Effect of Mirages on the Rendering of Natural Environments

by

Danny Robinson

A thesis presented to the University of Manitoba
in fullfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba

ISBN 0-315-85951-2

Canada

Name Danny Robinson

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Electronics and Electrical

SUBJECT TERM

0544 U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

EFFECT OF MIRAGES ON THE RENDERING OF NATURAL ENVIRONMENTS

BY

DANNY ROBINSON

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

The bending of light due to atmospheric refraction is the cause of some very interesting mirage phenomena. A computer graphics program which can produce images of real or imaginary environments exhibiting mirage phenomena is presented. An environment is modelled as a three-dimensional scene containing groups of objects represented by geometric shapes. Bent rays of light are traced backwards from an observer's eye into the scene to determine which objects are visible in a generated image. The curved path of a ray is calculated by a three-dimensional mirage model. The model is a simple extension to an existing two-dimensional mirage model. Images of previously studied mirage phenomena are produced. The generated images are used to create an animated sequence of an aircraft landing.

Acknowledgements

I would like to thank my advisor professor W.H.Lehn for providing me the opportunity to pursue this thesis topic and for his helpful suggestions and patience during its long course of completion.

Contents

1 Introduction

In a natural environment certain atmospheric conditions can cause severe optical distortions due to atmospheric refraction. These optical distortions are commonly known as mirages. This thesis will present a computer program capable of producing visually realistic mirage phenomena in the form of color images. As presented the program is intended to be used as a software tool to aid in the study of mirage phenomena, however, the techniques presented may also be applicable to the field of computer graphics.

Previous research has led to the implementation of a computer program which will accurately simulate mirages on a two dimensional vertical plane [1,2]. To develop a better understanding of mirage phenomena it is desirable to extend the model to three dimensions so that comparisons with real mirages can be made. A previous effort has been made to illustrate three dimensional mirages by non-uniformly mapping an object space into an image space and then displaying the results in the form of wire frame images [3]. Unfortunately wire frame images tend to be cluttered and depth information is often ambiguous. An image of a simulated mirage can show object distortions that are not at all intuitive and to someone not knowing what to look for it is hard to determine exactly what is being represented. To allow qualitive comparisons to be made between photographs of real mirages and their corresponding synthesized images it would be helpful if a simulation program could reproduce much of the detail contained in the original photographs.

A major goal in this thesis is to improve upon the visual realism of simulated mirage images by making use of rendering techniques widely used within the field of computer graphics. The use of an illumination model to shade textured surfaces with specified properties and the reproduction of shadows cast by one object onto another can provide more detail in a synthesized image enhancing the perceived sense of depth.

The presented program implements a computer graphics technique known as ray tracing which can be used to effectively simulate the physical properties of light and its interaction with surfaces in an environment. Light from a source may follow a very complex path before reaching our eyes. The light may be reflected many times and may pass through one or more transmitting media. Ray tracing can account for any changes that may occur in the color and path geometry of light as it is reflected or refracted at an object's surface.

To simplify a traditional ray tracing implementation it is common to ignore most physical properties of air and the effects they may have on the propagation of

light. Light is always assumed to follow a straight path through the atmosphere. Light rays do not always follow a straight line and under certain conditions can bend abruptly even over short distances, as is the case with mirages. To account for atmospheric refraction while ray tracing, a three dimensional extension to the model presented in [1] has been implemented. It is important that the model's functionality has remained completely unchanged so that known results may be reproduced using parameters identical to those previously tested with the two dimensional model. To simplify the program and to reduce image generation times all objects within the environment are assumed to be opaque. This eliminates the need to simulate the refraction of light through transparent objects. The program will be used to create images using data from previously studied mirage phenomena.

Chapter 2 provides a brief overview of the two dimensional mirage model and discusses some of the possible image distortions that may occur as a result of atmospheric refraction. Chapter 3 briefly describes ray tracing as it is used in computer graphics. The chapter will cover the ray tracing algorithm, viewing geometry, ray-object intersections, illumination, shading, texturing and object transformations. Chapter 4 explains how the two dimensional mirage model has been extended to three dimensions so that it may be used to account for atmospheric refraction while ray tracing. The chapter also discusses ray-object intersections with curved light rays. Chapter 5 discusses software and hardware implementation. Chapter 6 illustrates sample images produced by the program. Chapter 7 serves as a user manual for the program. The chapter discusses program execution and the command line options that may be specified. The chapter also explains a scene description language that is used to model the environment and specify any other required program data. A brief conclusion is presented at the end of this thesis.

2 Mirage Ray Tracing in Two Dimensions

An accurate two dimensional model for optical ray tracing in the surface layer of a refracting atmosphere has been previously developed and presented in [1,3]. The underlying theory of atmospheric refraction is also discussed in [3]. This section will briefly summarize the model.

2.1 Ray Curvature

Nearly horizontal light rays follow paths whose vertical curvature κ depends on the atmospheric density ρ (a function of elevation z). The curvature of a ray is given by

$$\kappa = -\{ \sin\theta \ / \eta \} \ d\eta / dz \qquad (2.1)$$

where θ is the angle between the ray and the vertical, η is the refractive index of air and the sign is such that positive curvature implies a ray concave towards the earth. The refractive index is expressed as a function of temperature using the relation

$$\eta = 1 + \varepsilon \rho = 1 + (\varepsilon \beta \ p / T) \qquad (2.2)$$

where ρ is density, p is pressure, T is absolute temperature and ε, β are constants (226×10^{-6} and 3.48×10^{-3} respectively).

Upon substitution of equation (2.1) into (2.2) and using the hydrostatic equation $dp / dz = -g \rho$, the curvature is expressed as

$$\kappa = \{ dT / dz + \beta g \} \varepsilon \beta \ p \ \sin\theta / \eta T^2 \qquad (2.3)$$

where g is acceleration due to gravity.

The atmosphere is modelled by a small, finite number of concentric spherical layers or shells with each layer boundary considered to be an isothermal surface. The atmosphere is assumed laterally homogeneous over the small ranges of interest so that ray calculations may be confined to a two dimensional plane. The light rays considered are nearly horizontal making small angles with the layer boundaries. It is assumed that a vertical temperature distribution is known, represented by a piecewise-linear function containing a discrete set of elevation, temperature pairs. Within a layer the temperature gradient is assumed constant.

If the temperature within a layer varies linearly, then it may be replaced by an average temperature with minimal error. With the layer temperature constant, the quantities p, η and $\sin\theta$ may be considered constant. With these approximations and the above assumptions the ray curvature κ as expressed by (2.3) is considered constant within a layer.

2.2 Atmospheric Model

The circular arcs representing the earth, ray and layer boundaries are all replaced by parabolic arcs as shown in figure 2.1. All layer boundaries are approximated by vertically displaced parabolas. The equation of a ray expressed as a parabolic arc, at a horizontal distance x from an initial elevation z_0 is

$$z = -(\kappa_{eff}/2)\, x^2 + x \tan\phi_0 + z_0 \qquad (2.4)$$

where ϕ_0 is the initial trajectory angle the ray makes with the horizontal and κ_{eff} is the effective ray curvature in each layer. The effective ray curvature accounts for the curvature of the earth allowing it to modelled as a flat surface:

$$\kappa_{eff} = \kappa_{ray} - \kappa_{earth}$$

Since the only rays of interest are those which reach an observer's eye, light rays are traced backwards away from an observer. Note that this is in the reverse direction to which light actually travels. Given that light enters our eyes, it may seem more natural to follow a ray's path from a source of light determining where it ends up. In practice this is not very efficient since a large number of rays would have to be considered with only a small number eventually reaching an observer's eye.

The path a ray takes through the layered atmosphere is found by incrementally stepping along the horizontal direction testing for ray intersections with layer boundaries. Since the atmosphere is assumed laterally homogeneous over the entire range of interest, the same temperature profile is used at each step (this allows the effective ray curvature and density of each layer to be pre-calculated). If a ray intersects a layer at a distance d, then a new coordinate system is established with the new origin taken to be d (This assumes very small ray curvatures). Within the shifted coordinate system a new ray trajectory is determined (ray slopes are continuous across layer boundaries [13]) and a different parabolic arc is traced as shown in figure 2.2. The path is followed until one of the following terminating conditions occur: a ray propagates upwards past the highest layer, a ray propagates downwards below the lowest layer, a ray hits the ground or a maximum distance has been reached. The complete ray path is

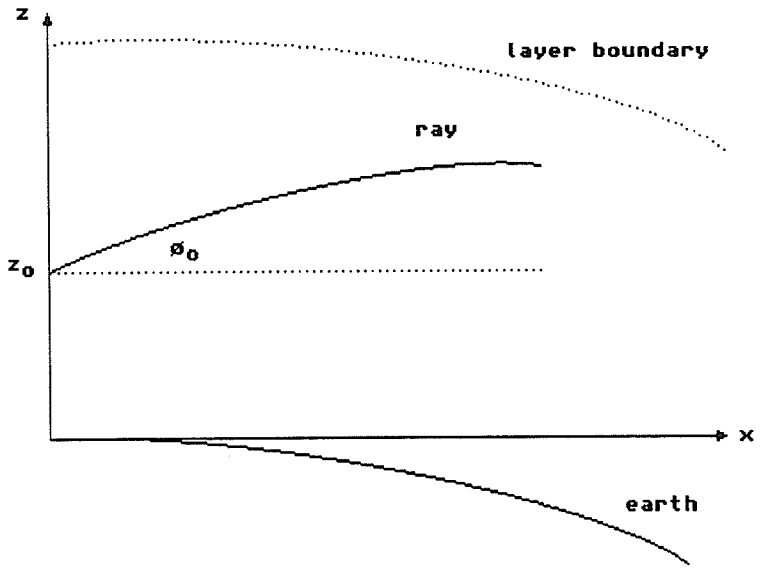

Figure 2.1 Coordinate system of parabolic ray

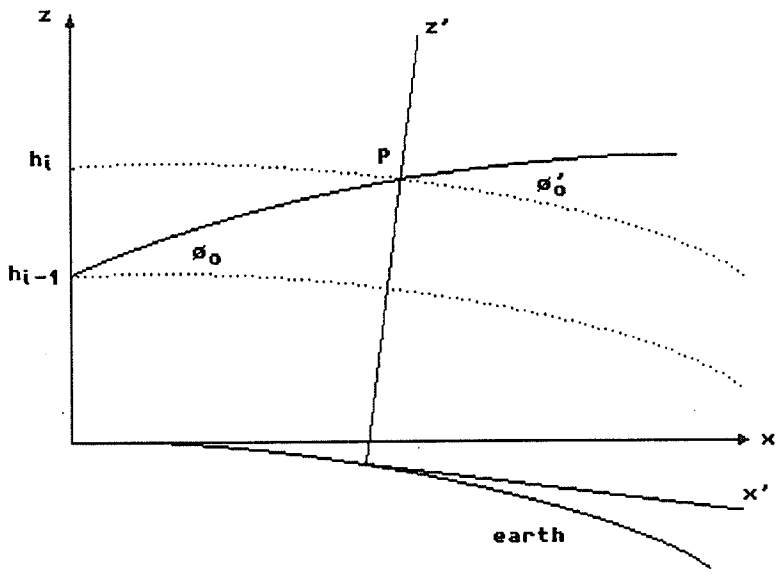

Figure 2.2 Change of coordinate system at layer boundary

broken up into many non-uniformly spaced intervals, each containing its own parabolic arc segment. Each parabolic ray segment is defined within the range spanning its two corresponding layer intersections. The first segment would be defined between the observer and the first layer intersected.

2.3 Effects of Atmospheric Refraction

As a ray propagates through the temperature layers it may bend upwards or downwards. In a temperature inversion, where a warmer less dense layer is positioned above a colder layer there will be a gradual change in refractive indices across the layer boundary. Light will tend to bend towards the colder region where the refractive index is higher. Since our eyes perceive incoming light to be in a straight line, an object will appear to be displaced vertically upwards as shown if figure 2.3. The apparent height h of an object, placed a distance d from an observer at an elevation e, is given by

$$h = e + d \tan \phi$$

where ϕ is the ray's trajectory angle from the eye. This is illustrated in figure 2.4.

If a colder layer is positioned above a warmer layer then light will tend to bend upwards with the apparent height of an object displaced vertically downwards.

The amount of vertical displacement is dependent on the temperature gradient of the temperature profile. If the temperature varies linearly with elevation then an object will apppear to be displaced uniformly, either upwards or downwards vertically. If the temperature varys nonuniformly with elevation then an object will appear to be either stretched or compressed vertically. Severe object distortion may be apparent for highly irregular refraction due to irregularities in a temperature profile. The resulting curvature of some rays may be large enough that rays may bend either upwards or downwards crossing other rays in which case an object will appear inverted. This is especially evident in thermoclines where abrupt changes in temperature occur over small changes in elevation. If a thermocline is positioned slightly above the elevation of an observer's eye then overlapping multiple distortions may possibly occur with an object appearing to be vertically displaced, inverted and stretched all at the same time.

The temperature profile, the elevation of the observer, the elevation of the object and the distance of the observer from that object all influence the amount and type of object distortion perceived.

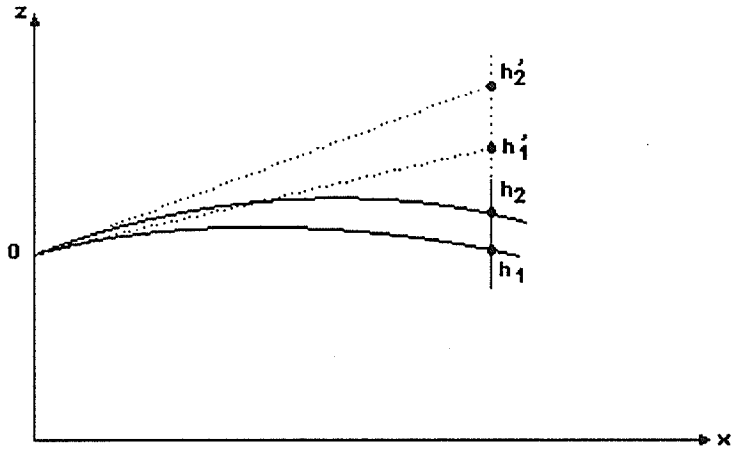

Figure 2.3 An object appearing vertically displaced upwards

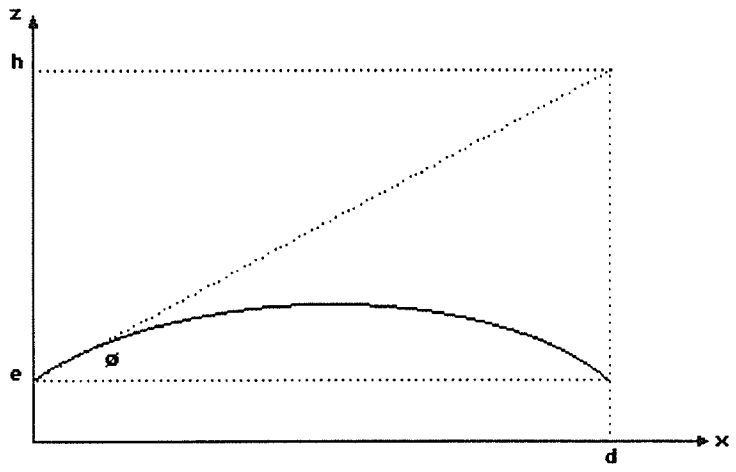

Figure 2.4 Calculating the vertical displacement of an object

3 Ray Tracing in Computer Graphics

One of the main goals in computer graphics is the production of visually realistic synthesized images. Although the term "visually realistic" has a rather broad meaning, in this context we desire to be capable of producing images "real" enough, so that some qualitative comparisons between a photograph of the environment being graphically recreated and its synthesized image can be made.

Within the computer graphics literature there are many methods available to synthesize photo-realistic images. No single method could be considered the "best" as they all have their own advantanges and disadvantages with one method possibly being more suited to a particular application than another. One of the more popular methods is ray tracing, which effectively simulates light propagation within an environment using basic principles of geometric optics. This method has been chosen due to the algorithm's simplicity and its similarity to mirage ray tracing.

Ray tracing simulates the propagation of light through an environment by tracing light rays through a scene to determine how they interact with mathematically modelled entities. The scene is a three dimensional, geometric representation of the environment consisting of groups of objects which represent the shapes of physical entities. In a natural environment we may be interested in modelling entities such as building structures, trees, rock formations or bodies of water for example. In general objects may be either opaque or transparent however opaque objects are only of interest therefore discussions regarding the interaction of light with transparent objects will not be made (a body of water may be modelled as an opaque object with a highly specular surface if viewed from a distance). Objects are positioned within the scene in such a way as to reflect the spatial arrangement of the entities being modelled. Objects are described by three dimensional shapes such as polygonal structures or quadrics. Associated with each object is a set of attributes describing its surface appearance such as color and depending on the amount of surface detail that is to be represented, other attributes such as material properties and texture may be specified.

Object attributes are used in conjunction with an illumination or lighting model for the scene which defines how object surfaces are to be shaded as they are illuminated by light sources. A light source is commonly modelled as an invisible point in space which radiates uniformly in all directions. A light source may also be a single point placed infinitely far away radiating uniformly in a single direction. Lights may or may not be visible in the environment. For example, in a

natural scene the main source of illumination is the sun which, if positioned high enough in the sky, does not fall within our field of view. If a light is visible, then it will also be represented by a three dimensional shape and will have attributes describing how it should appear. There are many factors which may be considered when attempting to represent the illumination of real entities in a natural environment. For example, if the sky is fairly overcast then the majority of illumination will be of a diffuse nature due to the scattering of sun light by cloud particles. On a clear day the environment will appear much more bright as it is illuminated by direct sunlight. Other factors such as a snow covered ground may be accounted for. A natural environment would appear much brighter as the sun's rays are scattered off the snow indirectly illuminating physical entities.

In natural environments light propagation is also affected by precipitation, haze or temperature. The scene may also be structured to model such atmospheric conditions.

This chapter discusses most of the techniques used within the program to produce a final image. The chapter will describe the ray tracing algorithm, viewing geometry, ray-object intersections, illumination and shading, texturing and geometric transformations.

3.1 Ray Tracing Algorithm

This section will give a brief overview of the ray tracing algorithm as applicable to a scene containing opaque objects only. More complete discussions of the ray tracing algorithm and its use in rendering a scene containing both opaque and transparent objects may be found elsewhere [4,5,6,10].

Note on Notation

In this section and all subsequent sections dealing with three dimensional geometry, points will be represented by uppercase letters in italics and vectors will be represented by uppercase letters in bold. A dot '•' is used to represent the vector dot product and a 'x' is used to represent the vector cross product.

The scene is viewed from a point in space out through an imaginary window contained within a viewplane. The view point represents an observer's eye and the window represents the image that will be synthesized. As with mirage ray tracing we are only interested in those rays that reach our eyes, therefore light rays are traced backwards from the observer's eye out through the window into the scene as shown in figure 3.1.

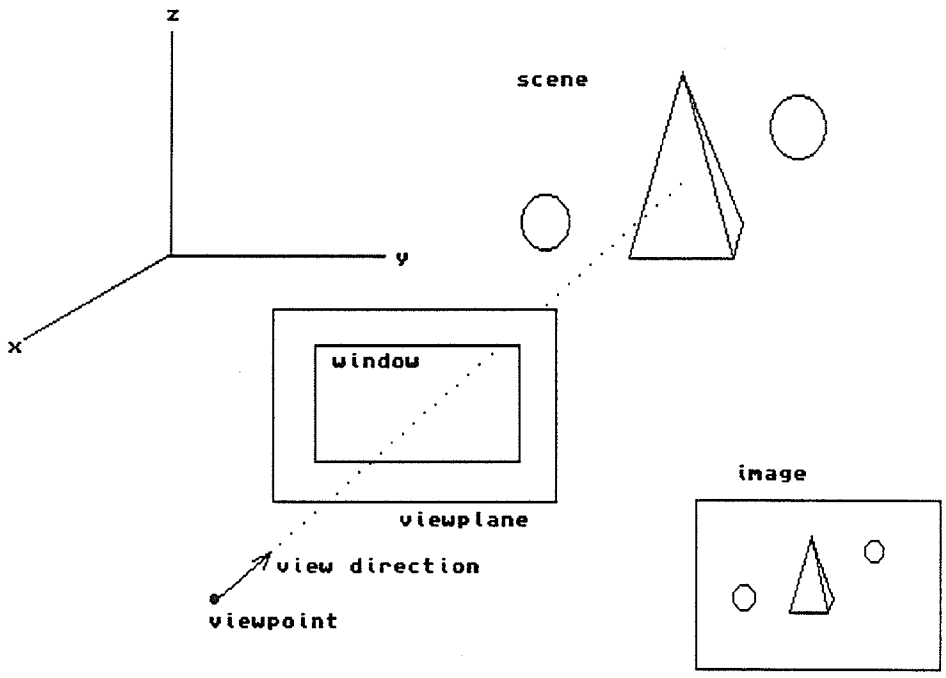

Figure 3.1 Observer looking out through a window into the scene

Since the window represents an image, it may be thought of as a rectangular grid of squares where each square corresponds to a certain pixel location within the image. For each pixel within the window, an "eye" ray **V** is shot from the view point *O*, through that pixel's center and out into the scene as shown in figure 3.2. Once the ray is sent out into the scene, ray-object intersection tests are performed to determine the nearest object hit by the ray, if any. The nearest object will be the one whose intersection point *P* is closest to the observer's viewpoint. Thus in figure 3.2 , the ray **V** has intersected both sphere 1 and polygon 1, however point P_1 is closer therefore sphere 1 is visible to the observer. At the point of intersection an illumination model is applied to determine the contribution to the object's surface color from indirect or ambient lighting and direct contributions from all light sources in the scene. When calculating the direct contribution from a light source an object's surface may be shadowed by another object. Shadows may be detected by sending a shadow or illumination ray **L** from the surface intersection point towards a light performing additional ray-object intersection tests. If the shadow ray does not hit any other object then the contribution to the surface color from that light source is calculated. This is also illustrated in figure 3.2 where shadow rays $\mathbf{L}_1$ and $\mathbf{L}_2$ are sent out towards lights 1 and 2 respectively. No objects lie within the path of $\mathbf{L}_1$ therefore P_1 is illuminated by light1. The shadow ray directed towards light 2 however hits sphere 2 therefore any possible contribution from light 2 is ignored.

At the point of intersection an additional reflected ray may be generated if the object's surface is specularly reflective. A reflected ray will be sent in a direction **R** given by

$$\mathbf{R} = \mathbf{V} - 2\,\mathbf{N}\,(\mathbf{V} \cdot \mathbf{N}) \qquad (3.1)$$

where **V** is the incident ray and **N** represents an object's surface normal at the point of intersection. This reflected ray is in turn traced to determine a reflected color. Figure 3.3 shows the geometry that is considered at each intersection point.

The ray tracing algorithm is inherently recursive in that each reflected ray may hit another object which can also generate additional shadow and reflected rays at its intersection point as shown in figure 3.4. These additional rays form what is commonly called a "ray tree" as shown in figure 3.5. Each node in the tree represents a ray-object intersection. A horizontal branch represents a shadow ray and the downward branch represents a reflected ray. A downward branch in the tree is terminated when either a reflected ray does not hit any other object within the scene or when some predetermined depth has been reached, in which case the ray is assigned a specified background or ambient color. If an

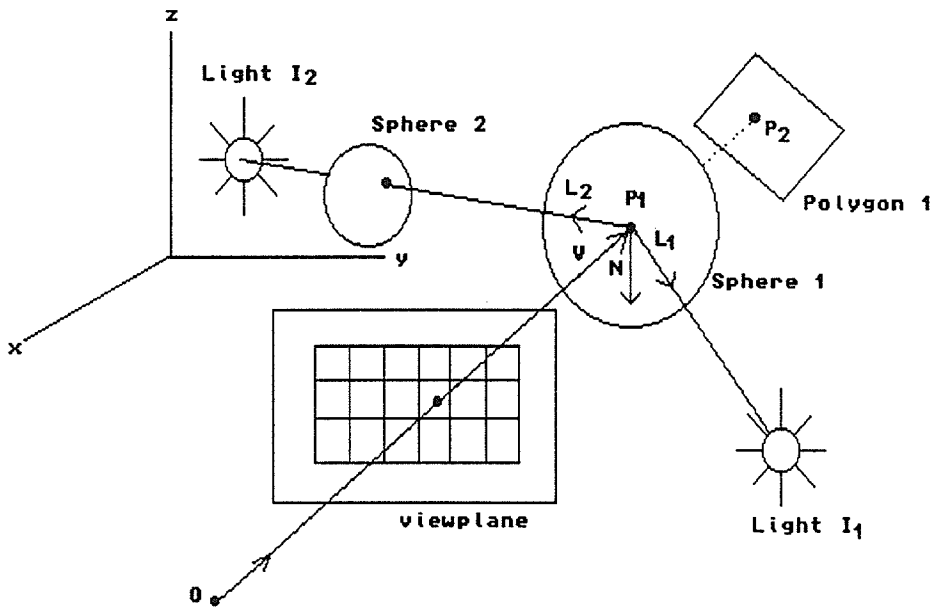

Figure 3.2 Light ray shot through an image pixel into the scene

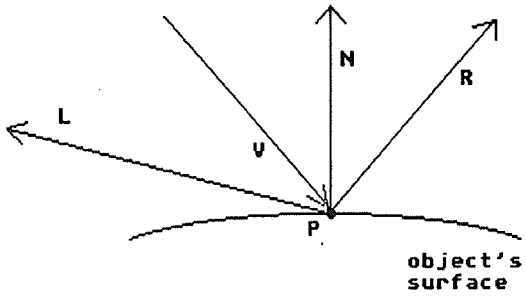

Figure 3.3 Geometry at ray-surface intersection point

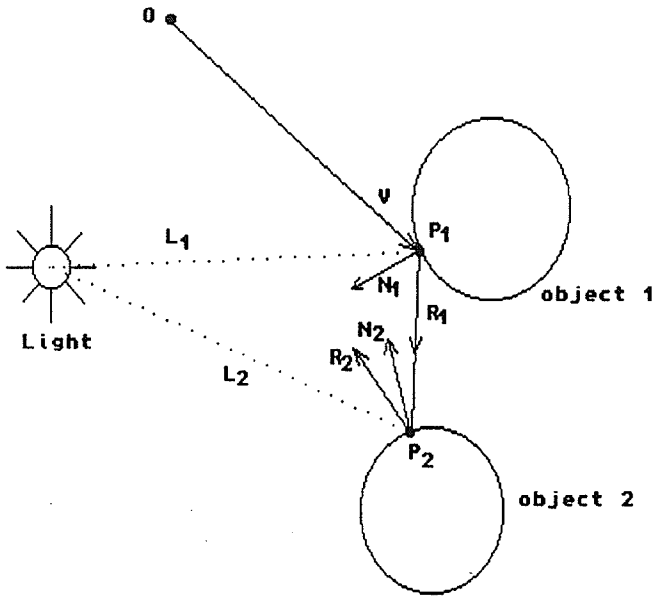

Figure 3.4 Ray propagation through a scene

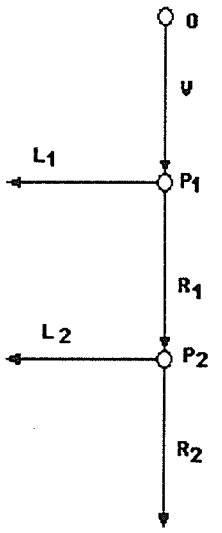

Figure 3.5 Ray tree

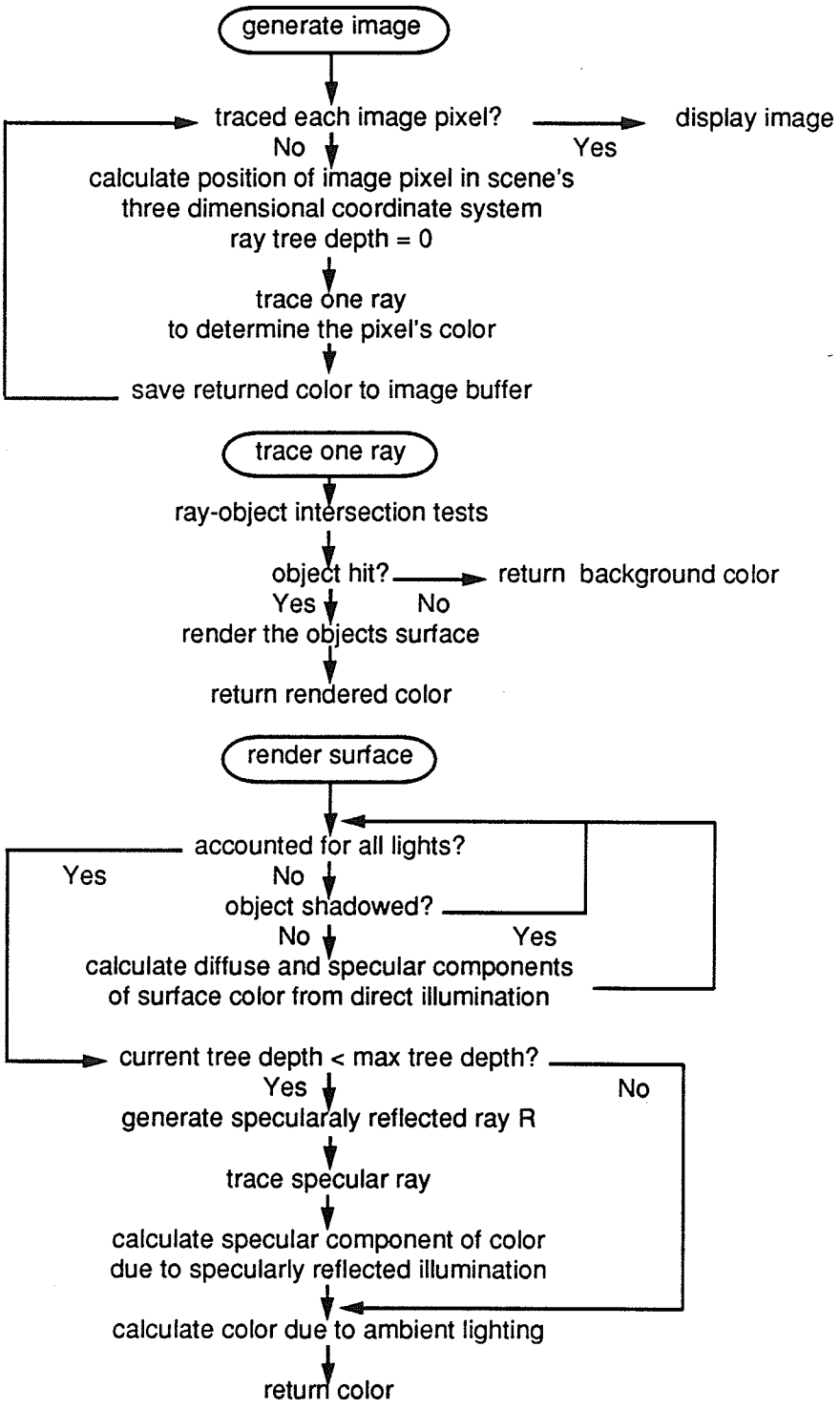

Figure 3.6 Flow chart for generic ray tracing with no transparent objects

intersected object has a completely diffuse surface then a reflected ray will not be generated and the tree will be terminated at that node.

The final color of the object's surface at the highest node of the tree, that is the object visible to the observer, is determined by passing the colors of object surfaces from the bottom up. Assuming that the object at the bottom of the tree has a diffuse surface its final color will only be due to contributions from indirect and direct lighting. This color is passed up to the next object as its reflected color. At this point an illumination model is used to determine how the reflected color should contribute to the object's appearance. The final color of this object's surface will be a combination of the colors calculated by both indirect and direct lighting as well as the reflected color. The tree is traversed upwards until the final color at the initial intersection point has been determined.

Figure 3.6 shows a flowchart of the ray tracing algorithm as presented.

3.2 Viewing Geometry

Before light rays may be traced through a scene their origin and initial direction must be known. This section will discuss the viewing geometry that is used to determine what area of the scene is to be rendered.

The observer is assumed to be looking through a pin-hole camera with a limited field of view as shown in figure 3.7. The observer's viewpoint O is positioned at the apex of a rectangular viewing pyramid or frustrum whose axis lies along the observer's viewing direction. The frustrum defines a three dimensional volume which contains all the visible objects within the scene. The frustrum cuts through a viewplane whose normal **VPN** is aligned with the observer's viewing direction. The intersection of the frustrum with the viewplane forms a rectangular viewport of unit height and width centered at a view reference point *VRP* as shown in figure 3.8. The distance between the viewpoint and the view reference point measured along the viewing direction is called the view distance Vd. It determines the amount of perspective that will be seen in the final image. The view distance is determined by the frustrum angle, β commonly referred to as the observer's field of view. Varying the field of view simulates a telephoto lens. Small angles (large view distances) will give the effect of zooming in on far away objects making them appear larger and limiting the extent of what is visible within the scene. Larger field of view angles (small view distances) produce a wide angle lens effect allowing more of the scene to be visible with objects appearing smaller. The observer's position, view direction and field of view provide enough information to determine where the viewport is located within the scene.

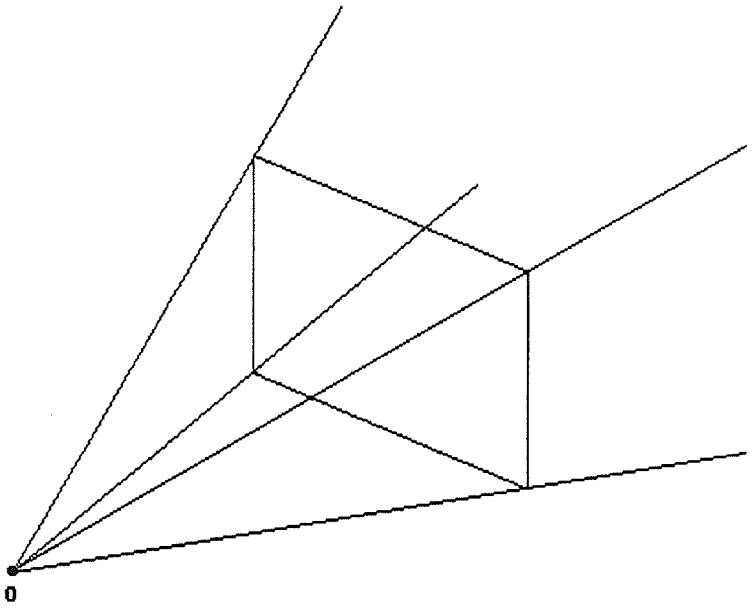

Figure 3.7 The viewing pyramid

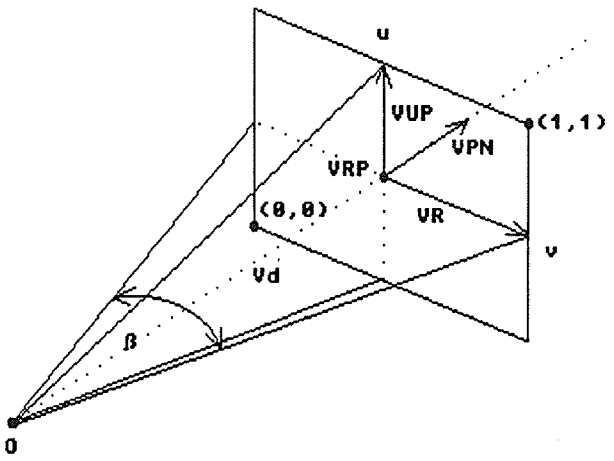

Figure 3.8 View geometry

To determine the viewplane's orientation about the view direction one needs to first establish a general **UP** direction. Once a general **UP** direction is known a left handed coordinate system may be formed consisting of the viewplane normal **VPN**, a vector **VUP** lying within the viewplane and a vector **VR** representing the observer's right, also lying within the viewplane. The orientation of the viewplane will affect the way the scene appears as the observer looks out through the viewport. The effect produced is similar to what one would see looking out of an airplane as it banks left or right.

To set up the viewing geometry the program requires the observer's position within the scene, the point at which the observer is looking, the desired field of view and a camera tilt (amount to rotate the viewplane clockwise about the viewing diection). Given that the observer is at a position *O*, looking at a target point *T*, the viewplane normal **VPN** is given by

$$\mathbf{VPN} = (T - O) / | T - O |$$

The general **UP** direction is arbitrarily set to be aligned with the positive Z axis in normalized form. If a tilt angle is specified then the **UP** vector is first rotated about the vector **VPN**. Once the **UP** vector is established the unit vectors **VR** and **VUP** are determined as follows:

$$\mathbf{VR} = \mathbf{VPN} \times \mathbf{UP}$$

$$\mathbf{VUP} = \mathbf{VR} \times \mathbf{VPN}$$

If the **VPN** vector happens to be aligned with either the positive or negative Z axis (the observer is either looking straight up or straight down) then the **UP** vector is arbitrarily set to either the positive or negative X axis so that the vector **VR** may still be calculated. The vectors **VPN**, **VR** and **VUP** determine the orientation of the viewport.

The location of the viewport's center *VRP* within the scene may be calculated from the field of view angle β (assumed to be expressed in degrees). The distance Vd between the viewplane and the observer along the direction **VPN** is first determined by

$$Vd = 0.5 \cot(0.5 \beta \pi / 180.0)$$

and is then used to calculate the view reference point *VRP* given by

$$VRP = O + Vd \ \mathbf{VPN}$$

Once the viewport has been located a two dimensional uv coordinate system may be established to map physical display coordinates into three dimensional points within the scene. The u coordinates run horizontally and the v coordinates run vertically. The bottom left hand corner of the viewport is given by the point (0,0) and the top right hand corner is represented by the point (1,1). If a display screen is 320 pixels wide by 200 pixels high, any display pixel (x,y) would map to the viewport coordinates (x/319, y/199). For any pixel (x,y) its corresponding three dimensional point *P* within the scene will be determined by

$$P = VRP + (2u - 1)\ \mathbf{VR} + (2v - 1)\ \mathbf{VUP}$$

If the origin of the display screen is considered to be in the upper left hand corner then the point *P* is given by

$$P = VRP + (2u - 1)\ \mathbf{VR} + (1 - 2v)\ \mathbf{VUP}$$

Once a pixel's location within the scene is known, the initial ray direction **D** is given by

$$\mathbf{D} = (P - O) / | P - O |$$

3.3 Ray-Object Intersections

In order to determine what is visible within the scene, intersection tests must be performed to determine whether or not the projected light rays hit any objects. A large portion of the time spent generating images is due to ray-object intersection tests; therefore there is a great deal of literature on this subject [4,5]. Since speed optimization was not a major concern the intersection algorithms used are very basic. This section will discuss the methods implemented.

3.3.1 What is an Object

Up to this point objects have been described fairly abstractly with some mention that they represent a physical element in the modelled environment. As implemented in the program an object is a collection of simple geometric primitives arranged in some user defined manner so as to form a more complex structure. The program makes use of the following simple geometric shapes: plane, triangle, rectangle and sphere. Chapter 7 discusses how these primitives may be used to create more complex shapes.

3.3.2 What Object is Hit

Any point along a ray with origin O and normalized direction $\mathbf{D}$ may be expressed in the parametric form:

$$r(t) = O + t\,\mathbf{D} \qquad (3.2)$$

where t is the distance of the point from the ray's origin. All of the following intersection tests substitute this ray equation into a primitive's defining equation to solve for a value of t. If a solution does not exist then the primitive is not hit. If t is negative then the intersection point lies behind the observer in which case it is ignored. All positive values of t are tested against a current t_{near} which represents the closest hit thus far. If the value of t for any primitive intersection is less than t_{near}, then that primitive is remembered and the value of t_{near} is reassigned the value t.

To determine the "best" t, each object within the scene is considered. For every object, each of its primitives is tested on an individual basis.

3.3.3 Sphere Intersection

A sphere is defined by an origin P and a radius r as shown in figure 3.9. The geometry of the problem as viewed on the plane containing the sphere's center is shown in figure 3.10 where the distance t to the intersection point I is being determined. From the figure one may observe the following:

$$v = \mathbf{R} \cdot \mathbf{D}$$

$$r^2 = d^2 + h^2$$

and

$$d^2 = |\mathbf{R}|^2 - v^2 = \mathbf{R} \cdot \mathbf{R} - (\mathbf{R} \cdot \mathbf{D})^2$$

where

$$\mathbf{R} = P - O$$

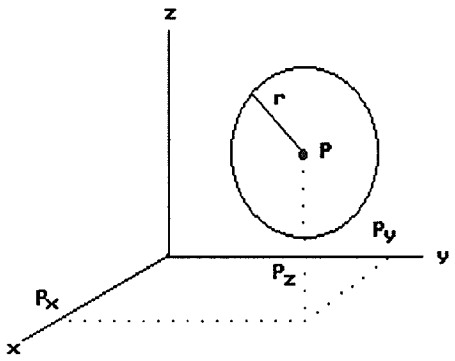

Figure 3.9 Sphere primitive

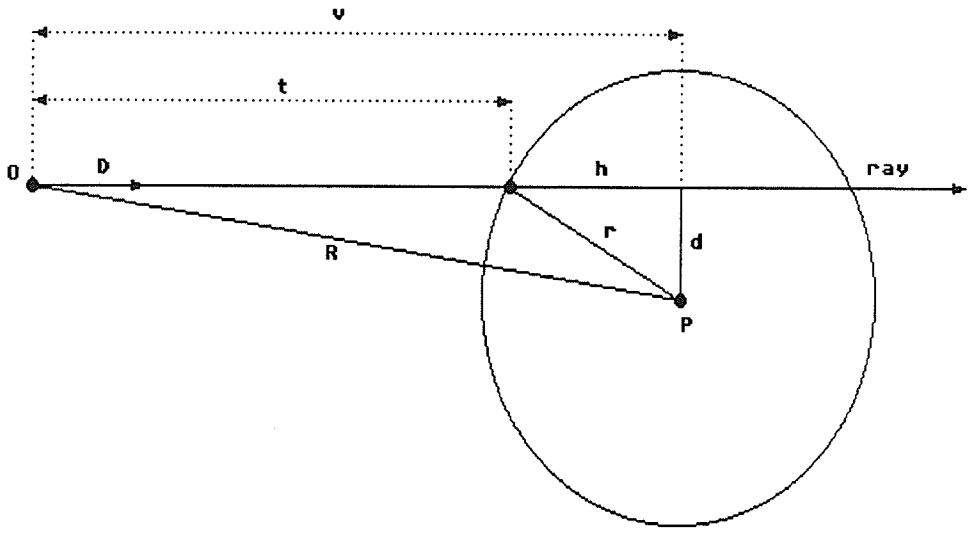

Figure 3.10 Geometry of ray-sphere intersection test

The value of t is then given by

$$t = v - h$$

$$= \mathbf{D} \cdot \mathbf{R} - h$$

$$= \mathbf{D} \cdot \mathbf{R} \; +/- \; (\; r^2 - d^2 \;)^{1/2}$$

$$= \mathbf{D} \cdot \mathbf{R} \; +/- \; (\; r^2 - \mathbf{R} \cdot \mathbf{R} + (\mathbf{R} \cdot \mathbf{D})^2 \;)^{1/2}$$

$$= a \; +/- \; b$$

If the discriminant under the square root is less than 0 then a solution does not exist and the sphere in question is not intersected by the ray. If the discriminant equals 0 then the ray grazes the sphere in which case it is also rejected; otherwise the complete expression is evaluated to obtain two values of t as follows:

$$t_1 = a - b$$

$$t_2 = a + b$$

If t_2 is less than or equal to zero then the intersection point is behind the origin and is rejected. If t_2 is greater than zero then we may determine whether the ray is entering or exiting the sphere. If t_1 is greater than 0 then the ray is entering the sphere at $t = t_1$ otherwise the ray is exiting the sphere at $t = t_2$. If the value of t is less than the current value of t_{near} then the sphere in question is considered the closest primitive hit thus far and t_{near} is reassigned the value of t. At the point of intersection the sphere's surface normal is given by $I - P$.

3.3.4 Planar Intersections

Each of the plane, triangle and rectangle primitives share a common intersection test as they all require a defining plane. Both the triangle and rectangle will require further consideration however this test is always applied first. Each of these primitives is defined by a point P lying within a plane whose normal **N** is given by

$$\mathbf{N} = \mathbf{U} \times \mathbf{V}$$

as shown in figure 3.11.

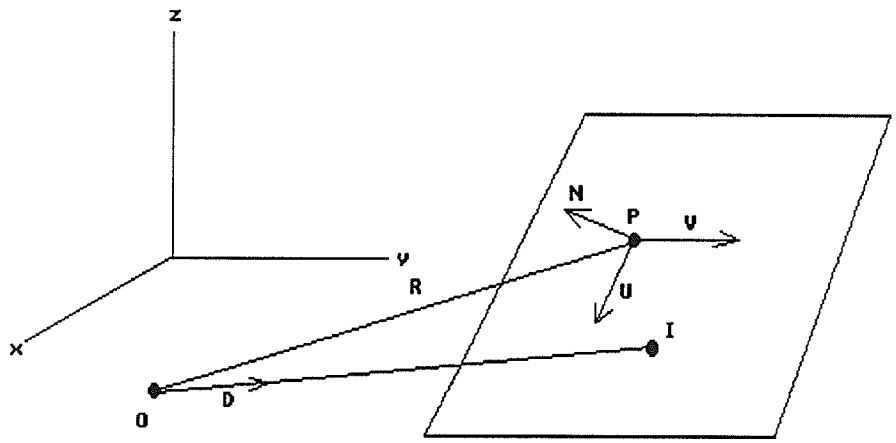

Figure 3.11 Geometry of ray-plane intersection test

For any point I on the plane the quantity $I \cdot \mathbf{N}$ is constant. The implicit representation of a plane and is given by

$$\mathbf{N} \cdot I + d = 0 \tag{3.3}$$

By substituting the parametric representation of the ray (3.2) into (3.3) the following expression is obtained at the point of intersection:

$$\mathbf{N} \cdot (O + t\,\mathbf{D}) + d = 0$$

The parameter t may then be determined as follows:

$$\begin{aligned} t &= -(d + \mathbf{N} \cdot O) / \mathbf{N} \cdot \mathbf{D} \\ &= -(\mathbf{N} \cdot P + \mathbf{N} \cdot O) / \mathbf{N} \cdot \mathbf{D} \\ &= \mathbf{N} \cdot (P - O) / \mathbf{N} \cdot \mathbf{D} \end{aligned} \tag{3.4}$$

If the ray lies within the defining plane, that is, if $\mathbf{N} \cdot \mathbf{D} = 0$, then the primitive is rejected. If t is negative then the intersection point is behind the ray's origin and the primitive is rejected. A positive value of of t is compared against t_{near} to determine if it is the smallest value found thus far. If the value of t is not less than t_{near} then no further tests are performed and the primitive is rejected. If t is less than t_{near}, and the primitive in question is a plane, then the plane is considered the closest primitive hit thus far. If the primitive is either a triangle or a rectangle then additional tests are required to determine whether or not the intersection point is contained within the primitive's interior.

3.3.5 Triangle Intersection

Given that the ray has intersected the plane containing a triangle at a "best" distance t along **D**, it still has to be determined whether or not the intersection point *I* lies within that triangle's interior. The geometry of the problem is shown in figure 3.12. Any point *I* within the triangle is given by

$$I = P + u\,\mathbf{U} + v\,\mathbf{V} \tag{3.5}$$

for u { 0 ..1 } and v { 0 ..1 } and u + v <= 1. At the point of intersection with the ray, equation (3.2) is substituted into (3.5) to obtain

$$O + t\,\mathbf{D} = P + u\,\mathbf{U} + v\,\mathbf{V}$$

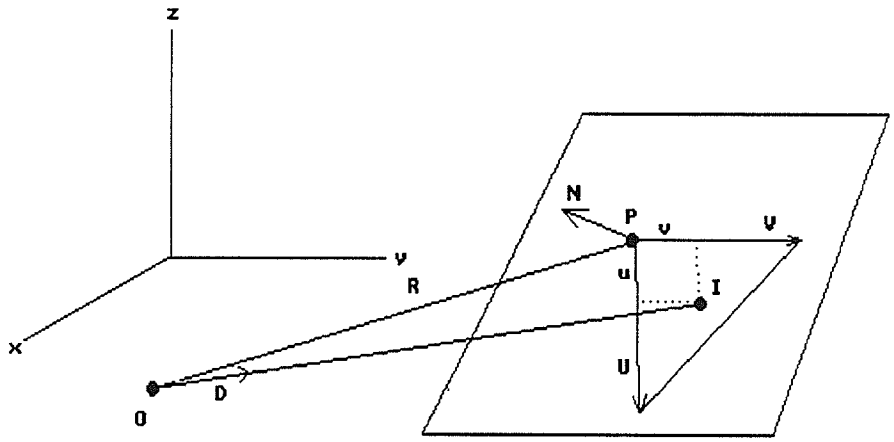

Figure 3.12 Geometry of ray-triangle intersection test

which can be expressed as

$$t\,\mathbf{D} - u\,\mathbf{U} - v\,\mathbf{V} = P - O = \mathbf{R} \tag{3.6}$$

The vector equation (3.6) represents three simultaneous equations (x, y and z components) with three unknown values. The quantities t, u and v are determined as follows:

$$t = \mathbf{R} \cdot (\mathbf{U} \times \mathbf{V}) / \mathbf{D} \cdot \mathbf{N} = \mathbf{R} \cdot \mathbf{N} / \mathbf{D} \cdot \mathbf{N}$$

$$u = -\mathbf{D} \cdot (\mathbf{R} \times \mathbf{V}) / \mathbf{D} \cdot \mathbf{N} \tag{3.7}$$

$$v = -\mathbf{D} \cdot (\mathbf{U} \times \mathbf{R}) / \mathbf{D} \cdot \mathbf{N} \tag{3.8}$$

The calculation of t is not performed here as it was previously determined in the ray-plane intersection test, equation (3.4).

If the intersection point *I* is within the interior of the triangle then the quantities u and v must be in the range { 0..1 } and their sum must be less than or equal to 1. If the conditions for u and v are met then the intersection point lies within the triangle's interior and the primitive is considered the closest hit thus far (we already know it is the closest as a result of the preceding plane test).

3.3.6 Rectangle Intersection

Given that the ray has intersected the plane containing a rectangle at a"best"distance t along **D**, it still has to be determined whether or not the intersection point *I* lies within that rectangle's interior. Any point *I* within the rectangle is given by

$$I = P + u\,\mathbf{U} + v\,\mathbf{V}$$

for u { 0 ..1 } and v { 0 ..1 }. The required test is similar to that of the triangle the only difference being that the sum of u and v must be less than or equal to 2. It is sufficient to just check the values of u and v to make sure both quantities are in the range { 0..1 }. If the two values are in range, then the intersection point lies within the rectangle's interior and the primitive is considered the closest hit thus far.

3.4 Illumination and Shading

Once an object is intersected its surface appearance at that point is determined by a shading model. There are many different methods that may be used to render a surface. This topic is thoroughly discussed in [4,5,6]. The illumination model used within the program is a "hacked" implementation of the physically based illumination model presented in [9] and the global illumination model presented in [10]. In refering to the implementation as "hacked", it is meant that numerous "kludges" are used (with no physical foundation) in order to make these models work and produce fairly realistic looking renderings. A surface is modelled as a collection of randomly oriented planar microfacets with each facet being a perfectly smooth reflector (mirrorlike). The light reflected from the surface is a combination of specular and diffuse components. Specular reflectance is a result of single reflections from the mirrorlike facets and is dependent on their geometry and distribution. Diffuse reflectance is the result of internal scattering and the interreflection of light amongst the microfacets. A surface or light color is approximated by a RGB (red, green, blue) value whose individual components are in the range {0 ..1}. A light's intensity is implied in its color. The color of an object's surface at a point *P* for n light sources is expressed in the form of an illumination equation given by

$$I = I_{amb} + \sum_{j=1}^{n} I_j (\mathbf{N} \cdot \mathbf{L}_j) (k_d O_d + k_s O_s) + I_s k_h k_s \qquad (3.9)$$

Each quantity is explained in the following discussion.

1) Diffuse Surface Color O_d

O_d is the color of light reflected from an object's surface as seen from any viewing direction, when that surface is illuminated by white light. This quantity is represented by a RGB triplet. This value may be modified by color texturing prior to the illumination calculation.

2) Surface Normal **N**

N is the object's surface normal at the point being illuminated. It may be "bumped" in a random direction by a specified "fuzziness" parameter or in a direction that is calculated by a procedural texture prior to its use in (3.9). The resulting appearance of the surface will have a simulated large scale roughness.

3) Contribution due to Ambient Light I_{amb}

I_{amb} is the component of a surface's color that is due to illumination from incident ambient light I_a. Ambient light accounts for illumination that is not attributable to direct contributions from light sources or specular contributions from reflecting surfaces. It is caused by the scattering of light in all directions from surfaces with both diffuse and specular components. Ambient light may be approximated by either a constant RGB value in which case it is assumed to be equally incident from all directions or by tracing a specified number m of randomly directed rays from the surface. In the former case I_{amb} is given by

$$I_{amb} = I_a \, k_a \, O_d$$

where I_a is a specified RGB triplet and the quantity k_a is a constant that determines the extent to which incident ambient light affects a surface's appearance. In the later case I_{amb} is given by

$$I_{amb} = 1/m \sum_{i=1}^{m} I_{ai} \, (\mathbf{N} \cdot \mathbf{D}_i) \, k_d \, O_d$$

where I_{ai} is one sample of the incident ambient light found by ray tracing in the random direction $\mathbf{D}_i$ and whose RGB value is also determined by (3.9). The sample rays do not intersect light sources and their initial direction $\mathbf{D}_i$ does not coincide with the specularly reflected direction **R**. To generate a sample ray, three uniform random numbers in the range { 0 .. 1 } are assigned to each of its x, y and z components. The sign of each component is also determined by a random number. The random rays are shot into the illuminating hemisphere enclosing the surface point therefore the quantity $\mathbf{N} \cdot \mathbf{D}_i$ must be greater than 0. If the dot product is less than 0, then the direction $\mathbf{D}_i$ is negated. If the dot product equals 0, then the sample ray lies on the surface and it is replaced by a new random direction. The quantity k_d is the diffuse reflection coefficient for the surface.

4) Direct Light I_j

I_j is the color of incident light from the *jth* light source. For an "infinite" light source this color will be some user specified RGB value. For a "point" light source the color I_j represents the RGB value given by

$$I_j = I_p \, (\mathbf{L}_j \cdot \mathbf{L}_d)^{\rho} / (1 + d)^{\alpha}$$

where I_p is the light's specified color, $\mathbf{L}_d$ is a specified illumination axis (the

direction the light is shining), d is the distance between the point source's position in space and the surface point *P*, α is an attenuation factor to account for reduction in light intensity with distance and ρ is a value that determines how directional the light's illumination is about the axis $\mathbf{L}_d$. Large values of ρ produce highly directional spotlight effects whereas low values produce more of a diffuse floodlight effect. A value of 0 for ρ produces a light source which radiates uniformly in all directions. The resulting color which I_j represents is dependent on user specified values. For example, if both ρ and α are specified as 0 then I_j would just represent I_p unmodified.

5) Light (Shadow) Ray $\mathbf{L}_j$

$\mathbf{L}_j$ is the vector directed towards the *jth* source of light that is illuminating the surface at *P*. For an infinite light source this direction is the same at any point on the object's surface. For a point source positioned in space at *S*, the direction **L** is given by

$$\mathbf{L} = (S - P) / | S - P |.$$

6) Diffuse and Specular Coefficients k_d k_s

The quantities k_d and k_s are respectively the diffuse and specular reflection coefficients where $k_d + k_s = 1$. Light reflecting from a surface may be composed of both diffuse and specular components. These quantities determine the amount contributable to each.

7) Specular Surface Color O_s

O_s is the color of light specularly reflected from a surface. The color is calculated and is dependent on many factors. This quantity is used to approximate the specular bidirectional reflectivity given as

$$O_s = O_{sF} \; D \; G / (\mathbf{N} \cdot \mathbf{V}) \; (\mathbf{N} \cdot \mathbf{L}_j)$$

where

V is the incident view direction. D is a microfacet distribution function that describes the orientation of the microfacets. It is calculated using the Beckman distribution function given by

$$D = \{ 1 / (4 r^2 \cos^4 \beta) \} \; e^{-\mu}$$

where

$$\mu = (\tan \beta / r)^2$$

$$\cos \beta = \mathbf{N} \cdot \mathbf{H}$$

and

$$\mathbf{H} = (\mathbf{V} + \mathbf{L}_j) / | \mathbf{V} + \mathbf{L}_j |$$

The quantity r is a specified roughness parameter. Small values of r will produce specular highlights that are highly directional. Larger values of r produce highlights that are more spread out. This function is scaled by a value that is interpolated from a table of pre-calculated constants indexed by r. This limits the function's value to the range {0..1} for any r.

G is a geometric attenuation factor which accounts for shadowing between microfacets. It is expressed as the minimum of three values:

$$G = \min\{ 1, 2 (\mathbf{N} \cdot \mathbf{H}) (\mathbf{N} \cdot \mathbf{V}) / (\mathbf{V} \cdot \mathbf{H}), 2 (\mathbf{N} \cdot \mathbf{H}) (\mathbf{N} \cdot \mathbf{L}) / (\mathbf{V} \cdot \mathbf{H}) \}$$

O_{sF} is the color of specular highlights and is dependent on the incident viewing direction and an average value determined by the Fresnel equation (an average refractive index is used). It is given by

$$O_{sF} = O_d' + m (I_j - O_d') \ \max\{ 0, (F_{avg\gamma} - F_{avg0}) / (1.0 - F_{avg0}) \}$$

where F_{avg0} is the Fresnel equation's value at normal incidence and $F_{avg\gamma}$ is the value of the Fresnel equation at an angle γ given by

$$\gamma = \cos^{-1} (\mathbf{V} \cdot \mathbf{H})$$

The quantity m is a measure of how metallic the surface is. For non-metallic surfaces the specular highlight will generally be the color of the incident light. The color of specularly reflected light from a metal will be that of its surface. O_d' is a color that is a blend from the object's original surface color O_d towards pure white. The amount of blending depends on the metallic parameter m. Large values of m will result in little change from the original color whereas smaller values will result in O_d approaching the color of the incident light.

8) Specularly Reflected Light I_s

I_s represents the color of specularly reflected light from other objects. It is the "net" color that is returned by calculating the surface colors for all those objects positioned at lower levels in the ray tree.

9) Hardness k_h

The quantity k_h is a hardness parameter which specifies to what degree a surface will reflect incident light due to interobject specular reflections. One may think of this parameter as a measure of light absorption. Large values of hardness will model the surface as a perfect reflector, such as a mirror, whereas smaller values will simulate more of a plastic reflector.

The specification of these parameters and their effect on an object's resultant surface appearance is covered in more detail in chapter 7.

3.5 Texturing

Prior to rendering an object's surface a procedural solid texture may be applied to modify its surface color or surface normal to add some interest to its rendered appearance. One may texture the surface so that it appears as a checker board, piece of wood, chunk of marble, spotted, blotchy, covered with snow etc. Modifying the surface normal allows one to simulate bumpy or rippling type surfaces. Any number of textures may be applied to a surface and each texture may be individually scaled, orientated and positioned as it is applied.

Many of the textures are based on the use of a noise function [8] which transforms a three-dimensional point within the scene into a three-dimensional texture space returning a pseudo-random number in the range {0 ..1}. Each point on an object's surface will have an associated point in texture space. An integer lattice is defined where each integral point has an associated random value. If the texture space point in question falls on a lattice point then the random value at that point is returned. If a texture space point falls between lattice points then a cubic interpolation is performed along the lattice edges, first along x, then along y and finally along z.

The program implements a lattice that is 4096 x 4096 x 4096. The lattice is duplicated along any coordinate axis by computing the texture space points modulo 4096. For example the point (4097,-2,1) will map to the lattice point (1,4094,1). Since storing a random value at each lattice point is impractical all three-dimensional points are hashed into a table of 256 random numbers.

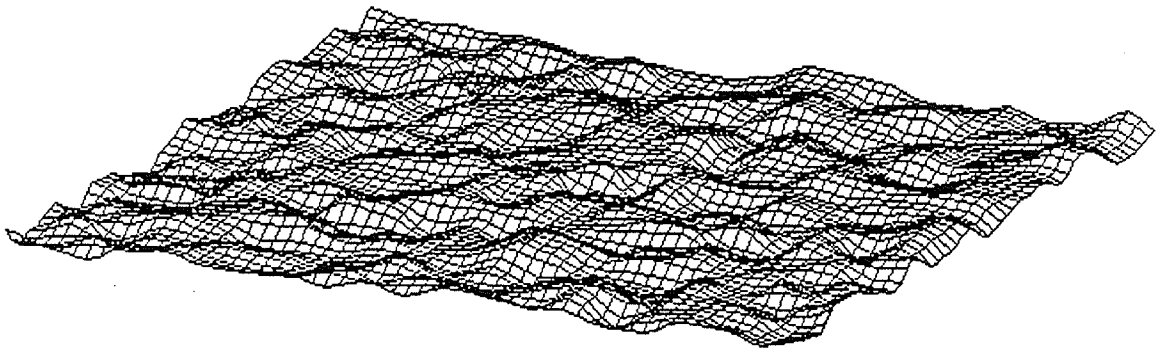

Figure 3.13 Two-dimensional slice of the noise function

Figure 3.13 shows a slice of the noise function at a z value of 1.

The noise function may be used in many ways to alter the appearance of an object's surface. For example, a blotchy looking texture may be created by performing a cubic interpolation between the original surface color and a specified texture color based on the value returned by the noise function at that surface point. Many of the textures mentioned in [8] have been implemented. Chapter 7 describes the textures that are implemented within the program.

3.6 Transformations

Transformations are used to size, place and orient both objects and primitives within the scene being modelled. They are also used to transform textures which are applied to surfaces during rendering. The use of transformations to model objects is thoroughly covered in chapter 7. This section will briefly present the transformations that are used. More complete discussions of transformations may be found in [4,7].

All transformations are represented by a 4 by 4 matrix using homogeneous coordinates. Within the program all points are explicitly represented in three-dimensional space as (x, y, z) with the implicit four-dimensional representation (x', y', z', h) where h is always 1. The transformation from homogeneous coordinates to ordinary coordinates is given by

$$[\, x'' \;\; y'' \;\; z'' \;\; 1 \,] = [\, x'/h \;\; y'/h \;\; z'/h \;\; 1 \,]$$

A point P is transformed to a point Q by some transformation matrix M as follows:

$$[\, Q_x \;\; Q_y \;\; Q_z \;\; h \,] = [\, P_x \;\; P_y \;\; P_z \;\; 1 \,]\,[\, M \,]$$

The program makes use of the following transformations: translation, scaling, rotation and reflection. Each of the individual transformation matrices will be presented.

3.6.1 Translation

A linear translation matrix is used to reposition a point to a new location in space. The matrix is as follows:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{vmatrix}$$

3.6.2 Scaling and Reflection

A scaling matrix is used to individually rescale each component of a point or vector. When scaling a vector its origin is not affected however its direction will change. The matrix is:

$$\begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

If a negative value is supplied for any of s_x, s_y or s_z then a reflection through a plane into another octant will occur. For example, a value of -1 for s_x would cause a reflection through the YZ plane.

3.6.3 Rotations

Rotations are used to reorient points in space. Positive rotations are in a counter-clockwise direction as one looks downward along any positive axis. For example a rotation about the X axis would rotate the positive Y axis towards the positive Z axis.

The matrix for a positive rotation about the X axis by an angle α is as follows:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

The matrix for a positive rotation about the Y axis by an angle β is as follows:

$$\begin{vmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

The matrix for a positive rotation about the X axis by an angle γ is as follows:

$$\begin{vmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

3.6.4 Multiple Transformations

A single transformation matrix may be composed of any number of scalings, reflections, translations and rotations combined in any order. If one applied each successive transformation individually to a point as they were specified they would be combined in the following manner:

$$Q = P\ M_1\ M_2\ M_3\ M_4 \ldots$$

The matrix nearest *P* generates the first transformation and the transformation farthest to the right, the last. Multiple transformations are combined in a pre-multiplied matrix M in the following order:

$$M = M_1\ (M_2\ (M_3\ M_4)))$$

Since combined transformations are obtained with matrix muliplication which is noncommutative, the order the transformations are specified in will affect the final result.

4 Mirage Ray Tracing in Three Dimensions

To account for atmospheric refraction while rendering the environment, the two dimensional mirage model presented in chapter 1 is extended to three dimensions. It is assumed that atmospheric refraction will only affect those rays arriving at an observer's eye and not light incident on object surfaces. That is, the initial rays that are traced from an observer to the first object hit are bent. Any shadow or reflected rays generated at that object's surface are subsequently traced along a straight path. This chapter will discuss the three dimensional mirage model and its incorporation into the ray tracing program.

4.1 Three Dimensional Mirage Model

In extending the two dimensional mirage model to three dimensions it is assumed that the atmosphere is laterally homogenous over the ranges of interest so that a given temperature profile will prevail everwhere within an observer's field of view. This will allow ray bending to be confined to a vertical plane in any radial direction from the observer and as a result no lateral object distortions will be present. It is further assumed that the observer is positioned somewhere within the temperature layers (a requirement of the two dimensional model) and that the field of view is small enough so as to not introduce severe perspective distortions. Given these assumptions, rays are traced radially outward from an observer always confined to vertical planes as shown in figure 4.1. Ray calculations are performed in a two dimensional coordinate system within the plane and are transformed back into the scene's three dimensional system using rotations and translations.

4.2 Ray Tracing with Atmospheric Refraction

This section will discuss how the mirage model is implemented within a conventional ray tracing program.

The curved rays originate from an observer at a point *O*, are intially projected in a direction **D** and propagate within a plane whose normal $\mathbf{N}_p$ is given by

$$\mathbf{N}_p = \mathbf{D} \times \mathbf{Z}$$

as shown in figure 4.2. The initial direction **D** is determined by the viewing geometry in a manner identical to that used for straight rays. The radial direction

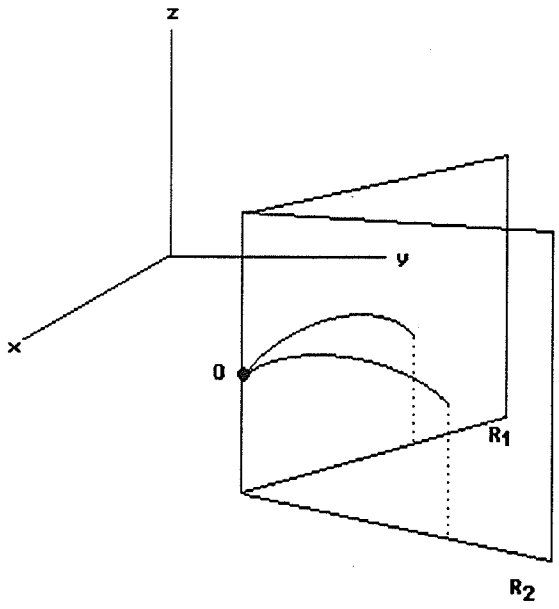

Figure 4.1 Parabolic rays traced within radial planes

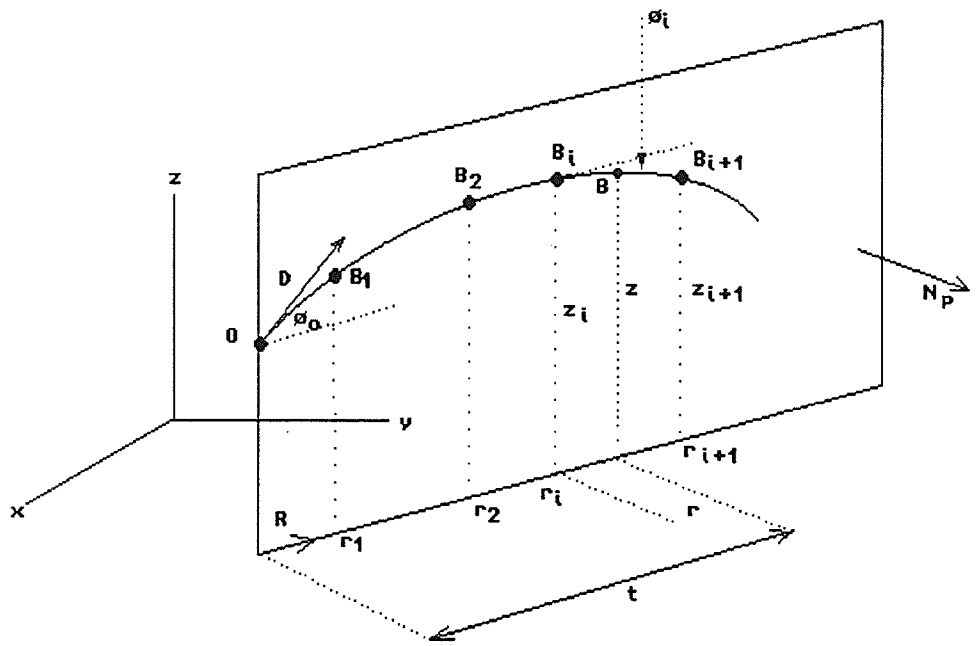

Figure 4.2 Two-dimensional coordinate system of a parabolic ray within the scene's three-dimensional coordinate system

R is the normalized projection of the initial ray direction **D** onto the XY plane. The z coordinate of any point along a parabolic ray is the same in both its two dimensional coordinate system and the scene's three dimensional coordinate system. The mirage algorithm requires the initial height of the observer z_0 and the initial angle ϕ_0 the ray makes with a horizontal plane. The initial height is determined by the z component of the observer's origin, that is O_z, and the initial ray angle is given by

$$\phi_0 = \pi / 2 - \cos^{-1} (\mathbf{Z} \cdot \mathbf{D})$$

where the vector **Z** represents the Z axis in normalized form. If the direction **D** is normalized then the dot product may be replaced by $\mathbf{D}_z$.The mirage algorithm also requires a maximum layer elevation above which curved rays are no longer traced and a maximum horizontal distance beyond which curved rays are no longer traced. These quantities are user supplied. The algorithm steps along the horizontal direction within the plane calculating layer boundary intersection points. Between any two layer boundary intersection points B_i and B_{i+1} a ray segment may be described by

$$z = A_p r^2 + B_p r + C_p \qquad (4.1)$$

where

$$A_p = - \kappa_{eff} / 2$$
$$B_p = \tan \phi_i$$
$$C_p = z_i$$

and where the ray is defined within the interval r_i and r_{i+1}. The quantity κ_{eff} is an effective ray curvature which accounts for the curvature of the earth allowing the atmospheric layers and the earth to be modelled as horizontal planes. It is given by

$$\kappa_{eff} = \kappa_{ray} - \kappa_{earth}$$

These quantities are all expressed in the two dimensional coordinate system of the parabolic ray. The distance r of any point along the parabolic ray within an interval is expressed relative to the origin r_i . Its distance t from the observer along the radial direction **R** is given by

$$t = r_i + r.$$

To express the parabolic ray's r coordinate as scene XY coordinates we make use of the vector **R** as follows:

$$x = O_x + t\,\mathbf{R}_x$$

$$y = O_y + t\,\mathbf{R}_y$$

In order to calculate intersections with the *ith* parabolic ray segment, the following information is associated with each ray: A_p, B_p, C_p, κ_{eff}, r_i and r_{i+1}. These quantities are provided by the mirage algorithm. Within each interval, intersection tests are performed with all objects in the scene. If an object is not intersected within the current interval then a new parabolic ray is generated for the interval r_{i+1} to r_{i+2} and tests are again carried out with all objects in the scene.

To test whether or not an object is hit, each of its primitives are tested individually. For each primitive it is first determined whether or not the ray's plane slices that primitive. If it does, then a two dimensional outline of the primitive will be formed upon the ray's plane at the point of intersection. If the ray's plane cuts a sphere, then a circle will be formed and if a planar shape is cut, a line will be formed. The intersection tests are thus reduced to a two dimensional problem where the parabolic ray may possibly intersect either a circle or a line.

If an object is hit within a parabolic ray's interval, the intersection point *I* is expressed within the scene's coordinate system as:

$$I_x = O_x + t\ \mathbf{R}_x$$

$$I_y = O_y + t\ \mathbf{R}_y$$

$$I_z = A_p\ r^2 + B_p\ r + C_p$$

where $r = t - r_i$. The vector **V** representing the ray incident to the objects's surface at *I*, is determined by a tangent vector to the parabolic ray at the intersection point and is given by

$$\mathbf{V}_x = \mathbf{R}_x$$

$$\mathbf{V}_y = \mathbf{R}_y$$

$$\mathbf{V}_z = 2\,A_p\,r + B_p$$

where $r = t - r_i$. In calculating the z component of both the intersection point *I* and

the incident ray **V**, the radial distance r must be expressed relative to the parabolic ray's origin r_i. This is required since the parabolic ray that does intersect an object is only defined within the interval $\{ r_i .. r_{i+1} \}$.

When no objects are hit within the current interval a new parabolic ray is generated by the mirage algorithm. Before doing so, certain criteria is evaluated to determine whether or not the calculations should be terminated. These terminating conditions were discussed in Chapter 2. If a terminating condition does occur, then a straight ray is subsequently traced. The straight ray's origin *O'* is positioned at the end of the last parabolic ray interval and is expressed in the scene's three dimensional coordinate system as:

$$O_x' = O_x + r_{i+1} \mathbf{R}_x$$

$$O_y' = O_y + r_{i+1} \mathbf{R}_z$$

$$O_z' = z_{i+1}$$

The new direction **D'** of the straight ray is determined by the tangent vector to the parabolic ray at *O'* and is given by

$$\mathbf{D}_x' = \mathbf{R}_x$$

$$\mathbf{D}_y' = \mathbf{R}_y$$

$$\mathbf{D}_z' = 2 A_p (r_i - r_{i+1}) + B_p$$

The direction vector **D'** is then normalized and a straight ray is traced from that point onward.

A flow chart describing the mirage tracing implementation is shown in figure 4.3. The flow chart is a modification of the one provided in [3].

There are two special cases which may occur, both of which cannot be handled by the mirage algorithm. If the viewing geometry is such that the initial ray direction **D** is nearly vertical, that is almost aligned with the +/- Z axis, then the mirage algorithm is not called and a straight ray is traced instead. If the observer's position does not fall within any temperature layer then the mirage algorithm cannot be initialized properly and the program issues an error.

One may notice that along any image row, the rays that are traced within the vertical planes are identical. As far as the mirage algorithm is concerned the rays

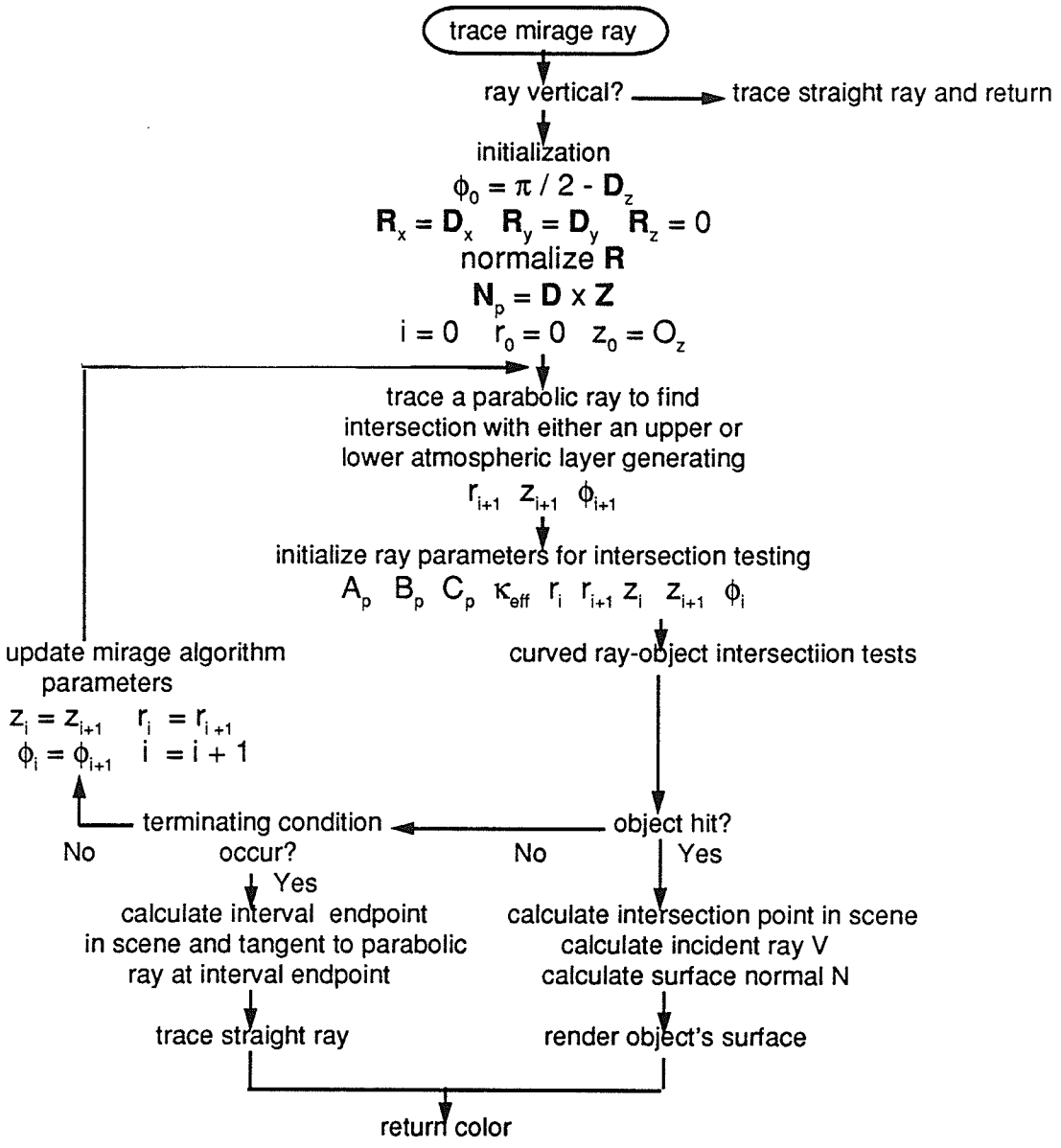

Figure 4.3 Flow chart of mirage tracing algorithm

all start at the same height O_z with the same trajectory angle ϕ_0. As the program considers each horizontal image pixel the mirage algorithm is called to perform the same calculations all over again. Another possible approach to the problem would of been to pre-calculate all the possible parabolic ray segments for a given row storing the quantities A_p, B_p, C_p, κ_{eff}, r_i and r_{i+1} for each ray segment. In addition, the radial direction **R** would have to be maintained for each vertical plane considered. For any given pixel on that row the intersection tests would be performed as previously discussed. This approach has been tested and it was observed that image generation times were reduced significantly however it has not been implemented for three reasons. As implemented, the mirage algorithm is very independent from the main program and with very minor modifications can be used to account for atmospheric refraction along a reflected ray's path. Since a reflected ray's initial trajectory can be in any direction the just described method is not applicable. Secondly, it was desired to leave open the possibility of implementing any anti-aliasing method [4,5]. Anti-aliasing is a method whereby image pixels are sub-sampled to produce additional representative colors for a given pixel. These extra colors are subsequently filtered to produce a "net" color for a pixel. The effects produced by anti-aliasing are very noticeable in low resolution images where the familiar staircase edges are replaced by a blurring of colors. In implementing anti-aliasing one may either sub-sample a pixel cell uniformly by subdividing the pixel into a finer grid of sub-pixels or one may randomly sample the pixel's area. If the later method is used, then it is not known beforehand which direction a ray will initially be shot. In this case the mirage algorithm would be supplied an initial trajectory angle ϕ_0 that is randomly determined and the above mentioned approach would not be applicable. Lastly it is desired to leave open the possibility of adding lateral temperature profiles in which case each radial plane would have to be considered.

4.3 Ray-Object Intersections with Parabolic Rays

This section will discuss the calculations required to determine whether or not objects have been intersected by the parabolic rays resulting from atmospheric refraction.

4.3.1 Sphere Intersection

One may observe that if the parabolic ray is to intersect the sphere then its containing plane must also intersect the sphere. The plane will cut through the sphere if the perpendicular distance from the plane to the sphere's center *P* is less than the sphere's radius r_s. The geometry of the problem as viewed on the horizontal plane containing the sphere's center is shown in figure 4.4.

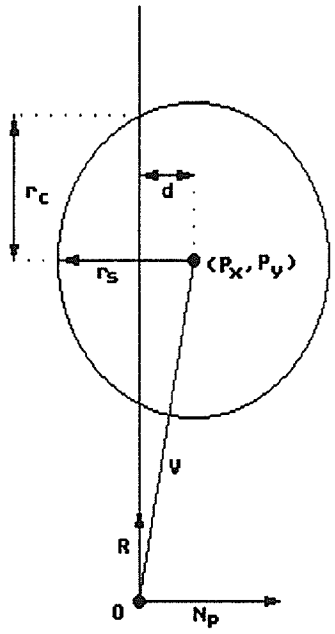

Figure 4.4 Plane of parabolic ray slicing a sphere

In performing this test we determine the radius r_c of the sphere's projected circle on the plane as

$$r_c = r_s^2 - d^2$$

where

$$d = \mathbf{N}_p \cdot \mathbf{V} \; / \; | \mathbf{N}_p |$$

and

$$\mathbf{V} = P - O$$

If the projected circle's radius r_c is greater than zero then the sphere will be cut by the plane. If the sphere is cut, then its projected center C is expressed in the ray's coordinate system as follows:

$$C_z = P_z$$

$$C_r = (P_x - O_x) \; \mathbf{R}_y + (P_y - O_y) \; \mathbf{R}_x$$

Both the projected circle center and radius may be pre-calculated and stored prior to calling the mirage algorithm. Once the projected circle's coordinates are known , a test to determine whether or not the circle partially or fully lies within the interval of the current parabolic ray is performed. If any portion of the circle lies within the interval then the circle may possibly be intersected. If this is the case, then the circle is first expressed in the form:

$$(r - C_r')^2 + (z - C_z)^2 - r_c^2 = 0 \qquad (4.2)$$

where C_r' is the r coordinate of the circle origin expressed relative to the start of the parabolic ray interval r_i, that is,

$$C_r' = C_r - r_i$$

At the point of intersection the z coordinates of the parabolic ray and the circle will be equal. Substituting (4.1) into (4.2) for z and simplifying, the following quartic expression is obtained:

$$A_q r^4 + B_q r^3 + C_q r^2 + D_q r + E_q = 0 \qquad (4.3)$$

where

$$A_q = A_p^{\,2}$$

$$B_q = 2\,A_p\,B_p$$

$$C_q = 2\,A_p\,(C_p - C_z) + B_p^{\,2} + 1$$

$$D_q = 2\,B_p\,(C_p - C_z) - 2\,C_r'$$

$$E_q = (C_p - C_z)^2 + C_r'^{\,2} - r_c^{\,2}$$

This equation is then solved for a positive real root using a hybrid algorithm that combines the Newton-Raphson and bisection methods [11]. The root finder requires the above equation, its first derivative and an interval in which to search. If a root can be found then its value will represent the distance from r_i at which the ray intersects the circle. If the origin of the parabolic ray lies outside of the circle then a search is performed within the interval $\{ r_i \,..C_r' \}$. If a root cannot be found within this range or if the origin of the parabolic ray lies inside the circle then a search is performed within the interval $\{ r_i \,..\, r_{i+1} \}$. If a root cannot be found within either range then the sphere is rejected. If a root is found, then a distance t expressed relative to the observer as

$$t = r_i + r\,,$$

is subsequently compared to t_{near}. If the value of t is less than t_{near} then the sphere in question is considered the closest primitive hit thus far and t_{near} is reassigned the value t (note that the value of t is along the projected direction **R** not along the initial ray direction **D**).

The intersection point *I* as expressed within the scene's coordinate system is given by:

$$I_x = O_x + t\ \mathbf{R}_x$$

$$I_y = O_y + t\ \mathbf{R}_z$$

$$I_z = A_p\ (t - r_i)^2 + B_p\ (t - r_i) + C_p$$

and the sphere's surface normal is thus given by

$$\mathbf{N} = I - P.$$

4.3.2 Planar Intersections

Each of the plane, triangle and rectangle primitives share a common intersection test as they all require a defining plane. Both the triangle and rectangle will require further consideration however this test is always applied first. Each of these primitives is again defined by a point *P* lying within a plane whose normal **N** is given by

$$\mathbf{N} = \mathbf{U} \times \mathbf{V}$$

as previously shown in figure 3.11. One may observe that if any parabolic ray segment is to intersect a planar primitive then its containing plane must also intersect the primitive. We may easily reject the primitive if its defining plane lies parallel to the plane containing the parabolic ray, that is, if

$$\mathbf{N}_p \cdot \mathbf{N} = 1.$$

One may also observe that a triangle or rectangle will not be intersected by any ray segment if all of the vertices of either primitive lie to one side of the plane containing the parabolic ray. For any vertex V_i ,the following value is determined:

$$s_i = \text{sign}\{ \mathbf{N}_p \cdot (V_i - O) \}$$

where s_i may be -1, 0 or 1. A triangle with vertices V_1, V_2 and V_3 is rejected if the following conditions are met:

$$s_1 = s_2 \quad s_1 = s_3 \quad s_2 = s_3$$

A rectangle with vertices V_1, V_2, V_3 and V_4 is rejected if the following conditions are met:

$$s_1 = s_3 \quad s_2 = s_4$$

These tests are performed prior to calling the mirage algorithm so that a rejected primitive will not be considered when tracing ray segments within the current radial plane.

At this point one may also observe that a triangle or rectangle will not be intersected by a ray if each of its projected vertices fall either before r_i or after r_{i+1} as shown in figure 4.5 for the case of a triangle. The projected vertices may be pre-calculated and stored prior to calling the mirage algorithm. The projected vertices are subsequently compared against each end point of an interval as new

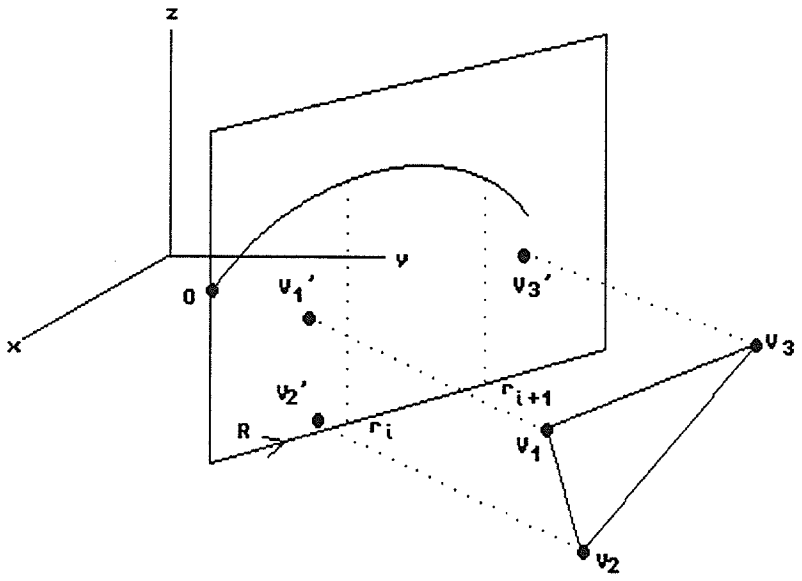

Figure 4.5 Test of projected triangle vertices

parabolic rays are tested.

If a primitive is not rejected we proceed to determine the equation of the line formed by the intersection of the two planes. The line equation is expressed in terms of the parabolic ray's coordinate system. A primitive's plane equation expressed relative to the parabolic ray's origin r_i is first determined as

$$A_n\ r + B_n\ y + C_n\ z + D_n = 0 \tag{4.4}$$

where

$$A_n = \mathbf{N}_x\ \mathbf{R}_x + \mathbf{N}_y\ \mathbf{R}_y$$
$$B_n = -\mathbf{N}_x\ \mathbf{R}_y + \mathbf{N}_y\ \mathbf{R}_x$$
$$C_n = \mathbf{N}_z$$
$$D_n = -\left(A_n\ \left(P_r' - r_i \right) + B_n\ P_y' + C_n\ P_z' \right)$$

and where the point P' given by

$$P_r' = \left(P_x - O_x \right)\ \mathbf{R}_x + \left(P_y - O_y \right)\ \mathbf{R}_y$$
$$P_y' = -\left(P_x - O_x \right)\ \mathbf{R}_y + \left(P_y - O_y \right)\ \mathbf{R}_x$$
$$P_z' = P_z$$

represents the pre-calculated transformed point P. The quantities A_n and $B_n\ P_y' + C_n\ P_z'$ are also pre-calculated and stored prior to calling the mirage algorithm.

The two dimensional equation of the line lying within the parabolic ray's plane is then determined by setting the y values in (4.4) to 0, and is given by

$$A_n\ r + C_n\ z + D_n = 0 \tag{4.5}$$

At the point of intersection the z coordinates of the parabolic ray and the line will be equal. Substituting (4.1) into (4.5) for z and simplifying the following quadratic expression is obtained

$$A_q\ r^2 + B_q\ r + C_q \tag{4.6}$$

where

$$A_q = A_p\ C_n$$
$$B_q = B_p\ C_n + A_n$$
$$C_q = C_p\ C_n + D_n$$

The quadratic is then solved for the smallest positive root which represents the distance from r_i at which the ray intersects the line. A negative root implies that the intersection has occured behind the ray's origin. If a positive root can not be found then the planar primitive is rejected. If a positive root is found, then a distance t expressed relative to the observer as

$$t = r_i + r,$$

is subsequently compared to t_{near}. If the value of t is not less than the current value of t_{near} then no further tests are performed and the primitive is rejected. If t is less than t_{near}, and the primitive in question is a plane, then the plane is considered the closest primitive hit thus far (again note that the value of t is along the projected direction **R** and not along the initial ray direction **D**). If the primitive is either a triangle or a rectangle then additional tests are required to determine whether or not the intersection point is contained within the primitive's interior. The intersection point *I* for any planar primitive as expressed within the scene's coordinate system is given by

$$I_x = O_x + t\ \mathbf{R}_x$$

$$I_y = O_y + t\ \mathbf{R}_y$$

$$I_z = A_p\ (t - r_i)^2 + B_p\ (t - r_i) + C_p$$

4.3.3 Triangle Intersection

Given that the parabolic ray has intersected the plane containing a triangle at a "best" distance t along **R**, it still has to be determined whether or not the intersection point *I* lies within that triangle's interior. The remaining steps in the test are very similar to those used for straight rays as discussed in section 3.3.5. Again, any point *I* within the triangle is given by

$$I = P + u\ \mathbf{U} + v\ \mathbf{V}$$

for u { 0..1 } and v { 0..1 } and u + v <= 1. Once again it is to be determined whether or not the quantities u and v are within their valid range at the intersection point *I*. By first selecting any point *Q* not on the triangle's plane and then forming a test ray **T** = *I* - *Q*, the quantities u and v are expressed as follows:

$$u = - \mathbf{T} \cdot (\mathbf{W} \times \mathbf{V}) / (\mathbf{T} \cdot \mathbf{N})$$

$$v = - \mathbf{T} \cdot (\mathbf{U} \times \mathbf{W}) / (\mathbf{T} \cdot \mathbf{N})$$

where $\mathbf{W} = P - Q$.

If the intersection point *I* is within the interior of the triangle, then the quantities u and v must be in the range { 0 ..1 } and their sum must be less than or equal to 1. If the conditions for u and v are met then the intersection point lies within the triangle's interior and the primitive is considered the closest hit thus far (we already know it is the closest as a result of the preceding plane test).

4.3.4 Rectangle Intersection

Given that the parabolic ray has intersected the plane containing a rectangle at a "best" distance t along **R**, it still has to be determined whether or not the intersection point *I* lies within that rectangle's interior. The required test is similar to that of the triangle, the only difference being that the sum of u and v must be less than or equal to 2. It is sufficient to just check the values of u and v to make sure they both are in the range { 0 ..1 }. If both values are in range, then the intersection point lies within the rectangle's interior and the primitive is considered the closest hit thus far.

5 Implementation

An overview of the hardware and software involved in this thesis is presented in this chapter.

5.1 Hardware

All software development and testing is carried out on two computer systems. The first is a 7.16 MHz 68000 based Amiga 500 with 4.5 megabytes of RAM. Images are displayed on this system using an external 24-bit color adapter. The second system used is a 14.32 MHz 68020 based Amiga 1200 with 2 megabytes of RAM. This system is capable of displaying 24-bit color images without external hardware. Images are generated with 24-bits of color information and are displayed at a screen resolution of 320 x 200 with 256,000 displayable colors out of a palette of 16.8 million. Color output on paper is performed by a local video service. Video output is achieved by recording directly to a VCR using the composite output of the Amiga 1200.

5.2 Software

The software for the ray tracing program is written entirely in C comprising approximately 330 kilobytes of source. Due to the program's size, source code listings have not been included with this document but are available in a separate software supplement. The code is compiled using the SAS/C v6.0 compiler for the Amiga using strict ANSI syntax. The program has been ported to an HP9000 only requiring the removal of Amiga specific display routines and nested comments. The program is divided up into 18 separate files as follows:

1) attr.c

This file contains the routines to manage object attributes. Information associated with object surfaces and lights is processed in this file.

2) display.c

This file is used to display color HAM (hold and modify) images on the Amiga [12]. 24-bit images are quantized to 12-bits and redisplayed using the Floyd-Steinberg dithering algorithm. Color table values are calculated using a modified popularity algorithm that takes advantage of the HAM display mode.

3) defs.h

This include file contains all the structure definitions, defines and prototypes for external functions.

4) main.c

This file contains the command line parser and performs scene initialization.

5) noise.c

This file implements the Perlin noise, turbulence and Dnoise functions.

6) objects.c

This file contains routines to handle solid and light objects.

7) parse.h

This file contains the structure definitions required by the routines that are used to parse the scene description language.

8) parse1.c
9) parse2.c

These two files contain the routines to parse statements in the scene description language.

10) parse3.c

This file performs the tokenizing of the input file, variable declarations, symbol table management and error handling.

11) primitives.c

This file contains the routines to manage the primitives that are supported. This file also contains all the intersection routines for straight and curved rays.

12) render.c

This file contains the routines which perform surface shading.

13) sphere.c

This file contains routines to handle sphere operations.

14) textures.c

This file contains all the routines to produce solid texturing.

15) trace.c

This file contains the routines to trace both straight and curved mirage rays. The implementation of the mirage algorithm is located in this file.

16) transforms.c

This file contains all the routines needed to support transformations and matrix operations.

17) view.c

This file contains the routines to initialize and calculate the viewing geometry.

18) wireframe.c

This file contains routines to support the wireframe display of a scene.

Due to the program's size there are far too many implementation details which could be covered but to provide an indepth discussion of each would require too much space, therefore it will not be attempted. The most important aspect of the program, that is the tracing of mirage rays through a scene and the rendering of visible objects within that scene, has been discussed already in previous chapters. The user manual that is provided in chapter 7 provides a good overview of what the program is capable of and discusses additional topics that have not been covered thus far. The following discussion will give a brief overview of the program's overall flow which is shown in Figure 5.1.

Upon startup the program performs some initializations. There are several structures used to store global information that is shared by more than one program module. The "scene" structure for example contains the maximum ray tree depth. These structures are initialized upon startup. Next any specified command line arguments are parsed. The available options are those that have

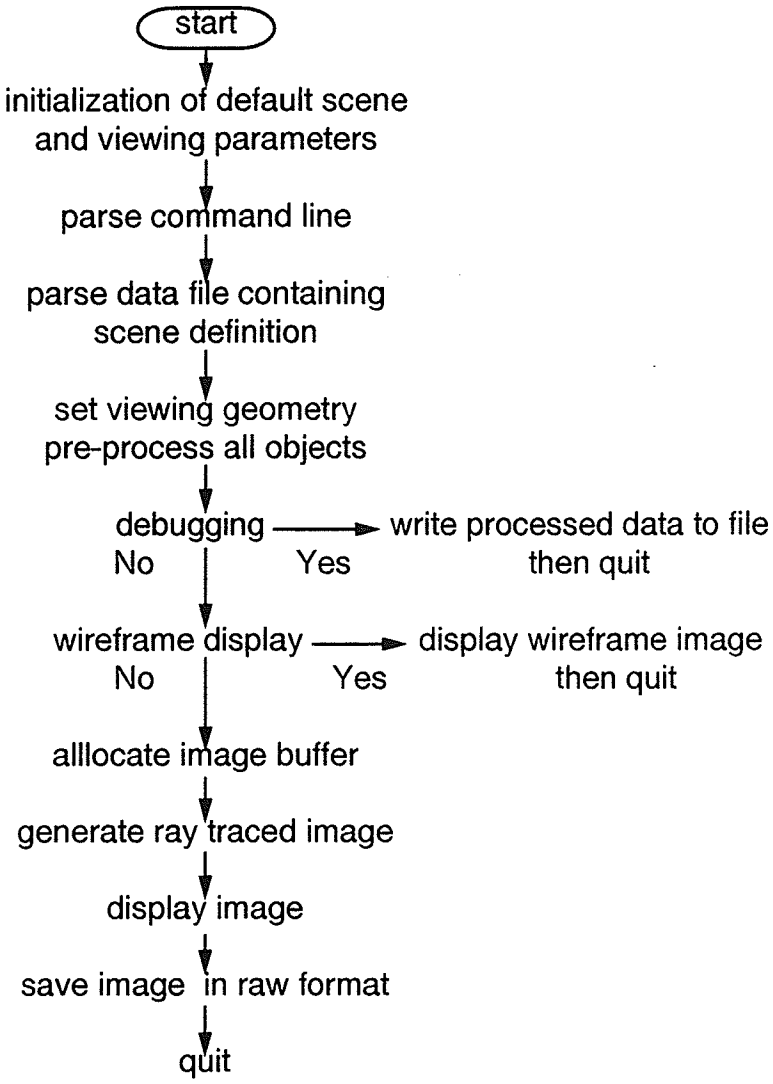

Figure 5.1 Overall program flow

proven to be useful during program testing. Next follows the parsing of the data file. All input to the program is specified through a description language that uses a C like syntax. The language is used to create the scene that is to be rendered and is also used to specify additional information such as image size and viewing parameters. After the input file is parsed all specified information that requires further processing is performed next. For example, all objects have an associated transformation matrix that is used to place that object within the scene. These transformations are performed prior to image generation. For debugging purposes, the program provides the option of dumping out all processed information to a file, bypassing image generation. If the debugging option is specified then the program will write out the information and quit. If the debugging option has not been specified then the program proceeds to check whether the scene is to be displayed as a wireframe image or if a full ray traced image is to be produced. Wireframe displays are used to obtain quick previews of the scene layout prior to ray tracing. If a ray traced image is to be generated then a routine similar to that shown in figure 3.4 is called upon. The main difference being that the program allows the option of accounting for atmospheric refraction. If atmospheric refraction is to be simulated, then the straight ray tracing routine shown in figure 3.4 is replaced by the mirage ray tracing routine shown in figure 4.6. The program allows either algorithm to be used. Once an image is generated it is displayed and then saved.

6 Results

This chapter will illustrate images produced by the program.

6.1 Merman Phenomenon

To test the program we make use of existing data to verify that the results produced are as expected. As a test case, the temperature profile shown in figure 6.1 is used to produce optical distortions that are described in [2]. The profile has a temperature inversion of 7.5 degrees C with a thermocline at an elevation of 2.2 m. To produce the most distortion, the observer is positioned slightly below the thermocline at an elevation of 2.1 m. The light rays that will enter the observer's eye will follow paths similar to those shown in figure 6.2. One can observe that light rays have a strong downward curvature with some of the rays crossing other rays at distances of 1 to 1.5 km. The most severe object distortion will occur at approximately 1.4 km where objects may appear stretched, compressed, inverted or vertically displaced. If objects are placed at 1.4 km then the type of distortion seen will depend on the object's size and its elevation.

A test scene is set up containing several objects of varying size positioned 1.4 km from the observer. Each object is positioned at a different elevation. The data file for the scene is listed in Appendix C. Figure 6.3 shows a conventional ray traced image of the scene. This image was generated in approximately 1/2 hour. Figure 6.4 shows the resulting image that is produced when atmospheric refraction is accounted for. The image shows some of the possible variations in image distortion that can occur. This image was generated in approximately 4 1/2 hours. Note that there are no lateral distortions present in the image.

One can immediately observe the vertical displacement of the ground in the distance forming a wall or cliff that appears to be folding back towards the observer. Both the dark blue sphere in the lower left and the rectangle show vertical stretching with parts of each object being inverted. The rectangle's bottom is positioned at a height of 0 m and its vertical length is 1 m. In the midsection of the rectangle the color gradient is going in the opposite direction to that of the original image. One can also observe that the specular highlights on the dark blue sphere have been inverted and then replicated at a higher vertical position. Both the purple sphere in the top left and the mirrored sphere in the top right have been compressed. The mirrored sphere on the right does not show that much distortion. It is positioned at an elevation of 4 m and from the ray plots in figure 6.2 it can be seen that the rays at that elevation are not affected that much. The rainbow colored sphere in the lower right appears to be distorted the

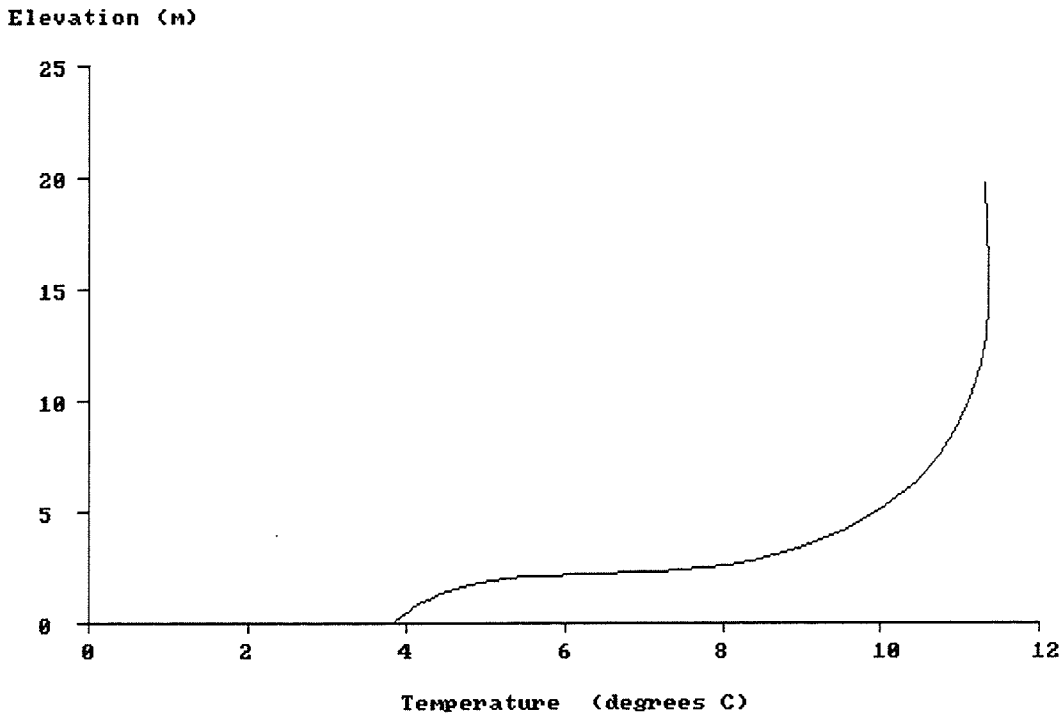

Figure 6.1 Merman temperature profile

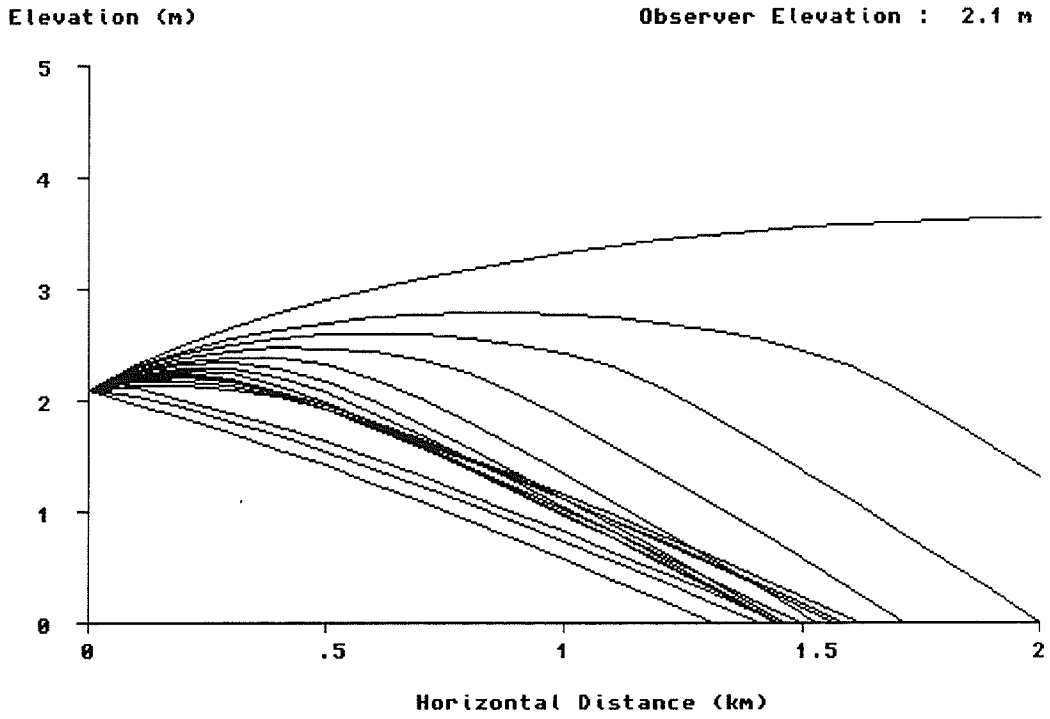

Figure 6.2 Ray plots for merman temperature profile

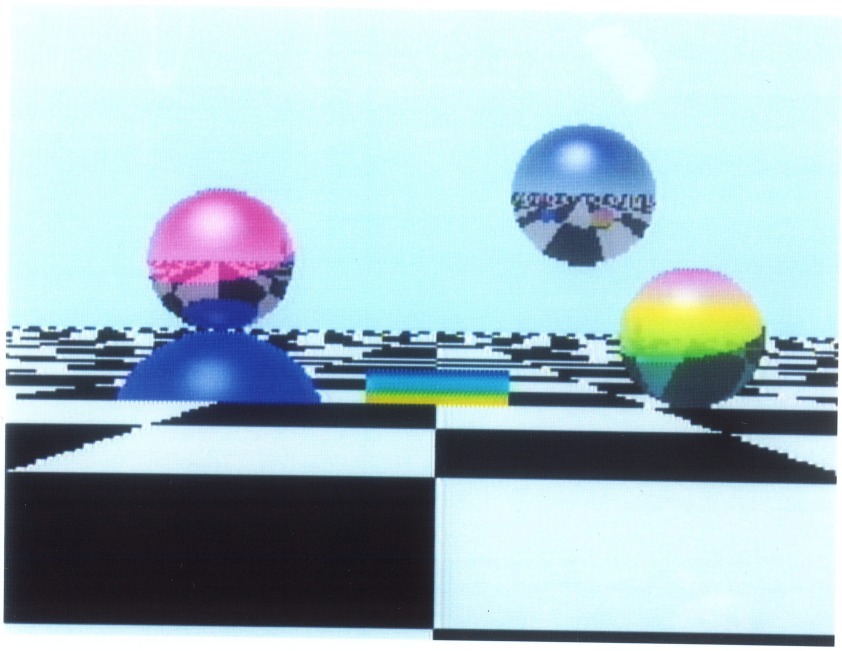

Figure 6.3 Merman test image with no atmospheric refraction

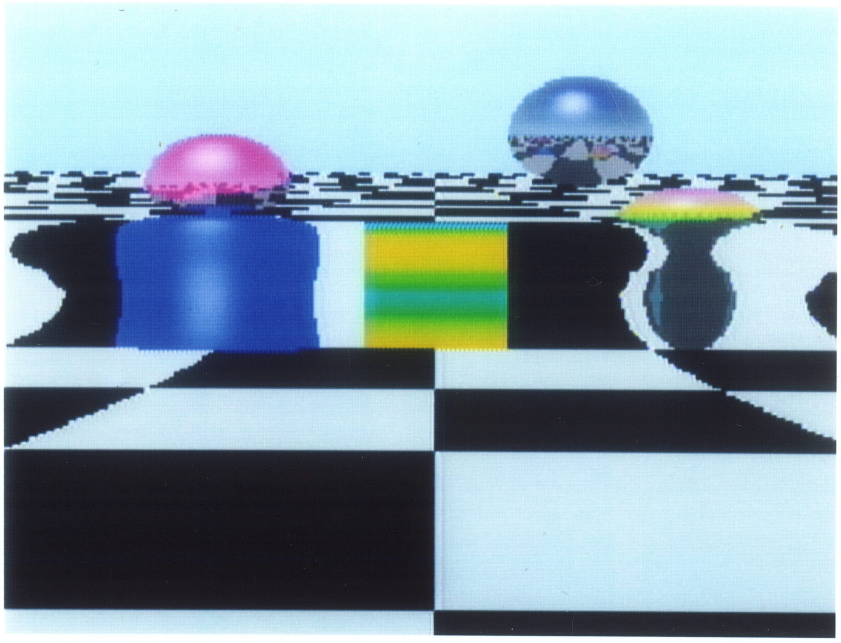

Figure 6.4 Merman test image with atmospheric refraction

most. The sphere is positioned at a height of 2 m and has a radius of 2 m. It is hard to tell whether or not any parts of the sphere are inverted as the lower hemisphere is in shadow, however the sphere does have a characteristic mushroom shape.

One final observation is that the reflected objects in the mirrored sphere at the top right have not been affected at all. This is only due to the fact that the program does account for atmospheric refraction when tracing specularly reflected rays.

6.2 Runway Scene

Sample images are generated to show the apparent distortion of a runway scene at different points along a plane's approach path. The runway environment is very similar to that used in [3] (Appendix C lists the data file used for the scene) . Figure 6.5 shows the layout of the scene (not to scale) where the center of the runway is taken to be the origin of the scene. Figure 6.6 shows a wireframe display of the scene looking from above. Figure 6.7 shows the approach path of the aircraft (observer). The observer starts at an elevation of 8m, 3.4 km from the runway's center and touches down at slightly less than 1 km from the runway's center. The temperature profile used for this scene is shown in figure 6.8. The profile is similar to the one used in the previous example however it is shifted up vertically by 4 m so that the observer will pass through the thermocline (now at 6.2 m) while descending. Figures 6.9-6.14 show the ray plots and corresponding observer's view at different points along the approach path. Each image took approximately 2 1/2 hours to generate on an HP9000. One can see the various object distortions as the observer descends through the thermocline. Above the thermocline there is very little object distortion. As the observer approaches the thermocline one can notice the vertical stretching and displacement of objects. At an observer elevation of 6.2 m one can see the ground rising in the distance with objects appearing displaced, stretched and inverted. As the observer continues to descend light rays are starting to show an upward curvature and in the distance one can see the familar "puddle on the road" effect.

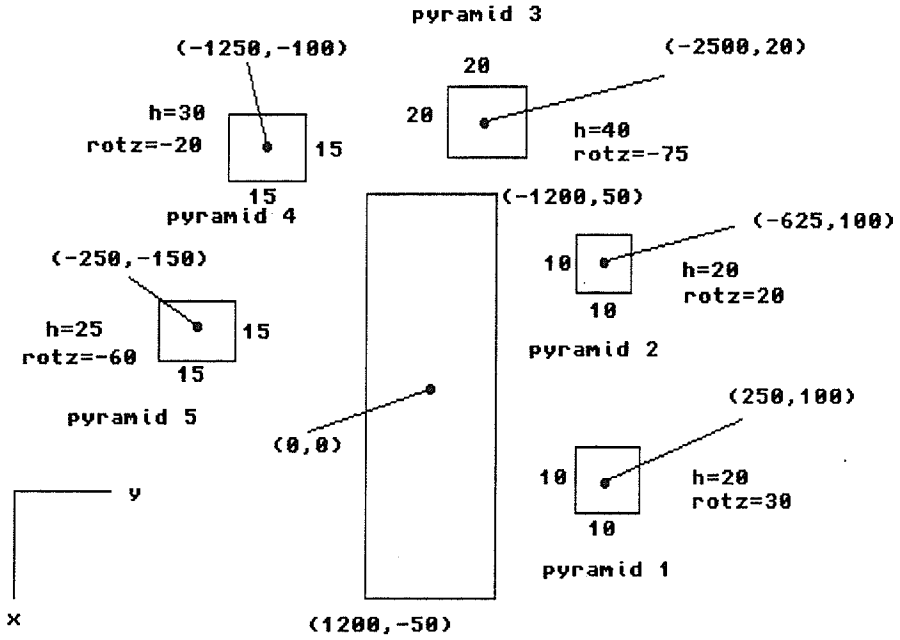

Figure 6.5 Layout of runway scene

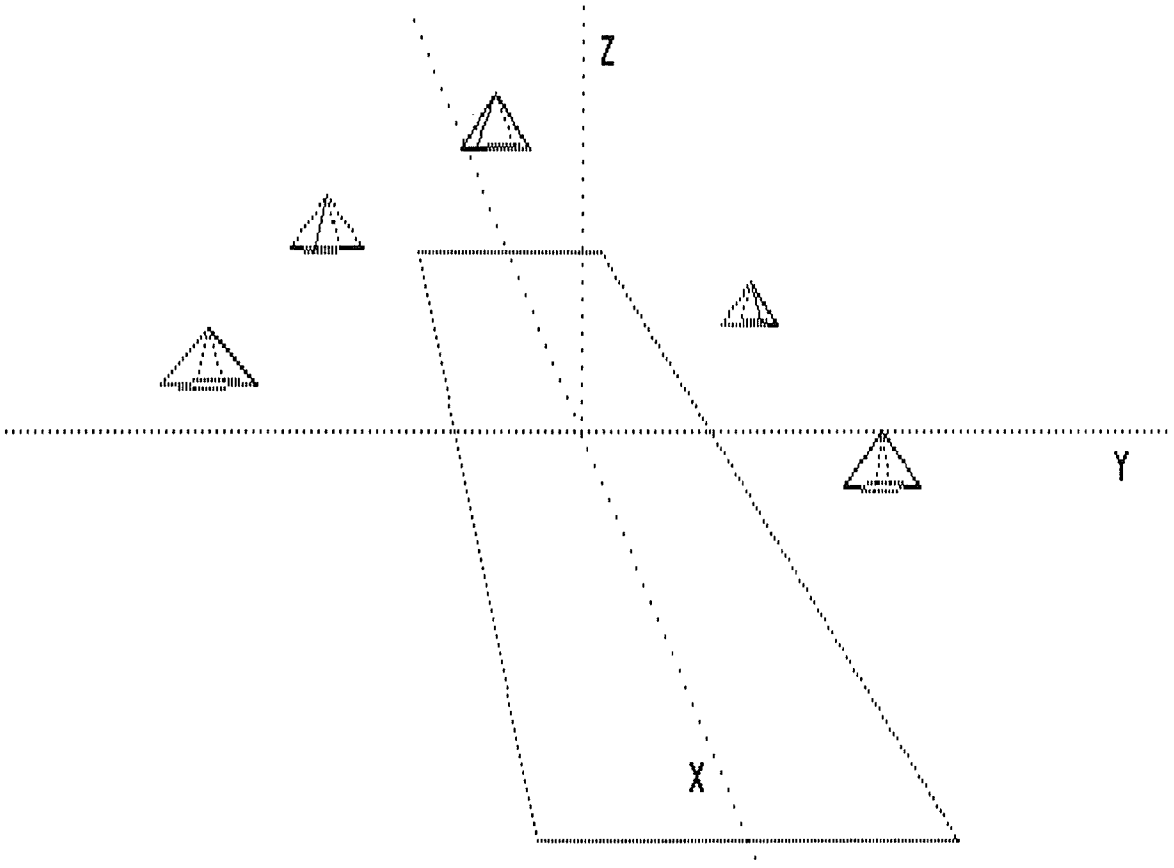

Figure 6.6 Wireframe view of runway scene

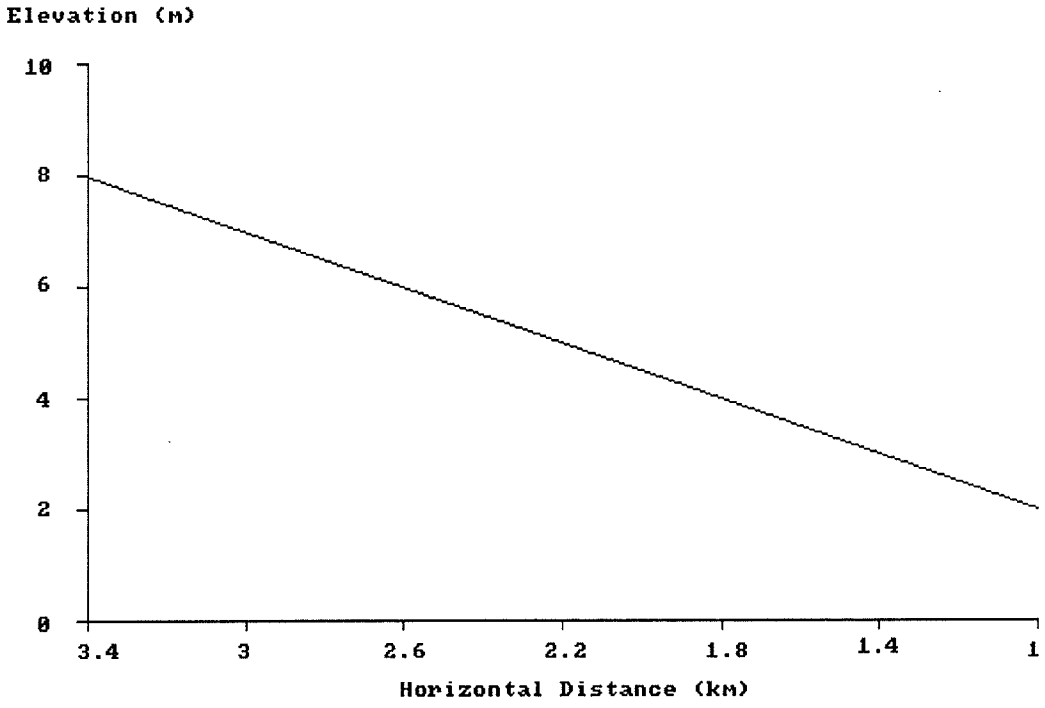

Figure 6.7 Approach path of observer

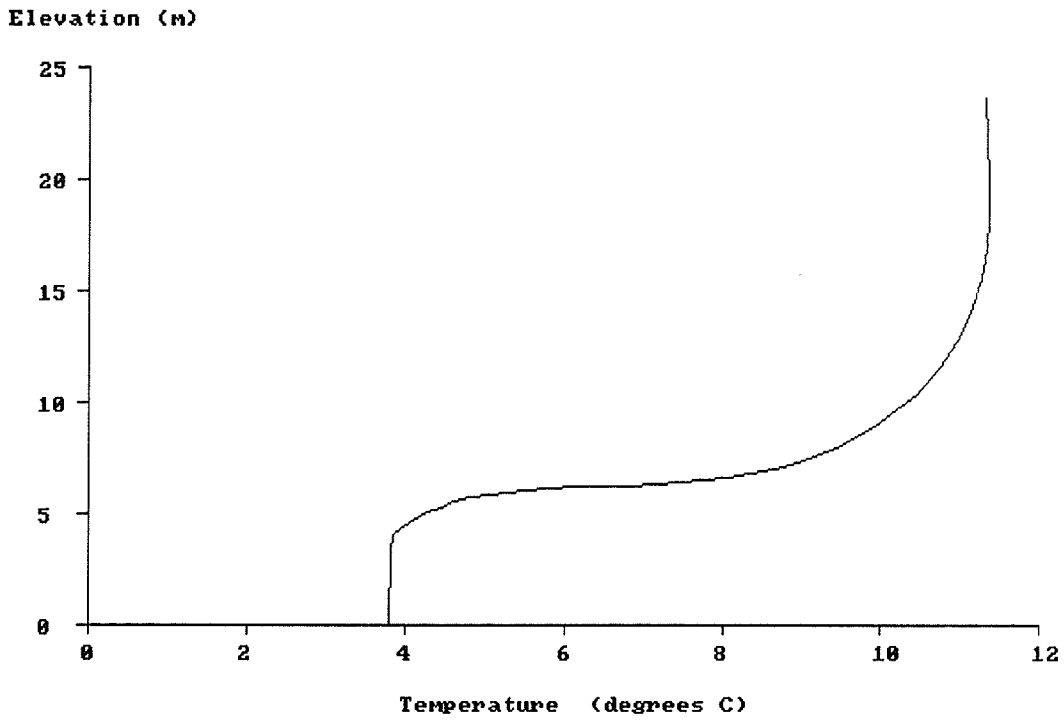

Figure 6.8 Temperature profile used for the runway scene

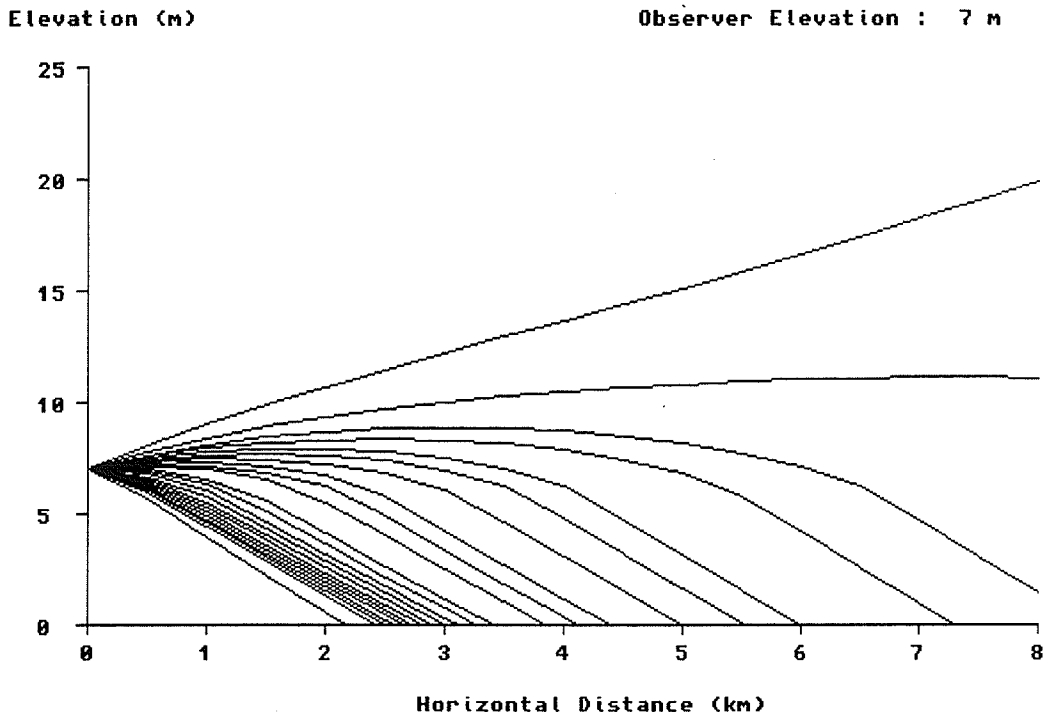

Figure 6.9a Ray plots for runway scene at 3.0 km

Figure 6.9b Observer's view of the scene at 3.0 km

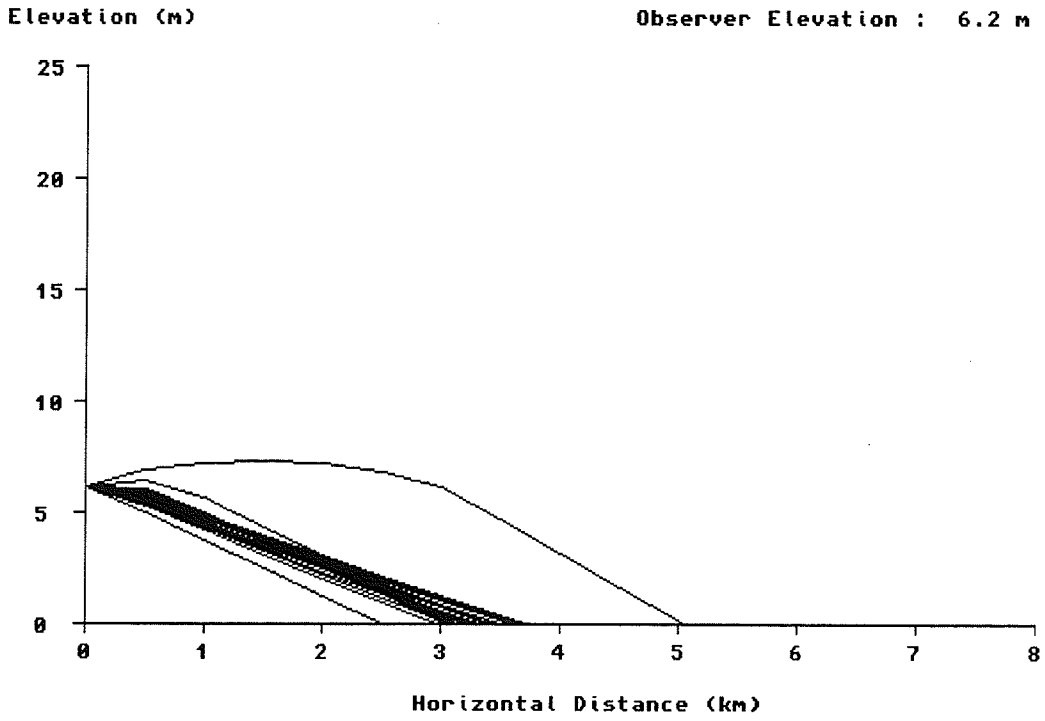

Figure 6.10a Ray plots for runway scene at 2.68 km

Figure 6.10b Observer's view of the scene at 2.68 km

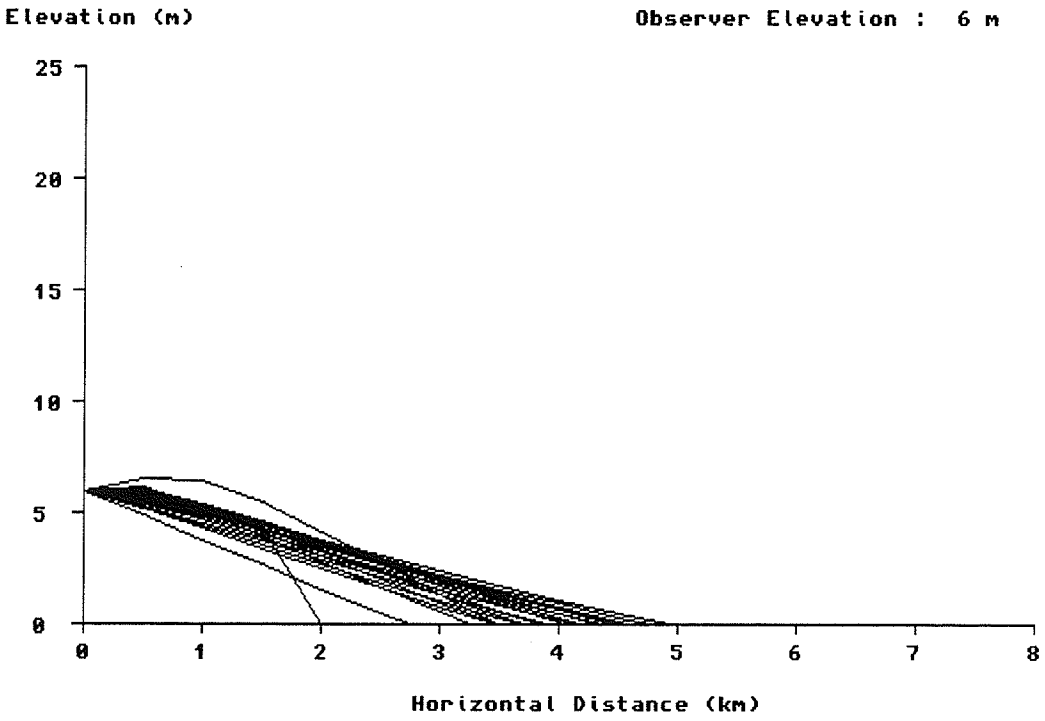

Figure 6.11a Ray plots for runway scene at 2.6 km

Figure 6.11b Observer's view of the scene at 2.6 km

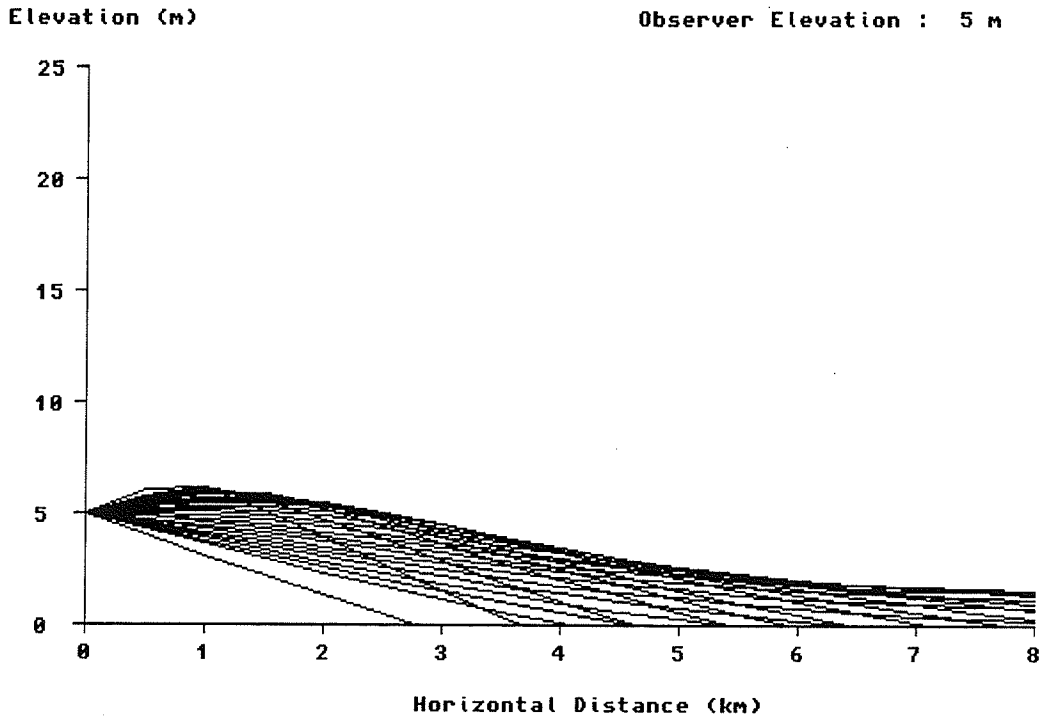

Figure 6.12a Ray plots for runway scene at 2.2 km

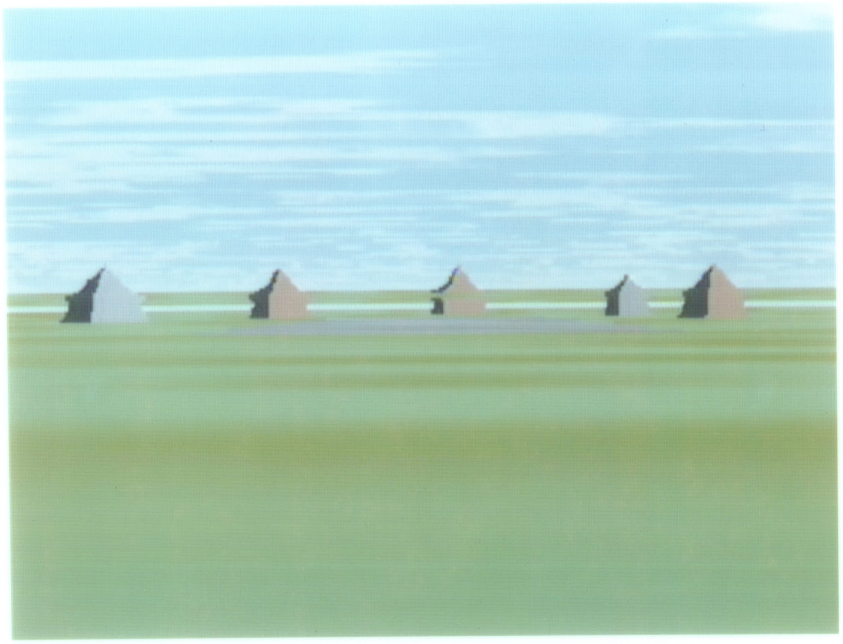

Figure 6.12b Observer's view of the scene at 2.2 km

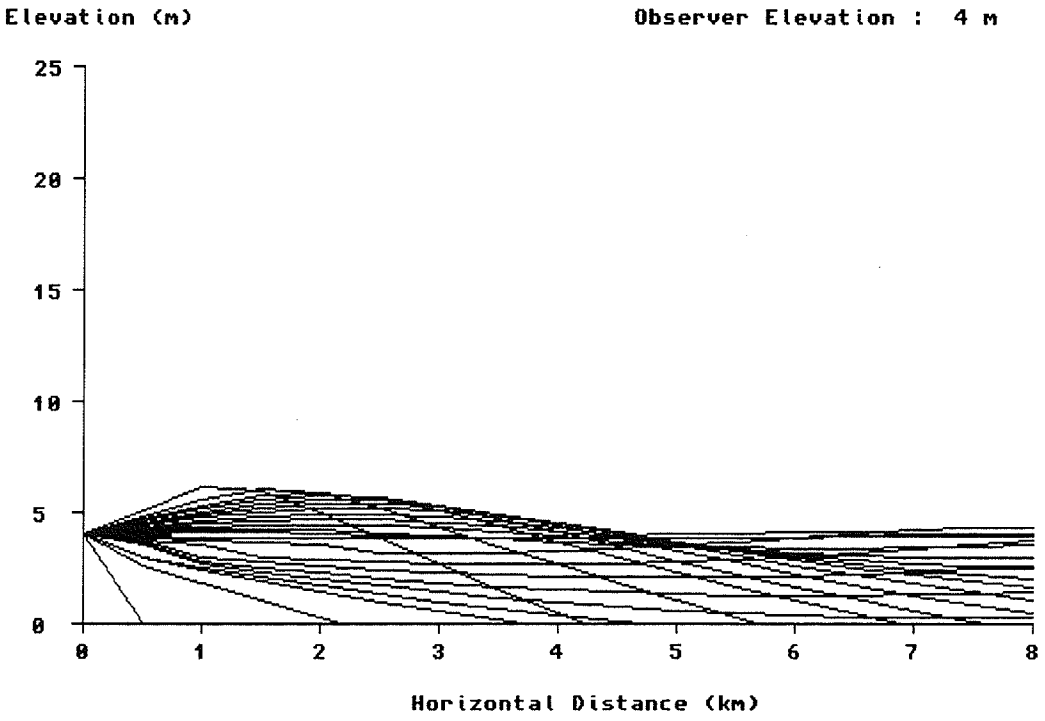

Figure 6.13a Ray plots for runway scene at 1.8 km

Figure 6.13b Observer's view of the scene at 1.8 km

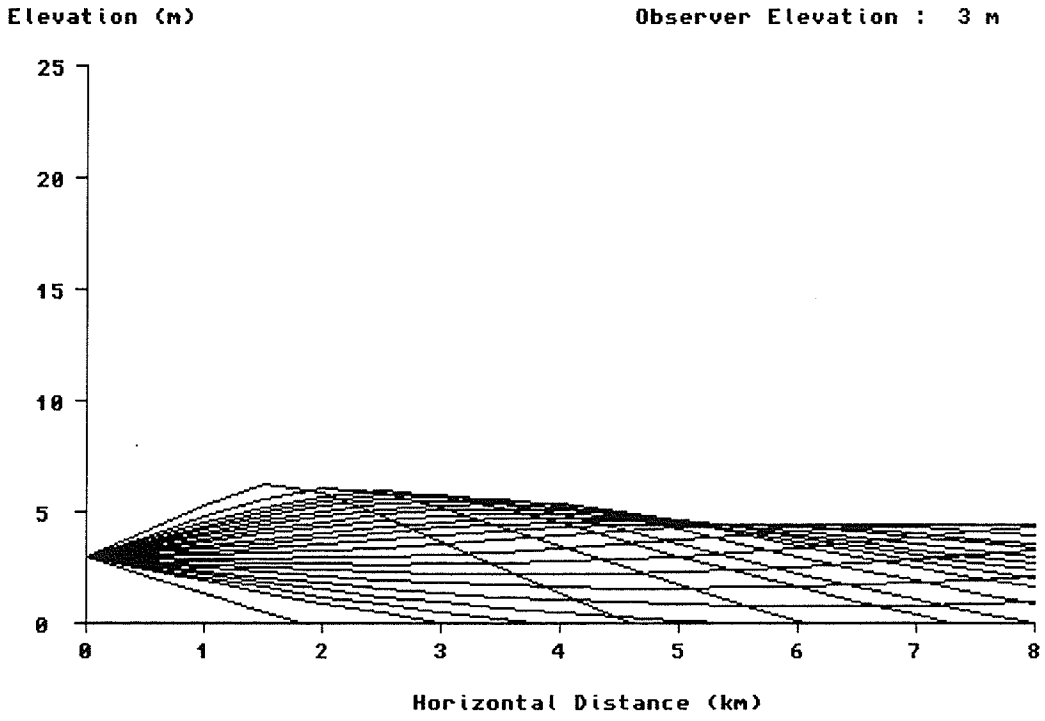

Figure 6.14a Ray plots for runway scene at 1.4 km

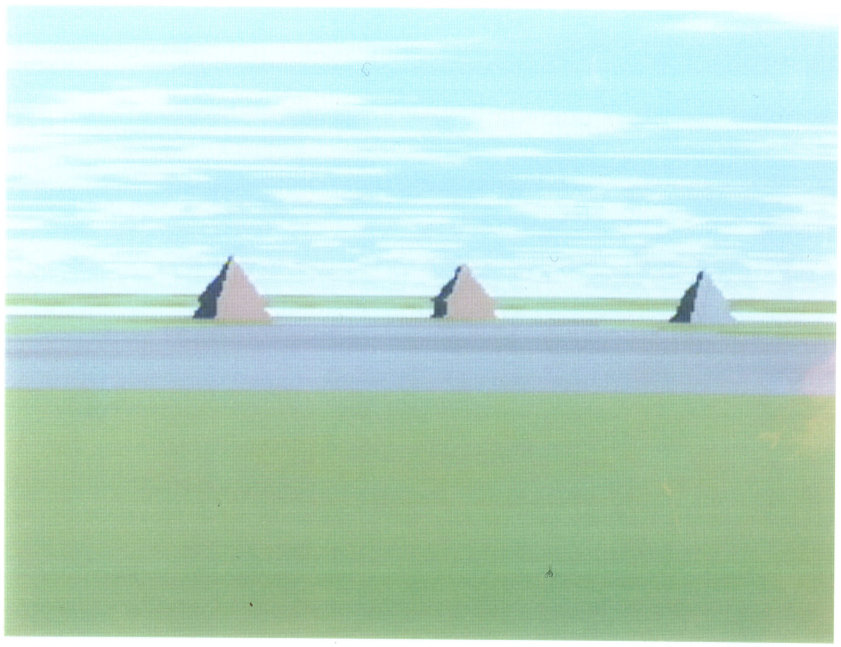

Figure 6.14b Observer's view of the scene at 1.4 km

7 User Manual

Previous chapters discussed the mathematics and the implementation of the simulation program. This chapter will describe how to use the program and how to create the necessary input files that specify the environment being modelled.

7.1 Overview

In order to generate an image the program must be supplied a descriptive database describing the modelled "scene". Like many 3D modelling and rendering programs, the simulator requires information about object geometry, rendering attributes, lighting information, atmospheric conditions, viewing geometry etc. While prototyping and testing the simulator it was found that inputting data as binary files becomes very tedious. To simplify matters, a descriptive language was developed that allows scene specification using C like syntax. The whole process becomes similiar to writing, compiling and testing a C program.

The environment is described in a standard ASCII file which may be created with any text editor. This file is then read by the program and interpreted by a parser. The parser is responsible for detecting syntactical errors in the scene description language, data validation and for partially building the environment database used by the ray tracing code. A pre-processing stage follows the parser to tidy up loose ends prior to any ray tracing. It may take a few trys to get through the parsing stage free of errors but once that is done the program will render the scene and generate an output image. Subsequently, the scene description may easily be "tweaked" to test different parameters and see their effect on image generation.

Generated images are saved in a raw 24-bit color format with three files being created, one for each of the red, green and blue components. Each color component contains 8-bits of information saved in row-column order (image height and width information is not saved so one has to remember to keep track of this).

7.2 Running Mirage3D

To run the program one enters the following at the command line:

Mirage3D "datafile" [-command, -command...]

The program expects a "datafile" which specifies a scene description file and will accept one or more of the following optional commands given in any order:

-r"filename"	Red output image file (red.raw)
-g"filename"	Green output image file (green.raw)
-b"filename"	Blue output image file (blue.raw)

-D	Debug, dump processed input to stdout (No)
-D"filename"	Debug, dump processed input to "filename"
-E	Echo input file while parsed (No)
-x "filename"	Scale image using filename as a temporary buffer (No)

-w#	Image width	0 <= # < 1024 (320)
-h#	Image height	0 <= # < 1024 (200)
-s#	Starting image row	0 <= # < 1024 (0)
-e#	Ending image row	0 <= # < 1024 (200)
-d#	Maximum recursion depth	# > 0 (2)
-l#	Log scale	# > 1.0 (0.0)
-S	Enable shadow detection	(No)
-N	Check the orientation of polygon normals	(No)
-W	Wireframe display (No)	

Entering the program name without any arguments will display a listing of what is available. There should be no intervening spaces between a command and its expected parameter.

Some of the command switches allow one to specify different options to the program without having to modify the data file, in which case they will take precedence.

There is no file naming convention used, however all specified file names should contain the complete path. The current directory is assumed otherwise. If no output image files are specified the following defaults are used for each of the red, green and blue files respectively:

red.raw	default file if none specified with -r
green.raw	default file if none specified with -g
blue.raw	default file if none specified with -b

The default ".raw" files will be saved under the current directory.

Commands -w and -h

The -w and -h commands allow specification of different image width and height. Each command should be immediately followed by a positive integer greater than 0 and less than 1024. If a region (a smaller area of a full image) has been defined in the data file it will be ignored.

Commands -s and -e

The -s and -e switches can be used to generate a smaller vertical area of a full image. The area generated will span the entire width of an image. These options may be useful for batch processing where different sections of an image can be generated on multiple computers. It will also eliminate the need to create a different input file for each system. The -s option defines the starting row and the -e option defines the last row. Both options should be followed by positive integers lying within the bounds of the defined image height. This will either be that specified with the -h command or, by default the height defined in the input file. Again if a region has been defined in the data file it will be ignored.

If one wishes to generate a smaller area of the image, width wise as well, then the region command should be used in the input file and none of the options -h, -w, -s or -e should be specified.

Command -d

The -d command allows specification of the maximum recursion depth used while tracing reflected rays. The command should be followed by a positive integer greater then 0. The program makes heavy use of the system stack so one should make sure that the recursion depth is not too deep. If the system stack can be changed then larger recursion depths should not be a problem. On an Amiga it was found that a stack of 10,000 bytes was adequate for a recursion depth of 5.

Command -S

The -S command signals the program to take into account shadowing while rendering the image. Specifying this option will add to the realism of the image at the expense of longer generation times.

Command -l

The log scaling option -l allows one to boost up the intensity of the darker areas of an

image relative to the brighter areas. The command should be followed by a positive floating point number greater than 1.0. For the test images produced it was found that values of 5.0 to 10.0 work reasonably well. Higher values may tend to wash out a final image.

Command -x

The -x option is used to specify color scaling after the image has been generated. By default, each color determined for a pixel is clipped to the range {0..1}. If this option is given the program will store all pixel colors to a buffer file while the image is being generated and a maximum color value will be maintained. After image generation the buffered file will be read back in and normalized before it is resaved in the regular image format. The buffer file contains three doubles per pixel therefore it can be quite large. In general this post-processing operation is not needed as the number of color values that do get clipped is small on the average.

Command -E

If the echo option -E is specified the program will echo the input data file as it is being parsed to stdout.

Command -D [filename]

If the debug option -D is specified the program will list all processed information to either stdout or the specified filename. Note that no image will be generated with this option. The debug option may be useful to initially get a feel for how the program compiles the scene description language and generates specific information used internally, a good example being concatenated transformations. Within the description language numerous transformations such as scaling, rotation and translation may be specified for an object in any order. Internally they are saved in a single matrix. The ordering of transformations will affect the final results. If one finds the final transformed object is not quite as expected then it is possible to work through the matrix multiplications by hand and check the final results with those generated by the program.

Command -N

When specifying normals for polygon facets to be used with Phong shading, it is possible that some surfaces will not be oriented correctly producing black holes in objects. If the command is specified the program will check that the supplied normals are oriented in the same direction as a calculated normal determined by the supplied vertices. The normal's orientation is determined by the right hand rule under the assumption that the vertices have been specified in a counter-clockwise order.

Command -W

Specifying this option will cause the program to display the scene as a wireframe drawing. This option is useful for testing object placement and different viewing parameters.

7.3 Errors

When running the program five types of errors may be encountered, all of which terminate program execution. If any error does occur it is generally during the parsing or pre-processing phases however there are a few which may occur during image generation.

7.3.1 Syntax Error

The first type involves syntactical errors found while parsing the input file. These errors are caused by improper use of the scene description language, for example an unrecognized statement or a missing bracket. When a parsing error of this type is encountered it will be displayed in a format similiar to the following:

```
122 vector_variable = < ?#$%@, 2.0, 3.5 >
                      ^
Syntax Error : Expecting a numeric constant or variable
```

The line number and its corresponding text will be displayed along with an up arrow (^) pointing to the approximate location of the error. This is followed by a brief descriptive message.

The -E option is useful when correcting syntax errors as the lines preceding the invalid syntax will also be on the display screen. Since only a single line is displayed some syntax errors will not be as obvious as the preceding example but become immediately apparent when seen in the context of the input file. Appendix B lists all the possible syntax errors that may occur.

7.3.2 Data Error

The next type of error is related to the validity of specified or calculated data. This type of error may or may not be detected at a specific line location in the input file however it will generally be flagged at the end of the appropriate structure statement block. For example:

```
400         RADIUS -5.8
                    ^
```

Data Error : Sphere radius must be > 0

In this case the range error was detected immediately at the input file location where it occurred. As another example assume that the -E option has been specified with the following input file statements displayed on screen before the error is flagged:

```
 88   VERTICES
 89   {
 90     8        /* # of vertices to follow */
 91     < 1,1,1 > /* vertex id 0 */
 92     < 1,1,1 > /* vertex id 1 */
 93     < 1,1,1 > /* vertex id 2 */
 94     < 0,0,1 > /* vertex id 3 */
 95     < 1,0,0 > /* vertex id 4 */
 96     < 1,1,0 > /* vertex id 5 */
 97     < 0,1,0 > /* vertex id 6 */
 98     < 0,0,0 > /* vertex id 7 */
 99   }
100
101    FACETS
102    {
103     12        /* # of facets to follow */
104     0 4 5     /* facet id 0 */
105     0 5 1     /* facet id 1 */

105     0 5 1     /* facet id 1 */
              ^
```

Data Error: Invalid vertices

In this case the first three vertices defined within the VERTICES statement block are all the same point. Later on in the FACET statement block the parser attempted to create triangular FACET 1 using the VERTICES 0, 5 and 1. Three non-collinear points are required to define a triangle thus the parser recognized the two collinear points and indicated the error. Note that this type of error could not be detected at the time the VERTICES were defined but only later when the FACET was being specified.

A list of all the possible data errors is provided in Appendix B.

7.3.3 Memory Allocation Error

The third type of error is due to a memory allocation request that did not succeed. If memory is not available for a required structure the program will display a message similiar to the following:

Allocation Error: No memory for SPHERE info structure

Appendix B lists all the possible allocation errors that may occur.

7.3.4 Internal Error

The fourth type of error is internal and is more than likely due to a program bug. The program makes internal checks for errors that may possibly cause a crash. The error most frequently checked for is a NULL pointer. An example of an internal error is as follows:

Internal Error : NULL Attributes in ParseSurface()

Checking for fatal errors of this type has only been left in the routines used for parsing and pre-processing. Appendix B lists all the possible internal errors that have been considered.

7.3.5 Unrecoverable Error

The last type of error is one which could not be classifed under one of the previous types and is always unrecoverable. It may be due to an invalid command line parameter or a file error for example. An example of an unrecoverable error is the following:

Unrecoverable Error: No input file specified

Appendix B lists all the possible unrecoverable errors.

7.4 Aborting

The executable that is run on an Amiga may be aborted by pressing Ctrl-c. The image that is generated up to that point is lost.

7.5 Scene Description Language

The following section describes the file format expected by the parsing phase of the program covering the syntax used by the scene description language and parameter specification.

The description language is largely based upon the C structures used internally within the program. As discussed in previous chapters these structures are used to hold information describing the modelled environment (for the remainder of this section it is assumed that all geometry information is specified in a right handed 3-dimensional coordinate system). Much of the language uses uses a C like syntax for initializing the members of these structures. The language also allows variable declaration and assignment for some of the more useful structure types.

7.5.1 Statement Blocks

As implemented in the language, structures may be defined by statement blocks containing one or more member assignment statements or by structure variables. A structure variable eliminates the need to explicitly specify a statement block.

A statement block starts with a keyword identifying the structure type followed by a body contained within opening and closing curly brackets. An assignment statement consists of a keyword identifying a structure member and generally a parameter list containing one or more arguments. An assignment statement may also be another statement block just as a member of a C stucture can be another nested structure. The syntax for a statement block is as follows:

```
keyword
{
  keyword parameter-list
      ...
  keyword { ... }
      ...
  keyword structure-identifier
      ...
  keyword structure-identifier { ... }
      ...
}
```

where { ... } denotes amother statement block and where parameter-list has

the form:

argument [, argument , argument ...]

where an argument may be one of the following:

numeric-constant
numeric-identifier
vector-constant
vector-identifier

Within a statement block there are three possible forms of syntax that may be used to define nested structures:

a) keyword { ... }
b) keyword structure-identifier
c) keyword structure-identifier { ... }

Forms b) and c) as well as the different type-identifiers are discussed in more detail in the section covering variables.

Within a block, statements may be specified in any order and in most cases as often as one likes (although there is no good reason to do so). If a statement is repeated then the assignment will be evaluated again. There are some assignment statements which represent nested structures which may not be repeated. These will be pointed out as they are discussed. Not all statements need be specified (in fact one may specify an empty statement block i.e { }) as most structures used by the program are initialized with default values.

All statements must be separated by at least one space and multiple statements can be placed on the same line or may be spread across several lines (even blank lines). The same applies to arguments in parameter lists. The following examples are all identical and valid:

```
a)
  SOLID
  {
    SPHERE
    {
      RADIUS 10          CENTER < 0, 0, 0 >
    }
  }
```

b)

```
SOLID { SPHERE { RADIUS 10   CENTER <0,0,0> } }
```

c)

```
SOLID
{
  SPHERE {
   RADIUS
   10

   CENTER < 0,
   0
   ,
     0  >
  }
}
```

7.5.2 General

Before covering specifics of the language in greater detail there are other more general aspects of it which should be mentioned first.

1) The parser is case-sensitive so for example the following would all be considered different identifiers:

color, Color, COLOR, CoLoR etc.

2) All RESERVED KEYWORDS in the language are entered in UPPERCASE. Keywords should not be broken up. For example, given the following

```
SPHERE { RAD
                  IUS 1.68 }
```

the parser would recognize "RAD" and "IUS" as two different identifiers and will generate an invalid statement error when encountering "RAD".

3) Nested comments are allowed and may span more than one line. Comments are specified by the delimiters "/*" and "*/". Comments may be placed any where within the input file. The following are some examples of valid comments:

a) /* here's
a
comment */ /* and another /* and another */ */

b) /* before a statement */ RADIUS /* inside a
statement */ 10 /* and after one */

Nested comments are useful for blocking out larger sections of the input file while testing specific portions without having to remove any meaningful comments that may already exist. Nested comments should be used sparingly as it easy to accidently forget to close a comment. The parser will catch a missing closing delimiter and will issue an unbalanced comment error. If an opening delimiter is missing chances are the parser will reach a closing delimiter some place within the input file and will issue an invalid statement error.

4) The parser recognizes two types of constants:

i) Numeric

A numeric is specified by an optional sign, a whole part an optional decimal followed by a fractional part and an optional exponent. A digit must preceed and follow a decimal if any. If an exponent is specified there should be no itervening spaces between it and the fractional part. An exponent consists of the letters "e" or "E", followed by an optional sign and one or more digits. The following are examples of numeric constants:

0 123 +5 -99 1.0 +5.2 -2.4 56.99e+5 1.2E-45

The parser makes no real distinction between integers and floating point constants. All numbers be it integral or floating point are considered to be of type Numeric. Internally they are stored as C doubles. The parser determines whether a C integer or a floating point is required by the context of the statement. If a statement requires an integer parameter and a floating point value is supplied then it is cast into an integer. Syntax errors such as "expecting integer" will not be generated. The converse applies to statements expecting floating point values.

What is the maximum and minimum values that may be specified for a numeric? This will depend on the statement being parsed. The parser assumes that all specified numeric values are floating point and just keeps pulling in characters from the input file as long as they follow the above mentioned format or until a space is encountered. The characters are then stored in a string which

is passed to the standard C library function atof(). The function converts an ASCII string into a C double precision floating point value. This method was easy and eliminated the need to do separate type checking for integers and floating point. It also allows one to enter integer constants for floating point arguments without always having to specify a decimal. However this method is also prone to possible bugs. As previously mentioned, any statement expecting an integer argument uses the C cast operator on the specified numeric. If the numeric actually represents a large floating point value then the cast operation will result in an invalid int. With this in mind one should limit the range of floating point arguments to that of a C double. For statements requiring integral arguments one should limit values to that of a C long int.

ii) Vector - numeric triplet

A vector constant consists of an opening angle bracket followed by three numeric constants separated by commas and a closing angle bracket. The following are examples of valid vector constants:

< 0, +1, -2 > < 1.3, -2, 4.8e-10 >

5) All 3-dimensional points and direction vectors are specified as XYZ triplets and have the following order when expressed as vector constants:

< x, y, z >

If a direction vector is requested as an argument it does not have to be in normalized form, this is done by the program.

6) The program does not assume any specific scaling, all coordinate values are unitless.

7) The 3-dimensional center of the environment is always the origin:

< 0, 0, 0 >

8) All specified X, Y and Z dimensions must lie within +/- 1.0e20.

9) All color values are specified as RGB triplets and have the following order when expressed as vector constants:

< Red, Green, Blue >

10) Each color component is limited to the range {0..1}.

11) All specified intensity values should be in the range {0..1}.

7.5.3 Data File Organization

A typical input file generally has the following logical format:

Variable Declarations
Variable Assignments
Scene Definition
Light Object Definitions
Solid Object Definitions

Syntactically the file may look as follows:

```
/* variable declarations */
VARIABLES { ... }

/*  variable assignments */
numeric-identifier   = ...
vector-identifier    = ...
structure-identifier = ...
     ...
/* scene definition */
SCENE { ... }

/* Solid object definitions, one or more specified
   using any of three different syntactical forms */
SOLID { ... }
SOLID structure-identifier
SOLID structure-identifier { ... }
     ...

/* Light object definitions, one or more specified
   using any of three different syntactical forms */
LIGHT { ... }
LIGHT structure-identifier
LIGHT structure-identifier { ... }
     ...
```

where { ... } indicates a statement block

The input file generally has one or more light object definitions that describe the light sources used to illuminate the scene. There will also be one or more solid object definitions describing the the physical structures that are to be rendered in the scene. Each light or solid object may be explicitly defined by a statement block or it may be assigned the values defined by a variable of the appropriate type.

The scene definition is used to specify all the information required by the program that is not related to either lights or solids. Only one scene definition may be specified within the input file.

One may also optionally use variables within the input file. Variables may be used to simplify the data file to some degree. Variables, once created are only used during the parsing phase and are destroyed after the file has been processed. Variables by themselves are never added to the scene and must be used in an assignment statement to have an effect.

If variables are used then they must be declared within a declaration block before they may be referenced in any assignment statement. Only one declaration block may be specified. It may however appear any where within the input file not neccessarily at the top. All assignments TO variables must occur outside of the scene definition and any light or solid object definitions. The next section discusses the use of variables in more detail.

7.5.4 Variables

This section provides an overview of the more commonly used structures and describes the syntax for declaring, assigning values to variables and assigning variables to structure members within statement blocks.

7.5.4.1 Types

The description language supports the declaration and assignment of the two basic types, numeric and vector, as well as several pre-defined structure types. Numeric and vector variables are most useful in assigning symbolic names to numeric or vector constants. The use of numerics or vectors within arithmetic expressions is not supported. Structure variables are useful in cases where one nested structure definition is common to a number of other structure definitions. For example, two or more objects may all share the same surface attributes. A variable may be used to define the surface information only once, after which it is assigned to each of the objects. The following types are supported and may be used as variables:

1) Numbers

NUMBER type

Many statements require a single integer or floating point value as argument. The value may be a constant or a variable. Since arithmetic expressions are not supported the use of variables is rather limited however they are useful for providing a symbolic name to an often used constant. Numeric variables may also be used instead of constants within a vector.

2) Vectors

VECTOR type

Statements requiring points, directions and colors expect either a vector constant or variable. Vector variables are very useful for providing symbolic names to often used vector constants. For example, sets of colors or common transformations may be declared with vector variables and then used throughout the input file.

3) Object Structures

Object structures are used to specify all the information required to represent

a modelled entity in the environment. Objects may be one of two types, either Solid or Light.

SOLID Type

Solid objects are always visible in a ray traced image and represent actual physical structures such as a house, tree or a mountain range. Associated with each solid object are surface attributes which define the appearance of that object as it is rendered. Solids always have opaque surfaces, transparent objects are not supported. The shape of the solid is defined by a list of basic primitive structures.

LIGHT Type

Light objects may or may not be visible in a ray traced image and are used to specify how a scene is to be illuminated by direct lighting. Associated with each light object are source attributes which define illumination characteristics such as color and intensity. The source attributes may also define certain geometric information such as the position of the source within the scene.

4) Object Attributes

Each object be it a light or solid has associated with it a set of attributes which provide rendering information used by the program. Surface attributes define how a solid object should appear within a scene. Source attributes define how a light object should illuminate the scene.

SURFACE Type

To render a solid object the program needs to know certain information about its surface such as its color, how diffuse or specular it is, whether it is textured etc. This information is defined in a surface attributes structure.

SOURCE Type

When rendering a solid object a certain portion of its resultant surface color is attributable to illumination from direct lighting. All defined light objects contribute to this calculation. For each light defined, the program needs to know its color, intensity, whether it represents a point or infinitely distant source, whether its intensity should be attenuated with distance etc. This information is defined in the source attributes structure.

5) Object Primitives

Any visible object within the scene will have its physical shape defined by a list of basic primitives. The basic primitives are used as building blocks in creating more complex structures.

SPHERE Type

The sphere is defined by a center and a radius. The sphere also has an implicit reference frame attached to it which is used for texturing.

PLANE Type

The plane is defined by three coplanar points and extends infinitely in space.

TRIANGLE Type

The triangle is defined by three coplanar points forming a three sided polygon.

RECTANGLE Type

The rectangle is defined by three coplanar points forming a four sided polygon whose opposite sides are parallel.

POLYGONS Type

The polygons primitive is defined by a set of triangular facets. The primitive allows one to construct basic shapes more complex than those provided by the program. Simple shapes such as a cone, cylinder or torus could be created or one may create a more complex shapes such as different tree or rock formations. The primitive also allows for Phong normal interpolation. The normal interpolation creates a simulated smooth surface across adjacent facets.

6) Transformations

Transformations allow one to size, place and orient both objects and primitives within the scene. They are also used to transform any textures which may have been defined for a surface.

TRANSFORM Type

The transform structure defines a single transformation matrix and its inverse. A single transformation matrix may be composed of any number of scalings, translations and rotations combined in any order.

7.5.4.2 Declaring

Before any variable may be used it must first be declared. A declaration allocates memory for the variable, initializes it with default values and inserts the supplied identifier name into the parser's symbol table. All declarations are made once within a single block with the following syntax:

```
VARIABLES
{
    type-specifier  identifier-name, identifier-name;
    type-specifier  identifier-name;
            ...
}
```

The VARIABLES keyword starts the declaration block whose body is contained within opening and closing curly brackets. The body may contain one or more declaration statements whose syntax is similiar to a C declaration statement. Each statement starts with a type specification and is followed by one or more names that identify the variables of that type. Each identifier name is separated by a comma and the statement is terminated by a semi-colon. Each declaration statement may be placed all on one line or may be spread out over multiple lines. Comments may be inserted.

The following is an example of a valid declaration block:

```
VARIABLES
{
    type-specifier1  identifier-name1,  /* first variable */
                identifier-name2,  /* next variable */
                identifier-name3;  /* last variable and end of
                                1st declaration statement */

    type-specifier2  identifier-name4; /* end of 2nd */
    type-specifier3 identifier-name5,
          identifier-name6, identifier-name7; /* end of 3rd */
}
```

All identifier names must start with an alphabetic character and may be composed of any combination of alphanumeric and underscore characters. All identifier names have a maximum length of 256 characters. Reserved keywords cannot be used as identifier names. Refer to Appendix A for a complete list of reserved keywords. The following example illustrates declaration statements for all the supported types using valid identifier names:

```
VARIABLES
{
  NUMBER    theta, Pi_times_2;
  VECTOR    WHITE, magenta, X_axis, tree_position;
  SPHERE    ball, sphere_not;
  PLANE     the_ground, tiled_floor;
  RECTANGLE north_wall, window_pane;
  TRIANGLE  triangle, Triangle, pie_slice;
  POLYGONS  cube, fractal_landscape;
  SURFACE   mostly_diffuse, metal_like, very_rough, textured;
  SOURCE    flood_light, PointSource;
  TRANSFORM move_along_X_by_2_then_rotate_about_Z_by_10_degrees;
  SOLID     house, F15, airport, jello_solid_as_a_rock;
  LIGHT     neon, spot_light;
}
```

7.5.4.3 Assignment To Variables

Once a variable is declared it is also initialized with default values. Generally the default initialization will be of little or no use as is. For example a SOLID variable will have no primitives. The initialized values are identical to those of an empty statement block. The statement block actually defines a structure that is added to the scene. If no structure assignment statements are made within the block i.e. the structure member values are not changed, then the default structure will be added to the scene. To be of practical use a variable should be initialized with specified values as there is no gain in assigning a default variable to a default structure member. The assignment of values to variables differs slightly for a number, vector and structure. The following syntax is used for each:

1) Number

```
numeric-identifier1 = numeric-constant
                      numeric-identifier2
```

2) Vector

```
vector-identifier1  = vector-constant
                      vector-identifier2
```

3) Structure

```
structure-identifier1 = statement-block
                        structure-identifier2
                        structure-identifier2 then statement-block
```

Where structure may be one of solid, light, surface, source, transform, sphere, plane, triangle, rectangle or polygons.

In the above syntax the second form is common to all, the assignment of one variable to another of the same type. In this type of an assignment, the variable on the left hand side of the equal sign becomes an identical copy of the variable on the right hand side. This does apply to variables representing structures. If a variable A represents a solid object with 200 primitives and it is assigned to a variable B, then variable B will also represent the same solid object with 200 primitives. Note that the identifier on either side of the equal sign must be unique. As an example, given the following declarations:

```
VARIABLES
{
  NUMBER    Pi, angle;
  VECTOR    Up, North;
  SPHERE    Ball, sphere;
  SURFACE   Speckled, Spots;
  SOURCE    simulated_sun, infinite_source;
  TRANSFORM object_motion, light_motion;
  SOLID     torus, donut;
}
```

the following assignments are valid:

```
angle = Pi
Up = North
Ball = sphere
Spots = Speckled
simulated_sun = infinite_source
light_motion = object_motion
```

donut = torus

The other forms of assignment are different for numeric, vector and structures therefore each will be discussed separately.

1) Number

The assignment of a numeric constant to a numeric variable is straight forward. The following are valid syntax:

Pi = 3.14159 red_component = -0.78 radius = 1.5e8 ray_depth = 5

2) Vector

The assignment of a vector constant to a vector variable is also straight forward. The following are valid syntax:

```
z_axis = < 0.0, 0.0, 1.0 >
ocean_blue = < 0.1, 0.8, 1.0 >
sphere_center = < 1, 2, 3 >
light_direction = < -1.5e-3, 4, -0.4E-3 >
```

Another form of vector assignment allows one to intermix numeric constants and numeric variables. The following are valid syntax:

```
object_rotation = < theta, phi, gamma >
any_color = < red, green, blue >
cube_size = < x_size, 0.5, 4 >
```

3) Structure

The remaining structure assignments are a little more complex. The first form allows one to assign a statement block to the variable. Within a statement block it is also valid syntax to use a variable as an argument to a statement. That is,

```
keyword
{
  keyword structure-identifier /* variable argument */
  keyword parameter-list
}
```

When assigning a statement block to a variable, the keyword that starts the

block is not required. The keyword starting the block indicates the type of structure being defined, however with variables this is already implied by the variable's type.

```
structure-identifier1 = /* keyword implied */
                {
                  keyword structure-identifier2
                  keyword parameter-list
                }
```

Depending on the structure type of the variable this assignment can become rather complicated. A basic example of an assignment to a SPHERE variable is the following:

```
ball = {
       CENTER < 1,2,3 >
       RADIUS ball_radius
     }
```

This will create a sphere variable centered at <1,2,3> with a radius specified by the numeric-identifier "ball_radius".

Only the specified structure members will be modified in the target variable therefore one may still use some of the default values. For example:

```
ball = { RADIUS 10 }
```

This will create a sphere variable centered at <0,0,0> the default, with a radius of 10 units.

A little more complicated example is as follows:

```
/* initialize a sphere primitive variable */

ball_2 = { /* SPHERE keyword implied */

          CENTER ball_2_center
          RADIUS ball_2_radius

        } /* end of sphere primitive */

/* initialize a solid object variable with two balls */
```

```
two_balls = { /* SOLID keyword implied */

            SPHERE /* ball_1  defined by statement block */
            {
              CENTER ball_1_center
              RADIUS ball_1_radius
            }

            /* example of "keyword structure-identifier" syntax */

            SPHERE  ball_2

          } /* end of solid object */

/* another example of "keyword structure-identifier" syntax */

SOLID two_balls  /* now add the object to the scene */

/* end of example */
```

The above will create a solid object variable containing two spheres. The first sphere "ball_1" is defined by a statement block and the second sphere "ball_2" is defined by an assignment from a previously intialized sphere variable. Finally a solid object is added to the scene by specifying the identifier "two_balls" as an argument to the SOLID keyword.

The last form of structure assignment is a combination of the first two forms being the most useful. It is a sort of "assign then modify" statement. This form has the following syntax:

```
structure-identifier1 = structure-identifier2
                {
                  keyword { ... }
                  keyword structure-identifier3
                  keyword structure-identifier4 { ... }
                  keyword parameter-list
                }
```

This syntax is equally valid with statement blocks:

```
keyword structure-identifier1
{
  keyword parameter-list
  keyword { ... }
  keyword structure-identifier2
  keyword structure-identifier3 { ... }
}
```

For example, two or more objects may all share the same surface attributes except they all have different colors. A variable may be used to define the surface information only once, after which it is assigned to each of the objects. Immediately following each surface assignment would be a statement block specifying the new color. For example:

```
/* initialize a surface variable with common surface attributes */
common_surface = { /* SURFACE keyword implied */
                   /* desired values specified in  this statement block */
                 }

/* solid definitions now follow */
SOLID /* statement block defining solid object 1 */
{
 /* primitive specifications for solid 1 would go here */

 SURFACE common_surface  /* heres the assignment */
 {
   COLOR orange        /* followed by the modification */
 }
}

SOLID /* statement block defining solid object 2 */
{
 /* primitive specifications for solid 2 would go here */
 SURFACE common_surface  /* heres the assignment */
 {
   COLOR <0,0,1>       /* followed by the modification (blue) */
 }
}
```

In the above, a surface variable is first defined, ie the identifier "common_variable". Two solid objects are added to the scene defined by their respective statement blocks. The surface attributes of each are defined, first by

the assignment of "common_surface" then by the statement block specifying a different color.

A little more complicated example is as follows:

```
/* initialize a surface variable with common surface attributes */

common_surface = {
            /* use default values except for color */
             COLOR bronze
           }

/* initialize a common sphere primitive variable */

ball = {
        CENTER ball_center
        RADIUS ball_radius
      }

/* initialize a solid object variable with two bronze balls
   one at "ball_center" the other at <1,2,3> */

two_balls = {
              SPHERE ball /* assign then modify */
              {   CENTER < 1,2,3 >  }

              SPHERE  ball  /* straight assignment */
              SURFACE common_surface /* straight assignment */
            } /* end of solid object assignment */

/* initialize a solid object variable with three purple balls
   one at "ball_center", one at <1,2,3> and another at <4,5,6> */

three_balls = two_balls /* already has 2 balls defined */
         {
            /* add one more ball */
            SPHERE ball  /* assign then modify */
            {  CENTER <4,5,6>  }

            SURFACE common_surface /* assign then modify */
            {   COLOR purple   }
         } /* end of solid object assignment */

/* now add objects to the scene */

/* --- add one blue ball --- */
SOLID /* specify solid object by a statement block */
{
   SPHERE /* centered at origin, the default */
```

```
    {   RADIUS 2  }

    SURFACE /* default surface attributes except for color */
    {  COLOR blue  }

  } /* --- end of blue ball definition --- */

  /*  --- add two bronze balls --- */
  SOLID two_balls /* straight assignment */

  /* --- add three yellow balls --- */
  SOLID two_balls /* assign then modify */
  {
    SPHERE ball { CENTER <4,5,6> } /* assign then modify */

    SURFACE common_surface /* assign then modify */
    {   COLOR yellow  }

    TRANSFORM /* specify transformation by a statement block */
    {
      /* move it somewhere different than two_balls */
    }
  } /* --- end of three yellow balls definition --- */

  /* --- add three purple balls --- */
  SOLID three_balls  /* assign then modify */
  {
    TRANSFORM /* specify transformation by a statement block */
    {
      /* move it somewhere different than two_balls */
    }
  } /* --- end of 3 purple balls definition --- */

  /* end of example */
```

The previous example is fairly complicated however it illustrates some of the different ways in which one may define structures. Generally there will be one or more ways one may specify information all with identical results. Since the input file can become rather confusing it is best to stick with one method and use it consistently.

7.5.5 Statements

This remainder of this chapter will discuss all the statements that are used within the description language.

7.5.5.1 Scene

This section will describe the statements within a scene definition block. The scene describes the environment being modelled. The majority of work in setting up a scene description involves the definition of the solid objects that are rendered and the light objects that illuminate the scene. There are a number of other elements required by the ray tracing code that also define the environment being rendered. Mirage specification, viewplane definition, ambient lighting and image resolution are some examples of these. All these extra elements are lumped together and specified within the scene statement block. All of the options which may be specified are initialized with default values. The syntax for the scene statement block is as follows:

```
SCENE
{
  SHADOWS
  LOGSCALE        floating-point
  DEPTH           integer
  BACKGROUND      vector
  AMBIENT { ... }
  HAZE { ... }
  SKY { ... }
  MIRAGE{ ... }
  VIEW { ... }
}
```

where { ... } indicates a statement block

The following discussion describes each of the statements that are used within the SCENE statement block.

SHADOWS

This is a boolean flag that will enable shadow detection when calculating illumination from direct light sources. Image generation times will be longer with this option enabled however the results will be more realistic. Specifying the -S command has the same effect. By default shadow detection is disabled to allow

quicker rendering times for testing purposes. When shadow detection is enabled for the scene one may still selectively disable shadows for specific light sources when they are defined. The converse is not possible when shadows have been disabled for the scene.

LOGSCALE scale

The log scaling statement allows one to boost up the intensity of the darker areas of an image relative to the brightest area using the following formula:

new_value = log(1 + scale old_value) / log(1 + scale)

The above formula is applied individually to each of the red, green and blue components of a color after they have been linearly scaled to the range { 0 ..1}. When an image is linearly scaled the brightest values may dominate if there is a large dynamic range. Log scaling compresses this dynamic range. The statement expects a single scaling parameter which must be a value greater than 1. Values less than or equal to 1 have no effect. Log scaling is not applied by default. For the test images produced it was found that values of 5.0 to 10.0 work reasonably well. The effect of log scaling tends to be negligible for values greater than 40. Large values may tend to wash out a final image. This option may also be specified on the command line with the -l switch.

DEPTH max_depth

The DEPTH statement allows specification of the maximum recursion depth used while tracing specularly reflected rays. The statement expects a positive integer greater then 0. If the specified value is less than 1 then the depth will be set to the default value of 2. The program makes heavy use of the system stack so one should make sure that the recursion depth is not too deep. If the system stack can be changed then larger recursion depths should not be a problem. On an Amiga it was found that a stack of 10,000 bytes was adequate for a recursion depth of 5. This option may also be specified on the command line with the -d switch.

BACKGROUND <r,g,b>

The BACKGROUND statement allows one to specify a RGB color that is to be used as default if a ray does not intersect any object within the scene. If the HAZE structure is defined then its specified color will be used instead and the background color will have no effect. The default value for the background color is all black, i.e. < 0, 0, 0 >.

AMBIENT

The ambient structure specifies how the program calculates indirect lighting within the scene. Indirect lighting accounts for illumination not attributable to direct contributions from light sources or specular contributions from reflecting surfaces. It is caused by the scattering of light in all directions from surfaces with both diffuse and specular components. It is common to approximate ambient lighting with a constant term. One may also attempt to approximate ambient lighting by tracing rays sent in random directions from a surface. The ambient statement allows specification of either method. The default is to use a constant ambient term. The syntax for the statement block is as follows:

```
AMBIENT
{
  INTENSITY  floating-point
  COLOR      vector
  NSAMPLES   integer
  DEPTH      integer
}
```

The following discussion describes the statements within the AMBIENT statement block.

INTENSITY brightness_value
COLOR <r,g,b>

These two statements are used to define a constant ambient term. The specified intensity value determines the amount of constant ambient illumination that should used for the scene. The value will be used to scale the specified color which is added to all rendered surfaces. Each individual surface may also specify the amount of ambient light which is to be added to it. Small intensity values are good for indoor scenes while larger values should be used for outdoor scenes. The default value is 0.2. For indoor scenes, using an ambient color which is slightly tinted towards the color of walls works well. A gray shade is appropriate for outdoor scenes. The default ambient color is < 0.5, 0.5, 0.5 >.

NSAMPLES max_samples
DEPTH max_depth

These two statements are used to specify a ray tracing approximation to diffuse interreflection. The value supplied for the NSAMPLES statement determines how many random rays will be traced. For even a small number of

propagating rays the resulting number of intersection calculations becomes quite large. For each surface that is hit additional random rays are generated and traced. As the program traverses the ray depth tree the number of rays has a geometric growth. For this reason a maximum depth may be specified with the DEPTH statement to limit the number of propagating rays.

When the program reaches the specified depth, random ray generation will cease and the values supplied for the constant ambient term will be used instead. Note that this depth value has no effect on the depth to which specularly reflected rays are traced (that is determined by the DEPTH statement in the main SCENE statement block). Also note that randomly directed rays impinging upon specular surfaces may generate additional reflected rays that need to be traced. To obtain a good approximation a large number of sample rays should be traced but generally this is impractical. Specifying 4 to 16 samples is reasonable. Specifying a maximum depth of 2 will limit the approximation to the first surface that is hit by an initial ray originating from the observer. The program uses a constant ambient term by default with both these values set to 0. The following is a sample AMBIENT definition:

```
AMBIENT  {
  INTENSITY  0.4
  COLOR  < 0.8, 0.8, 0.9 >
  DEPTH 2
  NSAMPLES 8
}
```

HAZE

The haze structure defines values which simulate atmospheric attenuation due to haze or fog. Haze provides a means of depth-cueing. Surface colors are blended towards the haze color by an amount that is a function of both the haze density and the distance the light has propagated. If haze is defined, then its color and intensity will also be used as the default background color when no objects in the scene are intersected by a ray. The syntax for the statement block is as follows:

```
HAZE
{
  INTENSITY floating-point
  COLOR    vector
  DENSITY   floating-point
}
```

The following discussion describes the statements within the HAZE statement block.

INTENSITY brightness_value
COLOR < r, g, b >

These two statements are used to specify the intensity and color of the haze. Generally the haze color is set to a light gray however using a dark blue or black is appropriate for nighttime scenes. If a SKY structure is defined then the haze is blended with its resultant color as well. When blending with the sky, the effect of the haze is attenuated with altitude. Setting the haze color to that of the horizon color works well.

DENSITY ρ

This statement allows one to specify how much light will be attenuated by haze for a given distance of ray propagation. Haze will be very noticeable at close distances if larger values of density are specified. The intensity is attenuated by:

$$I_x = I_0 \ e^{-\rho x}$$

where

I_0 - intensity with no attenuation
I_x - resultant intensity after the ray has propagated a distance x

If a value less than or equal to 0 is given then haze will have no effect. The default is to not apply haze attenuation while rendering.

The following is a sample HAZE definition:

```
HAZE
{
   INTENSITY 0.9
   DENSITY  0.0
   COLOR    < 0.9, 0.9, 0.9 >
}
```

SKY

The SKY structure defines a simulated sky for the upper hemisphere of the scene. It allows one to define an enclosing sphere for the environment and the position of a sun. Illumination from the sky is based on the CIE standard luminance functions for clear and overcast skys. One may specify the amount of overcast that is to be simulated. A defined sky is used in place of the background color when no objects are hit in the scene. If haze is specified then it will be blended with the sky color as well. One does not have to create any lights when a sky is defined since a light will be created to represent the sun. When using a simulated sky one should also create a ground plane to hide the lower hemisphere. If a ray does intersect the lower hemisphere then the resultant color is a mirrored image of the upper hemisphere. Note that the algorithm that implements sky rendering still needs a little work. The overall brightness of resultant images is very dependent on the amount of overcast. In addition specifying a completely overcast sky does not block out the sun it only affects the intensity of the sky's color. To compensate for this, one may specify a lower intensity value for the sun and disable its specular lighting calculation. This will create a sort of ambient light source. With a partially or totally clear sky the algorithm seems to work well and produces nice results when used in conjunction with a ray approximation to ambient lighting. The reflection of the sun and sky in mirrored surfaces looks good also.

The program does not simulate a sky by default. The syntax for the statement block is as follows:

```
SKY
{
  RADIUS          floating-point
  INTENSITY       floating-point
  ZENITHCOLOR     vector
  HORIZONCOLOR vector
  OVERCAST        floating-point

  SUN {...}
}
```

where {...} indicates a statement block

The following discussion describes the statements within the SKY statement block.

RADIUS radius_value

The specified radius determines how large the enclosing sphere is for the environment. The sphere is centered at the origin. The value specified should be large enough so that the sphere does enclose all objects in the scene. The radius should also be large enough so that the curvature of the sphere is not noticeable in any blended sky colors. If the specified radius is less than 1.0 or greater than 1.0e20 then it is set to the default value of 1.0e10.

INTENSITY brightness_value

The value specified determines the intensity of the sky at the zenith. The resultant intensity of any position in the sky is a blended value between the CIE overcast and clear sky function values based on the overcast parameter. The default value for the zenith intensity is 1.

ZENITHCOLOR <r,g,b>
HORIZONCOLOR <r,g,b>

These two statements allow specification of the sky color at the zenith and horizon. The resultant sky color at any position in the upper hemisphere is a blend between the zenith and horizon colors. The amount of blending is determined by the sky's position from the zenith. The default color for the zenith is <0 0,1> and for the horizon it is <0.9, 0.9, 1>.

OVERCAST amount_of

This statement specifies how overcast the sky is. The resultant intensity of any position in the sky is a blended value between the CIE overcast and clear sky functions based on this parameter. The value should be in the range {0..1}. The default overcast value is 0 which represents a clear sky.

SUN

When the sky is defined a light object representing a sun is automatically created. The sun is modelled as an infinite light source. An infinite source of light is considered to be originating from a point in space placed infinitely in the distance. All light rays incident to a surface are parallel. The values supplied for this structure define the intensity, color and direction from the sun. There are a variety of statements which allow specification of the sun's illumination direction in different ways. Although all may be specified only the last statement that would affect the direction is used. The statement block has the following syntax:

```
SUN
{
  INTENSITY  floating-point
  COLOR      color
  AZIMUTH    floating-point
  ZENITH     floating-point
  DIRECTION  vector
  FROM       vector
  TO         vector
  NOSPECULAR
  NOSHADOWS
}
```

The following discussion describes the statements within the SUN statement block.

INTENSITY brightness_value
COLOR <r,g,b>

These two statements are used to specify the intensity and color of the sun. The default intensity is 1.0 and the default color is < 1, 1, 1 >. A lower intensity value might be appropriate for more overcast skys.

DIRECTION <x,y,z>

This allows specification of the sun's illumination direction by a vector. The sun must be above the line of the horizon therefore the Z component of the direction should always be negative. The vector specified does not have to been in normalized form. The default value is < 0, 0, -1 >, in which case the sun is directly overhead at a zenith angle of 0 degrees.

FROM <x,y,z>
TO <x,y,z>

This allows specification of the sun direction by two points. The directed vector from the FROM point to the TO point defines the resultant direction. Using this form is easier than specifying a direction vector explicitly. The sun must be above the line of the horizon therefore the Z component of the FROM position should always be greater than the Z component of the TO position. The default FROM position is < 0, 0, 100 > and the initial TO position is the origin. Each time the parser encounters either statement the direction is recalculated so the order in which they are specified will be important.

AZIMUTH angle
ZENITH angle

This allows specification of the sun's illumination direction by its position in the sky. In this form of specification the illumination direction is from the sun's position towards the origin. The position is given by zenith and azimuth angles specified in degrees. The zenith angle is a rotation directed away from the positive Z axis towards the XY plane. The angle musty be positive with a magnitude greater than or equal to 0 and less than 90. An angle of 0 degrees positions the sun directly overhead while an angle close to 90 degrees places the sun on the horizon. The azimuth angle is a rotation directed away from the positive X axis in the XY plane. A positive azimuth angle is a rotation towards the positive Y axis and a negative azimuth angle is a rotation towards the negative Y axis. The angle must be in the range {-180 .. 180 }. An angle of 0 degrees places the sun aligned with the positive X axis and an angle of +/- 180 degrees places the sun aligned with the negative X axis.

NOSPECULAR

This is a boolean flag which will disable rendering calculations for specular highlights. This applies to the sun only and will not affect any other defined lights. Specifying nospecular may be appropriate for more overcast skys.

NOSHADOWS

This is a boolean flag which will disable shadow calculations for the sun even if shadows have been enabled for the entire scene(by the SHADOWS statement in the SCENE statement block or by the -S command line switch). This is appropriate for overcast skys where lighting is very diffuse and hard shadows created by a blocked sun would not be apparent. One may still want to specify shadows for the scene however if other illuminating sources have been created. For example, if there is dark overcast during the day, street lights generally come on in which case one would still want to see local shadowing in the area of the lights.

The following is a sample SKY definition:

```
SKY
{
  RADIUS      1.0e8
  INTENSITY   1.0
  ZENITHCOLOR < 0,0,0.9 >
```

```
    HORIZONCOLOR < 0.3,0.6,0.9 >
    OVERCAST    0.1

    SUN
    {
      INTENSITY 1.0
      COLOR    <1,1,1>
      AZIMUTH  -30
      ZENITH   20
      NOSPECULAR
    }
  }
```

MIRAGE

This structure defines the information required by the mirage algorithm which simulates the atmospheric effects of temperature on the propagation of light. The program only accounts for mirage effects on the initial rays sent from an observer into the scene. It does not affect specularly reflected rays, rays from light sources and any rays used in a diffuse interreflection approximation. Simply put, the program traces a "mirage" ray through vertically stacked horizontal layers of differing temperature. The mirage ray is represented by parabolic arc segments which are incrementally calculated as the ray propagates through each temperature layer. The ray segments follow a radial path directed away from the observer into the scene. Actual ray-object intersection tests are performed with the parabolic arc segments. The routine incrementally follows the ray's path through the temperature layers until an object in the scene is hit or a terminating condition is met. The termination criteria is supplied within this statement block. If a terminating condition is encountered before a ray-object intersection then a conventional straight ray tangent to the parabolic arc is traced from that point onward.

Simulating a mirage greatly increases image generation time but produces very interesting results. The defined objects in the scene become vertically distorted in size and may exhibit mirroring effects. The amount and type of apparent object distoration reproduced is dependent on a specified temperature profile and the placement of the observer and objects within the scene. Note that the observer must be vertically positioned somewhere within the defined temperature profile.

The syntax for the statement block is as follows:

```
MIRAGE
{
  SEGMENTS  integer
  DISTANCE  floating-point
  ELEVATION floating-point

  PROFILE
  {
    floating-point  floating-point
    floating-point  floating-point
            ...
  }
}
```

The following discussion describes the statements within the MIRAGE statement block.

SEGMENTS maximum_segments

As each parabolic ray segment is traced an accumulative count is maintained. This statement allows one to provide an upper limit on the number of ray segments traced after which mirage effects are no longer accounted for. From that point on a conventional straight ray is traced. This value is generally used as a safeguard in unforseen cases where all other terminating conditions used within the mirage simulation algorithm may fail. If the supplied value is less than 1 then the value is set to the default of 500.

DISTANCE maximum_distance

As each parabolic ray segment is traced an accumulative distance is maintained. This statement allows specification of a maximum distance after which mirage effects will no longer be accounted for. From that point on a conventional straight ray is traced. The value is assumed to be expressed in a base unit, for example meters as opposed to kilometers. If the value specified is less than 0 or greater than 1.0e20 then it will be set to the default value of 25000.

ELEVATION maximum_elevation

This statement allows specification of a maximum elevation a parabolic ray will be traced to, after which mirage effects will no longer be accounted for. From that point on a conventional straight ray is traced. The value is assumed to be expressed in a base unit, for example meters as opposed to kilometers. If the

value specified is greater than any elevation encountered in the defined temperature profile it will be ignored and the maximum found in the profile will be used instead. If the value specified is less than 0 or greater than 1.0e20 then it will be set to the default value of 500.0.

```
PROFILE
{
  z0  T0
  z1  T1
     ...
}
```

This statement allows specification of the temperature profile used by the mirage simulation algorithm. Each profile point is expressed by an elevation and a temperature. The entire profile is defined by a list of elevation - temperature pairs (z, T). Profile elevations are specified from low to high as one moves down the list. All elevations must be increasing. There is no restriction on the values specified for the temperatures.

Each individual profile point may be placed on the same line separated by at least one space or on separate lines. The number of points in the profile does not have to be specified beforehand however there should be at least 3 and no more than 200. There is no default temperature profile and an error will be issued if one is not defined within a MIRAGE statement block. The following is a sample MIRAGE definition:

```
MIRAGE
{
  DISTANCE  25000.0
  ELEVATION 100.0
  SEGMENTS  300

  PROFILE
  {
   /*  elevation  temperature */
    0.0    0.00
    5.0    0.30
    10.0   0.75
    11.0   0.88
    12.0   1.06
    14.0   1.50
    17.0   2.40
    20.0   3.85
  }
}
```

VIEW

The view statement block allows one to define the position, orientation and viewing direction of the observer within the scene. The observer is assumed to be looking through a pin-hole camera with a limited field of view. The field of view defines a frustrum which contains the elements of the scene that are visible to the observer. Statements within this block also define a viewing plane upon which the image is contained. The syntax for the statement block is as follows:

```
VIEW
{
  FROM     vector
  TO       vector
  FOV      floating-point
  TILT     floating-point
  WIDTH    integer
  HEIGHT   integer

  ASPECT   floating-point
  REGION   integer, integer, integer, integer
}
```

The following discussion describes the statements within the VIEW statement block.

FROM <x,y,z>

This statement is used to specify where the observer is within the scene by a position vector. If a mirage is being simulated then the observer must be vertically positioned somewhere within the defined temperature profile. The default position of the observer is < 20, 0, 6 >.

TO <x,y,z>

This statement is used to specify at what point the observer is looking at within the scene. If one is looking straight down from above, i.e. the line of sight is perpendicular to the XY plane directed along the -Z axis, then "up" is directed along the -X axis and "right" is along the +Y axis. If one is looking straight up from below then "up" is directed along the +X axis and "right" is directed along the +Y axis. The supplied point must be different from the specified observer's position. The observer is set to look at a default location of < 0, 0, 6 >.

FOV field_of_view

This statement is used to define the width of the viewing frustrum by an angle given in degrees. The frustrum is a three-dimensional volume containing all visible scene elements. Small field of view values will zoom in on far away scene elements similar to a telephoto lens. Large values will create a wide-angle lens affect and intermediate values produce a normal lens effect. The value must be in the range {0.5..175.0} degrees. The default field of view is 45.0 degrees.

TILT tilt_angle

This statement allows one to provide an angle in degrees which is used to rotate the camera about the viewing direction. A positive value will rotate the camera clockwise and a negative value will rotate the camera counter-clockwise. The effect produced is the same as that seen while looking out the front of an airplane as it is banked left or right. The value must be in the range of {-180.0..180.0} degrees. The default is 0.0 degrees of tilt.

WIDTH image_width
HEIGHT image_height

These two statements define the width and height of the final image in pixels. Both values must be in the range {1..1024}. The default width is 320 pixels and the default height is 200 pixels. These values may also be specified on the command line with -w and -h.

ASPECT aspect_ratio

This statement defines an aspect ratio correction factor for the image. Depending on the monitor used for viewing final images circles may appear as ellipses and squares may look like rectangles due to the differences in the way a pixel is displayed in the horizontal and vertical directions. Generally a displayed pixel is smaller in the horizontal direction than it is in the vertical direction therefore this value is used to scale the image horizontally. Program development was done on an Amiga with test images displayed at a maximum resolution of 320 pixels wide by 200 pixels high using an aspect ratio of 1.2. To display a perfect square 200 pixels high the width would be scaled to 240 pixels. If the aspect ratio provided is less than 1.0 it is set to 1.0. The default value is 1.

REGION top, left, bottom, right

While testing scene setups one may wish to ray trace a smaller, perhaps more

interesting area of an image. With this statement one defines the top left and bottom right corners of the area in pixels. The area must lie within the area of a full image defined by its width and height. For left and right values the valid range is {0..width-1} and for top and bottom values it is {0..height-1}. If any value exceeds the maximum or minimum allowable limit it will be set to that limit. If the specified right is less than the left it will be set to the left value and if the specified bottom is less than the top it will be set to the top value. The default area is 0, 0, 1023, 1023. Note that the REGION statement does not have to be used even though the defined image dimensions are less than the default area. The program will correct for it. Any specified region is ignored if any of the -w, -h, -s and -e command line switches are used.

The following is a sample VIEW definition:

```
VIEW
{
  FROM   < 100, 0, 5.5 >
  TO     < -10000, 0.001, 5.5001 >
  FOV    45.0
  TILT   20.0
  WIDTH  100
  HEIGHT 100

  ASPECT 1.2
  REGION 0, 0, 50, 50 /* only trace top left quarter */
}
```

7.5.5.2 Transformations

Transformations allow one to size, place and orient both objects and primitives within the scene. They may also be used to transform any textures which are to be applied to surfaces during rendering. All defined objects and primitives are initialized with a transformation matrix. A single transformation matrix may be composed of any number of scalings, translations and rotations combined in any order. Up to 20 transformations may be specified within the statement block at any one time. If more are required they may be specified in additional statement blocks. The syntax for the statement block is as follows:

```
TRANSFORM
{
  IDENTITY

  TRANSLATE vector
  TRANSX    floating-point
  TRANSY    floating-point
  TRANSZ    floating-point

  SCALE     vector
  SCALEX    floating-point
  SCALEY    floating-point
  SCALEZ    floating-point

  ROTATE    vector
  ROTX      floating-point
  ROTY      floating-point
  ROTZ      floating-point
      ...
}
```

Any one statement may appear more than once. The following discussion describes the statements within the TRANSFORM statement block.

IDENTITY

Upon entering a TRANSFORM statement block the transformation matrix being modified may already contain values from a previous definition (this is generally a consequence of TRANSFORM variable assignments). One may reinitialize the transformation matrix to represent an identity matrix with this statement. An identity transformation matrix has no effect on its corresponding object or

primitive. If this statement is used it should logically be the first within a statement block. If it is not specified first then any statements preceding it will be ignored.

TRANSLATE < x_amount, y_amount , z_amount >

This allows specification of a translation along all three coordinate axes. A positive value given for any of x, y or z will move a point that amount along the respective axis. If one does not need a translation along any one of the coordinate axes then a value of 0 should be supplied.

TRANSX x_amount
TRANSY y_amount
TRANSZ z_amount

Each of these statements performs the same function as the TRANSLATE statement but allows one to specify an amount for each axis individually. The order that successive translations are applied in does not have an effect on the final result therefore the TRANSLATE statement may be more useful. The more the merryier.

SCALE < x_size, y_size, z_size >

This allows one to scale differently along each of the three coordinate axes. The absolute magnitude of any of x, y or z must be greater than 0. Values less than 1 will compress points along the respective axis while values greater than one will cause an expansion. If a negative value is supplied for any of x, y or z then a reflection through a plane into another octant will occur. For example, a value of -1 for x would cause a reflection through the YZ plane. Specifying different values for each of the coordinate axes will result in nonuniform scaling for most objects but not all. For example a sphere will retain its shape but will have its center moved (if not at the origin) and its radius scaled by an amount not readily apparent.

If one does not need a scaling along any one of the coordinate axes then a value of 1.0 should be supplied.

SCALEX x_size
SCALEY y_size
SCALEZ z_size

Each of these statements performs the same function as the SCALE statement but allows one to specify an amount for each axis individually. The order that

successive scalings are applied does not have an effect on the final result therefore the SCALE statement may be more useful.

ROTATE < x_angle ,y_angle , z_angle >

This defines three separate rotations about each of the coordinate axes by three angles given in degrees. A positive rotation is in a counter-clockwise direction as one looks down the positive end of an axis. The rotations are applied in the order; X, followed by Y then Z. The order rotations are applied in does affect a transformed point's position therefore this statement may be too general for all purposes. If one does not need a rotation about any one of the coordinate axes then a value of 0 should be supplied.

ROTX x_angle
ROTY y_angle
ROTZ z_angle

Each of these statements allows specification of a rotation about a coordinate axis on an individual basis. Like the ROTATE statement, each angle is given in degrees and positive rotations are applied in a counter-clockwise direction. When applying successive rotations the final result is dependent on the order the rotations are specified in. This is due to the fact that combined rotations are calculated by matrix multiplication which is noncommutative. Use of these individual rotation statements offers more flexibility than the ROTATE statement alone. For example,

ROTY 20
ROTZ 30
ROTX 10

could not be expressed with a single ROTATE statement and would have to be specified as follows:

ROTATE < 0, 20, 0 >
ROTATE < 0, 0, 30 >
ROTATE < 10, 0, 0 >

which is a little more cumbersome.

The following is an example of a transform definition:

```
TRANSFORM
{
   TRANSX 25
   ROTZ  65
   ROTY  10
   SCALE  < 1, 2, 3 >
}
```

General Transformation Information

When a three-dimensional point within an object is transformed it is actually multiplied by the object's transformation matrix. If one applied each successive transformation individually to a point as they were specified they would be combined in the following manner:

$$P' = P \; M_c \; M_1 \; M_2 \; M_3 \; \ldots$$

where M_c is the object's current transformation matrix before entering the statement block and each M_j is a matrix representing the jth transformation specified within the statement block. The matrix nearest P generates the first transformation and the transformation farthest to the right, the last. Since no points are as yet available the transformations are combined in a pre-multiplied matrix M in the following order:

$$M = M_c \; (M_1 \; (M_2 \; M_3)))$$

To pre-multiply the matrices the transformations would normally have to be specified in the reverse order to that in which they are to be applied. To make transform specification less confusing the program utilizes a stack to temporarily save transformations as they are specified. As each transformation is parsed a matrix representing such is created and pushed onto a stack i.e. M_c will be pushed first, M_1 second and so on.

a) before statement block parsed

top of stack M_c

b) after M_1 parsed

top of stack M_1
M_c

c) after M_2 parsed

top of stack M_2
M_1
M_c

d) after M_3 parsed

top of stack M_3
M_2
M_1
M_c

After the last transformation has been specified the matrices are popped off the stack in the reverse order and multiplied i.e. M_3 is popped first, M_2 next and so on.

a) pop M_3 pre-multiplied $M = M_3\ I$

top of stack M_2
M_1
M_c

b) pop M_2 pre-multiplied $M = M_2\ (M_3\ I)$

top of stack M_1
M_c

c) pop M_1 pre-multiplied $M = M_1\ (M_2\ (M_3\ I))$

top of stack M_c

d) pop M_c pre-multiplied $M = M_c\ (M_1\ (M_2\ (M_3\ I)))$

top of stack empty

After all transformation matrices have been multiplied the object's current transformation matrix M_c is re-assigned to the pre-multiplied matrix M.

The stack can only hold 20 matrices at a time thus the reason only 20 transformations may be specified at a time. Specifying IDENTITY effectively clears the stack and sets an object's current transformation matrix M_c to I (an

identity matrix).

Since combined transformations are obtained with matrix muliplication which is noncommutative the order the transformations are specified in will affect the final result. For example, a cube centered at the origin that is first translated along the Y axis and then rotated about the Z axis will not be positioned and oriented the same as a cube that is first rotated about the Z axis and then translated along the Y axis.

When defining a transformation matrix it is not neccessary to specify all the possible combinations. Initially all transformation matrices represent identity matrices. An identity transformation matrix has no effect on its corresponding object or primitive.

If one is using scaling to just resize objects then it is advisable to apply the scaling to an object centered at the origin. If an object is also to be placed somewhere within the scene other than at the origin then it is sometimes helpful to apply the scaling before any rotations or translations are applied. For example if an object centered at the origin is first moved to < 0, 5, 0 > and then scaled by 2 along the Y axis it will move again to < 0, 10, 0 >. This may or may not be the desired result.

When scaling a sphere the radius will be resized by the greater of the three scale values.

When rotating objects about their center it is helpful if one uses < 0, 0, 0 > as the center. Applying the same rotations to an object centered other than the origin will produce different results.

What do I do w'dem?

Both primitives and objects have their own respective transformation matrices. The transformation defined for a primitive will affect that primitive only and the transformation defined for an object will affect all of its primitives. When modelling the objects within the scene one may approach the problem in either of two ways. With the first method one completely ignores transformations all together and specifies all geometrical information within the scene's global coordinate system. Since all objects are nothing more than abstract structures, each primitive is defined exactly as it will end up in scene. As an example let's say one were modelling an object composed of two vertically stacked spheres with the bottom sphere resting on the ground. The bottom sphere has a radius of 4 and the top sphere has a radius of 2. The spheres will be centered along an

axis positioned at < 5, 5, 0 >. To place this object within the scene one would specify a statement block as follows:

```
SOLID
{
   SPHERE /* bottom */
   {
      CENTER < 5, 5, 4 >
      RADIUS 4
   }

   SPHERE /* top */
   {
      CENTER < 5, 5, 10 >
      RADIUS 2
   }
}
```

The center of each sphere has to be calculated so that each will be positioned properly. The calculation is fairly trivial but what happens when we wish to model 10 of these stacked spheres positioned uniformly around a circle of arbitrary radius centered at some arbitrary point. The calculation of each sphere center is still not too difficult once we know the circle's radius and center however it becomes tedious. If we further create four more circles each with 10 stacked spheres and place them all on an incline then the calculations become overwhelming.

This example illustrates one of the major drawbacks to explicitly specifying all geometric information in the scene's global coordinate system. That is, it is very difficult to build complex objects in an organized hierarchial manner. This is a good example where transformations can help to organize modelled structures and simplify calculations. This leads to discussion of the alternative method one may use when modelling their objects within the scene.

Instead of explicitly specifying everything within the scene's global coordinate system each primitive or object is defined in its own local coordinate system and then subsequently placed or "instanced" into the appropriate parent coordinate system.

For example, each primitive is instanced into its containing object's coordinate system by that primitive's transformation matrix. Then each object is instanced into the scene's global coordinate system by that object's transformation matrix.

One may also perform geometrical transformations on a primitive or object before it is placed into the target coordinate system. We'll look at the previous stacked sphere example again this time making use of transformations. Both the bottom and top spheres will be defined in their own local coordinate system centered at the origin having a radius of 1. The containing object is also defined in its local coordinate system centered at the origin. To place this object within the scene one would now specify a statement block as follows:

```
SOLID
{
   SPHERE /* bottom */
   {
      CENTER < 0, 0, 0 > /* default sphere by the way */
      RADIUS 1

      TRANSFORM
      {
         /* geometric transformation to resize it */
         SCALE <4,4,4>

         /* position it on the ground within object */
         TRANZ 4
      }
   }

   SPHERE /* top */
   {
      CENTER < 0, 0, 0 >
      RADIUS 1

      TRANSFORM
      {
         /* geometric transformation to resize it */
         SCALE <2,2,2>

         /* position it on top of bottom sphere within object */
         TRANZ 10
      }
   }

   /* object transformation affecting all primitives */
   TRANSFORM
   {
      /* position object into the scene */
      TRANSLATE < 5, 5, 0 >
   }
}
```

Although the object's definition is now slightly longer it is more organized and

clearly indicates how the object has been built up from smaller components. To place 10 stacked spheres around a circle with a radius of 10 centered at <1,2,3> one could specify statement blocks similar to the following:

```
SOLID
{
   /* sphere definitions remain unchanged */

   TRANSFORM /* place into scene */
   {
     /* move spheres to desired radius first */
     TRANSLATE < 10, 0, 0 >
     /* rotate to position around circle */
     ROTZ multiple_of_36_degrees

     /* center it correctly */
     TRANSLATE < 1, 2, 3 >
   }
}
```

There is generally more than one way to construct a model to represent "something" within the scene. For example to model a city one may decide to make the city a single object and define primitives which represent all the buildings. One may also use multiple objects each representing a city block and define primitives that only represent buildings within that block. The program does not allow objects to be contained within other objects therefore the level to which one may create a structural hierarchy is limited.

Although using transformations helps to organize the modelled objects they do little to eliminate the amount of statements which must be specified to define the objects. The use of variables will significantly reduce the amount of statements one needs to supply when defining objects. As an example, an object modelling a set of wheel spokes will be defined by using a single POLYGONS variable and transformations. Each individual spoke will be represented by a stretched out cube. The spokes lie in the XY plane and are uniformily spaced around the wheel's axle represented by the Z axis.

The first step is to define a cube primitive centered at its origin. The cube must be resized and then positioned within the object. To resize each cube one applies a geometric scaling. To position each spoke within the wheel one should first move the spoke away from the Z axis leaving room for the axle and then rotate the spoke into its proper orientation. Since the axle is represented by the Z axis one should apply the rotations about that axis. Each spoke will initially lie

along the Y axis to serve as a reference before applying a rotation. The primitive transformations required would be the following:

a) resize cube	SCALE < x,y,z>
b) move away from axle	TRANSLATE < 0, y, 0 >
c) orientate	ROTZ angle

Since a primitive variable is being used one should put as much common information as possible into that variable's definition. The first two transformations are common to all the modelled spokes therefore they will be placed into the variable definition. The rotation is specific to each spoke so it will be specified as the primitive variable is assigned to the wheel object. The following sample statements illustrate how one could create the object with figures 7.1-7.6 showing the corresponding images.

```
/* define a generic cube centered at origin, figure 7.1 */
cube =
{
  VERTICES
  {
    8
    < 1, -1, 1 >      < 1,  1, 1 >
    < -1,  1, 1 >     < -1, -1, 1 >
    < 1, -1, -1 >     < 1,  1, -1 >
    < -1,  1, -1 >    < -1, -1, -1 >
  }

  FACETS
  {
    10
    0 4 5
    0 5 1
    1 5 6
    1 6 2
    2 6 7
    2 7 3
    3 7 4
    3 4 0
    0 1 2
    0 2 3
  }
} /* end of cube definition */
```

```
/* define a POLYGONS variable representing a spoke, figure 7.2 */

wheel_spoke = cube
{
  /* primitive transformations common to all spokes */
  TRANSFORM
  {
   /* scale the cube, geometric transform */
   SCALE < 0.4,1.2,0.4>
    /* move ends out along the Y axis
      to make room for axle */
   TRANSY 2.5
  }

} /* end of wheel spoke definition */

/* define a wheel object */

wheel =
{
  /* use default surface */

  /* Create wheel by instancing the spokes. Each instance is
     incrementally rotated another 45 degrees about the Z axis.
     The first spoke is lying along the y axis */

  POLYGONS wheel_spoke
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 45 } }
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 90 } }

  /* figure 7.3 illustrates the object at this point */
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 135 } }
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 180 } }
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 225 } }
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 270 } }
  POLYGONS wheel_spoke  { TRANSFORM { ROTX 315 } }

} /* end of wheel definition, figure 7.4 */

/* instance wheel into global scene coordinate system */
SOLID wheel
{
  /* object transformation */
  TRANSFORM
  {
    ROTY  -50 /* figure 7.5 */
    ROTX   35 /* figure 7.6 */
  }
}
```

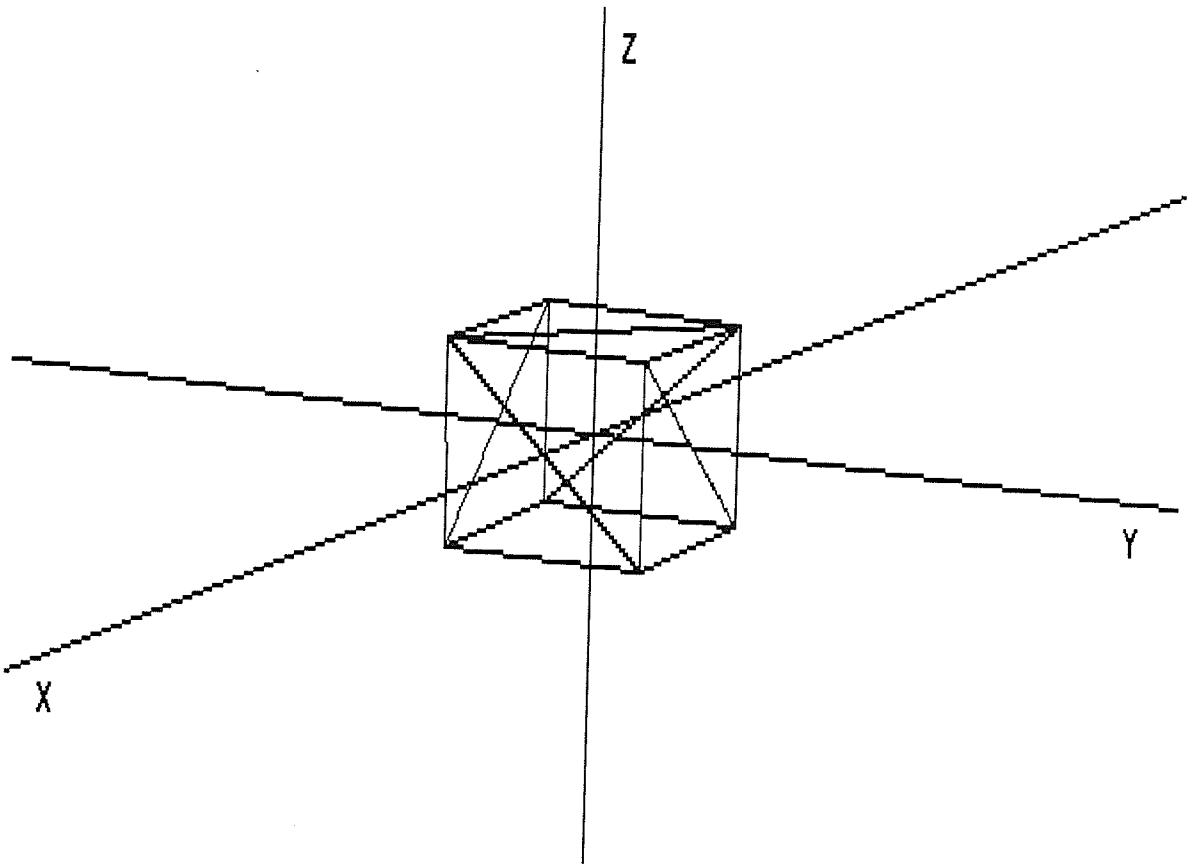

Figure 7.1 Cube centered at the origin

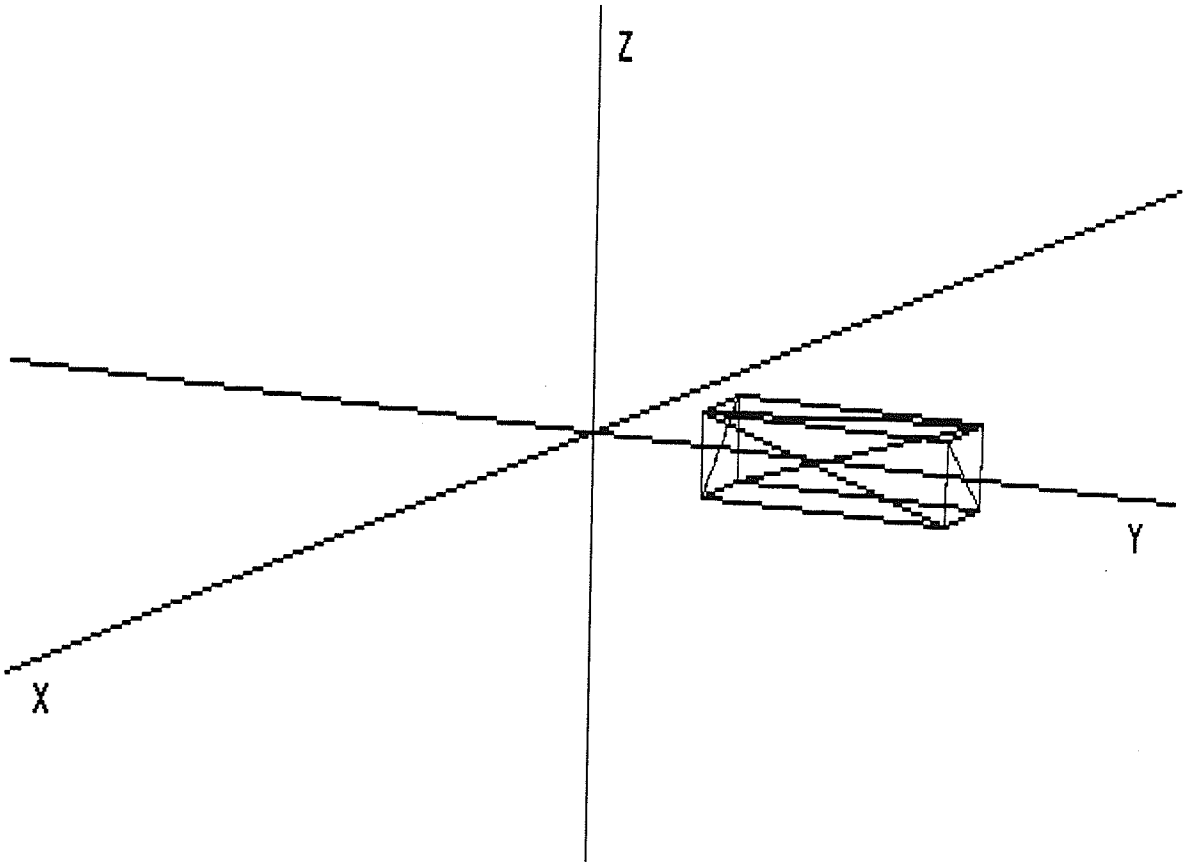

Figure 7.2 One spoke of the wheel

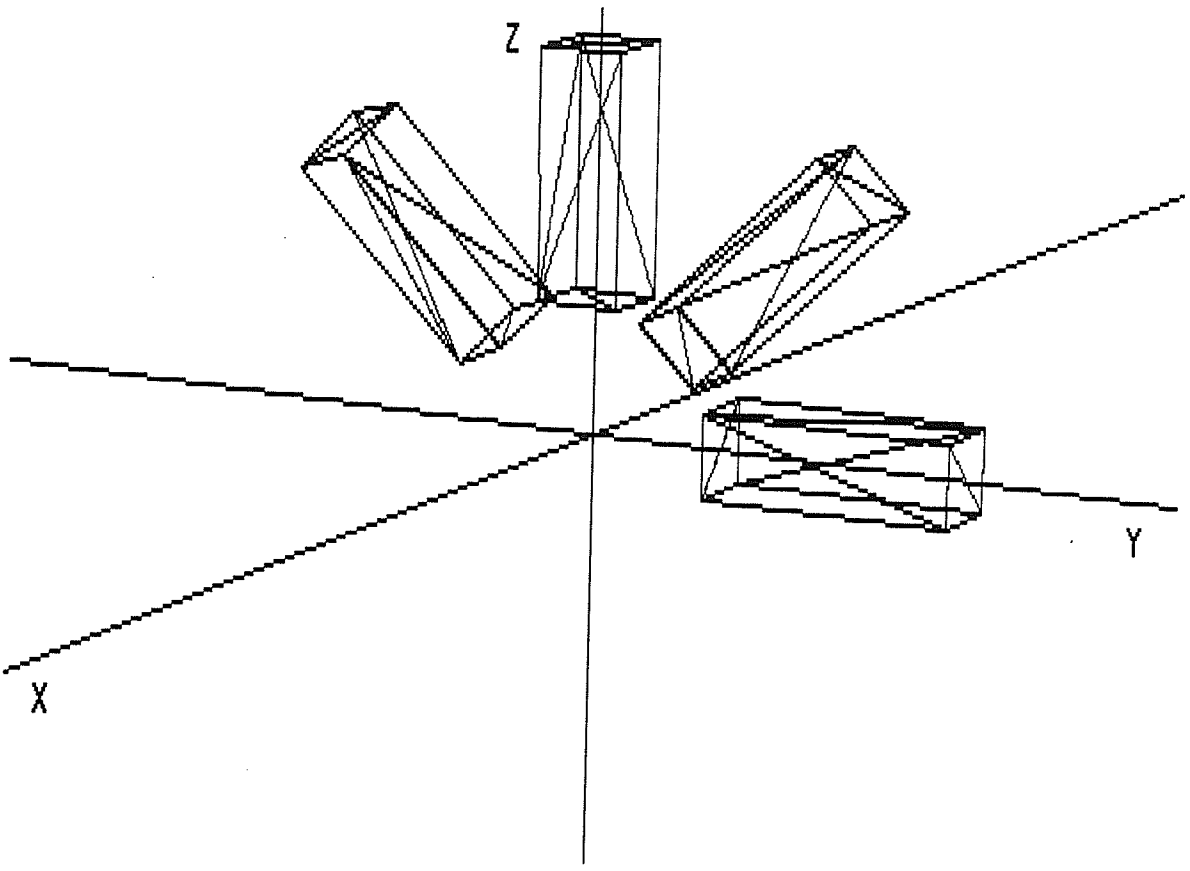

Figure 7.3 Four instances of the wheel spoke

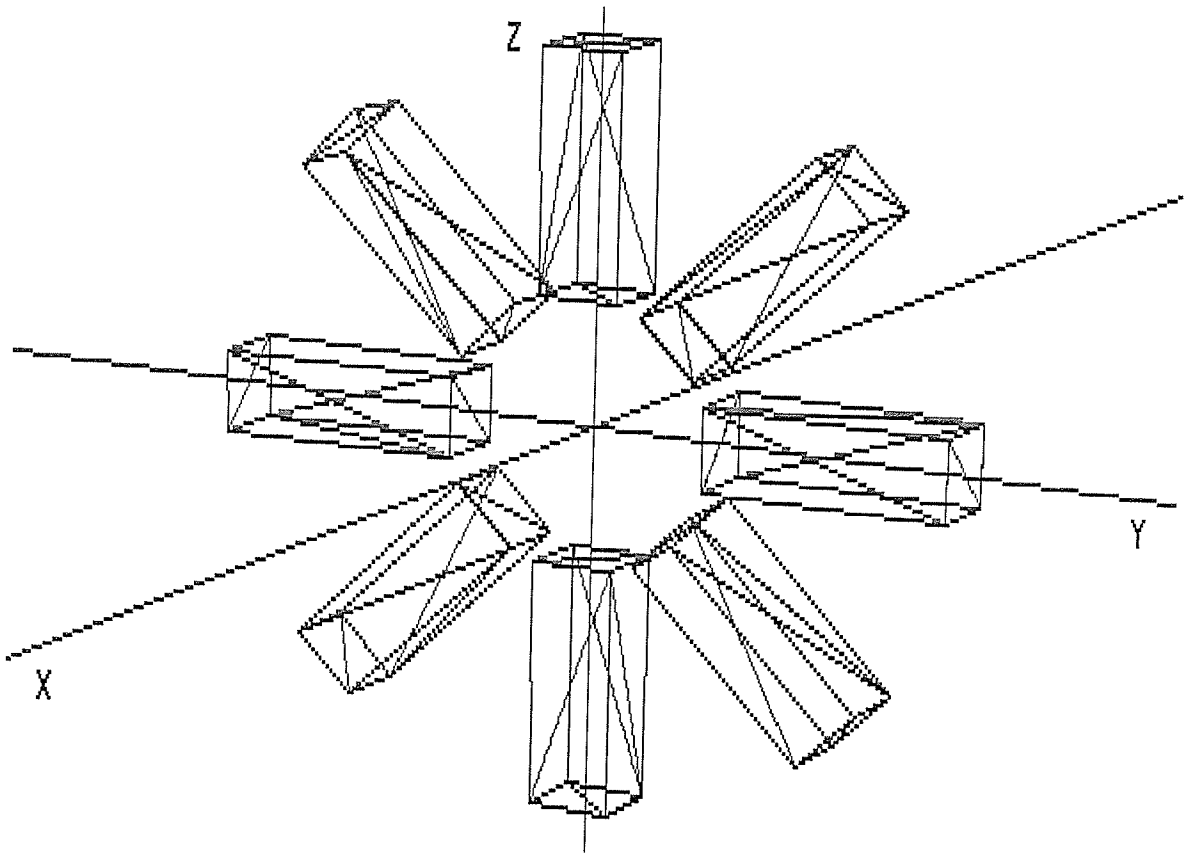

Figure 7.4 The complete wheel

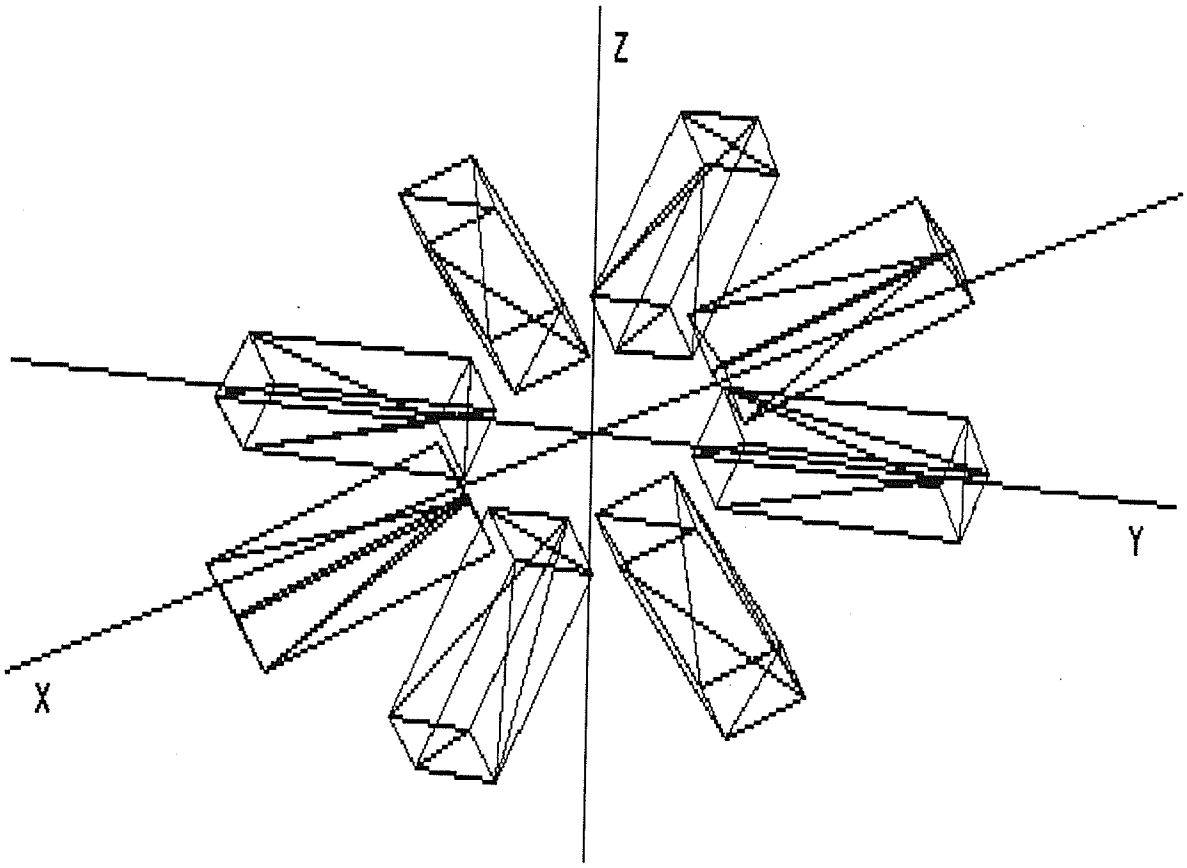

Figure 7.5 Wheel rotated -50 degrees about the Y axis

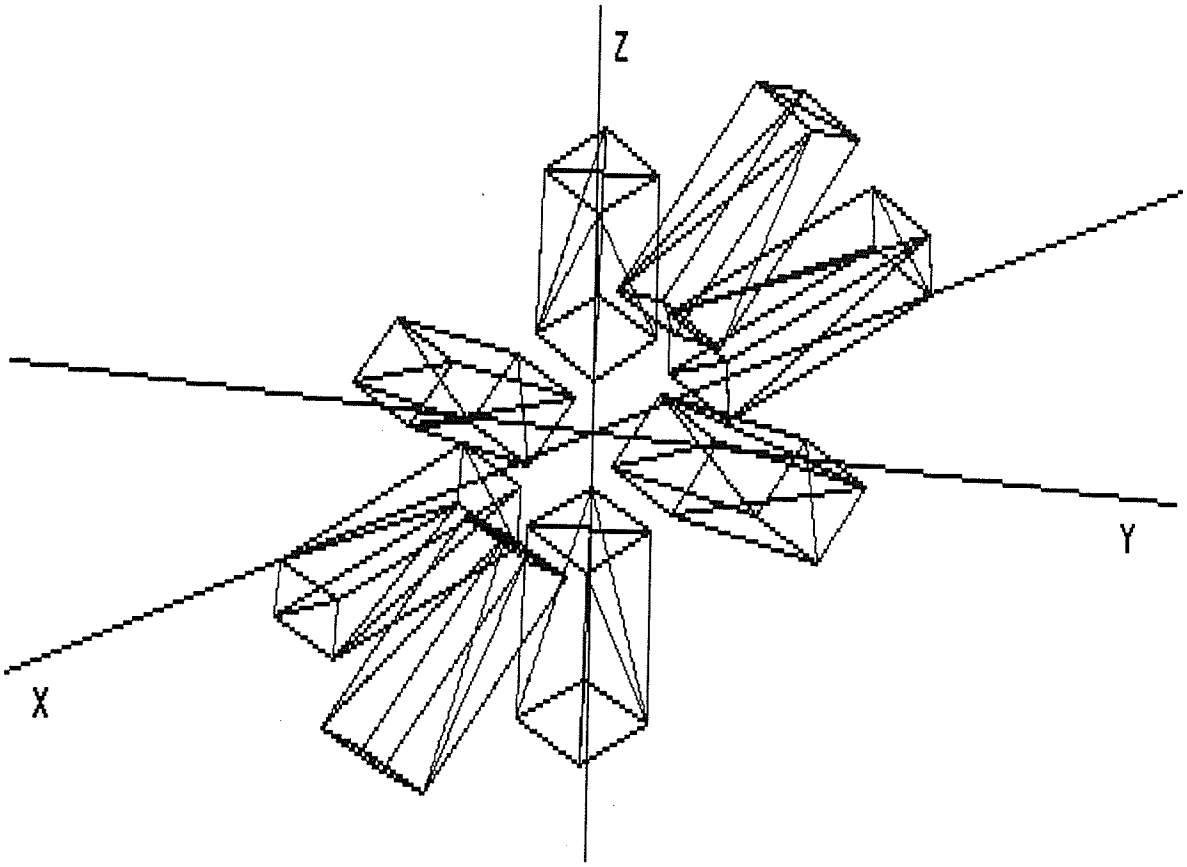

Figure 7.6 Wheel rotated 35 degrees about the X axis
to its final position

7.5.5.3 Primitives

Any visible object within the scene will have its physical shape defined by a list of basic primitives. The basic primitives are used as building blocks in creating more complex structures. All primitives contain geometric information only and it is not possible to specify individual attributes for each within an object. The defined attributes of the parent object are shared by all of its primitives. Each primitive also has its own transformation matrix. The following section describes the statement blocks for each of the primitives.

SPHERE

The sphere is defined by a center and a radius. The sphere also has an implicit uv coordinate system which is used by certain surface textures. If a sphere is rotated about its center then its reference frame will be rotated as well. A sphere is initialized with a right-handed coordinate frame aligned with the scene's XYZ coordinate axes. The sphere's u coordinate varies from 0 to 1 as one travels from the north to south poles. The sphere's v coordinate varies from 0 to 1 as one moves from the x axis in a counter-clockwise direction along the equator. The statement block is as follows:

```
SPHERE
{
   CENTER vector
   RADIUS floating-point

   TRANSFORM {... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the SPHERE statement block.

CENTER <x,y,z>

This statement is used to define the center of the sphere. The default center for a sphere is the origin.

RADIUS size

This statement is used to specify the radius of the sphere. The value supplied should be greater than 0. The default radius for the sphere is 1.

TRANSFORM

One may specify any number of transformations to size, place and orient the sphere directly into the scene's global coordinate system or into an object's local coordinate system. Any rotations will also affect the sphere's uv reference frame. The transformation matrix is by default set to an identity matrix and thus has no effect on the sphere.

The following is an example of a sphere definition:

```
SPHERE
{
   CENTER < 1, 2, 3 >
   RADIUS 5
}
```

PLANE

The plane is defined by three coplanar points and extends infinitely in space along its tangential directions. It may be useful as a ground or cloud plane. The statement block is:

```
PLANE
{
   VERTEX1 vector
   VERTEX2 vector
   VERTEX3 vector

   TRANSFORM {... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the PLANE statement block.

VERTEX1 $<x_1,y_1,z_1>$
VERTEX2 $<x_2,y_2,z_2>$
VERTEX3 $<x_3,y_3,z_3>$

The three supplied points should be unique, non-collinear and lie on the desired plane. Internallly a reference point *P*, a vector **U** and a vector **V** are stored to represent the plane. The plane's normal is determined by **U** x **V** using the right hand rule. The point supplied for VERTEX2 is used for *P*. **U** is the vector directed from VERTEX2 to VERTEX3 and **V** is the vector directed from VERTEX2 to VERTEX1. *P* has a default value of < 0, 0, 0 > and the default normal is < 0, 0, 1 >, i.e. the XY plane.

TRANSFORM

One may specify any number of transformations to size, place and orient the plane directly into the scene's global coordinate system or into an object's local coordinate system. The transformation matrix is by default set to an identity matrix and thus has no effect on the plane. The following is an example of a plane definition:

```
PLANE /* YZ plane */
{
    VERTEX1 < 0, 0, 0 >
    VERTEX2 < 0, 1, 0 >
    VERTEX3 < 0, 0, 1 >
}
```

TRIANGLE

The triangle is defined by three coplanar points forming a three sided polygon. The statement block is as follows:

```
TRIANGLE
{
   VERTEX1 vector
   VERTEX2 vector
   VERTEX3 vector
   TRANSFORM {... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the TRIANGLE statement block.

VERTEX1 $<x_1,y_1,z_1>$
VERTEX2 $<x_2,y_2,z_2>$
VERTEX3 $<x_3,y_3,z_3>$

The three supplied points should be unique, non-collinear and lie on the desired plane containing the triangle. The points may be given in any order however it is assumed they have been specified in a counter-clockwise direction. As with the plane primitive, a reference point *P*, a vector **U** and a vector **V** are internally stored to represent the triangle. The triangle's normal is determined by **U** x **V** using the right hand rule. The point supplied for VERTEX2 is used for *P*. **U** is the vector directed from VERTEX2 to VERTEX3 and **V** is the vector directed from VERTEX2 to VERTEX1. The default triangle is right angled lying in the XY plane with the reference point *P* set to < 0, 0, 0>. The defaults for each of the vertices 1, 2 and 3 are respectively < 0, 1, 0 >, < 0, 0, 0 > and < 1, 0, 0 >.

TRANSFORM

One may specify any number of transformations to size, place and orient the triangle directly into the scene's global coordinate system or into an object's local coordinate system. The transformation matrix is by default set to an identity matrix and thus has no effect on the triangle.

The following is an example of a triangle definition:

```
TRIANGLE /* lies in YZ plane, base on XY plane, apex at <0,0,5> */
{
   VERTEX1 < 0,  0,  5 >
   VERTEX2 < 0, -1,  0 >
   VERTEX3 < 0,  1,  0 >
}
```

RECTANGLE

The rectangle is defined by three points forming a four sided polygon whose opposite sides are parallel. The statement block is as follows:

```
RECTANGLE  {
   VERTEX1 vector
   VERTEX2 vector
   VERTEX3 vector
   TRANSFORM { ... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the RECTANGLE statement block.

VERTEX1 $<x_1,y_1,z_1>$
VERTEX2 $<x_2,y_2,z_2>$
VERTEX3 $<x_3,y_3,z_3>$

The three supplied points should be unique, non-collinear and lie on the desired plane containing the rectangle. The points may be given in any order however it is assumed they have been specified in a counter-clockwise direction. As with the plane primitive, a reference point *P*, a vector **U** and a vector **V** are internally stored to represent the rectangle. The rectangle's normal is determined by **U** x **V** using the right hand rule. The point supplied for VERTEX2 is used for *P*. **U** is the vector directed from VERTEX2 to VERTEX3 and **V** is the vector directed from VERTEX2 to VERTEX1. The default rectangle is a unit square lying in the XY plane with the reference point *P* set to < 0, 0, 0>. The defaults for each of the vertices 1, 2 and 3 are respectively < 0, 1, 0 >, < 0, 0, 0 > and < 1, 0, 0 >.

TRANSFORM

One may specify any number of transformations to size, place and orient the rectangle directly into the scene's global coordinate system or into an object's local coordinate system. The transformation matrix is by default set to an identity matrix and thus has no effect on the rectangle. The following is an example of a rectangle definition:

```
RECTANGLE /* lies in YZ plane , 5 units wide, 8 units high */
{
   VERTEX1 < 0, 0, 8 >
   VERTEX2 < 0, 0, 0 >
   VERTEX3 < 0, 5, 0 >
}
```

POLYGONS

A polygons primitive is composed of one or more triangular facets, each individually defined by 3 vertices. It allows one to construct basic shapes more complex than those provided by the program. One may create a library of shapes to be reused in their objects. Simple shapes such as a cone, cylinder or torus could be created or one may create a library of more complex shapes such as different tree or rock formations. The polygons primitive also allows for Phong normal interpolation. In order to apply Phong shading an average normal at each facet vertex must be supplied. The facet's normal at any position within its interior is an interpolation of the average normals at each of its three vertices. The normal interpolation creates a simulated smooth surface across adjacent facets. The syntax for the statement block is as follows:

```
POLYGONS
{
   PHONG
   NOPHONG

   VERTICES
   {
      integer
      vector
      vector
       ...
   }

   FACETS
   {
      integer
      integer integer integer
      integer integer integer
         ...
   }

   SFACETS
   {
      integer
      integer integer integer  vector vector vector
      integer integer integer  vector vector vector
                     ...
   }
```

```
    TRANSFORM { ... }
    TRANSFORM transform-identifier
    TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the POLYGONS statement block.

```
VERTICES
{
  n
  < x0, y0, z0 >   /* ID 0 */
  < x1, y1, z1 >   /* ID 1 */
  < x2, y2, z2 >   /* ID 2 */
    ...
  < xn-1, yn-1, zn-1 >   /* ID n-1 */
}
```

One uses this statement to list all the vertices that define the triangular facets of the primitive. The first parameter is a single integer indicating the number of vertices. At least 3 vertices must be specified. One may specify as many vertices as one wishes with the actual limit determined by the amount of available memory. This is then followed by the list of vertices given in vector form. The vectors may be placed on the same line or across multiple lines. Vertices are implicilty numbered in the order that they are listed. The first is given an ID of 0, the next an ID of 1 and so on. The last vertex specified is given an ID of n-1 with n being the total number of vertices. Each one of the listed vertices should represent one of any 3 vertices defining a triangular facet. A single vertex in this list may belong to one or more facets.

Only one VERTICES block may appear within the POLYGONS statement block. There are no default vertices created for a polygons primitive.

```
FACETS
{
  m
  vertex1_ID  vertex2_ID  vertex3_ID  /* facet 0 */
  vertex1_ID  vertex2_ID  vertex3_ID  /* facet 1 */
  ...
  vertex1_ID   vertex2_ID  vertex3_ID  /* facet m-1 */
}
```

One uses this statement to list all the facets forming the primitive. The first parameter is a single integer indicating the number of facets. At least one facet must be defined. One may specify as many facets as one wishes with the actual limit determined by the amount of available memory. This is then followed by a list of facet definitions. A single facet is defined by three integers, each representing a vertex in the VERTICES statement. For example if the first 3 vertices listed in the VERTICES statement are used to form the first facet then the first definition would be 0,1,2. The ordering of facet vertices is arbitrary. The vertices of a facet should all be unique and non-collinear.

Only one FACETS or SFACETS block may appear within the POLYGONS statement block. There are no default facets created for the polygons primitive.

```
SFACETS
{
  m
  /* facet 0 */
  vertex1_ID  vertex2_ID  vertex3_ID
  < N1x, N1y, N1z > < N2x, N2y, N2z > < N3x, N3y, N3z >

  /* facet 1 */
  vertex1_ID  vertex2_ID  vertex3_ID
  < N1x, N1y, N1z > < N2x, N2y, N2z > < N3x, N3y, N3z >
  ...
  /* facet m-1 */
  vertex1_ID  vertex2_ID  vertex3_ID
  < N1x, N1y, N1z > < N2x, N2y, N2z > < N3x, N3y, N3z >

}
```

This statement block is similar to the FACETS block however three average normals are also specified for each vertex. The normals are used with Phong shading. An average normal at any given vertex may be calculated by averaging the original normals of all those facets sharing that vertex. The specified average normals do not have to be in normalized form. If one is not using Phong normal interpolation (PHONG not specified or NOPHONG specified) then the ordering of facet vertices is arbitrary. If normal interpolation is being used (PHONG specified) then vertices should be specified in a counter-clockwise manner(the right hand rule is used by the program to determine the orientation of the original non-interpolated surface normals). The orientation of the supplied normals should also be defined in such a manner. One may use the -N command line option to verify the orientation of the supplied normals. The program will check

that the specified normals are oriented in the same direction as the original normal. In order that Phong interpolation does take effect one must still specify the PHONG keyword. Again the vertices of a facet should all be unique and non-collinear.

Only one SFACETS or FACETS block may appear within the POLYGONS statement block. There are no default facets created for the polygons primitive.

Appendix E provides a C program listing showing how one may create a POLYGONS primitive with SFACETS.

PHONG

Specifying this keyword alone will enable phong shading for the defined polygons. In order for this to have an effect, an SFACETS block must be specified. The program assumes that the structure has been defined with each specified average normal oriented in the correct direction. For example if one had constructed a closed surface such as a sphere then each supplied normal should be directed outward from the sphere's center. For this example one should also have the vertices ordered in such a manner that the original normal calculated by the program is directed outward from the sphere's center. Specifying vertices in a counter-clockwise direction will produce the correct results (normal determined by right hand rule).

The default is to not apply Phong normal interpolation.

NOPHONG

Specifying this keyword alone will disable Phong shading for the defined polygons. This is useful if one wishes to use the same SFACETS definition to produce both smooth shaded and polygonal shaded primitives. For example, if one had created a POLYGONS variable representing a cylinder using the SFACETS block then that variable could be assigned to different objects. During the assignment Phong Shading could be selectively enabled or disabled for a particular object. This eliminates the need to create two separate structures that represent the same shape; one without normals using a FACETS block and one with normals using the SFACETS block.

The default is to not apply Phong normal interpolation.

TRANSFORM

One may specify any number of transformations to size, place and orient the polygons structure directly into the scene's global coordinate system or into an object's local coordinate system. The transformation matrix is by default set to an identity matrix and thus has no effect on the defined vertices and facets.

The following is an example of a polygons definition for a pyramid:

```
POLYGONS
{
  VERTICES
  {     5
    < 0, 0, 1 >    < 1,-1, 0 >   < 1, 1, 0 >    < -1, 1, 0 >    < -1,-1, 0 >
  }

  FACETS
  {     4     0 1 2     0 2 3     0 3 4     0 4 1    }
}
```

TERRAIN

This a special form of polygons primitive which may be used to represent a two-dimensional grid of "elevations" or single-valued functions of two variables. It allows specification of the polygons structure by a two-dimensional array of vertices without the need to explicitly define each triangular facet, this will be done automatically. The plane containing the two-dimensional grid may be of any orientation but will generally be the XY plane. It is easier to first define the grid on the XY plane and then orientate it afterwards using a transformation. All specified "elevations" should be monotonicly increasing or decreasing as one moves laterally on the defining plane in either of the two grid directions. The grid itself may be non-uniformly spaced and dimensioned unequally however a line connecting any two grid points must not cross a line connecting any other two grid points. Any four grid points must define a four sided polygonally shaped cell. After all grid points have been parsed the program will create 2 facets per grid cell. A n by m grid will produce a polygons structure containing 2 (n-1)(m-1) triangular facets. Phong normal interpolation may also be specified to smooth out the defined surface. The syntax for the statement block is as follows:

```
TERRAIN
{
   PHONG

   VERTICES
   {
     integer
     integer

     vector
     vector
      ...
   }

   TRANSFORM { ... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

The following discussion describes the statements within the TERRAIN statement block.

VERTICES
{
  n m
  $< x_1, y_1, z > < x_1, y_2, z > ... < x_1, y_m, z >$
  $< x_2, y_1, z > < x_2, y_2, z > ... < x_2, y_m, z >$
    ...
  $< x_n, y_1, z > < x_n, y_2, z > ... < x_n, y_m, z >$
}

This statement is used to specify the grid dimensions, the lateral spacing between grid points and the "elevation" value at each grid point. For this discussion it is assumed the defining grid lies on the XY plane and all elevations are Z coordinate values. Grid rows {1..n} run along the X axis with coordinate values increasing. Grid columns {1..m} run along the Y axis with coordinate values increasing. When looking down at the grid from above, the top left corner represents the minimum X and Y coordinate values and the bottom right corner represents the maximum X and Y coordinate values.

The first two parameters specify the grid dimensions. The number of rows

should be given first followed by the number of columns. The supplied dimensions must be 2 or greater. There are no maximum allowable dimensions with the actual limit determined by the amount of available memory, This is followed by n m vectors each of which defines a grid point. The vectors should be specified in row-column order so that the first m vectors represent the first row of grid points along the Y direction.

Only one VERTICES block may be specified within the TERRAIN statement block. There are no default vertices created.

PHONG

Specifying this keyword alone will enable Phong shading for the defined grid surface. The program calculates all facets such that each facet normal is oriented in the Z direction. The average normals at each vertex are determined by gradient estimation.

TRANSFORM

One may specify any number of transformations to size, place and orient the terrain directly into the scene's global coordinate system or into an object's local coordinate system. The transformation matrix is by default set to an identity matrix and thus has no effect on the defined terrain.

Appendix E provides a listing of a C program that creates a TERRAIN primitive.

7.5.5.4 Object Attributes

Each object has associated with it a set of attributes which define its charcteristics during rendering. For a solid, the attributes describe the object's surface appearance as it interacts with propagating rays from direct light sources or other solids. For a light the attributes primarily describe how the light is to illuminate the solids within the scene. If the light is visible then the attributes also describe how the light is to appear.

SURFACE

To render a solid object the program needs to know certain information about its surface such as its color, how diffuse or specular it is, whether it is textured etc. This information is defined by surface attributes. The statement block is as follows:

```
SURFACE
{
  COLOR      vector
  SPECULAR floating-point
  ROUGH      floating-point
  HARD       floating-point
  METALLIC   floating-point
  REFINDEX  floating-point
  FUZZ       floating-point
  AMBIENT    floating-point

  TEXTURE { ... }
}
```

The following discussion describes the statements within the SURFACE statement block.

COLOR <r,g,b>

This defines the diffuse color of a solid's surface when illuminated by white light.

SPECULAR amount

This defines the proportion of specular and diffuse components in light that is reflected from a surface.

Light reflecting from a surface may be composed of both diffuse and specular components. Totally diffuse surfaces such as a latex painted wall appear equally bright when viewed from all directions because light is reflected with equal intensity in all directions. A shiny surface such as plastic will exhibit both diffuse and specular reflection. A bright light incident on a plastic object will produce a highlight caused by specular reflection while the rest of its appearance is due to diffuse reflection. Reflections from surfaces such as metal are primarily specular. A small value will produce a very dull diffuse surface. The given value should be in the range {0..1}. The default value is 0.2.

ROUGH roughness

This defines how rough the surface is on a microscopic level. It controls the degree to which a shiny surface will reflect light unequally in different directions thereby affecting the appearance of specular highlights when viewed from different positions. Incident light on a perfect reflector such as a mirror is only reflected in one direction in which case the specular highlight appears very small. When looking at a non-perfect reflector such as a shiny apple one will notice that the highlight is more spread out and appears to move when viewed from different positions. Specifying a small roughness value will simulate a mirror like surface. The supplied value should be in the range {0..1}. The default is 0.5.

HARD hardness

This defines to what degree a surface will reflect incident light due to interobject reflections. One may think of this parameter as a measure of light absorption. Large values of hardness will model the surface as a perfect reflector such as a mirror whereas smaller values will simulate more of a plastic reflector. This parameter works in conjunction with the specified roughness parameter. For example, if one were to model a mirror surface one should specify a value of 1 for the hardness and a value of 0 for the roughness. The surface must also be slightly specular for this to have an effect. All supplied values should be in the range {0..1}. The default value is 0.5.

METALLIC amount

This defines how metallic the surface is affecting the color of specular highlights. For non-metallic surfaces the specular highlight will generally be the color of the incident light. The color of specularly reflected light from a metal will be that of its surface. A value of 0 will produce highlights colored to that of the incident light and a value of 1 will produce highlights colored the same as the surface. Values inbetween will produce highlights that are a blend between the

two colors. When simulating a metallic surface a large specular value should be used. All supplied values should be in the range {0..1}. The default value is 0.

REFINDEX refractive_index

This parameter works in conjunction with the metal parameter. It defines an average index of refraction which is used in the Fresnel equation to approximate a shift in specular color based on the angle of incident light. If one is viewing a surface almost 180 degrees in line with the incident light and that light is grazing the surface then the color of specular highlights will shift towards that of the light. The surface must be metallic for this to have an effect. The supplied value must be greater than or equal to 1. The default is 1.

FUZZ fuzziness

This allows one to specify a very rough surface. The supplied value is used to determine how much random deviation should be applied to the surface normal. The supplied value should be in the range {0..1}. The default value is 0.

AMBIENT amount

This defines how much global ambient light should be applied to the surface. This value is used with the constant ambient information defined within the AMBIENT statement block. This value will not affect the ray tracing approximation to diffuse interreflection. The supplied value should be in the range {0..1}. The default value is 0.1.

TEXTURE

Each defined surface may have a procedural texture applied to it which may be used to modify its surface color or surface normal usually adding some interest to its rendered appearance. One may texture the surface so that it appears as a checker board, piece of wood, marble, spotted, blotchy etc. Modifying the surface normal allows one to simulate bumpy or rippling type surfaces. Any number of textures may be applied to a surface and each texture may be individually scaled, orientated and positioned as it is applied. Textures are applied in the order specified. The statement block is as follows:

```
TEXTURE
{
  CHECKER { ... }   WOOD   { ... }   MARBLE  { ... }   SPOTTED { ... }
  GRANITE { ... }   CLOUDS { ... }   SNOW    { ... }   GRADIENT { ... }
  BUMPS   { ... }   WAVES  { ... }   RIPPLES { ... }   UBLEND  { ... }
  VBLEND  { ... }
    ...
}
```

Each of the above statement blocks may be specified more than once. Within any one block there are a number of statements which may be used to specify texture parameters some being common to all textures some very specific. All may be specified whether a particular texture uses them or not. Most textures have default values but some will require additional information. The valid statements within each block will be described first followed by a description of each texture and its applicable parameters. The statements within any individual texture block may comprise one or more of the following:

```
texture_keyword
{
  LOCAL

  COLOR     vector
  COLORS    { ... }

  TURBULENCE floating-point
  SQUEEZE    floating-point
  SCALE      floating-point
  SLOPE      floating-point
  ELEVATION  floating-point

  WAVE { ... }
    ...

  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
    ...
}
```

The following is a description of each statement.

LOCAL

This enables one to make texturing local to the object. Normally all textures are applied using the scene's global coordinate system as reference. For example, if one applies a marble texture to an object composed of a single sphere at one position and then moves that object to another position the resulting marble texture on the sphere will appear different for the same specified parameters. Specifying local will enable the texture to track the object which is useful for animation purposes. The default is to apply global texturing.

COLOR <r,g,b>

This defines the texture's color. When a texture modifies a surface the resulting color will generally be a blend between the specified surface color and the texture color based on the texture function. The default color is white, i.e. <1,1,1>.

TURBULENCE amount
SQUEEZE amount
SCALE amount
ELEVATION amount
SLOPE angle

Each of these define values which are used to scale specific features of each texture. The valid ranges will depend on each texture. Each has a default value of 1 except for SLOPE which has a value of 75.

COLORS

Normally the resultant surface color is a blended value between the surface's specified color and the supplied texture color based on the texture function. One may also define a range of colors which may be used for the surface color. The statement block is as follows:

COLORS
{
u_1 v_1 <r,g,b> <r,g,b>
u_2 v_2 <r,g,b> <r,g,b>
....

u_n v_n <r,g,b> <r,g,b>
}

This block is used to specify a variable texture color based on ranges of blend colors. The parameters u and v define how the interval { 0..1 } should be divided into the different ranges. The parameter u defines the start of any given range and v defines its end. One also supplies two colors for the start and end of each blend range. All u and v values should be be given such that:

$$0 <= u_1 < v_1 <= u_2 < v_2 <= u_3 < v_3 .... <= u_n < v_n <= 1$$

When determining the color that is assigned to a texture, a value t is first obtained through a texture function. The value is then compared against the supplied ranges to find which interval it lies within. The ith range used is the one such that, $u_i <= t < v_i$. The resulting color will be a blend between the defined colors at the interval end points based on on the value of t.

It is usually best to keep the ranges continuous, i.e. $v_1 = u_2$, $v_2 = u_3$ and to supply the same color value at interval endpoints. If the ranges are discontinuous then the texture's color will be undefined in those areas and the assigned surface color will be unmodified.

One may define up to 20 different color ranges per texture. The number of ranges that are to be defined does not have to be specified beforehand. There are no default color ranges assigned to textures.

The following is an example of how one may define blend ranges:

```
COLORS
{
  0.0  0.25  < 1.0, 0.0, 0.0 >  < 1.0, 0.25, 0.0 >
  0.25  0.5  < 1.0, 0.25, 0.0 >  < 1.0, 0.5, 0.0 >
  0.5  0.75  < 1.0, 0.5, 0.0 >  < 1.0, 0.75, 0.0 >
  0.75  1.0  < 1.0, 0.75, 0.0 >  < 1.0, 1.0, 0.0 >
}
```

WAVE

Textures such as waves and ripples modify a surface normal using wave sources. A wave source defines a spherical wave front emanating in all directions from a specified point. One may specify the frequency (and thus wavelength) , amplitude and phase of the waves. One may also attenuate the amplitude of the waves with distance. One may create any number of wave sources for each texture. There are no wave sources created for textures by default. The statement block is as follows:

```
WAVE
{
  CENTER      vector
  AMPLITUDE floating-point
  FREQ        floating-point
  PHASE       floating-point
  DAMP        floating-point
}
```

The following is a description of each statement in the WAVE statement block.

CENTER <x,y,z>

This statement is used to define the origin of the wave. The point may be any position within the scene. When memory for a wave source is first allocated the center is initialized to <0,0,0>.

AMPLITUDE height

This defines the height of the waves. The supplied value should be greater than 0. A wave is initialized with an amplitude of 1.

FREQ frequency

This defines the frequency of the wave. The given value may be any number. The value given should be greater than 0. All wave sources are initialized with a frequency of 1.

PHASE phase_shift

This defines the amount of phase shift that is to be applied to a wave. It is useful for animation purposes where one may create waves which appear to

move with time. The value given may be any number. The wave source is initialized with a phase shift of 0.

DAMP amount

This specifies how much attenuation should be applied to the amplitude of the wave at any given distance from its center *C*. The amplitude A at a distance d from the wave's center is:

$$A = A_0 \; e^{-\text{amount } d}$$

where A_0 is the specified amplitude of the wave and d is given by

$$d = | P - C | \; \text{FREQ} + \text{PHASE}$$

The value given must be greater than or equal to 0 to have an effect. The initialized value is 0.

TRANSFORM

One may specify any number of scalings, rotations or translations to position the texture on a surface. Most textures have a default scale or orientation that may be changed by its own set of transformations. For example, applying the default checker texture to the XY plane will produce a checkerboard pattern whose squares are of unit length and whose sides are parallel to the X and Y axes. One may specify a scaling to resize the squares and a rotation to re-orient the patterns. If a scale of < 4, 6, 1 > were given, then the checker pattern would alternate colors every 4 units along the X axis and every 6 units along the Y axis. A translation is useful for a checkerboard since roundoff errors may produce spurious speckles. The checker texture is three-dimensional therefore all coordinate values affect its appearance. If it is placed in the XY plane, the z value at any given x and y coordinate may occasionally alternate between small positive and negative values thus producing speckles. Translating the texture along the Z axis by a small amount in either direction will eliminate sign changes due to roundoff errors.

The statement block is identical to that specified for objects and primitives.

Texture Descriptions

The following discussion will describe each individual texture and its applicable parameters.

CHECKER

This will allow one to create a checker board pattern on an object's surface. The patterns are initially aligned with the X, Y and Z coordinate axes with colors alternating between the texture COLOR and the surface color. The size and orientation of the patterns may be changed through the use of transformations. The applicable statements are as follows:

```
CHECKER
{
  LOCAL
  COLOR <r,g,b>
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
    ...
}
```

WOOD

This will allow one to create a wood-like texture on an object's surface. The rings are formed by concentric circles centered about the Z axis lying parallel to the XY plane, i.e.the grain runs parallel to the Z axis. The color of the rings is determined by the provided texture. COLOR(s). More natural looking rings may be created by specifying a TURBULENCE value greater than 0. This will deform the rings changing their shape and spacing. The rings may be made thinner by specifying a SQUEEZE amount greater than 0. The size/spacing of the rings and orientation of the grain may be changed through the use of transformations. The applicable statements are as follows:

```
WOOD
{
  LOCAL
  COLOR  <r,g,b>
  COLORS { ... }
  TURBULENCE amount
  SQUEEZE      amount
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
    ...
}
```

MARBLE

This will allow one to create a marble-like texture on an object's surface. The veins vary periodically in the Y direction running length-wise parallel to the Z axis. The vein color is determined by the provided texture COLOR(s). More natural looking rings may be created by specifying a TURBULENCE value greater than 0. This will deform the veins changing their shape and spacing. The veins may be made thinner by specifying a SQUEEZE amount greater than 0. The size, spacing and orientation of veins may be changed through the use of transformations. The applicable statements are as follows:

```
MARBLE
{
  LOCAL
  COLOR <r,g,b>
  COLORS { ... }
  TURBULENCE amount
  SQUEEZE      amount
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }    ...
}
```

SPOTTED

This will allow one to create a spotted or blotching looking texture on an object's surface. The color of the blotches is determined by the texture COLOR(s). Specifying a SCALE value greater than 0 will increase the density of the blotches. The size, spacing and orientation of the blotches may be changed through the use of transformations. The applicable statements are as follows:

```
SPOTTED
{
  LOCAL
  COLOR <r,g,b>
  COLORS { ... }
  SCALE amount
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
     ...
}
```

GRANITE

This will allow one to create a turbulent or rocky looking texture on an object's surface. The resultant color is a blended value between the surface color and the texture COLOR(s) dependent on the amount of TURBULENCE that is specified. The value must be greater than 0 to have an effect. The texture may be changed through the use of transformations. The applicable statements are as follows:

```
GRANITE
{
  LOCAL
  COLOR  <r,g,b>
  COLORS { ... }
  TURBULENCE amount
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
     ...
}
```

CLOUDS

This will allow one to create a cloudy looking texture on an object's surface. Initially the cloud patterns are scattered laterally parallel to the XY plane. A TURBULENCE value greater than 0 will affect the shape and arrangement of the clouds. A SQUEEZE value greater than 0 will affect the density of the clouds. The color of the cloud patterns is determined by the texture COLOR(s). The size, spacing and orientation of the clouds may also be changed through the use of transformations. The applicable statements are as follows:

```
CLOUDS
{
  LOCAL
  COLOR  <r,g,b>
  COLORS { ... }
  TURBULENCE amount
  SQUEEZE      amount
  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
     ...
}
```

SNOW

This will allow one to create the effect of snow falling on an object's surface. The the snow falls in the -Z direction and plops down on top of objects. Snow will not be allowed to rest on a FLAT surface below a specified ELEVATION. The value of ELEVATION is specified as a fraction of the distance between the maximum and minimum Z extents of the scene. The given value should be greater than or equal to 0 and less than 1. For example, if the maximum Z is 80, the minimum Z is -20 and ELEVATION is 0.7, then snow will only be visible at any given surface point above 50. The value is determined by

$$(Z_{max} - Z_{min})\ \text{ELEVATION}\ + Z_{min}.$$

The scene's maximum and minimum extents are calculated therefore it may be helpful to use the -D command line option (lists all scene information) to get some idea as to what elevation value to specify (planes are not included when determining the scene's extents).

Snow may also slide down sloped surfaces in which case it may settle a little below the minimum elevation (for a flat surface). A SCALE value greater than 0 will determine how much to account for slope. The value should be less than 1 to produce more subtle effects. If a surface is too steep then snow will just slide off. The angle given in degrees for SLOPE will determine the steepest incline which may hold snow before it slides off. The value should be greater than 0 and less then 90 degrees.

To add a bit more realism to the texture, a TURBULENCE value greater than 0 may be given to add randomness to where the snow eventually falls. A turbulence value of 1 will shift the given elevation by +/- 50%. Any transformation may be applied to the snow texture however it only affects the randomness applied. Scaling seems to be the most practical. The applicable statements are as follows:

```
SNOW
{
  LOCAL

  SCALE          amount
  SLOPE          angle
  TURBULENCE amount
  ELEVATION    relative_height
```

```
    COLOR    <r,g,b>

    TRANSFORM { ... }
    TRANSFORM transform-identifier
    TRANSFORM transform-identifier { ... }
        ...
  }
```

GRADIENT

This allows one to vary the color of an object's surface with elevation. This may be useful with objects such as a mountain . The texture works best with a supplied range of COLORS. As with the snow texture, the surface elevation used in the calculation is a fraction of the distance between the maximum and minimum Z extents of the scene. Randomness may be added to the texture by specifying a TURBULENCE value greater than 0. A turbulence value of 1 will shift the given elevation by +/- 50%. The applicable statements are as follows:

```
  GRADIENT
  {
    LOCAL

    COLOR  <r,g,b>
    COLORS { ... }

    TURBULENCE amount

    TRANSFORM { ... }
    TRANSFORM transform-identifier
    TRANSFORM transform-identifier { ... }
        ...
  }
```

BUMPS

This allows one to create a variety of different types of bumpy surfaces by modifying an object's surface normal. Specifying SCALE and SQUEEZE parameters greater than 0 will change the appearance of the bumps. Have not used this much so can't give any suggestions other than to play around with it. A transformation may be applied to the texture. The applicable statements are as follows:

```
BUMPS
{
  LOCAL

  COLOR <r,g,b>
  COLORS { ... }

  SCALE          amount
  TURBULENCE     amount

  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
     ...
}

RIPPLES
WAVES
```

This allows one to create a surface with a wave-like appearance by modifying the surface normal. In order to have an effect on the surface's appearance at least one WAVE source must be defined. The only difference between these two textures is that the wave sources used in a WAVES texture have their amplitudes scaled by frequency. Higher frequency waves have a smaller height. The texture is not just limited to planar objects. The wave sources may be placed any where within the scene and any shaped object may have a wave-like texture applied. One may deform the wave patterns using scaling. The applicable statements are as follows:

```
RIPPLES [ WAVES ]
{
  LOCAL

  WAVE { ... }
     ...

  TRANSFORM { ... }
  TRANSFORM transform-identifier
  TRANSFORM transform-identifier { ... }
     ...
}
```

UBLEND
VBLEND

All primitives except a plane have a local uv coordinate system. For spheres the u coordinate runs from its north to south poles and v runs around the equator in a counter-clockwise direction from the X axis. For the planar primitives u runs along the **U** vector and v runs along the **V** vector with both starting at the reference point *P*. All uv coordinates vary from 0 to 1. This texture will vary the surface color of an object by using either of the u and v values as blend parameters. If a range of COLORS is provided then u or v will be used to determine a resultant color from a defined range. If only a single COLOR is provided then u or v will be used to blend between the surface and texture color in the HSV color coordinate system. The resultant color will be based on a blend between the corresponding hues of the surface and texture. Note that all colors are still specified in the RGB color system and the conversion to and from the HSV color system is done by the program. Although this texture is defined for an object it actually affects each of its individual primitives differently and is not all that practical. It was used primarily with objects composed of single primitives while testing the mirage algorithm (it would of been more useful if the uv coordinates were defined over a bounding sphere which enclosed all primitives). Specifying a transformation will have NO effect on the resultant texture. The applicable statements are:

```
UBLEND [ VBLEND ]
{
  COLOR <r,g,b>
  COLORS { ... }
}
```

SOURCE

When rendering a solid object a certain portion of its resultant surface color is attributable to illumination from direct lighting. All defined light objects contribute to this calculation. A solid's surface will only be illuminated if it is facing the source of illumination and is not shadowed by another solid. For each light defined, the program needs to know its color, intensity, whether it represents a point or infinitely distant source, whether its intensity should be attenuated with distance etc. This information is defined by source attributes. The statement block is as follows:

```
LIGHT
{
  SOURCE
  {
    COLOR          vector
    INTENSITY      floating-point

    INFINITE
    POINT

    ATTENUATION floating-point

    ANGLE          floating-point
    EXPONENT       floating-point

    FROM           vector
    TO             vector
    DIRECTION      vector
    ZENITH         floating-point
    AZIMUTH        floating-point

    NOSHADOWS
    NOSPECULAR

}
```

The following discussion describes the statements within the SOURCE statement block.

COLOR <r,g,b>
INTENSITY brightness_value

These two statements are used to specify the intensity and color of the light. The default intensity is 1.0 and the default color is <1,1,1>, i.e. a nice bright white light.

INFINITE

This will cause the illumination source to be modelled as a point in space placed infinitely in the distance. All illuminating rays falling on a solid's surface are parallel to the source's illuminating direction. An infinite source does not have its intensity reduced by distance. When specifying an infinite source one should also make sure to supply the direction it is shining in. This is the default type.

POINT

This will cause the illumination source to be modelled as a point in space placed at some definite location within the scene. A point source uniformly radiates in all directions and may have its intensity vary with distance.

ATTENUATION α

This specifies how the intensity of a point source should vary by the distance d it is from an illuminated surface according to:

$$I = I_0 / (1 + d^{\alpha)}$$

where I_0 is the specified intensity of the point source.

If the given value is equal to 0 then distance will have no effect at all. Specifying a value of 2 will simulate squared attenuation reducing the intensity by the amount $1 / d^2$. This is how light actually varies physically however using squared attenuation has a very pronounced effect on rendered scenes. Specifying a value of 1 will simulate linear attenuation reducing the intensity by the amount 1 / d. This will generally produce better results. The default value is 0, resulting in no attenuation.

ANGLE β

This allows one to simulate a very directional spotlight with a point source. The angle given in degrees defines a cone whose apex is at the point source and

whose axis lies along the illuminating direction creating a volume of "visible" light. Only surfaces positioned within this volume will be illuminated. The supplied angle must be greater than 0 and less than or equal to 180 degres. The default value of 180 degrees makes the source visible from all directions.

EXPONENT ρ

This allows one to simulate a highly directional spot light or a diffuse flood light by specifying an intensity distribution for a point source. The intensity varies according to:

$$I = I_0 \cos^{\rho} \gamma$$

where I_0 is the specified intensity of the point source and γ is the angle between the illuminating direction and the direction to a surface point from the point source. For large values of ρ the light will be very concentrated around the illumination axis whereas small values of ρ will spread it out. If a solid's surface is "behind" the light, i.e. γ is greater than or equal to 90 degrees then the surface will not be illuminated. The default value s 0 where light uniformly radiates in all directions about the point source. If using this option one should remember to also specify the illuminating direction.

FROM <x,y,z>
TO <x,y,z>

These two statements allow specification of the light's illuminating direction by two points. The directed vector from the FROM point to the TO point defines the illuminating direction. Using this form is easier than specifying a direction vector explicitly. The default FROM position is < 0, 0, 100 > and the initial TO position is < 0, 0, 0 >. Each time the parser encounters either statement the direction is recalculated so the order in which they are specified will be important.

DIRECTION <x,y,z>

This allows specification of the light's illumination direction by a vector. The vector specified does not have to been in normalized form. The default value is <0, 0, -1 >, in which case the illuminating source is directly overhead at a zenith angle of 0 degrees.

ZENITH angle
AZIMUTH angle

This allows specification of the light's illuminating direction by its position on a sphere of infinite radius centered at the origin. In this form of specification the illuminating direction is from some point on the sphere towards the origin. The position is given by zenith and azimuth angles specified in degrees. The zenith angle is a rotation directed away from the positive Z axis towards the XY plane. The angle must be positive with a magnitude greater than or equal to 0 and less than 180. An angle of 0 degrees positions the light directly overhead while an angle close to 90 degrees places the light at the equator of the sphere. The azimuth angle is a rotation directed away from the positive X axis in the XY plane. A positive azimuth angle is a rotation towards the positive Y axis and a negative azimuth angle is a rotation towards the negative Y axis. The angle must be in the range {-180 .. 180 }. An angle of 0 degrees places the light aligned with the positive X axis and an angle of +/- 180 degrees places the light aligned with the negative X axis.

NOSHADOWS

This is a boolean flag which will disable shadow calculations for the light even if shadows have been enabled for the entire scene(by the SHADOWS statement in the SCENE statement block or by the -S command line switch). The default is to enable shadows for an individual light when they have been globally turned on.

NOSPECULAR

This is a boolean flag which will disable rendering calculations for specular highlights. This will only apply to any light which is assigned this attribute. Specifying nospecular may be appropriate for simulating a strong source of indirect lighting such as day light coming in through through a window. The default is to calculate specular highlights.

7.5.5.5 Objects

Object definitions are used to supply all the information required to represent a modelled entity in the environment. They may encompass all the previously described structures.

SOLID

Solid objects are always visible in a ray traced image and represent actual physical structures such as a house or a mountain range. Associated with each solid object are surface attributes which define the appearance of the object as it is rendered. Solids always have opaque surfaces, transparent objects are not supported. The shape of the solid is defined by a list of primitive structures.

Solid objects are initialized with default surface attributes, an identity transformation matrix and no primitives. The syntax for the statement block is:

```
SOLID
{
   SPHERE { ... }
   SPHERE sphere-identifier
   SPHERE sphere-identifier { ... }
      ...
   PLANE { ... }
   PLANE plane-identifier
   PLANE plane-identifier { ... }
      ...
   RECTANGLE { ... }
   RECTANGLE rectangle-identifier
   RECTANGLE rectangle-identifier { ... }
      ...
   TRIANGLE { ... }
   TRIANGLE triangle-identifier
   TRIANGLE triangle-identifier { ... }
      ...
   POLYGONS { ... }
   POLYGONS polygons-identifier
   POLYGONS polygons-identifier { ... }
      ...
   TERRAIN { ... }
      ...
   SURFACE { ... }
   SURFACE surface-identifier
   SURFACE surface-identifier { ... }
      ...
   TRANSFORM { ... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
      ...
}
```

One may specify any number of the above indicated primitives for a solid however the practical limit will be determined by the amount of available memory.

One may specify more than one set of surface attributes for the solid however only the most recent statements will have an effect.

One may specify any number of transformations to size, place and orient the solid into the scene's global coordinate system. The solid's transformation matrix is by default set to an identity matrix and thus has no effect on the defined object.

Solids may be explicitly added to the scene with a statement block as just described or they may be first defined as variables and then added later using either one of the following statements:

SOLID solid-identifier
SOLID solid-identifier { ... }

LIGHT

Light objects may or may not be visible in a ray traced image and are used to specify how a scene is to be illuminated by direct lighting. Associated with each light object are source attributes which define the illumination characteristics of the light such as color and intensity. A light may also be visible within the scene in which case it will also have a list of basic primitives describing its shape. A visible light is rendered as a special type of solid object with "luminous" surface characteristics. A visible light may be used to represent a light bulb or a fluorescent tube for example. It may not be used to represent a lighting fixture such as lamp with a base, stand etc. The light object's shape (as defined by any specified primitives) has no effect on the light's illumination characteristics; they are handled independently during rendering.

Transformations may also be specified for a light object. The transformation will be applied to any specified primitives as well as the position and direction of the illumination source. For example, if one modelled a spherical light bulb and placed a point source at the sphere's center one may use a transformation to position the bulb and point source within the scene at the same time. The illumination source will track its corresponding shape.

The PLANE and TERRAIN primitives may not be used within a light object. Light objects are initialized with default source attributes, an identity transformation matrix and no primitives. The statement block is as follows:

```
LIGHT
{
   SPHERE { ... }
   SPHERE sphere-identifier
   SPHERE sphere-identifier { ... }
        ...
   RECTANGLE { ... }
   RECTANGLE rectangle-identifier
   RECTANGLE rectangle-identifier { ... }
        ...
   TRIANGLE { ... }
   TRIANGLE triangle-identifier
   TRIANGLE triangle-identifier { ... }
        ...
   POLYGONS { ... }
   POLYGONS polygons-identifier
   POLYGONS polygons-identifier { ... }
        ...
   SOURCE { ... }
   SOURCE source-identifier
   SOURCE source-identifier { ... }
        ...
   TRANSFORM { ... }
   TRANSFORM transform-identifier
   TRANSFORM transform-identifier { ... }
        ...
}
```

One may specify any number of the above indicated primitives for a visible light however the practical limit will be determined by the amount of available memory.

One may specify more than one set of source attributes for the light however only the most recent statements will have an effect.

One may specify any number of transformations to size, place and orient the light into the scene's global coordinate system. The light's transformation matrix is by default set to an identity matrix and thus has no effect on the defined object.

Lights may be explicitly added to the scene with a statement block as just described or they may be first defined as variables and then added later using either one of the following statements:

LIGHT light-identifier
LIGHT light-identifier { ... }

8 Conclusion

This thesis has presented a computer program capable of producing visually realistic images of mirage phenomena. The use of illumination and shading has greatly enhanced the visual realism of the images. Object distortions are very distinct and can be clearly distinguished from one object to the next. The generated images compare well with the data and images provided in [2]. To further enhance the realism in generated images it is possible to modify the program to account for atmospheric refraction along a reflected ray's path. This would allow more accurate simulation of mirage phenomena in environments containing large bodies of water particularly along shorelines.

The animated sequence of an aircraft landing provides a practical example in the use of the program. The animation provides some insight into the effects of atmospheric refraction on an aircaft landing. The distortions perceived by a pilot may affect his judgement while approaching the runway. Depending on the atmospheric conditions present, the amount of distortion that is perceived by the pilot may result in a critical situation. By varying the temperature profiles that are input to the program one may possibly determine the exact conditions which could significantly influence a pilot's judgement in a given surrounding.

There are numerous enhancements that could be made to the existing software to increase its usefullness as a tool and to produce better images. The software would immediately benefit from some speed optimizations. Ray-object intersections could be improved and perhaps the implementation of a simpler illumination model may help reduce image generation times. The implementation of transparency, anti-aliasing and texture mapping would enhance the realism and quality of images. Accounting for atmospheric refraction while tracing reflected rays and adding support for lateral temperature profiles would be nice features for mirage simulation. Some improvements could be made in the way data are input to the program. A mouse driven interface with an integrated database modeller and perhaps support for animation would be very useful features to add. Providing the tools to graphically create and edit scenes would be a definite improvement over the currently used input method.

The program presented in this thesis serves as a useful software tool by providing considerable flexibility in the experimentation and testing of different atmospheric conditions and environmental surroundings. The program can be used to better one's understanding of atmospheric refraction or to just produce interesting images of mirage phenomena.

References

1. Lehn, W.H., A simple parabolic model for the optics of the atmospheric surface layer, Applied Mathematical Modelling, Vol. 9, 447-453, December, 1985.

2. Lehn, W.H., The Norse merman as an optical phenomenom, Nature, Vol.289, No. 5796, 362-366, January, 1981.

3. Tong, D., Influence of Atmospheric Refraction on the Aircraft Landing in Polar Regions, M.sc. thesis, University of Manitoba, Winnipeg, Manitoba, Canada, 1989.

4. Foley, J.D., Van Dam, A., Feiner, S., Hughes, J., Computer Graphics Principles and Practice, 2nd ed., Addison-Wesley Publishing Company Inc., 1990.

5. Glassner, S., An Introduction to Ray Tracing, Academic Press Inc., 1990.

6. Rogers, D.F., Procedural Elements for Computer Graphics, McGraw-Hill, Inc., 1985.

7. Rogers, D.F., Adams, J.A., Mathematical Elements for Computer Graphics, 2nd Ed., McGraw-Hill, Inc., 1990.

8. Perlin, K., An Image Synthesizer, SIGGRAPH 85, 287-296.

9. Cook, R.L., and Torrance, K.E., A reflection model for computer graphics. Comput. Graph. 15, 3 (Aug. 1981), 307-316.

10. Whitted, T., An improved illumination model for shaded display, Commun. ACM 23, 6 (June 1980), 343-349.

11. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T., Numerical Recipes in C, Cambridge University Press, 1988, Ch. 9.4.

12. Commodore-Amiga, Inc., AMIGA ROM Kernel Reference Manual: Libraries, 3rd ed. , Addison-Wesley Publishing Company.

13. Morrish, J. S., Inferior mirages and their corresponding temperature structures, M.sc. thesis, University of Manitoba, Winnipeg, Manitoba, Canada, 1985.

Appendix A

Reserved Keywords

AMBIENT	AMPLITUDE	ANGLE	ASPECT
ATTENUATION	AZIMUTH	BACKGROUND	BUMPS
CENTER	CHECKER	CLOUDS	COLOR
COLORS	DAMP	DENSITY	DEPTH
DIRECTION	DISTANCE	ELEVATION	EXPONENT
FACETS	FOV	FREQ	FROM
FUZZ	GRADIENT	GRANITE	HARD
HAZE	HEIGHT	HORIZONCOLOR	ID
IDENTITY	INFINITE	INTENSITY	LIGHT
LOCAL	LOGSCALE	LUMINANCE	MARBLE
METALLIC	MIRAGE	NOPHONG	NOSHADOWS
NOSPECULAR	NUMBER	OVERCAST	PHASE
PHONG	PLANE	POINT	POLYGONS
PRIMITIVES	PROFILE	RADIUS	RECTANGLE
REFINDEX	REGION	RIPPLES	ROTATE
ROTX	ROTY	ROTZ	ROUGH
SCALE	SCALEX	SCALEY	SCALEZ
SCENE	SEGMENTS	SFACETS	SHADOWS
SHADOWS	SKY	SLOPE	SNOW
SOLID	SOURCE	SPECULAR	SPHERE
SPOTTED	SQUEEZE	SUN	SURFACE
TERRAIN	TEXTURE	TILT	TO
TRANSLATE	TRANSFORM	TRANSX	TRANSY
TRANSZ	TRIANGLE	TURBULENCE	UBLEND
VARIABLES	VBLEND	VECTOR	VERTEX1
VERTEX2	VERTEX3	VERTICES	VIEW
WAVE	WAVES	WIDTH	WIREFRAME
WOOD	ZENITH	ZENITHCOLOR	

Appendix B

Program Errors

Syntax Errors

Expecting a '{'
Expecting a ','
Variable already defined
Invalid identifier
Undefined identifier
Invalid type specification
Identifier name too long
Incompatibles types
Can not assign a variable to itself
Expecting a '='
Expecting a '>'
Expecting a vector constant or variable
Expecting a numeric constant or variable
Unbalanced Comment
Expecting a digit

Invalid SURFACE statement
Invalid SOURCE statement
Invalid TEXTURE statement
Invalid TEXTURE Info statement
Invalid WAVE statement

Invalid TRANSFORM statement
MIRAGE data already defined
SKY data already defined

Invalid SCENE statement
Invalid AMBIENT statement
Invalid HAZE statement
Invalid VIEW statement
Invalid SKY statement
Invalid SUN statement
Invalid MIRAGE statement
No PROFILE specified

Scene definition already specified
Invalid statement
Invalid SOLID statement

Invalid LIGHT statement

Invalid SPHERE statement
Invalid PLANE statement
Invalid TRIANGLE statement
Invalid RECTANGLE statement

Invalid TERRAIN statement
Invalid POLYGONS statement

VERTICES already defined
FACETS already defined
No VERTICES defined
No FACETS defined
Vertex not defined

Data Errors

Overlapping terrain cells
Argument must be > 2
Invalid vertices
Number of VERTICES greater than specified
Number of VERTICES less than specified

Number of FACETS greater than specified
Number of FACETS less than specified
Invalid Normal
Normal on wrong side of facet

Argument must be in the range 1..1024
Argument must be in the range -180..180
Argument must be in the range 0..90
Sun must be above the horizon

Second index must be greater than first
Argument too small
Argument must be in the range 0.5 .. 175.0
Distance between FROM and TO is too small
Argument must be >= 0
Argument must be < 1024
REGION bottom must be >= top
REGION right must be >= left

Need at least 3 PROFILE points
PROFILE elevations must be increasing

Unrecoverable Errors

Empty data file
Observer position below all temperature layers
Observer position above all temperature layers
Adding Solid object with no primitives
Symbol table full (too many variables)
Transform stack underflow
Transform stack overflow

Missing file name for -r command
Missing file name for -g command
Missing file name for -b command
Invalid width specified for -w command
Missing argument for -w command
Invalid height specified for -h command
Missing argument for -h command
Invalid starting row specified for -s command
Missing argument for -s command
Invalid ending row specified for -e command
Missing argument for -e command
Invalid depth specified for -d command
Missing argument for -d command
Invalid log scale value specified for -l command
Missing argument for -l command

File Errors

No input file specified
Unable to open debug file
Unable to open buffer file
Reading image buffer file
Writing to image buffer file

Unable to open input file
Unable to open RED image file for output
Unable to open GREEN image file for output
Unable to open BLUE image file for output

Memory Allocation Errors

Polygon vertices
Polygon facets
Terrain vertices
Terrain facets
Terrain normals

Symbol entry
String buffer

Transform structure
Texture info structure
Texture structure
Wave structure
Colors structure

Attributes structure
Light info structure
Surface info structure

Object structure
Sphere info
Plane info
Rectangle info
Triangle info
Polygons info
Copied Polygon vertices
Copied Polygon facets
Assigned Polygon vertices
Assigned Polygon facets
Primitive structure

Internal Errors

NULL object in ParseSolidObject()
NULL object in ParseLightObject()

NULL primitive in ParseSphere()
NULL primitive in ParsePlane()
NULL primitive in ParseTriangle()
NULL primitive in ParseRectangle()

NULL primitive in ParseTerrain()
NULL primitive in ParsePolygons()

NULL attributes in ParseSurface()
NULL attributes in ParseSource()
NULL transform in ParseTransform()

NULL primitive in FreePrimitive()
NULL source primitive in CopyPrimitive()
NULL source primitive in AssignPrimitive()
NULL target primitive in AssignPrimitive()

NULL info in CopySphereInfo()
NULL source info in AssignSphereInfo()
NULL target info in AssignSphereInfo()
NULL info in FreeSphereInfo()

NULL info in CopyPlaneInfo()
NULL source info in AssignPlaneInfo()
NULL target info in AssignPlaneInfo()
NULL info in FreePlaneInfo()

NULL info in CopyRectangleInfo()
NULL source info in AssignRectangleInfo()
NULL target info in AssignRectangleInfo()
NULL info in FreeRectangleInfo()

NULL info in CopyTriangleInfo()
NULL source info in AssignTriangleInfo()
NULL target info in AssignTriangleInfo()
NULL info in FreeTriangleInfo()

NULL info in CopyPolygonsInfo()
Bad address for vertex in CopyPolygonsInfo()
NULL source info in AssignPolygonsInfo()
NULL target info in AssignPolygonsInfo()
Bad address for vertex in AssignPolygonsInfo()
NULL info in FreePolygonsInfo()
Bad address for polygon vertex

NULL light info in CopyLightInfo()
NULL source info in AssignLightInfo()

NULL target info in AssignLightInfo()
NULL light info in FreeLightInfo()

NULL surface info in CopySurfaceInfo()
NULL source info in AssignSurfaceInfo()
NULL target info in AssignSurfaceInfo()
NULL surface info in FreeSurfaceInfo()

NULL attributes in AddTexture()
NULL texture in AddTexture()

NULL attributes in InitAttributes()
NULL attributes in FreeAttributes()
NULL attributes in CopyAttributes()
NULL source attr. in AssignAttributes()
NULL target attr. in AssignAttributes()

NULL transform in CopyTransform()
NULL transform in FreeTransform()

NULL texture in FreeTexture()
NULL texture in CopyTexture()
NULL wave structure in FreeWave()
NULL wave structure in CopyWave()
NULL colors structure in FreeColors()
NULL colors structure in CopyColors()

NULL object in CopyObject()
NULL source object in AssignObject()
NULL target object in AssignObject()
NULL object in FreeObject()
NULL object in InitObject()
NULL object in AddObject()
NULL object in AddPrimitive()
NULL primitive in AddPrimitive()

Appendix C

Scene Description File for Merman Phenomenon Example 6.1

```
SCENE
{
  VIEW
  {
    FROM <1,0,2.1>
    TO <0,0,2.1001>
    FOV 1.5
    TILT 0.0
    WIDTH 320
    HEIGHT 200
    ASPECT 1.2
    REGION 0,0,199,319
  }

  LOGSCALE 15.0
  DEPTH  3

  AMBIENT
  {
    INTENSITY  0.2
    COLOR < 1.0, 1.0, 1.0 >
    DEPTH 0
    NSAMPLES 0
  }
  HAZE
  {
     INTENSITY 0.0
     DENSITY  0.0
     COLOR < 0.9, 0.9, 0.9 >
  }

  SKY
  {
    RADIUS   1.0e8
    INTENSITY 0.9
    ZENITHCOLOR < 0,0,0.9 >
    HORIZONCOLOR < 0.5,0.7,0.9 >
    OVERCAST 0.0

    SUN
    {
      INTENSITY 1.0
      AZIMUTH -30
      ZENITH  20
      COLOR <1,1,1>
    }
  }
```

```
    MIRAGE
    {
      DISTANCE 25000
      ELEVATION 100
      SEGMENTS  300

      PROFILE
      {
        /* merman */
        0.0   3.83
        0.1   3.85
        0.6   4.03
        0.84  4.14
        1.1   4.28
        1.32  4.42
        1.55  4.6
        1.75  4.81
        1.9   5.06
        2.0   5.27
        2.08  5.51
        2.15  5.8
        2.21  6.15
        2.25  6.5
        2.29  6.8
        2.35  7.17
        2.41  7.46
        2.5   7.75
        2.64  8.08
        2.8   8.35
        3.05  8.66
        3.35  8.95
        4.1   9.5
        5.1   10.0
        6.35  10.45
        7.6   10.75
        9.1   11.0
        10.35 11.15
        11.6  11.25
        12.9  11.32
        14.4  11.36
        16.0  11.35
        17.9  11.32
        19.7  11.3
      } /* profile */
    } /* mirage data */

} /* scene */
```

```
/* rainbow colored sphere lower right */
SOLID
{

 SPHERE
 {
   CENTER < -1400,7, 2 >
   RADIUS 2
  }

  SURFACE
  {
    COLOR <0.8, 0.0, 0.0>
    ROUGH 0.2
    HARD 0.21
    FUZZ  0
    AMBIENT 0.5
    METALLIC 0
    SPECULAR 0.3
    REFINDEX 1.4

    TEXTURE { UBLEND { COLOR <0,0.7,0.9> } }

  } /* surface */
} /* solid */

/* mirrored sphere upper right */
SOLID
{

 SPHERE
 {
   CENTER < -1405,4, 6 >
   RADIUS 2
  }

  SURFACE
  {
    COLOR <0.7, 0.7, 0.8>
    ROUGH 0.2
    HARD 1
    FUZZ  0
    AMBIENT 0.4
    METALLIC 1
    SPECULAR 1
    REFINDEX 1.4

  } /* surface */
} /* solid */
```

```
/* dark blue sphere lower left */
SOLID
{
  SPHERE
  {
    CENTER < -1400,-6, -1 >
    RADIUS 3
  }

  SURFACE
  {
    COLOR <0, 0, 0.9>
    ROUGH 0.2
    HARD 0.2
    FUZZ 0.0
    METALLIC 0.5
    AMBIENT 0.4
    SPECULAR 0.6
    REFINDEX 1.4

  } /* surface */
} /* solid */

/* magenta sphere upper right */
SOLID
{
  SPHERE
  {
    CENTER < -1400,-6, 4 >
    RADIUS 2
  }

  SURFACE
  {
    COLOR <0.8, 0.0, 0.7>
    ROUGH 0.3
    HARD 0.8
    FUZZ 0.0
    METALLIC 0
    AMBIENT 0.2
    SPECULAR 0.6
    REFINDEX 1.1

  } /* surface */
} /* solid */
```

```
/* checker ground */
SOLID
{
  PLANE
  {
    VERTEX1 < 0,0,0 >
    VERTEX2 < 0,1,0 >
    VERTEX3 < -1.0,0,0 >
  }

  SURFACE
  {
    COLOR <0.0, 0.0, 0.0>
    ROUGH 1.0
    HARD 0.0
    FUZZ 0.0
    METALLIC 0
    AMBIENT 0.1
    SPECULAR 0.0
    REFINDEX 1.1

    TEXTURE
    {
      CHECKER
      {
        COLOR < 1.0, 1.0, 1.0 >
        TRANSFORM
        {
          SCALEX 350
          SCALEY 6
          TRANSZ 0.1 /* eliminate random speckles */
        }
      } /* checker */
    } /* texture list */
  } /* surface */

} /* plane */

/* rainbow colored rectangle center */
SOLID
{
  RECTANGLE
  {
    VERTEX1 < -1400, -2, 1 >
    VERTEX2 < -1400, -2, 0 >
    VERTEX3 < -1400, 2,  0 >
  }
```

```
    SURFACE
    {
      COLOR <0.8, 0.8, 0.0>
      ROUGH 1.0
      HARD 0.0
      FUZZ 0.0
      AMBIENT 0.1
      METALLIC 0
      SPECULAR 0.0
      REFINDEX 1.1

      TEXTURE { VBLEND { COLOR < 0,0,0.8> } }
    } /* surface */

} /* solid */
```

Scene Description File for Runway Environment Example 6.2

```
VARIABLES
{
  SURFACE diffuse_surface;
  POLYGONS pyramid, cube;
}

SCENE
{
  VIEW
  {
    FROM < 3000, 0, 7 >

    TO < 0, 0, 0>
    FOV 14.0
    TILT 0.0
    WIDTH  320
    HEIGHT 200
    ASPECT 1.2
    REGION 0,0,199,319
  }

  BACKGROUND < 0.196, 0.6, 0.8 > /* sky blue */

  LOGSCALE 10.0
  DEPTH  2

  AMBIENT
  {
    INTENSITY  0.5
    COLOR < 1.0, 1.0, 1.0 >
    DEPTH 0
    NSAMPLES 0
  }

  MIRAGE
  {
    DISTANCE 25000
    ELEVATION 100
    SEGMENTS  300

/* merman profile shifted  up 4m */
PROFILE
{
  0.0   3.79
  3.5   3.8
  4.0   3.83
  4.1   3.85
```

```
    4.6   4.03
    4.84  4.14
    5.1   4.28
    5.32  4.42
    5.55  4.6
    5.75  4.81
    5.9   5.06
    6.0   5.27
    6.08  5.51
    6.15  5.8
    6.21  6.15
    6.25  6.5
    6.29  6.8
    6.35  7.17
    6.41  7.46
    6.5   7.75
    6.64  8.08
    6.8   8.35
    7.05  8.66
    7.35  8.95
    8.1   9.5
    9.1   10.0
    10.35  10.45
    11.6   10.75
    13.1   11.0
    14.35 11.15
    15.6  11.25
    16.9  11.32
    18.4  11.36
    20.0  11.35
    21.9  11.32
    23.7  11.3
  }

    } /* mirage DATA */

  } /* scene */

  /* Light object definition */
  LIGHT
  {
    SOURCE
    {
      INTENSITY 1
      COLOR <1,1,1>
      INFINITE
      ATTENUATION 0
      AZIMUTH 70
      ZENITH  50
    }
  }
```

```
/* set up variables */
cube =
{
  VERTICES
  {
    8
    < 1, -1, 1 >
    < 1,  1, 1 >
    < -1,  1, 1 >
    < -1, -1, 1 >
    < 1, -1, 0 >
    < 1,  1, 0 >
    < -1,  1, 0 >
    < -1, -1, 0 >
  }
  FACETS
  {
    10
    0 4 5
    0 5 1
    1 5 6
    1 6 2
    2 6 7
    2 7 3
    3 7 4
    3 4 0
    0 1 2
    0 2 3
  }
}

pyramid =
{
  VERTICES
  {
    5
    < 0, 0, 1 >
    < 1,-1, 0 >
    < 1, 1, 0 >
    < -1, 1, 0 >
    < -1,-1, 0 >
  }
  FACETS
  {
    4
    0 1 2
    0 2 3
    0 3 4
    0 4 1
  }
}
```

```
diffuse_surface =
{
  COLOR <1,1,1>
  ROUGH 1.0
  HARD 0.0
  FUZZ 0.0
  METALLIC 0
  AMBIENT 0.5
  SPECULAR 0.0
  REFINDEX 1.1
}

/* Solid object definitions */

SOLID  /* ground plane */
{
  PLANE
  {
    VERTEX1 < 0,0,0 >
    VERTEX2 < 1,0,0 >
    VERTEX3 < 1,1,0 >
   }

   SURFACE diffuse_surface
   { COLOR <0.419608, 0.556863, 0.137255> } /* medium forest green */
} /* ground plane */

SOLID  /* runway */
{
  RECTANGLE
  {
    VERTEX1 < -1200, -50, 0.1 >
    VERTEX2 < 1200, -50, 0.1 >
    VERTEX3 < 1200, 50, 0.1 >
   }
   SURFACE  diffuse_surface { COLOR < 0.752941, 0.752941, 0.752941 > }
} /* runway */

SOLID /* pyramid 1 */
{
  POLYGONS pyramid
  {
    TRANSFORM { SCALE < 10, 10, 20 > ROTZ 30}
  }
  SURFACE diffuse_surface
  {
     COLOR  < 0.858824, 0.576471, 0.439216>  /* tan */
  }
  TRANSFORM { TRANSLATE < 250, 100, 0 > }
} /* pyramid 1 */
```

```
SOLID /* pyramid 2 */
{
  POLYGONS pyramid
  {
    TRANSFORM { SCALE < 10, 10, 20 > ROTZ 20 }
  }
  SURFACE diffuse_surface
  {
     COLOR <0.560784, 0.560784, 0.737255>  /* light steel blue */
     COLOR  < 0.7, 0.7, 0.7 >
  }
  TRANSFORM { TRANSLATE < -625, 100, 0 > }
} /* pyramid 2 */

SOLID /* pyramid 3 */
{
  POLYGONS pyramid
  {
    TRANSFORM { SCALE < 20, 20, 40 > ROTZ -75}
  }
  SURFACE diffuse_surface
  {
     COLOR  < 0.858824, 0.576471, 0.439216>  /* tan */
  }
  TRANSFORM { TRANSLATE < -2500, 20, 0 > }
} /* pyramid 3 */

SOLID /* pyramid 4 */
{
  POLYGONS pyramid
  {
    TRANSFORM { ROTZ 45 SCALE < 15, 15, 30 > ROTZ -20 }
  }
  SURFACE diffuse_surface
  {
     COLOR  < 0.858824, 0.576471, 0.439216>  /* tan */
  }
  TRANSFORM { TRANSLATE < -1250, -100, 0 > }
} /* pyramid 4 */

SOLID /* pyramid 5 */
{
  POLYGONS pyramid {  TRANSFORM { SCALE < 15, 15, 25 > ROTZ -60 } }

  SURFACE diffuse_surface
  {
     COLOR  < 0.9, 0.9, 0.9 > /* grey */
  }
  TRANSFORM { TRANSLATE < -250, -150, 0 > }
} /* pyramid 5 */
```

Appendix D

Sample Debug File for Merman Example 6.1

Input File : mermandata
Date : Fri Sep 24 12:13:02 1993

Scene Definition

Max Specular Ray Depth ... : 2
Log Scale ..... : 15.000000
Image Scaling ..... : No
Shadows ..... : No
Wire Frame Display ..... : No

Maximum Extent
X ..... : -1397.000000
Y ..... : 9.000000
Z ..... : 8.000000

Minimum Extent
X ..... : -1407.000000
Y ..... : -9.000000
Z ..... : -4.000000

Background Color
R ..... : 0.000000
G ..... : 0.000000
B ..... : 0.000000

Haze Intensity ..... : 0.000000
Haze Density ..... : 0.000000

Haze Color
R ..... : 0.900000
G ..... : 0.900000
B ..... : 0.900000

Ambient Intensity ..... : 0.200000

Ambient Color
R ..... : 1.000000
G ..... : 1.000000
B ..... : 1.000000
Max Ambient Ray Depth .... : 0
Ambient Ray Samples .... : 0

View Plane

View Plane Normal
X ..... : -1.000000

Y ..... : 0.000000
Z ..... : 0.000100

View plane Right
X ..... : 0.000000
Y ..... : 1.000000
Z ..... : 0.000000

General UP direction
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

View Plane Up
X ..... : 0.000100
Y ..... : 0.000000
Z ..... : 1.000000

Looking From
X ..... : 1.000000
Y ..... : 0.000000
Z ..... : 2.100000

Looking Towards
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 2.100100

View Reference Point
X ..... : -75.390009
Y ..... : 0.000000
Z ..... : 2.107639

View Distance ..... : 76.390009
Field Of View ..... : 1.500000
Tilt ..... : 0.000000
Aspect Ratio ..... : 1.200000
Width ..... : 320
Height ..... : 200
Top ..... : 0
Left ..... : 0
Bottom ..... : 199
Right ..... : 319

Solid Object Definition # 1

Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Inverse Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Attributes
R ..... : 0.800000
G ..... : 0.800000
B ..... : 0.000000
Ambient Reflectance ..... : 0.100000
Diffuse Reflectance ..... : 1.000000
Specular Reflectance ..... : 0.000000
Refraction Index ..... : 1.100000
Roughness ..... : 1.000000
Hardness ..... : 0.000000
Metallic ..... : 0.000000
Fuzz ..... : 0.000000
Fresnel ..... : 0.002268
Beckman Normalization .... : 1.471510

Texture
Type ..... : vBlend
Local ..... : No
Scaling ..... : 1.000000
Turbulence/Deviation ..... : 1.000000
Squeeze/Elevation ..... : 1.000000
Maximum Slope ..... : 45.572996
R ..... : 0.000000
G ..... : 0.000000
B ..... : 0.800000

Number primitives ..... : 1

Rectangle
Reference P
X ..... : -1400.000000
Y ..... : -2.000000
Z ..... : 0.000000

U vector
X ..... : 0.000000
Y ..... : 4.000000
Z ..... : 0.000000

V vector
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Normal
X ..... : 4.000000
Y ..... : 0.000000
Z ..... : 0.000000

V1
X ..... : -1400.000000
Y ..... : -2.000000
Z ..... : 1.000000

V3
X ..... : -1400.000000
Y ..... : 2.000000
Z ..... : 0.000000

V4
X ..... : -1400.000000
Y ..... : 2.000000
Z ..... : 1.000000

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Solid Object Definition # 2

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Attributes
R ..... : 0.000000
G ..... : 0.000000
B ..... : 0.000000
Ambient Reflectance ..... : 0.100000

Diffuse Reflectance ..... : 1.000000
Specular Reflectance ..... : 0.000000
Refraction Index ..... : 1.100000
Roughness ..... : 1.000000
Hardness ..... : 0.000000
Metallic ..... : 0.000000
Fuzz ..... : 0.000000
Fresnel ..... : 0.002268
Beckman Normalization .... : 1.471510

Texture
Type ..... : Checker
Local ..... : No
Scaling ..... : 1.000000
Turbulence/Deviation ..... : 1.000000
Squeeze/Elevation ..... : 1.000000
Maximum Slope ..... : 45.572996
R ..... : 1.000000
G ..... : 1.000000
B ..... : 1.000000

Transform

350.000000	0.000000	0.000000	0.000000
0.000000	6.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.100000	1.000000

Inverse Transform

0.002857	0.000000	0.000000	0.000000
0.000000	0.166667	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	-0.100000	1.000000

Number primitives ..... : 1

Plane
Reference P
X ..... : 0.000000
Y ..... : 1.000000
Z ..... : 0.000000

U vector
X ..... : -1.000000
Y ..... : -1.000000
Z ..... : 0.000000

V vector
X ..... : 0.000000
Y ..... : -1.000000
Z ..... : 0.000000

Normal
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Solid Object Definition # 3

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Attributes
R ..... : 0.800000
G ..... : 0.000000
B ..... : 0.700000
Ambient Reflectance ..... : 0.200000
Diffuse Reflectance ..... : 0.400000
Specular Reflectance ..... : 0.600000
Refraction Index ..... : 1.100000
Roughness ..... : 0.300000
Hardness ..... : 0.800000
Metallic ..... : 0.000000
Fuzz ..... : 0.000000
Fresnel ..... : 0.002268
Beckman Normalization .... : 11.257787

Number primitives ..... : 1

Sphere
Radius ..... : 2.000000

Center
X ..... : -1400.000000
Y ..... : -6.000000
Z ..... : 4.000000

North Pole
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Equator
X ..... : 1.000000
Y ..... : 0.000000
Z ..... : 0.000000

Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Inverse Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Solid Object Definition # 4

Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Inverse Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Attributes
R ..... : 0.000000
G ..... : 0.000000
B ..... : 0.900000
Ambient Reflectance ..... : 0.400000
Diffuse Reflectance ..... : 0.400000
Specular Reflectance ..... : 0.600000
Refraction Index ..... : 1.400000
Roughness ..... : 0.200000
Hardness ..... : 0.200000

Metallic ..... : 0.500000
Fuzz ..... : 0.000000
Fresnel ..... : 0.027778
Beckman Normalization .... : 25.000000

Number primitives ..... : 1

Sphere
Radius ..... : 3.000000

Center
X ..... : -1400.000000
Y ..... : -6.000000
Z ..... : -1.000000

North Pole
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Equator
X ..... : 1.000000
Y ..... : 0.000000
Z ..... : 0.000000

Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Inverse Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Solid Object Definition # 5

Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Inverse Transform

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000

Attributes
R ..... : 0.700000
G ..... : 0.700000
B ..... : 0.800000
Ambient Reflectance ..... : 0.400000
Diffuse Reflectance ..... : 0.000000
Specular Reflectance ..... : 1.000000
Refraction Index ..... : 1.400000
Roughness ..... : 0.200000
Hardness ..... : 1.000000
Metallic ..... : 1.000000
Fuzz ..... : 0.000000
Fresnel ..... : 0.027778
Beckman Normalization .... : 25.000000

Number primitives ..... : 1

Sphere
Radius ..... : 2.000000

Center
X ..... : -1405.000000
Y ..... : 4.000000
Z ..... : 6.000000

North Pole
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Equator
X ..... : 1.000000
Y ..... : 0.000000
Z ..... : 0.000000

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Solid Object Definition # 6

Transform
1.000000 0.000000 0.000000 0.000000

0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Attributes
R ..... : 0.800000
G ..... : 0.000000
B ..... : 0.000000
Ambient Reflectance ..... : 0.500000
Diffuse Reflectance ..... : 0.700000
Specular Reflectance ..... : 0.300000
Refraction Index ..... : 1.400000
Roughness ..... : 0.200000
Hardness ..... : 0.210000
Metallic ..... : 0.000000
Fuzz ..... : 0.000000
Fresnel ..... : 0.027778
Beckman Normalization .... : 25.000000

Texture
Type ..... : uBlend
Local ..... : No
Scaling ..... : 1.000000
Turbulence/Deviation ..... : 1.000000
Squeeze/Elevation ..... : 1.000000
Maximum Slope ..... : 45.572996
R ..... : 0.000000
G ..... : 0.700000
B ..... : 0.900000

Number primitives ..... : 1

Sphere
Radius ..... : 2.000000

Center
X ..... : -1400.000000
Y ..... : 7.000000
Z ..... : 2.000000

North Pole
X ..... : 0.000000
Y ..... : 0.000000
Z ..... : 1.000000

Equator
X ..... : 1.000000
Y ..... : 0.000000
Z ..... : 0.000000

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Light Object Definition # 1

Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Inverse Transform
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

Attributes
Type ..... : Infinite
R ..... : 1.000000
G ..... : 1.000000
B ..... : 1.000000
Intensity ..... : 1.000000
Distance Attenuation ..... : 0.000000
Direction
X ..... : -0.270212
Y ..... : 0.156007
Z ..... : -0.857250
Shadows ..... : Yes
Specular Hilights ..... : Yes

Number primitives ..... : 0

Sky
Radius ..... : 1.000000e+08
Zenith Intensity ..... : 0.900000
Overcast ..... : 0.000000

Zenith Color
R ..... : 0.000000
G ..... : 0.000000
B ..... : 0.900000

Horizon Color
R ..... : 0.500000
G ..... : 0.700000
B ..... : 0.900000

Sun

Attributes
Type ..... : Infinite
R ..... : 1.000000
G ..... : 1.000000
B ..... : 1.000000
Intensity ..... : 1.000000
Distance Attenuation ..... : 0.000000
Direction
X ..... : -0.270212
Y ..... : 0.156007
Z ..... : -0.857250
Shadows ..... : Yes
Specular Hilights ..... : Yes

Mirage Definition
Max Horizontal Distance .. : 2.500000e+04
Max Elevation ..... : 1.970000e+01
Max Ray Segments ..... : 300

Temperature Profile

	Elevation	Temperature
0	0.000000	3.830000
1	0.100000	3.850000
2	0.600000	4.030000
3	0.840000	4.140000
4	1.100000	4.280000
5	1.320000	4.420000
6	1.550000	4.600000
7	1.750000	4.810000
8	1.900000	5.060000
9	2.000000	5.270000
10	2.080000	5.510000
11	2.150000	5.800000
12	2.210000	6.150000
13	2.250000	6.500000
14	2.290000	6.800000
15	2.350000	7.170000

16	2.410000	7.460000
17	2.500000	7.750000
18	2.640000	8.080000
19	2.800000	8.350000
20	3.050000	8.660000
21	3.350000	8.950000
22	4.100000	9.500000
23	5.100000	10.000000
24	6.350000	10.450000
25	7.600000	10.750000
26	9.100000	11.000000
27	10.350000	11.150000
28	11.600000	11.250000
29	12.900000	11.320000
30	14.400000	11.360000
31	16.000000	11.350000
32	17.900000	11.320000
33	19.700000	11.300000

Appendix E

```
/*

   Sample C program to generate POLYGONS data

   Danny Robinson
   July, 1993

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define ULONG unsigned long
#define DBL double

typedef DBL Vector[3];

#define SetVector( v,x,y,z ) {(v)[0] = (x); (v)[1] = (y); (v)[2] = (z);}

void reverse( char * );
void etoa( DBL, char *, ULONG );

void writePolygons( FILE *, ULONG , ULONG );
void writeSmoothPolygons( FILE *, ULONG, ULONG );
void writeNormal( FILE *,ULONG i, char *buffer );
void makeCylinder( ULONG, ULONG, ULONG *, ULONG *);
void makeCone( ULONG, ULONG, ULONG *, ULONG *);

Vector vertices[101];
ULONG  facets[100][3];

Vector curve[20];
Vector curveNormals[20];
Vector normals[100];

void main( void )
{
      ULONG nvertices, nfacets, nslices, ncurvepts;
      FILE *fp;

      /* make a cylinder, radius 1, axis along Z, height 1 */

      if( (fp = fopen( "cylinder", "w")) == NULL )
      {
        fprintf(stderr,"Unable to open file\n");
        exit(1);
      }

      nslices = 16;
```

```
        ncurvepts = 2;
        SetVector( curve[0], 1.0, 0.0, 1.0 );
        SetVector( curve[1], 1.0, 0.0, 0.0 );

        SetVector( curveNormals[0], 1.0, 0.0, 0.0 );
        SetVector( curveNormals[1], 1.0, 0.0, 0.0 );

        makeCylinder( nslices, ncurvepts, &nvertices, &nfacets );

        printf("Writing cylinder ...\n");
        writeSmoothPolygons( fp, nvertices, nfacets );

        fclose( fp );

        /* make a cone , apex <0,0,1>, axis along Z, base radius 1 */

        if( (fp = fopen( "cone", "w")) == NULL )
        {
          fprintf(stderr,"Unable to open file\n");
          exit(1);
        }


        nslices = 16;
        ncurvepts = 2;
        SetVector( curve[0], 0.0, 0.0, 1.0 );
        SetVector( curve[1], 1.0, 0.0, 0.0 );

        makeCone( nslices, ncurvepts, &nvertices, &nfacets );

        printf("Writing cone ...\n");
        writePolygons( fp, nvertices, nfacets );

        fclose( fp );

        printf("All done ...\n");
}

void reverse( char *str )
{
        char *s;
        char c;

        s = str + strlen(str) - 1;
        while( str <  s )
        {
                c = *str;
                *str++ = *s;
                *s-- = c;
        }
}
```

```
void etoa( DBL n, char *str, ULONG ndecimalpts )
{
      DBL fraction;
      ULONG i, whole;
      char *s, sign = ' ';

      if( n < 0 )
      {
         n = -n;
         sign = '-';
      }

      whole = (ULONG)n;
      fraction = n - (DBL)whole;

      s = str;
      do
      {
         *s++ = whole % 10 + '0';
      } while(  (whole = whole / 10) > 0 );

      *s++ = sign;
      *s = '\0';

      reverse( str );

      if( ndecimalpts > 0 )
      {
        *s++ = '.';
        i = 0;
        do
        {
          fraction *= 10.0;
          whole = (ULONG) fraction;
          *s++ = whole % 10 + '0';
          i++;
        } while( i < ndecimalpts );
        *s = '\0';
      }
}

/* write out VERTICES { ... } & FACETS { ... } */

void writePolygons( FILE *fp, ULONG nvertices, ULONG nfacets )
{
      ULONG i;
      char  buffer[80];

      /* start of block */
      fprintf( fp, "VERTICES {\n%d\n",nvertices);
```

```
        for( i = 0; i < nvertices; i++ )
        {
           etoa( vertices[i][0], buffer, 4 );
           fprintf(fp,"< %s, ",buffer);
           etoa( vertices[i][1], buffer, 4 );
           fprintf(fp,"%s, ",buffer);
           etoa( vertices[i][2], buffer, 4 );
           fprintf(fp,"%s >\n",buffer);
        }

        /* end of vertices block & start of facets block */
        fprintf( fp, "}\n");

        fprintf( fp, "\nFACETS {\n%d\n",nfacets);

        for( i = 0; i < nfacets; i++)
        {
           fprintf(fp,"%d  %d  %d\n",
                 facets[i][0],facets[i][1],facets[i][2]);

        }

        /* last facet */
        fprintf(fp,"}\n");
}

/* write out VERTICES { ... } & SFACETS { ... } for Phong shading */

void writeSmoothPolygons( FILE *fp, ULONG nvertices, ULONG nfacets )
{
      ULONG i;
      char  buffer[80];

      /* start of block */
      fprintf( fp, "VERTICES {\n%d\n",nvertices);

      for( i = 0; i < nvertices; i++ )
      {
         etoa( vertices[i][0], buffer, 4 );
         fprintf(fp,"< %s, ",buffer);
         etoa( vertices[i][1], buffer, 4 );
         fprintf(fp,"%s, ",buffer);
         etoa( vertices[i][2], buffer, 4 );
         fprintf(fp,"%s >\n",buffer);
      }

      /* end of vertices block & start of facets block */
      fprintf( fp, "}\n");

      fprintf( fp, "\nSFACETS {\n%d\n",nfacets);
```

```
        for( i = 0; i < nfacets; i++)
        {
           fprintf(fp,"%d  %d  %d\n",
                  facets[i][0],facets[i][1],facets[i][2]);

           /* normal at vertex 1 */
           writeNormal( fp, facets[i][0], buffer );

           /* normal at vertex 2 */
           writeNormal( fp, facets[i][1], buffer );

           /* normal at vertex 3 */
           writeNormal( fp, facets[i][2], buffer );

           fprintf(fp,"\n");
        }

        /* last facet */
        fprintf(fp,"}\n");
}

void writeNormal( FILE *fp, ULONG i, char *buffer )
{
   etoa( normals[i][0], buffer, 4 );
   fprintf(fp,"< %s, ",buffer);
   etoa( normals[i][1], buffer, 4 );
   fprintf(fp,"%s, ",buffer);
   etoa( normals[i][2], buffer, 4 );
   fprintf(fp,"%s >\n",buffer);
}

/* surface of revolution both ends open */

void makeCylinder( ULONG nslices, ULONG ncurvepts,
                   ULONG *nvertices, ULONG *nfacets )
{
      DBL da, angle, r, x, y;
      ULONG i,j,k,v;

      *nvertices = ncurvepts * nslices;
      *nfacets = 2 * nslices * ( ncurvepts-1);

      printf("# vertices : %d\n",*nvertices);
      printf("# facets   : %d\n",*nfacets);

      da = 2.0 * PI / (double)nslices;

      angle = 0.0;

      k = 0;
      for( i = 0; i < nslices ; i++, angle += da )
```

```
        {
          for( j = 0; j < ncurvepts; j++ )
          {
            r = curve[j][0];
            x = r * cos( angle );
            y = r * sin( angle );
            SetVector( vertices[k], x, y, curve[j][2] );

            r = curveNormals[j][0];
            x = r * cos( angle );
            y = r * sin( angle );
            SetVector( normals[k], x, y, curveNormals[j][2] );

            k++;
          }
        }

        /* open ends */
        v = 0;
        k = 0;
        for( i = 0; i < nslices-1 ; i++ )
        {
          for( j = 0; j < ncurvepts-1; j++ )
          {
            SetVector( facets[k], v, v + 1, v + ncurvepts + 1 );
            k++;
            SetVector( facets[k], v, v+ncurvepts+1, v + ncurvepts );
            k++;
            v++;
          }
          v++;
        }

        for( j = 0; j < ncurvepts-1; j++ )
        {
          SetVector( facets[k], v, v + 1, j+1 );
          k++;
          SetVector( facets[k], v, j+1, j );
          k++;
          v++;
        }

}


/* surface of revolution, bottom open, top joined by single vertex */

void makeCone( ULONG nslices, ULONG ncurvepts,
               ULONG *nvertices, ULONG *nfacets )
{
    DBL da, angle, r, x, y;
```

```
  ULONG i,j,k, v;

  *nvertices = (ncurvepts-1) * nslices + 1;
  *nfacets = 2 * nslices * ( ncurvepts-2) + nslices;

  printf("# vertices : %d\n",*nvertices);
  printf("# facets   : %d\n",*nfacets);

  da = 2.0 * PI / (double)nslices;

  angle = 0.0;

  /* top */
  SetVector( vertices[0], 0.0, 0.0, curve[0][2] );

  k = 1;
  for( i = 0; i < nslices ; i++, angle += da )
  {
     for( j = 1; j < ncurvepts; j++ )
     {
        r = curve[j][0];
        x = r * cos( angle );
        y = r * sin( angle );
        SetVector( vertices[k], x, y, curve[j][2] );
        k++;
     }
  }

  v = 1;
  k = 0;
  for( i = 0; i < nslices-1 ; i++ )
  {
     SetVector( facets[k], 0, v, v + ncurvepts-1 );
     k++;

     for( j = 1; j < ncurvepts-1; j++ )
     {
        SetVector( facets[k], v, v + 1, v + ncurvepts );
        k++;
        SetVector( facets[k], v, v+ncurvepts, v + ncurvepts-1 );
        k++;
        v++;
     }
     v++;
  }

  /* last slice */
  SetVector( facets[k], 0, v, 1 );
  k++;

  for( j = 1; j < ncurvepts-1; j++ )
```

```
        {
            SetVector( facets[k], v, v + 1, j+1 );
            k++;
            SetVector( facets[k], v, j+1, j );
            k++;
            v++;
        }
    }
```

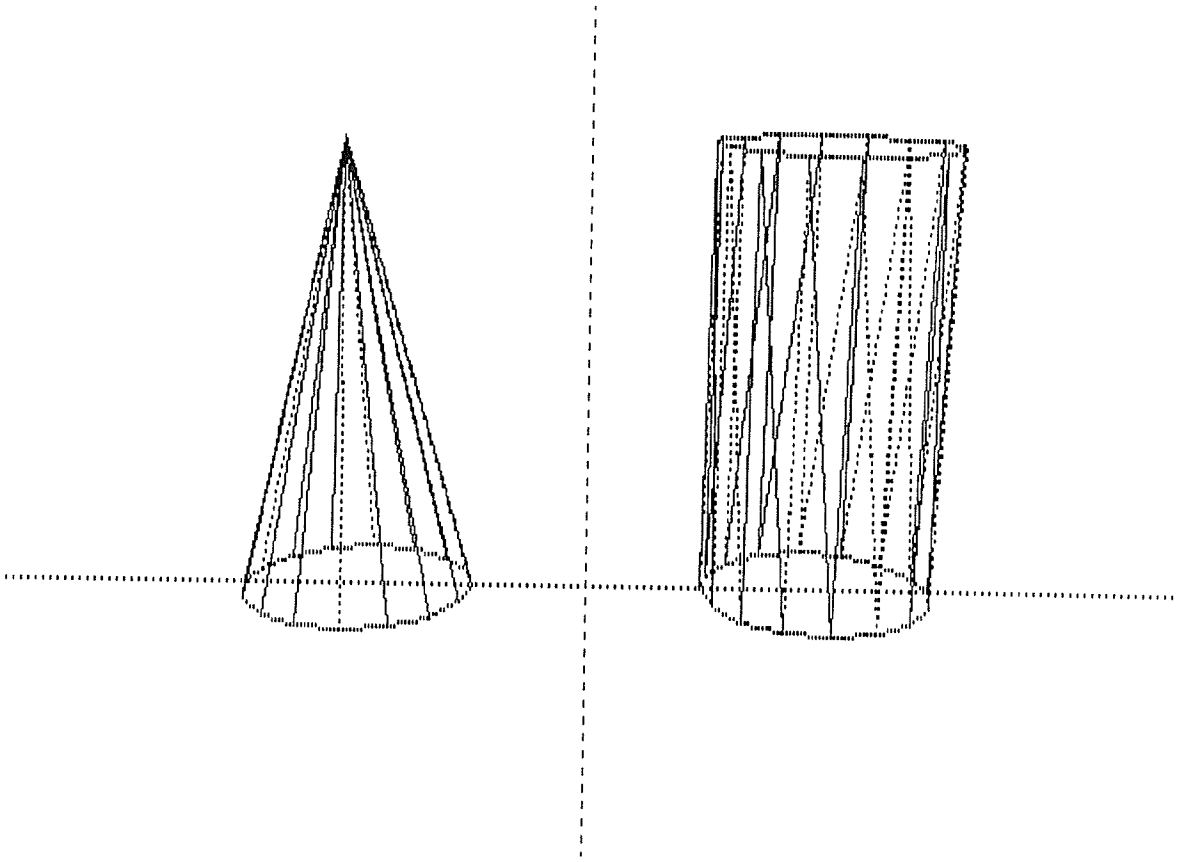

Figure E.1 Generated cone and cylinder polygons

Generated Cone Data

```
VERTICES {
17
< 0.0000, 0.0000, 1.0000 >
< 0.9999, 0.0000, 0.0000 >
< 0.9238, 0.3826, 0.0000 >
< 0.7071, 0.7071, 0.0000 >
< 0.3826, 0.9238, 0.0000 >
< 0.0000, 1.0000, 0.0000 >
< -0.3826, 0.9238, 0.0000 >
< -0.7071, 0.7071, 0.0000 >
< -0.9238, 0.3826, 0.0000 >
< -1.0000, 0.0000, 0.0000 >
< -0.9238, -0.3826, 0.0000 >
< -0.7071, -0.7071, 0.0000 >
< -0.3826, -0.9238, 0.0000 >
< 0.0000, -1.0000, 0.0000 >
< 0.3826, -0.9238, 0.0000 >
< 0.7071, -0.7071, 0.0000 >
< 0.9238, -0.3826, 0.0000 >
}

FACETS {
16
0 1 2
0 2 3
0 3 4
0 4 5
0 5 6
0 6 7
0 7 8
0 8 9
0 9 10
0 10 11
0 11 12
0 12 13
0 13 14
0 14 15
0 15 16
0 16 1
}
```

Generated Cylinder Data

```
VERTICES {
32
< 0.9999, 0.0000, 1.0000 >
< 0.9999, 0.0000, 0.0000 >
< 0.9238, 0.3826, 1.0000 >
< 0.9238, 0.3826, 0.0000 >
< 0.7071, 0.7071, 1.0000 >
< 0.7071, 0.7071, 0.0000 >
< 0.3826, 0.9238, 1.0000 >
< 0.3826, 0.9238, 0.0000 >
< 0.0000, 1.0000, 1.0000 >
< 0.0000, 1.0000, 0.0000 >
< -0.3826, 0.9238, 1.0000 >
< -0.3826, 0.9238, 0.0000 >
< -0.7071, 0.7071, 1.0000 >
< -0.7071, 0.7071, 0.0000 >
< -0.9238, 0.3826, 1.0000 >
< -0.9238, 0.3826, 0.0000 >
< -1.0000, 0.0000, 1.0000 >
< -1.0000, 0.0000, 0.0000 >
< -0.9238, -0.3826, 1.0000 >
< -0.9238, -0.3826, 0.0000 >
< -0.7071, -0.7071, 1.0000 >
< -0.7071, -0.7071, 0.0000 >
< -0.3826, -0.9238, 1.0000 >
< -0.3826, -0.9238, 0.0000 >
< 0.0000, -1.0000, 1.0000 >
< 0.0000, -1.0000, 0.0000 >
< 0.3826, -0.9238, 1.0000 >
< 0.3826, -0.9238, 0.0000 >
< 0.7071, -0.7071, 1.0000 >
< 0.7071, -0.7071, 0.0000 >
< 0.9238, -0.3826, 1.0000 >
< 0.9238, -0.3826, 0.0000 >
}

SFACETS {
32
0 1 3
< 0.9999, 0.0000, 0.0000 >
< 0.9999, 0.0000, 0.0000 >
< 0.9238, 0.3826, 0.0000 >

0 3 2
< 0.9999, 0.0000, 0.0000 >
< 0.9238, 0.3826, 0.0000 >
< 0.9238, 0.3826, 0.0000 >
```

2 3 5
< 0.9238, 0.3826, 0.0000 >
< 0.9238, 0.3826, 0.0000 >
< 0.7071, 0.7071, 0.0000 >

2 5 4
< 0.9238, 0.3826, 0.0000 >
< 0.7071, 0.7071, 0.0000 >
< 0.7071, 0.7071, 0.0000 >

4 5 7
< 0.7071, 0.7071, 0.0000 >
< 0.7071, 0.7071, 0.0000 >
< 0.3826, 0.9238, 0.0000 >

4 7 6
< 0.7071, 0.7071, 0.0000 >
< 0.3826, 0.9238, 0.0000 >
< 0.3826, 0.9238, 0.0000 >

6 7 9
< 0.3826, 0.9238, 0.0000 >
< 0.3826, 0.9238, 0.0000 >
< 0.0000, 1.0000, 0.0000 >

6 9 8
< 0.3826, 0.9238, 0.0000 >
< 0.0000, 1.0000, 0.0000 >
< 0.0000, 1.0000, 0.0000 >

8 9 11
< 0.0000, 1.0000, 0.0000 >
< 0.0000, 1.0000, 0.0000 >
< -0.3826, 0.9238, 0.0000 >

8 11 10
< 0.0000, 1.0000, 0.0000 >
< -0.3826, 0.9238, 0.0000 >
< -0.3826, 0.9238, 0.0000 >

10 11 13
< -0.3826, 0.9238, 0.0000 >
< -0.3826, 0.9238, 0.0000 >
< -0.7071, 0.7071, 0.0000 >

10 13 12
< -0.3826, 0.9238, 0.0000 >
< -0.7071, 0.7071, 0.0000 >
< -0.7071, 0.7071, 0.0000 >

12 13 15

< -0.7071, 0.7071, 0.0000 >
< -0.7071, 0.7071, 0.0000 >
< -0.9238, 0.3826, 0.0000 >

12 15 14
< -0.7071, 0.7071, 0.0000 >
< -0.9238, 0.3826, 0.0000 >
< -0.9238, 0.3826, 0.0000 >

14 15 17
< -0.9238, 0.3826, 0.0000 >
< -0.9238, 0.3826, 0.0000 >
< -1.0000, 0.0000, 0:0000 >

14 17 16
< -0.9238, 0.3826, 0.0000 >
< -1.0000, 0.0000, 0.0000 >
< -1.0000, 0.0000, 0.0000 >

16 17 19
< -1.0000, 0.0000, 0.0000 >
< -1.0000, 0.0000, 0.0000 >
< -0.9238, -0.3826, 0.0000 >

16 19 18
< -1.0000, 0.0000, 0.0000 >
< -0.9238, -0.3826, 0.0000 >
< -0.9238, -0.3826, 0.0000 >

18 19 21
< -0.9238, -0.3826, 0.0000 >
< -0.9238, -0.3826, 0.0000 >
< -0.7071, -0.7071, 0.0000 >

18 21 20
< -0.9238, -0.3826, 0.0000 >
< -0.7071, -0.7071, 0.0000 >
< -0.7071, -0.7071, 0.0000 >

20 21 23
< -0.7071, -0.7071, 0.0000 >
< -0.7071, -0.7071, 0.0000 >
< -0.3826, -0.9238, 0.0000 >

20 23 22
< -0.7071, -0.7071, 0.0000 >
< -0.3826, -0.9238, 0.0000 >
< -0.3826, -0.9238, 0.0000 >

22 23 25
< -0.3826, -0.9238, 0.0000 >

```
 < -0.3826, -0.9238,  0.0000 >
 <  0.0000, -1.0000,  0.0000 >

 22  25  24
 < -0.3826, -0.9238,  0.0000 >
 <  0.0000, -1.0000,  0.0000 >
 <  0.0000, -1.0000,  0.0000 >

 24  25  27
 <  0.0000, -1.0000,  0.0000 >
 <  0.0000, -1.0000,  0.0000 >
 <  0.3826, -0.9238,  0.0000 >

 24  27  26
 <  0.0000, -1.0000,  0.0000 >
 <  0.3826, -0.9238,  0.0000 >
 <  0.3826, -0.9238,  0.0000 >

 26  27  29
 <  0.3826, -0.9238,  0.0000 >
 <  0.3826, -0.9238,  0.0000 >
 <  0.7071, -0.7071,  0.0000 >

 26  29  28
 <  0.3826, -0.9238,  0.0000 >
 <  0.7071, -0.7071,  0.0000 >
 <  0.7071, -0.7071,  0.0000 >

 28  29  31
 <  0.7071, -0.7071,  0.0000 >
 <  0.7071, -0.7071,  0.0000 >
 <  0.9238, -0.3826,  0.0000 >

 28  31  30
 <  0.7071, -0.7071,  0.0000 >
 <  0.9238, -0.3826,  0.0000 >
 <  0.9238, -0.3826,  0.0000 >

 30  31  1
 <  0.9238, -0.3826,  0.0000 >
 <  0.9238, -0.3826,  0.0000 >
 <  0.9999,  0.0000,  0.0000 >

 30  1  0
 <  0.9238, -0.3826,  0.0000 >
 <  0.9999,  0.0000,  0.0000 >
 <  0.9999,  0.0000,  0.0000 >

}
```

```
/*

   Sample C program to generate TERRAIN data

   Danny Robinson
   July, 1993

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>


#define ULONG unsigned long
#define DBL double


typedef DBL Vector[3];

#define SetVector( v,x,y,z ) {(v)[0] = (x); (v)[1] = (y); (v)[2] = (z);}

void reverse( char * );
void etoa( DBL, char *, ULONG );

void writeTerrain( FILE *, ULONG , ULONG );
void makeTerrain( ULONG, ULONG );
DBL function( DBL, DBL );

Vector vertices[60][60];


void main( void )
{
      ULONG nrows, ncols;
      FILE *fp;


      if( (fp = fopen( "terrain", "w")) == NULL )
      {
        fprintf(stderr,"Unable to open file for output\n");
        exit(1);
      }

      nrows = 6;
      ncols = 6;
      makeTerrain( nrows, ncols );

      printf("Writing ...\n");
      writeTerrain( fp, nrows, ncols );
```

```
        fclose( fp );
        printf("All done ...\n");
}

void reverse( char *str )
{
        char *s;
        char c;

        s = str + strlen(str) - 1;
        while( str <  s )
        {
                c = *str;
                *str++ = *s;
                *s-- = c;
        }
}


void etoa( DBL n, char *str, ULONG ndecimalpts )
{
        DBL fraction;
        ULONG i, whole;
        char *s, sign = ' ';

        if( n < 0 )
        {
           n = -n;
           sign = '-';
        }

        whole = (ULONG)n;
        fraction = n - (DBL)whole;

        s = str;
        do
        {
           *s++ = whole % 10 + '0';
        } while(  (whole = whole / 10) > 0 );

        *s++ = sign;
        *s = '\0';

        reverse( str );

        if( ndecimalpts > 0 )
        {
          *s++ = '.';
          i = 0;
          do
```

```
        {
          fraction *= 10.0;
          whole = (ULONG) fraction;
          *s++ = whole % 10 + '0';
          i++;
        } while( i < ndecimalpts );
        *s = '\0';
      }
}

/* write out TERRAIN { VERTICES { ... } } */

void writeTerrain( FILE *fp, ULONG nrows, ULONG ncols )
{
      ULONG i,j;
      char  buffer[80];

      fprintf( fp, "TERRAIN {\n");
      fprintf( fp, "VERTICES {\n%d %d\n",nrows, ncols);

      for( i = 0; i < nrows; i++ )
      {
         fprintf( fp, "/* row %d */\n",i+1);
         for( j = 0; j < ncols; j++ )
         {
           etoa( vertices[i][j][0], buffer, 4 );
           fprintf(fp,"< %s, ",buffer);
           etoa( vertices[i][j][1], buffer, 4 );
           fprintf(fp,"%s, ",buffer);
           etoa( vertices[i][j][2], buffer, 4 );
           fprintf(fp,"%s >\n",buffer);
         }
      }

      /* end of VERTICES block & end of TERRAIN block */
      fprintf( fp, "}\n}\n");

}

void makeTerrain( ULONG nrows, ULONG ncols )
{
      DBL x, y, z, dx, dy;
      DBL xmax, xmin, ymax, ymin;
      ULONG i,j;

      xmin = -10.0;
      xmax = 10.0;
      ymin = -10.0;
      ymax = 10.0;

      dx = (xmax - xmin) / (DBL) nrows;
```

```
        dy = (ymax - ymin) / (DBL) ncols;

        for( x = xmin, i = 0; i < nrows ; i++, x += dx )
        {
           for( y = ymin, j = 0; j < ncols; j++, y += dy )
           {
              z = function( x,y );
              SetVector( vertices[i][j], x, y, z );
           }
        }
}

DBL function( DBL x, DBL y )
{
        DBL t;

        t = sqrt( x*x + y*y );
        if( fabs(t) < 1.0e-6 ) t =  4.0;
        else t  = 4.0 * sin( t ) / t;

        return( t );
}
```

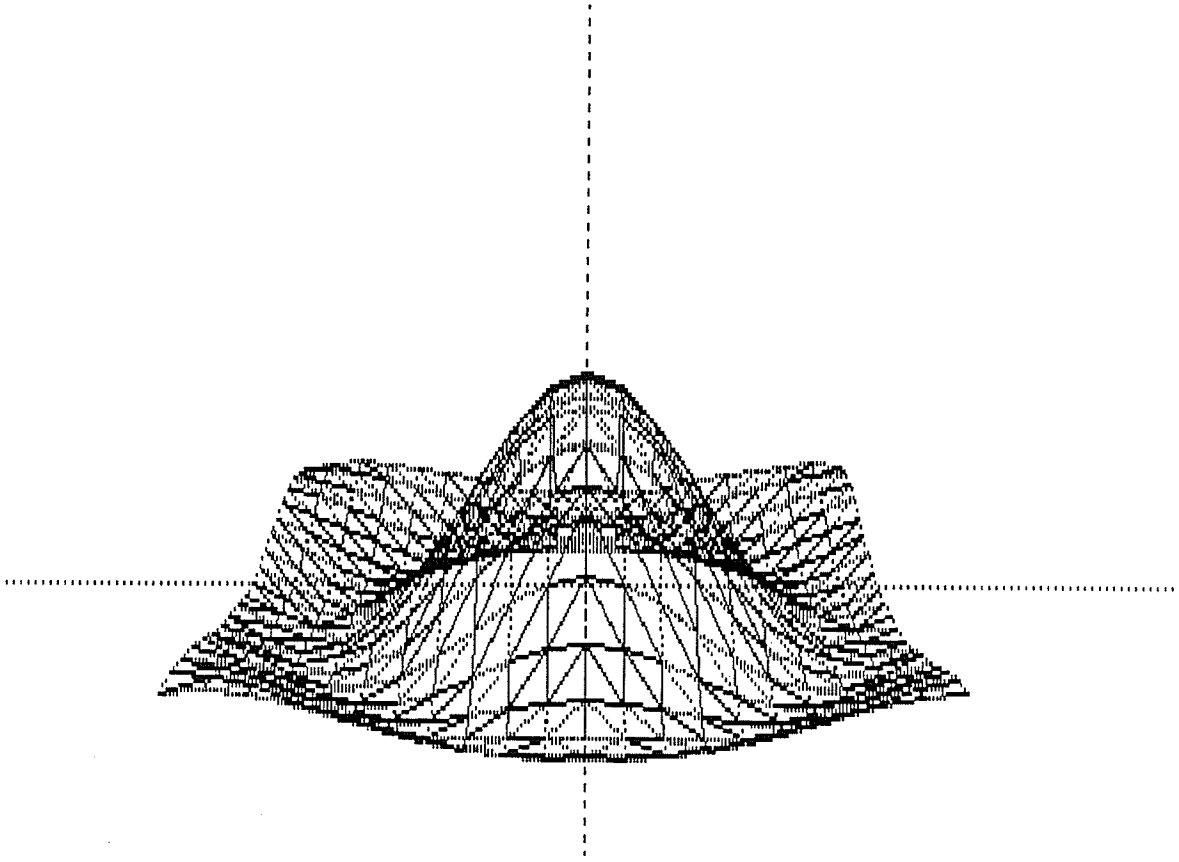

Figure E.2 Generated terrain polygons