Various functions needed to create and add texture maps to the    */
/*         model along with functions to do inverse mapping of these maps    */
/*         onto the various primitive types.                                 */
/*                                                                           */
/*****************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "define.h"
#include "paths.h"
#include "map.h"

extern void    *Malloc(size_t size),
               Free(void *block),
               Error(char *msg),
               UnitVector(Vector a,Vector n),
               VectorCross(Vector a,Vector b,Vector cross);
extern double  Roundoff(double x),
               PointVectorDot(Point a,Vector b),
               VectorDot(Vector a,Vector b),
               Sqr(double x);
void           AddMap(char *textFile),
               GetUVPoly(Vector n,int a,Point rI,Polygon poly,double *u,double *v),
               GetUVCube(Vector n,double *u,double *v,int *surface),
               GetUVPyramid(Vector n,double *u,double *v,int *surface),
               GetUVCylinder(Vector n,double *u,double *v,int *surface),
               GetUVCone(Vector n,double *u,double *v,int *surface),
               GetUVSphere(Vector n,double *u,double *v,int *surface);
double         ReadMap(int mapNum,int lambda,double u,double v);


const double   PiInv    = 0.318309886,  /* pi div 1.0 */
               TwoPiInv = 0.1591549;


void AddMap(textFileName)
char *textFileName;
/*****************************************************************************/
/*                                                                           */
/* Add a new texture map to the system as a binary mapFile.                  */
/*                                                                           */
/*****************************************************************************/
 {
  FILE         *textFile,
               *mapFile;
  int          mapNum,    /* The number to be assigned to this map file. */
               i,
               j,
               insertPos,
               numEntries;
  double       color;
  char         mapFileStr[40],
               mapNumStr[40];
  MapHeaderPt header;
  /**/
  header = (MapHeaderPt)Malloc(sizeof(MapHeader));
  strcpy(mapFileStr,MapPath);
  strcat(mapFileStr,textFileName);
  if((textFile = fopen(mapFileStr,"r")) == NULL)
   {
    Error("AddMap(): Map TextFile does not exist");
   }

  strcpy(mapFileStr,MapFile);
  fscanf(textFile,"%d",&mapNum); /* Get the mapFile number. */
  itoa(mapNum,mapNumStr,10);
  strcat(mapFileStr,mapNumStr);
  strcat(mapFileStr,"_ .map");  /* leave space in name for R, G B */
  insertPos = strlen(mapFileStr) - 5;
  fscanf(textFile,"%d %d",&header->uScale,&header->vScale);
  numEntries = header->uScale * header->vScale;
  for(i = 0;i < 1;i++)              /* for each of r,g and b */
   {
    switch(i)
     {
      case R:
       {
        mapFileStr[insertPos] = 'R';
        break;
       }
      case G:
       {
        mapFileStr[insertPos] = 'G';
        break;
       }
      case B:
       {
        mapFileStr[insertPos] = 'B';
        break;
       }
      default:
       {
        Error("AddMap:   Fell into case default");
       }
     }
    mapFile = fopen(mapFileStr,"w+b");
    fclose(mapFile);
    mapFile = fopen(mapFileStr,"r+b");
    fwrite(header,sizeof(MapHeader),1,mapFile);   /* Write uv values. */
    for(j = 0;j < numEntries;j++)  /* get map colors         */
     {
      fscanf(textFile,"%lf",&color);
      fwrite(&color,sizeof(double),1,mapFile);     /* Write rgb value. */
     }
    fclose(mapFile);
   }
  Free(header);
  fclose(textFile);
 }
```

```
double ReadMap(mapNum,lambda,u,v)
int    mapNum,  /* The number of the map file. */
       lambda;  /* Wavelength being read       */
double u,
       v;
/***************************************************************************/
/*                                                                         */
/* Return the color number at relative position uv in mapfile.             */
/*                                                                         */
/***************************************************************************/
 {
  char          mapFileName[40],
                tmpStr[40];
  FILE          *mapFile;
  int           newU,
                newV;
  double        color;
  MapHeaderPt header;
  /**/
  header = (MapHeaderPt)Malloc(sizeof(MapHeader));
  itoa(mapNum,tmpStr,10);
  strcpy(mapFileName,MapFile);
  strcat(mapFileName,tmpStr);
  switch(lambda)
   {
    case R:
     {
      strcat(mapFileName,"_R.map");
      break;
     }
    case G:
     {
      strcat(mapFileName,"_G.map");
      break;
     }
    case B:
     {
      strcat(mapFileName,"_B.map");
      break;
     }
    default:
     {
      Error("ReadMap:    Fell into default case statement");
      break;
     }
   }
  mapFile = fopen(mapFileName,"r+b");
  fread(header,sizeof(MapHeader),1,mapFile);
  newU = (int)Roundoff(u * (double)header->uScale);
  newV = (int)Roundoff(v * (double)header->vScale);
  if(newU < 0)
   {
    newU = 0;
   }
  fseek(mapFile,(long)
        (long)(sizeof(MapHeader) +
        sizeof(double) * (double)(newU + (header->uScale) * newV)),
        SEEK_SET);
  fread(&color,sizeof(double),1,mapFile);
  fclose(mapFile);
  Free(header);
  return(color);
 }


void GetUVSphere(n,u,v,surface)
Vector n;
double *u,
       *v;
int    *surface;
/***************************************************************************/
/*                                                                         */
/* Return the uv inverse mapping value for a sphere.                       */
/*                                                                         */
/***************************************************************************/
 {
  Vector cross   = {0.0,1.0,0.0};   /* VectorCross(pole,equator,cross) */
  double alpha,
         beta;
  /**/
  *surface = 0; /* Only one surface ie element 0 */
  alpha = acos(Roundoff(-n[Z]));
  if(((*v = alpha * PiInv) == 0.0) || (*v == 1.0))
   {
    *u = 0.0;
   }
  else
   {
    beta = acos(Roundoff(n[X] / sin(alpha))) * 1.0 * PiInv ;
    if(VectorDot(cross,n) > 0.0)
     {
      *u = beta;
     }
    else
     {
      *u = 1.0 - beta;
     }
   }
 }
```

```
void GetUVCube(n,u,v,surface)
Vector n;
double *u,
       *v;
int    *surface;
/****************************************************************************/
/*                                                                          */
/* Return the uv inverse mapping value for a cube.                          */
/*                                                                          */
/****************************************************************************/
 {
  if(Roundoff(n[X]) == 0.0)
   {
    *u = 1.0 - n[Z];
    *v = 1.0 - n[Y];
    *surface = 3;
    return;
   }
  if(Roundoff(n[Y]) == 0.0)
   {
    *u = n[X];
    *v = n[Z];
    *surface = 5;
    return;
   }
  if(Roundoff(n[Z]) == 0.0)
   {
    *u = n[X];
    *v = 1.0 - n[Y];
    *surface = 0;
    return;
   }
  if(Roundoff(n[X]) == 1.0)
   {
    *v = 1.0 - n[Y];
    *u = n[Z];
    *surface = 1;
    return;
   }
  if(Roundoff(n[Y]) == 1.0)
   {
    *u = n[X];
    *v = 1.0 - n[Z];
    *surface = 4;
    return;
   }
  if(Roundoff(n[Z]) == 1.0)
   {
    *u = 1.0 - n[X];
    *v = 1.0 - n[Y];
    *surface = 2;
    return;
   }
 }


void GetUVCylinder(n,u,v,surface)
Vector n;
double *u,
       *v;
int    *surface;
/****************************************************************************/
/*                                                                          */
/* Return the uv inverse mapping value for a cylinder.                      */
/*                                                                          */
/****************************************************************************/
 {
  if(Roundoff(n[Z]) == 0.0)
   {
    *surface = 1;
    *v = sqrt(Sqr(n[X]) + Sqr(n[Y]));
    *u = acos(n[X] / sqrt(Sqr(n[X]) + Sqr(n[Y]))) * TwoPiInv;
    if(n[Y] < 0.0)
     {
      *u = 1 - *u;
     }
   }
  else
   {
    if(Roundoff(n[Z]) == 1.0)
     {
      *surface = 2;
      *v = 1.0 - sqrt(Sqr(n[X]) + Sqr(n[Y]));
      *u = acos(n[X] / sqrt(Sqr(n[X]) + Sqr(n[Y]))) * TwoPiInv;
      if(n[Y] >= 0.0)
       {
        *u = 1.0 - *u;
       }
     }
    else /* intersects main body */
     {
      *surface = 0;
      *v = n[Z];
      *u = acos(Roundoff(n[X])) * TwoPiInv;
      if(n[Y] < 0.0)
       {
        *u = 1.0 - *u;
       }
     }
   }
 }


void GetUVCone(n,u,v,surface)
Vector n;
double *u,
       *v;
int    *surface;
/****************************************************************************/
/*                                                                          */
/* Return the uv inverse mapping value for a cylinder.                      */
/*                                                                          */
/****************************************************************************/
 {
  if(Roundoff(n[Z]) == 1.0)
   {
```

```
      *surface = 1;
      *v = 1.0 - sqrt(Sqr(n[X]) + Sqr(n[Y]));
      *u = 1.0 - acos(n[X] / sqrt(Sqr(n[X]) + Sqr(n[Y]))) * TwoPiInv;
      if(n[Y] < 0.0)
       {
        *u = 1 - *u;
       }
     }
    else /* intersects main body */
     {
      *surface = 0;
      *v = n[Z];
      if(n[Z] != 0.0)
       {
        *u = acos(Roundoff(n[X] / n[Z])) * TwoPiInv;
       }
      else
       {
        *u = 0.0;
       }
      if(n[Y] < 0.0)
       {
        *u = 1.0 - *u;
       }
     }
  }


void GetUVPyramid(rI,u,v,surface)
Vector rI;  /* intersection point */
double *u,
       *v;
int     *surface;
/*****************************************************************************/
/*                                                                           */
/* Return the uv inverse mapping value for a cube.                           */
/*                                                                           */
/*****************************************************************************/
  {
   Vector n;
   Polygon poly;
   /**/
   if(Roundoff(rI[Y]) == 0.0)
    {
     *u = rI[X];
     *v = rI[Z];
     *surface = 4;
     return;
    }
   /* front */
   if(Roundoff((rI[Y] * 0.44721359499958) - (rI[Z] * 0.894427190999916)) == 0.0)
    {
     poly[0][X] = 0.0;
     poly[0][Y] = 0.0;
     poly[0][Z] = 0.0;
     poly[1][X] = 1.0;
     poly[1][Y] = 0.0;
     poly[1][Z] = 0.0;
     poly[2][X] = 0.5;
     poly[2][Y] = 1.0;
     poly[2][Z] = 0.5;
     n[X] = 0.0;
     n[Y] = -0.447213595499958;
     n[Z] = 0.894427190999916;
     UnitVector(n,n);
     GetUVPoly(n,3,rI,poly,u,v);
     *surface = 0;
     return;
    }


   /* right */
   if(Roundoff((rI[X] * 0.894427190999916) +
               (rI[Y] * 0.44721359499958)  - 0.894427190999916) == 0.0)
    {
     poly[0][X] = 1.0;
     poly[0][Y] = 0.0;
     poly[0][Z] = 0.0;
     poly[1][X] = 1.0;
     poly[1][Y] = 0.0;
     poly[1][Z] = 1.0;
     poly[2][X] = 0.5;
     poly[2][Y] = 1.0;
     poly[2][Z] = 0.5;
     n[X] = -0.894427190999916;
     n[Y] = -0.44721359499958;
     n[Z] =  0.0;
     GetUVPoly(n,3,rI,poly,u,v);
     *surface = 1;
     return;
    }


   /* back */
   if(Roundoff((rI[Y] * 0.44721359499958) +
               (rI[Z] * 0.894427190999916)  - 0.894427190999916) == 0.0)
    {
     poly[0][X] = 1.0;
     poly[0][Y] = 0.0;
     poly[0][Z] = 1.0;
     poly[1][X] = 0.0;
     poly[1][Y] = 0.0;
     poly[1][Z] = 1.0;
     poly[2][X] = 0.5;
     poly[2][Y] = 1.0;
     poly[2][Z] = 0.5;
     n[X] =  0.0;
     n[Y] = -0.44721359499958;
     n[Z] = -0.894427190999916;
     GetUVPoly(n,3,rI,poly,u,v);
     *surface = 2;
     return;
    }
```

```
  /* left */
  if(Roundoff((rI[X] * 0.894427190999916) - (rI[Y] * 0.44721359499958)) == 0.0)
   {
    poly[0][X] = 0.0;
    poly[0][Y] = 0.0;
    poly[0][Z] = 1.0;
    poly[1][X] = 0.0;
    poly[1][Y] = 0.0;
    poly[1][Z] = 0.0;
    poly[2][X] = 0.5;
    poly[2][Y] = 1.0;
    poly[2][Z] = 0.5;
    n[X] = 0.894427190999916;
    n[Y] = -0.447213595499958;
    n[Z] = 0.0;
    GetUVPoly(n,3,rI,poly,u,v);
    *surface = 3;
    return;
   }
 }


void GetUVPoly(n,numPoints,rI,poly,u,v)
Vector  n;
int     numPoints;
Point   rI;
Polygon poly;
double  *u,
        *v;
/*****************************************************************************/
/*                                                                           */
/* Get the uv map coordinates for a polygon with numPoints vertices.        */
/*                                                                           */
/*****************************************************************************/
 {
  int    i;
  double dU[3],
         dV[3],
         tmp,
         dUX,
         dUY,
         dVX,
         dVY,
         kA,
         kB;
  Point  pA,
         pB,
         pC,
         pD;
  Vector nA,
         nB,
         nC,
         qUX,
         qUY,
         qVX,
         qVY;
  /**/
  if(numPoints == 3)    /* deal with triangangular polygon */
   {
    for(i = X;i < W;i++)
     {
      poly[3][i] = poly[2][i];
     }
   }
  for(i = X;i < W;i++)
   {
    pA[i] = poly[0][i] - poly[1][i] + poly[2][i] - poly[3][i];
    pB[i] = poly[1][i] - poly[0][i];
    pC[i] = poly[3][i] - poly[0][i];
    pD[i] = poly[0][i];
   }

  /* Get *u */
  VectorCross(pA,n,nA);
  VectorCross(pC,n,nC);
  dU[X] = PointVectorDot(pD,nC);
  dU[Y] = PointVectorDot(pD,nA) + PointVectorDot(pB,nC);
  dU[Z] = PointVectorDot(pB,nA);
  if(dU[Z] != 0.0) /* polygon is NOT a parallelogram */
   {
    for(i = X;i < W;i++)
     {
      qUX[i] = nA[i] / (2.0 * dU[Z]);
      qUY[i] = -nC[i] / dU[Z];
     }
    dUX = -dU[Y] / (2.0 * dU[Z]);
    dUY = dU[X] / dU[Z];
    kA = Roundoff(dUX + (PointVectorDot(rI,qUX)));
    kB = Roundoff(dUY + (PointVectorDot(rI,qUY)));
    tmp = sqrt(Sqr(kA) - kB);
    if(((*u = kA - tmp) > 1.0) || (*u < 0.0))
     {
      *u = kA + tmp;
     }
   }
  else /* polygon is a parallelogram */
   {
    *u = (VectorDot(rI,nC) - dU[X]) / (dU[Y] - (VectorDot(rI,nA)));
   }


  /* Get *v */
  VectorCross(pB,n,nB);
  dV[X] = PointVectorDot(pD,nB);
  dV[Y] = PointVectorDot(pD,nA) + PointVectorDot(pC,nB);
  dV[Z] = PointVectorDot(pC,nA);
  if(dV[Z] != 0.0) /* polygon is NOT a parallelogram */
   {
    for(i = X;i < W;i++)
     {
      qVX[i] = nA[i] / (2.0 * dV[Z]);
      qVY[i] = -nB[i] / dV[Z];
     }
    dVX = -dV[Y] / (2.0 * dV[Z]);
    dVY = dV[X] / dV[Z];
    kA = dVX + (PointVectorDot(rI,qVX));
```

```
      kB = dVY + (PointVectorDot(rI,qVY));
      tmp = sqrt(Sqr(kA) - kB);
      if(((*v = 1.0 - (kA - tmp)) > 1.0) || (*v < 0.0))
       {
        *v = 1.0 - (kA + tmp);
       }
    }
   else /* polygon is a parallelogram */
    {
     *v = 1.0 - ((VectorDot(rI,nB) - dV[X]) / (dV[Y] - (VectorDot(rI,nA))));
    }
  }
```

```
/****************************************************************************/
/*                                                                          */
/* Module: Transfrm (Transform)                                            */
/*         Various 3-D transformation functions.                           */
/*                                                                          */
/****************************************************************************/
#include "define.h"
#include "prim.h"
#include "ray.h"

extern void     PointMatrixMul(Point p,Matrix m,Point newP),
                VectorMatrixMul(Vector v,Matrix m,Vector newV);
extern double   Roundoff(double num),
                Sin(double num),
                Cos(double num);
void            Transform(PrimPt prim,Vector s,Vector t,Point pivot,
                          double rX,double rY,double rZ),
                Rotate(Matrix m,double rX,double rY,double rZ),
                RotateX(Matrix m,double rX),
                RotateY(Matrix m,double rY),
                RotateZ(Matrix m,double rZ),
                Scale(Matrix m,Vector s),
                Translate(Matrix m,Vector t),
                TransformRay(PrimPt prim,RayPt ray,RayPt newRay);


void Transform(prim,s,t,pivot,rX,rY,rZ)
PrimPt prim;
Point  s,
       t,
       pivot;
double rX,
       rY,
       rZ;
/****************************************************************************/
/*                                                                          */
/* Aquire the data for the scaling, translating, rotating about X Y and Z  */
/* axes of the Identity matrix. Place this data into prim.                 */
/*                                                                          */
/****************************************************************************/
 {
  int i,
      j;
  Point  v,
         c,
         v1,
         tmpPivot;
  /**/
  for(i = X;i < W;i++)
   {
    tmpPivot[i] = pivot[i];
    tmpPivot[i] *= -s[i];
   }
  Scale(prim->transform,s);
  Translate(prim->transform,tmpPivot);
  Rotate(prim->transform,rX,rY,rZ);
  Translate(prim->transform,t);  /* translate from origin */
 }


void Rotate(m,rX,rY,rZ)
Matrix m;
double rX,
       rY,
       rZ;
/****************************************************************************/
/*                                                                          */
/* Rotate the matrix m around the X, Y and Z axes by rX, rY, rZ degrees.   */
/*                                                                          */
/****************************************************************************/
 {
  RotateX(m,rX);
  RotateY(m,rY);
  RotateZ(m,rZ);
 }


void RotateX(m,rX)
Matrix m;
double rX;
/****************************************************************************/
/*                                                                          */
/* Rotate the matrix m around the X axis by rX degrees.                    */
/*                                                                          */
/****************************************************************************/
 {
  int i;
  double tmp,
         c,
         s;
  /**/
  c = Roundoff(Cos(rX));
  s = Roundoff(Sin(rX));
  for(i = 0;i < 4;i++)
   {
    tmp     = m[i][Y];
    m[i][Y] = (tmp * c) - (m[i][Z] * s);
    m[i][Z] = (m[i][Z] * c) + (tmp * s);
   }
 }
```

```
/*************************************************************************/
/*                                                                       */
/* Module: tList                                                         */
/*         Various low level functions involving the manipulation of tLists.*/
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include "define.h"
#include "TList.h"

extern void *Malloc(size_t size),
            Free(void *block),
            VectorCopy(Vector from,Vector to);
void        TListDel(TListPt *tList),
            Add(TListPt *pt),
            Kill(TListPt *pt),
            Copy(TListPt *from,TListPt *to),
            AddToTList(TListPt *tList,double t1,double t2,
                       Vector nIn,Vector nOut,int primNum);


extern TListPt tListPos;


void AddToTList(tList,t1,t2,nIn,nOut,primNum)
TListPt *tList;
double  t1,
        t2;
int     primNum;
Vector  nIn,
        nOut;
/*************************************************************************/
/*                                                                       */
/* Add data to a tList node.                                             */
/*                                                                       */
/*************************************************************************/
 {
  (*tList) = (TListPt)Malloc(sizeof(TList));
  (*tList)->val      = t1;
  (*tList)->primNum  = primNum;
  (*tList)->surfType = Out;
  VectorCopy(nIn,(*tList)->norm);

  (*tList)->next = (TListPt)Malloc(sizeof(TList));
  (*tList)->next->val      = t2;
  (*tList)->next->primNum  = primNum;
  (*tList)->next->surfType = In;
  VectorCopy(nOut,(*tList)->next->norm);

  (*tList)->next->next    = NULL;
 }


void TListDel(tList)
TListPt *tList;
/*************************************************************************/
/*                                                                       */
/* Free the memory being used by a tList.                                */
/*                                                                       */
/*************************************************************************/
 {
  TListPt tmpPt = NULL;
  /**/
  while(*tList != NULL)
   {
    tmpPt = *tList;
    *tList = (*tList)->next;
    Free(tmpPt);
   }
 }


void Copy(from,to)
TListPt *from,
        *to;
/*************************************************************************/
/*                                                                       */
/* Copy data from on tList to another.                                   */
/*                                                                       */
/*************************************************************************/
 {
  (*to)->val = (*from)->val;
  (*to)->primNum = (*from)->primNum;
  (*to)->surfType = (*from)->surfType;
  VectorCopy((*from)->norm,(*to)->norm);
 }


void Add(pt)
TListPt *pt;
/*************************************************************************/
/*                                                                       */
/* Add a node to the tList linked list.                                  */
/*                                                                       */
/*************************************************************************/
 {
  tListPos->next = *pt;
  tListPos       = (*pt)->next;
  *pt            = (*pt)->next->next;
  tListPos->next = NULL;
 }


void Kill(pt)
TListPt *pt;
/*************************************************************************/
/*                                                                       */
/* Remove a node from the tList linked list.                             */
/*                                                                       */
/*************************************************************************/
 {
  TListPt tmp;
  /**/
  tmp = *pt;
  *pt  = (*pt)->next->next;
  Free(tmp->next);
  Free(tmp);
 }
```

```
void RotateY(m,rY)
Matrix m;
double rY;
/*****************************************************************************/
/*                                                                           */
/* Rotate the matrix m around the Y axis by rY degrees.                      */
/*                                                                           */
/*****************************************************************************/
 {
  int i;
  double tmp,
         c,
         s;
  c = Roundoff(Cos(rY));
  s = Roundoff(Sin(rY));
  for(i = 0;i < 4;i++)
   {
    tmp     = m[i][X];
    m[i][X] = (tmp * c)  + (m[i][Z] * s);
    m[i][Z] = (m[i][Z] * c) - (tmp * s);
   }
 }


void RotateZ(m,rZ)
/*****************************************************************************/
/*                                                                           */
/* Rotate the matrix m around the Z axis by rZ degrees.                      */
/*                                                                           */
/*****************************************************************************/
Matrix m;
double rZ;
 {
  int i;
  double tmp,
         c,
         s;
  /**/
  c = Roundoff(Cos(rZ));
  s = Roundoff(Sin(rZ));
  for(i = 0;i < 4;i++)
   {
    tmp     = m[i][X];
    m[i][X] = (tmp * c)  - (m[i][Y] * s);
    m[i][Y] = (m[i][Y] * c) + (tmp * s);
   }
 }


void Scale(m,s)
Matrix m;
Vector s;
/*****************************************************************************/
/*                                                                           */
/* Scale the matrix m by a factor s.                                         */
/*                                                                           */
/*****************************************************************************/
 {
  int i;
  for(i = X;i < W;i++)
   {
    m[i][i] *= s[i];
   }
 }


void Translate(m,t)
Matrix m;
Vector t;
/*****************************************************************************/
/*                                                                           */
/* Translate the matrix m by the factor t.                                   */
/*                                                                           */
/*****************************************************************************/
 {
  int i;
  for(i = X;i < W;i++)
   {
    m[3][i] += t[i];
   }
 }


void TransformRay(prim,ray,newRay)
PrimPt prim;
RayPt  ray,
       newRay;
/*****************************************************************************/
/*                                                                           */
/* Given the source of ray (ray->xO) and a pt along ray (ray->xD) in         */
/* world coordinates transform ray into the objCoordinates of prim. As a     */
/* result of this function x0 will contain the source and x1 will contain    */
/* the unit direction vector of the ray in objCoordinates of prim.           */
/*                                                                           */
/*****************************************************************************/
 {
  PointMatrixMul(ray->xO,prim->inverse,newRay->xO);
  VectorMatrixMul(ray->xD,prim->inverse,newRay->xD);
 }
```

```
/*************************************************************************/
/*                                                                       */
/* Module: User_IO                                                       */
/*         Various user interface functions.                             */
/*                                                                       */
/*************************************************************************/
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "define.h"
#define WAIT while(getch() != (char)27)

int  KBEmpty(void);
void ClearBuffer(void),
     Error(char *msg),
     Message(char *msg),
     ExitOnEsc(void),
     WaitForEsc(void);


int KBEmpty(void)
/*************************************************************************/
/*                                                                       */
/* Return TRUE if keyboard is empty, FALSE otherwise.                    */
/*                                                                       */
/*************************************************************************/
 {
  union REGS xr;
  xr.h.ah = 1;
  int86(0x16,&xr,&xr);  /* zF is bit 3 of REGS.x.flags */
  return((xr.x.flags & (0x4)) != 0);
 }

void ClearBuffer(void)
/*************************************************************************/
/*                                                                       */
/* Clear keyboard buffer.                                                */
/*                                                                       */
/*************************************************************************/
 {
  union REGS xr;
  xr.h.ah = 0x0c;
  xr.h.al = 0x0;
  int86(0x21,&xr,&xr);
 }

void Error(msg)
char *msg;
/*************************************************************************/
/*                                                                       */
/* Print msg and exit from the program.                                  */
/*                                                                       */
/*************************************************************************/
 {
  printf("%s",msg);
  WAIT;
  exit(1);
 }

void Message(msg)
char *msg;
/*************************************************************************/
/*                                                                       */
/* Clear screen and print msg.                                           */
/*                                                                       */
/*************************************************************************/
 {
  clrscr();
  printf("wait....");
  printf("\n\n\n%s",msg);
 }

void ExitOnEsc(void)
/*************************************************************************/
/*                                                                       */
/* Halt program execution if 'ESC' key is hit.                           */
/*                                                                       */
/*************************************************************************/
 {
  if(!KBEmpty() && ((getch()) == (char)27))
   {
    exit(1);
   }
 }

void WaitForEsc(void)
/*************************************************************************/
/*                                                                       */
/* Wait for 'Esc' key to be hit.                                         */
/*                                                                       */
/*************************************************************************/
 {
  WAIT;
 }
```