The University of Teesside
MSc 3D Computer Aided Graphical Technology Applications
September 1992

An Atmospheric Illumination Model for Computer Graphics

John Irwin BSc(Hons), DASS

Project Supervisors
Dr John Dormand, Senior Lecturer
Dr Gordon Selley, Real World Graphics Ltd

School of Computing and Mathematics
Middlesbrough, Cleveland, TS1 3BA

Table of Contents

Abstract

A technique for rendering outdoor scenes in the presence of an atmosphere is presented. The light scattering properties of the atmospheric medium are used to simulate certain conditions seen in the real atmosphere, namely the blue of the sky and the appearance of fog of various densities and altitudes. The geometry of a plane-parallel atmosphere is considered with restrictions to take into account the size and shape of the terrestrial atmosphere. A single external source of parallel white light is used to represent the illumination provided by the sun. The model is based on the single scattering of light by particles constituting the atmosphere and results have shown that this approximation cannot be realistically used across the full range of conditions found in nature. It is proposed that models which incorporate multiple scattering need to be developed and used in such cases.

Acknowledgements

I would like to thank my supervisors for their support and encouragement throughout the project. Thanks also go to Gary Quinn and Hamish Lawson for invaluable help on various matters in the computer laboratories. By no means least, it's worth mentioning the contribution made to my general well being during some very difficult times in the labs from the Victoria Road Posse!, namely, Dylan, Ian, Edward, Deborah, Tim, Mike, Peter and others.

1 Introduction

A lot of work has been published which has striven to produce realistic computer graphic images by considering the detailed physics of how light interacts with material surfaces; see for example [1]. The results are certainly very convincing. However, to be strictly true to life the space between objects must also be considered. Traditionally, realistic image rendering using the methods of ray-tracing[2] and radiosity[3] has been performed in a *vacuum*, where it is assumed either consciously or not that the effect of the space between objects on the quality and/or quantity of the light is insignificant or non-existent. In reality, though, there is a *participating medium*, an atmosphere, which can contribute significantly to the illumination in a scene. The amount that it does add depends on the physical conditions prevailing in the atmosphere at a given moment. When there are large amounts of dust or water vapour in the atmosphere, scattering, reflection and absorption of light takes place resulting in more or less light reaching the eye along a given line of sight. Even the effects of air molecules are significant when the distances of travel are large enough which, because the scattering is wavelength dependent, gives rise to the blue colour of the clear sky. The colour dependency is important to consider since it influences the perceived colour of objects in subtle ways.

It is the purpose of this project to investigate and implement an atmospheric illumination model which can simulate the effects of an intervening medium on the rendering of a scene. The aim is to be as realistic as possible, of course, and base the model on the physical principles governing the scattering and absorption events.

There have been a number of papers published over the last decade which have gone some way in exploring the effects of atmospheric scattering. Early attempts to simulate the appearance of fog were biased towards just creating the *effect* of fog and were not based upon rigorous methods. They simply attenuated an image by linearly interpolating between the object colour and the fog colour as a function of distance. See for example[4]. Because it is assumed that the fog colour is white (quite reasonable), the effect of the fog is to *desaturate* the colours in the image. The results were adequate for their needs but very limited.

The first attempt to model the scattering process was undertaken by Blinn[5]. His work considers the single scattering of light in a medium composed of small spheres. The medium is restricted to thin planar geometries and the motivation for the work seems to have been to simulate the correct appearance of Saturn's rings. The results were not found to be appropriate for use in this project.

The paper by Willis[6] describes a simple atmospheric model for use in flight simulation systems and addresses more directly the problem of visually re-creating the appearance of fog using physical principles. He uses a result from radiative transfer theory called *Beer's Law* which describes the exponential decay of the intensity of light travelling through a scattering medium. The results are applied to an atmosphere which is spatially uniform in density and effectively extends to infinity in all directions.

The model due to Willis is essentially that used by Real World Graphics (RWG). As a preliminary exercise, we will implement this model and incorporate a few additions which haven't yet been tried. This includes using empirically measured extinction coefficients [1] and their wavelength dependence, using direct lighting calculations to determine the colour reflected from object surfaces and to localise the fog in some way instead of assuming it reaches to infinity. For the latter extension the simplest geometry to consider is that of a layer of fog which extends from ground level upto some specified height or altitude. A detailed explanation of the model can be found in Section 3.3 and the results of the calculations are given in Section 4.1.

1. Extinction coefficients determine the rate of decay of the light intensity in Beer's Law.

One problem with Willis' method is that it is still not totally physically based. The model requires a value for the sky background colour for a given density of fog. This colour ranges from blue (clear air) to white (very dense fog) and represents the colour seen when looking through an infinite range of medium. Willis does not mention how this would be determined and it can only be assumed that it is left as a variable parameter to be set by the user. The model as used by RWG provides a simple heuristic algorithm to calculate background colours. The formula has been implemented in this project but it still represents a very unrealistic way of determining sky colours since it only provides a single uniform colour to represent the whole sky. A simple observation of the real atmosphere on a clear day, for example, gives evidence for a range of shades which depend on elevation above the horizon and the position of the sun.

Clearly what is needed is a more complete model which follows through the scattering processes from source to sink (the viewers eye). In this respect the work of Klassen[7] is indispensable. This excellent paper describes a model for scattering in great detail and principally uses the results to calculate the colours of the sun and the sky for any position of the sun above the horizon in the presence of variable amounts of haze. He also describes an example of how the lighting model is applied to the rendering of a layer of fog lit by the sun without shadows. A number of simplifying assumptions are made which restricts the Earth's surface to be flat and the sun to be high in the sky. He also demands that the height of the layer to be small compared to the height of the atmosphere and the viewing position is on the ground. The calculations are again based on the assumption that the light is scattered only once between the source and the eye.

The second part of this project involved taking Klassen's model for scattering and applying it to the layer of fog without any restrictions. Thus the height of the layer can be any value and the position of the viewing point can be anywhere, even above the fog sitting in clear air. The great advantage to using Klassen's method is that it automatically calculates the background colour in any given direction. So it is technically possible to calculate the colours of the clear sky. All the extensions used in the simple atmospheric model were incorporated into this one too. The complete algorithm can be found in Section 3.4. Calculations were made using the model under the same conditions as those for the first model and the results are presented in Section 4.2. All the results have shown up interesting differences which will be discussed.

It is worth mentioning additional published work in the field of atmospheric illumination which cover other issues not addressed in this project. Two of the new themes include *shadows* and *variable or patchy fog*.

Work done in the area of shadows comes from Max[8] and Nishita *et al* [9] who report good results. The correct rendering of shadows must take into account the volumes of atmosphere which do not receive direct light and so do not contribute to the total scattered intensity. This has been considered by Max and Nishita. Their computations can also render shafts of light, like those through a window into a darkened and dusty room. In this case the total light reaching the eye comes solely from scattering. Max demonstrates his methods by rendering a natural outdoor scene in which the sun shines through openings between clouds producing shafts of light down to the ground.

There is also the problem with the rendering of patchy fog. The reason for choosing to consider layers of fog of uniform density is that the calculations are relatively easy. When the fog is generally patchy so that its density is spatially variable in some prescribed manner, then a new technique is required to render it. Kajiya and Von Herzen have developed a method for doing this[10]. Their approach is very general and can be applied to situations other than an atmospheric medium. However, it's quite mathematically sophisticated despite the idea being simple. Computational costs are also very high. No attempt has been made in this project to implement their model.

Another variation on the theme of atmospheric illumination has been investigated by Nishita and Nakamae[11]. They model the entire sky as a hemispherical source of light with varying intensity, calculated from the scattering in the atmosphere of the radiation from the sun. Objects on the ground are therefore not only illuminated by direct sunlight but also from other regions of the sky. The lighting produced by the sky on an object is calculating by integrating over this hemisphere. The algorithm is applicable to overcast conditions as well as to the rendering of shadows.

Lastly, there has been some work done by Rushmeier *et al* [12] which is an extension of the radiosity model to handle scattering. Analogous to the division of object surfaces into patches suitable for the radiosity calculations, the space between the objects is divided into small cubic volumes. The calculations not only deal with surface-to-surface interactions but also surface-to-volume and volume-to-volume interactions. The results are extremely realistic.

It should be mentioned that a major part of this project involved the development of a rendering system alluded to earlier. Anticipating that a requirement of the atmospheric models was the determination of range distances from the eye to all parts of the scene, a ray-tracing approach was taken. A central part of the ray-tracing method calculates such distances so they are readily available. RWG provided a polygonal model of an airfield scene to use in the rendering. The overall surface environment was considered to be purely diffuse so no inter-reflections were taken into account. This resulted in the ray-tracing approach becoming a *ray-casting* one. The rendering program was written in C (see Appendix C) and is discussed next.

2 Rendering System

This section describes the development of the ray-casting renderer. The basic framework of a renderer which could provide good (and realistic) images was deemed sufficient. The workings of the renderer can be divided into the following three broad functions; *input*, *processing* and *output*. The input deals with reading the data file (and possibly the command line) which describes the image to be computed. Processing of the data then takes place using the methods of ray-casting. This stage essentially calculates the colour of each pixel in the image. Finally, once the image has been computed the results are output for viewing. Two possibilities are common for viewing, (a) display on the computer screen as each scan line is calculated and (b) writing the pixel information to a disk file for viewing at a later time. Both approaches have been incorporated in to the rendering program, details are given below.

The processing stage can further be broken down into several sections. First a viewing system is set up based upon the input data. Pixel locations on the viewing plane are calculated and eye rays are projected through each pixel out into the three dimensional scene. Intersections with objects in the scene are then computed and the nearest intersection indicates the surface visible in that pixel. Lighting calculations can then be performed at the intersection point to determine the colour reflected from that particular surface towards the eye. The reflected colour then represents the colour allocated to that pixel.

Part of the lighting calculation determines whether the surface point is in shadow with respect to the light source. If it is in shadow, no direct light can be reflected other than that from some low intensity ambient illumination.

During the process of rendering, anti-aliasing can also be performed. Essentially, this means sampling the image plane at sub-pixel resolution by shooting more than one eye ray through each pixel in some prescribed distribution. The final colour for the pixel is computed by filtering the values returned by the samples. Note that all colour calculations are done in RGB space throughout the processing stage.

This has been a brief overview of the renderer. The following sections will outline in more detail the mechanics of various parts of the system. Discussion of that part of the renderer dealing with the atmospheric illumination models will be postponed until the algorithms have been given.

2.1 Input Data

The information needed by the renderer comes from two sources: a disk file and the command line. The bulk of the data is provided via a text file. An example file is given in Appendix B. This file contains viewing and lighting information followed by a listed description of the polygon model. The view and lighting parameters are specified in the input file by giving a keyword and one or three qualifying values. Table 1 below defines the meaning of each parameter.

Table 1: Keyword definitions

Keyword	Description
eyepos	Vector components of the viewing point
lookpoint	Vector components of the look point
vpd	View plane distance from the eye
hfov	Horizontal field of view in degrees
up	Vector components of the *up* direction
lightdir	Vector components of the direction *towards* the parallel light source
lightintensity	Intensity of the direct light source
ambientintensity	Intensity of the ambient light

The viewing parameters are explained in more detail in Section 2.2. Note that it is assumed that the light sources (direct and ambient) are not coloured. The intensities are therefore applied equally to each RGB component.

There is an additional parameter in the input file introduced by the keyword **scale**. The value following it is used to scale all the x, y and z coordinates of the polygon vertices. This will be found useful if it is required to express all positions in some convenient units (e.g. kilometres) appropriate to the problem. Note that the view point, look point and view-plane distance are not scaled and should be set accordingly.

The list following the keyword parameters specify for each polygon its RGB colour (0-255) and the x, y and z coordinates of its vertices. The end of the list is terminated when another keyword, **end**, is encountered.

When the input file has been read by the rendering program, calculations are performed using the viewing information to set up a viewing system, described in Section 2.2. When the polygon data is read it is stored in memory as a linked list, where each link contains information about polygon colour, normal vector, vertex positions, bounding volume, plus others (see definition in file *Render.h* of the program in Appendix C).

Input data is also given to the renderer via the command line as a set of options. The various options and their default values are summarised in Table 2 below.

The default values or actions become active if the option is not specified on the command line when the renderer is run. The meaning of the options relating to anti-aliasing and the atmospheric models with be covered in due course.

2.2 Viewing System

The system of viewing adopted here is the one suggested by Glassner[13]. The layout of the system is shown in Figure 1 and will be briefly explained.

Table 2: Command line options

Option	Description	Default
-r *x y*	Specify the image resolution as x by y pixels.	$x = 200$, $y = 150$.
-ns	Do not render shadows.	Shadows are rendered.
-s *n*	Anti-aliasing is performed using adaptive supersampling with n^2 samples per pixel.	Do not anti-alias ($n = 1$).
-j	Use jittered supersampling.	Use uniform sampling.
-nd	Do not display the image as it is rendered. If an output file is not specified via the **-o** option then an error occurs.	Image is displayed.
-o *file*	Write the image to a disk file when rendering has finished.	Do not write to file.
-f *rg ht*	Use fog of the given visual range and height.	Fog is not rendered.
-a *rg ht*	Set the visual range and height of the atmosphere (air).	$rg = 250$km, $ht = 8.4$km.
-m	Do not use the simple atmospheric model.	Use simple model.

The eye is located at E with position vector $\mathbf{E}$ and is looking in the direction of $\mathbf{G}$. The vector $\mathbf{G}$ can be directly calculated from the eye and look point positions. The two viewing angles 2θ and 2ϕ are the horizontal and vertical fields of view of the view plane (and hence the image), respectively and indicate the amount of perspective in the picture. If it assumed that the pixels in the image are square, then the two angles are not independent. Only one angle needs to be specified (2θ), the other can be computed from knowledge of the image resolution.

One more piece of information is needed before the view plane is uniquely defined. An *up* vector ($\mathbf{U}$ in the diagram) is required to indicate the orientation of the view plane about the look direction.

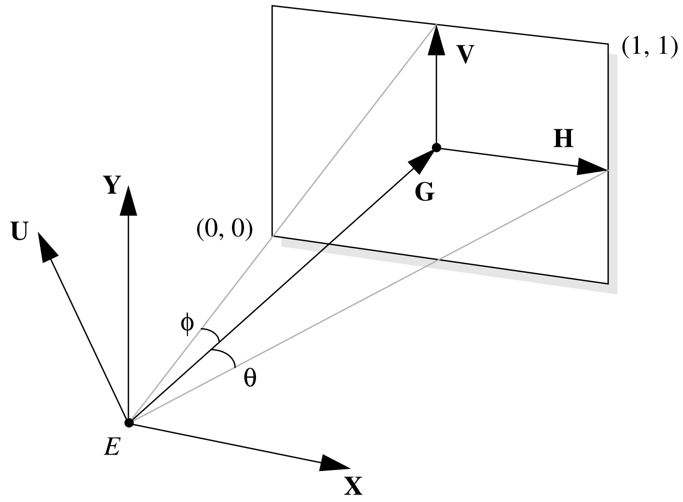


Figure 1: V iewing Layout

The purpose of the viewing system is to aid the computation of eye rays through the image pixels. To do this a coordinate system is setup on the view plane such that the bottom left-hand corner is the origin and the upper right-hand corner is at (1,1). Now construct two vectors **X** and **Y** from

$$\begin{aligned}\mathbf{X} &= \mathbf{G} \times \mathbf{U} \\ \mathbf{Y} &= \mathbf{X} \times \mathbf{G}\end{aligned} \tag{2-1}$$

These vectors are orthogonal to each other and parallel to the view plane. From the geometry it can be seen that the vectors **H** and **V** can be written

$$\begin{aligned}\mathbf{H} &= d \tan \theta \mathbf{X} \\ \mathbf{V} &= d \tan \phi \mathbf{Y}\end{aligned} \tag{2-2}$$

when **X** and **Y** are normalised and d is the distance of the view plane from the eye. Using **H** and **V**, a point on the view plane, say S, can be expressed as

$$\mathbf{S} = \mathbf{E} + d\mathbf{G} + (2x' - 1)\mathbf{H} + (2y' - 1)\mathbf{V} \tag{2-3}$$

where **G** has now been normalised and (x', y') is the position of S in the view plane coordinate system. If the point S represents the centre of a pixel, then in terms of the screen position (x, y)

$$x' = \frac{x + 0.5}{R_x}$$
$$y' = \frac{y + 0.5}{R_y} \tag{2-4}$$

where R_x and R_y are the image resolution. Now it is a simple matter to calculate the direction of the ray through S,

$$\mathbf{P} = \mathbf{S} - \mathbf{E} = d\mathbf{G} + (2x' - 1)\mathbf{H} + (2y' - 1)\mathbf{V} \tag{2-5}$$

The vector $\mathbf{P}$ when normalised, together with the eye position E, completely defines that ray. This ray can be written in parametric form as

$$\mathbf{r}(t) = \mathbf{O} + \mathbf{D}t \tag{2-6}$$

where $\mathbf{O}$ is the ray origin (eye point) and $\mathbf{D}$ is the unit vector along its direction. Here t represents the distance along the ray. This form will be found useful in later discussions.

2.3 Computing Object Intersections

Given the parameters specifying a particular eye ray, calculations are performed by the renderer to determine the nearest object in the scene with which it intersects. The renderer must test the ray against each polygon and if an intersection is found, decide whether it is the nearest. When the ray has been tested against every polygon, the renderer will find itself in one of two situations. Typically intersections have been found and the identity of the nearest is known together with the distance to the intersection point. Otherwise the eye ray has completely missed all objects and is pointing into the background. In the latter case a background colour is determined, which in this project will involve some calculations based upon an atmospheric illumination model. For the former, lighting calculations are performed at the surface point where the intersection has taken place. This is discussed later.

An algorithm is needed to execute the ray-polygon intersection test. The following method is due to Badouel[14] and is restricted to *convex* polygons. This is no limitation in the present case since all the polygons in the model are known to be convex.

The algorithm proceeds in stages and firsts determines whether the ray intersects with the *embedding plane* of the polygon. This plane is defined by the equation,

$$\mathbf{N} \bullet \mathbf{P} + d = 0 \tag{2-7}$$

where $\mathbf{N}$ is the unit normal, d is the plane constant and $\mathbf{P}$ is the position vector of any point in the plane. The normal vector is equal to the polygon's normal and can be computed from

$$\mathbf{N} = \frac{(\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)}{\left|(\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)\right|} \tag{2-8}$$

where $\mathbf{V}_0$, $\mathbf{V}_1$ and $\mathbf{V}_2$ are the position vectors of three of the polygon's vertices. The plane constant is given by

$$d = -\mathbf{V}_0 \bullet \mathbf{N} \tag{2-9}$$

Both the normal vector and the plane constant are pre-computed during the input reading stage and stored in the polygon linked-list where they are readily available.

From Equation 2-6 and Equation 2-7 we can evaluate the parameter t corresponding to the intersection point as

$$t = -\frac{d + \mathbf{N} \bullet \mathbf{O}}{\mathbf{N} \bullet \mathbf{D}} \tag{2-10}$$

Note first that if $\mathbf{N} \bullet \mathbf{D} = 0$ then the ray and plane (hence the polygon) are parallel and no intersection is possible. Secondly, if $t \leq 0$ the intersection lies behind the eye, which is of no interest (since we can't see it!). Thirdly, if t is greater than the current nearest intersection distance, then the present intersection cannot be the nearest and is rejected. Only if all these tests prove false do we proceed to the next stage.

We now determine whether the intersection point lies within the boundary of the polygon. First the polygon is divided into triangles, which have a common vertex. Then each triangle is tested in turn to see if the intersection point lies within it. This is done by setting up a simple (u, v) coordinate system on the plane containing the triangle and expressing the position of the intersection point in terms of this system. The point lies in the triangle if and only if (a) $0 \leq u \leq 1$, (b) $0 \leq v \leq 1$ and (c) $(u+v) \leq 1$. Details can be found in [14] and in the program code (file *Polygon.c* in Appendix C).

2.4 Speeding-up the Intersection Tests

Testing the eye ray against every polygon for an intersection is a time-consuming operation which gets worse the greater the number of objects in the scene. The operation is also more expensive the greater the image resolution, since more eye rays are generated. It is therefore advantageous to investigate ways of cutting down on the time taken to find the nearest intersection and so speed-up the calculation of the image.

Three very efficient and effective methods for speeding-up the algorithm have been reported in the literature and are generally based upon exploiting spatial coherence in the object scene. Testing then only occurs on a sub-set of the objects by determining which objects are likely to be intersected by a particular ray. The methods involve the spatial subdivision of the scene and different approaches have been taken.

Redefining the scene in terms of an *octree* is suggested by Glassner[15] (a similar algorithm is proposed by Samet[16]) and he reports speed-up factors in the range 3-30 times depending on scene complexity. A variation on the theme is offered by Kaplan[17] who subdivides the scene using *binary space partitioning* instead of octrees. The idea is essentially the same as Glassner's and obtains similar performance improvements. A slightly different approach is taken by Fujimoto *et al* [18]. Their algorithm is based on sorting the objects into a uniform voxel grid and the method of moving from node to node is accomplished by using a *digital differential analyser*. This proved to be extremely effective and they claim exceptional results which far exceed other currently used methods. Worthy of note is the result that the rendering time is virtually independent of scene complexity.

All of these approaches are very sophisticated and certainly produce substantial savings of time during the rendering stage. Unfortunately, due to the time constraints of the project, an implementation was not considered viable and a simpler idea, discussed next, was investigated and used.

2.4.1 Bounding Volumes

The approach taken for this project involves the use of *bounding volumes*. The bounding volume of an object is a simple primitive type (e.g. sphere or box) which has the smallest enclosed volume surrounding the object. During the ray intersection test with an object, a test is first performed on the bounding volume, which is usually easier and faster. Only if the bounding volume is intersected is the test applied to the object itself. We thus save time by avoiding unnecessary calculations on complex ray-object intersection tests.

The investigation into the use of an appropriate bounding volume suggested experimenting with spheres and axis-aligned boxes, to determine which would produce the most efficient scheme. The results overwhelmingly favoured using spheres. The determination of bounding spheres for polygons and the ray-sphere intersection test will be outlined below. Details for the boxes can be found in the program code (file *Bounds.c* in Appendix C) and also in [19] and [20] which provided the ray-box intersection algorithms.

2.4.2 Calculating the Bounding Sphere

This bounding sphere algorithm is actually based upon the axis-aligned box which encloses the polygon.

The bounding box is given simply by the maximum and minimum extents of the polygon vertices in each of the coordinate directions. If the polygon happens to be perpendicular to one of the coordinate axes, this approach will produce a degenerate box, which has a side of zero length. To avoid this situation, a small length (10^{-5}) is given to the side in the degenerate direction if necessary.

The centre of the bounding sphere is then just given by the centroid of the bounding box. Its radius is given by the maximum of the distances between the centre and the polygon vertices.

This algorithm is crude but still gives a good bounding sphere. Ideally, the best sphere is determined by simultaneously varying *both* the centre and the radius and looking for the minimum radius which encloses the whole polygon.

The radius (in fact the radius squared) and the centre position of the bounding sphere of each polygon in the scene is pre-computed when the polygon data is read and this information is stored in the linked-list.

2.4.3 Ray-Sphere Intersection

The basic requirement of the ray-bounding-sphere intersection calculation is to determine only that an intersection has taken place. The calculation of the position of the intersection point is not needed. The following algorithm is taken from Burger and Gillies[21] and at most proceeds in three stages.

The first stage tests whether the ray origin is inside the sphere. Referring to Figure 2, which shows a sphere of radius r, centre $\mathbf{C}$ and a ray as defined by Equation 2-6, we see that for this condition to be satisfied we must have

$$|\mathbf{C} - \mathbf{O}| < r \qquad (2\text{-}11)$$

In practice it is more efficient to use this expression in the form

$$|\mathbf{C} - \mathbf{O}|^2 < r^2 \qquad (2\text{-}12)$$

which avoids calculating a square-root (very inefficient). If this condition is satisfied, the sphere is intersected by the ray and no further testing is necessary. Otherwise we go on

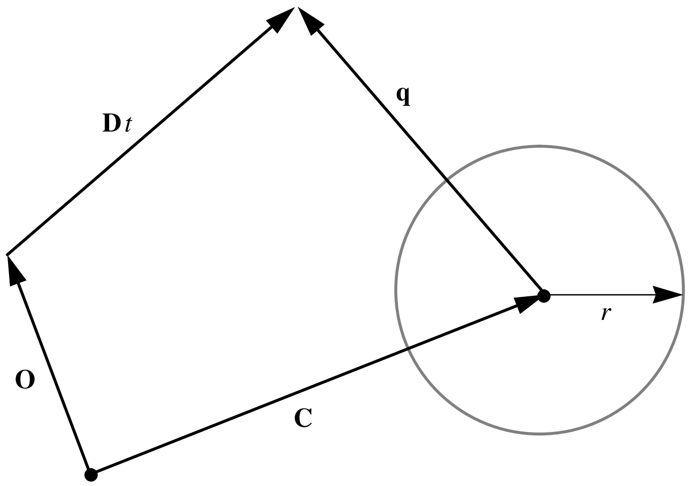


Figure 2: Ray-Sphere Intersection

to the next stage and calculate the value of the parameter t for the point of closest approach to the sphere. This point is given by the vector $\mathbf{q}$ in Figure 2 relative the sphere's centre and is perpendicular to the ray. We can write

$$\mathbf{O} + \mathbf{D}t = \mathbf{C} + \mathbf{q} \tag{2-13}$$

giving

$$t = (\mathbf{C} - \mathbf{O}) \bullet \mathbf{D} \tag{2-14}$$

Note that $\mathbf{C} - \mathbf{O}$ will have already been calculated, so that t is obtained by an additional dot product. At this point we compare the value of t with zero and if $t < 0$ then the point of closest approach lies behind the eye and the intersection, if any, is of no interest. If $t \geq 0$ then a third stage in the calculation is done.

We need to compare $|\mathbf{q}|^2$ with r^2. It can be shown that

$$|\mathbf{q}|^2 = (\mathbf{C} - \mathbf{O})^2 - t^2 \tag{2-15}$$

If $|\mathbf{q}|^2 < r^2$ the point of closest approach lies inside the sphere and the ray has intersected its surface. Otherwise we can conclude that the ray misses the sphere completely. Using this algorithm in the rendering system, speed-up enhancements in the region of 20% were achieved.

2.5 Lighting Calculations

The most important and most extensive part of the rendering system is the stage concerned with calculating the colour and intensity of the light reflected from objects. In the present work additional calculations are performed to take into account the light scattered by the atmosphere, contributing to the total light received by the eye. The background to the latter types of calculation will be covered in a later section. Here, we will give an overview of the calculations involved with the interaction of light with the polygon surfaces.

2.5.1 Diffuse Reflection

It was mentioned earlier that only diffuse reflection of light is considered. The reflection model of Lambert is used, in which the intensity of the reflected light is proportional to the cosine of the angle between the surface normal and the direction to the light source. The constant of proportionality is just the product of the diffuse colour of the polygon and the light source intensity. We can express this by

$$I_D^{(\lambda)} = I_S^{(\lambda)} k_D (\mathbf{N} \bullet \mathbf{L}) \qquad (2\text{-}16)$$

where I_S is the intensity of the incident light, k_D the diffuse reflectance or colour of the surface, $\mathbf{N}$ is the surface normal and $\mathbf{L}$ is the unit vector pointing to the light source. I_D is the diffusely reflected intensity. The superscript λ indicates that the equation is wavelength or colour dependent. It was noted earlier that the lighting calculations are performed in a normalised RGB space, hence Equation 2-16 represents three separate ones, one for each component. The light source is white, so $I_S{}^{(R)} = I_S{}^{(G)} = I_S{}^{(B)}$. Note that because we're only considering a single, parallel light source, the term $(\mathbf{N} \bullet \mathbf{L})$ is constant over the whole polygon surface. Consequently, the images will look faceted. This is not so much a problem since most of the scene is flat, but it raises questions about using such a model scene in realistic rendering.

2.5.2 Ambient Illumination

An additional source of light is considered in the calculations. An *ambient* term can be added to the right-hand side of Equation 2-16 to mimic (crudely) the light incident from global illumination. This term can be expressed by

$$I_A^{(\lambda)} = I_{AS}^{(\lambda)} k_A^{(\lambda)} \qquad (2\text{-}17)$$

where k_A is the ambient reflectance of the surface and I_{AS} is the ambient intensity. Here k_A is taken to be equal to k_D. The ambient light is also white, hence $I_{AS}{}^{(R)}$ = $I_{AS}{}^{(G)} = I_{AS}{}^{(B)}$.

Ambient illumination has its use in this project, primarily, to avoid rendering black shadows and surfaces which do not receive direct light. It also has a use in the atmospheric models which will be discussed later.

2.5.3 Colour Clipping

The problem with performing lighting calculations in RGB space is there is no guarantee that the component intensities remain within the range 0-1. Frequently, colour values can occur which are not capable of being displayed on a (RGB) monitor screen. When this situation arises, some form of colour clipping is necessary and there are several ways of doing this.

Hall has suggested three methods[22] and the one adopted here is called *intensity scaling*. In this scheme, if any of the RGB components are greater than unity, all the components are scaled such that the offending component becomes equal to one. According to Hall this clipping method maintains the hue and saturation of the original colour but reduces its intensity.

Clipping is performed in the rendering stage only at the very end of the lighting calculations and before colour values are allocated to image pixels, prior to being displayed on the monitor or stored in a disk file.

2.6 Shadows

During the lighting calculations, if the option to render shadows is active, a test is made to determine whether the polygon surface at the intersection point is in shadow with respect to the light source. Testing for shadows is very simple in theory. We first construct a ray whose origin is the point of intersection and whose direction is towards the light source. This ray, called the *shadow ray* or *shadow feeler*, is then tested for an intersection with all the polygons, as was the case with finding the original intersection. As soon as a hit is found then we are guaranteed that the point is in shadow. This is true for two reasons. Firstly, we have assumed that all objects are opaque. Secondly, we assume that the light source is effectively an infinite distance away, so that no polygons lie beyond it.

Note that an intersection test with the polygon that was originally intersected should be avoided to prevent what has been termed *ray-tracing acne* (RTA)[23]. This is caused by inaccuracies in the calculated position of the intersection point, causing it to fall a very short distance away from the polygon surface. If the point should fall in the space behind the polygon with respect to the light source, then an intersection of the shadow ray with its polygon will be detected. This will produce the appearance of a dark spot in the image.

A problem arose in the implementation of this simple algorithm and its source is due to some of the polygons in the model being *coincident*. In such cases omitting the original polygon from the shadow intersection tests is not sufficient to prevent RTA. In an attempt to circumvent this, an additional test was included which demanded that the distance to the intersection point from the shadow ray origin be greater than some tolerance value. This value was set to be much greater than any inaccuracies which the calculation were thought to yield, but small enough not to miss any real, non-coincident polygons. A value of 10^{-5} was used. The results were only partly successful and some RTA remained. This problem needs further investigation.

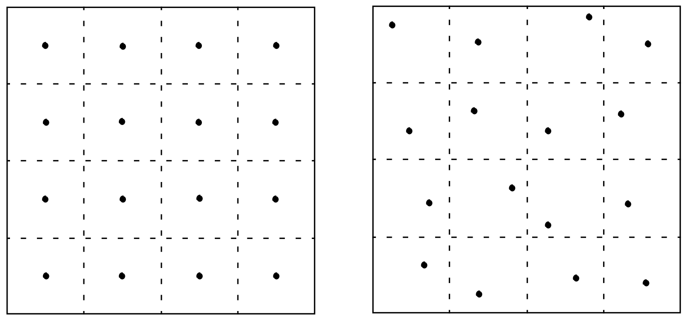
2.7 Anti-Aliasing

An anti-aliasing scheme has been implemented to improve the quality of the images and the idea for it comes from the ray-tracing system written by Kolb[24]. The scheme used in this project is basically the same, with minor alterations.

The default action of the renderer is not to anti-alias the image. In this case each pixel is sampled once at its centre. If you choose to invoke the anti-aliasing using the appropriate command-line option (see Table 2), then the image is *supersampled* using several samples per pixel. For example, specifying '-s 4' indicates using sixteen samples per pixel.

There are two methods available for determining how the samples are distributed within the pixel. These perform *uniform* and *jittered* sampling. In uniform sampling, the pixel is subdivided into sub-pixels and the centre of each sub-pixel is sampled. Using the example given above, each sample in a uniform subdivision of the pixel is positioned as shown in Figure 3a. In jittered sampling, the sample points in the uniform sampling picture are displaced by small random amounts from their original positions. An example is given in Figure 3b.

The default distribution used by the renderer is uniform. To use jittered sampling, include the option '-j' on the command-line.



(a) Uniform (b) Jittered

Figure 3: Sampling Distributions.

A problem with uniform supersampling, specially if the sample density is small, is that aliasing artifacts can still remain, albeit at a reduced perceptible level. The appearance of jagged edges may still be detectable, particularly so with ones which have very large or very small gradients relative to the image raster. The use of jittered sampling improves the situation but with the side effect of adding noise to the image. Nevertheless, uniform sampling with nine or more samples per pixel has been found satisfactory.

The determination of a suitable sampling distribution is an active field of computer graphics. Extensive investigations of the effectiveness of various approaches have been conducted (see for example Cook[25]). It has been found that jittered sampling is only useful if the random fluctuations in the positions of the samples follows a Poisson distribution. This essentially means that the scattering of samples is truly random but they cannot be closer to one another than a set minimum distance. This particular distribution also minimises the amount of noise contributed by the jittering to the image.

In practice, one way of establishing a Poisson distribution of the sampling points is to use a *uniform random number generator*, commonly available on many computers. Additionally, for the most efficient implementation, the random numbers should be saved in a look-up table and accessed accordingly. This idea has been used and comes from the method reported by Cychosz[26].

The supersampling is also performed *adaptively* in the following sense. Only pixels in regions of the image which have large contrast variations are supersampled. The method thus has the ability to detect large changes of intensity corresponding, typically, to object edges. This scheme improves the efficiency of the anti-aliasing in that most of the effort and calculation is concentrated in the part of the image that really needs it. Put another way, large areas of uniform colour will not be, unnecessarily, anti-aliased. The adaptive approach therefore saves rendering time.

The large contrast areas are detected in the following way. Each pixel on the current scan line is first sampled once. Then for each pixel the contrast between it and its four immediate neighbours is computed, and if the contrast is greater than a pre-set maximum in any of the RGB components, then the pixel *and* its neighbours are supersampled. This process is repeated for the current scan line until a pass is made without any pixel being supersampled. The RGB contrast values used are 0.25, 0.20 and 0.40 respectively.

Filtering of the sampled colours is finally necessary to arrive at a representative value for the pixel. The filter used here is a *box* type. A box filter simply takes the arithmetic mean of the samples. More realistic filters could have been used, such as the *gaussian* filter, which weights the values of the samples according to their distance from the centre of the original pixel in the averaging. This sort of filter actually mimics more closely the intensity variation across a pixel produced by the monitor electron beam.

2.8 Image Output

The image output stage of the rendering is very machine dependent. Since the program was developed initially on a PC and then extended to run on Sun hardware, output to both types of machine is considered. First a few words about the computer set-ups.

The PC environment consists of an Intel 80386 32-bit processor running at 33MHz supported by a 80387 floating-point coprocessor. Limited graphics facilities are available on the PC so RWG kindly loaned their PC4000, a high performance video graphics board[27]. The PC4000 supports 24 bits per pixel (16 million colours), has a maximum displayable resolution of 768x576 pixels (PAL standard) and has a dedicated Motorola 68020 processor to handle all graphics operations. Software support is provided via a graphics library.

The Sun machine is a SPARCstation IPC which has a RISC processor capable of 2.1MFLOPS with floating-point coprocessor. High quality graphics is provided by a RasterOps card driving a Sony Trinitron monitor, providing 24 bits per pixel and a resolution of 1152x900 pixels. Program development is conducted in the SunView windowing environment[28] based upon the Pixrect graphics library[29].

The setup with the renderer at the present moment is that it runs on the IPC and outputs to a SunView window and/or Sun rasterfile, whichever the user chooses (see Table 2). The reason for this is that the IPC runs as much as five times faster as the PC (also see below). However, we have developed a pair of complementary programs which enable the rasterfiles to be displayed on the PC4000 (see files *Encode.c* and *Show.c (PC4000)* in Appendix D). The first runs on the IPC and converts the rasterfile to a run-length encoded file. The encoding is necessary to cut down on the amount of disk space taken up by the images [1] (1.77MBytes, PAL resolution). The second program, running on the PC, then decodes and simultaneously displays the image file on the PC4000. A program has also been written to display a rasterfile image in a SunView window for viewing (see file *Show.c (SUN)* in Appendix D).

Additional programs have been created which gamma correct the displays (see files *Gamcor.c(PC4000)* and *Gamcor.c(SUN)* in Appendix D). Gamma correction is necessary to remove the non-linear response of the phosphors on the monitor screen to

1. Sun provides an encoding scheme for rasterfiles based on byte sized data. Since full colour needs at least 3 bytes (24 bits) this scheme is very inefficient and gives only very small amount of compression. An encoding method has been devised here which uses 4 bytes as the base unit since rasterfiles are stored on disk at 32 bits per pixel. The PC4000 frame buffer also has the same depth. Compression ratios are now much greater .

changes of intensity. The gamma values are generally different from monitor to monitor, but typically range from 2.0 to 3.0. For really accurate colour work it would be necessary to measure the gamma value of the monitor. However, without the means to do this an estimate was used. A gamma value of 2.5 was thought adequate.

Rendering times for images varied greatly between machines. The time is also dependent on image resolution, whether shadows were included and if anti-aliasing is used. Run times of three hours were typical of images generated by the PC (full resolution, nine samples per pixel). This was reduced to a manageable 50 minutes on the IPC. Occasionally, during off-peak periods, use was made of the central network file-server (a very powerful machine) and rendering times were further compressed to about 25 minutes.

3 Atmospheric Illumination

This section will cover aspects of atmospheric scattering and absorption relevant to the project. After explaining the basics physics involved in Section 3.1, later sections will use this knowledge to derive algorithms for the simulation of varied atmospheric conditions. As was reported in the introduction, we will consider two illumination models. The first, simple, model will be referred to as **Model1** and is based on a combination of physical principles and rule of thumb. **Model1** is discussed in Section 3.3. The second, more sophisticated model, **Model2**, is then described. **Model2** is totally based on physics and is explained in Section 3.4.

3.1 Physics of Scattering

Of the three types of interaction between light and a transparent medium, i.e. scattering, absorption and emission, the process of scattering predominates in the atmosphere under all sorts of conditions. A small amount of absorption occurs and emission is only significant in the infra-red region of the spectrum. The absorption of visible light usually reappears as infra-red or heat radiation. Therefore, most of the discussion concentrates on the scattering of light from small particles suspended in the atmosphere and from the air molecules themselves.

3.1.1 Scattering Coefficients

The intensity of a beam of light travelling through a scattering medium is attenuated due to light being lost from the beam direction. The situation is illustrated in Figure 4a, which shows light incident on an element of the medium and scattering events taking place in various directions. The attenuation can be precisely defined. Refer to Figure 4b. Assuming that the change in intensity due to scattering, dI_λ, over an element of thickness ds in the direction of travel is proportional to ds and the mass density, ρ, then we can write[30]

$$dI_\lambda = -\sigma_\lambda \rho I_\lambda ds \tag{3-1}$$

where I_λ is the incident intensity. The constant of proportionality σ_λ is called the *mass scattering coefficient* (units kg $^{-1}$m^2) and all quantities are dependent on the wavelength of the radiation.

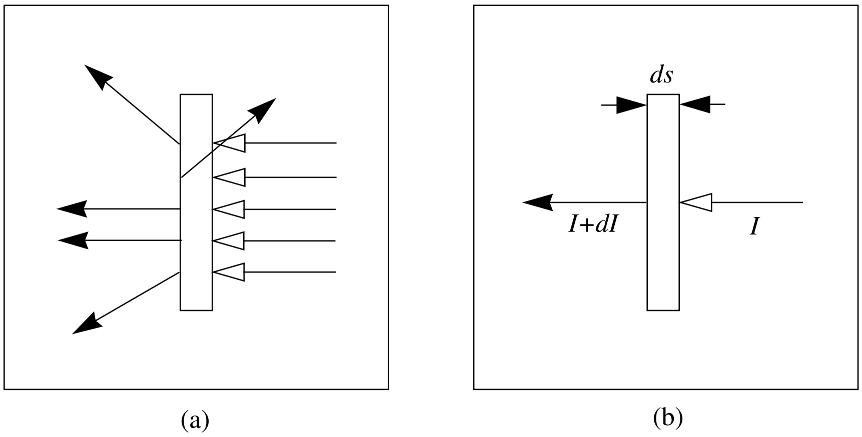


Figure 4: Scattering processes

It is convenient to introduce the volume-scattering coefficient using

$$\alpha_\lambda = \sigma_\lambda \rho \tag{3-2}$$

which has units of m $^{-1}$ and is generally better known as the *extinction coefficient*.

The extinction coefficient and its wavelength dependence are central to the correct simulation of atmospheric scattering. Values for the coefficients are needed for various conditions (medium types and densities) and the results produced by Foitzik[31] have been found to be very useful.

The graphs inFigure 5 show measurements of the relative extinction coefficients as a function of a quantity called the *visual range*. The visual range is an indicator of the density of the medium and is the distance equivalent to 98% of the attenuation of light. Thus, the intensity of light reaching the eye, reflected from an object $x$km away, is only 2% of the original when it has travelled through a medium with a visual range of $x$km. As a result the object is just beginning to blend in with the background colour.

The relative extinction coefficients are given in Figure 5 for the red, green and blue (unspecified) regions of the visible spectrum and cover a wide range of conditions, from very thick fog to clear sky. Note the almost colour independence of the scattering due to fog and the predominance of the blue component in pure air, which gives rise to the blue colour of the sky.

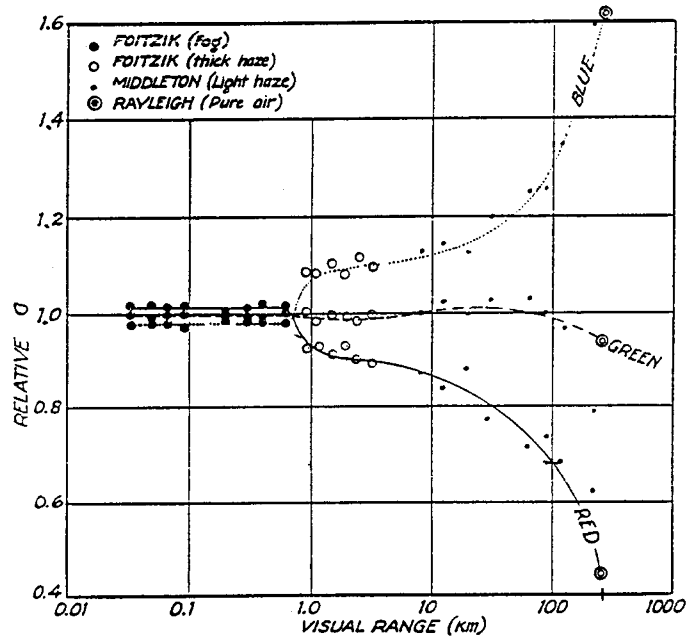


Figure 5: Relative extinction coefficients as a function of visual range

The curves through the experimental points were used in the rendering program. To achieve this the graphs were digitised and incorporated into a look-up table (see file *Coeffs.c* in Appendix C).

From the relative values, the absolute extinction coefficients are calculated from *Koshmeider's equation* [31]

$$\alpha = \frac{3.91}{R} \tag{3-3}$$

where R is the visual range and α is the *generalised* extinction coefficient. This value is then modified according to the graphs to give values for $\alpha^{(R)}$, $\alpha^{(G)}$ and $\alpha^{(B)}$.[1]

1. The atmospheric model used by R WG did not consider the colour dependence of the extinction coefficient and simply used the value given by Equation 3-3.

3.1.2 Angular Dependence

A feature of the scattering process is its dependence on the angle between the direction of the incident and the scattered radiation. The intensity of the scattered light forms a three-dimensional pattern, $\beta_{sc}^{(\lambda)}(\theta, \phi)$, which depends on the size and shape of the scattering particles and on the wavelength of the light involved. Figure 6 shows an example of a single scattering event defined by the polar angles θ and ϕ. The angular dependence is predominantly on ϕ[7], so it is sufficient to write $\beta_{sc}^{(\lambda)}(\phi)$.

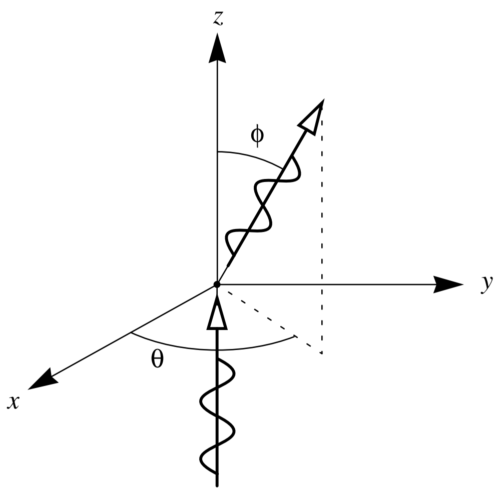


Figure 6: The geometry of a scattering event

By averaging the scattering function over all directions to remove the angular dependence, results in a coefficient essentially equal to the mass scattering coefficient and hence the extinction coefficient.

The distribution of β_{sc} as a function of angle is generally anisotropic. In clear air, Rayleigh scattering predominates which has a scattering function proportional to $(1 + \cos^2\phi)$, hence as much is scattered forward than backwards, with a reduced intensity in a direction perpendicular to the incident light. For larger particles the scattering is much more concentrated in the forward direction. The larger the particle the more concentrated it becomes. This is certainly true for fog. The shadow work of Nishita *et al* [9] mentioned in the introduction uses angular functions proportional to (1+ $A\cos^k\phi$) where the exponent, k, is equal to 16 and 64. The phase functions of Blinn[5] serve the same function as β_{sc} and he has investigated various forms.

The incorporation of the angular dependence of the scattering coefficient was not undertaken in this project and is left for future developments. Consequently, the extinction coefficients and scattering functions are used identically. However, in describing the atmospheric illumination model in Section 3.4 the distinction is still made.

3.1.3 Beer's Law

The expression given in Equation 3-1 is one of the basic premises of a branch of physics call *radiative transfer theory*, whose development is both sophisticated and mathematical[32]. Both Klassen[7] and Willis[6] use this equation to derive a formula which goes by the name *Beer's Law*. Its derivation is simple, as will be now shown.

Using Equation 3-2 in Equation 3-1 and integrating we get

$$\int_{I_0}^{I_f} \frac{dI}{I} = -\alpha \int_0^x ds \tag{3-4}$$

where I_0 is the original intensity and I_f is the final intensity after travelling a distance x through the medium. The wavelength dependence is still assumed. Performing the integration we find that

$$I_f = I_0 e^{-\alpha x} \tag{3-5}$$

which is Beer's law. Hence we see that the intensity of light falls exponentially with distance travelled. This result will be extensively used in deriving the atmospheric scattering algorithms.

Note that α has been treated as a constant in the right-hand side of Equation 3-4.

This would be true if the medium is homogeneous and of uniform density. It is assumed that this is the case throughout the project.

3.1.4 Optical Depth

A few words are necessary about a term which will be used on occasion and whose concept is important to know. In Equation 3-5 the combination αx is a dimensionless quantity called the *optical depth* or *optical distance* and is denoted $\tau(x)$. Thus we can write

$$I_f = I_0 e^{-\tau} \tag{3-6}$$

and

$$\tau(x) = \int_0^x \alpha ds \tag{3-7}$$

The optical depth is the fundamental quantity to consider since it indicates almost directly how much attenuation is present.

3.2 A Basis for the Illumination Models

Before describing the details of the atmospheric illumination models, a preamble is appropriate which covers aspects of the models common to both.

Each model performs two types of calculation to determine the intensity and colour of the light reaching the eye, which depend on whether we're looking at an object or looking into the background. In the former case, the eye-ray has intersected a polygon while in the latter, no intersection has been found.

The calculation of the intensity from the background is model dependent but essentially finds the contribution of the light scattered from the atmosphere alone. The calculation of the intensity coming from the direction of an object can be further broken down into two parts. The first part is called the *object component* and includes the intensity reflected from the object surface, attenuated by the atmosphere before reaching the eye. The second part, called the *fog component*, is the contribution from the intervening thickness of atmosphere. This component is simply the sum total of the scattered light from all points along the line of sight, attenuated by factors depending on the distance travelled to the eye point. The total intensity is then just the sum of the object and fog component.

Note that the word 'fog' is used here just as a term to describe an atmospheric medium of any type from clear air to thick fog. Also note that the calculation of the background intensity is just a special case of the fog component when the object component is zero. This fact is used to advantage in **Model2**.

Now we're in a position to describe both illumination models in detail. **Model1** will be covered first followed immediately by **Model2**. In both cases the model is applied to the situation of a layer of 'fog' on the ground. Remember throughout the next sections, all intensity calculations are colour dependent even though this is not explicitly stated.

3.3 Illumination Model1

Model1 is the algorithm based on Willis[6] and used by RWG. The model atmosphere (as Willis describes it) can be summarised by the following expression,

$$I(t) = I_0 e^{-\alpha t} + I_B \left[1 - e^{-\alpha t}\right] \qquad (3\text{-}8)$$

where t is the distance to the object (returned by the eye-ray polygon intersection calculation), I_0 the intensity of the light reflected by the object, α is the extinction coefficient of the intervening atmosphere and I_B is the background intensity. The derivation of Equation 3-8 can be found in Willis' paper, where it is assumed that the fog is uniform and extends to infinity in all directions. The eye and the object are therefore always in the fog.

The first term in Equation 3-8 is the object component (cf. Equation 3-5) and the second is the fog component. The calculation of I_B is unspecified by Willis, but a simple formula has been devised by RWG[31]. This is given in RGB by

$$\begin{aligned} I_B^{(R)} &= \frac{1}{2e^{R/250} - 1} \\ I_B^{(G)} &= \frac{1}{2 - e^{-R/250}} \\ I_B^{(B)} &= 1 \end{aligned} \tag{3-9}$$

where R is the visual range in km. Note how the intensities are normalised to the blue component. Hence as $R \to 0$, i.e. the atmosphere becomes more and more foggy, the background tends to white, whereas when $R \to 250$, the background tends to a blue colour. The factor 250 is taken to be the visual range of clear air in kilometres (see Figure 5).

The calculation of I_0 is given by Willis simply as

$$I_0 = k_D I_B \tag{3-10}$$

where k_D is the diffuse reflectance of the surface. One the extensions that we have made to this model is to replace Equation 3-10 by Equation 2-16 and Equation 2-17, which introduces direct lighting into the model. However, this does not include any attenuation of the direct light by the atmosphere before reaching the object surface. Hence I_0 is taken to be the same whether or not a participating medium is present.

The results of the model will now be applied to the layer of fog. This part of the algorithm is another of our extensions of Willis' method. The fog extends from the ground to some specified height and the calculations will depend on the positions of the eye and object relative to the fog. It is assumed that the air above the fog contributes nothing but a constant colour into the computations.

3.3.1 Background Intensity Calculation

Two situations are possible when calculating the intensity in the direction of the background. These are depicted in Figure 7. When the eye is above the fog in position E_1, say, then we're looking at clear sky and the intensity seen in this direction is given by

$$I_1 = I_B^{(air)} \tag{3-11}$$

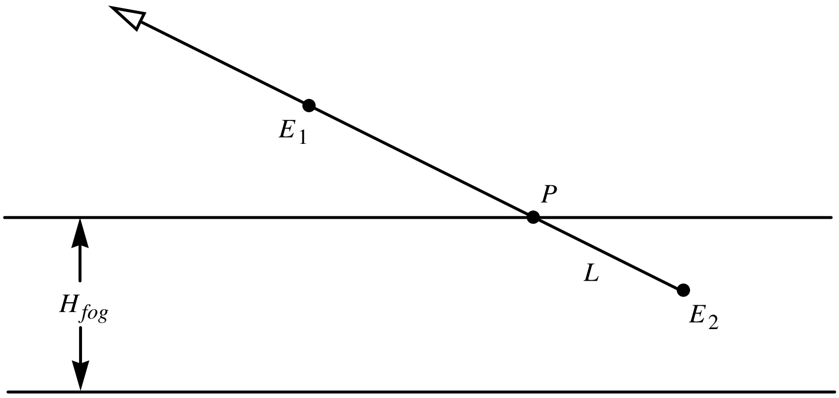


Figure 7: Calculating the Background Intensity

where $I_B^{(air)}$ is given by Equation 3-9 with $R = 250\text{km}$. Otherwise the eye is in the fog at E_2, for example, and in this situation, given the thickness of fog along the line of sight, L, the background intensity is

$$I_2 = I_B^{(air)} e^{-\alpha_{fog} L} + I_B^{(fog)} \left[1 - e^{-\alpha_{fog} L} \right] \tag{3-12}$$

where $I_B^{(fog)}$ is given by Equation 3-9 for the specified visual range of the fog.

Equation 3-12 has been derived from Equation 3-8 by imagining an object placed at the point P, which has a colour the same as the sky background. Note that I_1 is given by Equation 3-12 when $L = 0$.

The calculation of the distance L involves finding the position of P, the intersection of the eye-ray with the plane representing the top of the fog. This problem has already been covered in another context; see Equation 2-10 and Equation 2-6.

3.3.2 Object Intensity Calculation

Calculation of the intensity of light reaching the eye from the direction of an object is again dependent on the eye and object positions. There are four situations to consider.

(a) *The eye and object are above the fog*

In this case no intervening fog is present and we just see the colour reflected from the object surface. This is the simplest situation expressed by

$$I_3 = I_0 \tag{3-13}$$

(b) *The eye and the object are both in the fog*

This is a case where Equation 3-8 is directly applicable. Hence

$$I_4 = I_0 e^{-\alpha_{fog} t} + I_B^{(fog)} \left[1 - e^{-\alpha_{fog} t}\right] \tag{3-14}$$

Note that $I_3 = I_4$ when $\alpha_{fog} = 0$.

(c) *The eye is above the fog, the object is in the fog*

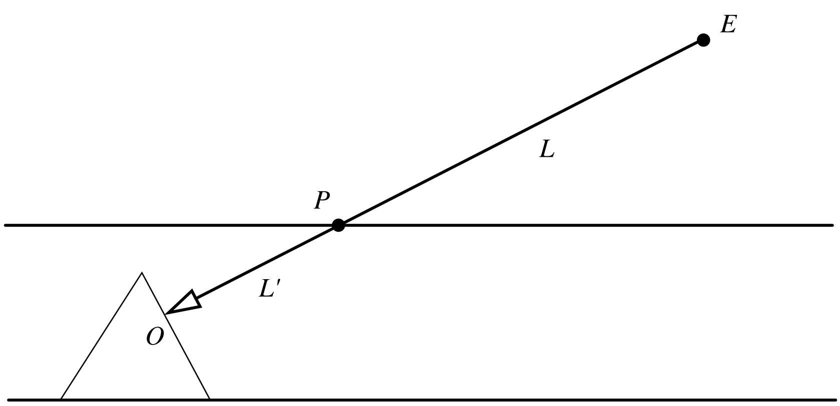


Figure 8: Eye above fog and object in fog

This situation is shown in Figure 8 and is equivalent to moving the eye point, E, to P at the top of the fog and applying Equation 3-14 with $t = L'$. Now, $L + L'$ is the intersection distance and L can be calculated as in Section 3.3.1. Hence L' can be found.

(d) *The eye is in the fog, the object is above the fog*

This case is depicted in Figure 9. Only the distance, L, in the fog along the line of sight contributes to the calculation and we can again imagine the object point, O, at position P at the top of the fog. This distance is the same as that in Figure 7 and is similarly calculated. The resulting intensity reaching the eye is then given by Equation 3-14 with $t = L$.

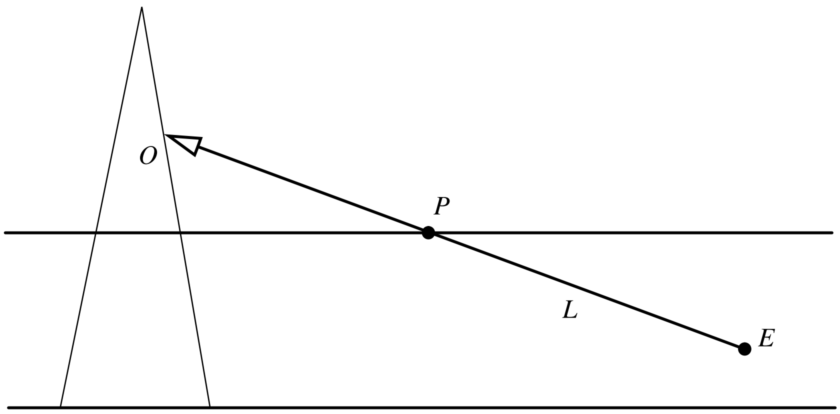


Figure 9: Eye in fog and object above fog

3.4 Illumination Model2

This model is based, as was noted earlier, on the approach taken by Klassen[7]. It starts from scratch and follows through all the interactions between the light and atmosphere from source to eye. The method is best illustrated *in situ* by performing the calculations on a specific example. Therefore, we shall go straight into the algorithm involving the layer of fog. Note that in this model the effect of the air above the fog is included, both in the object and fog components and in the background calculation. The air, like the fog, is assumed to be uniform in density and extends upto some specified height (see Section 3.5) greater or equal to the height of the fog. No objects are assumed to be higher than the top of the atmosphere.

As with **Model1**, there are several situations to consider. An expression is derived for the object component which can be applied to all the cases involving the position of the eye and object. The derivation of the fog component, however, will be illustrated for one situation only, namely, where both the eye and the object are inside the fog. It has been found that, given an additional expression for the fog component when both the eye and object are above the fog, then equivalent expressions for all other situations can be quickly obtained, *including* those for the background intensity calculations. The results are quoted in detail.

Note that the work of Klassen has been used as a guide only. The construction of **Model2**, described in the rest of Section 3.4, is all original work.

3.4.1 Object Component

Consider the diagram shown in Figure 10. The fog has a height H_{fog}, while the atmosphere extends up to H_{atm}. The eye is located at E at a height H_{eye} and is looking at an object surface at O. The surface point has a height H_{obj} above the ground. Light is incident on the atmosphere at an angle ϕ with an intensity I_0.

Before the light reaches the object surface, it travels through a distance, d_1, of air and then a distance, d_2, of fog. Thus, using Beer's law

$$I_1 = I_0 e^{-\alpha_{air} d_1} \tag{3-15}$$

and

$$I_2 = I_1 e^{-\alpha_{fog} d_2} \tag{3-16}$$

Combining Equation 3-15 and Equation 3-16 gives

$$I_2 = I_0 e^{-\alpha_{air} d_1 - \alpha_{fog} d_2} \tag{3-17}$$

which is the intensity arriving at the surface. We can now use Equation 2-16 to find the reflected intensity, I_3, by substituting I_2 for I_S.

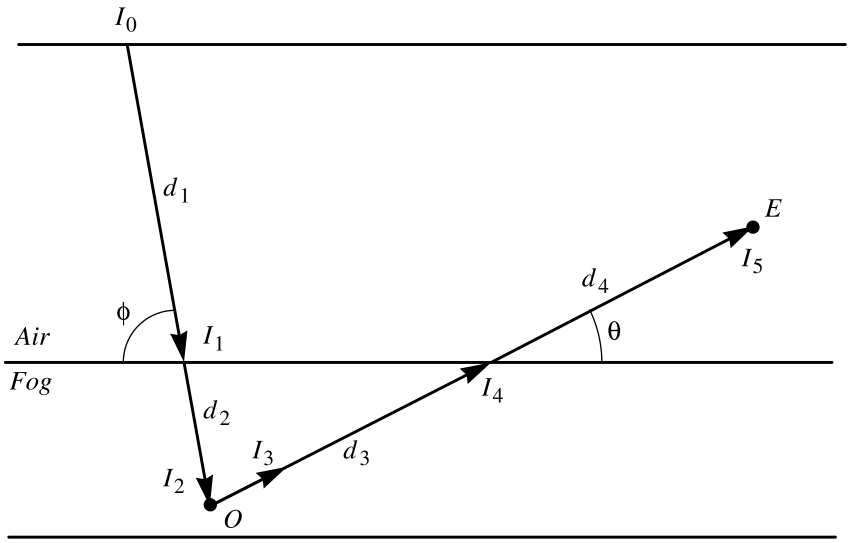


Figure 10: Calculation of the Object Component

There is a secondary source of light incident on the object, scattered by the atmosphere at some small angle, which produces a slight increase in the illumination of the surface. This effect is approximated by adding an ambient term to the calculation of the reflected light intensity and the expression in Equation 2-17 is used. The reflected intensity can be written

$$I_3 = I_2 R(\varphi) + I_A \tag{3-18}$$

where $R(\varphi)$ is the fraction of light reflection by the surface. Having calculated I_3, we now continue to follow the path of the light back to the eye. The light travels through thicknesses, d_3, of fog and, d_4, of air, so we can write

$$I_4 = I_3 e^{-\alpha_{fog} d_3} \tag{3-19}$$

and

$$I_5 = I_4 e^{-\alpha_{air} d_4} \tag{3-20}$$

giving

$$I_5 = I_3 e^{-\alpha_{fog} d_3 - \alpha_{air} d_4} \tag{3-21}$$

The intensity I_5 is the intensity received by the eye. Equation 3-17, Equation 3-18 and Equation 3-21 constitute the calculation of the object component. The calculation of the distances d_1 through d_4 is obtained from the geometry and t, the eye-object distance. So for the example shown in Figure 10

$$\begin{aligned} d_1 &= (H_{atm} - H_{fog}) \csc\phi \\ d_2 &= (H_{fog} - H_{obj}) \csc\phi \\ d_3 &= t - d_4 \\ d_4 &= (H_{eye} - H_{fog}) \csc\theta \end{aligned} \tag{3-22}$$

The angles ϕ and θ can be calculated from the components of the light vector and the eye-ray respectively. The values determined for the distances d_1 through d_4 depend on the positions of the eye and the object and can in some cases be zero.

3.4.2 Fog Component

The detailed derivation of the fog component for the case shown in Figure 11 is now presented. The eye is at E and the object surface point at O. We wish to find the intensity of light scattered by the fog at all points along OE. To do this consider a typical scattering event as depicted.

The intensity of the direct light before being scattered, I_2, is given by

$$I_2 = I_0 e^{-\alpha_{air} d_1 - \alpha_{fog} d_2} \tag{3-23}$$

The light is then scattered at an angle ψ in the direction of the eye and we can write

$$I_3 = I_2 \beta_{sc}^{(fog)} \tag{3-24}$$

where $\beta_{sc}^{(fog)} = \beta_{sc}^{(fog)}(\psi)$ is the scattering function of the fog. Finally, the light travels a further distance d_3 and is attenuated, hence

$$I_4 = I_3 e^{-\alpha_{fog} d_3} \tag{3-25}$$

Combining gives

$$I_4 = I_0 \beta_{sc}^{(fog)} e^{-\alpha_{air} d_1} e^{-\alpha_{fog}(d_2 + d_3)} \tag{3-26}$$

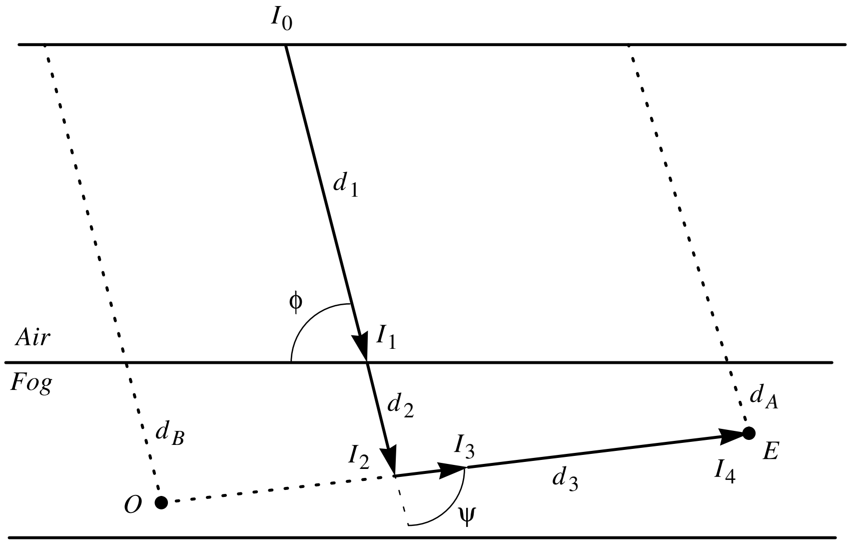


Figure 1 1: Calculation of the Fog Component

I_4 is the intensity reaching the eye due to only one point in the fog. To find the total intensity we integrate the contributions from all points along OE, hence

$$I = \int_0^t I_4 d d_3 = I_0 \beta_{sc}^{(fog)} e^{-\alpha_{air} d_1} \int_0^t e^{-\alpha_{fog}(d_2 + d_3)} d d_3 \qquad (3\text{-}27)$$

where t is again the eye-object intersection distance. To evaluate the integral we can express d_2 in terms of d_3. From the geometry, d_2 is a linear function of d_3 and it can be shown that

$$d_2 = \left[\frac{d_B - d_A}{t}\right] d_3 + d_A \qquad (3\text{-}28)$$

where d_A and d_B are as shown in the diagram. Substituting Equation 3-28 into Equation 3-27 and integrating produces the following result

$$I = \frac{I_0 \beta_{sc}^{(fog)}}{\alpha_{fog}} \left[\frac{t}{d_B - d_A + t}\right] e^{-\alpha_{air} d_1 - \alpha_{fog} d_A} \left[1 - e^{-(d_B - d_A + t)\alpha_{fog}}\right] \qquad (3\text{-}29)$$

The distances d_1, d_A and d_B are given by

$$\begin{aligned} d_1 &= (H_{atm} - H_{fog}) \csc\phi \\ d_A &= (H_{fog} - H_{eye}) \csc\phi \\ d_B &= (H_{fog} - H_{obj}) \csc\phi \end{aligned} \qquad (3\text{-}30)$$

This example illustrates just one special case of the fog component. The next section will summarise the results of similar calculations for all situations involving eye-fog-object positions.

3.4.3 Summary of the Fog Components

(a) *Eye and object above the fog*

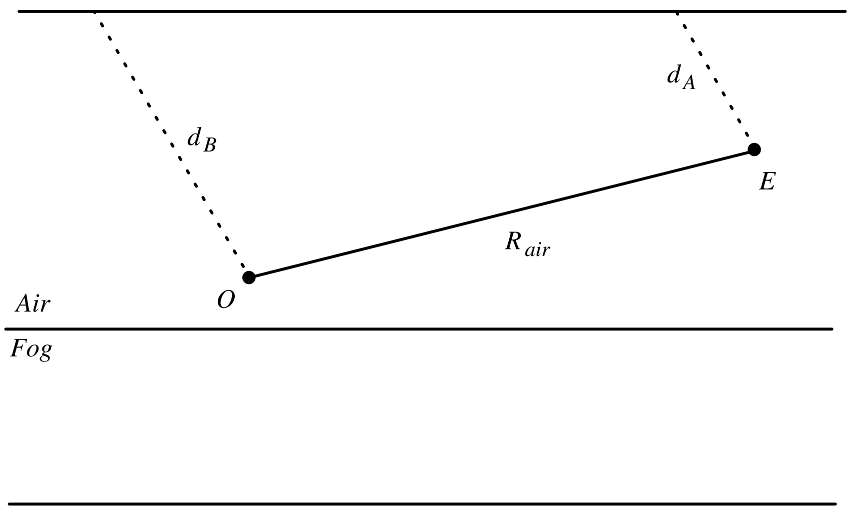


Figure 12: Eye and object above fog

$$I_1 = \frac{I_0 \beta_{sc}^{(air)}}{\alpha_{air}} \left[\frac{R_{air}}{d_B - d_A + R_{air}} \right] e^{-\alpha_{air} d_A} \left[1 - e^{-(d_B - d_A + R_{air})\alpha_{air}} \right] \tag{3-31}$$

where

$$\begin{aligned} d_A &= (H_{atm} - H_{eye}) \csc\phi \\ d_B &= (H_{atm} - H_{obj}) \csc\phi \end{aligned} \tag{3-32}$$

and ψ, ϕ are as described above. The distance R_{air} is the object intersection distance.

(b) *Eye and object in the fog*

This is the situation derived in detail earlier. The result will be recast in slightly different notation for consistency and future use.

$$I_2 = \frac{I_0 \beta_{sc}^{(fog)}}{\alpha_{fog}} \left[\frac{R_{fog}}{d_D - d_C + R_{fog}} \right] e^{-\alpha_{air} D - \alpha_{fog} d_C} \left[1 - e^{-(d_D - d_C + R_{fog})\alpha_{fog}} \right] \tag{3-33}$$

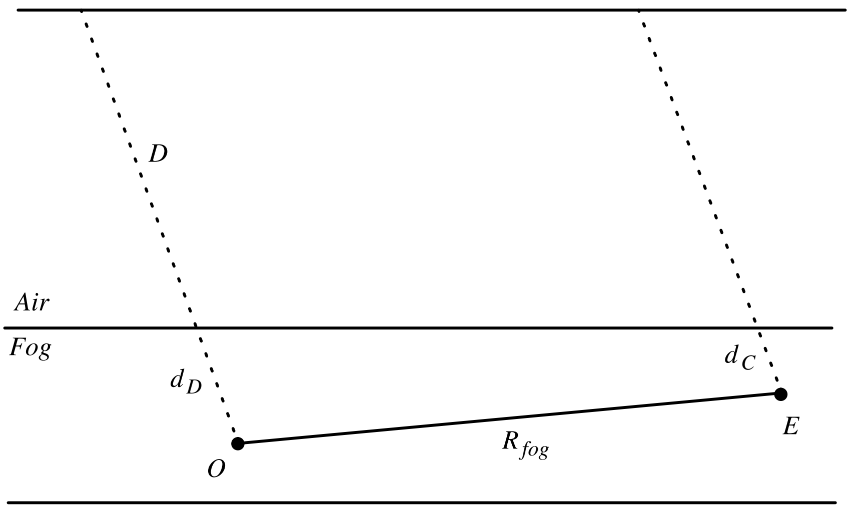


Figure 13: Eye and object in fog

where

$$
\begin{aligned}
d_C &= (H_{fog} - H_{eye}) \csc\phi \\
d_D &= (H_{fog} - H_{obj}) \csc\phi \\
D &= (H_{atm} - H_{fog}) \csc\phi
\end{aligned}
\qquad (3\text{-}34)
$$

The distance R_{fog} is again just the object intersection distance. The distance D is the thickness of air through which the light travels from the top of the atmosphere to the top of the fog. Here it is called the *sky depth* and is a constant for a given light source direction.

(c) *Eye above the fog, object in the fog*

This situation can be evaluated by using the previous results. The contribution from the air, I_{air}, is given by I_1 in Equation 3-31 by putting $d_A = d_1$ and $d_B = D$. The contribution from the fog, I_{fog}, is given by I_2 in Equation 3-33 by putting $d_D = d_2$ and $d_C = 0$, and then multiplying by the factor exp($-\alpha_{air} R_{air}$), representing the extra attenuation along the further distance to the eye. Hence

$$
I_{air} = \frac{I_0 \beta_{sc}^{(air)}}{\alpha_{air}} \left[\frac{R_{air}}{D - d_1 + R_{air}} \right] e^{-\alpha_{air} d_1} \left[1 - e^{-(D - d_1 + R_{air})\alpha_{air}} \right] \qquad (3\text{-}35)
$$

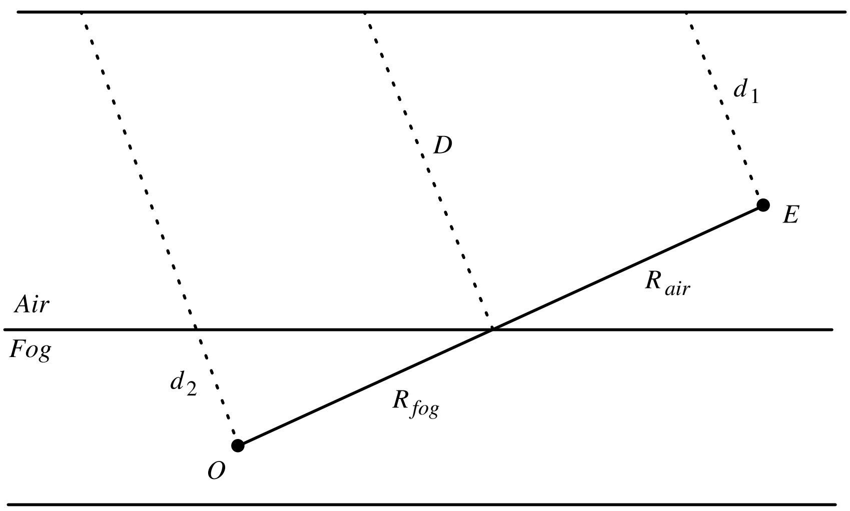


Figure 14: Eye above fog and object in fog

and

$$I_{fog} = \frac{I_0\beta_{sc}^{(fog)}}{\alpha_{fog}}\left[\frac{R_{fog}}{d_2 + R_{fog}}\right]e^{-(R_{air}+D)\alpha_{air}}\left[1 - e^{-(d_2 + R_{fog})\alpha_{fog}}\right] \tag{3-36}$$

The total fog component is then just

$$I_3 = I_{air} + I_{fog} \tag{3-37}$$

The distances d_1 and d_2 are given by d_A in Equation 3-32 and d_D in Equation 3-34 respectively. The distances R_{air} and R_{fog} are calculated in exactly the same manner as described in **Model1** .

(d) *Eye in the fog, object above the fog*

Again previous results come in useful. The contribution from the fog is given by I_2 in Equation 3-33 by putting $d_D = 0$ and $d_C = d_1$. The contribution from the air is given by I_1 in Equation 3-31 by putting $d_A = D$ and $d_B = d_2$. An extra attenuation factor of exp($-\alpha_{fog}R_{fog}$) is also applied to I_1. Then

$$I_{fog} = \frac{I_0\beta_{sc}^{(fog)}}{\alpha_{fog}}\left[\frac{R_{fog}}{R_{fog} - d_1}\right]e^{-\alpha_{air}D - \alpha_{fog}d_1}\left[1 - e^{-(R_{fog} - d_1)\alpha_{fog}}\right] \tag{3-38}$$

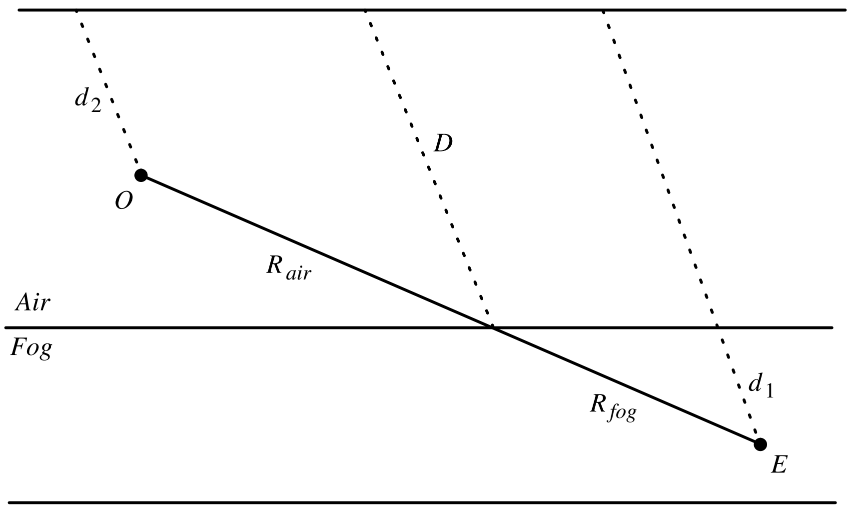


Figure 15: Eye in fog and object above fog

and

$$I_{air} = \frac{I_0 \beta_{sc}^{(air)}}{\alpha_{air}} \left[\frac{R_{air}}{d_2 - D + R_{air}}\right] e^{-\alpha_{air} D - \alpha_{fog} R_{fog}} \left[1 - e^{-(d_2 - D + R_{air})\alpha_{air}}\right] \tag{3-39}$$

The total fog component is given by

$$I_4 = I_{fog} + I_{air} \tag{3-40}$$

Note that I_{fog} in Equation 3-38 becomes undefined when $R_{fog} = d_1$. In this situation the following expression must be used

$$I_{fog} = I_0 \beta_{sc}^{(fog)} R_{fog} e^{-\alpha_{air} D - \alpha_{fog} d_1} \tag{3-41}$$

which is derived by calculating the limit of Equation 3-38 as (R_{fog} - d_1) $\rightarrow$ 0. The distances R_{air} and R_{fog} are determined as in **Model1** .

3.4.4 Summary of the Background Intensity Calculations

The results presented in previous sections can also be used to quickly derive expressions for the intensity reaching the eye when it is looking into the background. As in **Model1** two cases occur.

(a) *Eye above the fog*

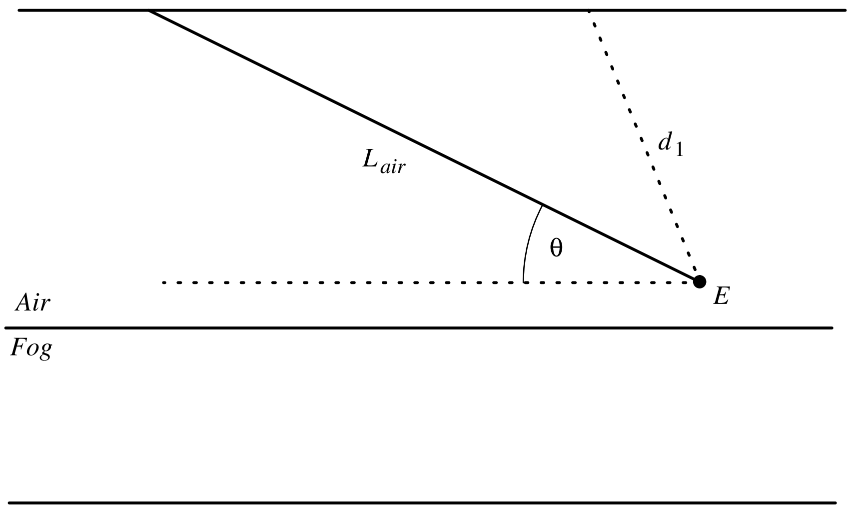


Figure 16: Eye above fog

Here the intensity of the background comes solely from scattering in the air and we can use Equation 3-31 with $d_A = d_1$, $d_B = 0$ and $R_{air} = L_{air}$, where L_{air} is the thickness of air between the eye and the top of the atmosphere along the line of sight. Therefore

$$I_5 = \frac{I_0 \beta_{sc}^{(air)}}{\alpha_{air}} \left[\frac{L_{air}}{L_{air} - d_1}\right] e^{-\alpha_{air} d_1} \left[1 - e^{-(L_{air} - d_1)\alpha_{air}}\right] \qquad (3\text{-}42)$$

and

$$L_{air} = (H_{atm} - H_{eye}) \csc\theta \qquad (3\text{-}43)$$

where θ is the angle the eye-ray makes with the horizontal. A singularity occurs in Equation 3-42 when $L_{air} = d_1$. To deal with this a limit is taken, giving

$$I_5 = I_0 \beta_{sc}^{(air)} L_{air} e^{-\alpha_{air} d_1} \qquad (3\text{-}44)$$

(b) *Eye in the fog*

The solution to this problem follows from I_4 in Equation 3-40 by putting $d_2 = 0$ and $R_{air} = L_{air}$. The contribution from the fog is identical and given by Equation 3-38.

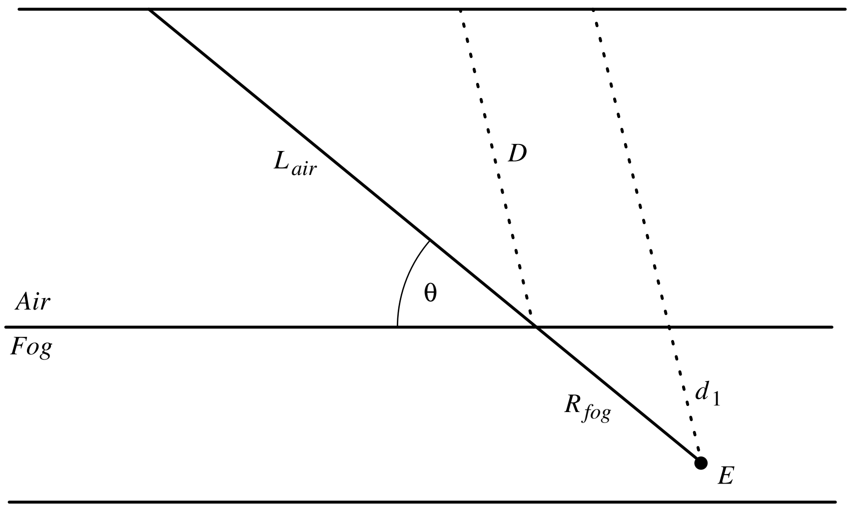


Figure 17: Eye in the fog

The contribution from the air becomes

$$\begin{aligned} I_{air} &= \frac{I_0 \beta_{sc}^{(air)}}{\alpha_{air}} \left[\frac{L_{air}}{L_{air} - D} \right] e^{-\alpha_{air} D - \alpha_{fog} R_{fog}} \left[1 - e^{-(L_{air} - D)\alpha_{air}} \right] \\ &= I_0 \beta_{sc}^{(air)} L_{air} e^{-\alpha_{air} D - \alpha_{fog} R_{fog}} \end{aligned} \tag{3-45}$$

when $L_{air} = D$. Finally,

$$I_6 = I_{fog} + I_{air} \tag{3-46}$$

3.5 Implementation

The implementation of both illumination models can be followed by looking at the source code (see file *Shade.c* in Appendix C). For **Model1** the relevant functions are *ShadePolygon1* and *ShadeBackground1* , whereas **Model2** uses *ShadePolygon2* and *ShadeBackground2* . Lack of time prevented an efficient implementation, so the equations describing the models are used directly as given with little optimisation.

A few points are necessary now, which relate to the geometry of the atmosphere in **Model2** and its approximation to the real terrestrial atmosphere. The models presented here are based on the theory of the Flat Earth - the air and/or fog is assumed to be uniform in density and extend to infinity in all horizontal directions. Nevertheless, we have attempted to re-introduce some features of the real atmosphere and one of these concerns the height of the atmosphere.

The default atmosphere height of 8.4km used by the renderer is equal to the *scale height* of the real atmosphere[33]. Assuming that the density of the real atmosphere falls exponentially with height above the ground, then a model atmosphere whose density is constant (equal to the real density at sea-level) and whose height is 8.4km, has approximately the same *optical depth* as the real atmosphere in most directions. Therefore, similar amounts of scattering and attenuation in the real atmosphere are expected to be found in the model atmosphere.

Another feature concerns the amount of atmosphere along the line of sight looking towards the horizon. In the model atmosphere this would tend to infinity the closer one looks to the horizon, resulting in unrealistic amounts of scattering and attenuation. To circumvent this, a minimum is set on the slope of the eye-ray. The equivalent range corresponds to the optical depth of the real (curved) atmosphere looking horizontally at the earth's surface. This is equal to about 290km. Hence for elevations below about 1.7° no colour intensity changes are expected to be seen.

4 Presentation of the Results

Calculations using the algorithms prescribed were conducted to produce rendered images of the airfield under various atmospheric conditions. The images which are presented here try to illustrate features of the model used to generate them. General views are given for each model. However, for the purposes of comparison, two series of images are made for each model under exactly the same conditions. As will be seen, each series is a collection of nine views in which the height of the fog is successively increased. The difference between the two sequences is the density of the fog. These images show up major differences between the models.

All single images were calculated at 768x576 pixels resolution with anti-aliasing at nine uniform samples per pixels. The series images are 250x185 pixels without shadows and without anti-aliasing. Unless otherwise stated, the view point is 3m above the ground. For the images produced by **Model2**, the visual range and height of the atmosphere (air) were fixed at their default values (see Table 2).

4.1 Model1 Calculations

Figure 18 was generated using **Model1** without any fog. It therefore gives a general idea of the structure of the polygonal model in the absence of any interfering atmosphere. The hills in the background are further away than one would think. The peak on the left is 4.5km distant. Also, the length of the runway is 500m, the tower on the left is 13m high at a distance of 250m, while the buildings on the right are 165m away.

The main point to make about this image, concerns the appearance of the sky background. The image clearly shows the effect of using a single colour to represent the background. No real sense of depth is given to the view, which may be the reason for mis-judging the distance of the hills.

The image in Figure 19 incorporates a layer of fog 30m high with a visual range of 1km. The eye is therefore inside the fog producing the graded effect in the colour of the sky. The fog is not dense enough to effect the visibility of much of the foreground. Figure 20 shows a view of dense fog extending to 8.4km, the maximum height, and most of the view except the near foreground is obscured.

Figure 21 is a different view looking out towards the horizon. The layer of fog is not very dense (more like haze) and is relatively low-lying. The height of the eye in this view is 30m above the ground. The view gives an idea of the effect of the fog along the horizon, which should be compared to the same view generated using **Model2** given later.

The sequences shown in Figure 22 and Figure 23 represent fog of different densities and heights. The heights range from 2m to 512m where each height is twice the previous one. The first sequence uses fog of 1km visual range and the results show a general trend from clear sky to low visibility. The second series has a visual range of 0.1km and shows that no further changes in the appearance of the fog occur once the height has reached about 30m.

Note that in all the images produces by **Model1**, no great diminution of the overall lighting occurs, even with thick fog. This is guaranteed since the background colours are normalised as in Equation 3-9. Very dense fog will then just produce a blanket of white over the whole image.

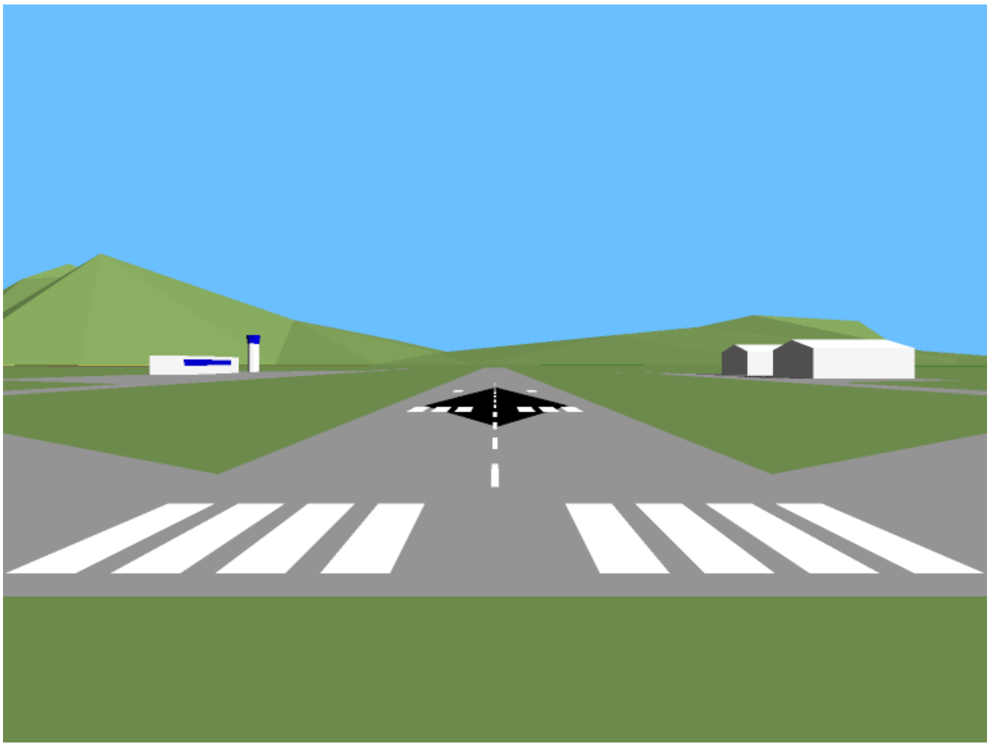


Figure 18: No fog rendered

Figure 19: V isual range 1km, height 30m

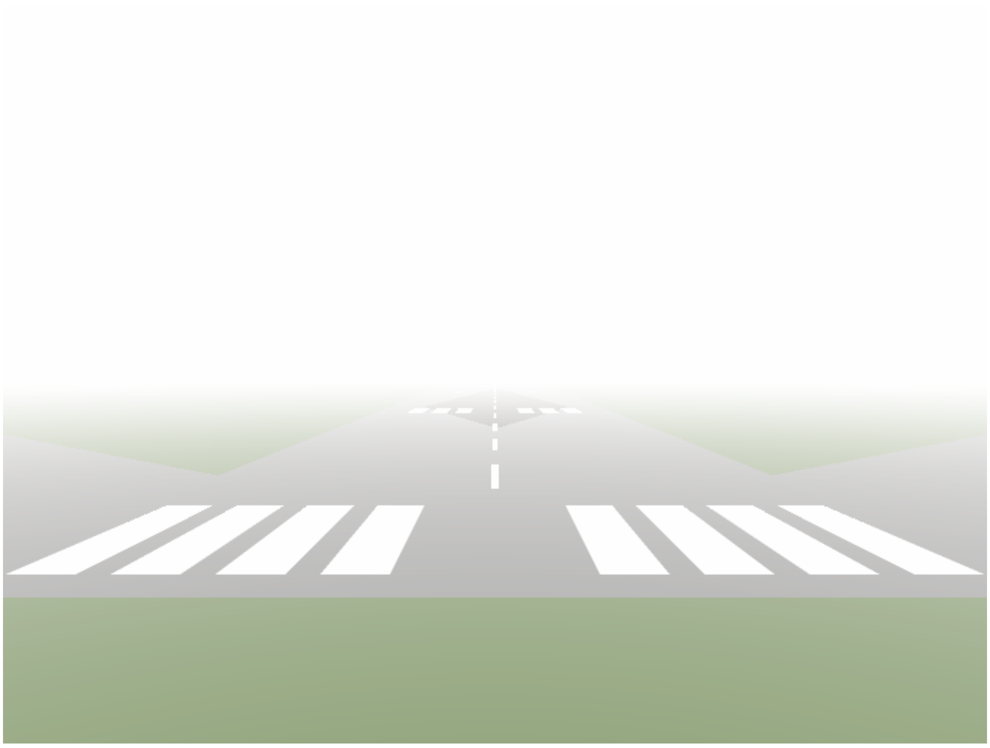


Figure 20: V isual range 0.1km, height 8.4km

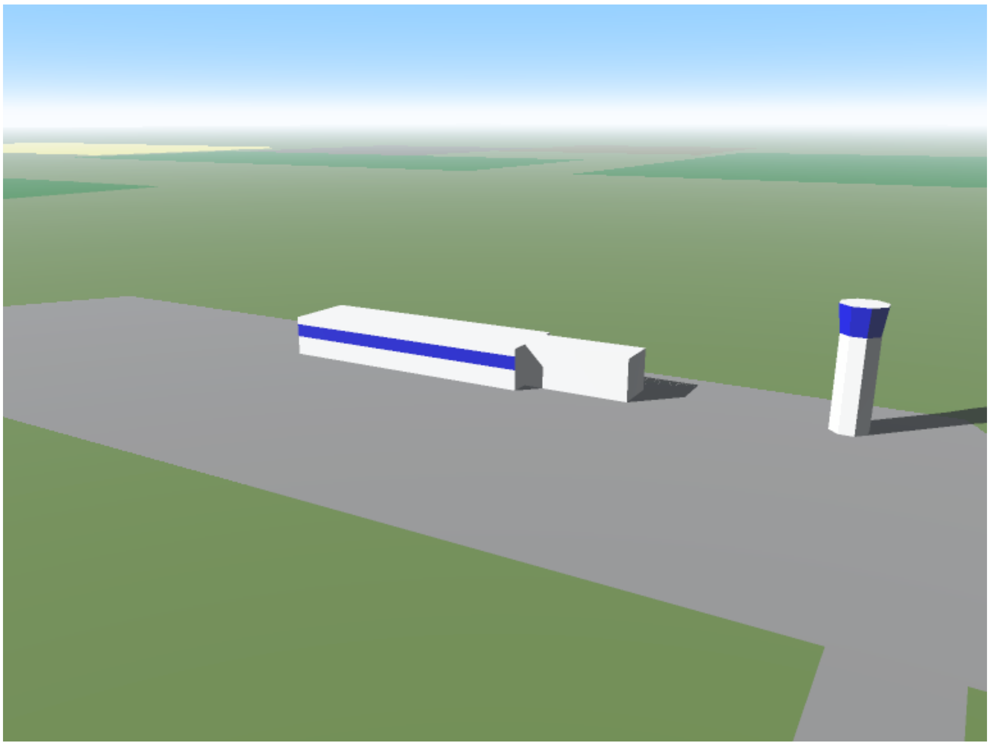


Figure 21: V isual range 5km, height 80m

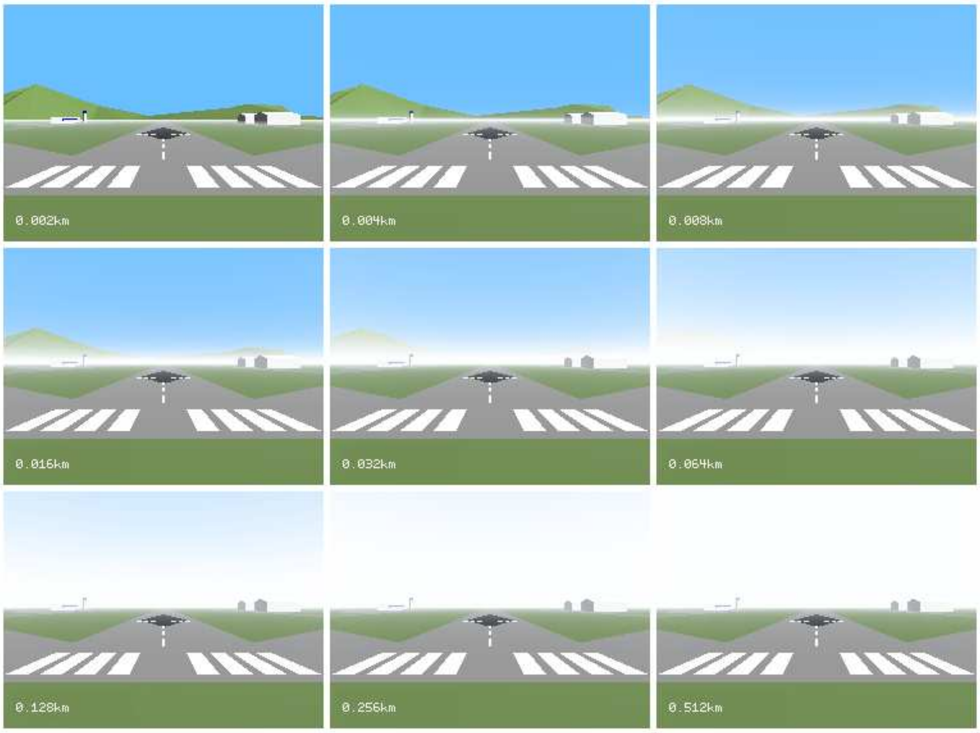


Figure 22: V isual range 1km

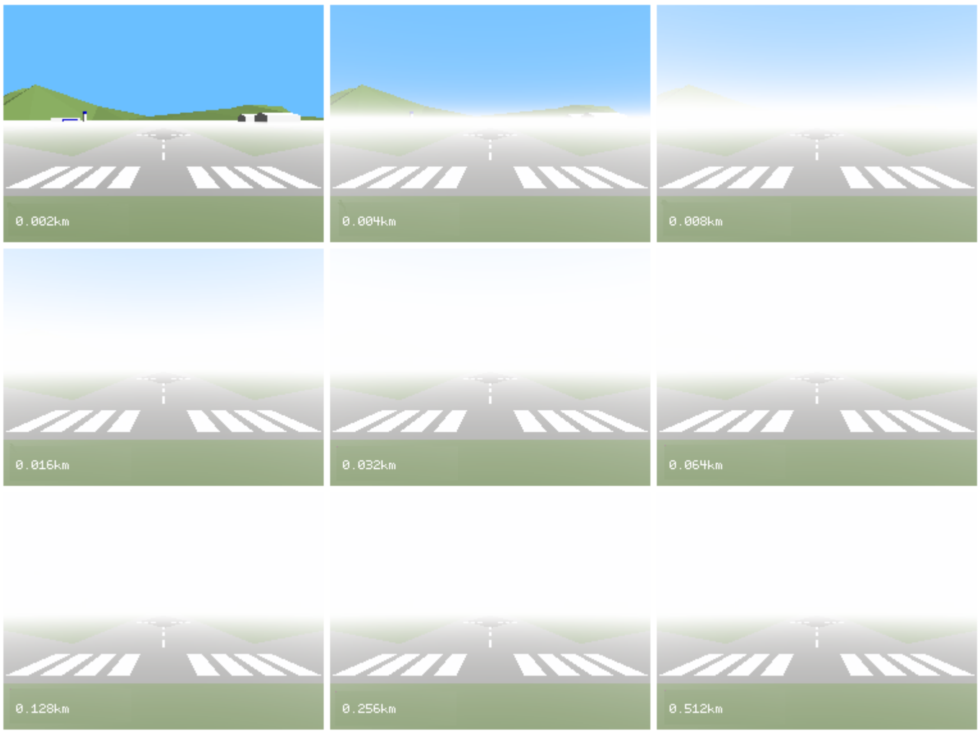


Figure 23: V isual range 0.1km

4.2 Model2 Calculations

Producing images based on **Model2** has created some interesting results. Consider the image in Figure 24. This view was calculated under the same conditions as that in Figure 18. No fog is included but the computations take into account the scattering due to the air. This has produced the blue colour of the sky and the gradual change to white towards the horizon. Note also the slight blue cast on the hills. A wide angle view (100°) is given in Figure 25. The zenith is approximately one quarter of the way down the image from the top.

An artifact can be seen in both of these views, particularly in the second. A low altitude, light coloured band is just detectable across the sky. It has been established that this effect is due to Mach banding and the altitude of the band is constant at the same altitude of the direct light source. The artifact is related to the singularity in Equation 3-42 mentioned earlier.

Figure 26 and Figure 27 show two images which have been rendered with fog. The first has low-lying fog in it, only 2m high, so the view point is just above the top. The other uses quite dense fog which is 1km thick. The hills are just beginning to be invisible.

The alternative views depicted inFigure 28 and Figure 29 focuses attention on to the horizon. The first is sky only, no fog, and shows more clearly the whitening effect close to the horizon. The second resulted from the same conditions used in Figure 21. First note the reddish tinge of the layer of fog above the horizon which indicates, to some extent, that excessive amounts of attenuation have been calculated, particularly in the air. Also note that the change in the colour of the background, moving towards and below the horizon, is not smooth. A change of intensity and colour is apparent at the horizon. This effect is thought to be created by the clamping of the background colour, below 1.7° above the horizon, to the colour at 1.7° (see Section 3.5). Hence there is a mismatch of colours calculated by one part of the algorithm and another. One part involves an object, the other doesn't. The problem really lies in the finite nature of the model scene and the plane-parallel approximation to the real atmosphere.

Finally, Figure 30 and Figure 31 show sequences produced under the same conditions as Figure 22 and Figure 23. The first sequence generally produces satisfactory results. The exception is the last one which has the greatest fog height. Large amounts of attenuation of light are present, so much so that the scene looks very dark. Notice how the buildings seem to glow in the darkness. Under such conditions the ambient illumination has become greater than that from direct light. Banding of the colours are also visible in the last image. This is an effect from the gamma correction on the low intensity regions.

The darkening effect is even more pronounced in the second sequence which includes denser fog. Even the intensity of the ambient light reflected from the buildings is attenuated below perceptible levels.

All these results bring into question the validity of the model, both the illumination model and the renderer. This will be discussed in the next, concluding section.

Figure 24: No fog rendered

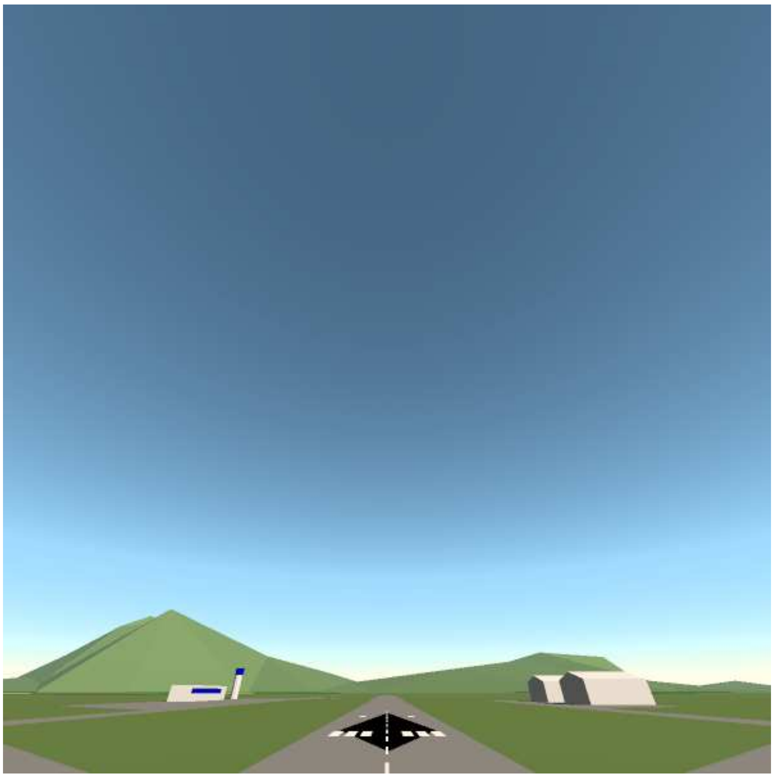


Figure 25: No fog rendered, wide-angle view

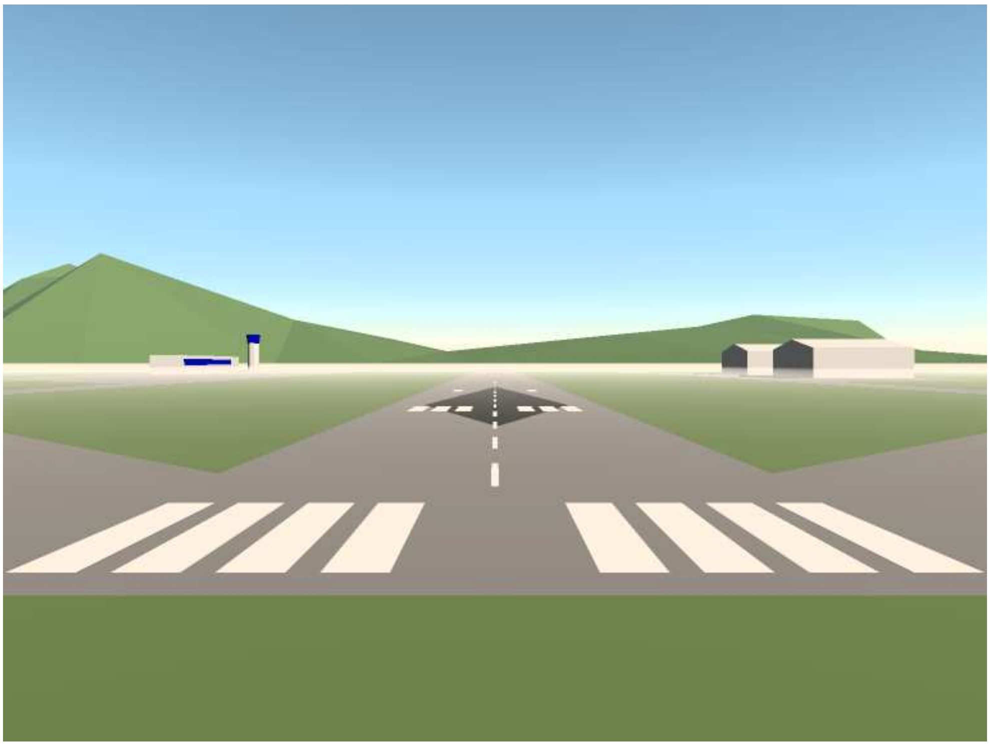


Figure 26: V isual range 1km, height 2m

Figure 27: V isual range 5km, height 1km

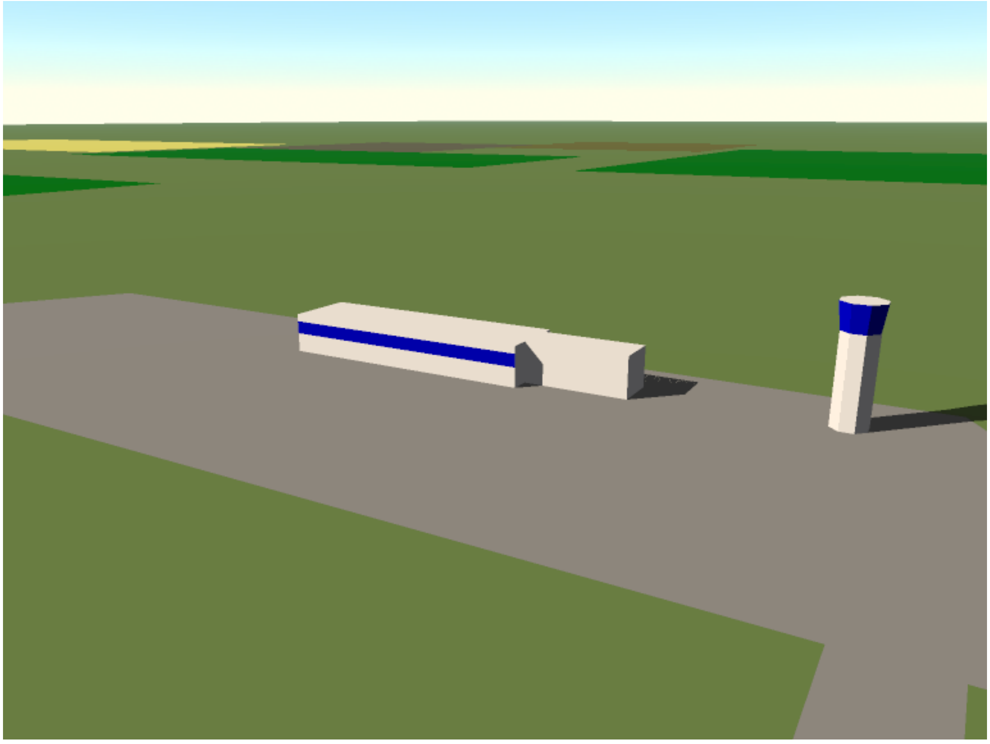


Figure 28: No fog rendered

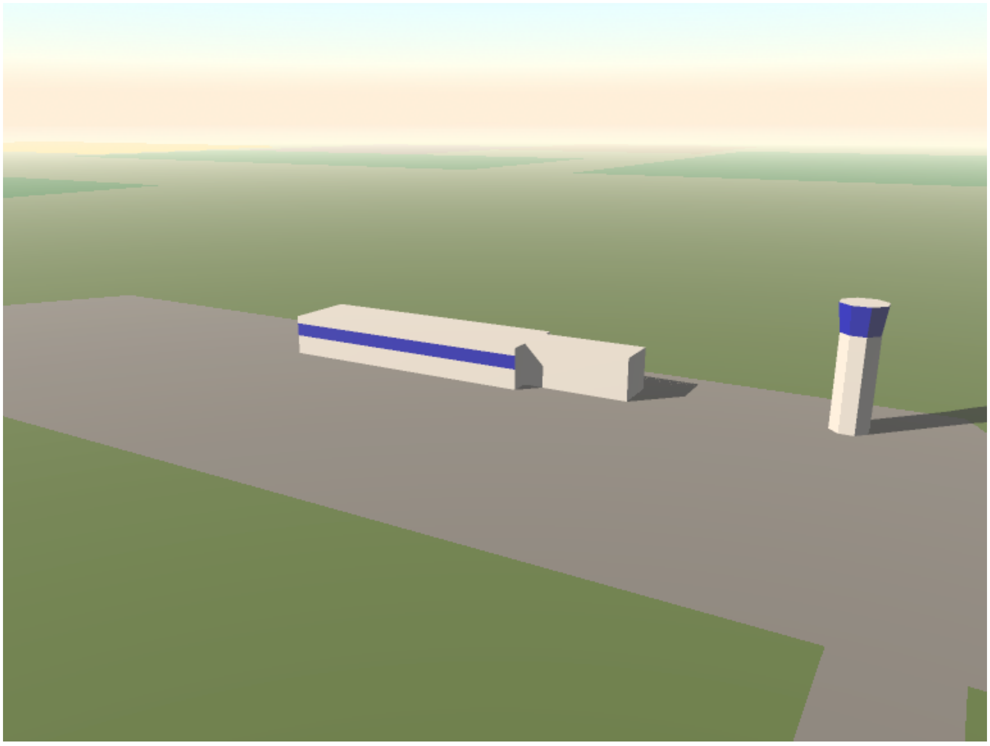


Figure 29: V isual range 5km, height 80m

Figure 30: Visual range 1km

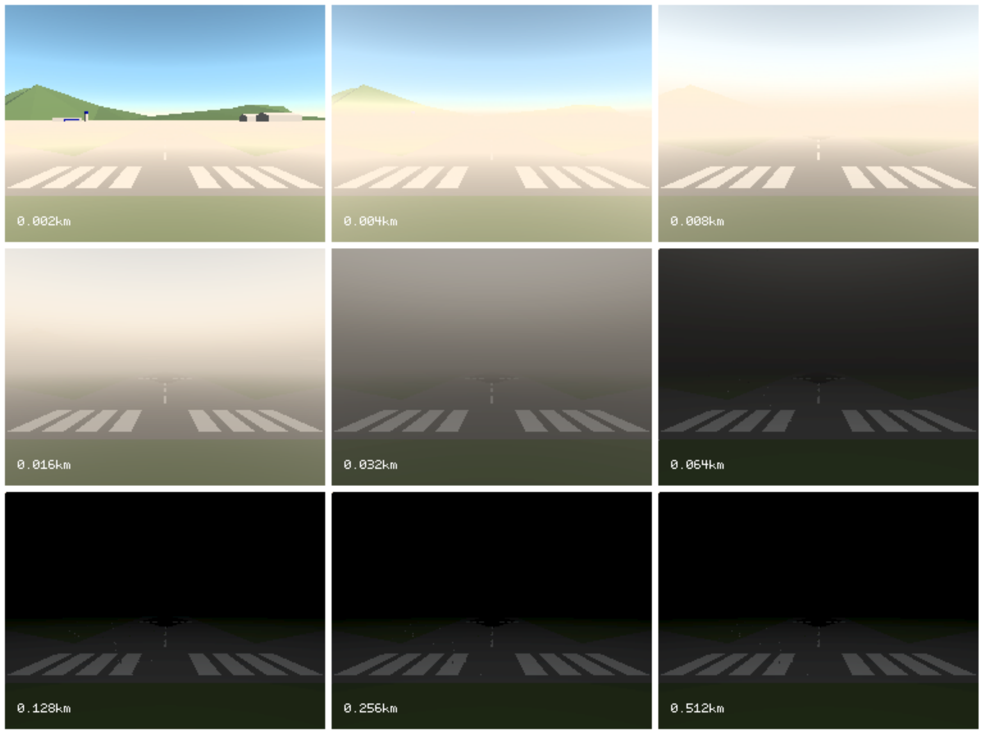


Figure 31: Visual range 0.1km

5 Conclusions and Recommendations

The project has succeeded in its initial objectives to implement an atmospheric illumination model. In fact two models have been established. The first was an extension of an existing model proposed by Willis and implemented by RWG. The second is new and based upon the methods suggested by Klassen.

The first model, **Model1**, is complete and could realistically be used by RWG, whereas the other, while producing some good results, is still incomplete and in need of some development to produce convincing images over the whole range of conditions experienced in the real atmosphere.

The results for **Model2** tend to fail when the optical depths involved become very large. This occurs if the height of the fog is very large, even for a relatively tenuous fog. The results are more appropriate to a thick dusty or sooty atmosphere than to fog. The effect could be helped by a more involved role for the ambient illumination, which at the moment is fixed. The ambient light could be considered as a variable source, dependent on fog density, with the effect of producing an illumination which mimics more closely the scattering in a real fog. This includes small angle scattering and multiple scattering.

Incorporating multiple scattering will retrieve some of the light not directly incident on an object, increasing the intensity at the surface and generally brightening the scene.

Note that the viewing system used by the renderer is analogous to a pin-hole or fixed-aperture camera. This is just the opposite to the system in the eyes which have apertures which open and close according to the overall brightness of the scene. In this way great ranges of intensity can be accommodated. This is not so with the fixed-aperture system. If the brightness falls very low, for example, then the view seen by this system will look much darker than that perceived by the eye. This effect may be present in the images produced here. Compensation is needed to account for it and the incorporation of a variable-aperture viewing system is one approach.

Nevertheless, the results produced of the clear sky are very effective. The colours tend to mimic those seen in the real sky but the problem of Mach banding remains and needs investigation. One approach would be to consider the geometry of the real, curved atmosphere. The incorporation of an air/fog density which falls exponentially with height is also recommended. Overall the results from **Model2** are satisfactory provided the fog is not too dense and/or too high.

A References

[1] Cook, R. L. and Torrance, K. E.
A reflectance model for computer graphics
Comp. Graph., **15** (3) (Aug. 1981) 307 - 316 (SIGGRAPH '81)

[2] Whitted, T.
A*n improved illumination model for shaded display*
Comm. ACM, **23** (6) (June 1980) 343 - 349

[3] Goral, C. M., Torrance, K. E., Greenberg, D.P. and Battaile, B.
Modeling the interaction of light between diffuse surfaces
Comp. Graph. **18** (3) (July 1984) 213 - 222 (SIGGRAPH '84)

[4] Glassner, A. S.
Simulating fog and haze
in Graphics Gems (Glassner, Ed.), Academic Press (London 1990) 364 - 365

[5] Blinn, J. F.
Light reflection functions for simulation of clouds and dusty surfaces
Comp. Graph. **16** (3) (July 1982) 21 - 29 (SIGGRAPH '82)

[6] Willis, P. J.
Visual simulation of atmospheric haze
Comp. Graph. Forum **6** (1987) 35 - 42

[7] Klassen, R. V.
Modeling the effect of the atmosphere on light
ACM Trans. Graph. **6** (3) (July 1987) 215 - 237

[8] Max, N. L.
Atmospheric illumination and shadows
Comp. Graph. **20** (4) (August 1986) 117 - 124 (SIGGRAPH '86)

[9] Nishita, T., Miyawaki, Y. and Nakamae, E.
A shading model for atmospheric scattering considering the luminous intensity distribution of light source s
Comp. Graph. **21** (4) (July 1987) 303 - 310 (SIGGRAPH '87)

[10] Kajiya, J. and Von Herzen, B. P.
Ray tracing volume densities
Comp. Graph. **18** (3) (July 1984) 165 - 173 (SIGGRAPH '84)

[11] Nishita, T. and Nakamae, E.
Continuous tone representation of three-dimensional objects illuminated by skylight
Comp. Graph. **20** (4) (August 1986) 125 (SIGGRAPH '86)

[12] Rushmeier, H. E. and Torrance, K.E.
The zonal method for calculating light intensities in the presence of a participating medium
Comp. Graph. **21** (4) (July 1987) 293-302 (SIGGRAPH '87)

[13] Glassner, A. S.
Ray tracing for realism
BYTE (December 1990) 263-271

[14] Badouel, D.
An efficient ray-polygon intersection
in Graphics Gems (Glassner, Ed.), Academic Press (London 1990) 390-393

[15] Glassner, A. S.
Space subdivision for fast ray-tracing
IEEE Comp. Graph. & Appl. **4** (10) (October 1984) 15-22

[16] Samet, H.
Implementing ray-tracing with octrees and neighbor finding
Comput. & Graph. **13** (4) (1989) 445-460

[17] Kaplan, M. R.
The use of spatial coherence in ray-tracing
in *Techniques for Computer Graphics*, Springer-Verlag (New York, 1987) 173-193

[18] Fujimoto, A., Tanaka, T. and Iwata, K.
ARTS: Accelerated ray-tracing system
IEEE Comp. Graph. & Appl. **6**(4) (April 1986) 16-26

[19] Woo, A.
Fast ray-box intersection
in Graphics Gems (Glassner, Ed.), Academic Press (London 1990) 395-396

[20] Haines, E.
Essential ray-tracing algorithms
in An Introduction to Ray Tracing (Glassner, Ed.) Academic Press (London 1989) 65-67

[21] Burger, P. and Gillies, D.
Interactive Computer Graphics, Addison-Wesley (Wokingham, 1989)

[22] Hall, R.
Illumination and Color in Computer Generated Imagery, Springer-Verlag (New York, 1989) 125-128

[23] Haines, E.
in An Introduction to Ray Tracing (Glassner, Ed.), Academic Press (London,1989) p.46

[24] Kolb, C. E.
Rayshade users guide and reference manual, Draft 0.4 (10 January 1992)
Available in electronic form with source code via anonymous FTP from INTERNET site princeton.edu

[25] Cook, R. L.
Stochastic sampling in computer graphics
ACM Trans. Graph. **5**(1) (January 1986) 51-72

[26] Cychosz, J. M.
Efficient generation of sampling jitter using look-up tables
in Graphics Gems (Glassner, Ed.), Academic Press (London 1990) 64-74

[27] Real World Graphics Ltd.
PC4000 graphics board: User's guide

[28] Sun Microsystems Ltd.
SunView programming manual
(Rev A January 1990)

[29] Sun Microsystems Ltd.
Pixrect reference manual
(Rev A 27 March 1990)

[30] Kondratyev, K. Ya.
Radiation in the atmosphere
International Geophysics Series Volume 12, Academic Press (1969) p.16

[31] Selley, G.
Personal communication.

[32] Chandrasekhar, S.
Radiative Transfer, Dover (New York 1960)

[33] Tennent, R. M.
Science Data Book, Longman (Edinburgh 1971) p.79

B Sample Input File

```
eyepoint  -15e-4 30e-4 100e-4
lookpoint -15e-4 0e-4 -2000e-4
vpd 1e-4
hfov 70
up 0 1 0
lightdir 1 1 1
lightintensity 2.0
ambientintensity 0.2
scale 0.0001

Polygon Colour :
  45 67 25

Polygon vertices
not cpv 10521.768500 0.000000 10204.059485
not cpv 10521.768500 0.000000 -10258.336487
not cpv -11026.166524 0.000000 -10258.336487
not cpv -11026.166524 0.000000 10204.059485

Polygon Colour :
  45 67 25

Polygon vertices
not cpv -11026.168524 0.000000 10204.059485
not cpv -11026.168524 0.000000 -10258.336487
not cpv -32574.103547 0.000000 -10258.336487
not cpv -32574.103547 0.000000 10204.059485

Polygon Colour :
  45 67 25

Polygon vertices
not cpv 10521.768500 0.000000 -10258.337487
not cpv 10521.768500 0.000000 -30720.733459
not cpv -11026.166524 0.000000 -30720.733459
not cpv -11026.166524 0.000000 -10258.337487


(... a further 375 polygon descriptions ommited)


Polygon Colour :
  252 253 253

Polygon vertices
not cpv -60.000003 0.000000 -4340.000206
not cpv -60.000003 0.000000 -4290.000204
not cpv -50.000002 0.000000 -4290.000204
not cpv -50.000002 0.000000 -4340.000206

Polygon Colour :
  252 253 253

Polygon vertices
not cpv 20.000001 0.000000 -4340.000206
not cpv 20.000001 0.000000 -4290.000204
not cpv 30.000001 0.000000 -4290.000204
not cpv 30.000001 0.000000 -4340.000206

end
```

C Main Program Code

List of files:

- Utility .h
- Render .h
- Bounds.c
- Coeffs.c
- Image.c
- Jitter .c
- Polygon.c
- Readdata.c
- Render .c
- Sampling.c
- Shade.c
- Vector .c

```
/*
 * Utility.h
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 26 May 1992
 *
 */

#include <malloc.h>

typedef int boolean;

#ifndef __sys_types_h
typedef unsigned char u_char;
typedef unsigned long int u_long;
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define MIN(x,y) (((x)<(y))?(x):(y))
#define MAX(x,y) (((x)>(y))?(x):(y))

/*
 * Allocate memory for an array of types (y elements of x)
 */
#define NEWTYPE(x,y) (x *)(malloc((unsigned)((y)*sizeof(x))))

#define EPSILON (double)1e-5
#define EQUAL(x,y) (fabs((x)-(y))<EPSILON)
#define FAR_AWAY 1.0e15

/*
 * Clamp the input(x) to the specified range(l to h)
 */
#define CLAMP(x,l,h) ((x)<(l) ? (l) : (x)>(h) ? (h) : x)
```

```
/*
 * Render.h
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 26 May 1992
 *
 */

typedef struct _point3 {
	double x, y, z;
} Point3;

typedef Point3 Vector3;

typedef struct _view {
	Point3 EyePos, LookPoint;
	Vector3 LookDir;
	Vector3 HVect, VVect;
	double VPD;
	double HFov;
	int ImageX, ImageY;
	double VisRange, FogHeight;
} View;

typedef struct _ray {
	Point3 Origin;
	Vector3 Dir;
} Ray;

typedef struct _RGBColour {
	double r, g, b;
} RGBColour;

typedef RGBColour ExtinctCoeff;
typedef RGBColour ScattCoeff;

typedef union _pixel {
	u_char RGB[4];
	u_long Colour;
} Pixel;

typedef struct _scanline {
	Pixel *Pixels;
	int *Sampled;
} ScanLine;

typedef struct _sphere {
	Point3 Centre;
	double Radius2;
} Sphere;

typedef struct _box {
	Point3 Low;
	Point3 High;
} Box;

typedef struct _polygon {
	u_char NVert;                 /* No. of vertices */
	u_char DomAxis;               /* Dominant axis of normal */
	double D;                     /* Plane constant */
	Vector3 Normal;               /* Polygon normal */
	Point3 *Vertices;             /* Array of vertex positions */
	RGBColour Colour;             /* Diffuse colour of polygon */
	boolean (*BVIntersect)();     /* Bounding volume intersection test */
	Sphere *BoundSphere;
	Box *BoundBox;
	struct _polygon *Next;
```

```
} Polygon;

typedef struct _vertex {
        Point3 Position;
        struct _vertex *Next;
} Vertex;

#define SPHERE 0
#define BOX1 1
#define BOX2 2
```

```
/*
 * Bounds.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 22 June 1992
 *
 */

#include <stdio.h>
#include "utility.h"
#include "render.h"

/*
 * This bounding sphere algorithm is based uponthe axis-aligned box whichencloses the polygon.
 * The centre of the sphere is then just the centroid of the box. The radius is given by the
 * maximum of the distances between the centre and the polygon vertices.
 * Ideally, the best sphere is determined by varying BOTH the centre and the radius to find the
 * minimum radius which encloses the polygon. A simpler approach is taken here.
 */
Sphere *GetBoundingSphere(Poly)
Polygon *Poly;
{
        Box *BVBox;
        Sphere *BVSphere;
        double Radius2;
        Vector3 Temp;
        int i;

        Box *GetBoundingBox();
        double V3SquaredLength();

        if( (BVSphere = NEWTYPE(Sphere,1)) == (Sphere *)NULL ) {
                fprintf(stderr, "\nMemory Full\n\n" );
                exit(1);
        }

        /*
         * First get the box which surrounds the polygon
         */
        BVBox = GetBoundingBox(Poly);

        BVSphere->Centre.x = 0.5 * (BVBox->High.x + BVBox->Low.x );
        BVSphere->Centre.y = 0.5 * (BVBox->High.y + BVBox->Low.y );
        BVSphere->Centre.z = 0.5 * (BVBox->High.z + BVBox->Low.z );

        V3Sub(&Poly->Vertices[0], &BVSphere->Centre, &Temp );
        BVSphere->Radius2 =V3SquaredLength(&Temp);

        for(i=1; i<Poly->NVert; i++ ) {
                V3Sub(&Poly->Vertices[i], &BVSphere->Centre, &Temp );
                Radius2 = V3SquaredLength(&Temp);
                if( Radius2 > BVSphere->Radius2 )
                        BVSphere->Radius2 = Radius2;
        }

        free(BVBox);

        return(BVSphere);
}



Box *GetBoundingBox(Poly)
Polygon *Poly;
{
        Box *BVBox;
        int i;
```

```
        if( (BVBox = NEWTYPE(Box, 1)) == (Box *)NULL ) {
                fprintf(stderr, "\nMemory Full\n\n" );
                exit(1);
        }

        BVBox->High = BVBox->Low = Poly->Vertices[0];

        for(i=1; i<Poly->NVert; i++ ) {

                if( Poly->Vertices[i].x < BVBox->Low.x )
                        BVBox->Low.x = Poly->Vertices[i].x;
                if( Poly->Vertices[i].x > BVBox->High.x )
                        BVBox->High.x = Poly->Vertices[i].x;

                if( Poly->Vertices[i].y < BVBox->Low.y )
                        BVBox->Low.y = Poly->Vertices[i].y;
                if( Poly->Vertices[i].y > BVBox->High.y )
                        BVBox->High.y = Poly->Vertices[i].y;

                if( Poly->Vertices[i].z < BVBox->Low.z )
                        BVBox->Low.z = Poly->Vertices[i].z;
                if( Poly->Vertices[i].z > BVBox->High.z )
                        BVBox->High.z = Poly->Vertices[i].z;

        }

        if( BVBox->Low.x == BVBox->High.x ) {
                BVBox->Low.x -= EPSILON;
                BVBox->High.x += EPSILON;
        }
        if( BVBox->Low.y == BVBox->High.y ) {
                BVBox->Low.y -= EPSILON;
                BVBox->High.y += EPSILON;
        }
        if( BVBox->Low.z == BVBox->High.z ) {
                BVBox->Low.z -= EPSILON;
                BVBox->High.z += EPSILON;
        }

        return(BVBox);
}


boolean BVSphereIntersect(Poly, ARay)
Polygon *Poly;
Ray *ARay;
{
        double Dist2, t;
        Vector3 Temp;

        double V3Dot(), V3SqauaredLength();

        /*
         * First test.
         * Is ray origin inside sphere? If it is ray must intersect sphere.
         * If not go on to next test.
         */
        V3Sub(&Poly->BoundSphere->Centre, &ARay->Origin, &Temp );

        if( (Dist2 = V3SquaredLength(&Temp)) < Poly->BoundSphere->Radius2 )
                return(TRUE);

        /*
         * Second test.
         * Calculate value of ray parameter t for point of closest approach to sphere centre.
         * If t<0 the sphere is behind the ray direction, so no intersection of interest.
         */
        if( (t = V3Dot(&Temp, &ARay->Dir)) < 0.0 )
```

```
                return(FALSE);

        /*
         * Third test.
         * Calculate the distance of closest approach of ray to sphere centre. If less than
         * the radius then intersection takes place. Otherwise ray misses.
         */
        if( Dist2 - t*t < Poly->BoundSphere->Radius2 )
                return(TRUE);
        else
                return(FALSE);

}


/*
 * Bounding box intersection test based upon the algorithm written by
 * Andrew Woo in "Graphics Gems"
 */
#define LOWER 0
#define UPPER 1
#define MIDDLE 2

boolean BVBoxIntersect1(Poly, ARay)
Polygon *Poly;
Ray *ARay;
{
        boolean Inside = TRUE;
        int Section[3], WhichPlane, i;
        double CandidatePlanes[3], MaxT[3], Coord;

        /*
         * First find candidate planes - ones which the ray can possibly intersect with first
         */
        if( ARay->Origin.x < Poly->BoundBox->Low.x ) {
                Section[0] = LOWER;
                CandidatePlanes[0] = Poly->BoundBox->Low.x;
                Inside = FALSE;
        }
        else if( ARay->Origin.x > Poly->BoundBox->High.x ) {
                Section[0] = UPPER;
                CandidatePlanes[0] = Poly->BoundBox->High.x;
                Inside = FALSE;
        }
        else
                Section[0] = MIDDLE;

        if( ARay->Origin.y < Poly->BoundBox->Low.y ) {
                Section[1] = LOWER;
                CandidatePlanes[1] = Poly->BoundBox->Low.y;
                Inside = FALSE;
        }
        else if( ARay->Origin.y > Poly->BoundBox->High.y ) {
                Section[1] = UPPER;
                CandidatePlanes[1] = Poly->BoundBox->High.y;
                Inside = FALSE;
        }
        else
                Section[1] = MIDDLE;

        if( ARay->Origin.z < Poly->BoundBox->Low.z ) {
                Section[2] = LOWER;
                CandidatePlanes[2] = Poly->BoundBox->Low.z;
                Inside = FALSE;
        }
        else if( ARay->Origin.z > Poly->BoundBox->High.z ) {
                Section[2] = UPPER;
                CandidatePlanes[2] = Poly->BoundBox->High.z;
                Inside = FALSE;
```

```
        }
        else
                Section[2] = MIDDLE;

        /*
         * If ray origin is inside the box we have an intersection
         */
        if( Inside )
                return(TRUE);

        /*
         * Calculate t distances to candidate planes
         */
        if( Section[0] != MIDDLE && ARay->Dir.x != 0.0 )
                MaxT[0] = ( CandidatePlanes[0]-ARay->Origin.x )/ARay->Dir.x;
        else
                MaxT[0] = -1.0;

        if( Section[1] != MIDDLE && ARay->Dir.y != 0.0 )
                MaxT[1] = ( CandidatePlanes[1]-ARay->Origin.y )/ARay->Dir.y;
        else
                MaxT[1] = -1.0;

        if( Section[2] != MIDDLE && ARay->Dir.z != 0.0 )
                MaxT[2] = ( CandidatePlanes[2]-ARay->Origin.z )/ARay->Dir.z;
        else
                MaxT[2] = -1.0;

        /*
         * Get the plane identity with the largest MaxT
         */
        WhichPlane = 0;
        for(i=1; i<3; i++ )
                if( MaxT[WhichPlane] < MaxT[i] )
                    WhichPlane = i;

        /*
         * Finally check candidate intersection is actually inside the box
         */
        if( MaxT[WhichPlane] < 0.0 )
                return(FALSE);

        if( WhichPlane != 0 ) {
                Coord = ARay->Origin.x + MaxT[WhichPlane]*ARay->Dir.x;
                if( (Section[0] == UPPER && Coord < Poly->BoundBox->Low.x) ||
                        (Section[0] == LOWER && Coord > Poly->BoundBox->High.x ) )
                                return(FALSE);
        }

        if( WhichPlane != 1 ) {
                Coord = ARay->Origin.y + MaxT[WhichPlane]*ARay->Dir.y;
                if( (Section[1] == UPPER && Coord < Poly->BoundBox->Low.y) ||
                        (Section[1] == LOWER && Coord > Poly->BoundBox->High.y ) )
                                return(FALSE);
        }

        if( WhichPlane != 2 ) {
                Coord = ARay->Origin.z + MaxT[WhichPlane]*ARay->Dir.z;
                if( (Section[2] == UPPER && Coord < Poly->BoundBox->Low.z) ||
                        (Section[2] == LOWER && Coord > Poly->BoundBox->High.z ) )
                                return(FALSE);
        }

        return(TRUE);
}


/*
 * Bounding box intersection test based upon the algorithm written by
```

```
 * Eric Haines in "An Introduction to Ray Tracing"
 */
boolean BVBoxIntersect2(Poly, ARay)
Polygon *Poly;
Ray *ARay;
{
        double TNear = -FAR_AWAY, TFar = FAR_AWAY;
        double T1, T2;

        boolean TestTs();

        /*
         * Each pair of planes (X,Y and Z) associated with the box are tested in turn with the ray.
         * If the ray direction is parallel to the planes and the ray origin is not between them,
         * no intersection is possible with the box. Otherwise the intersection distances with
         * the planes are calculated and compared to decide whether an intersection is possible.
         */

        if( ARay->Dir.x == 0.0 )
                if( ARay->Origin.x < Poly->BoundBox->Low.x ||
                         ARay->Origin.x > Poly->BoundBox->High.x )
                                return(FALSE);
        else {
                T1 = (Poly->BoundBox->Low.x - ARay->Origin.x)/ARay->Dir.x;
                T2 = (Poly->BoundBox->High.x - ARay->Origin.x)/ARay->Dir.x;

                if( TestTs(T1, T2, &TNear, &TFar) )
                        return(FALSE);
        }

        if( ARay->Dir.y == 0.0 )
                if( ARay->Origin.y < Poly->BoundBox->Low.y ||
                        ARay->Origin.y > Poly->BoundBox->High.y )
                                return(FALSE);
        else {
                T1 = (Poly->BoundBox->Low.y - ARay->Origin.y)/ARay->Dir.y;
                T2 = (Poly->BoundBox->High.y - ARay->Origin.y)/ARay->Dir.y;

                if( TestTs(T1, T2, &TNear, &TFar) )
                        return(FALSE);
        }

        if( ARay->Dir.x == 0.0 )
                if( ARay->Origin.z < Poly->BoundBox->Low.z ||
                        ARay->Origin.z > Poly->BoundBox->High.z )
                                return(FALSE);
        else {
                T1 = (Poly->BoundBox->Low.z - ARay->Origin.z)/ARay->Dir.z;
                T2 = (Poly->BoundBox->High.z - ARay->Origin.z)/ARay->Dir.z;

                if( TestTs(T1, T2, &TNear, &TFar) )
                        return(FALSE);
        }

        return(TRUE);
}


boolean TestTs(T1, T2, Near, Far)
double T1, T2;
double *Near, *Far;
{
        double Temp;

        if( T1 > T2 ) {
                Temp = T1;
                T1 = T2;
                T2 = Temp;
        }
```

```
        if( T1 > *Near )
                *Near = T1;

        if( T2 < *Far )
                *Far = T2;

        if( *Near > *Far || *Far < 0.0 )
                return(TRUE);
        else
                return(FALSE);

}
```

```
/*
 * Coeff.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 3 July 1992
 *
 */

#include <math.h>
#include "utility.h"
#include "render.h"

#define KOSH 3.91

/*
 * The following table contains values of the red, green and blue relative extinction
 * coefficients as a function of the log(base 10) of the visual range. The values were obtained
 * by digitising the graphs given by Foitzik. The 27 values cover the range -0.2 to 2.4 inclusive
 * at 0.1 intervals. This corresponds to 0.631 to 251km. The coefficients for values less than
 * -0.2 are virtually constant and equal to those at -0.2. Coefficients above 2.4 can also be
 * considered constant and equal to those at 2.4.
 */
static float RelExtinctCoeff[27][3] = {
      {  1.014343,  0.996931,  0.978681 },
      {  0.959551,  0.988648,  1.031693 },
      {  0.931300,  0.986694,  1.068148 },
      {  0.915352,  0.985973,  1.085297 },
      {  0.907783,  0.985360,  1.088763 },
      {  0.904342,  0.984712,  1.094903 },
      {  0.902155,  0.985474,  1.096992 },
      {  0.901386,  0.986094,  1.100342 },
      {  0.896610,  0.988233,  1.102432 },
      {  0.891792,  0.991748,  1.107235 },
      {  0.885599,  0.993852,  1.110739 },
      {  0.879244,  1.000084,  1.115503 },
      {  0.868922,  1.002152,  1.120228 },
      {  0.857063,  1.004255,  1.129160 },
      {  0.846701,  1.007734,  1.136716 },
      {  0.834963,  1.009802,  1.148438 },
      {  0.820472,  1.009189,  1.158745 },
      {  0.804605,  1.008540,  1.171805 },
      {  0.785906,  1.007891,  1.189033 },
      {  0.763159,  1.005866,  1.211686 },
      {  0.741830,  0.998263,  1.235795 },
      {  0.714914,  0.992146,  1.264031 },
      {  0.685246,  0.988708,  1.310079 },
      {  0.651489,  0.979801,  1.365839 },
      {  0.602516,  0.969519,  1.425648 },
      {  0.545325,  0.959272,  1.506295 },
      {  0.450901,  0.941964,  1.605935 },
};


/*
* The table of relative extinction coefficients given above are used to calculate the absolute
* values for the given visual range. The coefficients for visual ranges not specified in the
* table are obtained by linear interpolation . The absolute coefficients are calculated by
* multiplying the relative values by the generalised value given by Koshmeider's equation, which
* is a = 3.91/VR.
*/
void GetCoeff(VisRange, ExtCoeff, ScatCoeff)
double VisRange;
ExtinctCoeff *ExtCoeff;
ScattCoeff *ScatCoeff;
{
        double LogVisRange, IndexVR;
        int Index;
```

```
        LogVisRange = log10(VisRange);
        LogVisRange = CLAMP(LogVisRange, -0.2, 2.4);

        Index = (int)((LogVisRange+0.2)*10.0);

        IndexVR = Index*0.1 - 0.2;

        ExtCoeff->r = ((RelExtinctCoeff[Index+1][0] -
                       RelExtinctCoeff[Index][0])*10.0)*(LogVisRange-IndexVR)
                              + RelExtinctCoeff[Index][0];

        ExtCoeff->g = ((RelExtinctCoeff[Index+1][1] -
                       RelExtinctCoeff[Index][1])*10.0)*(LogVisRange-IndexVR)
                              + RelExtinctCoeff[Index][1];

        ExtCoeff->b = ((RelExtinctCoeff[Index+1][2] -
                       RelExtinctCoeff[Index][2])*10.0)*(LogVisRange-IndexVR)
                              + RelExtinctCoeff[Index][2];

        /*ScatCoeff->r = ExtCoeff->r/ExtCoeff->b;
        ScatCoeff->g = ExtCoeff->g/ExtCoeff->b;
        ScatCoeff->b = 1.0;*/

        ExtCoeff->r = ExtCoeff->r * KOSH / VisRange;
        ExtCoeff->g = ExtCoeff->g * KOSH / VisRange;
        ExtCoeff->b = ExtCoeff->b * KOSH / VisRange;

        ScatCoeff->r = ExtCoeff->r;
        ScatCoeff->g = ExtCoeff->g;
        ScatCoeff->b = ExtCoeff->b;
}


RGBColour GetBackground(VisRange)
double VisRange;
{
        RGBColour Colour;

        /*
         * Sky background colour calculation originated by RWG
         */
        Colour.r = 1.0/(2.0*exp(VisRange/250.0)-1.0);
        Colour.g = 1.0/(2.0-exp(-VisRange/250.0));
        Colour.b = 1.0;

        return(Colour);

}
```

```
/*
 * Image.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 1 July 1992
 *
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include "utility.h"
#include "render.h"

extern int ImageX, ImageY;
extern boolean ImageToFile, ImageToWindow;
extern View *View1;

static Frame BaseFrame;
static Pixwin *ImagePw;
static Pixrect *ImagePr;

void OutputInit()
{
	if( ImageToFile && !ImageToWindow )
		/*
		 * Create a memory pixrect to hold the image as it is rendered
		 */
		if( !(ImagePr = mem_create(ImageX, ImageY, 32)) ) {
			fprintf(stderr, "Cannot create image pixrect\n\n" );
			exit(1);
		}

	if( ImageToWindow ) {
		/*
		 * Create and display the output window
		 */
		Canvas MainCanvas;
		static struct singlecolor Background = {128,128,128};
		static struct singlecolor Foreground = {0,0,0};

		BaseFrame = window_create(NULL, FRAME,
			FRAME_LABEL, "Render",
			FRAME_NO_CONFIRM, TRUE,
			FRAME_BACKGROUND_COLOR, Background,
			FRAME_FOREGROUND_COLOR, Foreground,
			0 );

		MainCanvas  = window_create( BaseFrame, CANVAS,
			CANVAS_COLOR24, TRUE,
			CANVAS_AUTO_EXPAND, FALSE,
			CANVAS_AUTO_SHRINK, FALSE,
			CANVAS_WIDTH, ImageX,
			CANVAS_HEIGHT, ImageY,
			WIN_WIDTH, ImageX,
			WIN_HEIGHT, ImageY,
			0 );

		if( !MainCanvas ) {
			fprintf(stderr, "Cannot open 24-bit canvas on this machine\n\n" );
			exit(1);
		}

		ImagePw = canvas_pixwin(MainCanvas);

		window_fit(BaseFrame);
		window_set(BaseFrame, WIN_SHOW, TRUE, 0 );
```

```
                (void)notify_dispatch();
        }

}


void OutputScanLine(Scan, Line)
Pixel *Scan;
int Line;
{
        /*
         * Create and write scan line to the image pixrect and/or image window
         */
        Pixrect *ImageScanLine;

        ImageScanLine = mem_point(ImageX, 1, 32, Scan );

        if( ImageToWindow ) {
                pw_rop(ImagePw, 0, ImageY-Line-1, ImageX, 1, PIX_SRC, ImageScanLine, 0, 0 );
                (void)notify_dispatch();
        }

        if( ImageToFile && !ImageToWindow )
                pr_rop(ImagePr, 0, ImageY-Line-1, ImageX, 1, PIX_SRC, ImageScanLine, 0, 0 );

        /*
         * Free the temp pixrect - do not use pr_close, this frees Scan memory too!
         */
        free(ImageScanLine);

}


void DumpImage(ImageFileName)
char *ImageFileName;
{
        FILE *RasterFile;
        char ImageFile[128];

        strcpy(ImageFile, "compress >"  );
        strcat(ImageFile, ImageFileName );
        strcat(ImageFile, ".Z" );

        if( !(RasterFile = popen(ImageFile, "w")) ) {
                fprintf(stderr, "Cannot open rasterfile %s\n\n", ImageFileName );
                exit(1);
        }

        if( ImageToWindow ) {
                ImagePr = mem_create(ImageX, ImageY, 32 );
                pw_read(ImagePr, 0, 0, ImageX, ImageY, PIX_SRC, ImagePw, 0, 0 );
        }

        pr_dump(ImagePr, RasterFile, (colormap_t *)0, RT_STANDARD, 1);

        if( fwrite(View1, sizeof(View), 1, RasterFile) != 1 ) {
                fprintf(stderr, "Error writing trailer to rasterfile\n\n" );
                fclose(RasterFile);
                exit(1);
        }

        pr_close(ImagePr);
        pclose(RasterFile);

}


void ReleaseWindow()
{
```

```
        /*if( fork() == 0 )*/
                /*
                 * This is the child - continue to support the display window
                 */
                window_main_loop(BaseFrame);

        /*
         * Otherwise this is the parent - return and exit
         */
}
```

```
/*
 * Jitter.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 17 June 1992
 *
 */

 /*
  * Contains the routines for generation of sampling jitter using look-up tables.
  * Taken from "Graphics Gems", written by Joseph M. CychosZ.
  */

#define NRAN 1024

static double URANX[NRAN], URANY[NRAN];
static int IRAND[NRAN];
static int MASK = NRAN-1;

void Jitter1(x, y, s, xj, yj)
int x, y, s;
double *xj, *yj;
{
        *xj = URANX[ (x+(y<<2)+IRAND[(x+s)&MASK]) & MASK ];

        *yj = URANY[ (y+(x<<2)+IRAND[(y+s)&MASK]) & MASK ];
}


void Jitter2(x, y, s, xj, yj)
int x, y, s;
double *xj, *yj;
{
        *xj = URANX[ ((x|(y<<2))+IRAND[(x+s)&MASK]) & MASK ];

        *yj = URANY[ ((y|(x<<2))+IRAND[(y+s)&MASK]) & MASK ];
}


void JitterInit()
{
        int i;
        extern double drand48();

        /*
         * drand48() generates uniform random numbers in the range 0 to 1.
         * Note that seeding (srand48()) has not been used.
         */

        for(i=0; i<NRAN; i++ )
                URANX[i] = drand48();

        for(i=0; i<NRAN; i++ )
                URANY[i] = drand48();

        for(i=0; i<NRAN; i++ )
                IRAND[i] = (int)(NRAN*drand48());

}
```

```
/*
 * Polygon.c
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 26 May 1992
 *
 */

#include <stdio.h>
#include <math.h>
#include "utility.h"
#include "render.h"

extern int BoundingVolume;
extern boolean BVSphereIntersect(), BVBoxIntersect1(), BVBoxIntersect2();

/*
 * Create a new vertex structure and add it to the current vertex list
 */
void VertexCreate(VList, x, y, z)
Vertex **VList;
double x, y, z;
{
        Vertex *VTemp;
        static Vertex *LastVertex;

        /*
         * Allocate memory for the new vertex and copy the coordinates
         */
        if( (VTemp = NEWTYPE(Vertex,1)) != (Vertex *)NULL ) {
                VTemp->Position.x = x;
                VTemp->Position.y = y;
                VTemp->Position.z = z;
                VTemp->Next = (Vertex *)NULL;
        }
        else {
                fprintf(stderr, "Memory Full\n" );
                exit(1);
        }

        /*
         * Stick the completed vertex onto the list
         */
        if( *VList == (Vertex *)NULL )/* List is empty */
                *VList = VTemp;
        else
                LastVertex->Next = VTemp;

        LastVertex = VTemp;
}



/*
 * Create a polygon structure and add it to the polygon list.
 * The vertex list is freed as the data is transferred.
 */
void PolygonCreate(PList, VList, VCount, r, g, b)
Polygon **PList;
Vertex **VList;
int VCount;
unsigned r, g, b;
{
        int i;
        Polygon *PTemp;
        Vertex *VTemp, *NextVertex = *VList;
        static Polygon *LastPolygon;
```

```
Vector3 A, B, Norm;
double MaxNorm;

double V3Dot();
Sphere *GetBoundingSphere();
Box *GetBoundingBox();

/*
 * Allocate memory fo new polygon structure and array for vertex data
 */
if( (PTemp = NEWTYPE(Polygon,1)) == (Polygon *)NULL ||
        (PTemp->Vertices = NEWTYPE(Point3,VCount)) == (Point3 *)NULL ) {
        fprintf(stderr, "Memory Full\n" );
        exit(1);
}

PTemp->NVert = (u_char)VCount;

/*
 * Copy the vertex data from the vertex list into the array, freeing the list as the data
 * is transferred
 */
for(i=0; i<VCount; i++ ) {
        PTemp->Vertices[i] = NextVertex->Position;
        VTemp = NextVertex->Next;
        free(NextVertex);
        NextVertex = VTemp;
}

/*
 * The vertex list should be completely freed now, so...
 */
*VList = (Vertex *)NULL;

/*
 * Copy the colour components
 */
PTemp->Colour.r = (double)r/255.0;
PTemp->Colour.g = (double)g/255.0;
PTemp->Colour.b = (double)b/255.0;

/*
 * Calculate the normal vector of the polygon
 */
V3Sub(&PTemp->Vertices[1], &PTemp->Vertices[0], &A );
V3Sub(&PTemp->Vertices[2], &PTemp->Vertices[0], &B );

V3Cross(&A, &B, &PTemp->Normal );
V3Normalise(&PTemp->Normal);

/*
 * Calculate the plane constant
 */
PTemp->D = -V3Dot(&PTemp->Normal, &PTemp->Vertices[0] );

/*
 * Determine the dominant axis of the normal
 */
Norm.x = fabs(PTemp->Normal.x);
Norm.y = fabs(PTemp->Normal.y);
Norm.z = fabs(PTemp->Normal.z);

MaxNorm = MAX(Norm.z, Norm.y );
MaxNorm = MAX(Norm.x, MaxNorm );

if( MaxNorm == Norm.x )
        PTemp->DomAxis = 0;
else if( MaxNorm == Norm.y )
        PTemp->DomAxis = 1;
```

```
        else
                PTemp->DomAxis = 2;

        /*
         * Get bounding volume.
         */
        switch( BoundingVolume )
        {
                case SPHERE:
                        PTemp->BoundSphere = GetBoundingSphere(PTemp);
                        PTemp->BVIntersect = BVSphereIntersect;
                        PTemp->BoundBox = (Box *)NULL;
                        break;

                case BOX1:
                        PTemp->BoundBox = GetBoundingBox(PTemp);
                        PTemp->BVIntersect = BVBoxIntersect1;
                        PTemp->BoundSphere = (Sphere *)NULL;
                        break;

                case BOX2:
                        PTemp->BoundBox = GetBoundingBox(PTemp);
                        PTemp->BVIntersect = BVBoxIntersect2;
                        PTemp->BoundSphere = (Sphere *)NULL;
                        break;
        }

        PTemp->Next = (Polygon *)NULL;

        /*
         * Stick the completed polygon structure onto the polygon list
         */
        if( *PList == (Polygon *)NULL ) /* List is empty */
                *PList = PTemp;
        else
                LastPolygon->Next = PTemp;

        LastPolygon = PTemp;

}



/*
 * Perform a polygon intersection test with the given polygon and ray description.
 * Returns TRUE if polygon is intersected, FALSE otherwise. Also returns the
 * parameter distance along the ray if intersected.
 * Based upon the algorithm written by Didier Badouel in "Graphics Gems".
 * NOTE: Applies only to CONVEX polygons
 */
boolean PolygonIntersect(Poly, Rayy, MinDist)
Polygon *Poly;
Ray *Rayy;
double *MinDist;
{
        double t;/* Parameter t along ray */
        double u0, v0, u1, v1, u2, v2;
        boolean Intersection;
        double Alpha, Beta;
        int i;

        double V3Dot();

        t = V3Dot(&Poly->Normal, &Rayy->Dir );

        if( fabs(t) < EPSILON )
                /*
                 * Ray parallel to polygon
                 */
```

```
        return(FALSE);

t = ( -V3Dot(&Poly->Normal, &Rayy->Origin) - Poly->D ) / t;

if( t <= (double)0.0 || t > *MinDist )
        /*
         * Intersection behind eye or to far
         */
        return(FALSE);

/*
 * Calculate point of intersection on the projected plane
 */
switch( Poly->DomAxis )
{
        case 0:
                u0 = Rayy->Origin.y + t * Rayy->Dir.y - Poly->Vertices[0].y;
                v0 = Rayy->Origin.z + t * Rayy->Dir.z - Poly->Vertices[0].z;
                break;
        case 1:
                u0 = Rayy->Origin.z + t * Rayy->Dir.z - Poly->Vertices[0].z;
                v0 = Rayy->Origin.x + t * Rayy->Dir.x - Poly->Vertices[0].x;
                break;
        case 2:
                u0 = Rayy->Origin.x + t * Rayy->Dir.x - Poly->Vertices[0].x;
                v0 = Rayy->Origin.y + t * Rayy->Dir.y - Poly->Vertices[0].y;
                break;
}


/*
 * Do the intersection test.
 * The polygon is viewed as NVert-2 triangles with a common
 * vertex ( Vertex[0] ). Each triangle is tested in turn.
 */
Intersection = FALSE;
i = 2;

do {
        switch( Poly->DomAxis )
        {
                case 0:
                        u1 = Poly->Vertices[i-1].y - Poly->Vertices[0].y;
                        v1 = Poly->Vertices[i-1].z - Poly->Vertices[0].z;
                        u2 = Poly->Vertices[i].y   - Poly->Vertices[0].y;
                        v2 = Poly->Vertices[i].z   - Poly->Vertices[0].z;
                        break;

                case 1:
                        u1 = Poly->Vertices[i-1].z - Poly->Vertices[0].z;
                        v1 = Poly->Vertices[i-1].x - Poly->Vertices[0].x;
                        u2 = Poly->Vertices[i].z   - Poly->Vertices[0].z;
                        v2 = Poly->Vertices[i].x   - Poly->Vertices[0].x;
                        break;

                case 2:
                        u1 = Poly->Vertices[i-1].x - Poly->Vertices[0].x;
                        v1 = Poly->Vertices[i-1].y - Poly->Vertices[0].y;
                        u2 = Poly->Vertices[i].x   - Poly->Vertices[0].x;
                        v2 = Poly->Vertices[i].y   - Poly->Vertices[0].y;
                        break;
        }

        if( u1 == 0.0 ) {
                Beta = u0/u2;
                if( (Beta >= 0.0) && (Beta <= 1.0) ) {
                        Alpha = (v0 - Beta*v2) / v1;
                        Intersection = (Alpha >= 0.0) && (Alpha+Beta <= 1.0);
                }
```

```
                }
                else {
                        Beta = (v0*u1 - u0*v1) / (v2*u1 - u2*v1);
                        if( (Beta >= 0.0) && (Beta <= 1.0) ) {
                                Alpha = (u0 - Beta*u2) / u1;
                                Intersection = (Alpha >= 0.0) && (Alpha+Beta <= 1.0);
                        }
                }

        }
        while( (!Intersection) && (++i < Poly->NVert) );

        if( Intersection )
                *MinDist = t;

        return(Intersection);
}
```

```
/*
 * ReadData.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 9 June 1992
 *
 */

#include <stdio.h>
/*#include <string.h>*/
#include <math.h>
#include "utility.h"
#include "render.h"

extern Polygon *Polygons;
extern View *View1;
extern Vector3 LightDir;
extern double InvLightDir;
extern double LightIntensity, AmbientIntensity;
extern int ImageX, ImageY;
extern double FogVR, FogHeight, AtmHeight, SkyDepth;

double Scale;

/*
 * Read the view and initialisation data and fill a View structure.
 * The viewing system follows the method suggested by Andrew Glassner in
 * "Ray Tracing for Realism", BYTE, Dec. 1990
 */
View *ReadHeaderData(DataFile)
FILE *DataFile;
{
        View *V;
        char Line[64], Temp[64];
        Vector3 Up, G, X, Y;
        double Tempd;

        /*
         * Allocate memory for View structure
         */
        if( (V = NEWTYPE(View,1)) == (View *)NULL ) {
                fprintf(stderr, "\nMemory Full\n\n" );
                exit(1);
        }

        fgets(Line, 64, DataFile);
        sscanf(Line, "%s %lf %lf %lf", Temp, &V->EyePos.x, &V->EyePos.y, &V->EyePos.z );
        fgets(Line, 64, DataFile);
        sscanf(Line, "%s %lf %lf %lf", Temp, &V->LookPoint.x, &V->LookPoint.y,
                &V->LookPoint.z);
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf", Temp, &V->VPD );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf", Temp, &V->HFov );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf %lf %lf", Temp, &Up.x, &Up.y, &Up.z );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf %lf %lf", Temp, &LightDir.x, &LightDir.y, &LightDir.z );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf", Temp, &LightIntensity );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf", Temp, &AmbientIntensity );
        fgets(Line, 64, DataFile );
        sscanf(Line, "%s %lf", Temp, &Scale );
        fgets(Line, 64, DataFile );

        /*
```

```
         * Calculate view parameters
         */
        V3Sub(&V->LookPoint, &V->EyePos, &G);
        V3Cross(&G, &Up, &X);
        V3Cross(&X, &G, &Y);

        V3Normalise(&G);
        V3Normalise(&X);
        V3Normalise(&Y);

        /*
         * Tempd = VPD*tan( HFov/2 ) when HFov is in degrees.
         * However tan requires radians and there ain't no PI,
         * so...
         */
        Tempd = V->VPD*tan(atan(1.0)*V->HFov/90.0);

        V3Scale(&X, Tempd);
        V3Scale(&Y, Tempd*ImageY/ImageX);

        V->LookDir = G;
        V->HVect = X;
        V->VVect = Y;
        V->ImageX = ImageX;
        V->ImageY = ImageY;
        V->VisRange = FogVR;
        V->FogHeight = FogHeight;

        V3Normalise(&LightDir);
        InvLightDir = 1.0/LightDir.y;
        SkyDepth = (AtmHeight-FogHeight)*InvLightDir;

        return(V);
}



/*
 * Read polygon data from file and create list of polygon structures.
 * A pointer to the list is returned.
 */
Polygon *ReadPolygonData(DataFile)
FILE *DataFile;
{
        int i = 0, PolygonCount, VertexCount;
        char Line[128], Temp[64], Symbol[5];
        unsigned r, g, b;
        double x, y, z;
        double MaxX, MaxY, MaxZ, MinX, MinY, MinZ;

        Polygon *PolygonList = (Polygon *)NULL;
        Vertex *VertexList = (Vertex *)NULL;

        MaxX = MaxY = MaxZ = -1000000.0;
        MinX = MinY = MinZ =  1000000.0;

        PolygonCount = 0;
        strcpy(Symbol, "-\\|/" );
        fgets(Line, 128, DataFile );/* Start of first polygon */
        sscanf(Line, "%s", Temp );

        fprintf(stderr, "\nReading polygon data..." );

        while( strcmp(Temp, "Polygon") == 0 ) {
                fgets(Line, 128, DataFile );  /* Colours */
                sscanf(Line, "%u %u %u", &r, &g, &b );
                fgets(Line, 128, DataFile );  /* Blank line */
                fgets(Line, 128, DataFile );  /* "Polygon vertices" */
```

```
			VertexCount = 0;
			fgets(Line, 128, DataFile );  /* First line of vertex data */

			while( sscanf(Line, "%s %s %lf %lf %lf", Temp, Temp, &x, &y, &z ) == 5 ) {
				x *= Scale;
				y *= Scale;
				z *= Scale;

				VertexCreate(&VertexList, x, y, z );
				VertexCount++;
				fgets(Line, 128, DataFile );/* Next line of vertex data */

				MinX = MIN(x, MinX );
				MaxX = MAX(x, MaxX );
				MinY = MIN(y, MinY );
				MaxY = MAX(y, MaxY );
				MinZ = MIN(z, MinZ );
				MaxZ = MAX(z, MaxZ );
			}

			PolygonCreate(&PolygonList, &VertexList, VertexCount, r, g, b );
			PolygonCount++;

			fgets(Line, 128, DataFile );  /* Start of next polygon */
			sscanf(Line, "%s", Temp );

			if( !(PolygonCount%20) ) {
				fprintf(stderr, "\b%c", Symbol[i%4] );
				fflush(stderr);
				i++;
			}

		}

		fprintf(stderr, "\r%d polygons read              \n\n", PolygonCount );
		fprintf(stderr, "Max X = %15.6f  Min X = %15.6f\n", MaxX, MinX );
		fprintf(stderr, "Max Y = %15.6f  Min Y = %15.6f\n", MaxY, MinY );
		fprintf(stderr, "Max Z = %15.6f  Min Z = %15.6f\n", MaxZ, MinZ );

		return(PolygonList);
}
```

```
/*
 * Render.c (MAIN)
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 28 May 1992
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "utility.h"
#include "render.h"

extern RGBColour ShadePolygon1(), ShadePolygon2(), ShadeBackground1(), ShadeBackground2();

#define EARTHRADIUS 6371.0

Polygon *Polygons;
View *View1;
RGBColour FogBackground, SkyBackground;
Vector3 LightDir;
double LightIntensity, AmbientIntensity;
ExtinctCoeff SkyExtCoeff, FogExtCoeff;
ScattCoeff SkyScattCoeff, FogScattCoeff;
double MinDirY;

/*
 * Default option values
 */
int ImageX = 200, ImageY = 150;
int SideSamples = 1, TotalSamples = 1;
boolean ImageToWindow = TRUE, ImageToFile = FALSE;
boolean RenderShadows = TRUE;
boolean Jitter = FALSE;
int BoundingVolume = SPHERE;
double AtmHeight = 8.4, AtmVR = 250, FogHeight = -10.0, FogVR = 250;
RGBColour (*ShadePolygonProc)() = ShadePolygon1, (*ShadeBackgroundProc)() = ShadeBackground1;


RGBColour SampleScreen(x, y)
double x, y;
{
        Vector3 H, V, G;
        Ray EyeRay;
        Polygon *Nearest = (Polygon *)NULL;
        Polygon *CurrentPolygon = Polygons;
        double MinDist = FAR_AWAY;
        RGBColour Colour;

        boolean PolygonIntersect();
        void NormaliseColour();

        /*
         * Construct the eye ray
         */
        H = View1->HVect;
        V = View1->VVect;
        G = View1->LookDir;
        V3Scale(&H, (2.0*x-1.0) );
        V3Scale(&V, (2.0*y-1.0) );
        V3Scale(&G, View1->VPD );
        V3Add(&G, &H, &EyeRay.Dir );
        V3Add(&V, &EyeRay.Dir, &EyeRay.Dir );
        V3Normalise(&EyeRay.Dir);

        EyeRay.Origin = View1->EyePos;
```

```
        /*
         * Find the nearest intersection, if any
         */
        while( CurrentPolygon != (Polygon *)NULL ) {
                if( (*CurrentPolygon->BVIntersect)(CurrentPolygon, &EyeRay) &&
                        PolygonIntersect(CurrentPolygon, &EyeRay, &MinDist) )
                                Nearest = CurrentPolygon;
                CurrentPolygon = CurrentPolygon->Next;
        }

        if( Nearest == (Polygon *)NULL )
                Colour = (*ShadeBackgroundProc)(EyeRay);
                /*Colour.r = Colour.g = Colour.b = 0.0;*/
        else
                Colour = (*ShadePolygonProc)(Nearest, EyeRay, MinDist);
                /*Colour = Nearest->Colour;*/

        /*
         * If any of the component colours are greater than 1.0, the colours are normalised to
         * the greatest value.
         * This clipping method maintains hue and saturation but clips intensity.
         * (Hall, "Illumination and Colour in Computer Generated Imagery" ).
         */
        if( Colour.r > 1.0 || Colour.g > 1.0 || Colour.b > 1.0 )
                NormaliseColour(&Colour);

        return(Colour);

}


boolean ShadowTest(IPolygon, IRay, Dist)
Polygon *IPolygon;
Ray *IRay;
double Dist;
{
        Polygon *CurrentPolygon;
        Ray SRay;
        double Distance = FAR_AWAY;
        Vector3 Temp;
        boolean Shadow = FALSE;

        boolean PolygonIntersect();

        /*
         * Construct the shadow ray
         */
        Temp = IRay->Dir;
        V3Scale(&Temp, Dist);
        V3Add(&IRay->Origin, &Temp, &SRay.Origin);
        SRay.Dir = LightDir;

        /*
         * Perform the shadow ray intersection tests, excluding the intersected polygon to
         * avoid self-intersection. Also test whether the distance to the 'shadowing' polygon
         * is large enough to avoid coincident polygons. Both of these effects produce RTA,
         * Ray Tracing Acne.
         * Note: The very first intersection by the shadow ray is sufficient to decide whether
         * the surface point is in shadow. This follows because the light source is at infinity
         * so that no other polygons will lie beyond it.
         */
        CurrentPolygon = Polygons;

        while( CurrentPolygon != (Polygon *)NULL ) {
            if( CurrentPolygon != IPolygon &&
                    (*CurrentPolygon->BVIntersect)(CurrentPolygon, &SRay) &&
                            PolygonIntersect(CurrentPolygon, &SRay, &Distance) &&
                                    Distance > EPSILON ) {
```

```
                                        Shadow = TRUE;
                                        break;
            }
            CurrentPolygon = CurrentPolygon->Next;
        }

        return(Shadow);

}


void NormaliseColour(Colour)
RGBColour *Colour;
{
        double MaxColour, Scale;

        MaxColour = MAX(Colour->r, Colour->g);
        MaxColour = MAX(Colour->b, MaxColour);

        Scale = 1.0/MaxColour;

        Colour->r *= Scale;
        Colour->g *= Scale;
        Colour->b *= Scale;
}


void GetOptions(argc, argv, InFile, OutFile)
int argc;
char **argv;
char **InFile, **OutFile;
{
        /*
         * The following options are allowed:
         *
         * -ns:
         *      Do not render shadows.
         *      (Default: shadows rendered.)
         * -s <n>:
         *      Adaptively supersample image with n^2 samples.
         *      (Default: do not supersample, n=1.)
         * -j:
         *      Use jittered supersampling.
         *      (Default: use uniform supersampling)
         * -r <x><y>:
         *      Specify image resolution as x by y pixels.
         *      (Default: x=200, y=150.)
         * -nd:
         *      Do not display the image as it is being rendered.
         *      If an output file is not specified via the option -o then -nd is ignored.
         *      (Default: use display window.)
         * -o <outfile>:
         *      Write the image to a Sun rasterfile when rendering has finished.
         *      (Default: do not write to file.)
         * -b <box>:
         *      Use axis-aligned boxes as the bounding volume. Two different intersection
         *      tests are available specified as <box>=1 or 2.
         *      Option included for development purposes only.
         *      (Default: use bounding spheres.)
         * -f <range> <height>:
         *      Use fog of given visual range and height in rendering.
         *      (Default: <range> = 250km, <height> = -10km, ie no fog!)
         * -a <range> <height>:
         *      Set the visual range and height of the atmosphere.
         *      (Default: <range> = 250km, <height> = 8.4km)
         * -m:
         *      Do not use simple atmospheric model.
         *      (Default: use simple model.)
         */
```

```
char *Option;
char *ImageDir, *DataFile, *ImageFile;

while( --argc > 0 ) {
        Option = *++argv;

        if( !strcmp(Option, "-o")) {
                ImageFile = *++argv;
                ImageToFile = TRUE;
                argc--;
        }
        else if( !strcmp(Option, "-ns")) {
                RenderShadows = FALSE;
        }
        else if( !strcmp(Option, "-s")) {
                SideSamples = atoi(*++argv);
                TotalSamples = SideSamples*SideSamples;
                argc--;
        }
        else if( !strcmp(Option, "-j")) {
                Jitter = TRUE;
        }
        else if( !strcmp(Option, "-m")) {
                ShadePolygonProc = ShadePolygon2;
                ShadeBackgroundProc = ShadeBackground2;
        }
        else if( !strcmp(Option, "-r")) {
                ImageX = atoi(*++argv);
                ImageY = atoi(*++argv);
                argc -= 2;
        }
        else if( !strcmp(Option, "-nd")) {
                ImageToWindow = FALSE;
        }
        else if( !strcmp(Option, "-b")) {
                if( atoi(*++argv) == 1 )
                        BoundingVolume = BOX1;
                else if( atoi(*argv) == 2 )
                        BoundingVolume = BOX2;
                argc--;
        }
        else if( !strcmp(Option, "-f")) {
                sscanf(*++argv, "%lf", &FogVR );
                sscanf(*++argv, "%lf", &FogHeight );
                argc -= 2;
        }
        else if( !strcmp(Option, "-a")) {
                sscanf(*++argv, "%lf", &AtmVR );
                sscanf(*++argv, "%lf", &AtmHeight );
                argc -= 2;
        }
        else if( !strncmp(Option, "-", 1) ) {
                fprintf(stderr, "\nUnrecognised option: %s\n", Option );
                fprintf(stderr, "Options available:\n" );
                fprintf(stderr, " -ns -s <samples> -j -r <x><y> -nd -o <outfile>\n" );
                fprintf(stderr, " -b <box> -f <range><height> -a <range><height> -m\n" );
                exit(1);
        }
}

DataFile = *argv;

if( (ImageDir = getenv("IMAGES")) == (char *)NULL ) {
        fprintf(stderr, "\nEnvironment variable IMAGES not set\n" );
        exit(1);
}

strcpy(InFile, ImageDir );
```

```
        strcat(InFile, "/" );
        strcat(InFile, DataFile );

        if( ImageToFile ) {
                strcpy(OutFile, ImageDir );
                strcat(OutFile, "/" );
                strcat(OutFile, ImageFile );
        }

        if( !ImageToFile && !ImageToWindow ) {
                fprintf(stderr, "No output requested!!\n\n" );
                exit(1);
        }

        if( FogVR > AtmVR || FogHeight > AtmHeight ) {
                fprintf(stderr,
                        "Fog parameters must be less than those of the atmosphere!!\n\n" );
                exit(1);
        }

        MinDirY = sqrt(AtmHeight*0.5/EARTHRADIUS);
        /*
         * This seems to be the best place to put this.
         * Firstly, note that the default atmosphere height of 8.4km is equal to the scale height
         * of the real atmosphere. Assuming that the real atmosphere density falls exponentially
         * with height then a model atmosphere whose density is constant (equal to real atmosphere
         * density at sea level) and height is 8.4km, has the same OPTICAL DEPTH as the real
         * atmosphere when looking straight up.
         * Secondly, a minumum is set on the slope of the eye ray to avoid looking through
         * excessive amounts of air and/or fog in the model atmosphere. The equivalent range
         * corresponds to the optical depth of the real (curved) atmosphere when looking
         * horizontally at the Earth's surface, equal to about 290km.
         */
}


void main(argc, argv)
int argc;
char **argv;
{
        char ImageFileName[128], DataFileName[128];
        FILE *Data;
        int y;
        ScanLine Scan0, Scan1, Scan2;
        Pixel *TempPixels;
        int *TempSampled;

        Polygon *ReadPolygonData();
        View *ReadHeaderData();
        void ScanLineInit(), SampleScanLine(), SuperSampleScanLine();
        void OutputInit(), OutputScanLine();
        void DumpImage();
        void ReleaseWindow();
        void JitterInit();
        void GetCoeff();
        RGBColour GetBackground();

        /*
         * Parse the command line and return io file names
         */
        GetOptions(argc, argv, DataFileName, ImageFileName );

        /*
         * Open input data file
         */
        if( (Data = fopen(DataFileName, "r" )) == (FILE *)NULL ) {
                fprintf(stderr, "\nCannot open %s\n", DataFileName );
                exit(1);
        }
```

```
/*
 * Read the header data
 */
View1 = ReadHeaderData(Data);

/*
 * Read the polygon data into a linked list (queue type)
 */
Polygons = ReadPolygonData(Data);

fclose(Data);

ScanLineInit(&Scan0, &Scan1, &Scan2);

/*
 * Get the red, green and blue extinction and relative scattering coefficients of the fog
 * for the given visual range and those for the sky
 */
GetCoeff(FogVR, &FogExtCoeff, &FogScattCoeff);
GetCoeff(AtmVR, &SkyExtCoeff, &SkyScattCoeff);

/*
 * Get background colours (redundant?)
 */
FogBackground = GetBackground(FogVR);
SkyBackground = GetBackground(AtmVR);

if( Jitter )
        JitterInit();

printf("\nStarting image\n\n");

/*
 * All systems are GO, we have lift-off
 */

OutputInit();

SampleScanLine(0, &Scan0);
SampleScanLine(1, &Scan1);

for(y=1; y<ImageY-1; y++ ) {

        SampleScanLine(y+1, &Scan2);

        if( SideSamples > 1 )
                SuperSampleScanLine(y, &Scan0, &Scan1, &Scan2);

        OutputScanLine(Scan0.Pixels, y-1);

        TempPixels = Scan0.Pixels;
        TempSampled = Scan0.Sampled;
        Scan0.Pixels = Scan1.Pixels;
        Scan0.Sampled = Scan1.Sampled;
        Scan1.Pixels = Scan2.Pixels;
        Scan1.Sampled = Scan2.Sampled;
        Scan2.Pixels = TempPixels;
        Scan2.Sampled = TempSampled;

        if( !ImageToWindow && !(y%10) )
                printf("Finished line %3d\n", y );
}

OutputScanLine(Scan0.Pixels, y-1);
OutputScanLine(Scan1.Pixels, y);

printf("\nFinished image\n\n" );
```

```
        if( ImageToFile ) {
                printf("Writing image file..." ); fflush(stdout);
                DumpImage(ImageFileName);
                printf("Done\n\n");
        }

        if( ImageToWindow )
                ReleaseWindow();

}
```

```
/*
 * Sampling.c
 *
 * This file is part of Render
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 25 June 1992
 *
 */

/*
 * Most of this code is based on that in Rayshade 4.0 modified for use here
 */

#include <math.h>
#include "utility.h"
#include "render.h"

extern int ImageX, ImageY;
extern int SideSamples, TotalSamples;
extern boolean Jitter;

#define SINGLESAMPLED -1
#define SUPERSAMPLED -2

void ScanLineInit(S0, S1, S2)
ScanLine *S0, *S1, *S2;
{
        int i;

        S0->Pixels = NEWTYPE(Pixel, ImageX);
        S1->Pixels = NEWTYPE(Pixel, ImageX);
        S2->Pixels = NEWTYPE(Pixel, ImageX);

        S0->Sampled = NEWTYPE(int, ImageX);
        S1->Sampled = NEWTYPE(int, ImageX);
        S2->Sampled = NEWTYPE(int, ImageX);

        /*
         * Initialise pixels to zero
         */
        for(i=0; i<ImageX; i++) {
                S0->Pixels[i].Colour = 0L;
                S1->Pixels[i].Colour = 0L;
                S2->Pixels[i].Colour = 0L;
        }
}

/*
 * Sample each pixel in the scanline once.
 */
void SampleScanLine(Line, Scan)
int Line;
ScanLine *Scan;
{
        double x, y, Jx = 0.5, Jy = 0.5;
        int i;
        RGBColour Colour;

        RGBColour SampleScreen();
        void Jitter1();

        for(i=0; i<ImageX; i++ ) {

                if( Jitter )
                        Jitter1(i, Line, 0, &Jx, &Jy);

                x = ((double)i+Jx)/ImageX;
                y = ((double)Line+Jy)/ImageY;
```

```
                Colour = SampleScreen(x, y);

                Scan->Pixels[i].RGB[3] = (u_char)(255.0*Colour.r);
                Scan->Pixels[i].RGB[2] = (u_char)(255.0*Colour.g);
                Scan->Pixels[i].RGB[1] = (u_char)(255.0*Colour.b);
                Scan->Pixels[i].RGB[0] = 0;

                Scan->Sampled[i] = SINGLESAMPLED;
        }
}


void SuperSampleScanLine(Line, S0, S1, S2)
int Line;
ScanLine *S0, *S1, *S2;
{
        int i, Done;

        boolean ExcessiveContrast();
        void SuperSamplePixel();

        /*
         * Run down S1 and look for excessive contrast between pixel and it's FOUR neighbours.
         * If found then supersample all FIVE pixels. Repeat until there are no high contrast
         * regions or all regions have been supersampled
         */

        do {
                Done = TRUE;
                for(i=1; i<ImageX-1; i++ )

                if( ExcessiveContrast(i, S0->Pixels, S1->Pixels, S2->Pixels) ) {

                        if( S1->Sampled[i-1] != SUPERSAMPLED ) {
                                Done = FALSE;
                                SuperSamplePixel(i-1,Line, &S1->Pixels[i-1],&S1->Sampled[i-1] );
                        }

                        if( S0->Sampled[i] != SUPERSAMPLED ) {
                                Done = FALSE;
                                SuperSamplePixel(i, Line-1, &S0->Pixels[i], &S0->Sampled[i] );
                        }

                        if( S1->Sampled[i+1] != SUPERSAMPLED ) {
                                Done = FALSE;
                                SuperSamplePixel(i+1,Line, &S1->Pixels[i+1],&S1->Sampled[i+1] );
                        }

                        if( S2->Sampled[i] != SUPERSAMPLED ) {
                                Done = FALSE;
                                SuperSamplePixel(i, Line+1, &S2->Pixels[i], &S2->Sampled[i] );
                        }

                        if( S1->Sampled[i] != SUPERSAMPLED ) {
                                Done = FALSE;
                                SuperSamplePixel(i, Line, &S1->Pixels[i], &S1->Sampled[i] );
                        }

                }
        } while( !Done );

}

/*
 * Calculate the contrast between a group of five neighbouring pixels in each
 * colour channel. The contrast is defined as (max-min)/(max+min) where max and
 * min are the maximum and miniumum intensities.
 */
```

```
boolean ExcessiveContrast(i, Pix0, Pix1, Pix2)
int i;
Pixel *Pix0, *Pix1, *Pix2;
{
        int Min, Max, Diff, Sum;
        float RedContrast = 0.25, GreenContrast = 0.20, BlueContrast = 0.40;

        Max = MAX( Pix0[i].RGB[3], Pix1[i-1].RGB[3]);
        Max = MAX( Pix1[i].RGB[3], Max );
        Max = MAX( Pix1[i+1].RGB[3], Max );
        Max = MAX( Pix2[i].RGB[3], Max );

        Min = MIN( Pix0[i].RGB[3], Pix1[i-1].RGB[3]);
        Min = MIN( Pix1[i].RGB[3], Min );
        Min = MIN( Pix1[i+1].RGB[3], Min );
        Min = MIN( Pix2[i].RGB[3], Min );

        Diff = Max - Min;
        Sum = Max + Min;
        if( Sum > 0 && (float)Diff/Sum > RedContrast )
                return(TRUE);

        Max = MAX( Pix0[i].RGB[2], Pix1[i-1].RGB[2]);
        Max = MAX( Pix1[i].RGB[2], Max );
        Max = MAX( Pix1[i+1].RGB[2], Max );
        Max = MAX( Pix2[i].RGB[2], Max );

        Min = MIN( Pix0[i].RGB[2], Pix1[i-1].RGB[2]);
        Min = MIN( Pix1[i].RGB[2], Min );
        Min = MIN( Pix1[i+1].RGB[2], Min );
        Min = MIN( Pix2[i].RGB[2], Min );

        Diff = Max - Min;
        Sum = Max + Min;
        if( Sum > 0 && (float)Diff/Sum > GreenContrast )
                return(TRUE);

        Max = MAX( Pix0[i].RGB[1], Pix1[i-1].RGB[1]);
        Max = MAX( Pix1[i].RGB[1], Max );
        Max = MAX( Pix1[i+1].RGB[1], Max );
        Max = MAX( Pix2[i].RGB[1], Max );

        Min = MIN( Pix0[i].RGB[1], Pix1[i-1].RGB[1]);
        Min = MIN( Pix1[i].RGB[1], Min );
        Min = MIN( Pix1[i+1].RGB[1], Min );
        Min = MIN( Pix2[i].RGB[1], Min );

        Diff = Max - Min;
        Sum = Max + Min;
        if( Sum > 0 && (float)Diff/Sum > BlueContrast )
                return(TRUE);

        return(FALSE);

}

/*
 * A pixel is supersampled by dividing it into TotalSamples sub-pixels and the
 * centre of each sub-pixel is sampled. The resultant colour of the parent pixel
 * is computed simply by averaging the colours of the sub-pixels, i.e. a box
 * filter is used. The original (single sampled) colour of the pixel is not
 * used in the filtering.
 */
void SuperSamplePixel(xp, yp, Pix, Samp)
int xp, yp;
Pixel *Pix;
int *Samp;
{
        double x, y, Jx = 0.5, Jy = 0.5;
```

```
        double Temp1 = (double)xp*SideSamples;
        double Temp2 = (double)yp*SideSamples;
        double Temp3 = 1.0/(SideSamples*ImageX);
        double Temp4 = 1.0/(SideSamples*ImageY);
        int i;
        RGBColour SampleColour, PixelColour;

        RGBColour SampleScreen();
        void Jitter1();

        PixelColour.r = PixelColour.g = PixelColour.b = 0.0;

        for(i=0; i<TotalSamples; i++ ) {

                if( Jitter )
                        Jitter1(xp, yp, i, &Jx, &Jy);

                x = (Temp1 + i%SideSamples + Jx)*Temp3;
                y = (Temp2 + i/SideSamples + Jy)*Temp4;

                SampleColour = SampleScreen(x, y);

                PixelColour.r += SampleColour.r;
                PixelColour.g += SampleColour.g;
                PixelColour.b += SampleColour.b;

        }

        PixelColour.r /= TotalSamples;
        PixelColour.g /= TotalSamples;
        PixelColour.b /= TotalSamples;

        Pix->RGB[3] = (u_char)(255.0*PixelColour.r);
        Pix->RGB[2] = (u_char)(255.0*PixelColour.g);
        Pix->RGB[1] = (u_char)(255.0*PixelColour.b);
        Pix->RGB[0] = 255;

        *Samp = SUPERSAMPLED;

}

/*
 * What use can be made of the previously calculated single-sampled
 * colour of the pixel? Could it be usefully incorporated into the
 * supersampling scheme? Possibly use it in a weighted filter function.
 * Would any of this improve the quality of the image?
 * Life, The Universe and Everything, indeed!!
 */
```

```
/*
 * Shade.c
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 16 July 1992
 *
 */

#include <stdio.h>
#include <math.h>
#include "utility.h"
#include "render.h"

extern Polygon *Polygons;
extern View *View1;
extern RGBColour FogBackground, SkyBackground;
extern Vector3 LightDir;
extern double LightIntensity, AmbientIntensity;
extern boolean RenderShadows;
extern double FogHeight, AtmHeight, MinDirY;
extern ExtinctCoeff SkyExtCoeff, FogExtCoeff;
extern ScattCoeff SkyScattCoeff, FogScattCoeff;

/*
 * Reciprocal of LightDir.y
 * Used to avoid lots of floating-point divisions!
 */
double InvLightDir;

/*
 * Distance along light source ray from top of atmosphere to top of fog
 */
double SkyDepth;

RGBColour ShadePolygon1(IPolygon, IRay, Dist)
Polygon *IPolygon;     /* Intersected polygon */
Ray IRay;              /* Intersecting ray */
double Dist;           /* Distance parameter to polygon */
{
        RGBColour Colour, PolygonColour;
        double Oy = IRay.Origin.y;
        double FogRange, PolygonHeight;
        double RedExp, GreenExp, BlueExp;

        RGBColour Shade();

        /*
         * Calculate the height of the polygon intersection point
         */
        PolygonHeight = Oy + IRay.Dir.y*Dist;

        /*
         * Shade the polygon according to direct light
         */
        PolygonColour = Shade(IPolygon, IRay, Dist);

        /*
         * Calculate the fog component
         */
        if( Oy < FogHeight ) {

                if( PolygonHeight < FogHeight )
                        /*
                         * Eye point and polygon intersection point are in the fog
                         */
                        FogRange = Dist;
                else
```

```
                        /*
                         * Eye point in the fog and polygon intersection point above the fog
                         */
                        FogRange = (FogHeight-Oy)/IRay.Dir.y;
        }
        else {
                if( PolygonHeight < FogHeight )
                        /*
                         * Eye point above the fog and polygon intersection point in the fog
                         */
                        FogRange = Dist - (FogHeight-Oy)/IRay.Dir.y;
                else
                        /*
                         * Both the eye point and the polygon intersection point are above the
                         * fog. No intervening fog, polygon colour is not affected.
                         */
                        return( PolygonColour );
        }

        RedExp = exp( -FogExtCoeff.r*FogRange );
        GreenExp = exp( -FogExtCoeff.g*FogRange );
        BlueExp = exp( -FogExtCoeff.b*FogRange );

        /*
         * Total colour = object component + fog component
         */
        Colour.r = PolygonColour.r*RedExp + FogBackground.r*(1.0-RedExp);
        Colour.g = PolygonColour.g*GreenExp + FogBackground.g*(1.0-GreenExp);
        Colour.b = PolygonColour.b*BlueExp + FogBackground.b*(1.0-BlueExp);

        return(Colour);

}


RGBColour Shade(IPolygon, IRay, Dist, Height)
Polygon *IPolygon; /* Intersected polygon */
Ray *IRay;      /* Intersecting ray */
double Dist;    /* Distance parameter to polygon */
{
        RGBColour Colour;
        double NDotL, NDotRayDir;
        boolean InShadow = FALSE;

        double V3Dot();
        boolean ShadowTest();

        Colour.r = AmbientIntensity*IPolygon->Colour.r;
        Colour.g = AmbientIntensity*IPolygon->Colour.g;
        Colour.b = AmbientIntensity*IPolygon->Colour.b;

        NDotL = V3Dot(&IPolygon->Normal, &LightDir);
        NDotRayDir = V3Dot(&IPolygon->Normal, &IRay->Dir );

        /*
         * Test intersection point for shadows only if we're looking at the illuminated side
         */
        if( RenderShadows && NDotRayDir < 0.0 )
                InShadow = ShadowTest(IPolygon, IRay, Dist);

        if( !InShadow && NDotL > 0.0 && NDotRayDir < 0.0 ) {

                Colour.r += LightIntensity*IPolygon->Colour.r*NDotL;
                Colour.g += LightIntensity*IPolygon->Colour.g*NDotL;
                Colour.b += LightIntensity*IPolygon->Colour.b*NDotL;
        }

        return(Colour);
```

```
}


RGBColour ShadeBackground1(ARay)
Ray ARay;
{
        RGBColour Colour;
        double Oy = ARay.Origin.y;
        double FogRange;
        double RedExp, GreenExp, BlueExp;

        /*
         * It is assumed that the fog extends to infinity in both the x and z directions and from
         * 0 to FogHeight in the y direction, even though the polygon model is finite (usual
         * case).
         */

        /*
         * Test whether the eye point is above the fog. If it is, there is no intervening fog and
         * we're looking at clear sky. Note that we maybe above the fog looking 'downwards' and
         * intercepting some fog beyond the extents of the polygon model. We assume that this has
         * no effect and we just see clear sky as well.
         */
        if( Oy > FogHeight )
                return(SkyBackground);

        /*
         * Otherwise we're looking through some thickness of fog. We could be looking
         * horizontally in which case we see the fog background colour. If we're looking downwards
         * we assume that the thickness of fog is large enough so that we are also seeing the
         * background colour.
         */
        if( ARay.Dir.y < EPSILON )
                return(FogBackground);

        /*
         * We must now be looking upwards through a finite depth of fog.
         * Calculate thickness of fog and resultant colour.
         */
        FogRange = (FogHeight-Oy)/ARay.Dir.y;

        RedExp = exp( -FogExtCoeff.r*FogRange );
        GreenExp = exp( -FogExtCoeff.g*FogRange );
        BlueExp = exp( -FogExtCoeff.b*FogRange );

        Colour.r = SkyBackground.r*RedExp + FogBackground.r*(1.0-RedExp);
        Colour.g = SkyBackground.g*GreenExp + FogBackground.g*(1.0-GreenExp);
        Colour.b = SkyBackground.b*BlueExp + FogBackground.b*(1.0-BlueExp);

        return(Colour);
}


RGBColour ShadePolygon2(IPolygon, IRay, Dist)
Polygon *IPolygon; /* Intersected polygon */
Ray IRay;       /* Intersecting ray */
double Dist;    /* Distance parameter to polygon */
{
        RGBColour Colour, ObjectComp, FogComp;
        double PolygonHeight;
        double Oy = IRay.Origin.y;

        RGBColour ObjectShade(), FogShade();

        /*
         * Calculate the height of the polygon intersection point
         */
        PolygonHeight = Oy + IRay.Dir.y*Dist;
```

```
        /*
         * Calculate the intensity reflected by the polygon from direct illumination
         * - the object component
         */
        ObjectComp = ObjectShade(IPolygon, IRay, Dist, PolygonHeight);
        /*ObjectComp.r = ObjectComp.g = ObjectComp.b = 0.0;*/

        /*
         * Calculate the intensity reflected by the intervening fog
         * - the fog component
         */
        FogComp = FogShade(IRay, Dist, PolygonHeight);
        /*FogComp.r = FogComp.g = FogComp.b = 0.0;*/

        /*
         * Total colour = object component + fog component
         */
        Colour.r = ObjectComp.r + FogComp.r;
        Colour.g = ObjectComp.g + FogComp.g;
        Colour.b = ObjectComp.b + FogComp.b;

        return(Colour);

}


RGBColour ObjectShade(IPolygon, IRay, Dist, Height)
Polygon *IPolygon; /* Intersected polygon */
Ray IRay;       /* Intersecting ray */
double Dist;    /* Distance parameter to polygon */
double Height; /* Polygon height obove the ground */
{
        RGBColour Colour;
        double NDotL, NDotRayDir;
        boolean InShadow = FALSE;
        double SkyRange, FogRange;
        double Oy = IRay.Origin.y;

        double V3Dot();
        boolean ShadowTest();

        Colour.r = AmbientIntensity*IPolygon->Colour.r;
        Colour.g = AmbientIntensity*IPolygon->Colour.g;
        Colour.b = AmbientIntensity*IPolygon->Colour.b;

        NDotL = V3Dot(&IPolygon->Normal, &LightDir);
        NDotRayDir = V3Dot(&IPolygon->Normal, &IRay.Dir );

        /*
         * Test intersection point for shadows only if we're looking at the illuminated side
         */
        if( RenderShadows && NDotRayDir < 0.0 )
                InShadow = ShadowTest(IPolygon, IRay, Dist);

        /*
         * Shade the polygon point if (a) the point is not in shadow (b) the point is illuminated
         * and (c) we're looking at the illuminated side.
         */
        if( !InShadow && NDotL > 0.0 && NDotRayDir < 0.0 ) {

                double LightRed, LightGreen, LightBlue;

                /*
                 * Calculate the intensity of the direct light at the polygon surface point
                 * after transversing the atmosphere and the fog
                 */
                if( Height > FogHeight ) {
                        /*
                         * The point on the polygon is above the fog
```

```
                 */
                SkyRange = (AtmHeight-Height)*InvLightDir;
                FogRange = 0.0;
        }
        else {
                /*
                 * The point on the polygon is in the fog
                 */
                SkyRange = SkyDepth;
                FogRange = (FogHeight-Height)*InvLightDir;
        }

        LightRed   = LightIntensity*
                        exp(-SkyExtCoeff.r*SkyRange-FogExtCoeff.r*FogRange);
        LightGreen = LightIntensity*
                        exp(-SkyExtCoeff.g*SkyRange-FogExtCoeff.g*FogRange);
        LightBlue  = LightIntensity*
                        exp(-SkyExtCoeff.b*SkyRange-FogExtCoeff.b*FogRange);

        /*
         * Calculate the intensity reflected from the polygon surface. Purely diffuse
         * reflection is considered, i.e. N.L
         */
        Colour.r += LightRed*NDotL*IPolygon->Colour.r;
        Colour.g += LightGreen*NDotL*IPolygon->Colour.g;
        Colour.b += LightBlue*NDotL*IPolygon->Colour.b;

}

/*
 * Calculate the intensity of the reflected light reaching the eye point
 */
if( Oy < FogHeight ) {
        if( Height < FogHeight ) {
                /*
                 * Both the eye point and the polygon surface point are in the fog
                 */
                SkyRange = 0.0;
                FogRange = Dist;
        }
        else {
                /*
                 * The eye point is in the fog and the polygon intersection point
                 * is above the fog
                 */
                FogRange = (FogHeight-Oy)/IRay.Dir.y;
                SkyRange = Dist-FogRange;
        }
}
else {
        if( Height < FogHeight ) {
                /*
                 * The eye point is above the fog and the polygon intersection  point
                 * is in the fog
                 */
                SkyRange = (FogHeight-Oy)/IRay.Dir.y;
                FogRange = Dist-SkyRange;
        }
        else {
                /*
                * Both the eye point and the polygon intersection point are above the fog
                 */
                SkyRange = Dist;
                FogRange = 0.0;
        }
}

Colour.r *= exp(-SkyExtCoeff.r*SkyRange-FogExtCoeff.r*FogRange);
Colour.g *= exp(-SkyExtCoeff.g*SkyRange-FogExtCoeff.g*FogRange);
```

```
        Colour.b *= exp(-SkyExtCoeff.b*SkyRange-FogExtCoeff.b*FogRange);

        return(Colour);

}


RGBColour FogShade(IRay, Dist, Height)
Ray IRay;       /* Intersecting ray */
double Dist;    /* Distance parameter to polygon */
double Height; /* Polygon height obove the ground */
{
        RGBColour Colour;
        double SkyRange, FogRange;
        double Oy = IRay.Origin.y;

        if( Oy < FogHeight ) {
                if( Height < FogHeight ) {
                        /*
                         * Both the eye point and the polygon surface point are in the fog.
                         * Calculate intensity scattered by the fog.
                         */
                        double DC, DD, Const1, Const2;

                        FogRange = Dist;

                        DC = (FogHeight-Oy)*InvLightDir;
                        DD = (FogHeight-Height)*InvLightDir;

                        Const1 = DD-DC+FogRange;
                        Const2 = LightIntensity*FogRange/Const1;

                        Colour.r = Const2*FogScattCoeff.r
                                        * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*DC)
                                        * (1.0 - exp(-FogExtCoeff.r*Const1))/FogExtCoeff.r;

                        Colour.g = Const2*FogScattCoeff.g
                                        * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*DC)
                                        * (1.0 - exp(-FogExtCoeff.g*Const1))/FogExtCoeff.g;

                        Colour.b = Const2*FogScattCoeff.b
                                        * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*DC)
                                        * (1.0 - exp(-FogExtCoeff.b*Const1))/FogExtCoeff.b;
                }
                else {
                        /*
                         * The eye point is in the fog and the polygon intersection point
                         * is above the fog. Light is scattered by both the fog and sky.
                         */
                        double Const1, Const2, D1, D2;
                        RGBColour Colour1, Colour2;

                        FogRange = (FogHeight-Oy)/IRay.Dir.y;
                        SkyRange = Dist-FogRange;

                        /*
                         * Calculate intensity scattered by the fog
                         */
                        D1 = (FogHeight-Oy)*InvLightDir;

                        if( EQUAL(FogHeight,D1) ) {
                                Const1 = LightIntensity*FogRange;

                                Colour1.r = Const1*FogScattCoeff.r
                                                * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*D1);
                                Colour1.g = Const1*FogScattCoeff.g
                                                * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*D1);
                                Colour1.b = Const1*FogScattCoeff.b
                                                * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*D1);
```

```
            }
            else {
                    Const1 = FogRange-D1;
                    Const2 = LightIntensity*FogRange/Const1;

                    Colour1.r = Const2*FogScattCoeff.r
                            * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*D1)
                            * (1.0 - exp(-FogExtCoeff.r*Const1))/FogExtCoeff.r;

                    Colour1.g = Const2*FogScattCoeff.g
                            * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*D1)
                            * (1.0 - exp(-FogExtCoeff.g*Const1))/FogExtCoeff.g;

                    Colour1.b = Const2*FogScattCoeff.b
                            * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*D1)
                            * (1.0 - exp(-FogExtCoeff.b*Const1))/FogExtCoeff.b;
            }

            /*
             * Calculate intensity scattered by the sky
             */
            D2 = (AtmHeight-Height)*InvLightDir;

            Const1 = D2-SkyDepth+SkyRange;
            Const2 = LightIntensity*SkyRange/Const1;

            Colour2.r = Const2*SkyScattCoeff.r
                            * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*FogRange)
                            * (1.0 - exp(-SkyExtCoeff.r*Const1))/SkyExtCoeff.r;

            Colour2.g = Const2*SkyScattCoeff.g
                            * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*FogRange)
                            * (1.0 - exp(-SkyExtCoeff.g*Const1))/SkyExtCoeff.g;

            Colour2.b = Const2*SkyScattCoeff.b
                            * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*FogRange)
                            * (1.0 - exp(-SkyExtCoeff.b*Const1))/SkyExtCoeff.b;

            Colour.r = Colour1.r + Colour2.r;
            Colour.g = Colour1.g + Colour2.g;
            Colour.b = Colour1.b + Colour2.b;

        }
}
else {
        if( Height < FogHeight ) {
            /*
             * The eye point is above the fog and the polygon intersection  point
             * is in the fog. Light is scattered by both the sky and the fog.
             */
            double Const1, Const2, D1, D2;
            RGBColour Colour1, Colour2;

            SkyRange = (FogHeight-Oy)/IRay.Dir.y;
            FogRange = Dist-SkyRange;

            /*
             * Calculate intensity scattered by the sky
             */
            D1 = (AtmHeight-Oy)*InvLightDir;

            Const1 = SkyDepth-D1+SkyRange;
            Const2 = LightIntensity*SkyRange/Const1;

            Colour1.r = Const2*SkyScattCoeff.r*exp(-SkyExtCoeff.r*D1)
                            * (1.0 - exp(-SkyExtCoeff.r*Const1))/SkyExtCoeff.r;

            Colour1.g = Const2*SkyScattCoeff.g*exp(-SkyExtCoeff.g*D1)
                            * (1.0 - exp(-SkyExtCoeff.g*Const1))/SkyExtCoeff.g;
```

```
                Colour1.b = Const2*SkyScattCoeff.b*exp(-SkyExtCoeff.b*D1)
                               * (1.0 - exp(-SkyExtCoeff.b*Const1))/SkyExtCoeff.b;

                /*
                 * Calculate intensity scattered by the fog
                 */
                D2 = (FogHeight-Height)*InvLightDir;

                Const1 = FogRange+D2;
                Const2 = LightIntensity*FogRange/Const1;

                Colour2.r = Const2*FogScattCoeff.r
                               * exp(-SkyExtCoeff.r*(SkyRange+SkyDepth))
                               * (1.0 - exp(-FogExtCoeff.r*Const1))/FogExtCoeff.r;

                Colour2.g = Const2*FogScattCoeff.g
                               * exp(-SkyExtCoeff.g*(SkyRange+SkyDepth))
                               * (1.0 - exp(-FogExtCoeff.g*Const1))/FogExtCoeff.g;

                Colour2.b = Const2*FogScattCoeff.b
                               * exp(-SkyExtCoeff.b*(SkyRange+SkyDepth))
                               * (1.0 - exp(-FogExtCoeff.b*Const1))/FogExtCoeff.b;

                Colour.r = Colour1.r + Colour2.r;
                Colour.g = Colour1.g + Colour2.g;
                Colour.b = Colour1.b + Colour2.b;

            }
            else {
                /*
                 * Both the eye point and the polygon intersection point are above the
                 * fog. Calculate intensity scattered by the sky.
                 */
                double DA, DB, Const1, Const2;

                SkyRange = Dist;
                DA = (AtmHeight-Oy)*InvLightDir;
                DB = (AtmHeight-Height)*InvLightDir;

                Const1 = DB-DA+SkyRange;
                Const2 = LightIntensity*SkyRange/Const1;

                Colour.r = Const2*SkyScattCoeff.r*exp(-SkyExtCoeff.r*DA)
                               * (1.0 - exp(-SkyExtCoeff.r*Const1))/SkyExtCoeff.r;

                Colour.g = Const2*SkyScattCoeff.g*exp(-SkyExtCoeff.g*DA)
                               * (1.0 - exp(-SkyExtCoeff.g*Const1))/SkyExtCoeff.g;

                Colour.b = Const2*SkyScattCoeff.b*exp(-SkyExtCoeff.b*DA)
                               * (1.0 - exp(-SkyExtCoeff.b*Const1))/SkyExtCoeff.b;
            }
        }

        return(Colour);
}


RGBColour ShadeBackground2(ARay)
Ray ARay;
{
        RGBColour Colour;
        double SkyRange, FogRange;
        double Const1, Const2, D1;
        double Oy = ARay.Dir.y;

        /*
         * It is assumed that the fog extends to infinity in both the x and z directions and from
         * 0 to FogHeight in the y direction, even though the polygon model is finite (usual
```

```
 * case).
 */

if( Oy > FogHeight ) {
        /*
         * The eye point is above the fog, there is no intervening fog and
         * we're looking at clear sky.
         */

        if( ARay.Dir.y < MinDirY )
                SkyRange = (AtmHeight-Oy)/MinDirY;
        else
                SkyRange = (AtmHeight-Oy)/ARay.Dir.y;

        D1 = (AtmHeight-Oy)*InvLightDir;

        if( EQUAL(SkyRange, D1) ) {
                Const1 = LightIntensity*SkyRange;

                Colour.r = Const1*SkyScattCoeff.r*exp(-SkyExtCoeff.r*D1);
                Colour.g = Const1*SkyScattCoeff.g*exp(-SkyExtCoeff.g*D1);
                Colour.b = Const1*SkyScattCoeff.b*exp(-SkyExtCoeff.b*D1);
        }
        else {
                Const1 = SkyRange-D1;
                Const2 = LightIntensity*SkyRange/Const1;

                Colour.r = Const2*SkyScattCoeff.r*exp(-SkyExtCoeff.r*D1)
                                * (1.0 - exp(-SkyExtCoeff.r*Const1))/SkyExtCoeff.r;

                Colour.g = Const2*SkyScattCoeff.g*exp(-SkyExtCoeff.g*D1)
                                * (1.0 - exp(-SkyExtCoeff.g*Const1))/SkyExtCoeff.g;

                Colour.b = Const2*SkyScattCoeff.b*exp(-SkyExtCoeff.b*D1)
                                * (1.0 - exp(-SkyExtCoeff.b*Const1))/SkyExtCoeff.b;
        }

}
else {
        /*
         * Otherwise we're looking through some thickness of fog.
         */
        RGBColour Colour1, Colour2;

        if( ARay.Dir.y < MinDirY ) {
                FogRange = (FogHeight-Oy)/MinDirY;
                SkyRange = (AtmHeight-FogHeight)/MinDirY;
        }
        else {
                FogRange = (FogHeight-Oy)/ARay.Dir.y;
                SkyRange = (AtmHeight-FogHeight)/ARay.Dir.y;
        }

        /*
         * Calculate intensity scattered by the fog
         */
        D1 = (FogHeight-Oy)*InvLightDir;

        if( EQUAL(FogRange, D1) ) {
                Const1 = LightIntensity*FogRange;

                Colour1.r = Const1*FogScattCoeff.r
                                * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*D1);

                Colour1.g = Const1*FogScattCoeff.g
                                * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*D1);

                Colour1.b = Const1*FogScattCoeff.b
                                * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*D1);
```

```
        }
        else {
                Const1 = FogRange-D1;
                Const2 = LightIntensity*FogRange/Const1;

                Colour1.r = Const2*FogScattCoeff.r
                                * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*D1)
                                * (1.0 - exp(-FogExtCoeff.r*Const1))/FogExtCoeff.r;

                Colour1.g = Const2*FogScattCoeff.g
                                * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*D1)
                                * (1.0 - exp(-FogExtCoeff.g*Const1))/FogExtCoeff.g;

                Colour1.b = Const2*FogScattCoeff.b
                                * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*D1)
                                * (1.0 - exp(-FogExtCoeff.b*Const1))/FogExtCoeff.b;
        }

        /*
         * Calculate intensity scattered by the sky
         */
        if( EQUAL(SkyRange, SkyDepth) ) {
                Const1 = LightIntensity*SkyRange;

                Colour2.r = Const1*SkyScattCoeff.r
                                * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*FogRange);

                Colour2.g = Const1*SkyScattCoeff.g
                                * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*FogRange);

                Colour2.b = Const1*SkyScattCoeff.b
                                * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*FogRange);
        }
        else {
                Const1 = SkyRange-SkyDepth;
                Const2 = LightIntensity*SkyRange/Const1;

                Colour2.r = Const2*SkyScattCoeff.r
                                * exp(-SkyExtCoeff.r*SkyDepth-FogExtCoeff.r*FogRange)
                                * (1.0 - exp(-SkyExtCoeff.r*Const1))/SkyExtCoeff.r;

                Colour2.g = Const2*SkyScattCoeff.g
                                * exp(-SkyExtCoeff.g*SkyDepth-FogExtCoeff.g*FogRange)
                                * (1.0 - exp(-SkyExtCoeff.g*Const1))/SkyExtCoeff.g;

                Colour2.b = Const2*SkyScattCoeff.b
                                * exp(-SkyExtCoeff.b*SkyDepth-FogExtCoeff.b*FogRange)
                                * (1.0 - exp(-SkyExtCoeff.b*Const1))/SkyExtCoeff.b;
        }

        Colour.r = Colour1.r + Colour2.r;
        Colour.g = Colour1.g + Colour2.g;
        Colour.b = Colour1.b + Colour2.b;
    }

    return(Colour);
}
```

```
/*
 * Vector.c
 *
 * This file is part of Render.
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 26 May 1992
 *
 */

#include <math.h>
#include "utility.h"
#include "render.h"

double V3SquaredLength(a)
Vector3 *a;
{
	return( (a->x*a->x) + (a->y*a->y) + (a->z*a->z) );
}


double V3Length(a)
Vector3 *a;
{
	double V3SquaredLength();

	return( sqrt(V3SquaredLength(a)) );
}


void V3Normalise(v)
Vector3 *v;
{
	double V3Length();

	double len = V3Length(v);

	if( len != 0.0 ) {
		v->x /= len;
		v->y /= len;
		v->z /= len;
	}
}


void V3Add(a, b, c)
Vector3 *a, *b, *c;
{
	c->x = a->x + b->x;
	c->y = a->y + b->y;
	c->z = a->z + b->z;
}


void V3Sub(a, b, c)
Vector3 *a, *b, *c;
{
	c->x = a->x - b->x;
	c->y = a->y - b->y;
	c->z = a->z - b->z;
}



double V3Dot(a, b)
Vector3 *a, *b;
{
	return( (a->x*b->x) + (a->y*b->y) + (a->z*b->z) );
}
```

```
void V3Cross(a, b, c)
Vector3 *a, *b, *c;
{
        c->x = (a->y*b->z) - (a->z*b->y);
        c->y = (a->z*b->x) - (a->x*b->z);
        c->z = (a->x*b->y) - (a->y*b->x);
}


void V3Scale(v, s)
Vector3 *v;
double s;
{
        v->x *= s;
        v->y *= s;
        v->z *= s;
}


boolean V3Equal(a, b)
Vector3 *a, *b;
{
        return( EQUAL(a->x, b->x) && EQUAL(a->y, b->y) && EQUAL(a->z, b->z) );
}
```

D Auxiliary Program Code

List of files:

- Gamcor .c (SUN)
- Show .c (SUN)
- Display .c
- Encode.c
- Show .c (PC4000) (Not available)
- Gamcor .c (PC4000) (Not available)

```
/*
 * Gamcor.c (SUN)
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 16 June 1992
 *
 */

#include <stdio.h>
#include <pixrect/pixrect_hs.h>
#include <math.h>

/*
 * Gamma corrects the RasterOps card by re-writing the colour lookup tables
 * The default gamma value is 2.5, others are specified on the
 * command line
 */
void main(argc, argv)
int argc;
char **argv;
{
        int i;
        double Gamma, Ga;
        u_char Red[256], Green[256], Blue[256];
        Pixrect *Pr;

        if( argc == 1 )
                Gamma = 2.5;
        else {
                sscanf(argv[1], "%lf", &Gamma );
                if( Gamma == 0.0 ) {
                        fprintf(stderr, "\nIllegal gamma value: 0.0\n\n" );
                        exit(1);
                }
        }

        Ga = 1.0/Gamma;

        if( Gamma == 1.0 )
                for(i=0; i<256; i++ )
                        Red[i] = Green[i] = Blue[i] = (u_char)i;
        else
                for(i=0; i<256; i++ )
                        Red[i] = Green[i] = Blue[i] = (u_char)(255.0*pow((double)i/255.0, Ga));

        Pr = pr_open("/dev/cgeight0");
        pr_set_plane_group(Pr, PIXPG_24BIT_COLOR);
        pr_putlut(Pr, 0, 256, Red, Green, Blue );
        pr_close(Pr);

}
```

```
/*
 * Show.c (SUN)
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 3 July 1992
 *
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <stdlib.h>
#include "utility.h"
#include "render.h"

/*
 * Reads an image from a Sun rasterfile and displays it in a SunView window
 * on a 24-bit display. It is assumed that the file has been compressed using
 * compress(1).
 */

Frame BaseFrame;
Pixwin *ImagePw;
Pixrect *ImagePr;

struct singlecolor Background = {128,128,128};
struct singlecolor Foreground = {0,0,0};

void main(argc, argv)
int argc;
char **argv;
{
	char ImageFileName[128], *ImageDir, ZcatPtr[256];
	FILE *Image;
	View View1;
	Canvas MainCanvas;

	void RefreshImage();

	if( (ImageDir = getenv("IMAGES")) == (char *)NULL ) {
		fprintf(stderr, "\nEnvironment variable IMAGES not set\n" );
		exit(1);
	}

	strcpy(ImageFileName, ImageDir );
	strcat(ImageFileName, "/" );
	strcat(ImageFileName, argv[1] );

	strcpy(ZcatPtr, "zcat " );
	strcat(ZcatPtr, ImageFileName );

	if( (Image = popen(ZcatPtr, "r" )) == (FILE *)NULL ) {
		fprintf(stderr, "\nCannot open %s\n", ImageFileName );
		exit(1);
	}

	ImagePr = pr_load(Image, 0);

	if( fread(&View1, sizeof(View), 1, Image) != 1 ) {
		fprintf(stderr, "\nError reading trailer\n");
		exit(1);
	}

	BaseFrame = window_create(NULL, FRAME,
		FRAME_LABEL, argv[1],
		FRAME_NO_CONFIRM, TRUE,
		FRAME_BACKGROUND_COLOR, Background,
		FRAME_FOREGROUND_COLOR, Foreground,
		0 );
```

```
	MainCanvas  = window_create( BaseFrame, CANVAS,
		CANVAS_COLOR24, TRUE,
		CANVAS_RETAINED, FALSE,
		CANVAS_REPAINT_PROC, RefreshImage,
		CANVAS_AUTO_CLEAR, TRUE,
		CANVAS_AUTO_EXPAND, FALSE,
		CANVAS_AUTO_SHRINK, FALSE,
		CANVAS_WIDTH, View1.ImageX,
		CANVAS_HEIGHT, View1.ImageY,
		WIN_WIDTH, View1.ImageX,
		WIN_HEIGHT, View1.ImageY,
		0 );

	if( !MainCanvas ) {
		fprintf(stderr,"Cannot open 24-bit canvas on this machine\n\n" );
		exit(1);
	}

	ImagePw = canvas_pixwin(MainCanvas);

	window_fit(BaseFrame);

	printf("\nImage      : %s\n", ImageFileName );
	printf("Eye point  : %g %g %g\n", View1.EyePos.x, View1.EyePos.y,
			View1.EyePos.z );
	printf("Look point : %g %g %g\n", View1.LookPoint.x, View1.LookPoint.y,
			View1.LookPoint.z );
	printf("HFov       : %g\n", View1.HFov );
	printf("Vis. Range : %g\n", View1.VisRange );
	printf("Fog Height : %g\n", View1.FogHeight );
	printf("Image size : %dx%d\n\n", View1.ImageX, View1.ImageY );

	pclose(Image);

	window_main_loop(BaseFrame);
}

void RefreshImage()
{
	pw_rop(ImagePw, 0, 0, ImagePr->pr_size.x, ImagePr->pr_size.y,
		PIX_SRC, ImagePr, 0, 0);
}
```

```
/*
 * Display.c
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 13 July 1992
 *
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <stdlib.h>
#include “utility.h”
#include “render.h”

#define MARGIN 5
#define FONT “/usr/lib/fonts/fixedwidthfonts/screen.r.13”

struct singlecolor Background = {0, 0, 0};
struct singlecolor Foreground = {128, 128, 128};

#define RANGE 1
#define HEIGHT 2
#define SINGLE 3

main(argc, argv)
int argc;
char **argv;
{
        Frame BaseFrame;
        Canvas BaseCanvas;
        Pixwin *BasePw;
        Pixrect *Pr;
        Pixfont *Font;
        FILE *Image;
        char *ImageDir, Caption[32], FrameLabel[64], ZcatPtrRoot[128],
                ZcatPtr[128], Number[8];
        View View1;
        int i, x, y, XOffset, YOffset, Type;

        if( argc != 3 ) {
                fprintf(stderr, “Usage: display <dir> <info type>\n\n” );
                exit(1);
        }

        if( !(Font = pf_open(FONT)) ) {
                fprintf(stderr, “Cannot open font\n\n” );
                exit(1);
        }

        if( !(ImageDir = getenv(“IMAGES”)) ) {
                fprintf(stderr, “Environment variable IMAGES not set\n\n” );
                exit(1);
        }

        if( !strcmp(argv[2], “r”) )
                Type = RANGE;
        else if ( !strcmp(argv[2], “h”) )
                Type = HEIGHT;
        else if ( !strcmp(argv[2], “s”) )
                Type = SINGLE;
        else {
                fprintf(stderr, “Unknown information type\n\n” );
                exit(1);
        }

        BaseFrame = window_create( NULL, FRAME,
                FRAME_NO_CONFIRM, TRUE,
                FRAME_BACKGROUND_COLOR, Background,
```

```
        FRAME_FOREGROUND_COLOR, Foreground,
        WIN_X, 0, WIN_Y, 0,
        0 );

BaseCanvas = window_create( BaseFrame, CANVAS,
        CANVAS_COLOR24, TRUE,
        CANVAS_WIDTH, 1142,
        CANVAS_HEIGHT, 875,
        WIN_WIDTH, 1142,
        WIN_HEIGHT, 875,
        0 );

BasePw = canvas_pixwin(BaseCanvas);
window_fit(BaseFrame);
window_set(BaseFrame, WIN_SHOW, TRUE, 0);
notify_dispatch();

if( Type != SINGLE ) {
        strcpy(ZcatPtrRoot, "zcat " );
        strcat(ZcatPtrRoot, ImageDir );
        strcat(ZcatPtrRoot, "/" );
        strcat(ZcatPtrRoot, argv[1] );
        strcat(ZcatPtrRoot, "/fog" );

        for( i=0; i<9; i++ ) {

                strcpy(ZcatPtr, ZcatPtrRoot );
                sprintf(Number, "%d", i+1 );
                strcat(ZcatPtr, Number );

                if( !(Image = popen(ZcatPtr, "r")) ) {
                        fprintf(stderr, "Cannot open image file\n\n" );
                        exit(1);
                }

                Pr = pr_load(Image, 0);

                if( fread(&View1, sizeof(View), 1, Image) != 1 ) {
                        fprintf(stderr,
                                "Cannor read image file trailer\n\n" );
                        exit(1);
                }

                pclose(Image);

                XOffset = (1142 - 3*View1.ImageX - 2*MARGIN)/2;
                YOffset = (875 - 3*View1.ImageY - 2*MARGIN)/2;
                x = XOffset + (i%3)*(View1.ImageX + MARGIN);
                y = YOffset + (i/3)*(View1.ImageY + MARGIN);

                pw_rop(BasePw, x, y, View1.ImageX, View1.ImageY,
                        PIX_SRC, Pr, 0, 0);
                pr_close(Pr);

                if( Type == RANGE )
                        sprintf(Caption, "%g", View1.VisRange );
                else if( Type == HEIGHT )
                        sprintf(Caption, "%g", View1.FogHeight );

                strcat(Caption, "km" );
                pw_ttext(BasePw, x+10, y+View1.ImageY-10, PIX_SRC,
                        Font, Caption);
                notify_dispatch();

        }

        pf_close(Font);

        strcpy(FrameLabel, argv[1] );
```

```
		if( Type == RANGE ) {
			sprintf(Caption, "%g", View1.FogHeight );
			strcat(FrameLabel, " : Fog Height = " );
			strcat(FrameLabel, Caption );
			strcat(FrameLabel, "km" );
		}
		else if( Type == HEIGHT ) {
			sprintf(Caption, "%g", View1.VisRange );
			strcat(FrameLabel, " : Fog VR = " );
			strcat(FrameLabel, Caption );
			strcat(FrameLabel, "km" );
		}
	}
	else {
		strcpy(ZcatPtr, "zcat " );
		strcat(ZcatPtr, ImageDir );
		strcat(ZcatPtr, "/" );
		strcat(ZcatPtr, argv[1] );

		if( !(Image = popen(ZcatPtr, "r" )) ) {
			fprintf(stderr, "Cannot open image file\n" );
			exit(1);
		}

		Pr = pr_load(Image, 0);

		if( fread(&View1, sizeof(View), 1, Image) != 1 ) {
			fprintf(stderr, "Cannor read image file trailer\n\n" );
			exit(1);
		}

		pclose(Image);

		x = (1143 - View1.ImageX)/2;
		y = (875 - View1.ImageY)/2;

		pw_rop(BasePw, x, y, View1.ImageX, View1.ImageY,
			PIX_SRC, Pr, 0, 0);
		pr_close(Pr);

		sprintf(FrameLabel, "%s: Fog VR = %gkm Fog Height = %gkm",
				argv[1], View1.VisRange, View1.FogHeight );

	}

	window_set(BaseFrame, FRAME_LABEL, FrameLabel, 0);

	window_main_loop(BaseFrame);
	exit(0);

}
```

```
/*
 * Encode.c
 *
 * Author: John Irwin, cmmg12@scm.tp.ac.uk
 * Date: 21 July 1992
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <pixrect/pixrect_hs.h>
#include "utility.h"
#include "render.h"

/*
 * Reads an image from a Sun rasterfile and converts it into a run-length encoded
 * file based on unsigned longs. It is assumed that the file has been compressed using
 * compress(1).
 */

void main(argc, argv)
int argc;
char **argv;
{
        int i, j, k, Start;
        u_long *Scan, FLAG = 128L, *Temp;
        Pixel Count, Current;
        char ImageFileName[128], *ImageDir, ZcatPtr[256];
        FILE *Image;
        View View1;
        Pixrect *Pr;
        struct mpr_data *Data;

        if( !(ImageDir = getenv("IMAGES")) ) {
                fprintf(stderr, "\nEnvironment variable IMAGES not set\n" );
                exit(1);
        }

        strcpy(ImageFileName, ImageDir );
        strcat(ImageFileName, "/" );
        strcat(ImageFileName, argv[1] );

        strcpy(ZcatPtr, "zcat " );
        strcat(ZcatPtr, ImageFileName );

        if( !(Image = popen(ZcatPtr, "r" )) ) {
                fprintf(stderr, "\nCannot open %s\n", ImageFileName );
                exit(1);
        }

        Pr = pr_load(Image, 0);

        if( fread(&View1, sizeof(View), 1, Image) != 1 ) {
                fprintf(stderr, "\nError reading trailer\n");
                exit(1);
        }

        pclose(Image);

        strcat(ImageFileName, ".rle" );
        if( !(Image = fopen(ImageFileName, "w" )) ) {
                fprintf(stderr, "\nCannot open %s\n", ImageFileName );
                exit(1);
        }

        fprintf(Image, "%g %g %g\n", View1.EyePos.x, View1.EyePos.y, View1.EyePos.z );
        fprintf(Image, "%g %g %g\n", View1.LookPoint.x, View1.LookPoint.y, View1.LookPoint.z );
        fprintf(Image, "%g\n", View1.VPD );
        fprintf(Image, "%g\n", View1.HFov );
```

```
	fprintf(Image, “%d %d\n”, View1.ImageX, View1.ImageY );
	fprintf(Image, “%g %g\n”, View1.VisRange, View1.FogHeight );

	Data = mpr_d(Pr);
	Scan = (u_long *)Data->md_image;

	Count.Colour = 1L;
	Start = 1;

	for(i=0; i<View1.ImageY; i++ ) {
		Temp = Scan;

		if( i == 0 )
			Current.Colour = *Temp++;
		else
			Start = 0;

		for(j=Start; j<View1.ImageX; j++ ) {
			if( *Temp == Current.Colour )
				Count.Colour++;
			else {
				if( Count.Colour != 1L ) {
					if( Count.Colour > 2L ) {
						fwrite(&FLAG, sizeof(u_long), 1, Image );
						for(k=3; k>=0; k-- )
							fwrite(&Count.RGB[k], sizeof(u_char),
								1, Image );
					}
					else {
						for(k=3; k>=0; k-- )
							fwrite(&Current.RGB[k], sizeof(u_char),
								1, Image );
					}

					Count.Colour = 1L;
				}
				for(k=3; k>=0; k-- )
					fwrite(&Current.RGB[k], sizeof(u_char), 1, Image );

				Current.Colour = *Temp;
			}
			Temp++;
		}
		Scan = (u_long *)((char *)Scan + Data->md_linebytes);
		if( !(i%50) )
			printf(“Finished line %d\n”, i );
	}

	if( Count.Colour != 1L ) {
		fwrite(&FLAG, sizeof(u_long), 1, Image );
		for(k=3; k>=0; k-- )
			fwrite(&Count.RGB[k], sizeof(u_char), 1, Image );
	}
	for(k=3; k>=0; k-- )
		fwrite(&Current.RGB[k], sizeof(u_char), 1, Image );

	fclose(Image);
	pclose(Pr);

}
```