The Waterloo CGL Ray Tracing Package

Michael A.J. Sweeney

CS-85-35

October, 1985

The Waterloo CGL Ray Tracing Package

Michael A. J. Sweeney

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario

ABSTRACT

Ray tracing is the current method for state-of-the-art production of realistic computer generated imagery. It provides perspective, clipping, hidden surface elimination, shadows, reflections and transparency in one simple elegant algorithm.

A ray tracing package is described in detail. The package treats spheres, cylinders, polygons, fractals and B-spline surfaces as primitive objects. Relatively complex images may be built out of collections of primitive objects and modelling transformations. Each object may be rendered using Lambert's Law, the Phong illumination model, or the Cook-Torrance model. All objects may be texture mapped as well.

New features of the package are:

- an algorithm to ray trace free−form B−spline surfaces,
- translation of the Cook-Torrance illumination model to the the ray tracing context.
- an improved ambience function.
- texture mapped fractal surfaces.

This thesis serves as the external documentation of the ray tracing package, and as a user's guide as well. In particular, the format of the scene description file is presented in an appendix.

Contents

1. Introduction

Ray tracing is the current method for state-of-the-art production of realistic computer generated imagery. It provides perspective, clipping, hidden surface elimination, shadows, reflections and transparency in one simple elegant algorithm.

Unfortunately, the disadvantages of the algorithm are almost as great as its advantages. Ray tracing has a tremendous appetite for floating point calculations. This makes it not only the best but also the slowest of the current rendering techniques. This thesis does not attempt to deal with the issue of how one can make ray tracing go faster. However, there are four specific suggestions in Section 12.

The original motivation behind the thesis was to incorporate the Cook-Torrance illumination model into a ray tracing program. We then found ourselves wanting more primitive objects to render than just spheres and polygons. Fractals were incorporated, but they were not enough. So, finally, an algorithm to ray trace free-form B-spline surfaces was developed.

Production of the Waterloo CGL ray tracing package began in January 1983, and the package stabilized about a year later. It treats spheres, cylinders, polygons, fractals and B-spline surfaces as primitive objects. Relatively complex images may be built out of collections of primitive objects, some of which may be rotated, translated, and/or scaled. Each object may be rendered using Lambert's Law, the Phong illumination model, or the Cook-Torrance model. All objects may be texture mapped as well.

New features of the package are:

- an algorithm to ray trace free-form B-spline surfaces,
- translation of the Cook-Torrance illumination model to the the ray tracing context.
- an improved ambience function.
- texture mapped fractal surfaces.

This thesis serves as the external documentation of the package, and as a user's manual as well. Following the introduction, the various primitive objects are examined one by one. For each, the preprocessing steps and intersection algorithm are given. Next the heart of our program, how we generate rays and what we do with them, is discussed. This includes a look at the anti-aliasing algorithm used. Then the calculations for the various illumination models are given, followed by a comparison with other ray tracing programs, and suggestions for further work.

If you are reading this thesis simply to learn how to use the package, the relevant material is found in Section 1 (introduction), Appendix 1 (format of the scene file), and Appendix 2 (syntax of the program call statement).

Also as a part of an independent class project, Monol, a structured VAX assembler language has been developed. When all of the time critical routines of the package are coded in Monol, the resulting speedup is approximately 40 per cent. A Monol Manual is presented as Appendix 4.

1.1. Background

Turner Whitted's paper in the June 1980 issue of the CACM [Whitted80] is the primary reference for this thesis. It will be set forth in some detail below.

Whitted defines a virtual screen and a virtual viewpoint in the same coordinate system as the objects to be rendered. A line ("ray") is projected from the viewpoint through every pixel in this virtual screen. This ray is tested, one by one, with every object in the picture. If it intersects none of them, you see background on the corresponding real screen pixel.

If the ray does hit something, then a number of new rays may be generated. If the object struck is reflective, the first ray goes off in the direction of perfect reflection (ie angle of incidence equals angle of reflection). This ray is tested, one by one, with every object in the picture. If it hits something then more rays may be recursively generated.

A second ray may go off through the surface in the direction indicated by the surface index of refraction (Snell's law) if the object is transparent. This ray is tested, one by one, with every object in the picture (by now it should be becoming clear why ray tracing is so slow). If it hits something then more rays may be recursively generated.

Rays are projected from each intersection point towards each light source as well. If a ray hits something on the way, the the intersection point is in shadow with respect to that light source, and the light source's contribution to the diffuse reflection is attenuated.

Thus a tree is built, where each node represents the intersection point of a ray with the closest object in the scene, and each edge represents a ray. This tree is passed to the shader, which traverses it applying the following calculation at each node:

$$I = I_a + k_d \sum_{j=1}^{j=n} (\vec{N} \cdot \vec{L_j}) + k_r R + k_t T$$

where

I_a is the ambience intensity

k_d is the diffuse reflection constant

$\vec{N}$ is the surface normal

$\vec{L_j}$ is a vector in the direction of the light source

k_r is the surface reflectivity

R is the intensity coming in from the reflected ray

k_t is the surface transparency

T is the intensity coming in from the refracted ray

Notice that the specular term (Section 9.5) $k_r R$ reflects only in the true mirror-like direction. To produce a proper looking specular reflection, a small random perturbation must be added to the surface normal. In case of specular reflection directly from a light source, the Phong specular term (Section 9.8) is used in place of $k_r R$

Because of the spectacular images it produced, Whitted's work spawned a whole generation of ray tracing programs. This is one of them. An excellent summary of the current state of ray tracing can be found in Kajiya's Siggraph'83 tutorial [Kajiya83b]. I will not repeat what is said (better) there, only update it on two counts.

In all that has been written about ray tracing [Barr83, Hall83a, Hall83b, Hanrahan83, Kajiya82, Kajiya83a, Kay79, Max81, Potmesil82, Roth80, Roth82, Rubin80] since Whitted's seminal paper, only [Hanrahan83, Kajiya82, Rubin80, Whitted80] attempt to treat surfaces of higher than second order. None consider the problem of rendering truly free-form surfaces by ray tracing. As the literature now stands, such surfaces must be broken into bicubic (or slightly higher order) patches, and each of these patches rendered individually by one of the following techniques. Whitted [Whitted80, Rubin80] recursively subdivides a parametric patch with a method similar to those used in scan line algorithms; Kajiya computes the intersection of a ray and a bicubic patch exactly [Kajiya82] using results from algebraic geometry; Hanrahan [Hanrahan83] employs a symbolic algebra system to compute intersections with individual patches.

Also, Hall [Hall83a] stresses the importance of sampling the lighting intensities at more than the traditional three wavelengths (red, green and blue), especially when ray tracing. It is unclear from his article whether the "Hall improved interface model" is in fact Cook-Torrance shading adapted to the ray tracing context.

1.2. Overview of the Package

The Waterloo CGL ray tracing package is an implementation of Whitted's algorithm with little modification. The program reads a scene file and some time later produces a file of rgb values.

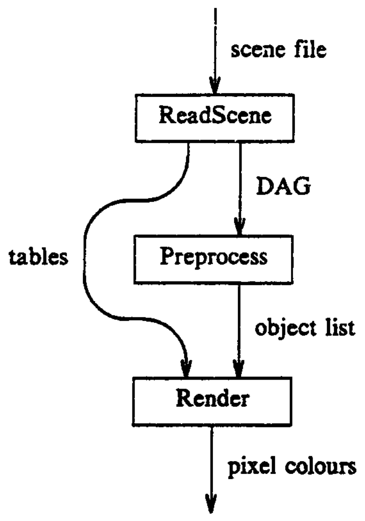


Figure 1. An overview.

The scene file contains a complete description of the picture to be rendered. This includes information about the geometry and surface characteristics of every object in the picture, the lighting environment, and the position and orientation of the viewer. The manner in which this information is organized was adapted from the scene file designed by K.S.Booth for the introductory course in graphics at the University of Waterloo.

ReadScene reads the scene file, producing a directed acyclic graph (DAG). *DAG nodes* represent either transformations or primitive objects, as read directly from the scene file. ReadScene also saves lighting information and surface characteristics in tables for use by the shader. An entry in the table of surface characteristics will be refered to as a *surface descriptor* in later sections.

A preprocessor walks this DAG, applying the transformations and processing the primitive objects into *object nodes*. An object node contains all of the information of the corresponding DAG node, but in a form which will be faster to intersect with rays. The output of the preprocessor is a linked list of object nodes.

Finally, the ray tracing algorithm is run. For every pixel a primary ray is projected into the object space. This ray is tested for intersection with every object in the object list, and the closest intersection found. Secondary rays are projected from the intersection point towards every light source to determine shadowing. Also if the object happens to be transparent or reflective, other rays are projected recursively in the appropriate directions. All information collected is passed to the shader, which returns an rgb value for that pixel. The output of the package is a run-length encoded file of pixel colours.

1.3. An Example

What follows is an example of a scene file, the DAG, the object list, and the final image produced. A rough English translation of this scene file follows Figure 2. See Appendix 1 for a full explaination of the file format.

```
* scene:  a mirror ball above a green/blue checkerboard
  5 2 4 2 1 4 1

* program parameters
1 depth 3
2 shadows
3 xleft  31
4 xright 480
5 yhigh  286

* display parameters
1 eye 0.0 0.0 3.5
2 sight towards 0.0 0.0 0.0

* vertices
1  -1.0 -0.0 -1.0
2  -1.0 -0.0 +1.0
3  +1.0 -0.0 +1.0
4  +1.0 -0.0 -1.0

* surface 1:  the checkerboard texture map
1 lambert textured 0.0 0.0 0.0
  7.0
  /u/majsweeney/textures/checkerboard 1 1
* surface 2:  a perfect reflector
2 phong normal 1.0 0.0 0.0
  0.0 2.5 25.0
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06

* ambience
  0.00050
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6

* lights
1 infinity  0.60000 1.00000 0.600000  0.00060
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6

* scene nodes
1 rotate    02 04       x -22.0
2 translate 03 00       0.0 -0.8 0.0
3 polygon   00 00 1      4 4 3 2 1
4 sphere    00 00 2      0.3 -0.18 0.0   0.16    x y
```

Figure 2. A sample scene description.

This scene description specifies that the package is to ray trace to a recursive depth of 3, and produce shadows. The output is bounded on the left, right and top. The observer is positioned at (0.0,0.0,3.5) looking towards the origin. There are two surfaces and one light source. One surface is a texture map of a checkerboard. The second is a dark gray material rendered with phong shading and a high degree of reflectiveness. The single light is at infinity, to the upper left and behind the observer, and is the colour of sunlight on a cloudy day.

The scene is composed of two primitive objects, a sphere and a polygon. The polygon is a rectangle whose vertices are listed. It has been translated and rotated about the x axis. The polygon is surface number one. The sphere is centered on (0.3,-0.18,0.0) and is surface number two.

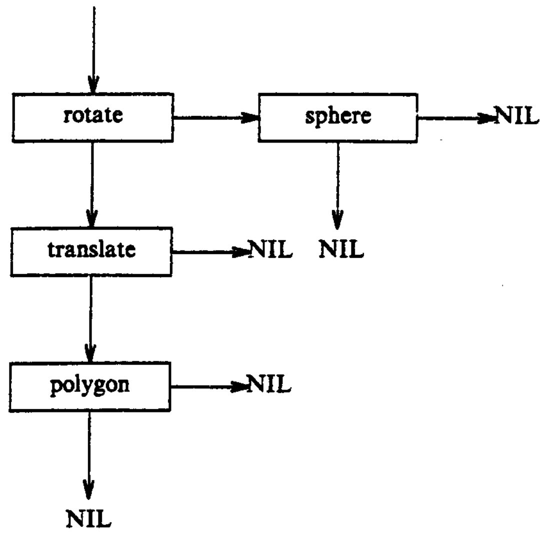


Figure 3. The corresponding DAG.

The corresponding DAG is given in Figure 3. This DAG is converted by the preprocessor into the linked list of object nodes given in Figure 4. Transformations are applied (cumulatively) to the nodes pointed to by the down pointers. Nodes pointed to by the right pointers are unaffected.

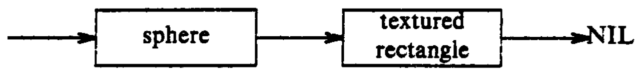


Figure 4. The corresponding object list.

Of course, both DAG nodes and object nodes contain much more information than presented here. They will be described in detail presently.

Finally, the ray tracing algorithm is run over the object list. 18 minutes later, the following picture appears.

Coloured image can be seen on page 163.

Figure 5. The final image produced (18:30 minutes).

This concludes all discussion of scene file structure, ReadScene, and transformations. For more details on transformations and how they are applied, see any of the standard graphics textbooks, for example [Foley82, Newman73]. The following sections examine the various primitive objects: how they are preprocessed and tested for intersection.

2. Spheres

The sphere offers the simplest ray-object intersection to compute [Kajiya83b]. Because of this, and no doubt because of the cover of the June 1980 CACM, almost every ray tracing program in existence can render spheres as primitive objects. This program is no exception.

2.1. Preprocessing

A sphere DAG node contains the following information (Figure 6) copied from the scene file.

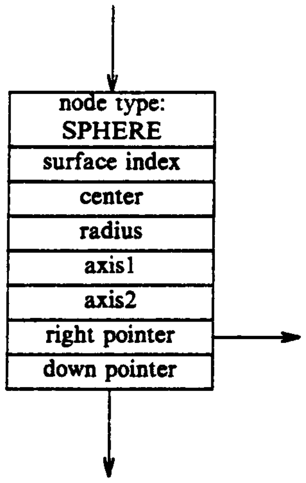


Figure 6. The DAG node associated with a sphere.

1. node type: an identification of the object described.

2. surface index: the number of the surface descriptor containing information on how light interacts with this surface.

3. center: the center point of the sphere.

4. radius: its radius

5. axis1, axis2: two axes which control the orientation of a texture map (if any).

6. right pointer: a pointer to another DAG node.

7. down pointer: a pointer to another DAG node.

Other than applying the cumulative transformation matrix to the center position, there is very little preprocessing done on a sphere. The package maintains the radius independent of scaling transformations. A ball with radius 0.1 will appear approximately 100 pixels wide.

A sphere object node contains the following information (Figure 7).

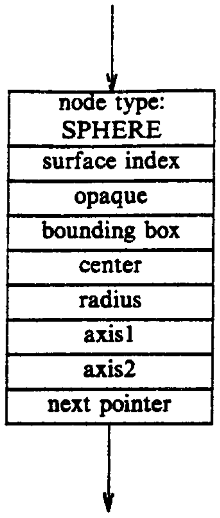


Figure 7. The object node associated with a sphere.

1. node type: an identification of the object described.

2. surface index: the number of the surface descriptor containing information on how light interacts with this surface, copied from the DAG node.

3. opaque: a boolean value set to TRUE if the associated surface descriptor has a transparency of 0.0.

4. bounding box: minimal rectilinear bounding box containing the sphere, calculated by the preprocessor.

5. center: the transformed center ($\vec{C}$) of the sphere.

6. radius: its radius squared (r^2).

7. axis1: one of the two axes ($\vec{A_1}$), copied from the DAG node.

8. axis2: the other axis ($\vec{A_2}$), copied from the DAG node.

9. next pointer: a pointer to another object node.

2.2. Intersection Testing

A ray is defined parametrically as

$$R(t) = \vec{O} + t\vec{D}$$

where $O(x,y,z)$ is the origin of the ray, and $D(x,y,z)$ its direction. The goal of an intersection calculation is to find the smallest t such that $R(t)$ is also a member of the surface being tested, if such t exists.

The sphere is defined by a center point $\vec{C}$ and a radius r. At the intersection point, the distance from the center to $R(t)$ is equal to the radius.

$$(O_x + tD_x - C_x)^2 + (O_y + tD_y - C_y)^2 + (O_z + tD_z - C_z)^2 = r^2$$

Let

$$\vec{\Lambda} = \vec{O} - \vec{C}$$

Substituting

$$(tD_x + \Lambda_x)^2 + (tD_y + \Lambda_y)^2 + (tD_z + \Lambda_z)^2 = r^2$$

and rearranging

$$(\Lambda_x^2 + \Lambda_y^2 + \Lambda_z^2 - r^2) + (2D_x\Lambda_x + 2D_y\Lambda_y + 2D_z\Lambda_z)t + (D_x^2 + D_y^2 + D_z^2)t^2 = 0$$

This is a quadratic equation in one variable, t, and is solved by the quadratic formula

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where

$$a = (D_x^2 + D_y^2 + D_z^2)$$
$$b = (2D_x\Lambda_x + 2D_y\Lambda_y + 2D_z\Lambda_z)$$
$$c = (\Lambda_x^2 + \Lambda_y^2 + \Lambda_z^2 - r^2)$$

If the ray misses the sphere entirely, the value of the discriminant $b^2 - 4ac$ will be less than zero, all calculation can halt and the sphere intersection routine returns failure. If the discriminant is greater than or equal to zero, there is at least one ray-sphere intersection, and the smallest $t > 0$ will usually be given by

$$t_i = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

It is possible for this t_i to be equal to zero (all reflected rays intersect the surface from which they were reflected with $t_i = 0$), or less than zero (if the surface is behind the point from which the ray was cast). If either of these are the case, then

$$t_i = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

If t_i calculated in this way is also less than or equal to zero, the sphere interestion routine returns failure.

Otherwise, from t_i the actual intersection point $\vec{I}$ can be calculated

$$\vec{I} = \vec{O} + t_i\vec{D}$$

The surface normal $\vec{N}$ at the intersection point is then

$$\vec{N} = \vec{I} - \vec{C}$$

(ie the vector from the sphere center to the intersection point). This vector must be normalized before being used by the shader.

2.3. Calculation of Texture Map Indices

Texture mapping is the projection of an external image onto some surface in the image being calculated (see Section 10). This external image may be a photograph which has been scanned in, or a pre-calculated, computer generated image.

In order to map a two dimensional image onto a three dimensional surface, you must be able to associate a (fairly) unique pair of indices (u,v), $0.0 \le u,v \le 1.0$, with every point on the surface. For spheres, this is trivial. The indices may be calculated from the surface normal $\vec{N}$ and the two axes $\vec{A}_1$ and $\vec{A}_2$ included in the sphere object node as follows:

$$u = \frac{(\vec{N} \cdot \vec{A}_1) + 1.0}{2.0}$$

$$v = \frac{(\vec{N}\cdot\vec{A}_2)+1.0}{2.0}$$

Notice that this method of calculating the indices repeats after 180 degrees. But this is seldom objectionable.

3. Cylinders

This package also treats cylinders as primitive objects. However, none of the other quadric surfaces (such as cones) are incorporated. If needed, these surfaces would be approximated by splines.

3.1. Preprocessing

A cylinder DAG node contains the following information (Figure 8) copied from the scene file. Most of the fields have already been described for a sphere DAG node in Section 2.1.

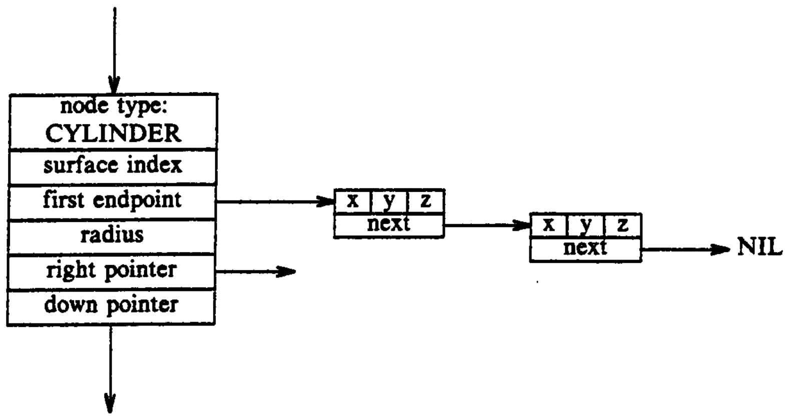


Figure 8. The DAG node associated with a cylinder.

1. first endpoint: a pointer to a linked list of two points defining a line segment which runs through the center of the cylinder.

2. radius: the cylinder radius.

As it turns out, the most convenient cylindrical form for the intersection processor is as a parametric line segment and a radius, so the two end points are put through the cumulative transformation matrix, and the line segment converted to parametric form:

$$C(u) = \vec{A}+u\vec{P}$$

where $0\leq u \leq 1$. $A(x,y,z)$ will be refered to as the cylinder anchor, and $P(x,y,z)$ as its path. As is the case for spheres, the radius is maintained independent of scaling transformations. A cylinder radius 0.1 will appear approximately 100 pixels wide.

One axis $\vec{A}_1$ perpendicular to $\vec{P}$ must be calculated for texture mapping. If $\vec{P}$ is not parallel to the z axis, then

$$\vec{A}_1 = \vec{P}\times(0.0,0.0,1.0)$$

otherwise

$$\vec{A}_1 = \vec{P}\times(0.0,1.0,0.0)$$

A cylinder object node contains the following information (Figure 9). Most of the fields have already been described for a sphere object node in Section 2.1.

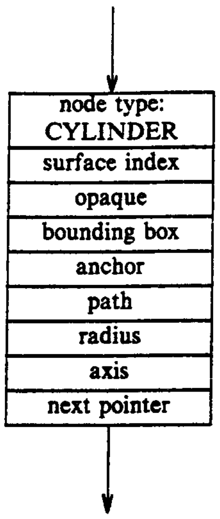


Figure 9. The object node associated with a cylinder.

1. anchor: the cylinder anchor ($\vec{A}$)

2. path: the cylinder path ($\vec{P}$).

3. radius: its radius squared (r^2).

4. axis: the axis for texture mapping ($\vec{A}_1$).

3.2. Intersection Testing

A ray is defined parametrically as

$$R(t) = \vec{O}+t\vec{D}$$

Again, the goal of an intersection calculation is to find the smallest t such that $R(t)$ is also a member of the surface being tested, if such t exists. The cylinder is defined by the parametric line segment

$$C(u) = \vec{A}+u\vec{P}$$

$0 \leq u \leq 1$ and a radius r. At the intersection point, the distance between $R(t)$ and $C(u)$ is equal to the radius.

$$\begin{aligned} &((O_x+tD_x)-(A_x+uP_x))^2 \\ &\quad+((O_y+tD_y)-(A_y+uP_y))^2 \\ &\quad+((O_z+tD_z)-(A_z+uP_z))^2 = r^2 \end{aligned} \tag{3.2.1}$$

Also, at the intersection point the vector $R(t)-C(u)$ must be perpendicular to the cylinder direction vector $\vec{P}$

$$\begin{aligned} &((O_x+tD_x)-(A_x+uP_x))P_x \\ &\quad+((O_y+tD_y)-(A_y+uP_y))P_y \end{aligned} \tag{3.2.2}$$

$$+((O_z+tD_z)-(A_z+uP_z))P_z = 0$$

Rearranging (3.2.2) to get u in terms of t.

$$(O_xP_x-A_xP_x+O_yP_y-A_yP_y+O_zP_z-A_zP_z)$$
$$+(D_xP_x+D_yP_y+D_zP_z)t$$
$$-(P_xP_x+P_yP_y+P_zP_z)u = 0$$

or $u = \alpha t+\beta$ where

$$\alpha = (D_xP_x+D_yP_y+D_zP_z)/(P_xP_x+P_yP_y+P_zP_z)$$
$$\beta = (O_xP_x-A_xP_x+O_yP_y-A_yP_y+O_zP_z-A_zP_z)/(P_xP_x+P_yP_y+P_zP_z)$$

Substituting $u = \alpha t+\beta$ into (3.2.1)

$$((O_x+tD_x)-(A_x+(\alpha t+\beta)P_x))^2$$
$$+((O_y+tD_y)-(A_y+(\alpha t+\beta)P_y))^2$$
$$+((O_z+tD_z)-(A_z+(\alpha t+\beta)P_z))^2 = r^2$$

Rearranging slightly

$$((O_x-A_x-\beta P_x)+(D_x-\alpha P_x)t)^2$$
$$+((O_y-A_y-\beta P_y)+(D_y-\alpha P_y)t)^2$$
$$+((O_z-A_z-\beta P_z)+(D_z-\alpha P_z)t)^2 = r^2$$

Let

$$\vec{\Lambda} = (\vec{O}-\vec{A}-\beta\vec{P})$$
$$\vec{\Omega} = (\vec{D}-\alpha\vec{P})$$

Substituting

$$(\Lambda_x+\Omega_xt)^2+(\Lambda_y+\Omega_yt)^2+(\Lambda_z+\Omega_zt)^2 = r^2$$

and rearranging

$$(\Lambda_x^2+\Lambda_y^2+\Lambda_z^2-r^2)+(2\Omega_x\Lambda_x+2\Omega_y\Lambda_y+2\Omega_z\Lambda_z)t+(\Omega_x^2+\Omega_y^2+\Omega_z^2)t^2 = 0$$

This is a quadratic equation in one variable, t, and is solved by the quadratic formula

$$t = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

where

$$a = (\Omega_x^2+\Omega_y^2+\Omega_z^2)$$
$$b = (2\Omega_x\Lambda_x+2\Omega_y\Lambda_y+2\Omega_z\Lambda_z)$$
$$c = (\Lambda_x^2+\Lambda_y^2+\Lambda_z^2-r^2)$$

If the ray misses the cylinder entirely, the value of the discriminant b^2-4ac will be less than zero, all calculation can halt and the cylinder intersection routine return failure. If the discriminant is greater than or equal to zero, there is a possible ray-cylinder intersection, and the smallest $t>0$ will usually be given by

$$t_i = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

As with spheres, it is possible that this value of t_i is less than or equal to zero. If this is the case, then

$$t_i = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

If t_i calculated in this way is also less than or equal to zero, the cylinder interestion routine returns failure. The value of

$$u_i = \alpha t_i + \beta$$

must also be checked. If u_i is outside the [0,1] interval, the routine returns failure.

Otherwise, from t_i the actual intersection point $\vec{I}$ can be calculated

$$\vec{I} = \vec{O} + t_i\vec{D}$$

and the surface normal $\vec{N}$ at the intersection point is

$$\vec{N} = \vec{I} - (\vec{A} + u_i\vec{P})$$

(i.e. the vector from the cylinder center at the level of the intersection to the point of intersection). This vector must be normalized before being used by the shader.

Notice that no attempt is made to intersect the ray with the ends of the cylinder. This package considers a cylinder to be an open object, rather than an object closed at both ends as is the case in most solid modelling programs.

3.3. Calculation of Texture Map Indices

This is even easier than for spheres. Having found that the ray does in fact hit the cylinder, and having calculated the surface normal $\vec{N}$ at the intersection point, we have

$$v = \frac{(\vec{N}\cdot\vec{A}_1)+1.0}{2.0}$$

where $\vec{A}_1$ is the axis included in the cylinder object node. The other index is just u_i.

4. Polygons

Another primitive object that nearly every ray tracing algorithm (and certainly every scan line algorithm) can handle is the convex, planar, non-degenerate polygon.

The key to our intersection method is the fact the polygon is planar. We first find the intersection point of the ray with the plane containing the polygon, then determine if this point is inside each polygon edge in turn.

It is easy to determine on which side of a plane a point falls. The point-normal definition of the plane $p(x,y,z)$ is

$$(\vec{p}-\vec{P})\cdot\vec{N} = 0$$

where $P(x,y,z)$ is any point in the plane, and $N(x,y,z)$ is a normal to the plane. Then if $q(x,y,z)$ is any point in three space, the value of

$$(\vec{q}-\vec{P})\cdot\vec{N} \tag{4.1}$$

will be positive if $\vec{q}$ is on the same side of the plane as the normal points, negative otherwise.

A slight saving in storage (and computation time) can be gained as follows. Expanding 4.1

$$q_xN_x - P_xN_x + q_yN_y - P_yN_y + q_zN_z - P_zN_z$$

and letting

$$d = P_xN_x + P_yN_y + P_zN_z$$

the test becomes

$$(\vec{q}\cdot\vec{N}) - d$$

To repeat, this value is positive if q is "in" with respect to $\vec{N}$, negative otherwise.

The test can be adapted to determine on which side of a polygon edge a point falls, simply by making $\vec{N}$ lie in the plane of the polygon, and making $\vec{P}$ be a vertex point.

4.1. Preprocessing

A polygon DAG node contains the following information (Figure 10) copied from the scene file. Most of the fields have already been described for a sphere DAG node in Section 2.1.

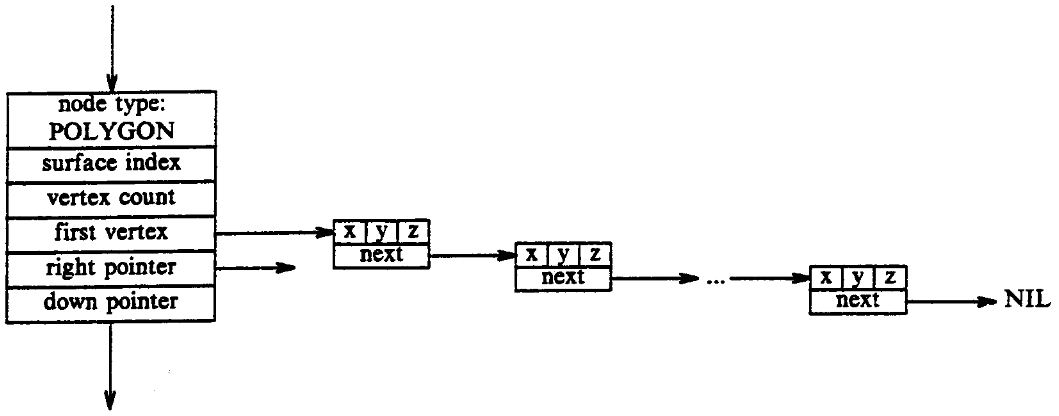


Figure 10. The DAG node associated with a polygon.

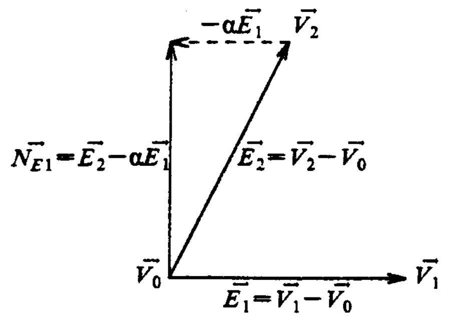
1. vertex count: the number of vertices in the polygon.

2. first vertex: a pointer to the vertices themselves, in a linked list.

The problem then, is to translate a collection of n polygon vertices, into n pairs of edge normals and d values. In fact, only triangles and rectangles can be rendered at present. However, trivial modification would allow more general polygons.

Consider a triangle with vertices $\vec{V_0}$, $\vec{V_1}$, and $\vec{V_2}$. These vertices are local copies (vertices may belong to more than one object) which have been put through the cumulative transformation matrix.

A normal $\vec{N_{E1}}$ to the first edge $\vec{E_1}$ which is also in the plane of the triangle can be found as follows (Figure 11).



$-\alpha\vec{E}_1$
$\vec{V}_2$
$\vec{N}_{E1}=\vec{E}_2-\alpha\vec{E}_1$
$\vec{E}_2=\vec{V}_2-\vec{V}_0$
$\vec{V}_0$
$\vec{V}_1$
$\vec{E}_1=\vec{V}_1-\vec{V}_0$

Figure 11. Calculation of an edge normal.

Here α is chosen to make $\vec{E}_1$ and $\vec{N}_{E1}$ perpendicular, i.e.

$$\alpha = \frac{\vec{E}_1 \cdot \vec{E}_2}{\vec{E}_1 \cdot \vec{E}_1}$$

and the first d value can be calculated from the line normal and any point on the line (e.g. $\vec{V}_0$).

$$d_1 = \vec{N}_{E1} \cdot \vec{V}_0$$

Normals to the other two edges and d values are obtained by permuting the points.

In the introduction to polygons it was mentioned that we first find the intersection of the ray with the plane containing the polygon. To accomplish this two other pieces of information are necessary: the normal to the plane and a point in the plane. The polygon normal can be calculated as

$$\vec{N} = \vec{N}_{E1} \times \vec{N}_{E2}$$

and any vertex, for instance $\vec{V}_0$, can be chosen as the point.

Also, notice that any point $\vec{q}$ in the polygon may be expressed as

$$\vec{q} = \vec{V}_0 + u\vec{E}_1 + v\vec{N}_{E1}$$

for some choice of (u,v). This (u,v) pair may be used as indices for texture mapping. However, the final texture map indices must be in the range [0.0,1.0], so the following equation is solved using, in turn, each transformed vertex $\vec{V}_n$.

$$\vec{V}_n = \vec{V}_0 + u\vec{E}_1 + v\vec{N}_{E1}$$

These are three equations in two unknowns, (u,v), and are solved immediately. The maximum and minimum u and v are accumulated, and stored in the polygon object node. These values are used to range check the texture map indices calculated by the intersection routine (Section 4.3).

A triangle object node contains the following information (Figure 12). Many of the fields have already been described for a sphere object node in Section 2.1.

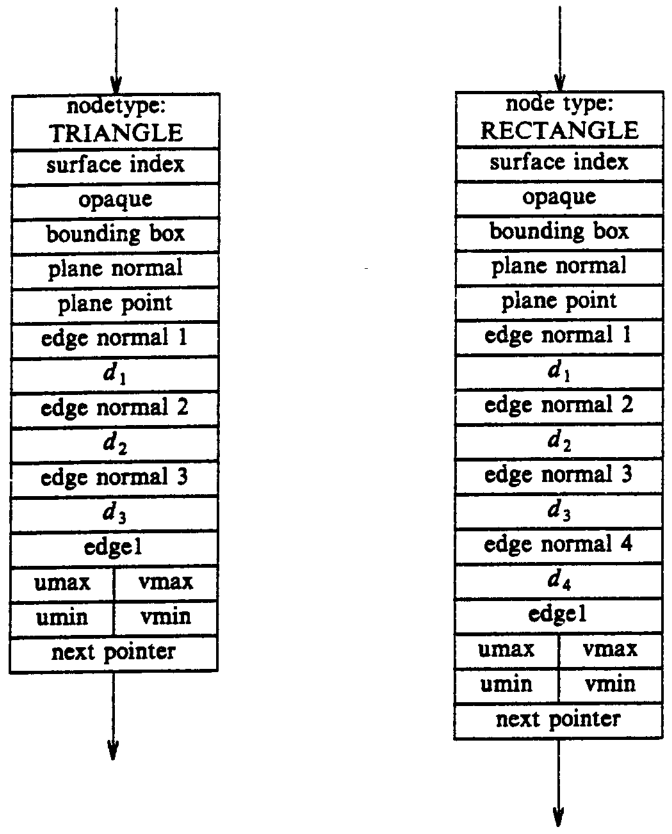


Figure 12. The object nodes associated with polygons.

1. plane normal: the normal to the plane of the triangle ($\vec{N}$).

2. plane point: a point in the plane of the triangle ($\vec{V_0}$).

3. the three sets of edge normals ($\vec{N_{E1}}$, $\vec{N_{E2}}$, $\vec{N_{E3}}$) and the associated d values.

4. edgel: one edge ($\vec{E_1}$) for texture mapping.

5. umax: the maximum possible u value for texture mapping ($u_{\max}$).

6. vmax: the maximum possible v value for texture mapping ($v_{\max}$).

7. umin: the minimum possible u value for texture mapping ($u_{\min}$).

8. vmin: the minimum possible v value for texture mapping ($v_{\min}$).

The preprocessing for a rectangle is similar, except that one more edge normal and d must be calculated.

4.2. Intersection Testing

Intersection with triangles will be discussed. The procedure for rectangles is identical, except than one extra test is performed with the fourth edge normal.

A ray is defined parametrically as

$$R(t) = \vec{O} + t\vec{D}$$

The triangle is defined by a plane $p(x,y,z)$ in point-normal form

$$(\vec{p} - \vec{V_0}) \cdot \vec{N} = 0$$

and three pairs of edge normals and d values. At the ray-plane intersection

$$(R(t_i) - \vec{V_0}) \cdot \vec{N} = 0$$

This is an equation in one variable, t_i, and is solved immediately. If $t_i \leq 0$, the triangle intersection routine returns failure. Otherwise, the ray-plane intersection point is

$$\vec{I} = \vec{O} + t_i\vec{D}$$

It remains to determine if this intersection point is in the interior of the triangle. This can be determined by three tests in succession. If any one of these tests fail, the calculation can halt, and the intersection routine returns failure. The tests are

$$(\vec{I} \cdot \vec{N_{E1}}) - d_1 \geq 0$$
$$(\vec{I} \cdot \vec{N_{E2}}) - d_2 \geq 0$$
$$(\vec{I} \cdot \vec{N_{E3}}) - d_3 \geq 0$$

Notice that if all of the tests are passed, the surface normal (plane normal) is available as a part of the triangle object node.

4.3. Calculation of Texture Map Indices

Having found that the ray does in fact hit the polygon, and having calculated the actual intersection point $\vec{I}$, texture map indices (u,v) may be obtained as follows:

$$\vec{I} = \vec{V_0} + u\vec{E_1} + v\vec{N_{E1}}$$

where $\vec{V_0}$, $\vec{E_1}$, and $\vec{N_{E1}}$ are all included in the polygon object node. These are three equations in two unknowns, and are solved immediately.

These values must then be mapped to the range [0,1].

$$u = \frac{u - u_{\min}}{u_{\max} - u_{\min}}$$
$$v = \frac{v - v_{\min}}{v_{\max} - v_{\min}}$$

where again $u_{\max}$, $u_{\min}$, $v_{\max}$ and $v_{\min}$ are included in the polygon object node.

5. Fractal Surfaces

A recurrent problem in computer graphics is making the visual complexity of the real world somehow computationally tractable. Fractal surfaces can be used to represent a wide class of irregular objects to arbitrary detail from small data bases. Loren Carpenter's mountains are good examples [Carpenter80].

Fractals originate from work by Benoit Mandelbrot on one-dimensional Gaussian stochastic processes [Mandelbrot77]. His curves were statistically self-similar, that is, a portion of the curve magnified to any degree had the same statistical properties as the whole curve. This is exactly what you want to be able to render an irregular surface at arbitrary magnification.

Mandelbrot's methods were dependent on generating sequences of random numbers, viewed as data in a time domain, converting them into a frequency domain by Fourier transform, adjusting the frequencies to approximate a "1/f" spectrum, and finally back transforming the result into the time domain. This technique is extremely time intensive.

However, Fournier, Fussel and Carpenter have produced an adaptive subdivision procedure which gives similar results [Fournier82]. Their method to generate an "inferior fractal" (Mandelbrot's term) is simple. Start with a triangle. Recursively subdivide it into 4 smaller triangles, randomly perturbing the y component of the subdivision corners according to

$$s \times 2^{-hl} \times G$$

where s is a scale factor, h is the "fractal dimension", l is the level of recursion and G is a Gaussian random number with zero mean and unit variance. Halt the recursion when the desired amount of detail has been generated.

Coloured image can be seen on page 163.

Figure 13. The original triangle (13:49 minutes).

Coloured image can be seen on page 163.

Figure 14. After 1 subdivision (22:45 minutes).

Coloured image can be seen on page 163.

Figure 15. After 2 subdivisions (26:37 minutes).

Coloured image can be seen on page 163.

Figure 16. After 4 subdivisions (32:53 minutes).

Coloured image can be seen on page 163.

Figure 17. After 5 subdivisions (36:39 minutes)

Coloured image can be seen on page 164.

Figure 18. After 10 subdivisions (54:06 minutes)

An interesting fractal scene may contain hundreds of thousands of triangles, so ray tracing them directly is out of the question. Kajiya [Kajiya83a] has reported a method for finding the intersection of rays with fractals and other surfaces he calls ***height fields***. His algorithm has the nice property that it correctly handles surfaces which intersect rays at more than one spot. The algorithm is not limited to height fields, but can be applied to any open or closed three dimensional surface (e.g. free-form B-spline surfaces, Section 6).

We render fractals by Kajiya's method, with three modifications. First, we use a rectilinear bounding box as the extent, rather than Kajiya's pie shape. Second, using the box extents, the virtual address space on our VAX is large enough to hold the subdivision tree fully instantiated whereas Kajiya generates the tree on the fly. And third, we maintain the list of active nodes sorted in order of increasing distance.

5.1. Preprocessing

Fractals are the most volatile objects in the package. It seems that almost every scene we do with them requires some change to the preprocessor. What is described below is the package as it stands April, 1984.

A fractal DAG node contains the following information (Figure 19) copied from the scene file. Most of the fields have already been described for a sphere DAG node in Section 2.1.

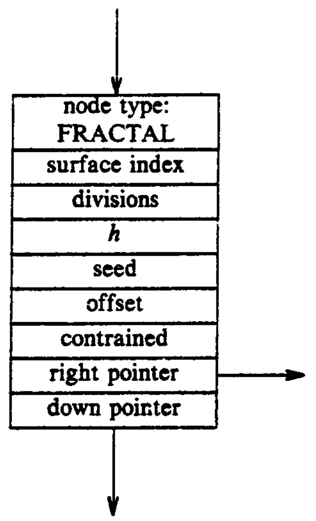


Figure 19. The DAG node associated with a fractal.

1. divisions: the number of subdivisions to be done to generate the surface.

2. *h*: the fractal dimension.

3. seed: a seed for the Gaussian random number generator.

4. offset: an offset which is added to every entry in the matrix of random numbers. This effectively becomes the mean of the Gaussian distribution

5. constrained: a boolean value. If constrained is TRUE, the perimeter of the fractal surface will remain in the (possibly transformed) plane of the original triangle. Otherwise, the perimeter is subject to the same random variations as the rest of the surface.

5.2. Tree Construction

The purpose of the fractal preprocessor is to produce a tree of nested rectilinear bounding boxes, whose leaf nodes contain the randomly oriented triangular facets of the fractal surface.

The surface is generated by the recursive algorithm given in [Fournier82, page 376] adapted to three dimensions. Briefly, at each level of the recursion, the current triangle is subdivided into four smaller triangles by joining the midpoints of its sides. A random perturbation of the y component of the the midpoints is added according to

$$s \times 2^{-hl} \times G$$

where the variables are defined as before.

A standard triangle is defined in the preprocessor by 3 vertices $(-1.0,0.0,1.0)$, $(1.0,0.0,1.0)$ and $(0.0,0.0,-1.0)$. It is this same triangle which is always subdivided - the current transformation matrix is applied to the terminal triangles of the subdivision process. This ensures "internal consistency" [Fournier82], that is, independence of the object's appearance on its orientation or distance.

The same point can come up several times at the same level of recursion. The same perturbation must be applied to this point each time, otherwise tears will appear in the surface. This is refered to in [Fournier82] as "external consistency". In his SIGGRAPH'83 tutorial [Whitted83], Whitted mentioned that a solution to this problem was to use the x and z coordinates of the midpoints as indices into a hash table of Gaussian random values. Our solution is similar. We use x and z coordinates as direct indices into a 201×201 matrix of random values to obtain G. This is possible because both x and z are guaranteed to be in the range [-1.0,1.0]. 40401 numbers appear to be a wide enough sample that regularities do not become appearant in the surface.

At each level of the recursion the fractal generation procedure also allocates a node of the tree of bounding boxes and connects it to the four nodes to be allocated at the next level.

The recursion terminates when the user defined number of subdivisions has been reached. The current triangle corners are put through the transformation matrix, and a point ($\vec{A}$) and two vectors ($\vec{U}$, $\vec{V}$) are calculated so that the transformed triangle is given by

$$F(u,v) = \vec{A} + u\vec{U} + v\vec{V}$$

where $0.0 \le u \le 1.0$ and $0.0 \le v \le u$. This information is incorporated into the tree node just allocated, which becomes a leaf node. A minimal rectilinear bounding box is also calculated from the three triangle corners.

As the procedure returns through the recursion, the parent nodes are tagged as internal nodes and ever larger bounding boxes are calculated to contain the bounding boxes of the children.

5.3. Preprocessing for Texture Mapping

A fractal is texture mapped as if it were a triangle, temporarily ignoring the y coordinate. The three vertices $(-1.0,0.0,1.0)$, $(1.0,0.0,1.0)$ and $(0.0,0.0,-1.0)$ are put through the current transformation matrix, producing the three transformed vertices $\vec{V}_0$, $\vec{V}_1$ and $\vec{V}_2$. From these transformed vertices, an edge ($\vec{E}_1$) and an edge normal ($\vec{N}_{E1}$) are calculated exactly as for triangles (Figure 11)

This time, it is **not** true that any point $\vec{q}$ in the fractal surface can be expressed as

$$\vec{q} = \vec{V}_0 + u\vec{E}_1 + v\vec{N}_{E1}$$

for some choice of (u,v). However, unless the fractal is rotated so that $\vec{V}_0$, $\vec{V}_1$ and $\vec{V}_2$ all lie in the x-y or the z-y plane, the system of equations

$$q_x = V_{0x} + uE_{1x} + vN_{E1x}$$
$$q_z = V_{0z} + uE_{1z} + vN_{E1z}$$

will have a solution for (u,v). This (u,v) pair may be used as indices for texture mapping.

Again, the final texture map indices must be in the range [0.0,1.0], so the following equations are solved using, in turn, each transformed facet vertex $\vec{V}_n$.

$$V_{nx} = V_{0x} + uE_{1x} + vN_{E1x}$$
$$V_{nz} = V_{0z} + uE_{1z} + vN_{E1z}$$

The maximum and minimum u and v are accumulated, and stored in the polygon object node. These values are used to range check the texture map indices calculated by the intersection routine (Section 5.7).

5.4. The Fractal Object Node

A fractal object node contains the following information (Figure 20). Many of the fields have already been described for a sphere object node in Section 2.1.

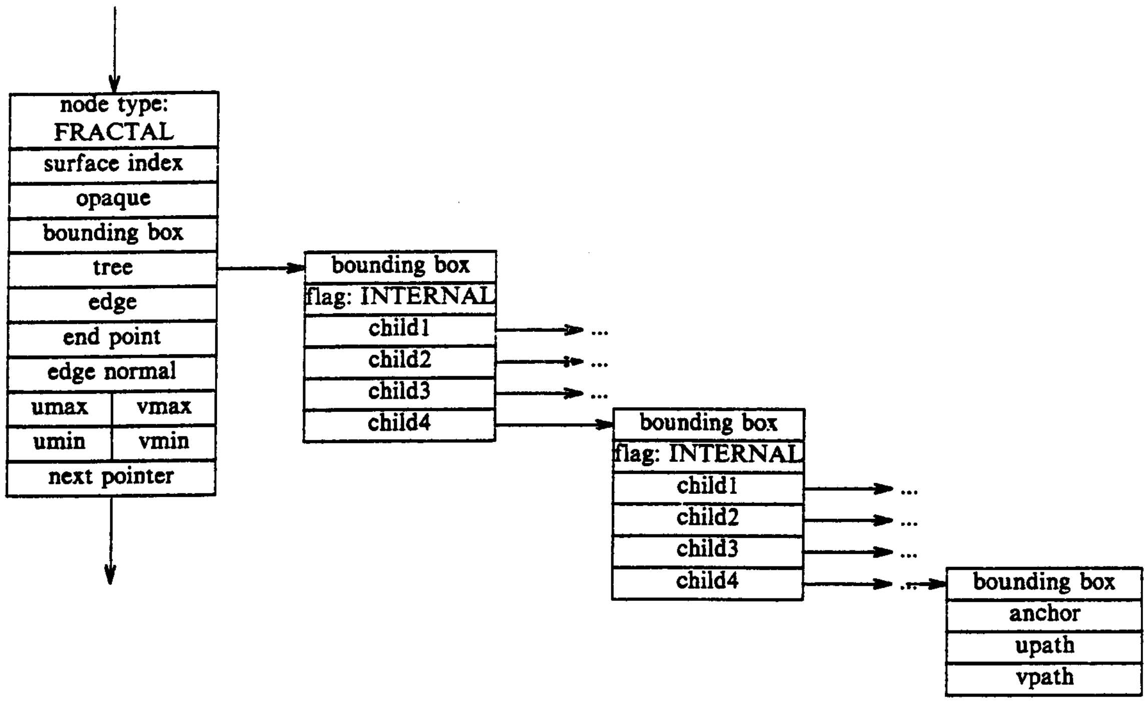


Figure 20. The object node associated with a fractal.

1. tree: a pointer to the root of the tree of bounding boxes generated as above.

2. edge: one edge ($\vec{E}_1$) of the possibly transformed original triangle.

3. end point: an end point of the edge ($\vec{V}_0$).

4. edge normal: a normal to this edge ($\vec{N}_{E1}$).

5. umax: the maximum possible u value for texture mapping ($u_{\max}$).

6. vmax: the maximum possible v value for texture mapping ($v_{\max}$).

7. umin: the minimum possible u value for texture mapping ($u_{\min}$).

8. vmin: the minimum possible v value for texture mapping ($v_{\min}$).

An internal node of the tree of nested rectilinear bounding boxes contains the following information:

1. bounding box: the box.

2. flag: an indentification that this node is in fact internal to the tree.

3. 4 pointers to descendants.

And finally, a leaf node of the tree of nested bounding boxes contains the following information:

1. bounding box: the box.

2. anchor: one vertex ($\vec{A}$) of the triangular facet of the fractal surface.

3. upath, vpath: two edges ($\vec{U}$, $\vec{V}$) of the facet of the fractal surface.

The intersection would go faster if a fractal facet were defined as a triangle polygon (by three edge normals, three d values, a plane normal and a point). But that would almost double the size of the already large fractal object node.

As it is, the memory requirements of a fractal surface are largely determined by the size of the tree of bounding boxes, and this is dictated, in turn, by the number of subdivisions used to generate the surface. Requirements for various levels of subdivision are shown in Table 1.

divisions	number of facets	size (bytes)
3	64	8640
4	256	25200
5	1024	89840
6	4096	334800
7	16384	1319040
8	65536	5252400
9	262144	20982240

Table 1. Memory Requirements for Various Surfaces.

5.5. Intersection Testing

We use Kajiya's algorithm, as applied to fully instantiated subdivision trees [Kajiya83, page 178]. Briefly, for each ray we maintain a linked list of active nodes. Attached to those nodes are various subtrees of the tree of bounding boxes built by the preprocessor. With each node is associated a distance from the ray origin to the closest intersection with the bounding box of the root of the attached subtree. We maintain the list of active nodes sorted by increasing distance.

The algorithm proceeds as follows:

- Choose the first (closest) node on the active node list, and remove it.
- If the root of the attached subtree is interior to the tree consider in turn each of its four children.
- If the ray hits the bounding box of a child, then attach the child to an active node, and sort the node into the the active node list.
- If the root of the attached subtree is a leaf, attempt to intersect the ray with the fractal facet contained.

The algorithm terminates when the active node list is empty (failure), or the distance to the fractal surface, as returned by the facet intersection routine, is less than the distance to the first (closest) node on the active node list (success).

5.6. Facet Intersection Testing

A ray is defined parametrically as

$$R(t) = \vec{O}+t\vec{D}$$

A fractal facet is defined by a point and two vectors

$$F(u,v) = \vec{A}+u\vec{U}+v\vec{V}$$

where $0.0 \leq u \leq 1.0$ and $0.0 \leq v \leq u$. At the intersection point

$$\vec{O}+t_i\vec{D} = \vec{A}+u_i\vec{U}+v_i\vec{V}$$

These are three equations in three unknowns t_i, u_i and v_i, which are solved by Cramer's rule. If the system does in fact have a solution, and the conditions on u_i and v_i are satisfied, then the actual intersection point is

$$\vec{I} = \vec{O}+t_i\vec{D}$$

the surface normal of the triangle (and so, of the fractal surface) at the intersection point is

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

and the distance to the fractal surface is, of course, just t_i.

5.7. Calculation of Texture Map Indices

Having found that the ray does in fact hit the fractal, and having calculated the actual intersection point $\vec{I}$, texture map indices (u,v) may be obtained as follows:

$$I_x = V_{0x}+uE_{1x}+vN_{E1x}$$

$$I_z = V_{0z}+uE_{1z}+vN_{E1z}$$

where $\vec{V_0}$, $\vec{E_1}$, and $\vec{N_{E1}}$ are all included in the fractal object node. These are two equations in two unknowns, and are solved immediately.

These values must then be mapped to the range [0,1].

$$u = \frac{u-u_{\min}}{u_{\max}-u_{\min}}$$

$$v = \frac{v-v_{\min}}{v_{\max}-v_{\min}}$$

where again $u_{\max}$, $u_{\min}$, $v_{\max}$ and $v_{\min}$ are included in the fractal object node.

6. Splines

The representation of curved surfaces has always been the anathema of computer graphics. The usual solution in the past has been to subdivide the surface until the component patches are flat enough to be reasonably represented by planar polygons [Catmull74], possibly with some shading tricks [Gouroud71, Phong75] to disguise their planar nature. It is only recently that some attempt has been made to render parametric patches or whole free form surfaces directly [Schweitzer82, Kajiya82a, Whitted in Lane80]. These methods have the advantages over a polygon approximation of being resolution independent, more accurate, and usually faster.

B-spline representations of curved surfaces are especially attractive for several reasons. First, a single B-spline surface is capable of representing a great variety of shapes, both closed and open. Second, such shapes may be defined by a very limited number of control points. Third, they have the property of local control. That is, movement of a control point affects only a very limited portion of the whole surface. These three reasons, taken together, make B-splines surfaces ideal for free-form modelling applications.

Up to now, the only way to ray trace a B-spline surface has been to break it into component bicubic patches and trace the patches individually by one of the methods mentioned in the introduction (Section 1.1). Here we present a new algorithm for intersecting rays with B-spline surfaces, based on the recurrence properties of B-splines [Riesenfeld80] and on the fractal intersection algorithm of Kajiya [Kajiya83a].

We consider surfaces constructed as B-spline weighted control graphs and render them with the aid of two preprocessing steps. First, the control graph is refined to produce local information about the surface suitable for use in starting Newton's iteration. Second, a tree of nested rectilinear bounding boxes is built on top of the refined vertices. The intersection processing itself involves the same hierarchical testing as was used for fractals, and ends with 2 to 3 Newton iterations (on the average) per ray strike.

6.1. Some Preliminaries

B-spline representation techniques to construct curves and surfaces for use in interactive computer graphics are derived from the work of [Riesenfeld73]. In this section we present an overview of these techniques, and introduce our notation. A more thorough treatment can be found in [Bartels83].

Let $\bar{u}$ be a parameter in the range $-\infty < \bar{u} < +\infty$, and let a sequence of distinguished values called *knots* be given,

$$\bar{u}_{-4}, \ldots, \bar{u}_{-1}, \bar{u}_0, \bar{u}_1, \ldots, \bar{u}_m, \bar{u}_{m+1}, \ldots, \bar{u}_{m+4}$$

with $\bar{u}_i < \bar{u}_{i+1}$ for each i. A cubic spline curve is a parametarized curve $(x(\bar{u}), y(\bar{u}), z(\bar{u}))$ with the properties that

(A) each of $x(\bar{u})$, $y(\bar{u})$, and $z(\bar{u})$ is a cubic polynomial in the parameter $\bar{u}$ on any of the intervals $\bar{u}_i \leq \bar{u} < \bar{u}_{i+1}$

(B) these cubic polynomials are tied together so that each of $x(\bar{u})$, $y(\bar{u})$, and $z(\bar{u})$ is twice continuously differentiable on the full $\bar{u}$ range. As a result, the curve $(x(\bar{u}), y(\bar{u}), z(\bar{u}))$ will be continuous and will have continuous tangent and curvature.

Cubic spline curves can be represented as a linear combination of basis functions known as *cubic B-splines* on any parametric range $[umin, umax)$ provided that $\bar{u}_{-1} \leq umin < umax \leq \bar{u}_{m+1}$. The cubic B-spline (*with support on* $\bar{u}_i, \ldots, \bar{u}_{i+4}$) is the piecewise cubic function the form

$$B_{i,4}(\bar{u}) = \begin{cases} 0 & \bar{u} < \bar{u}_i \\ b_{i,s}(\bar{u}) & \bar{u}_{i+s} \leq \bar{u} < \bar{u}_{i+s+1} \quad s = 0,1,2,3 \\ 0 & \bar{u}_{i+4} < \bar{u} \end{cases}$$

where each $b_{i,s}(\bar{u})$ is a cubic polynomial

$$b_{i,s}(\bar{u}) = c_{i,s,3} \cdot \bar{u}^3 + c_{i,s,2} \cdot \bar{u}^2 + c_{i,s,1} \cdot \bar{u} + c_{i,s,0}$$

and the 16 coefficients $c_{i,s,r}$ are chosen so that

(A) $B_{i,4}(\bar{u})$ is continuous and has two continuous derivatives for all $\bar{u}$;

(B) $B_{i,4}(\bar{u}) > 0$ for $\bar{u}_i < \bar{u} < \bar{u}_{i+4}$;

(C) $B_{i,4}(\bar{u}) + B_{i-1,4}(\bar{u}) + B_{i-2,4}(\bar{u}) + B_{i-3,4}(\bar{u}) = 1$
for all $i = -1, \ldots, m$ and all $\bar{u}_i \leq \bar{u} < \bar{u}_{i+1}$.

For example:

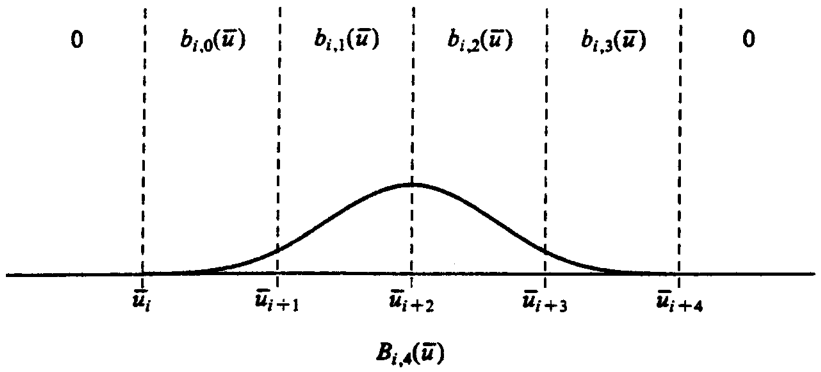


Figure 21. Graph of a typical cubic B-spline.

The easiest method of using cubic B-splines to construct splines is to employ them as *weighting functions*. If any collection of points $\mathbf{U}_i$ (called *control vertices*) are chosen for $i = -4, \ldots, m$, then the function

$$\mathrm{Q}(\bar{u}) = \sum_{i=-4}^{m} \mathbf{U}_i B_{i,4}(\bar{u})$$

will trace out a curve as $\bar{u}$ runs from *umin* to *umax* for any interval $[\mathit{umin}, \mathit{umax})$ as above. By selecting the interval $[\mathit{umin}, \mathit{umax})$ with care, and by not restricting the points $\mathbf{U}_i$ to be unique, many kinds of closed and open curves can be created [Barsky82, Barsky83].

If another parameter $\bar{v}$ is considered, with knots $\bar{v}_{-4}, \ldots, \bar{v}_{n+4}$ and with B-splines $B_{j,4}(\bar{v})$, and if a doubly indexed collection of control vertices $\mathbf{U}_{i,j}$ is arranged, then the 2 parameter function

$$\mathrm{Q}(\bar{u},\bar{v}) = \sum_{i=-4}^{m} \sum_{j=-4}^{n} \mathbf{U}_{i,j} B_{i,4}(\bar{u}) B_{j,4}(\bar{v}) \tag{6.1.1}$$

will define a surface for $\bar{u},\bar{v}$ running through any intervals $[\mathit{umin}, \mathit{umax})$ and $[\mathit{vmin}, \mathit{vmax})$ satisfying

$$\begin{aligned} \bar{u}_{-1} &\le \mathit{umin} \le \bar{u} < \mathit{umax} \le \bar{u}_{m+1} \\ \bar{v}_{-1} &\le \mathit{vmin} \le \bar{v} < \mathit{vmax} \le \bar{v}_{n+1} \end{aligned} \tag{6.1.2}$$

As in the case of curves, the surface defined by Q will have continuous tangents and curvature.

Taken together, (B) and (C) above imply the *convex hull property*: each point on the surface $\mathbf{P} = \mathrm{Q}(\bar{u},\bar{v})$ for any fixed $\bar{u}_i \le \bar{u} < \bar{u}_{i+1}$ and fixed $\bar{v}_j \le \bar{v} < \bar{v}_{j+1}$ lies in the convex hull of 16 control vertices

$$\begin{matrix} \mathbf{U}_{i-3,j-3} & \mathbf{U}_{i-3,j-2} & \mathbf{U}_{i-3,j-1} & \mathbf{U}_{i-3,j} \\ \mathbf{U}_{i-2,j-3} & \mathbf{U}_{i-2,j-2} & \mathbf{U}_{i-2,j-1} & \mathbf{U}_{i-2,j} \\ \mathbf{U}_{i-1,j-3} & \mathbf{U}_{i-1,j-2} & \mathbf{U}_{i-1,j-1} & \mathbf{U}_{i-1,j} \\ \mathbf{U}_{i,j-3} & \mathbf{U}_{i,j-2} & \mathbf{U}_{i,j-1} & \mathbf{U}_{i,j} \end{matrix}$$

We note here for future reference that the point $Q(\bar{u},\bar{v})$ on the surface which can be expected to come closest to $\mathbf{U}_{i,j}$ is that which corresponds to $\bar{u} = \bar{u}_{i+2}$, $\bar{v} = \bar{v}_{j+2}$. The reason for this shift of 2 is visible from Figure 21: since $B_{i,4}(\bar{u}) \cdot B_{j,4}(\bar{v})$ weights $\mathbf{U}_{i,j}$, the point on the surface showing the strongest influence of $\mathbf{U}_{i,j}$ is the one for which $\bar{u} = \bar{u}_{i+2}$, and similarly for $\bar{v}$.

The B-splines which arise in the simple case for which the difference $\bar{u}_{i+1} - \bar{u}_i$ is the same for all i are known as the ***uniform cubic B-splines***. In this case, the segment polynomials $b_{i,s}(\bar{u})$ are the same for all $B_{i,4}(\bar{u})$. Reparameterised to the unit interval they are:

$$\begin{aligned}
b_{i,0}(u) &= \frac{1}{6}u^3 \\
&\quad \text{for } 0 \le u < 1 \text{ and } u = (\bar{u}-\bar{u}_i)/(\bar{u}_{i+1}-\bar{u}_i) \\
b_{i,1}(u) &= \frac{1}{6}(1+3u+3u^2-3u^3) \\
&\quad \text{for } 0 \le u < 1 \text{ and } u = (\bar{u}-\bar{u}_{i+1})/(\bar{u}_{i+2}-\bar{u}_{i+1}) \\
b_{i,2}(u) &= \frac{1}{6}(4-6u^2+3u^3) \\
&\quad \text{for } 0 \le u < 1 \text{ and } u = (\bar{u}-\bar{u}_{i+2})/(\bar{u}_{i+3}-\bar{u}_{i+2}) \\
b_{i,3}(u) &= \frac{1}{6}(1-3u+3u^2-u^3) \\
&\quad \text{for } 0 \le u < 1 \text{ and } u = (\bar{u}-\bar{u}_{i+3})/(\bar{u}_{i+4}-\bar{u}_{i+3})
\end{aligned} \tag{6.1.3}$$

It is these polynomials which we are currently using.

6.2. Preprocessing

A spline DAG node contains the following information (Figure 22) copied from the scene file. Most of the fields have already been described for a sphere DAG node in Section 2.1.

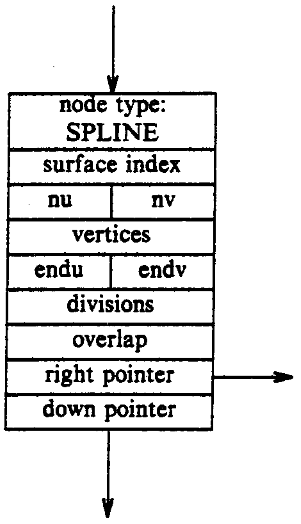


Figure 22. The DAG node associated with a spline.

1. nu: the number of control points in the u parametric direction.

2. nv: the number of control points in the v parametric direction.

3. vertices: an nu $\times$ nv matrix of control vertices.

4. endu: a flag indicating the end condition to be applied in the u parametric direction (single, double, triple or closed).

5. endv: a flag indicating the end condition to be applied in the v parametric direction (single, double, triple or closed).

6. divisions: the number of segments into which each knot interval is divided by the refinement preprocessor.

7. overlap: used in the tree building phase (see Section 6.5).

6.3. Refinement

The first step in preprocessing a spline surface involves replacing its representation as a weighted average of the given control vertices (which have been put through the current transformation matrix) by a representation as an average of more control vertices which lie closer to the surface.

Riesenfeld et. al. have considered the problem of taking a surface $Q(\bar{u},\bar{v})$, generated in terms of one set of control vertices $U_{-4,-4}, \ldots, U_{m,n}$, and representing it in terms of a larger set $W_{-4,-4}, \ldots, W_{p,q}$. This refinement process is presented in [Riesenfeld80] as the *Oslo Algorithm*. Briefly, since each control vertex defining $Q(\bar{u},\bar{v})$ is weighted by a product of B-splines $B_{i,4}(\bar{u})B_{j,4}(\bar{v})$ and since each of these B-splines in turn is defined by knots $\bar{u}_i, \ldots, \bar{u}_{i+4}$ and $\bar{v}_j, \ldots, \bar{v}_{j+4}$, it follows that more knots must be introduced in order to accommodate more control vertices. Restricting the discussion to $\bar{u}$ alone, consider taking the sequence $\bar{u}_{-4}, \ldots, \bar{u}_{m+4}$ and adding extra knots to produce the finer sequence $\bar{w}_{-4}, \ldots, \bar{w}_{p+4}$. We want $p > m$ and $\{\bar{u}_{-4}, \ldots, \bar{u}_{m+4}\} \subset \{\bar{w}_{-4}, \ldots, \bar{w}_{p+4}\}$. Because of a mathematical technicality, however, we must require that $\bar{w}_{-4} = \bar{u}_{-4}, \ldots, \bar{w}_{-1} = \bar{u}_{-1}$ and $\bar{w}_{p+1} = \bar{u}_{m+1}, \ldots, \bar{w}_{p+4} = \bar{u}_{m+4}$. These refined knots will define a superspace of spline functions whose basis B-splines, $C_{i,4}(\bar{u})$, can be used to represent the original B-splines:

$$B_{i,4}(\bar{u}) = \sum_{r=-4}^{p} \alpha_{i,4}(r) C_{r,4}(\bar{u}) \tag{6.3.1}$$

The numbers $\alpha_{i,4}(r)$ are the entries of the *basis representation matrix*. Riesenfeld et. al. [Riesenfeld80] show that the α's satisfy the recurrence

$$\alpha_{i,1}(r) = \begin{cases} 1 & \bar{u}_i \leq \bar{w}_r < \bar{u}_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\alpha_{i,l}(r) = \frac{(\bar{u}_{i+l} - \bar{w}_{r+l-1})}{(\bar{u}_{i+l} - \bar{u}_{i+1})} \alpha_{i+1,l-1}(r) + \frac{(\bar{w}_{r+l-1} - \bar{u}_i)}{(\bar{u}_{i+l-1} - \bar{u}_i)} \alpha_{i,l-1}(r) \tag{6.3.2}$$

for $l = 2,3,4$.
From (6.1.1) and (6.3.1) we have

$$\begin{aligned} Q(\bar{u},\bar{v}) &= \sum_{i=-4}^{m} \sum_{j=-4}^{n} U_{i,j} B_{i,4}(\bar{u}) B_{j,4}(\bar{v}) \\ &= \sum_{r=-4}^{p} \sum_{s=-4}^{q} \left[\sum_{i=-4}^{m} \sum_{j=-4}^{n} \alpha_{i,4}(r) \alpha_{j,4}(s) U_{i,j} \right] C_{r,4}(\bar{u}) C_{s,4}(\bar{v}) \end{aligned}$$

and so we may let

$$\mathbf{W}_{r,s} = \sum_{i=-4}^{m} \sum_{j=-4}^{n} \alpha_{i,4}(r)\alpha_{j,4}(s)\mathbf{U}_{i,j} \tag{6.3.3}$$

It is shown in [Riesenfeld80] that the α's are nonnegative and sum to 1. Further, for each fixed r, no more than 4 of the numbers $\alpha_{i,4}(r)$ can be nonzero; namely, those for which $i \in \{\delta-3, \delta-2, \delta-1, \delta\}$, where δ is the index satisfying $\bar{u}_\delta \le \bar{w} < \bar{u}_{\delta+1}$. Hence, each $\mathbf{W}_{r,s}$ constitutes a local average of at most 4 of the U's.

As an example, if a new knot is added at the midpoint of each knot interval $[\bar{u}_i, \bar{u}_{i+1})$, $i = -1, \ldots, m$; e.g.

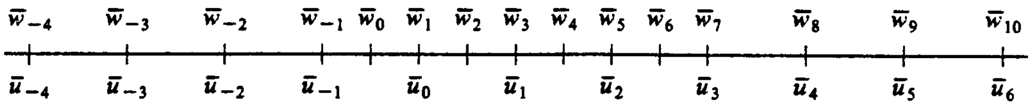

$\bar{w}_{-4}$ $\bar{w}_{-3}$ $\bar{w}_{-2}$ $\bar{w}_{-1}$ $\bar{w}_0$ $\bar{w}_1$ $\bar{w}_2$ $\bar{w}_3$ $\bar{w}_4$ $\bar{w}_5$ $\bar{w}_6$ $\bar{w}_7$ $\bar{w}_8$ $\bar{w}_9$ $\bar{w}_{10}$
$\bar{u}_{-4}$ $\bar{u}_{-3}$ $\bar{u}_{-2}$ $\bar{u}_{-1}$ $\bar{u}_0$ $\bar{u}_1$ $\bar{u}_2$ $\bar{u}_3$ $\bar{u}_4$ $\bar{u}_5$ $\bar{u}_6$

Figure 23. The special case of refinement by midpoints.

then the resulting $\alpha_{i,4}(r)$ values will be

	-4	-3	-2	-1	0	1	2	3	4	5	6
-4	1	$\frac{1}{6}$									
-3		$\frac{5}{6}$	$\frac{1}{2}$	$\frac{1}{8}$							
-2			$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{8}$					
-1				$\frac{1}{8}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{8}$			
0						$\frac{1}{8}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$		
1								$\frac{1}{8}$	$\frac{1}{2}$	$\frac{5}{6}$	
2										$\frac{1}{6}$	1

Table 2. Discrete splines for Figure 23.

The rows in this table correspond to i and the columns to r. The table relates to (6.3.3) as follows. Suppose both the $\bar{u}$ and $\bar{v}$ parameter axes have original and refined knots as indicated by Figure 23. Then $\mathbf{W}_{1,2}$ would be the weighted average of U's given by

$$\begin{aligned}\mathbf{W}_{1,2} &= \sum_{i=-4}^{2} \alpha_{i,4}(1) \left[\sum_{j=-4}^{2} \alpha_{j,4}(2) \mathbf{U}_{i,j} \right] \\ &= \frac{1}{8}\left[\frac{1}{2}\mathbf{U}_{-2,-1} + \frac{1}{2}\mathbf{U}_{-2,0}\right] + \\ &\quad \frac{3}{4}\left[\frac{1}{2}\mathbf{U}_{-1,-1} + \frac{1}{2}\mathbf{U}_{-1,0}\right] + \\ &\quad \frac{1}{8}\left[\frac{1}{2}\mathbf{U}_{0,-1} + \frac{1}{2}\mathbf{U}_{0,0}\right]\end{aligned}$$

Such a refinement corresponds in our program to *divisions* = 2. A similar table can be built for *divisions* = k, *i.e.* for the splitting of each interval $[\bar{u}_i, \bar{u}_{i+1})$ into k equal subintervals.

The recurrence (6.3.2) given above, if substituted into (6.3.3), will yield a recurrence for the W's in terms of the U's. This is the essence of the Oslo Algorithm developed by Riesenfeld et. al. Our own practice has been to stop short of the control vertex recurrence and to work with α's as shown above; that is, we use (6.2.3) and (6.3.3) explicitly. Simplicity was a factor in our decision: any control vertex graph can be handled by a single α pattern for a chosen level of refinement. The price paid, for the moment, is a lack of adaptability in setting the refinement level differently on different regions of the surface in response to local curvature.

The control vertices must be refined sufficiently in advance of the ray tracing process so that:

(1) the projection of each facet

$$\begin{matrix}\mathbf{W}_{i-1,j-1} & \mathbf{W}_{i-1,j} \\ \mathbf{W}_{i,j-1} & \mathbf{W}_{i,j}\end{matrix}$$

covers no more than a few hundred pixels on the screen, and

(2) the knots $\bar{w}_{r+2}$ (for parameter $\bar{u}$) and $\bar{t}_{s+2}$ (for parameter $\bar{v}$) associated with the control vertex $\mathbf{W}_{r,s}$ constitute acceptably good starting guesses for the Newton iteration, which is used to refine the location of a ray's intersection with the bounding box for facet (1) into the location of the ray's intersection with the spline surface itself.

The following pictures show a simple control vertex graph and the results of three selected levels of refinement.

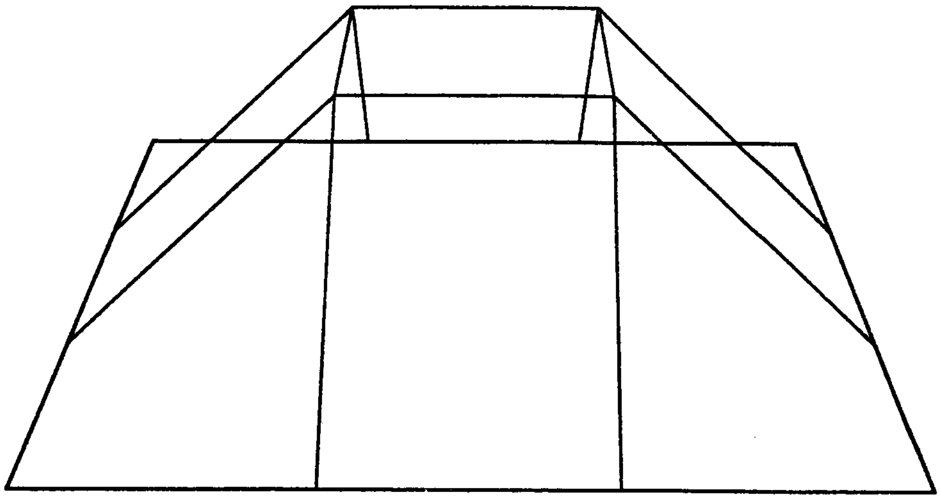


Figure 24. A simple control vertex graph.

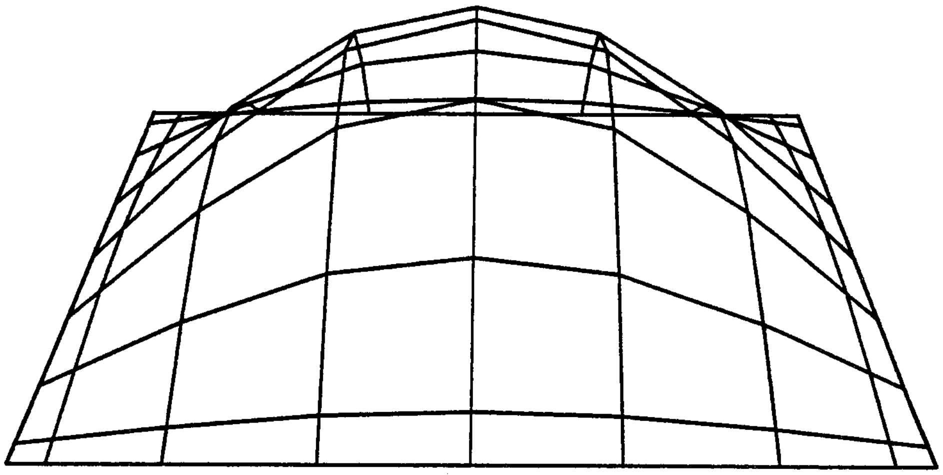


Figure 25. The control vertex graph with *divisions* = 2.

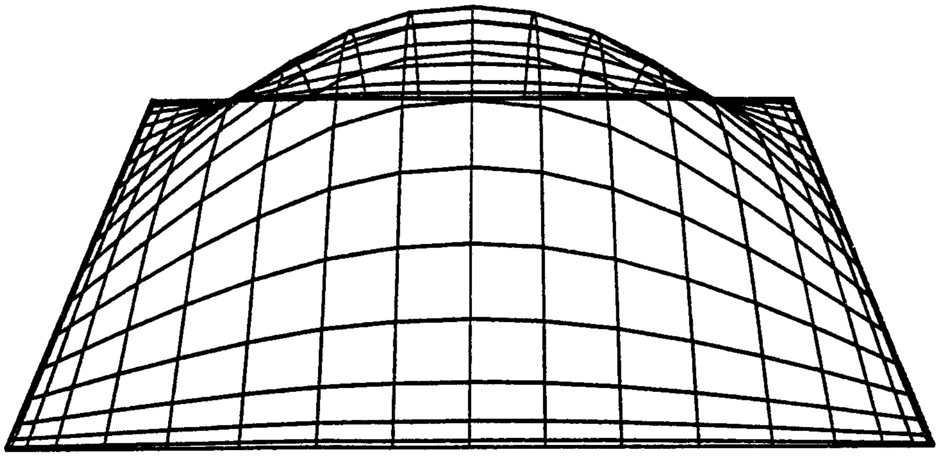


Figure 26. The control vertex graph with *divisions* = 4.

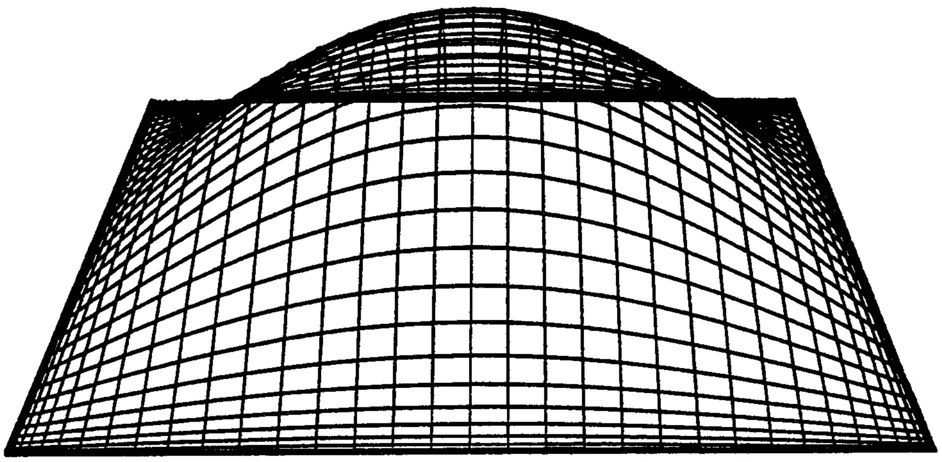


Figure 27. The control vertex graph with *divisions* = 8.

6.4. Boundary Vertices and Closed Surfaces

Note that two benefits result if the boundary vertices (those with indices -4, $m+4$, or $n+4$) are tripled. The first of these benefits can be seen from Table 2. Restricting our attention to the parameter $\bar{u}$ alone, consider the case shown by the control graph in Figure 24 with

$$\mathbf{U}_{-4,j} = \mathbf{U}_{-3,j} = \mathbf{U}_{-2,j}$$

We have

$$\mathbf{W}_{-4,j} = 1\mathbf{U}_{-4,j}$$

$$\mathbf{W}_{-3,j} = \frac{1}{6}\mathbf{U}_{-4,j} + \frac{5}{6}\mathbf{U}_{-3,j} = \mathbf{W}_{-4,j}$$

and

$$\mathbf{W}_{-2,j} = \frac{1}{2}\mathbf{U}_{-3,j} + \frac{1}{2}\mathbf{U}_{-2,j} = \mathbf{W}_{-4,j}$$

But notice that

$$\mathbf{W}_{-1,j} = \frac{1}{8}\mathbf{U}_{-3,j} + \frac{3}{4}\mathbf{U}_{-2,j} + \frac{1}{8}\mathbf{U}_{-1,j} \neq \mathbf{W}_{-4,j}$$

This means that the tripling is maintained at each stage of refinement.

The second benefit which results is a convention which permits us to recover single, double and triple boundary vertex surfaces from triple boundary vertex control graphs. We merely need to restrict the ranges of the parameters $\bar{u}$ and $\bar{v}$ appropriately. Figures 28-30 illustrate this.

Coloured image can be seen on page 164.

Figure 28. A patch as it would result from single boundary vertices (2:55 minutes).

The patch shown in Figure 28 is precisely that which would result if a maximum possible parameter range were used

$$\bar{u}_{-1} \leq \bar{u} < \bar{u}_{m+1} \tag{6.4.1}$$
$$\bar{v}_{-1} \leq \bar{v} < \bar{v}_{m+1}$$

and if the boundary control vertices were only single. In point of fact, the boundary vertices are triple, and the patch was produced by restricting the parameters to

$$\bar{u}_1 \leq \bar{u} < \bar{u}_{m-1}$$
$$\bar{v}_1 \leq \bar{v} < \bar{v}_{n-1}$$

Coloured image can be seen on page 164.

Figure 29. A patch as it would result from double boundary vertices (26:49 minutes).

In Figure 29 the results of double boundary vertices and the parameter ranges of (6.4.1) are reproduced by triple boundary vertices and the restriction of the parameter ranges to

$$\bar{u}_0 \leq \bar{u} < \bar{u}_m$$
$$\bar{v}_0 \leq \bar{v} < \bar{v}_n$$

Coloured image can be seen on page 164.

Figure 30. A patch as it would result from triple boundary vertices (32:26 minutes).

Finally, Figure 30 displays the "true" patch defined by the control vertices, resulting from the triple control vertices interacting with the full parameter range (6.4.1).

The pictures above are of open surfaces. Closed surfaces are obtained by "wrapping control vertices around upon themselves".

Coloured image can be seen on page 164.

Figure 31. A closed spline donut (93:04 minutes).

The donut of Figure 31, for example, was produced by making

$$\mathbf{U}_{i,j} = \mathbf{U}_{i,j+5}$$
$$\mathbf{U}_{i,j} = \mathbf{U}_{i+5,j}$$

for an arrangement of 81 control vertices.

6.5. Tree Construction

The refinement process described above constitutes a first step in the preprocessing of each spline surface. The second step in preprocessing involves building a tree of nested rectilinear bounding boxes on top of the refined vertices. Each leaf of the tree represents a small bounding box which is centered on one particular refined vertex. There is one leaf per vertex and one vertex per leaf. Each internal node of the tree represents a bounding box which is just large enough to contain the bounding boxes of its four children.

A rectilinear bounding box is defined by two points $(xmin, ymin, zmin)$ and $(xmax, ymax, zmax)$. For a leaf bounding box centered on the vertex $\mathbf{W}_{r,s}$, these two points are determined as follows:

$$\begin{aligned}
xmax &= (1.0-overlap)\mathbf{W}[x]_{r,s} + overlapMAX(\mathbf{W}[x]_{r,s}\mathbf{W}[x]_{r+1,s}\mathbf{W}[x]_{r-1,s}\mathbf{W}[x]_{r,s+1}\mathbf{W}[x]_{r,s-1}) \\
xmin &= (1.0-overlap)\mathbf{W}[x]_{r,s} + overlapMIN(\mathbf{W}[x]_{r,s}\mathbf{W}[x]_{r+1,s}\mathbf{W}[x]_{r-1,s}\mathbf{W}[x]_{r,s+1}\mathbf{W}[x]_{r,s-1}) \\
ymax &= (1.0-overlap)\mathbf{W}[y]_{r,s} + overlapMAX(\mathbf{W}[y]_{r,s}\mathbf{W}[y]_{r+1,s}\mathbf{W}[y]_{r-1,s}\mathbf{W}[y]_{r,s+1}\mathbf{W}[y]_{r,s-1}) \\
ymin &= (1.0-overlap)\mathbf{W}[y]_{r,s} + overlapMIN(\mathbf{W}[y]_{r,s}\mathbf{W}[y]_{r+1,s}\mathbf{W}[y]_{r-1,s}\mathbf{W}[y]_{r,s+1}\mathbf{W}[y]_{r,s-1}) \\
zmax &= (1.0-overlap)\mathbf{W}[z]_{r,s} + overlapMAX(\mathbf{W}[z]_{r,s}\mathbf{W}[z]_{r+1,s}\mathbf{W}[z]_{r-1,s}\mathbf{W}[z]_{r,s+1}\mathbf{W}[z]_{r,s-1}) \\
zmin &= (1.0-overlap)\mathbf{W}[z]_{r,s} + overlapMIN(\mathbf{W}[z]_{r,s}\mathbf{W}[z]_{r+1,s}\mathbf{W}[z]_{r-1,s}\mathbf{W}[z]_{r,s+1}\mathbf{W}[z]_{r,s-1})
\end{aligned}$$

where $\mathbf{W}[x]_{r,s}$ is the x component of the vertex $\mathbf{W}_{r,s}$

Thus the size of a leaf bounding box is determined by how close the contained vertex $\mathbf{W}_{r,s}$ is to the neighboring vertices. The parameter *overlap* is also read from the scene file, and controls the amount that adjacent bounding boxes overlap one another. Both *divisions* and *overlap* could be selected adaptively, but for the present experiments we prefer to choose them ourselves. For fairly flat surfaces, and those with gentle curves only (e.g. the bump of Figure 30), an *overlap* of 0.5 is sufficient. Otherwise *overlap* must be increased, to a maximum of 1.0 for objects with very sharp curves. An *overlap* less than 0.5 produces an interesting effect (see Figure 55)

The leaves are organized into a tree by a procedure which recursively subdivides the $\bar{u},\bar{v}$ parameter rectangle (6.1.2). At each level of recursion, the procedure allocates a node of the tree (the *current node*) and connects to it the four nodes to be allocated at the next level. The current node is associated with a rectangular section of the $\bar{u},\bar{v}$ range (6.1.2) (in particular the root node is associated with the entire rectangle (6.1.2)), and the current node's rectangular section is quartered by halving its sides to produce the subrectangles given to the current node's children.

The recursion terminates when the current node's rectangluar section of the $\bar{u},\bar{v}$ plane contains only one pair of refined knot coordinates $\bar{w}_r,\bar{t}_s$, which we associate with the unique control vertex weighted by the B-spline product $C_{r,4}(\bar{u})\cdot C_{s,4}(\bar{v})$. The current node is tagged as a leaf node, and a leaf bounding box is calculated as above. The pair $\bar{w}_{r+2},\bar{t}_{s+2}$ are included in the leaf's data structure as a starting point for the Newton process.

As the procedure returns through the recursion, the parent nodes are tagged as internal nodes, and ever larger bounding boxes are calculated to contain the bounding boxes of the children.

6.6. The Spline Object Node

A spline object node contains the following information (Figure 32). Most of the fields have already been described for a sphere object node in Section 2.1.

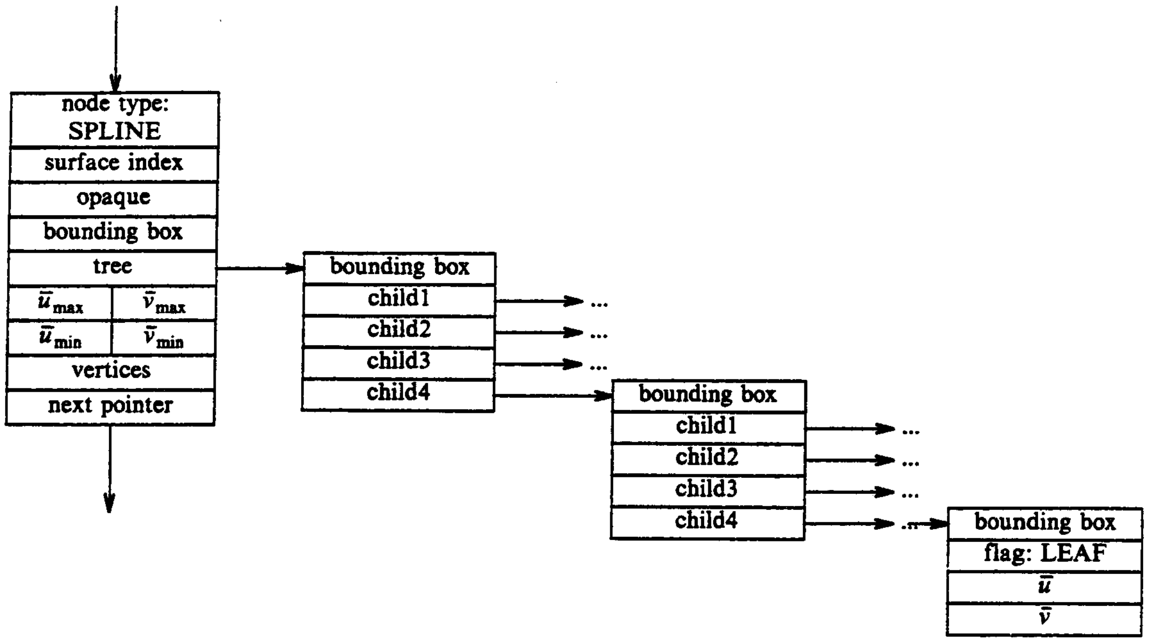


Figure 32. The object node associated with a spline.

1. tree: a pointer to the root of the tree of bounding boxes generated as above.

2. $\bar{u}_{max}$: the maximum possible $\bar{u}$ value, as determined by the end conditions to be applied in the u parametric direction (Section 6.4).

3. $\bar{v}_{max}$: the maximum possible $\bar{v}$ value, as determined by the end conditions to be applied in the u parametric direction (Section 6.4).

4. $\bar{u}_{min}$: the minimum possible $\bar{u}$ value, as determined by the end conditions to be applied in the v parametric direction (Section 6.4).

5. $\bar{v}_{min}$: the minimum possible $\bar{v}$ value, as determined by the end conditions to be applied in the v parametric direction (Section 6.4).

6. vertices: the original nu $\times$ nv matrix of control vertices, which has been put through the current transformation matrix.

An internal node of the tree of nested rectilinear bounding boxes contains the following information:

1. bounding box: the box.

2. 4 pointers to descendants.

And finally, a leaf node of the tree of nested bounding boxes contains the following information:

1. bounding box: the box.

2. flag: an indentification that this node is in fact a leaf of the tree.

3. ($\bar{u}$,$\bar{v}$): the starting values for a newton iteration.

The memory requirements for a spline surface are determined largely by the size of the tree of bounding boxes, and this is dictated, in turn, by the number of given control vertices and the level of refinement. Requirements for three of the surfaces featured in Figures 30, 31 and 53 are shown in Table 3.

surface	bump	donut	mask
number original vertices	64	81	2025
divisions	8	16	2
number refined vertices	1849	9801	7569
number tree nodes	3374	21534	16374
bytes	133K	843K	659K

Table 3. Memory Requirements of Various Surfaces

6.7. Intersection Testing

Recall that the leaf nodes of our tree of bounding boxes contain starting values for a Newton iteration. We use Kajiya's algorithm, as applied to fully instantiated subdivision trees, [Kajiya83a, page 178], to select candidate leaf nodes for further processing by the Newton iteration routine.

To reiterate what was said in Section 5.5, for each ray we maintain a linked list of active nodes. Attached to those nodes are various subtrees of the tree of bounding boxes described above. With each node is associated a distance from the ray origin to the closest intersection with the bounding box of the root of the attached subtree. We maintain the list of active nodes sorted by increasing distance. The algorithm proceeds as follows:

- Choose the first (closest) node on the active node list, and remove it.
- If the root of the attached subtree is interior to the tree consider in turn each of it's four children.
- If the ray hits the bounding box of a child, then attach the child to an active node, and sort the node into the the active node list.
- If the root of the attached subtree is a leaf, use the contained ($\bar{u}$,$\bar{v}$) parameter values to initiate a Newton process.

The algorithm terminates when the active node list is empty (failure), or the distance to the surface, as returned by the Newton iteration routine, is less than the distance to the first (closest) node on the active node list (success).

6.8. The Newton Iteration

The goal of an intersection computation is that of finding a pair of parameter values $\bar{u}$,$\bar{v}$ such that a point Q($\bar{u}$,$\bar{v}$) on the surface is also a point contained in a given ray. The two unknowns, $\bar{u}$ and $\bar{v}$, can be expressed as the roots of a pair of polynomial equations by formulating the desired intersections as the locus of all points on the surface which lie simultaneously in two planes containing the ray. This formulation was borrowed from [Kajiya82], although the rest of our intersection process is entirely different from the one he presented. We have chosen to compute the intersection from Newton's iteration, a full account of which can be found in [Henrici64]. We have

Plane 1: $\vec{N_1}\cdot(x,y,z) = \alpha_1$

Plane 2: $\vec{N_2}\cdot(x,y,z) = \alpha_2$

where

$$(x,y,z) = (x(\bar{u},\bar{v}),y(\bar{u},\bar{v}),z(\bar{u},\bar{v})) = Q(\bar{u},\bar{v})$$

In particular, for a ray given parametrically as

$$R(t) = \vec{O}+t\vec{D}$$

then

$$\vec{N_1} = \vec{O}\times\vec{D}$$
$$\vec{N_2} = \vec{N_1}\times\vec{D}$$
$$\alpha_1 = \vec{N_1}\cdot\vec{O}$$
$$\alpha_2 = \vec{N_2}\cdot\vec{O}$$

Using (6.1.1) we have the equations

$$E_k(\bar{u},\bar{v}) \equiv \sum_{i=-4}^{m}\sum_{j=-4}^{n}\left[\vec{N_k}\cdot \mathrm{U}_{i,j}\right]B_{i,4}(\bar{u})B_{j,4}(\bar{v}) - \alpha_k = 0 \qquad (6.8.1)$$

for $k = 1,2$.

Let $\bar{u}^{(0)},\bar{v}^{(0)}$ stand for the values stored in a leaf node. Newton's method starts with these values as an approximation to the solution of (6.8.1) and refines them

$$\bar{u}^{(0)} \rightarrow \cdots \rightarrow \bar{u}^{(l)} \rightarrow \bar{u}^{(l+1)} \rightarrow \cdots$$
$$\bar{v}^{(0)} \rightarrow \cdots \rightarrow \bar{v}^{(l)} \rightarrow \bar{v}^{(l+1)} \rightarrow \cdots$$

according to

$$\begin{bmatrix}\bar{u}^{(l+1)}\\ \bar{v}^{(l+1)}\end{bmatrix} = \begin{bmatrix}\bar{u}^{(l)}\\ \bar{v}^{(l)}\end{bmatrix} - \begin{bmatrix}\Delta\bar{u}^{(l)}\\ \Delta\bar{v}^{(l)}\end{bmatrix}$$

where $\Delta\bar{u}^{(l)}$, $\Delta\bar{v}^{(l)}$ solve the 2×2 system

$$\begin{bmatrix}\dfrac{\partial E_1}{\partial \bar{u}} & \dfrac{\partial E_1}{\partial \bar{v}}\\ \dfrac{\partial E_2}{\partial \bar{u}} & \dfrac{\partial E_2}{\partial \bar{v}}\end{bmatrix}\begin{bmatrix}\Delta\bar{u}^{(l)}\\ \Delta\bar{v}^{(l)}\end{bmatrix} = \begin{bmatrix}E_1(\bar{u}^{(l)},\bar{v}^{(l)})\\ E_2(\bar{u}^{(l)},\bar{v}^{(l)})\end{bmatrix}$$

to produce a (usually) more accurate solution of (6.8.1). The partial derivatives $\dfrac{\partial E_k}{\partial \bar{u}}$ for $k = 1,2$ are given by

$$\frac{\partial E_k}{\partial \bar{u}} = \sum_{i=-4}^{m}\sum_{j=-4}^{n}\left[\vec{N_k}\cdot \mathrm{U}_{i,j}\right]B'_{i,4}(\bar{u})B_{j,4}(\bar{v})$$

and similarly for $\dfrac{\partial E_k}{\partial \bar{v}}$. The derivatives of the B-splines are found by differentiating (6.1.3). The control vertices which are used in the iteration should be the original, unrefined set as stored in the spline object node.

The Newton iteration is terminated successfully (that is, $\bar{u}^{(l+1)},\bar{v}^{(l+1)}$ are taken as defining an intersection) if

$$|E_1(\bar{u}^{(l+1)},\bar{v}^{(l+1)})| + |E_2(\bar{u}^{(l+1)},\bar{v}^{(l+1)})| < \textit{tolerance}$$

Currently a *tolerance* of 1.0 is being used with good results.

Failures are registered (that is, a ray strike is regarded as not occurring) if the Newton iterates $\bar{u}^{(l+1)},\bar{v}^{(l+1)}$ wander outside the bounds of the parametric intervals; *i.e.*

$$\bar{u}^{(l+1)} < \bar{u}_{\min} \text{ or } \bar{u}^{(l+1)} > \bar{u}_{\max} \text{ or } \bar{v}^{(l+1)} < \bar{v}_{\min} \text{ or } \bar{v}^{(l+1)} > \bar{v}_{\max}$$

or if the value of $|E_1(\bar{u}^{(l+1)},\bar{v}^{(l+1)})| + |E_2(\bar{u}^{(l+1)},\bar{v}^{(l+1)})|$ has increased over that of $|E_1(\bar{u}^{(l)},\bar{v}^{(l)})| + |E_2(\bar{u}^{(l)},\bar{v}^{(l)})|$.

6.9. Some Useful Things to Calculate

The normal vector to the spline surface is available as

$$\bar{N} = \left[\sum_{i=-4}^{m}\sum_{j=-4}^{n} \mathrm{U}_{i,j} B'_{i,4}(\bar{u}^{(l+1)}) B_{j,4}(\bar{v}^{(l+1)})\right] \times \left[\sum_{i=-4}^{m}\sum_{j=-4}^{n} \mathrm{U}_{i,j} B_{i,4}(\bar{u}^{(l+1)}) B'_{j,4}(\bar{v}^{(l+1)})\right]$$

This vector must be normalized before being used by the shader.

The calculations can be arranged so that both sides of the cross product are obtained as a part of the calculation of the partial derivatives., i.e.

$$\frac{\partial E_k}{\partial \bar{u}} = \sum_{i=-4}^{m}\sum_{j=-4}^{n} \left[\vec{N_k}\cdot \mathrm{U}_{i,j}\right] B'_{i,4}(\bar{u}) B_{j,4}(\bar{v})$$

$$= \left[\sum_{i=-4}^{m}\sum_{j=-4}^{n} \mathrm{U}_{i,j} B'_{i,4}(\bar{u}) B_{j,4}(\bar{v})\right]\cdot \vec{N_k}$$

And similarly for $\dfrac{\partial E_k}{\partial \bar{v}}$.

Also, the actual intersection point is available as

$$\bar{I} = \mathrm{Q}(\bar{u}^{(l+1)},\bar{v}^{(l+1)}) = \sum_{i=-4}^{m}\sum_{j=-4}^{n} \mathrm{U}_{i,j} B_{i,4}(\bar{u}^{(l+1)}) B_{j,4}(\bar{v}^{(l+1)})$$

where, again, the calculations may be arranged so that this is obtained during the computation of $E_k(\bar{u},\bar{v})$.

The distance to the surface, t_i is just

$$t_i = \frac{I_z - O_z}{D_z}$$

If the ray does, in fact, hit the surface then the pair $(\bar{u}^{(l+1)},\bar{v}^{(l+1)})$ returned by the Newton routine may be used as indices into a texture map. These values must be mapped to the range [0,1] before use.

$$u = \frac{\bar{u}^{(l+1)} - \bar{u}_{\min}}{\bar{u}_{\max} - \bar{u}_{\min}}$$

$$v = \frac{\bar{v}^{(l+1)} - \bar{v}_{\min}}{\bar{v}_{\max} - \bar{v}_{\min}}$$

7. Super Boxes

There is one other type of DAG node in addition to those which represent transformations and those which represent primitive objects (Section 1.2) which deserves discussion here. It is the box DAG node. This node causes a rectilinear bounding box to be created which contains all of the (possibly transformed) primitive objects in the portion of the DAG connected to its down pointer. Only if the ray intersects this box will it be tested against the contents of the box. This can be a great time saver if there are a lot of objects clustered together in some small section of the image.

7.1. Preprocessing

A box DAG node contains the following information (Figure 33) copied from the scene file. All of the fields have already been described for a sphere DAG node in Section 2.1.

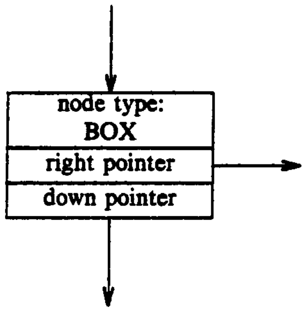


Figure 33. The DAG node associated with a box.

Notice that one of the fields in each of the object nodes is a minimal bounding box for that particular primitive object (Sections 2.1, 3.1, 4.1, 5.4 and 6.6). On encountering a box DAG node, the preprocessor first processes all of the DAG connected to the box node's down pointer into object nodes. It then calculates a larger containing bounding box from the bounding boxes of all of these object nodes, creates a new object node with type box, and links all of these object nodes to the contents field of the new node.

A box object node contains the following information (Figure 34). Many of the fields have already been described for a sphere object node in Section 2.1.

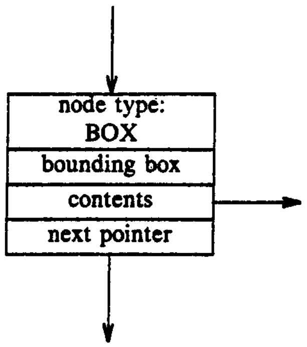


Figure 34. The object node associated with a box.

1. contents: a pointer to a list of object nodes.

7.2. Use

The object list is not necessarily a linear linked list. Consider the following DAG (Figure 35) and the corresponding object list produced by the preprocessor (Figure 36).

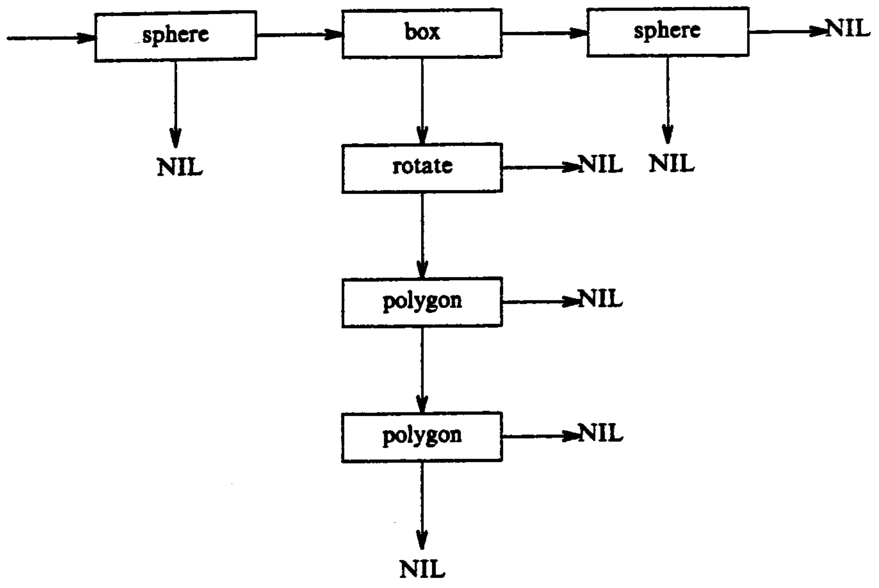


Figure 35. A DAG containing a box node

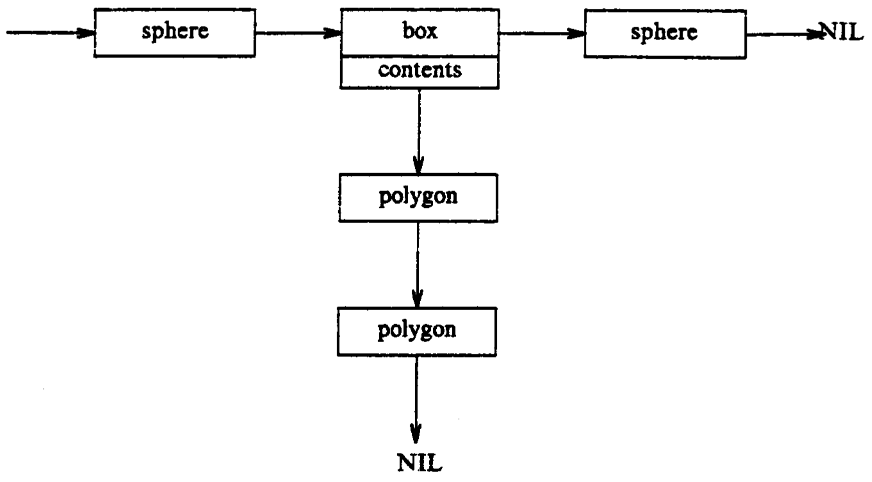


Figure 36. The corresponding object list.

Here, the ray will be tested against the two polygons only if it intersects the bounding box of the box object node.

8. Procedure Render

So far, we've described how we preprocess a scene file to produce an object list, and how we intersect rays with individual members of that list. We will now discuss the heart of our program. Procedure Render generates the primary rays and runs the ray tracing algorithm. Depending on how the package was configured at compile time, this may or may not involve anti-aliasing.

Render calls the recursive procedure Trace, which decides what to do with a ray if it in fact does intersect something. Reflective and/or transmitted rays may be cast depending on the characteristics of the surface hit.

We will begin with a description of the ray data structure.

8.1. Rays

The data structure representing a ray contains the following information (Figure 37).

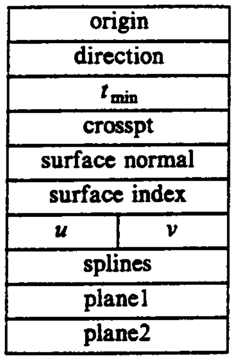


Figure 37. The internal structure of a ray.

1. origin: the origin ($\vec{O}$) of the ray.

2. direction: the direction ($\vec{D}$) of the ray. The ray is then $R(t) = \vec{O}+t\vec{D}$.

3. $t_{\min}$: the minimum ray parameter in any object/ray intersected found thus far. $t_{\min}$ is initialized when the ray is generated, and will not change if the ray fails to hit any object.

4. crosspt: the intersection point ($\vec{I}$) of the ray with the closest object found.

$$\vec{I} = \vec{O}+t_{\min}\vec{P}$$

5. surface normal: the normal ($\vec{N}$) at the intersection point.

6. surface index: the number (s) of the surface descriptor containing information about how light interacts with the closest object. This is copied from the object node of the closest object.

7. u, v: indices into a texture map if the closest object is textured.

8. splines: boolean flag set to TRUE by the intersection processor if the ray penetrates the outer bounding box of any spline surface, and so two planes containing the ray must be calculated (see Section 6.8).

9. plane1, plane2: if splines is TRUE, the two planes containing the ray calculated by the spline intersection processor.

The intersection processor traverses the list of object nodes, testing the current ray with each member. If the ray does hit an object, the ray parameter t_i at the intersection point is compared to the $t_{\min}$ stored in the ray data structure. If t_i is less than $t_{\min}$, the $t_{\min}$, intersection point, surface normal, surface index, and texture map indices in the ray data structure are reset.

This implies that when a ray is returned from the intersection processor, either $t_{\min}$ hasn't changed, or $\vec{I}$, $\vec{N}$, s, u and v give information on the closest intersection.

There are fields suggested by others which are *not* included in our data structure. Kajiya suggests several fields [Kajiya83b] which are unnecessary because our algorithm runs recursively. Both Hall [Hall83a] and Kajiya include the maximum possible contribution the ray can make to a pixel. We control the number of rays generated both by a user-specified value for the maximum depth of recursion, and by a test for uniformity of colour in our anti-aliasing algorithm.

8.2. Primary Ray Generation

Before we consider how to generate primary rays, it might be instructive to consider how **not** to generate them. One might think of applying the perspective transformation to the scene, then simply shooting rays parallel to the z-axis, one per pixel (Figure 38).

Coloured image can be seen on page 164.

Figure 38. An image with incorrect perspective (≈90 minutes).

The perspective is perfect on the objects themselves in the above picture, but the reflections are wrong. This is because a perspective transformation assumes a specific viewpoint, whereas the mirror simulates another viewpoint. Compare Figure 38 to Figure 39, where the perspective is calculated correctly.

Coloured image can be seen on page 165.

Figure 39. The same image with correct perspective (109:28 minutes).

The way to calculate perspective correctly in ray tracing is as Whitted describes in [Whitted80]. A virtual screen and a virtual viewpoint are defined in the same coordinate system as the objects to be rendered. Each ray is cast from this viewpoint through a pixel in the screen. It's as if your eye were really there, looking though a window into the object space.

Notice however that this leaves plenty of freedom about how and where to place the screen and viewpoint. You could be looking at the same scene from the positive z axis inwards, or the negative y axis upwards.

In the package, the position and orientation of the virtual screen/viewpoint are specified by display parameters in the scene file (see Appendix 1). These parameters are converted into the data structure given in Figure 40.

viewpoint
start
xpath
ypath

Figure 40. The data structure defining the virtual screen/viewpoint.

The virtual viewpoint is simply a point $\vec{V}$. The screen is a plane, and as such can be defined by a starting point $\vec{S}+\vec{V}$ and two vectors $\vec{X}$ and $\vec{Y}$ These values are set up so that a primary ray

$$R(t) = \vec{O}_{primary} + t\vec{D}_{primary}$$

is calculated simply by

$$\vec{O}_{primary} = \vec{V}$$
$$\vec{D}_{primary} = \vec{S} + x\vec{X} + y\vec{Y}$$

where (x,y) is the current pixel location. $\vec{D}_{primary}$ must be normalized before use because it enters into lighting calculations.

Also, for primary rays, the initial $t_{\min} = 10^{10}$

8.3. Control of Ray Tracing

Whitted [Whitted80] and Hall [Hall83a] build a complete tree of rays, then pass the entire tree to the shader. This approach is necessary if you want to use the Potmesil camera and lens postprocess [Potmesil82]. We don't use this approach. We run the algorithm recursively, shading each ray as it comes up.

The input to procedure Render is a linked list of object nodes, and the output is a run-length encoded file of pixel colours. See Figure 41.

```
Render()
{
for( each scanline y )
   {
   for( each pixel x in the scanline )
      {
      generate a primary ray from (x,y);
      cv = Trace(ray,0);
      colour=ToRGB(cv);
      scanline[x] = colour;
      }
   run length encode the scanline;
   }
}
```

Figure 41. Control structure for ray tracing without anti-aliasing.

```
colourvector *Trace(ray,level)
{
  Intersect(ray,object_list);

  if( nothing is hit )
     return( background );

  if( level>depth )
     return( Shade(ray,background,NIL,background,NIL) );

  if( the object hit is reflective )
    {
    reflected_ray = Reflect( ray_direction,intersection_point,surface_normal );
    reflected_colour = Trace( reflected_ray,level+1 );
    }
  else
    {
    reflected_ray = NIL;
    reflected_colour = background;
    }

  if( the object hit is transparent )
    refracted_ray = Refract( ray_direction,intersection_point,surface_normal,refractive_index );
    refracted_colour = Trace( refracted_ray,level+1 );
    }
  else
    {
    refracted_ray = NIL;
    refracted_colour = background;
    }

  return( Shade(ray,reflected_colour,reflected_ray,refracted_colour,refracted_ray) );
}

Intersect(ray,object_list)
{
  for( p=object_list; p!=NIL; p=p->next )
    if( the ray intersects object p )
      if( the intersection $t_i$ is less than the current ray $t_{min}$ )
        {
        reset the $t_{min}$, intersection point, surface normal in the ray data structure;
        reset the surface index, texture map indices in the ray data structure;
        }
}
```

Figure 41. Control structure for ray tracing without anti-aliasing (continued).

Notice that all of the information needed by Trace is either a part of the ray data structure returned by Intersect (the intersection point, surface normal) or available from the surface descriptor whose index is a part of that data structure (the surface reflectivity, transparency, refractive index).

The program parameter **depth** (see Appendix 1) sets a limit on the recursive depth of ray tracing.

8.4. Calculation of Reflected Rays

One of the things which makes ray tracing unique is the ability to accurately simulate reflection. The way in which a reflected ray contributes to the colour of a primary ray depends on the illumination model being used, but it is basically additive. See Section 9 for details.

The geometry of reflection is illustrated in Figure 42.

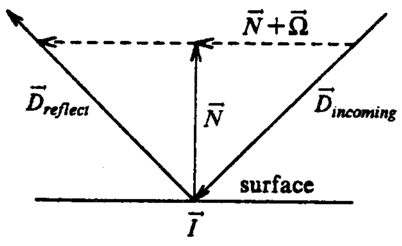


Figure 42. The geometry of reflection.

A reflected ray

$$R(t) = \vec{O}_{reflect} + t\vec{D}_{reflect}$$

is determined by the simple rule that the angle of incidence must equal the angle of reflection [Whitted80].

$$\vec{O}_{reflect} = \vec{I}$$
$$\vec{D}_{reflect} = \vec{\Omega} + 2\vec{N}$$

where

$$\vec{\Omega} = \frac{\vec{D}_{incoming}}{\left|\vec{D}_{incoming} \cdot \vec{N}\right|}$$

and the sign of $\vec{N}$ is adjusted so that $\vec{D}_{incoming} \cdot \vec{N}$ is less than zero

$\vec{D}_{reflect}$ must be normalized before use because it enters into lighting calculations. The data structure representing a reflected ray is the same as that used for a primary ray (Section 8.1). As with primary rays, the initial $t_{\min} = 10^{10}$.

The intersection point ($\vec{I}$), surface normal ($\vec{N}$) and incoming ray direction ($\vec{D}_{incoming}$) are parameters to procedure Reflect, which returns the reflected ray.

8.5. Calculation of Refracted Rays

The geometry of refraction is illustrated in Figure 43.

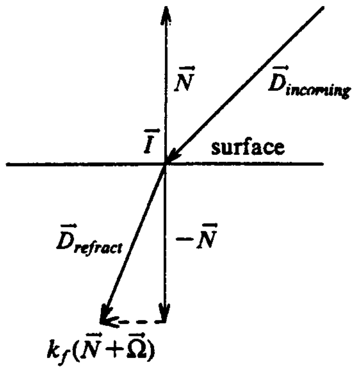


Figure 43. The geometry of refraction.

A refracted ray

$$R(t) = \vec{O}_{refract} + t\vec{D}_{refract}$$

is calculated in accordance with Snell's law [Whitted80]

$$\vec{O}_{refract} = \vec{I}$$
$$\vec{D}_{refract} = k_f(\vec{N}+\vec{\Omega})-\vec{N}$$

where

$$\vec{\Omega} = \frac{\vec{D}_{incoming}}{\left|\vec{D}_{incoming}\cdot\vec{N}\right|}$$
$$k_f = \left(n^2\left|\vec{\Omega}\right|^2 - \left|\vec{\Omega}+\vec{N}\right|^2\right)^{-1/2}$$

n = the index of refraction

and the sign of $\vec{N}$ is adjusted so that $\vec{D}_{incoming}\cdot\vec{N}$ is less than zero

$\vec{D}_{refract}$ must be normalized before use because it enters into lighting calculations. The data structure representing a refracted ray is the same as that used for a primary ray (Section 8.1). As with primary and reflected rays, the initial $t_{\min} = 10^{10}$.

The intersection point ($\vec{I}$), surface normal ($\vec{N}$), incoming ray direction ($\vec{D}_{incoming}$) and index of refraction (n) are parameters to procedure Refract, which returns the refracted ray.

8.6. Control of Anti-Aliasing

If the program has been configured to do anti-aliasing, the control structure is somewhat different. We do anti-aliasing by super-sampling as described by Whitted [Whitted80]. Four rays are cast to the corners of each pixel. The colours returned are compared, and if the difference exceeds a threshold, the pixel is recursively subdivided until either the colours at the corners of the subdivided pixel match or a recursion limit is exceeded.

What Whitted does not mention in his paper is that much can be gained from saving the rays already cast. The upper right hand corner of one pixel is same as the upper left hand corner of the next pixel (to the right). In fact, assuming a completely uniform object space and 512×512 pixel resolution, only 1023 more rays need be cast at four per pixel (to the corners) than at one per pixel (to the center).

```
Render()
{
for( each scanline y )
   {
   for( each pixel x in the scanline )
      scanline[x] = Subdivid(x,y+1.0,x+1.0,y,aalevel);
   run length encode the scanline;
   }
}

rgb *Subdivid(xleft,yupper,xright,ylower,level)
{
  if( the table entry for (xleft,yupper) is empty )
     {
     generate a primary ray from (xleft,yupper);
     cv = Trace( ray,0 );
     upperleft_colour = ToRGB(cv);
     store upperleft_colour in the table;
     }
  else
     retrieve upperleft_colour from table;

  if( the table entry for (xright,yupper) is empty )
     {
     generate a primary ray from (xright,yupper);
     cv = Trace( ray,0 );
     upperright_color = ToRGB(cv);
     store upperright_colour in the table;
     }
  else
     retrieve upperright_colour from table;

  if( the table entry for (xleft,ylower) is empty )
     {
     generate a primary ray from (xleft,ylower);
     cv = Trace( ray,0 );
     lowerleft_colour = ToRGB(cv);
     store lowerleft_colour in the table;
     }
  else
     retrieve lowerleft_colour from table;

  if( the table entry for (xright,ylower) is empty )
     {
     generate a primary ray from (xright,ylower);
     cv = Trace( ray,0 );
     lowerright_colour = ToRGB(cv);
     store lowerright_colour in the table;
     }
  else
     retrieve lowerright_colour from table;
```

Figure 44. Control structure for ray tracing with anti-aliasing.

```
    level /= 2;
    if( level>0 &&
       difference(upperleft_colour,upperright_colour,lowerleft_colour,lowerright_colour)>aathreshold )
          {
          xmid=(xleft+xright)/2.0;
          ymid=(yupper+ylower)/2.0;

          upperleft_colour = Subdivid(xleft,yupper,xmid,ymid,level);
          upperright_colour = Subdivid(xmid,yupper,xright,ymid,level);
          lowerleft_colour = Subdivid(xleft,ymid,xmid,ylower,level);
          lowerright_colour = Subdivid(xmid,ymid,xright,ylower,level);
          }

    return( (upperleft_colour+upperright_colour+lowerleft_colour+lowerright_colour)/4 );
}

colourvector *Trace(ray,level)
{
  /*
   * As per Figure 41.
   */
}
```

Figure 44. Control structure for ray tracing with anti-aliasing (continued).

The program parameters **aalevel** and **aathreshold** (see Appendix 1) give the user control of the amount of anti-aliasing done, and hence the time taken.

Coloured image can be seen on page 165.

Figure 45. The effects of various amounts of anti-aliasing. From left to right, the balls were generated by: the program not configured to do anti-aliasing (1.01 minutes), **aalevel**=1 (1.06 minutes), **aalevel**=2 (1.33 minutes), **aalevel**=4 (1.55 minutes), **aalevel**=8 (2.33 minutes). In all cases, the **aathreshold** was set to 40.0.

9. The Shader

The shader takes as input the current ray, the reflected ray (if any), the colour of the reflected ray (R_λ), the refracted ray (if any), the colour of the refracted ray (T_λ), and the tables of lighting information and surface descriptors produced by ReadScene. It generates the colour to be associated with the current ray.

The colour is generated in accordance with one of the three illumination models supported (Figure 46).

Coloured image can be seen on page 165.

Figure 46. The three illumination models supported, and our ambience function (9:04).

The following table lists the meanings of the symbols used in this section.

symbol	meaning	source
dw	solid angle (brightness) of incident light	lighting table
e	exponent used in phong shading model	surface descriptor
k_a	ambient reflection constant	ambience colour-vector
k_d	diffuse reflection constant	surface descriptor
k_r	fraction of light reflected by object	surface descriptor
k_s	specular reflection constant	surface descriptor
k_t	fraction of light transmitted by object	surface descriptor
k_λ	extinction coefficient	not used
n_λ	index of refraction	not used
m	rms deviation of facet slope distribution	surface descriptor
D	facet slope distribution function	calculated
D_λ	surface normal reflectance	surface descriptor
$\vec{D}_{reflect}$	direction of reflected ray	input to shader
$\vec{D}_{refract}$	direction of refracted ray	input to shader
F_λ	surface fresnel reflectance	surface descriptor
G	geometrical attenuation factor	calculated
$\vec{H}$	unit angular bisector of $\vec{V}$ and $\vec{L}$	calculated
I_λ	intensity of incident light	lighting table
$I_{a\lambda}$	intensity of ambient light	ambience colour-vector
$\vec{I}$	current ray/object intersection point	ray data structure
$\vec{L}$	unit vector in direction of light	lighting table
$\vec{N}$	unit surface normal at intersection point	ray data structure
$\vec{P}$	unit vector in the direction of perfect reflection	calculated
R_λ	colour of reflected ray	input to shader
S_λ	total specular term in Cook-Torrance model	calculated
T_λ	colour of transmitted ray	input to shader
$\vec{V}$	negative of current ray direction (ie view direction)	ray data structure
Z_λ	colour of current ray	calculated

Table 4. Summary of symbols.

9.1. Representation of Colour

Two light sources with very different spectral compositions may appear to be the same colour. This phenomenon is called metamerism, and the light sources are said to be metamers. Additive colour mixing occurs when two light sources are close enough spatially and temporally that they appear as one colour. Taken together, metamerism and additive colour mixing imply that all colours can be produced from a minimal set of primary colours [Cowan83]. This minimal set has exactly three members.

The trick in realistic image synthesis is to produce metamers of real world colours using only the 3 phosphor colours of a particular monitor. Furthermore, it is desirable to be able to reproduce those metamers on another monitor (or even in another medium) with a minimum of recoding.

The Commission Internationale de l'Eclairage (CIE) has defined a standard set of primarys. A colour is specified by a triplet of positive numbers (X,Y,Z), known as its *tristimulus values*, which are defined in terms of the spectral power $\phi(\lambda)$ of the light source.

$$X = \int \phi(\lambda)\bar{x}_\lambda d\lambda$$

$$Y = \int \phi(\lambda)\bar{y}_\lambda d\lambda$$

$$Z = \int \phi(\lambda)\bar{z}_\lambda d\lambda$$

where $\bar{x}_\lambda$, $\bar{y}_\lambda$ and $\bar{z}_\lambda$ are known as the colour matching functions, and are tabulated at 5 nanometer intervals in [Handbook72, page 6-189]. Colour monitors can be calibrated to accurately display colours in terms of their CIE coordinates [Cowan83].

The basic unit of colour used throughout the package is the *colour-vector*. A colour-vector c_λ is a 33 element array representing the reflectivity of a surface or the intensity of a light source between 380 and 700 nanometers in 10 nanometer intervals. It was found that 33 values were suitable for Cook-Torrance shading.

For output, the colour-vector must be converted to an rgb value. The colour-vector is first changed to the corresponding tristimulus values (X,Y,Z) by the method in [Handbook72] adapted to 10 nanometer intervals, and cut off at 700 nanometers.

$$X = 1360 \sum_{\lambda=380}^{\lambda=700} c_\lambda \bar{x}_\lambda$$

$$Y = 1360 \sum_{\lambda=380}^{\lambda=700} c_\lambda \bar{y}_\lambda$$

$$Z = 1360 \sum_{\lambda=380}^{\lambda=700} c_\lambda \bar{z}_\lambda$$

The resulting tristimulus values may then be displayed as detailed in [Cowan83]. Briefly they are converted to rgb by passing through the inverse phosphor chromaticity matrix of the particular monitor for which the package is configured, then displayed on the calibrated monitor. For our Electrohome ECM 1301 colour monitor

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \frac{1}{15} \begin{pmatrix} 0.62 & 0.21 & 0.15 \\ 0.33 & 0.675 & 0.06 \\ 0.05 & 0.115 & 0.79 \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

The rgb values produced are occasionally outside of the gamut of the monitor. However unlike Cook [Cook81a], we do not try to correct for this. The values are simply clipped to be within the range [0,255]. The program formerly printed a warning whenever a value was out of range. However, no artifacts were noticeable in the image, and eventually even this warning was removed.

9.2. Sources of Real World Colours

The problem arises of where to get colour-vectors.

For light sources, the spectral composition of CIE standard illuminants A and D_{6500} are tabled in [Handbook72, page 6-185]. The colour of the sky at various elevations can be found in [Wyszecki82 page 8-9]. Also, theatrical lighting manufactures supply sample gel filters complete with graphs of $\phi(\lambda)$ v.s. λ.

For surfaces, there are several volumes of data in [Purdue70]. A listing of the spectral reflectance for a most astonishing collection of surfaces (e.g. a meadow with clover and timothy in bloom) is in [Krinov47]. Also the back of Cook's thesis [Cook81b] lists the spectral reflectances he actually used in his pictures.

Finally, there is an auxiliary program which, given an arbitrary rgb value, produces the equivalent colour-vector.

9.3. Lights

It was mentioned in Section 1.2 that ReadScene saves lighting information in a table to be used by the shader. An entry in the lighting information table contains the following information (Figure 47).

intensity
inscene
direction
dw

Figure 47. An entry in the table of lighting information.

1. intensity: the intensity (I_λ) of the light source between 380 and 700 nanometers in 10 nanometer intervals.

2. inscene: a boolean value.

3. direction: if inscene is FALSE, this field represents a normalized direction vector ($\vec{L}$) to the light source, which is assumed to be at infinity. If inscene is TRUE, it represents the actual position ($\vec{l}$) of the light source, which may be within the scene.

4. *dw*: the solid angle that the light source subtends. This is the projected area of the light source divided by the square of the distance to it, and is required by the Cook-Torrance model. However, a moment's thought reveals that this is a meaningless concept when applied to a point source at infinity. So, in fact, *dw* acts as a brightness control which applies to all shading models.

If inscene is TRUE, the light direction vector must be calculated. This is easily accomplished.

$$\vec{L} = \vec{l} - \vec{I}$$

i.e. the direction to the light source equals the position of the light source ($\vec{l}$) minus the position of the ray/object intersection point ($\vec{I}$). This vector must be normalized before being used.

If $\vec{L} \cdot \vec{N} < 0$ the surface normal is pointing away from the light source, and all calculation for this light source can halt. Otherwise, a ray is cast from the ray/object intersection point towards the light source. A shadow ray

$$R(t) = \vec{O}_{shadow} + t\vec{D}_{shadow}$$

is calculated simply by

$$\vec{O}_{shadow} = \vec{I}$$
$$\vec{D}_{shadow} = \vec{L}$$

If this ray hits any object before it reaches the light, the intersection point is shadowed from this light source, and all calculation for this light source can halt. The phrase "before it reaches the light" causes a minor complication. If the light source is at infinity

$$\text{initial } t_{\min} = 10^{10}$$

as usual (Sections 8.2, 8.4 8.5). However, if the source is within the scene, we have

$$\text{initial } t_{\min} = \frac{l_x - I_x}{L_x}$$

Because an object must be closer than the current $t_{\min}$ to reset it, the test whether $t_{\min}$ has been reset (Section 8.1) will still work.

In fact, there is one field of the object nodes about which we have said nothing yet: opaque, a boolean value set to TRUE if the associated surface descriptor has a transparency of 0.0. This is actually a very crude attempt at what could easily be made into casting coloured, attenuated shadows. As it stands, if opaque is FALSE, the intersection routine for calculating shadows ignores the object node. Consequently transparent objects do not cast shadows. This was necessary in order to be able to create objects (typically spheres) to hold light sources in the scene.

9.4. Surface Descriptors

It was also mentioned in Section 1.2 that ReadScene saves surface characteristics (that is, information about how light interacts with a surface and how this interaction is to be modelled) in a table, an entry of which is called a *surface descriptor*. It was seen in Section 8.1 that the current ray passed to the shader contains the number (s) of the surface descriptor which applies to the object to be shaded. It is this descriptor which will determine which illumination model will be used to shade the object.

Every surface descriptor contains the following information (Figure 48).

shading model	
reflectivity	
transparency	
refractive index	
textured	
umult	vmult

Figure 48. The fields common to surface descriptors.

1. shading model: the model to be used

2. reflectivity: the fraction of light reflected from the surface (k_r).

3. transparency: the fraction of light transmitted by the surface (k_t).

4. refractive index: the index of refraction of the surface.

5. textured: a boolean flag.

6. umult, vmult: two real numbers indicating how many times the texture map (if any) is to be repeated over the surface.

The shading model, reflectivity, transparency, and texture are almost completely independent. So for instance, you can specify a transparent texture map with a phong highlight, or a perfectly reflective Cook-Torrance copper surface.

The one thing which is not supported is texture mapped surface shaded by the Cook-Torrance model. This is not impossible, but calculation of the Fresnel reflectance (see Section 9.9) from the texture map indices was considered to be too expensive.

9.5. Illumination Models

In general, an illumination model in computer graphics treats the interaction of light with matter as consisting of three components: ambient, diffuse and specular. Ambient light is the sum of all light which can not be attributed at any specific source, and is reflected equally in all directions. The diffuse component is attributable to specific sources, and is reflected equally in all directions. The specular component is also attributable to specific sources, but is reflected preferentially in one direction.

In addition any illumination model used for ray tracing must include a contribution from the reflected ray, and one from the transmitted ray.

9.6. Ambient Illumination

As stated above, ambient light is the sum of all light which can not be attributed to any specific light source. The archetypical situation in which ambient illumination is observed is outside on a totally overcast day.

However, even in such a situation objects do not look totally flat. Think of looking at the metal support of a stop sign on such a day.

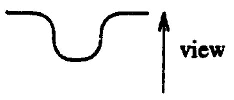


Figure 49. The cross section of a metal stop sign support.

There will be some shading evident. Those portions which make greater right angles to your line of sight will appear marginally darker.

The ambience term in all of our models takes account of this effect.

$$k_a((\vec{N}\cdot\vec{V})+0.5)I_{a\lambda}\pi D_\lambda$$

π is included as a concession to the Cook-Torrance model (see Section 9.9). No justification for the 0.5 term is made other than the fact that the results "look right". The sphere marked "ambient" in Figure 46 was calculated this way.

Allowance should also be made for the fact that we naturally expect more light to come from above, even on completely shadowless overcast days. This is future work.

The ambience information is obtained from ReadScene as one colour-vector incorporating $k_a\pi I_{a\lambda}$.

9.7. Lambert's Law

There are a few surfaces which are dull enough to scatter light equally in all directions, that is, to reflect only diffusely. For such surfaces, Lambert's law relates the amount of light reaching the viewer to the intensity of the light (I_λ), the direction of the light ($\vec{L}$) and the surface normal ($\vec{N}$) by

$$k_d(\vec{L}\cdot\vec{N})I_\lambda$$

where k_d is the diffuse reflection constant, and is a property of the surface.

This simple law produces limited realism, but is useful for matte background surfaces such as the walls of a room. It is also the least expensive shading model to compute.

A surface descriptor which specifies Lambert's Law shading contains the following information (Figure 50). Most of the fields have already been described in Section 9.4.

shading model LAMBERT	
reflectivity	
transparency	
refractive index	
textured	
umult	vmult
colour	
texture	
diffuse	

Figure 50. A surface descriptor specifying Lambert's Law shading.

1. colour: if textured is FALSE, this is the colour-vector which indicates the normal reflectance (D_λ) of the surface.

2. texture: if textured is TRUE, this is a 512X512 texture map, from which the normal reflectance (D_λ) will be calculated. See Section 10 for details.

3. diffuse: the diffuse reflection constant (k_d).

Translating Lambert's law to the ray tracing context, the colour of the current ray is (refer to Table 4 for the symbols):

$$Z_\lambda = k_a((\vec{N}\cdot\vec{V})+0.5)I_{a\lambda}\pi D_\lambda + k_r R_\lambda + k_t T_\lambda + \sum_n k_d(\vec{N}\cdot\vec{L}_n)D_\lambda I_{\lambda n} dw_n$$

where the sum is taken over only those lights not eliminated as in Section 9.3.

The sign of $\vec{N}$ is adjusted so that $\vec{N}\cdot\vec{V} \geq 0$. This may not always be desirable (for example, a polygon and its back reflection would look strange). However it is extremely convenient when working with splines not to have to worry about which way the normal vector is going to point. No artifacts have been noticed in any of the pictures we have produced to date, so for now we are keeping this feature.

9.8. Phong

A perfect mirror reflects light in precisely one direction. Very few surfaces are perfect mirrors. Yet many surfaces do reflect light preferentially in one direction, with a rapid decline as the direction changes (that is, they have a highlight). Based on empirical observation, Phong [Phong75] proposed that this fall off occurs as $(\vec{P}\cdot\vec{V})^e$, where $\vec{P}$ is the vector in the direction of perfect reflection of the light source off of the surface, and e is a property of the surface. For a perfect mirror e would be infinite.

Phong highlights are always the colour of the light source, which tends to make all surfaces look as if made out of plastic (see Figure 46). This effect is explained in [Cook81a]. However, it is often not objectionable. Also, suitably low D_λ which make the object almost vanish but does not affect the highlight makes convincing bubbles, mirrors and glass.

A surface descriptor which specifies Phong shading contains the following information (Figure 51). Most of the fields have already been described in Section 9.4.

shading model PHONG	
reflectivity	
transparency	
refractive index	
textured	
umult	vmult
colour	
texture	
diffuse	
specular	
exponent	

Figure 51. A surface descriptor specifying Phong shading.

1. colour: if textured is FALSE, this is the colour-vector which indicates the normal reflectance (D_λ) of the surface.

2. texture: if textured is TRUE, this is a 512X512 texture map, from which the normal reflectance (D_λ) will be calculated.

3. diffuse: the diffuse reflection constant (k_d).

4. specular: the specular reflection constant (k_s).

5. exponent: the power (e) to which ($P{\cdot}V$) is raised.

Translating Phong shading to the ray tracing context, the colour of the current ray is

$$Z_\lambda = k_a((\vec{N}\cdot\vec{V})+0.5)I_{a\lambda}\pi D_\lambda + k_r R_\lambda + k_t T_\lambda + \sum_n \left[k_d(\vec{N}\cdot\vec{L}_n)D_\lambda I_{\lambda n} dw_n + k_s(\vec{P}\cdot\vec{V})^e I_{\lambda n} dw_n\right]$$

where the sum is taken over only those lights not eliminated as in Section 9.3. Here $\vec{P}$ is the vector in the direction of perfect reflection of the light source off of the surface

$$\vec{P} = 2\vec{N} - \frac{\vec{L}_n}{(\vec{N}\cdot\vec{L}_n)}$$

This vector must be normalized before being used.

As with Lambert's Law shading, the sign of $\vec{N}$ is adjusted so that $\vec{N}\cdot\vec{V} \geq 0$.

9.9. Cook-Torrance

The Cook-Torrance model [Cook81a] is based on the measured spectral reflectance of a surface. The ability to support the this illumination model in the context of ray tracing was one of the major design goals of this project. The Cook-Torrance model is the only illumination model which is capable of acurately rendering the "real world colours" of Section 9.2.

Briefly, the total illumination is given by

$$I_{a\lambda}A_\lambda + \sum_n I_{\lambda n}(\vec{N}\cdot\vec{L}_n)dw_n(k_s S_\lambda + k_d D_\lambda)$$

Here A_λ is restricted to be a linear combination of S_λ and D_λ. Cook [Cook81b] appearantly always sets it to πD_λ, so we do likewise.

The angular spread of the specular component (S_λ) is based on the assumption [Torrance67] that the surface is made up of micro facets, each of which reflects as a perfect mirror.

$$S_\lambda = \frac{F_\lambda}{\pi}\frac{DG}{(\vec{N}\cdot\vec{L}_n)(\vec{N}\cdot\vec{V})}$$

G is a geometrical attenuation factor

$$G = \min\left\{1, \frac{2(\vec{N}\cdot\vec{H})(\vec{N}\cdot\vec{V})}{(\vec{V}\cdot\vec{H})}, \frac{2(\vec{N}\cdot\vec{H})(\vec{N}\cdot\vec{L}_n)}{(\vec{V}\cdot\vec{H})}\right\}$$

where

$$\vec{H} = \frac{\vec{V}+\vec{L}_n}{|\vec{V}+\vec{L}_n|}$$

D is the facet slope distribution function. Like Cook we use the Beckman distribution function.

$$D = \frac{1}{m^2(\vec{N}\cdot\vec{H})^4}e^{-\left\{\frac{\tan(\arccos(\vec{N}\cdot\vec{H}))}{m}\right\}^2}$$

However, have always found one scale of roughness per surface to be sufficient.

F_λ may be calculated theoretically from the Fresnel equation [Sparrow66, pages 63-64], given the index of refraction (n_λ) and the extinction coefficient (k_λ) of the surface, and the angle of illumination ($\theta = \arccos(\vec{V}\cdot\vec{H})$). However, we have found the values of n_λ and k_λ tabulated in [Handbook72] to be inadequate and consequently we always use the compromise suggested by Cook. An external program calculates an effective n_λ from the normal reflectance.

$$n_\lambda = \frac{1+\sqrt{D_\lambda}}{1-\sqrt{D_\lambda}}$$

We then assume that $k_\lambda = 0$ and calculate $F_{\lambda\theta}$ for 66 values of θ between 0 and 90 degrees using the Fresnel equation. We tabluate these results in the scene description file (that is 33 wavelengths at 66 angles), and they become a part of the Cook-Torrance surface descriptor. The calculation of F_λ then becomes a table lookup, where the index into the table is

$$\frac{\arccos(\vec{V}\cdot\vec{H})}{33\pi}$$

coerced to an integer.

Notice that the normal reflectance used to calculate the Fresnel table need not be the same as the D_λ of the surface.

A surface descriptor which specifies Cook-Torrance shading contains the following information (Figure 52). Most of the fields have already been described in Section 9.4.

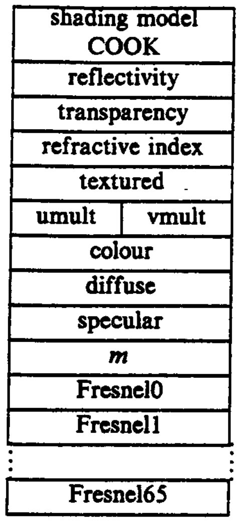


Figure 52. A surface descriptor specifying Cook-Torrance shading.

1. colour: the normal reflectance (D_λ).

2. diffuse: the diffuse reflection constant (k_d).

3. specular: the specular reflection constant (k_s).

4. m: the rms deviation of the facet slope distribution function.

6. Fresnel0 .. Fresnel65: 66 colour-vectors representing the Fresnel reflectance ($F_{\lambda,\theta}$) at angles between 90 and 0 degrees.

To convert to a ray tracing context, first notice that for each light source, the illumination is a function of I_λ, $\vec{L}$, dw, $\vec{N}$, k_s, k_d, D_λ, $\vec{V}$, m and $F_{\lambda\theta}$. That is, the total illumination is given by

$$I_{a\lambda}A_\lambda + \sum_n \Psi(I_{\lambda n},\vec{L}_n,dw_n,\vec{N},k_s,k_d,D_\lambda,\vec{V},m,F_{\lambda\theta})$$

A reflected ray, and a transmitted ray are treated exactly the same as any other light source. For a reflected ray the intensity of the light source is the colour of the ray, R_λ, the direction to the light source is the direction of the reflected ray $\vec{D}_{reflect}$, and the solid angle (brightness) of the source is the surface reflectivity, k_r. For a transmitted ray the light intensity is the colour of the ray, T_λ, the direction to the light source is the negative of the transmitted ray direction $-\vec{D}_{refract}$, and the solid angle (brightness) of the source is the surface transparency, k_t.

The colour of the current ray is, then,

$$\begin{aligned} Z_\lambda = \; & k_a((\vec{N}\cdot\vec{V})+0.5)I_{a\lambda}\pi D_\lambda + \\ & \Psi(R_\lambda,\vec{D}_{reflect},k_r,\vec{N},k_s,k_d,D_\lambda,\vec{V},m,F_{\lambda\theta}) + \\ & \Psi(T_\lambda,-\vec{D}_{refract},k_t,\vec{N},k_s,k_d,D_\lambda,\vec{V},m,F_{\lambda\theta}) + \end{aligned}$$

$$\sum_n \Psi(I_{\lambda n}, \vec{L}_n, dw_n, \vec{N}, k_s, k_d, D_\lambda, \vec{V}, m, F_{\lambda\theta})$$

where the sum is taken over only those lights not eliminated as in Section 9.3.

As with Lambert's Law shading, the sign of $\vec{N}$ is adjusted so that $\vec{N}\cdot\vec{V} \geq 0$.

Some reflections on Cook's paper: we have found that the ambient illumination he reported using $(0.01 I_\lambda)$ to be about 2 orders of magnitude too high. We found that, if used in a scanline algorithm, the highlighted areas looked realistic, but otherwise the shading was extremely flat. The unhighlighted areas in Cooks pictures [Cook81b] also appear flat. The reason that we have a non-standard ambience function is to try to give some shading to these flat regions. Also, although the Cook-Torrance model accurately predicts the surface shading of metal, we have found that true metallic appearance comes as much from the coherence of reflections as from the shading (see Figure 54). Calculation between 380 and 700 nanometers in 10 nanometer intervals appears to be sufficient. Cook gives no hint of the range or precision he used.

10. Texture Mapping

Lack of detail is an immediate clue that an image is computer generated rather than photographed from the real world. The visual complexity of most parts of the real world is orders of magnitude higher than is computationally tractable.

There are currently two approaches being taken towards generating images with a more realistic amount of detail.

1. Data base amplification: this involves specifying a simple primitive and a procedural method for generating detail, generally involving an element of randomness. Fractals are one example of this approach. The particle systems of Bill Reeves [Reeves83] are another.

2. Texture mapping: this is the projection of an external image onto some surface in the image being calculated [Catmull74]. This external image may be a photograph which has been scanned in, or a pre-calculated computer generated image. This allows one to produce images of greatly increased complexity with no corresponding increase in the complexity of the geometric object description [Whitted82].

Earlier we described how we generate texture map indices (u,v), $0.0 \leq u,v \leq 1.0$, in the intersection routines (Sections 2.3, 3.3, 4.3, 5.7 and 6.9). Also, recall that the surface descriptors for Lambert's Law (Section 9.7) and the Phong (Section 9.8) illumination models contain the following fields, read from the scene file:

1. textured: a boolean flag.

2. umult, vmult: two real numbers $(u_m,\ v_m)$ indicating how many times the texture map (if any) is to be repeated over the surface.

3. texture: if textured is TRUE, this is a 512×512 texture map.

The external image is originally a run-length encoded file, which is decoded by ReadScene and stored in the surface descriptor as a 512×512 array of 32 bit integers. This image is projected onto the surface by using (u,v) as indices into this array to determine the surface colour (D_λ) at the intersection point.

The calculated (u,v) are not used directly. Instead†

$$u_i = (512uu_m) mod 512$$
$$v_i = (512vv_m) mod 512$$

then (u_i,v_i), coerced to integers, are used as indices into the array to choose the appropriate entry.

Next, this integer must be converted into the standard unit of colour in the package: the *colour-vector*.

† This method of repeating a texture map is due to David Forsey.

The low 8 bits of the integer represent the red component (r), the next 8 bits represent the green component (g), the next 8 bits the blue component (b), and the high 8 bits are unused. These three components are converted into a colour-vector by running the process described in Section 9.1 in reverse.

The rgb values are converted to tristimulus values by passing through the phosphor chromaticity matrix of the particular monitor for which the package is configured. For our Electrohome ECM 1301 colour monitor

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = 15 \begin{pmatrix} 0.62 & 0.21 & 0.15 \\ 0.33 & 0.675 & 0.06 \\ 0.05 & 0.115 & 0.79 \end{pmatrix} \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

There remains the problem of finding a 33 value colour-vector which will produce these tristimulus values. Recall

$$X = 1360 \sum_{\lambda=380}^{\lambda=700} D_\lambda \bar{x}_\lambda$$

$$Y = 1360 \sum_{\lambda=380}^{\lambda=700} D_\lambda \bar{y}_\lambda$$

$$Z = 1360 \sum_{\lambda=380}^{\lambda=700} D_\lambda \bar{z}_\lambda$$

where $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$ are the colour matching functions, tabulated in [Handbook72].

Three wavelengths were chosen, 450 nm, 550 nm, and 600 nm, as being the maxima of the colour matching functions. Then

$$X = 1360(D_{450}\bar{x}_{450} + D_{550}\bar{x}_{550} + D_{600}\bar{x}_{600})$$

$$Y = 1360(D_{450}\bar{y}_{450} + D_{550}\bar{y}_{550} + D_{600}\bar{y}_{600})$$

$$Z = 1360(D_{450}\bar{z}_{450} + D_{550}\bar{z}_{550} + D_{600}\bar{z}_{600})$$

These are three equations in three unknowns (D_{450}, D_{550}, D_{600}), and is solved by Cramer's rule. The rest of D_λ is set to zero. This D_λ is used in the further shading calculation.

11. Comparison with Other Programs

The major point I want to make in this section is that each ray is fully shaded as it comes up in the recursion. By this I mean that the specular term in particular is not determined completely by the light intensity coming in from the reflected ray [Whitted80]. In fact, reflected rays are quite often **not** projected at all. The specular term is fully calculated according to the particular lighting model at each intersection point.

This also means that we do not build a complete tree of rays, then pass this tree as a whole to the shader, as do both Whitted [Whitted80] and Hall [Hall83a].

Roth and Barr have shown the usefulness of associating transformations with objects. The transformation is applied to the ray before it is intersected with the object. In this way, each object can be thought of as being centered in its own local coordinate system, making intersection (with quadrics especially) easy [Roth80]. Barr has extended the concept of solid modelling primitives to include bent, twisted, tapering or otherwise deformed primitives, each thought of as an undeformed primitive in its own coordinate system and a transformation [Barr83]. It is safe to say (with 20-20 hindsight) that we erred in leaving out this capability.

The other thing that we can not do, but which is particularly easy in the ray tracing context [Roth80, Roth82, Kajiya83b], is to form boolean combinations of primitives.

11.1. Operation Times

The above algorithm was coded in C, with the bounding box intersection routine coded in Monol, and run on a VAX 11/780 under the UNIX† operating system. Our program executes at roughly the same speed as those few for which figures have been published [Whitted80, Hall83a, Kajiya83a]. The time critical routines are, predictably, those that test for intersection, and those that do shading. The following table gives the average times to perform each of the listed operations once.

operation	time (milliseconds)
intersect sphere	1.31
intersect cylinder	1.30
intersect rectangle	0.25
intersect fractal (6subdivisions)	12.43
intersect spline (bump)	83.85
intersect super-box	0.09
shade ray (lamberts law)	2.99
shade ray (phong)	4.45
shade ray (cook-torrance)	3.81

Table 5. Operation Times

These times, presented in isolation as they are above, are probably of little meaning to the reader. Perhaps a better idea of the performance characteristics of the package may be gained from the timing given in the caption beneath each picture in this thesis. All times refer to a 512 by 512 resolution, but with the image tightly bounded by program parameters (see Appendix 1).

11.2. Spline Algorithm Results

Table 6 gives a breakdown of times spent in various parts of the program, expressed as per cent of total time, together with other operating statistics, for three sample B-spline surfaces. The bump surface refers to that presented in Figure 30, the donut in Figure 31, and the mask in Figure 53. Again, all times refer to a 512 by 512 image of the surface tightly bounded by program parameters, with no other objects in the scene, using Phong shading, and not anti-aliased.

surface	bump	donut	mask
surface type	open	closed	open
number of original vertices	64	81	2025
divisions	8	16	2
overlap	0.5	0.75	0.55
number refined vertices	1849	9801	7569
refinement time (%)	0.6	0.8	0.8
tree building time (%)	0.2	0.4	0.6
bounding box intersect time (%)	22.2	21.9	22.0
Newton time (%)	46.3	66.7	48.1
shading and overhead time (%)	30.7	10.2	28.5
average boxes checked/ray	25.44	105.39	36.66
average Newton calls/ray	0.84	4.43	1.47
average iterations/call	3.04	3.09	2.95
total time	32 min.	93 min.	49 min.

Table 6. Performance Characteristics for Various Surfaces

Because the majority of the time is spent doing Newton iterations, it might be conjectured that making the starting guesses for the iteration better (by increasing refinement of the control vertices) would lead to

† UNIX is a foot note of Bell Laboratories.

lower overall times. This indeed is the case for the torus. However, for sufficiently flat surfaces (e.g. the bump), just the opposite occurs: increasing refinement leads to *greater* overall times.

12. Further Work

This package has proven to be capable of rendering texturally complex, visually beautiful images, and should continue to be useful for doing so. It's construction is modular enough to allow for very easy incorporation of additional primitive objects and shading models, as the opportunities present themselves.

There are four possible methods for obtaining images from the package significantly faster than is currently possible. The first two involve only some recoding, the third involves moving the algorithm to a multiprocessor architecture, and the fourth is a research project.

The first method is to associate transformations with objects as discussed in the previous section. This should speed the intersection calculation with spheres, cylinders and polygons. It is also possible that considering each bounding box as a canonical box and an associated transformation matrix might prove to be faster than the current method for rendering B-spline surfaces and fractals.

The second method is, of course, to move more of the code to Monol. Previously, the package has been in such a state of flux that maintaining two versions of routines (one in C for development and portability, and another in Monol) has been impractical. One attempt was made to code all of the intersection routines and the shader in Monol. This lead to an approximately 40 per cent speed increase. The package has since changed, and only the routine which performs intersections with bounding boxes was left in Monol. However, a "release" version of the package is about to become a reality. It is expected that this version will be stable enough to make recoding it in Monol a reasonable thing to do.

Although the code was written to be implemented on a strictly serial processor, almost all of it could be transferred to a multiprocessor machine organized as per [Ullner83] and presently under development at the University of Calgary [Cleary83]. Ullner's machine partitions space into discrete volumes, each represented by unit processor. Each processor has its own object list, representing those objects present in that portion of space (object nodes representing B-spline and fractal surfaces need not contain the entire tree of bounding boxes, only the sub-tree whose root node bounding box encloses the part of the surface present in that portion of space). If a ray does not intersect any member of a processor's object list, the ray is handed to a neighbouring processor. If there is an intersection, new rays are generated and traced. This machine gets its speed from the shorter object list present in each processor, plus the fact that reflected, refracted and shadow rays may be traced in parallel.

The author plans to use this package as the starting point of an investigation into yet another method of speeding up ray traced animation: the use of frame−to−frame coherence.

Roth's scheme of tracing a low resolution grid, and then concentrating on the "interesting" pixels [Roth82] would probably not result in a significant speed up. In complex images, most of the pixels would be "interesting". Using program parameters to set the left, right, high and low boundaries of the image has much the same effect.

Two questions have come out of this work which, curiously enough, have little to do with ray tracing. The first is: do polished metals reflect qualitatively differently than dull ones? This was prompted by observing the bright green reflection in the copper mask of Figure 54, and considering how a real copper object would appear under similar circumstances.

The second question is concerned with motion cues. David Forsey and I have made a short animated sequence "Crater Lake", which begins with a flight above the lake from the inlet of Figure 58 to a circle around the two central columns of Figure 57. Some people (I am not one of them) claim to see a distinct jump, as the circle begins, from the situation where the observer is moving and the scene is stationary, to the situation where the observer is stationary and the scene is moving. So the question is: what is the visual cue that the observer is moving?

13. Gallery

We finish by displaying a few more images which show the range of effects possible with the package. All pictures were composed into an Adage/IKONAS frame buffer and photographed using a Dunn camera without gamma correction.

Coloured image can be seen on page 165.

Figure 53. A Halloween mask was cut into strips and digitized, giving the control mesh for this spline surface (72:29 minutes).

Coloured image can be seen on page 165.

Figure 54. View of a gallery room with a spline mask. The mask is rendered in copper, and the sphere and cylinder are stainless steel. The copper and steel were achieved using the Cook-Torrence illumination model. The sphere and mask reflect each other, as well as a tapestry and doorway directly behind the viewer and pictures on the side walls ($\approx$35 hours).

Coloured image can be seen on page 165.

Figure 55. Three vases with insufficient overlap. From the highest to the lowest vases, the overlaps are 0.4, 0.3 and 0.2 (90:32 minutes).

Coloured image can be seen on page 166.

Figure 56. The texture maps used in this image are the work of local artist Karen Fletcher (73.44 minutes).

Coloured image can be seen on page 166.

Figure 57. Crater lake looking north towards the inlet. The hills are texture mapped fractals, the columns are spline surfaces, and the water is a reflective rectangle. Blinn's dusty surface lighting model was specially adapted by David Forsey to render the glowing sphere (106:38 minutes).

Coloured image can be seen on page 166.

Figure 58. Crater lake as seen from the inlet (228:09 minutes).

14. Acknowledgements

At last it is time to spend the credit around a bit for producing this package.

Much is due to Kelly Booth and John Beatty who created the UofW CGL environment. There's nothing like a hot cappuccino, a cold pizza, and the stereo on **loud** at 4:00 in the morning to get the ideas flowing. Dave Martindale gave me 24 Mbytes of virtual address space to play with. Dave Forsey, through a mixture of constant criticism and friendship, shaped the package. Almost every other lab member has helped in some way or other.

Dan Field and Forbes Burkowski both proofread this document not once but twice!

And then there's my supervisor, Richard Bartels, who continually disavows any input into this thesis. Many (if not most) of the good ideas contained are his. He has been a well of mathematical knowledge, from which I've drunk many times.

Appendix 1. Format of the Scene Description File

1. Introduction

It is suggested that the best way to use this document is to read it all the way through quickly once, then slowly once, then try some scenes, referring back to the document as necessary.

The scene file contains a complete, detailed description of the picture to be rendered. This includes information about the geometry and surface characteristics of every object in the picture, the lighting environment, the position and orientation of the viewer, and the operation of the program itself.

The file format will be given in a simplified, incomplete Backus-Naur Form, followed by English descriptions of the various fields. The following table lists the meanings of the meta-symbols used.

meta-symbol	meaning
::=	shall be defined to be
|	alternatively
()	used for grouping
x	the terminal symbol x
lower-case-name	a non-terminal symbol
.	end of this definition
EOLN	the end of a line

Table 7. Meta-symbols used.

In general, the scene file "language" is line oriented, like FORTRAN or BASIC. Tokens of the language are keywords, pathnames, integers, and real numbers. All keywords must be in lower case. Any number of blanks and/or tabs may separate tokens, as long as there is at least one such entry.

Lines which begin with an asterisk ("*"), as well as completely blank lines, are considered to be comments and both are ignored by the parser. All characters following the required fields on a line are ignored as well, and may be employed as the user wishes. Thus the meta-symbol EOLN designates the end of a line, possibly preceded by ignored characters.

2. An Example

Here is a sample scene file. It is, in fact, the same example which was given in the thesis introduction, and produces an approximation to Whitted's famous checkerboard image.

```
* scene:  a mirror ball above a green/blue checkerboard
  5 2 4 2 1 4 1

* program parameters
1 depth 3
2 shadows
3 xleft  31
4 xright 480
5 yhigh  286

* display parameters
1 eye 0.0 0.0 3.5
2 sight towards 0.0 0.0 0.0

* vertices
1  -1.0 -0.0 -1.0
2  -1.0 -0.0 +1.0
3  +1.0 -0.0 +1.0
4  +1.0 -0.0 -1.0

* surface 1:  the checkerboard texture map
1 lambert textured 0.0 0.0 0.0
  7.0
  /u/majsweeney/textures/checkerboard  1  1
* surface 2:  a perfect reflector
2 phong normal 1.0 0.0 0.0
  0.0 2.5 25.0
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06 0.06
0.06 0.06 0.06

* ambience
  0.00050
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6

* lights
1 infinity  0.60000 1.00000 0.600000  0.00060
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6

* scene nodes
1 rotate    02 04        x -22.0
2 translate 03 00         0.0 -0.8 0.0
3 polygon   00 00 1        4  4 3 2 1
4 sphere    00 00 2       0.3 -0.18 0.0   0.16    x y
```

Figure 59. A sample scene description.

3. Basic Format

```
scene-file   ::=   numbers-of-things
                   program-parameters
                   display-parameters
                   vertices
                   surface-descriptors
                   ambience-descriptor
                   light-sources
                   scene-nodes .
```

As is evident from the above example, the scene file is divided into 7 major functional divisions. The *numbers-of-things* gives the number of entries in each division.

The *program-parameters*, and *display-parameters* control the operation of the program itself. *surface-descriptors* contain all of the information on the surface characteristics of objects. There is a surface descriptor associated with each primitive object named in the *scene-nodes*. The *ambience-descriptor* and *light-sources* describe the lighting environment of the scene. Finally, the geometry of the scene is described by the *vertices* and the *scene-nodes*.

4. Number of Things

```
numbers-of-things   ::=   number-of-program-parameters
                          number-of-display-parameters
                          number-of-vertices
                          number-of-surface-descriptors
                          number-of-light-sources
                          number-of-scene-nodes
                          DAG-root-node
                          EOLN .
```

Every scene file begins with 7 integers, giving the number of entries in the various divisions that follow. The one anomaly is the *ambience-descriptor*. There is always exactly one *ambience-descriptor*, so it isn't mentioned in the *numbers-of-things*.

Therefore, the first two lines of the example scene file

```
* scene:  a mirror ball above a green/blue checkerboard
  5 2 4 2 1 4 1
```

mean there are 5 program parameters to follow, 2 display parameters, 4 vertices, 2 surface descriptors, (the single ambience descriptor), 1 light source, and 4 scene nodes. The root of the DAG to be generated is scene node 1.

With the exception of the *ambient-descriptor*, the members of each division are numbered. This number is referred to as *count* in the following text. Within most divisions, these numbers must be consecutive, starting at 1. This is because the program requires a label for a table entry or a DAG node. Within the *program-parameters* and *display-parameters* these restrictions are relaxed.

5. Program Parameters

```
program-parameters   ::=   program-parameters program-parameter  |
                           /* empty */.

program-parameter    ::=   output-parameter  |
                           yhigh-parameter   |
                           ylow-parameter    |
                           xleft-parameter   |
```

xright-parameter |
depth-parameter |
shadows-parameter |
memory-parameter |
brightness-parameter |
background-parameter |
quiet-parameter |
aathreshold-parameter |
aalevel-parameter .

Program parameters control the operation of the program itself: the size of the picture to be generated, where the output goes, the depth of ray tracing, etc. Program parameters are so called because they also appear as options on the Program Call statement. For example, the output file name may be given as a -o option, and this would override any output file name given in the scene file. See Appendix 2 for the form of the Program Call statement.

The table below shows the possible range of values, the default value, and suggested values for the various program parameters. Each will be discussed individually.

parameter	range	default	suggested value
output		to the ikonas	/tmp/junk
ylow	0-511	0	close to the image bottom boarder
yhigh	0-511	511	close to the image top boarder
xleft	0-511	0	close to the image left boarder
xright	0-511	511	close to the image right boarder
depth	0-10	0	as small as possible †
shadows			not present if shadows unwanted
memory	1-24‡	4	as small as possible †
brightness	0.0-1.0	0.0	0.8 for background routine supplied
background	each 0-255	0 0 0	to produce desired background colour
quiet			not present if output unwanted
aalevel	0,2,4,8	2	as small as possible †
aathreshold	5.0-100.0	40.0	40.0

Table 8. Program parameters.

If a program parameter is not present, and does not appear as an option on the program call statement, then the default value is used. Notice that the same program would not accept both brightness and background, and might not accept either aalevel or aathreshold, depending on how it was configured at compile time.

output-parameter ::= *count* output *pathname* EOLN .

This specifies that the final image should be placed in the file specified by *pathname*. If the package was compiled with -Dencode, this file is run length encoded. The encoding scheme used is based on 32 bit words. The high 8 bits specify the number of pixels which follow having the colour specified by the low 24 bits. This colour is given as 8 bits blue, 8 bits green and the low 8 bits red. If the package was not compiled with -Dencode, then the file is a raw frame buffer image, a 512×512 array of 32 bit integers. Each integer contains the colour of a pixel as 8 bits unused, 8 bits blue, 8 bits green, and the low 8 bits red.

If no output-parameter appears in the scene file, and no -o option appears on the program call statement, the output goes directly to the frame buffer.

† see the descriptions of the individual program parameters.
‡ or the maximum virtual address space that the machine is configured for.

yhigh-parameter ::= *count* yhigh integer EOLN .

This gives the uppermost scanline to be rendered. If yhigh is less than 511, the space will be padded with black scanlines. Images are rendered from the top down.

ylow-parameter ::= *count* ylow integer EOLN .

This gives the lowermost scan line to be rendered. If ylow is greater than yhigh, nothing is rendered. Individual scanlines may be rendered by setting ylow equal to yhigh.

xleft-parameter ::= *count* xleft integer EOLN .

This gives the leftmost pixel to be rendered on each scan line. If xleft is greater than 0, the space will be padded with black pixels.

xright-parameter ::= *count* xright integer EOLN .

This gives the rightmost pixel to be rendered on each scan line. If xright is less than 511, the space will be padded with black pixels. Setting both xleft and xright to 256 is a fast way to locate an image on the screen.

depth-parameter ::= *count* depth integer EOLN .

This specifies the recursive depth of ray tracing. At the default setting (0), only primary rays are cast. Reflections and transparency are not modeled, with a commensurate time savings. The default setting is recommended for previewing images.

shadows-parameter ::= *count* shadows EOLN .

This is a switch, setting shadows on. With the default setting (off), objects do not cast shadows. Again, this is recommended for previewing images. Depending on the image, shadows may not add realism.

memory-parameter ::= *count* memory integer EOLN .

The program takes one large block of memory from the system, then does its own allocation. This option sets the size of the block, in Mbytes. The program will report how this buffer has been allocated. For the checkerboard scene given in the example:

```
allocated: 1069148 from bottom, 2952 from top, 2927899 free
```

So it seems that we could have used a memory parameter of 2 Mbytes. The program will also report if the buffer is exhausted:

```
ALLOC: alloc_buffer exhausted, use a larger -m
```

The program halts immediately after this message.

The basic program requires 1 Mbyte of memory. Each texture map requires 1 Mbyte, and it is a good rule of thumb to add a quarter Mbyte for each spline and fractal surface.

brightness-parameter ::= *count* brightness real-number EOLN .

If the program has been compiled with -Dbackground, then the user is expected to provide a procedure Background in the file Render.c, which is called to colour all rays which do not intersect any object in the scene. The rays themselves are parameters to this procedure, so the colours generated may be based on the ray orientation. The brightness parameter controls the overall intensity of these background colours.

background-parameter ::= *count* background *red green blue* EOLN .

Even if the program has not been compiled with -Dbackground, a uniform background colour can still be provided by the background parameter as three integers *red, green* and *blue.*

quiet-parameter ::= *count* quiet EOLN .

This is another switch, setting quiet mode on. Unless this mode is set, the program will report the settings of the program parameters after the scene file has been read, the contents of the object list after the DAG has been preprocessed, and how the programs internal memory buffer has been allocated.

aathreshold-parameter ::= *count* aathreshold real-number EOLN .

This parameter gives a threshold for anti-aliasing and is only valid if the program has been compiled with -Daa. The four corners of a pixel are considered to be the same colour if the sum of the differences in their red, green and blue components is less than this threshold. Otherwise, the pixel is subdivided and the four corners of each subdivision compared to determine if still further subdivision is necessary. Smaller aathreshold values mean better anti-aliasing.

aalevel-parameter ::= *count* aalevel integer EOLN .

This parameter controls the maximum number of recursive subdivisions permitted when doing anti-aliasing and is only valid if the program has been compiled with -Daa. The integer must be one of 1, 2, 4 or 8, and larger values mean better anti-aliasing. At an aalevel of 2, the effective size of the screen is 1024×1024, although additional rays are cast only as necessary (see Section 8.6 in the thesis proper).

6. Display Parameters

```
display-parameters    ::=   display-parameters display-parameter  |
                            /* empty */ .

display-parameter     ::=   eye-point  |
                            sight-direction  |
                            up-direction  |
                            dist  |
                            viewport  .
```

Display parameters determine the location and orientation of the viewer. A left handed coordinate system is used throughout the package. The positive z axis points out of the screen.

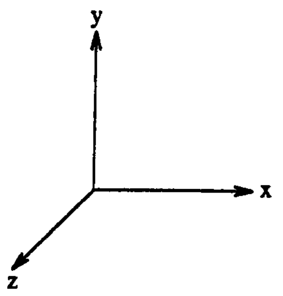


Figure 60. The coordinate system used.

Examples of the display parameters will be given showing the default values. Then each will be discussed individually.

1	eye		0.0 0.0 3.5
2	sight	towards	0.0 0.0 0.0
3	up		0.0 1.0 0.0
4	dist		2.5
5	viewport		2.0 2.0

The defaults are set up so that the viewer is looking from the positive z axis towards the origin, at a visible object space which is (*very* roughly speaking) a cube with corners (1.0,1.0,1.0) and (−1.0,−1.0,−1.0). Actually, the visible space in an infinite pyramid. There is no far clipping plane, and the only effect which approximates a near clipping plane is actually going behind the eye position.

eye-point ::= *count* eye *x y z* EOLN .

This defines the position of the viewer in three-space. *x, y* and *z* are real numbers.

sight-direction ::= *count* sight direction *x y z* EOLN |
count sight towards *x y z* EOLN .

The first form defines the direction in which the viewer is looking, **not** the point at which he/she is looking. In the second form, *x, y* and *z* define the point at which the viewer is looking. The sight vector is calculated from the current eye position, so the eye point should be set before the sight direction. *x, y* and *z* are real numbers.

up-direction ::= *count* up *x y z* EOLN .

This tells the program which way is up. *x, y* and *z* are real numbers. In fact, the up vector determines the direction of increment of the ray path as it passes from scanline to scanline. The direction of the increment along one scanline is determined by the cross product of the up vector with the sight vector.

dist ::= *count* dist real-number EOLN .

This gives the distance from the eye point to the *virtual screen.* See Section 8.2 in the body of the thesis for details.

viewport ::= *count* viewport *height width* EOLN .

This defines the dimensions of the virtual screen. Both *height* and *width* are real numbers.

7. Vertices

vertices ::= *vertices vertex* |
/* empty */ .

vertex ::= *count x y z* EOLN.

Each vertex gives the coordinates of a point in three-space. These points may be referenced by polygon or cylinder scene nodes, to define the corners of the polygon or the end points of the cylinder line segment. *x, y* and *z* are real numbers.

The four points listed under vertices in the example scene file

```
* vertices
1  -1.0 -0.0 -1.0
2  -1.0 -0.0 +1.0
3  +1.0 -0.0 +1.0
4  +1.0 -0.0 -1.0
```

are used as the corners of the checkerboard (after being translated and rotated).

8. Surface Descriptors

```
surface-descriptors   ::=   surface-descriptors surface-descriptor  |
                            /* empty */ .

surface-descriptor    ::=   lambert-surface  |
                            textured-lambert-surface  |
                            phong-surface  |
                            textured-phong-surface  |
                            cook-surface  .
```

A surface descriptor is associated with every primitive object named in the *scene-nodes* division (although several objects may share a descriptor). The descriptor expresses the way in which the interaction of light with the object will be modelled. Accordingly, surface descriptors can become quite complex. The things which can be specified are:

1. The degree of surface reflectiveness.

2. The degree of transparency and refractive index.

3. The surface colour, if the object is to be one solid colour, a texture map otherwise.

4. The type of shading to be used: either Lambert's law, the Phong illumination model, or the Cook-Torrance model. Each of these models has its own variety of parameters for diffuse reflection, specular reflection etc.

These four things are (almost) completely independent. For instance you can specify a transparent texture map with a phong highlight, or a perfectly reflective copper surface. This also means that you can specify optically unrealistic surfaces, such as a matte Lambert surface with a high degree of reflectiveness.

It is especially important to realize the difference between the ray tracing parameters (reflectiveness and transparency) and the shading model parameters. The ray tracing parameters determine the contribution to the surface colour that the reflected (or refracted) ray have. The shading model parameters determine the the contribution that the various light sources have. These two things are completely independent.

It is also important to have the depth program parameter set appropriately for a scene which has reflective or transparent surfaces. For example, if there were two brown reflective spheres in the scene, the brightness set to 0.8 and the depth set to 1, the spheres themselves would reflect the background colour, and each other. However, the spheres in reflection would appear brown. This is because the ray tracing stops after one reflective bounce - and at that level it sees a brown sphere not a reflective brown sphere.

The form of the first line of any surface descriptor is the same.

count (lambert|phong|cook) (textured|normal) *reflect refract index* EOLN.

For example from the checkerboard scene

```
2 phong normal 1.0 0.0 0.0
```

The first word specifies the illumination model to be used: Lambert, Phong, or Cook-Torrance. Notice that the keyword is all in lower case in the scene file.

The second word specifies whether the surface is one solid colour, or texture mapped. For texture mapping, a surface is divided into a two dimensional 512×512 grid. The colour of a point (u,v) on this grid is specified by the colour of a corresponding point (u,v) in the image in a specified file. The orientation of a texture map is best described as random - if it doesn't come out right first time, try re-ordering the vertices.

reflect is a real number in the range [0.0,1.0] representing how much light is reflected from the surface. 1.0 is perfectly reflective. However, a reflectiveness of 1.0 should be avoided except where the shading parameters make the object itself almost invisible (e.g. a mirror or bubble). This is because the contribution of the reflected ray is basically additive, so the reflection may come out looking brighter than the reflected object itself.

refract is a real number in the range [0.0,1.0] representing how much light is transmitted through the surface. 1.0 is perfectly transparent. The same remarks as made for *reflect* apply here also.

index is a real number which gives the refractive index of the surface at (say) 550 nanometers. The refractive index is only used if *transparency* is non-zero. However, some number must appear there in any case. Typical values range from 0.5 to 1.5.

Following this standard first line are a variable number of lines depending on the illumination model being used. See the Section 9 in body of the thesis for explanations of the meanings of the various coefficients, and how they are used.

lambert-surface ::= *count* lambert normal *reflect refract index* EOLN
diffuse EOLN
colour-vector EOLN .

This surface is one solid colour illuminated according to Lambert's Law.

diffuse is a real number in the range [0.0-20.0] used as the diffuse coefficient in the calculation.

The *colour-vector* consists of 33 real numbers representing the surface reflectance (colour) between 380 and 700 nanometers at 10 nanometer intervals. These numbers are arranged as 3 lines of 10 and 1 line of 3.

Typical values are:

```
* surface: flat blue
1 lambert normal 0.0 0.0 0.0
  7.0
0.1273 0.1337 0.1432 0.1528 0.1751 0.1910 0.2069 0.2228 0.2133 0.1910
0.1751 0.1655 0.1528 0.1369 0.1210 0.1050 0.0955 0.0796 0.0637 0.0573
0.0477 0.0392 0.0382 0.0350 0.0350 0.0382 0.0382 0.0414 0.0446 0.0955
0.1273 0.1592 0.2228
```

textured-lambert-surface ::= *count* lambert textured *reflect refract index* EOLN
diffuse EOLN
(*pathname*|last) *umod vmod* EOLN .

This surface is a texture map illuminated according to Lambert's Law.

diffuse is a real number in the range [0.0-20.0] used as the diffuse coefficient in the calculation.

pathname is the path to a file which contains a run-length encoded image. The encoding format as described for the output program parameter.

umod is the number of times this image will be repeated in the u parametric direction (usually the vertical image direction). *vmod* is the number of times this image will be repeated in the v parametric direction (usually the horizontal image direction). Both *umod* and *vmod* are real numbers greater than 0.0.

"last" specifies that the texture map of the immediately previous surface is to be used for this surface descriptor as well. Although a pointer to the same 512×512 array is used, none of the other parameters (shading-model, *reflect*, *refract*, *index*, *difuse*, *umod*, *vmod*) need be the same. Each texture map occupies 1

Mbyte of memory, so using last represents a considerable savings.

The checkerboard texture map in the example was

```
* surface:  a checkerboard texture map
2 lambert textured 0.0 0.0 0.0
  7.0
  /u/majsweeney/textures/checkerboard  1.0 1.0
```

phong-surface ::= *count* phong normal *reflect refract index* EOLN
diffuse specular exponent EOLN
colour-vector EOLN .

This surface is one solid colour illuminated according to the Phong model.

diffuse is a real number in the range [0.0-20.0] used as the diffuse coefficient in the calculation.

specular is a real number in the range [0.0-20.0] used as the specular coefficient in the calculation. This controls the brightness of the highlight.

exponent is a real number in the range [10.0-100.0] used as the exponent to which $(V \cdot R)$ is raised. Larger numbers imply a more concentrated highlight.

The *colour-vector* consists of 33 real numbers representing the surface reflectance (colour) between 380 and 700 nanometers at 10 nanometer intervals. These numbers are arranged as 3 rows of 10 followed by 1 row of 3.

Typical values are:

```
* surface:  reflective phong red
3 phong normal 0.6 0.0 0.0
  7.0 3.0 25.0
0.0127 0.0095 0.0064 0.0032 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0127 0.0159 0.0286
0.0477 0.0637 0.0987 0.1273 0.1592 0.1910 0.2228 1.2000 1.2000 1.2000
1.2000 1.2000 1.2000
```

textured-phong-surface ::= *count* phong textured *reflect refract index* EOLN
diffuse specular exponent EOLN
(*pathname* | last) *umod vmod* EOLN .

This surface is a texture map illuminated according to the Phong model. *diffuse*, *specular* and *exponent* are as explained directly above. *pathname*, last, *umod* and *vmod* are as explained for a textured lambert surface.

So, a transparent texture map of the mandrill with a phong highlight is

```
* surface:  transparent phong mandrill
4 phong textured 0.0 0.8 1.2
  7.0 1.0 15.0
  /u/majsweeney/textures/mandrill  1.0 1.0
```

cook-surface ::= *count* cook normal *reflect refract index* EOLN
diffuse EOLN
normal-colour-vector EOLN
specular *m* EOLN
fresnel-colour-vector-1 EOLN
fresnel-colour-vector-2 EOLN

....
fresnel-colour-vector-66 EOLN .

This surface is one solid colour illuminated according to the Cook-Torrance model. Texture mapped Cook-Torrance surfaces are not supported.

diffuse is a real number in the range [0.0-1.0] used as the diffuse coefficient in the calculation.

The *normal-colour-vector* consists of 33 real numbers representing the normal (in the sense of perpendicular) surface reflectance between 380 and 700 nanometers at 10 nanometer intervals.

specular is a real number in the range [0.0-1.0] used as the specular coefficient in the calculation.

m is a real number in the range [0.1-1.0] representing the rms deviation of the facet slope distribution function. Larger numbers imply duller surfaces.

The *fresnel-colour-vector*'s are values from the fresnel equation at 66 angles between 0 and 90 degrees and 33 wavelengths between 380 and 700 nanometers.

A typical Cook-Torrance surface descriptor follows. Only the first two of the 66 fresnel colour-vectors are shown. Refer to Section 9.9 in the body of the thesis for an explanation of how to calculate them all, or use the cook utility mentioned in the next section.

```
* surface  reflective copper
8 cook normal 0.8 0.0 0.0
  0.0
* normal reflectance vs wavelength
0.0223 0.0242 0.0261 0.0280 0.0302 0.0337 0.0372 0.0407 0.0446 0.0497
0.0547 0.0602 0.0653 0.0700 0.0748 0.0796 0.0844 0.0955 0.1066 0.1311
0.1560 0.1757 0.1958 0.2044 0.2133 0.2174 0.2219 0.2263 0.2308 0.2330
0.2355 0.2381 0.2403
  1.0  0.3
*  fresnel reflectance vs wavelength & angle
0.0700 0.0760 0.0820 0.0880 0.0950 0.1060 0.1170 0.1280 0.1400 0.1560
0.1720 0.1890 0.2050 0.2200 0.2350 0.2500 0.2650 0.3000 0.3350 0.4120
0.4900 0.5520 0.6150 0.6420 0.6700 0.6830 0.6970 0.7110 0.7250 0.7320
0.7400 0.7480 0.7550
0.0700 0.0760 0.0820 0.0880 0.0950 0.1060 0.1170 0.1280 0.1400 0.1560
0.1720 0.1890 0.2050 0.2200 0.2350 0.2500 0.2650 0.3000 0.3350 0.4120
0.4900 0.5520 0.6150 0.6420 0.6700 0.6830 0.6970 0.7110 0.7250 0.7320
0.7400 0.7480 0.7550
```

9. Useful Utilities

There are three useful auxiliary programs which go along with the package: tovector, cook, and rl.

Usage: tovector <red> <green> <blue>, output to stdout and stderr.

tovector takes a colour specified as an rgb triple on the program call statement and prints the corresponding colour-vector on the standard output. It also reconverts this colour-vector to an rgb value and prints this on the standard error as a check.

Usage: cook <outfile>, input from stdin

cook prompts the user for the normal reflectance between 380 and 700 nanometers at 10 nanometer intervals, and other parameters, then calculates a complete cook surface descriptor from the values entered. See Section 9.2 in the body of the thesis for sources of normal reflectances. The descriptor is written on the output file named on the program call statement.

Usage: rl <filename>, output to the frame buffer

rl run-length decodes a file produced by the package, and displays it on the frame buffer.

10. Ambience Descriptor

ambience-descriptor ::= *ambient-reflection-constant* EOLN
colour-vector EOLN.

The ambience descriptor gives the amount of non-directional (ambient) light in the scene. There is always exactly one ambience descriptor in a scene file.

The *ambient-reflection-constant* is a real number in the range [0.0-0.001] which corresponds to k_a of Section 9.6 in the body of the thesis.

The *colour-vector* consists of 33 real numbers representing the intensity of ambient light between 380 and 700 nanometers at 10 nanometer intervals, and corresponds to $I_{a\lambda}$ in Section 9.6. As with surface descriptors, these 33 values are arranged as 3 rows of 10 followed by 1 row of 3.

A typical ambience-descriptor follows.

```
* ambience
 0.00050
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6
```

In the example above, the *colour-vector* is CIE Standard Illuminate D6500, which is the colour of daylight on an overcast day.

11. Light Sources

light-sources ::= *light-sources light-source* |
/* empty */ .

light-source ::= *source-at-infinity* |
source-in-scene .

This section contains descriptions of all explicit light sources. Associated with each is a position in space, a colour, and a brightness. Only point sources are modelled. The source may be modelled as being at infinity, or as being at a defined location (possibly within the scene).

source-at-infinity ::= *count* infinity *x y z dw* EOLN
colour-vector EOLN .

Here *x y z* is a direction vector to the light source, which is assumed to be at infinity.

dw is the solid angle that the light subtends. This is the projected area of the light source divided by the square of the distance to it, and is required by the Cook-Torrance model. However, a moments thought reveals that this is a meaningless concept when applied to a point source at infinity. So, in fact, *dw* acts as the brightness control which applies to all shading models. Useful values are in the range [0.0001-0.005].

The *colour-vector* consists of 33 real numbers representing the intensity (colour) of the light source between 380 and 700 nanometers at 10 nanometer intervals. Again, the *colour-vector* is arranged as 3 rows of 10 numbers followed by 1 row of 3.

Typical values are:

```
* light sources
1  infinity 1.0 1.0 1.0 0.0006
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6
```

Once again the example *colour-vector* is the CIE Standard Illuminate D6500. However, there may be many light sources, each a different colour, and the ambience may be still another colour.

source-in-scene ::= *count* inscene *x y z dw* EOLN
colour-vector EOLN .

Here, *x y z* is the actual position of the light source, which may in fact be within the scene. However, light sources are **not** tested directly for ray intersection. This means that any light source is itself invisible, only its effects on the surroundings can be seen. Thus, the most effective way to model an actual, visible light source within the scene is as a transparent sphere with a light source at its center.

dw and the *colour-vector* are as described above. Note that shadows from this type of light source are more expensive to calculate than for sources at infinity.

Typical values are:

```
2  inscene  0.1 0.0 0.0 0.001
 50.0  54.6  82.8  91.5  93.4  86.7 104.9 117.0 117.8 114.9
115.9 108.8 109.4 107.8 104.8 107.7 104.4 104.0 100.0  96.3
 95.8  88.7  90.0  89.6  87.7  83.3  83.7  80.0  80.2  82.3
 78.3  69.7  71.6
```

12. DAG Formation

The basic form of a scene node is

count keyword down right other-parameters EOLN .

for example, from the checkerboard scene

```
1  rotate  02 04      x -22.0
```

Both *down* and *right* are the indices of other scene nodes. A directed acyclic graph (DAG) is built, starting from the scene node specified as the *DAG-root-node* in the *numbers-of-things*, and following the path indicated by the *down* and *right* pointers.

The significance of *down* vs. *right* is that if the node represents a transformation, the transformation will be applied only to the node specified by the *down* pointer, the node specified by the *right* pointer will be unaffected. A *down* or *right* index of 0 indicates a NIL pointer in the DAG.

Until one gets familiar with the way that DAGs are formed, it is probably useful to draw the desired DAG before writing the scene file. Consider the following scene nodes.

```
1 translate 04 02    +0.0 -0.7 +0.0
2 translate 04 03    +0.0 +0.0 +0.0
3 translate 04 00    +0.0 +0.7 +0.0
4 translate 07 05    -0.7 +0.0 +0.0
5 translate 07 06    +0.0 +0.0 +0.0
6 translate 07 00    +0.7 +0.0 +0.0
7 rotate    08 00    x -22.0
8 scale     09 00    0.2 0.2 0.2
```

```
 9 polygon   00 10 1   4   2 3 4 5
10 polygon   00 11 1   4   6 7 8 1
11 polygon   00 12 1   4   1 8 3 2
12 polygon   00 13 1   4   5 4 7 6
13 polygon   00 14 1   4   1 2 5 6
14 polygon   00 00 1   4   8 7 4 3
```

This scene describes 9 cubes formed by

(1) three translations in y of

(2) three translations in x of

(3) a rotation about the x axis of

(4) a scaling (about the origin) of

(5) the six polygons defining a cube.

The corresponding DAG is

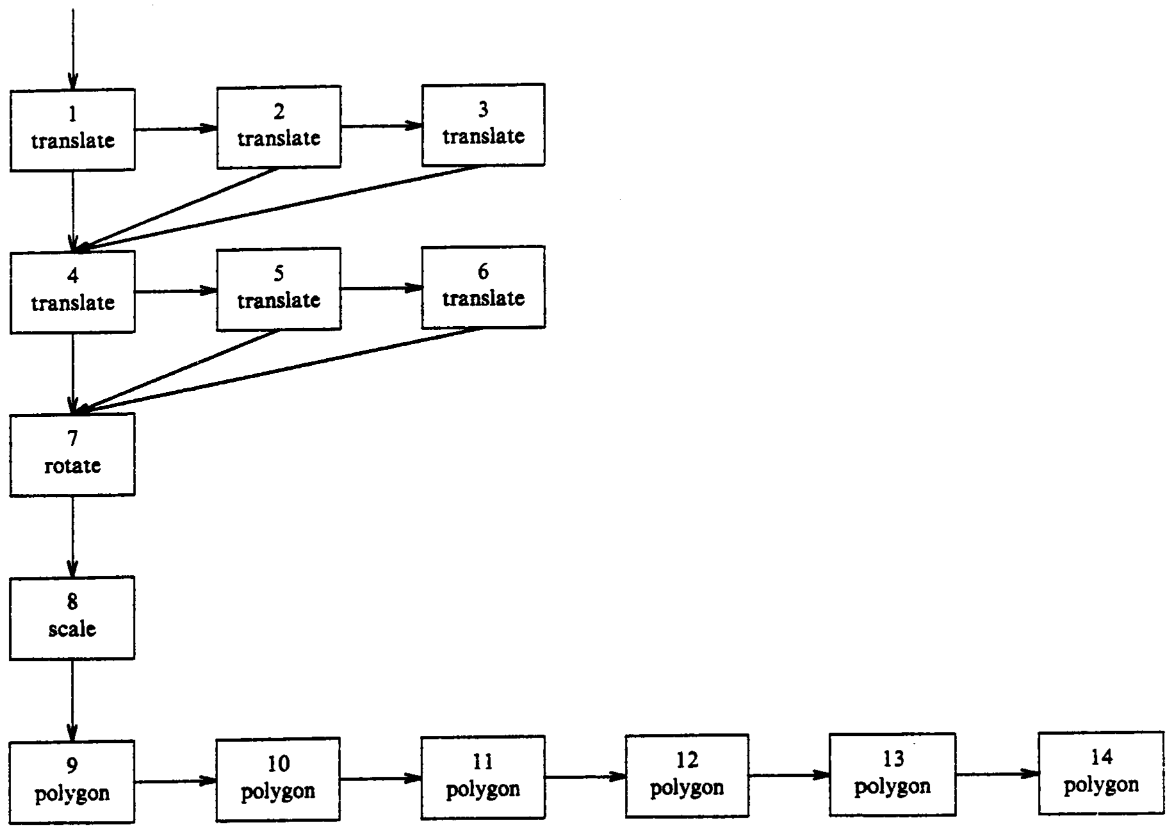


Figure 61. The corresponding DAG.

The scene produced is as follows.

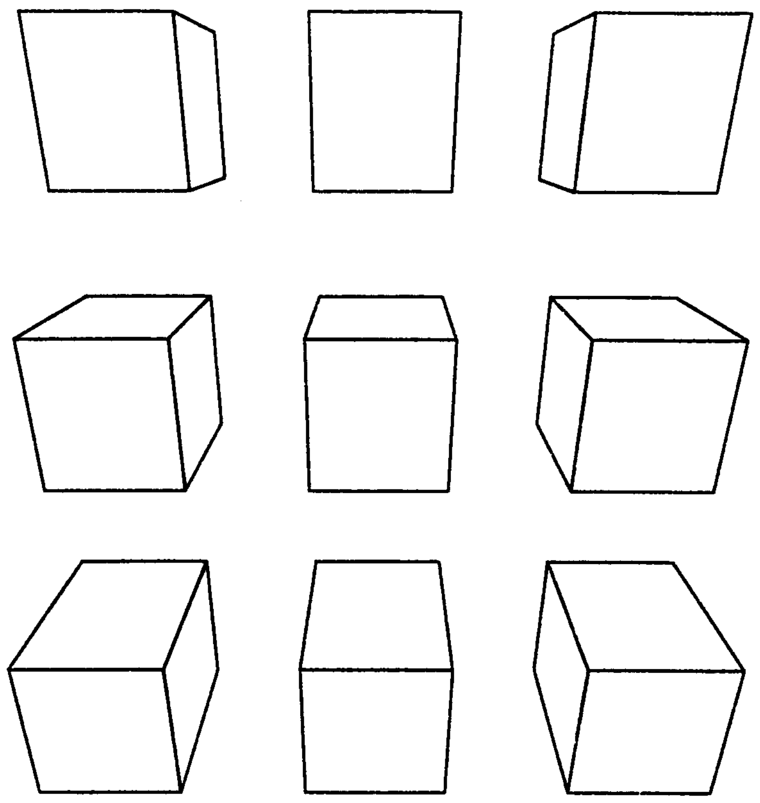


Figure 62. A line drawing of the scene.

There is one more point worth mentioning. The right pointers are processed into object nodes before the down pointers. Also, the object list is a fifo structure: the first object processed becomes the last member of the list. This is important when there are spline or fractal surfaces in the object list.

It is advantageous to have these nodes **last** on the object list, because the intersection processor compares the distance to the outer bounding box to the current ray $t_{\min}$. If $t_{\min}$ is less, then the processor ignores the object, with a considerable savings in time. Thus spline and fractal nodes should appear near the **top** of the DAG. The program will print out the object list for external inspection. One should check that spline and fractal nodes appear last.

13. Scene Nodes

scene-nodes	::=	*scene-nodes scene-node* |
		/* empty */.
scene-node	::=	*translate-node* |
		rotate-node |
		scale-node |
		sphere-node |
		cylinder-node |
		polygon-node |

fractal-node |
spline-node |
box-node |
dummy-node .

Scene nodes define the objects of the image, and how they are arranged in space. A scene node can represent either a transformation (translation, rotation, scaling) or a primitive object (sphere, cylinder, polygon, fractal or spline).

translate-node ::= *count* translate *down right x y z* EOLN .

This node specifies that a translation of (x,y,z) be applied to the portion of the DAG connected to its down pointer. x, y and z are real numbers.

An example of a translate node is

```
2  translate  03 00     0.0 -0.8 0.0
```

scale-node ::= *count* scale *down right x y z* EOLN .

This node specifies that a scaling of (x,y,z) be applied to the portion of the DAG connected to its down pointer. x, y and z are real numbers.

For example

```
3  scale  04 00     0.5 0.5 0.5
```

A warning message is generated and the node ignored if any of the values are 0.0.

rotate-node ::= *count* rotate *down right* (x|y|z) *theta* EOLN .

This node specifies that a rotation be applied to the portion of the DAG connected to its down pointer. The axis of rotation is chosen from x, y, or z. The rotation is around the selected axis by *theta* **degrees** in a clockwise direction looking down the axis towards the origin. *theta* is a real number.

An example of a rotate node is

```
4  rotate  05 00     x -22.0
```

sphere-node ::= *count* sphere *down right surface-index*
x y z radius (x|y|z) (x|y|z) EOLN .

All primitive-object nodes contain an additional field besides *down* and *right*. *surface-index* is the number of the surface descriptor containing the information about how light interacts with this object.

The remaining fields of a sphere scene node are the center, the radius, and two axis. The center is given as three real numbers *x y z*. The *radius* is also a real number, and is independent of scaling transformations. A sphere radius 0.1 will appear approximately 100 pixels wide. Two sets of (x|y|z) control the orientation of a texture map.

An example sphere node follows.

```
5  sphere 00 06 1   0.0 0.0 0.0 0.1  x y
```

cylinder-node ::= *count* cylinder *down right surface-index*
radius v1 v2 EOLN .

This node represents a cylinder, which is defined by a line segment and a radius. *radius* is a real number, and is independent of scaling transformations. A cylinder radius 0.1 will appear approximately 100 pixels wide. *v1* and *v2* indicate members of the *vertices* section of the scene file which will give the end points of the line segment. *surface-index* is as described above for spheres.

A example cylinder follows.

```
6  cylinder 00 07 1   0.15  1 2
```

polygon-node ::= *count* polygon *down right surface-index*
n v1 v2 ... vn EOLN .

This node represents the polygon formed by drawing *n* lines from *v1* to *v2* ... to *vn* to *v1*. As for cylinders, *v1*, *v2*, etc. indicate members of the *vertices* section of the scene file. Notice that only triangles and rectangles are supported at present. *surface-index* is as described above for spheres.

A example polygon follows.

```
7  polygon 00 08 1   4  1 2 3 4
```

fractal-node ::= *count* fractal *down right surface-index*
divisions h offset seed (constrained|free) EOLN .

This node represents a fractal triangle, whose initial vertices are (−1.0,0.0,1.0), (1.0,0.0,1.0) and (0.0,0.0,−1.0). Of course, the final fractal may be translated, rotated and/or scaled arbitrarily.

divisions is an integer in the range [1,9] which determines the number of subdivisions done to generate the fractal surface. Solid coloured fractals usually start to look interesting at 6 divisions, and there is little change above 8-9 divisions. Usually, fewer subdivisions are necessary for texture-mapped fractals.

h is a real number in the range [0.5,1.5] which represents the fractal dimension of the surface. As *h* rises from 0.5 to 1.5, the surface becomes smoother (the variation in y at the lower levels of subdivision becomes less). Below 0.5 the surface is unrealistically jagged, above 1.5 the surface becomes completely flat. Suggested values are 0.9 for mountains and 1.3 for rolling hills.

offset is a real number in the range [-1.0,1.0] which is effectively the mean of the random numbers used. As offset increases, the perturbations of the subdivision corners tend to rise more than fall.

seed is an integer which is used to seed the random number generator. The choice of seed value has the greatest effect on the final form of the surface. Unfortunately, there are no rules as to what makes a good seed.

A outer edge of a fractal may be constrained or free. In constrained mode the perimeter of the fractal remains in the same (possibly transformed) plane as the original triangle. In free mode, the edge is subject to the same random variations as is the rest of the surface.

An example fractal follows.

```
39 fractal    00 00 1   5 1.3 0.5 9772 constrained
```

spline-node ::= *count* spline *down right surface-index*
    nu (single|double|triple|closed)
    nv (single|double|triple|closed)
    divisions overlap EOLN
count x y z EOLN
count x y z EOLN
....
count x y z EOLN .

This node represents a free-form spline surface. *nu* is the number of vertices in the u parametric direction. This is followed by a word representing the end condition to be applied in the u parametric direction. *nv* is the number of vertices in the v parametric direction. This is followed by a word representing the end condition to be applied in that parametric direction.

divisions is the number of sub-intervals into which the intervals between knots are divided during the refinement preprocessing. *overlap* is the overlap between successive bounding boxes in the leaves of the tree data structure which represents the spline. For information on appropriate settings of these parameters see Section 6 in the body of the thesis, but 2 and 0.5 are reasonable initial values.

Following this are *nv* groups of *nu* rows of vertices which make up the control polygon. Each vertex is an integer *u* count followed by a triple of real numbers.

A tube parallel to the y axis, with a bulge in its center is

```
4 spline   00 00   1 4 closed 5 double 8 0.6
1 -0.2  0.8 -0.2
2 -0.2  0.8  0.2
3  0.2  0.8  0.2
4  0.2  0.8 -0.2
1 -0.3  0.6 -0.3
2 -0.3  0.6  0.3
3  0.3  0.6  0.3
4  0.3  0.6 -0.3
1 -0.4  0.4 -0.4
2 -0.4  0.4  0.4
3  0.4  0.4  0.4
4  0.4  0.4 -0.4
1 -0.3  0.2 -0.3
2 -0.3  0.2  0.3
3  0.3  0.2  0.3
4  0.3  0.2 -0.3
1 -0.2  0.0 -0.2
2 -0.2  0.0  0.2
3  0.2  0.0  0.2
4  0.2  0.0 -0.2
```

box-node ::= *count* box *down right* EOLN .

This node causes a bounding box to be created which contains all primitive objects in the portion of the DAG connected to its down pointer. Only if the ray intersects this box will it be tested against the contents of the box.

For example:

```
1 rotate  02 00  x -22.0
2 box     03 00
3 box     04 05
4 polygon 00 00  1   3  1 2 3
5 box     06 00
6 scale   07 00  0.5 0.5 0.5
7 polygon 00 00  1   3  2 3 4
```

Here the ray is tested against the super box in node 2. If it intersects that box, it is tested against the boxes in nodes 3 and 5. If it intersects the box in node 3, then it is tested against the polygon in node 4. If it intersects the box in node 5, then it is tested against the polygon in node 7. Both polygons are rotated -22 degrees around the x axis, and the polygon in node 7 is scaled by half.

This example is artificial in that it takes approximately a third as long to test for ray-bounding box intersection as it does to test for ray-polygon intersection. However if the scene contains a number of primitive objects clustered together, placing a bounding box around the group can result in significant time savings.

dummy-node ::= *count* dummy *down right* EOLN .

This node is a no-op, and is useful to organize a scene file, as a place holder, or as a replacement for a node you want to remove temporarily. Recall the earlier statement that the meta-symbol EOLN designates the end of a line, possibly preceded by ignored characters. Thus, to temporarily remove a polygon node

```
7  polygon 00 08    4  24 24 26 27
```

all that is necessary is to change "polygon" into "dummy"

```
7  dummy   00 08    4  24 24 26 27
```

Notice that this effects only the one node, and not anything attached to its right or down pointers.

14. Hints

It is very easy to set up a scene file to take ten (or fifty) times as long to compute as is necessary. This section contains a miscellany of suggestions on how to get images at a reasonable rate.

- An anti-aliasing level of more than 2 is seldom necessary.
- Avoid having reflective objects reflecting other reflective objects. If this is necessary, then severly limit the recursion of ray tracing with the depth program parameter.
- Once the extent of the image on the screen is known, tightly bound it with the xleft, xright, ylow and yhigh program parameters.
- Put super-boxes around clusters of objects in the scene, for instance around the 6 polygons which form a cube. Put super-boxes around clusters of super-boxes.
- Arrange the object list so that splines and fractals appear last.
- Shadows are often unnecessary. An object which is both reflective and transparent is almost never necessary.
- Use texture maps for parts of the image which appear only in reflections.
- Use as low a number for the *overlap* parameter in a spline node as you can get away with, without having holes appear in the surface.
- Set up the image using no anti-aliasing, shadows, reflections or transparency.

Appendix 2. The Program Call Statement

1. Introduction

This appendix describes the syntax of the Program Call statement. Everything which can be specified as an option on the call statement can also be specified in the scene file (Appendix 1, Section 5). However, call statement options take precedence.

Thus if a scene file test contained the program parameter

```
1 yhigh 100
```

but was rendered using the call

```
trace -yh 511 test
```

the image would in fact begin at scanline 511.

2. Call Statement Options

To invoke the ray tracing package from a UNIX shell, type

trace *options scene-file*

where *options* are members of the various arguments listed below, and the *scene-file* contains a description of the scene to be rendered (Appendix 1).

The following options may appear in any order. In each case, the flag -X must be separated from the *pathname|integer|real-number* argument by a space. The actions of some options (as noted) are dependent on how the package was configured at compile time.

-o *pathname*
: This specifies that the final image should be placed in the file specified by *pathname*. If no -o option appears on the program call statement, and no output-parameter appears in the scene file, the output goes directly to the Adage/Ikonas.

-yh *integer*
: This gives the uppermost scanline which will be rendered. The default is 511. If *integer* is less than the default, the space will be padded with black scanlines. Images are rendered from the top down.

-yl *integer*
: This gives the lowermost scan line which will be rendered. The default is 0. If ylow is greater than yhigh, nothing is rendered. However, individual scanlines can be rendered by setting ylow equal to yhigh.

-xl *integer*
: This gives the left most pixel on each scan line which will be rendered. The default is 0. If *integer* is greater than the default, the space will be padded with black pixels.

-xr *integer*
: This gives the right most pixel on each scan line which will be rendered. The default is 511. If *integer* is less than the default, the space will be padded with black pixels.

-d *integer*
: This specifies the recursive depth of ray tracing to be carried out. At the default setting (0), only primary rays are cast. Reflections and transparency are not modeled, with a commensurate time savings.

-s

This is a switch, setting shadows on. With the default setting (off), objects do not cast shadows, again with a commensurate savings of time.

-m *integer*
The program takes one large block of memory from the system, then does its own allocation. This option sets the size of the block, in Mbytes. The default is usually 4, depending on the value of the constant "size" at compile time. The program will report how this buffer has been allocated. The program will also report if the buffer is exhausted, and halt immediately.

-b *real-number*
If the program has been compiled with -Dbackground, then the user is expected to provide a procedure Background in the file Render.c, which is called to colour all rays which do not intersect any object in the scene. The -b option controls the overall intensity of these background colours. Default is 0.0.

-b *integer integer integer*
Even if the program has not been compiled with -Dbackground, a uniform background colour can still be provided by the -b option as three integers: red, green and blue respectively. Default is 0 0 0.

-q
This is another switch, setting quiet mode on. Unless this mode is set, the program will report the settings of the program parameters after the scene file has been read, the contents of the object list after the DAG has been preprocessed, and how the programs internal memory buffer has been allocated.

-t *real-number*
This option gives a threshold for anti-aliasing and is only valid if the program has been compiled with -Daa. The four corners of a pixel are considered to be the same colour if the sum of the differences in their red, green and blue components is less than this threshold. Otherwise the pixel is subdivided, and the four corners of each subdivision are compared to determine if still further subdivision is necessary. The default is 40.0, and smaller values mean better anti-aliasing.

-a *integer*
This option controls the maximum number of recursive subdivisions permitted when doing anti-aliasing and is only valid if the program has been compiled with -Daa. The *integer* must be one of 1, 2, 4 or 8. The default is 2, and larger values mean better anti-aliasing.

Appendix 3. Configuring the Package

1. Package Structure

The package is divided among 10 files and a makefile. Very brief descriptions follow.

defs.h
: This file contains all #defines, typedefs, and macros used by the package.

var.c
: All global variables.

driver.c
: This is the main driver, which calls ReadScene, Preprocess and Render in sequence.

ReadScene.c
: ReadScene.c contains everything necessary to convert a scene file into a DAG and a set of tables.

Preprocess.c
: Preprocess.c contains everything necessary to convert a DAG into a linked list of object nodes.

Render.c
: This file drives the ray-tracing routine, does the anti-aliasing, converts *colour-vectors* returned by the shader into rgb values, and run-length encodes the final file.

Intersect.c
: Intersect.c is the intersection processor and contains all of the specialized ray-object intersection routines.

ICube.m
: This routine, which performs ray-bounding box intersections, is by far the most time critical in the package. Accordingly, it is coded in Monol. It is expected that as time permits, more of the intersection and shading routines will migrate to Monol.

Shade.c
: Shade.c is the shader.

Debug.c
: This file contains various debugging aids: routines to write out *colour-vectors*, the ray_info data structure, vectors, etc.

2. Configuring the Package

The package makes heavy use of the conditional compilation facility provided by the C programming language [Kerninghan78]. For those unfamiliar with this facility, a brief explanation follows. A line of the form

#ifdef *identifier*

checks whether the *identifier* is defined, that is, has been the subject of a #define control line, or a -D compiler option. A line of the form

#ifndef *identifier*

checks whether the *identifier* is undefined.

Both forms may be followed by an arbitrary number of lines, possibly containing a control line

#else

and then by the line

```
#endif
```

If the checked condition is false, then all lines between the test and the #endif or the #else are ignored. If the condition is true and an #else is present, then all lines between the #else and the #endif are ignored. These constructs may be nested.

In all, there are 9 identifiers which are subjects of #ifdefs in the package. They are listed below, together with their effect.

aa
This switches the anti-aliasing routine on. If aa is not defined, exactly one ray is cast per pixel (and the program runs faster). Otherwise, 4 rays are cast to the corners of each pixel. The colours returned are compared, and if the difference exceeds a threshold the pixel is recursively subdivided until either the colours at the corners of the subdivided pixel match, or a recursion limit is exceeded.

encode
If encode is defined, the output file is run-length encoded. Otherwise, it contains a raw frame buffer image. This may be up to 512×512×4 bytes, depending on the yhigh, ylow, xleft and xright settings.

background
If background is defined, the user is expected to provide a procedure Background in the file Render.c, which is called to colour all rays which do not intersect any object in the scene. The rays themselves are parameters to this procedure, so the colours generated can be based on the ray orientation.

ntsc
This specifies that the ntsc recommended phosphor chromaticity matrix should be used when converting from tristimulus values (see Section 9.1 in the body of the thesis) to rgb.

electrohome
This specifies that the phosphor chromaticity matrix of an Electrohome ECM 1301 colour monitor should be used when converting from tristimulus values to rgb. Either ntsc or electrohome must be defined.

monol
Time critical routines are coded in Monol. However, in the interest of portability, C versions of these routines also reside in the package. Defining "monol" selects the faster routines.

sequential
If you are sure that sequential calls to malloc give sequential memory locations, then defining "sequential" gives you better control over memory usage. Otherwise, because of the inner workings of malloc, a program with a memory program parameter of 3 and one in which memory is 4 may in fact be the same size.

corrected
Direct texture mapping onto spheres and cylinders causes distortions in the mapped image, which may or may not be desired. If "corrected" is defined, these distortions are minimized at the cost of one call of arccos for each successful ray-sphere and ray-cylinder intersection.

cgraph
If cgraph is defined, the refined control vertex matrix of a spline surface is dumped into a file called "vertices". The file format is two 32 bit integers giving the number of control points in the u and v directions respectively, followed by v groups of u points. Each point consists of three 32 bit real numbers. The control graph is independent of the observer's position and orientation, and no perspective transformation has been applied.

Also, the identifier size (all lower case) must be defined for the package to compile. This sets the default size of the program's allocation buffer. Thus a typical compilation of the program is

```
cc -O -Dsize=4 -Daa -Dencode -Delectrohome -Dmonol -Dsequential *.c
```

Appendix 4. A Structured VAX Assembler Language

1. Introduction

A definition of the Monol programming language is given. Monol runs on the VAX† family of computers under the UNIX‡ operating system. It is designed to provide an easier vehicle for writing machine code than the VAX assembler language. It gives the programmer unambiguous control over the execution of his/her program, while using abstract control structures.

A note about the name: Monol is not an acronym ("my own noodly optimization language"). It is an abbreviation for monolithic, a property of the code generated by macro substitutions.

1.1. Scope of this Manual

This manual describes a preliminary version of Monol, implemented in C running on a VAX 11/780 under 4.1bsd UNIX. The reader is assumed to have experience with the VAX assembler language [Vax81], with C [Kerninghan78], and with running programs in the UNIX environment [Unix83].

This section gives a brief introduction to the language. A formal definition is given in Section 2. Some practical information on how to run the compiler follows in Section 3, a larger programming example in Section 4, and the collected syntax in Section 5.

1.2. Overview of the Language

Monol is not for the faint of heart. If you ask it to compare a double constant and the string "compile me", it will. It will attempt to compile a for-statement into an aobleg opcode, and a switch-statement into a jump table, no matter what. Monol assumes some knowledge on the part of the programmer. You have been warned.

There is a strong resemblance to C. However, only six data types are available: byte, word, longword, float, double, and block(n). An identifier of type block(n) is bound to the first n contiguous bytes of memory. There are no multi-dimensional arrays, records or pointer types.

Monol is designed to be directly translatable into assember. This is most evident in assignment statements. The expression on the right hand side can have at most two operands. A unique opcode is determined by the operator and the type of the left hand operand if two operands are present, otherwise by the type of the single operand. Because of this, there are explicit constants for each of the data types. For example

```
main()
{
  int i,j;
  i=1;
  j=1B+i;
}
```

is translated into

```
        .data
        .text
        .align  1
        .globl  _main
_main:  .word   0
        subl2   $8,sp
        movl    $1,-4(fp)
```

† VAX is a trademark of the Digital Equipment Corporation.
‡ UNIX is a footnote of Bell Laboratories.

```
        addb3   -4(fp),$1,-8(fp)
        ret
```

Also, any identifier may be considered to be any type by postfixing with "@"*type*.

Control structures include an if-then-else, switch, for, while, repeat, break and goto statement. Primary conditions (those involving "==", "<" and other relational operators) can have only two operands, but they can be built into arbitrarily complex conditions with "&&", "||" and "!" operators.

The calling sequence is identical to that used by C, so that the UNIX libraries of C functions may be used. In particular, the standard input/output functions are available. /lib/libc.a is automatically loaded, as it is in C.

Monol also offers an expanded macro definition facility. Macros may have parameters and/or local variables, may be contained on multiple lines, and may be directly or mutually recursive.

1.3. Design Philosophy

If there is one guiding principle in Monol, it is that the programmer have explicit control over the assembler code produced. The Monol compiler will not make any decisions that are not well defined in the code.

A case in point is the &~ operator. VAX assembler lacks a straightforward logical *and*. The bic instruction forms the logical *and* of one of its operands with the complement of the other operand. Thus the C statement

```
        i=j&k;
```

may be translated, assuming that i is located at -4(fp), j at -8(fp), and k at -12(fp), into

```
        mcoml   -12(fp),-12(fp)
        bicl3   -12(fp),-8(fp),-4(fp)
```

or

```
        mcoml   -8(fp),-8(fp)
        bicl3   -8(fp),-12(fp),-4(fp)
```

or worse, a temporary register could be involved

```
        mcoml   -12(fp),r0
        bicl3   r0,-8(fp),-4(fp)
```

or

```
        mcoml   -8(fp),r0
        bicl3   r0,-12(fp),-4(fp)
```

There are circumstances where each of these four choices would be best. However, you would not be surprised if a compiler occasionally chose wrongly.

The Monol compiler makes no attempt to choose between the four. Monol does not contain an & operator. It contains the operator &~ which translates directly into the bic instruction.

Every statement must be translatable into one (or at most, a few) well specified assembler instructions. This philosophy is reflected in the form of the language definition in section 2. In almost every section, the description of a construct is followed by the assembler code generated by it.

Monol is designed to give the programmer control over all 16 general registers. The identifiers r0, r1 ,..., r11, fp, sp, ap, and pc are predeclared, and other names can be given to specified registers. Registers can be typed double (not allowed in C). There are no hidden register references.

The requirement that all register references be explicit precluded the use of abstract data types. The first Monol compiler included arrays, records and pointers as possible types. However, it was found to be impossible to de-reference something like

```
i[j][k] = 1;
```

without using a temporary register to calculated the actual index from j and k. Once a temporary register is allowed in address calculations, there is no justification for not evaluating complex expressions, and one finds one's self trying to write the perfect optimizing C compiler.

Another goal of this language is to allow an experienced assembly language programmer to do anything he/she could do in raw assembly code. Easy access to all VAX addressing modes is provided. And, as a catch-all for any facility otherwise unavailable, the Monol compiler accepts in-line assember code.

In summary, our philosophy is to build a totally non-optimizing compiler. It allows abstract control structures, but these are compiled by simple rules into well specified assembler instructions.

2. Language Definition

What follows is a formal definition of the Monol programming language as it now stands (January, 1984). Monol is defined very much in terms of the assembler instructions produced. The reader unfamiliar with the VAX instruction set is encouraged to consult [Vax81], especially chapters 4, 5, 11 and 13. Occasionally prose in the Lexical Tokens section has been copied from the C Reference Manual [Kerninghan78].

The meta-language used is based on Backus-Naur Form. The following table lists the meanings of the various meta-symbols used.

meta-symbol	meaning
::=	shall be defined to be
|	alternatively
[x]	0 or 1 instance of x
{x}	0 or more instances of x
()	used for grouping
"x"	the terminal symbol x
lower-case-name	a non-terminal symbol

Table 9. Meta-symbols used.

The following text contains many examples of Monol declarations/executable statements and the code which is generated. In each case, the Monol is shown on the left, and the code generated is on right.

2.1. Lexical Tokens

There are five classes of tokens: identifiers, keywords, constants, operators, and explicit opcodes. Blanks, tabs, newlines, and comments (collectively, "white space") are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

2.1.1. Comments

The characters "/*" introduce a comment, which terminates with the characters "*/". Comments do not nest.

2.1.2. Keywords

The following tokens are reserved for use as keywords. All keywords are in lower case.

block	break	byte	case
default	double	downto	else
extern	float	for	goto
if	int	longword	register
repeat	restore	return	save
static	switch	to	until
while	word		

2.1.3. Constants

Appropriate constants may be declared for every type in Monol.

2.1.3.1. Integers

digit	::=	"0" |"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
digit-sequence	::=	*digit* {*digit*} .
digit-not-zero	::=	"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

```
hexidecimal-digit     ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
                              "A" | "B" | "C" | "D" | "E" | "F" |
                              "a" | "b" | "c" | "d" | "e" | "f" .
decimal-integer       ::=  digit-not-zero {digit}.
octal-integer         ::=  "0" {digit}.
hexidecimal-integer   ::=  "0x" {hexidecimal-digit} |
                           "0X" {hexidecimal-digit}.
```

A decimal-integer is read as base 10, an octal-integer is read as base 8, and a hexidecimal-integer is read as (you guessed it) base 16. The type of an integer may be byte, word, or longword.

2.1.3.2. Byte Constants

```
byte-constant   ::=  decimal-integer ("B" | "b") |
                     octal-integer ("B" | "b") |
                     hexidecimal-integer ("B" | "b") .
```

It is an error if the value of the integer exceeds 255.

2.1.3.3. Word Constants

```
word-constant   ::=  decimal-integer ("W" | "w") |
                     octal-integer ("W" | "w") |
                     hexidecimal-integer ("W" | "w") .
```

It is an error if the value of the integer exceeds 65535.

2.1.3.4. Longword Constants

```
longword-constant   ::=  decimal-integer |
                         octal-integer |
                         hexidecimal-integer |
                         longword-constant "+" longword-constant |
                         longword-constant "-" longword-constant |
                         longword-constant "*" longword-constant |
                         longword-constant "/" longword-constant |
                         longword-constant "&" longword-constant |
                         longword-constant "|" longword-constant |
                         longword-constant "^" longword-constant |
                         longword-constant "<<" longword-constant |
                         longword-constant ">>" longword-constant |
                         "(" longword-constant ")" .
```

The meaning and precedence of the operators is as in C. An expression, if present, is evaluated at compile time and replaced by the result. Attempts to divide by zero are flagged by the compiler, but there are no checks for overflow during constant folding.

2.1.3.5. Floating Constants

floating-exponent	::=	("e"|"E") *["+"|"-"] digit-sequence* .
floating-constant	::=	*digit-sequence* "." |
		digit-sequence "." *digit-sequence* |
		digit-sequence floating-exponent |
		digit-sequence "." *digit-sequence floating-exponent* .

Notice that all floating constants begin with a digit.

2.1.3.6. Double Constants

A double constant is defined by:

double-exponent	::=	("d"|"D") *["+"|"-"] digit-sequence* .
double-constant	::=	*digit-sequence double-exponent* |
		digit-sequence "." *digit-sequence double-exponent* .

The only syntactic difference between floating and double constants is that "D" replaces "E".

2.1.3.7. Character Constants

A character constant is a single character enclosed in single quotes. Certain non-graphic characters, both quotes and the backslash may be represented as well, according to the following table of escape sequences

newline	\n
tab	\t
backspace	\b
carriage return	\r
form feed	\f
backslash	\\
single quote	\'
double quote	\"
bit pattern	\ddd

The escape \ddd consists of the backslash followed by 1, 2 or 3 octal digits which are taken to specify the value of the desired character. The type of all character constants is byte.

2.1.3.8. String Constants

A string is a sequence of characters surrounded by double quotes. The escape sequences described above also count as characters. The compiler adds a null byte \0 to the end of each string. The type of a string constant is int, where the value of the integer is the storage address of the string in the bss segment.

2.1.4. Identifiers

letter	::=	"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
		"J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
		"S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
		"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
		"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
		"s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_" .
identifier	::=	*letter* {*letter* | *digit*}.

Identifiers denote registers, memory locations, labels, and global names for the link editor. All of the characters of an identifier are significant, and upper and lower case letters are distinct.

2.1.5. Explicite Opcodes

The following tokens are reserved for use as explicit opcodes.

chmk	chme	chms	chmu	prober	probew	rei
svpctx	mfpr	mtpr	xfc	bpt	bug	halt
movw	movl	movq	movo	movf	movd	movg
pushl	clrb	clrw	clrl	clrq	clro	clrf
clrg	clrh	mnegb	mnegw	mnegl	mnegf	mnegd
mnegh	mcomb	mcomw	mcoml	mcomf	mcomd	mcomg
cvtlb	cvtlw	cvtbf	cvtbd	cvtbg	cvtbh	cvtwf
cvtwg	cvtwh	cvtlf	cvtld	cvtlg	cvtlh	cvtfb
cvtgb	cvthb	cvtfw	cvtdw	cvtgw	cvthw	cvtfl
cvtdl	cvtrdl	cvtgl	cvtrgl	cvthl	cvtrhl	cvtfd
cvtfh	cvtdf	cvtdh	cvtgf	cvtgh	cvthf	cvthd
movzbw	movzbl	movzwl	cmpb	cmpw	cmpl	cmpf
cmpg	cmph	incb	incw	incl	tstb	tstw
tstf	tstd	tstg	tsth	addb2	addw2	addl2
addd2	addg2	addh2	addb3	addw3	addl3	addf3
addg3	addh3	adwc	adawl	subb2	subw2	subl2
subd2	subg2	subh2	subb3	subw3	subl3	subf3
subg3	subh3	decb	decw	decl	sbwc	mulb2
mull2	mulf2	muld2	mulg2	mulh2	mulb3	mulw3
mulf3	muld3	mulg3	mulh3	emul	emodf	emodd
emodh	divb2	divw2	divl2	divf2	divd2	divg2
divb3	divw3	divl3	divf3	divd3	divg3	divh3
bitb	bitw	bitl	bisb2	bisw2	bisl2	bisb3
bisl3	bicb2	bicw2	bicl2	bicb3	bicw3	bicl3
xorw2	xorl2	xorb3	xorw3	xorl3	ashl	ashq
poly	pushr	popr	movpsl	bispsw	bicpsw	movab
moval	movaq	movao	movaf	movad	movag	movah
pushaw	pushal	pushaq	pushao	pushaf	pushad	pushag
index	insque	remque	insqhi	insqti	remqhi	remqti
ffs	extv	extzv	cmpv	cmpzv	insv	bneq
beql	beqlu	bgtr	bleq	bgeq	blss	bgtru
bvs	bvc	bgequ	blssu	bcc	bcs	brb
jmp	bbs	bbc	bbss	bbcs	bbsc	bbcc
bbcci	blbs	blbc	acbb	acbw	acbl	acbf
acbg	acbh	aoblss	aobleq	sobgeq	sobgtr	caseb
casel	bsbb	bsbw	jsb	rsb	callg	calls
movc3	movc5	movtc	movtuc	cmpc3	cmpc5	scanc
locc	skpc	matchc	crc	movp	cmpp3	cmpp4
addp6	subp4	subp6	mulp4	mulp6	divp4	divp6
cvtpl	cvtpt	cvttp	cvtps	cvtsp	ashp	.ABORT
.file	.align	.data	.text	.org	.space	.byte
.long	.quad	.float	.double	.ascii	.asciz	.comm
.globl	.set	.lsym	.stab	.stabs	.stabn	.stabd
jnequ	jeql	jeqlu	jgtr	jleq	jgeq	jlss
jlequ	jgequ	jlssu	jbss	jbcs	jbsc	jbcc
jlbc	jcc	jcs	jvs	jvc	jbs	jbc

2.2. Program Structure

program	::=	*program global-declarations* |
		program function-definition |
		program explicit-assembler |
		empty .
function-definition	::=	*[class] [type] identifier "(" formal-parameter-list ")"*
		parameter-declarations block .
block	::=	*"{" local-declarations statement-sequence "}"*
statement-sequence	::=	{ *statement* } .

If the *type* is missing in a *function-definition*, the type of the function is longword. The function identifier, with "_" prefixed, is made the label of a .word 0 pseudo-op. If the *class* is "extern" or missing, the function identifier is also made known to the link editor by a .globl instruction.

2.3. Declarations

Associated with every identifier are two attributes, a class and a type. These two attributes determine what is bound to the identifier: a number of memory locations, a register, a function invocation, or a program location.

2.3.1. Classes

There are seven classes of identifiers.

class	binding
register	a specified register
local	memory locations from the frame pointer (fp)
parameter	memory locations from the argument pointer (ap)
static	memory locations from the bss segment allocated by the .lcomm pseudo-op.
extern	memory locations from the bss segment allocated by the .comm pseudo-op.
function	a function invocation
label	a program location

Table 10. Bindings of the various classes of identifiers.

2.3.2. Types

The number of bytes of memory associated with local, parameter, static and external identifiers depends on the type of the identifier. Variable types are also significant in the translation of executable statements into assembler.

type	::=	"byte" |
		"word" |
		"longword" |
		"int" |
		"float" |
		"double" |
		"block" "(" *longword-constant* ")" .

Longword and int are synonyms.

type	number of bytes
byte	1
word	2
longword	4
float	4
double	8
block(n)	n

Table 11. Bytes of memory associated with *type*

2.3.3. Global Declarations

```
global-declaration   ::=   register-declaration |
                           static-declaration |
                           initialized-static-declaration |
                           extern-declaration |
                           initialized-extern-declaration |
                           default-declaration |
                           initialized-default-declaration |
                           function-declaration .
```

2.3.3.1. Register Declarations

A register is given a name and a type by:

```
register-declaration   ::=   "register" [type] identifier "(" register-number ")"
                             { "," identifier "(" register-number ")" } ";" .
register number        ::=   "r0" | "r1" | "r2" | "r3" | "r4" | "r5" | "r6" |
                             "r7" | "r8" | "r9" | "r10" | "r11" | "ap" | "fp" | "sp" | "pc" .
```

If the type is missing, it is assumed to be type longword. There is no reason not to declare several identifiers to be the same register, but of different types. If, for example, you want to use r0 as an integer loop counter in one part of the program, and later on use it to hold a floating value, you might do something like this:

```
register i(r0);
register float f(r0);
```

2.3.3.2. Static Declarations

An identifier may be declared as static by

```
static-declaration   ::=   "static" [type] identifier {"," identifier } ";" .
```

If the type is missing, it is assumed to be type longword. A unique identifier is generated (as in C), and all references to the declared identifier are translated into this new identifier. The number of bytes specified by the type is allocated from the bss segment and the new identifier assigned the location of the first byte by the .lcomm pseudo-op.

```
static word w;         .lcomm L10,2
static block(22) b;    .lcomm L11,22
```

```
w=0W;                  | clrw L10
```

2.3.3.3. Initialized Static Declarations

An identifier may be associated with initialized data in the bss segment by:

initialized-static-declaration	::=	"static" *[type] identifier* "=" *initialization* {"," *identifier* "=" *initialization*} ";" .
initialization	::=	*constant* | "{" *constant* { "," *constant* } "}".
constant	::=	*byte-constant* | *word-constant* | *longword-constant* | *floating-constant* | *double-constant* | *character-constant* | *string-constant* .

If the type is missing, it is assumed to be type longword. A unique identifier is generated, and all references to the declared identifier are translated into this new identifier. The new identifier is made into a label of the first of the .byte, .word, .long, .float, .double and/or .ascii pseudo-op's which enter the constants into the bss segment.

```
static byte i = ' 9';                    | L10:  .byte 9
static int j = {0B,0W,0,0E0,0D0};        | L11:  .byte  0
                                         |       .word   0
                                         |       .long   0
                                         |       .float  0f0.0000000e0
                                         |       .double  0d0.00000000000000000e0
                                         |
i=0B                                     |       clrb L10
```

2.3.3.4. Extern Declarations

extern-declaration ::= "extern" *[type] identifier* {"," *identifier* } ";" .

If the type is missing, it is assumed to be type longword. An entry point identifier is generated (by prefixing with "_"), and all references to the declared identifier are translated into this new identifier. The number of bytes specified by the type are allocated from the bss segment, and the new identifier is assigned the location of the first byte by the .comm pseudo-op.

```
static word w;          | .comm _w,2
static block(22) b;     | .comm _b,22
                        |
w=0B;                   | clrw _w
```

2.3.3.5. Initialized Extern Declarations

initialized-extern-declaration ::= "extern" *[type] identifier* "=" *initialization* {"," *identifier* "=" *initialization*} ";" .

If the type is missing, it is assumed to be type longword. An entry point identifier is generated (by prefixing with "_"), and all references to the declared identifier are translated into this new identifier. The new identifier is made known to the link editor by a .globl pseudo-op. The new identifier is also made into a label of the first of the .byte, .word, .long, .float, .double and/or .ascii pseudo-op's which enter the constants into the bss segment.

```
static byte i = ' 9';                |       .globl _i
                                     |  _i:  .byte 9
                                     |
static int j = {0B,0W,0,0E0,0D0};    |       .globl _j
                                     |  _j:  .byte  0
                                     |       .word   0
                                     |       .long   0
                                     |       .float  0f0.0000000e0
                                     |       .double  0d0.0000000000000000e0
                                     |
i=0                                  |       clrb _i
```

2.3.3.6. Default External Declarations

default-declaration ::= *type identifier* {"," *identifier*} .

The default class in a global declaration is extern. Precisely the same code is generated for a *default-declaration* on the global level as would be generated for an *extern-declaration* of the same identifiers.

2.3.3.7. Initialized Default Declarations

initialized-extern-declaration ::= *type identifier* "=" *initialization*
{"," *identifier* "=" *initialization*} ";" .

Precisely the same code is generated for a *initialized-default-declaration* as would be generated for an *initialized-extern-declaration* of the same identifiers.

2.3.3.8. Function Declarations

function-declaration	::=	*class [type] function-identifier* {"," *function-identifier*} ";" |
		type function-identifier {"," *function-identifier*} ";" .
class	::=	"extern" |
		"static" .
function-identifier	::=	*identifier* "(" ")" .

If the type is missing, it is assumed to be type longword. The *class* is included only as a convenience to C programmers (who are used to writing things like "extern double sqrt();"), and is ignored. The class of the identifier is function. No code is generated. The usefulness of this declaration is to associate a type with a function name for use in the translation of executable statements into assembler.

2.3.4. Parameter Declarations

parameter-declaration ::= *type identifier* {"," *identifier*} ";" .

There is no *class* in a *parameter-declaration.* All identifiers so declared are class parameter, and are bound to memory locations which are allocated from the argument pointer (ap) stack.

The order in which space is allocated from the ap stack is determined by the order of the identifiers in the *formal-parameter-list.*

```
formal-parameter-list   ::=   identifier {"," identifier} |
                              empty .
```

If an identifier appears in the *formal-parameter-list* but is not declared in the *parameter-declaration* of a *function-definition,* then the type of the identifier is assumed to be longword. It is an error for an identifier to appear in the *parameter-declaration* but not in the *formal-parameter-list.*

Byte, word, longword, and float parameters are allocated 4 bytes. Double parameters get 8 bytes. Block(n) parameters get n bytes. All parameters are allocated so as to start on a longword boundary.

```
blarg(b,w,i,f,d,bk)  | b: 4(ap)
word w;              | w: 8(ap)
double d;            | i: 12(ap)
float f;             | f: 16(ap)
block(10) bk;        | d: 24(ap)
byte b;              | bk: 32(ap)
{                    |
}                    |
```

2.3.5. Local Declarations

```
local-declaration   ::=   register-declaration |
                          static-declaration |
                          initialized-static-declaration |
                          extern-declaration |
                          function-declaration |
                          default-declaration .
```

A *register-declaration, static-declaration, initialized-static-declaration,* or a *function-declaration* occuring on the local level has exactly the same syntax and results in exactly the same code as discussed above. Only static declarations may be initialized at the local level.

2.3.5.1. Extern Declarations

An *extern-declaration* may occur at the local level as well. An entry point identifier is generated (by prefixing with "_"), and all references to the declared identifier are translated into this new identifier. However, space is **not** allocated from the bss segment. The usefulness of this declaration is to associate a type with an external name for use in the translation of executable statements into assembler.

2.3.5.2. Default Local Declarations

The default class in a *local-declaration* is local. All identifiers so declared are bound to memory locations which are allocated from the frame pointer (fp) stack.

The number of bytes allocated is specified by the *type* in the declaration (see Table 3). Longword, float, double and block locals are allocated to start on a longword boundary. Word locals are allocated to start on a word boundary.

```
blarg()
{
        byte b;        | b: -1(fp)
        word w;        | w: -4(fp)
        longword l;    | l: -8(fp)
        float f;       | f: -12(fp)
        double d;      | d: -20(fp)
        block(10) b;   | bk: -36(fp)
}                      |
```

2.4. Pre-Declared Identifiers

The 15 identifiers r0 through r11, ap, fp, sp, and pc are pre-declared as typeless registers, and are bound to the appropriate hardware registers. A type can be associated with each by the "@" *type* construct. See Section 2.5.8 Variables below for details.

2.5. Operands

```
operand  ::=  byte-constant |
              word-constant |
              longword-constant |
              floating-constant |
              double-constant |
              character-constant |
              string-constant |
              variable .
```

Associated with each operand is a type and a mark. The mark is the actual assembler operand which corresponds to the Monol operand.

2.5.1. Byte Constants

The type of an operand which consists of a *byte-constant* is byte, and the mark is $n where n is the value of the constant. The following shows Monol on the left and the code which is generated on the right.

```
b=10B; |        movb $10,-1(fp)
```

2.5.2. Word Constants

The type of an operand which consists of a *word-constant* is word, and the mark is $n where n is the value of the constant. Again, there's Monol on the left and the code which is generated on the right.

```
w=10W; |        movw $10,-4(fp)
```

2.5.3. Longword Constants

The type of an operand which consists of a *longword-constant* is longword, and the mark is $n where n is the value of the constant.

```
l=(3+2)*2; |        movl $10,-4(fp)
```

2.5.4. Floating Constants

Floating constants are usually placed in the bss segment, and their reference converted to the unique label of data generated. The type of an such an operand is float, and the mark is the label. A special case is 0.0 in the assignment statement f=0.0.

```
f=1.0;          .data
                .align 2
```

```
          | L12:  .float 0f1.000000e0
          |       .text
          |       movf L12,-12(fp)
f=0.0;    |       clrf -12(fp)
```

2.5.5. Double Constants

Double constants are usually placed in the bss segment, and their reference converted to the unique label of data generated. The type of an such an operand is double, and the mark is the label. A special case is 0D0 in the assignment statement d=0D0.

```
d=1D0;    |       .data
          |       .align 2
          | L12:  .double 0d1.0000000000000e0
          |       .text
          |       movf L12,-12(fp)
d=0D0;    |       clrf -12(fp)
```

2.5.6. Character Constants

The type of an operand which consists of a *character-constant* is byte, and the mark is $n where n is the value of the constant.

```
b='a';  |       movb $97,-1(fp)
```

2.5.7. String Constants

String constants are placed in the bss segment, and their reference converted to the unique label of data generated. The type of an such an operand is longword, and the mark is $label.

```
i="1234";  |       .data
           |       .align 2
           | L12:  .ascii "1234 "
           |       .text
           |       movl $L12,-4(fp)
```

2.5.8. Variables

The full range of VAX assembler addressing modes is available to the Monol programmer.

```
variable          ::=  typed-identifier |
                       "(" typed-identifier ")" |
                       "(" typed-identifier ")+" |
                       "*(" typed-identifier ")+" |
                       "-(" typed-identifier ")" |
                       longword-constant "(" typed-identifier ")" |
                       "*" longword-constant "(" typed-identifier ")" |
                       "*" typed-identifier |
                       "(" typed-identifier ")" "[" identifier "]" |
                       "(" typed-identifier ")+" "[" identifier "]" |
                       "*(" typed-identifier ")+" "[" identifier "]" |
                       "-(" typed-identifier ")" "[" identifier "]" |
                       longword-constant "(" typed-identifier ")" "[" identifier "]" |
                       "*" longword-constant "(" typed-identifier ")" "[" identifier "]" |
                       "*" typed-identifier "[" identifier "]" .
typed-identifier  ::=  identifier |
                       identifier "@" type |
```

identifier "@" *longword-constant* .

The identifier between "[" and "]", if present, must be class register.

If "@" is not present, the type of a *typed-identifier* is the declared type of the contained *identifier*. If "@" is followed by a *type*, the type of a *typed-identifier* is given by the *type*. If "@" is followed by a *longword-constant*, the type of a *typed-identifier* is block(n) where n is the value of the *longword-constant*. The type of a variable is given by the type of the *typed-identifier*.

The mark of a variable depends on the syntax and the class of the identifier in the *typed-identifier*. The table below refers to the following declarations:

```
int e;
blarg(p)
{
    register r(r0),k(r5);
    static s;
    longword l;
}
```

syntax	class	addressing mode	mark
r	register	register	r0
e	extern	absolute	_e
s	static	absolute	L10
p	parameter	byte displacement	4(ap)
l	local	byte displacement	-4(fp)
(r)	register	register deferred	(r0)
(e)	extern	error	
(s)	static	error	
(p)	parameter	error	
(l)	local	error	
(r)+	register	auto increment	(r0)+
(e)+	extern	error	
(s)+	static	error	
(p)+	parameter	error	
(l)+	local	error	
*(r)+	register	auto increment deferred	*(r0)+
*(e)+	extern	error	
*(s)+	static	error	
*(p)+	parameter	error	
*(l)+	local	error	
-(r)	register	auto decrement	-(r0)
-(e)	extern	error	
-(s)	static	error	
-(p)	parameter	error	
-(l)	local	error	
2(r)	register	byte displacement	2(r0)
2(e)	extern	absolute	_e+2
2(s)	static	absolute	L10+2
2(p)	parameter	byte displacement	4+2(ap)
2(l)	local	byte displacement	-4+2(fp)
*2(r)	register	byte displacement deferred	*2(r0)
*2(e)	extern	absolute deferred	*_e+2
*2(s)	static	absolute deferred	*L10+2
*2(p)	parameter	byte displacement deferred	*4+2(ap)
*2(l)	local	byte displacement deferred	*-4+2(fp)
*r	register	error	
*e	extern	absolute deferred	*_e
*s	static	absolute deferred	*L10
*p	parameter	byte displacement deferred	*4(ap)
*l	local	byte displacement deferred	*-4(fp)

Table 12. Addressing modes.

syntax	class	addressing mode	mark
(r)[k]	register	register deferred indexed	(r0)[r5]
(e)[k]	extern	error	
(s)[k]	static	error	
(p)[k]	parameter	error	
(l)[k]	local	error	
(r)+[k]	register	auto increment indexed	(r0)+[r5]
(e)+[k]	extern	error	
(s)+[k]	static	error	
(p)+[k]	parameter	error	
(l)+[k]	local	error	
*(r)+[k]	register	auto increment deferred indexed	*(r0)+[r5]
*(e)+[k]	extern	error	
*(s)+[k]	static	error	
*(p)+[k]	parameter	error	
*(l)+[k]	local	error	
-(r)[k]	register	auto decrement indexed	-(r0)[r5]
-(e)[k]	extern	error	
-(s)[k]	static	error	
-(p)[k]	parameter	error	
-(l)[k]	local	error	
2(r)[k]	register	byte displacement indexed	2(r0)[r5]
2(e)[k]	extern	absolute indexed	_e+2[r5]
2(s)[k]	static	absolute indexed	L10+2[r5]
2(p)[k]	parameter	byte displacement indexed	4+2(ap)[r5]
2(l)[k]	local	byte displacement indexed	-4+2(fp)[r5]
*2(r)[k]	register	byte displacement deferred indexed	*2(r0)[r5]
*2(e)[k]	extern	absolute deferred indexed	*_e+2[r5]
*2(s)[k]	static	absolute deferred indexed	*L10+2[r5]
*2(p)[k]	parameter	byte displacement deferred indexed	*4+2(ap)[r5]
*2(l)[k]	local	byte displacement deferred indexed	*-4+2(fp)[r5]
*r[k]	register	error	
*e[k]	extern	absolute deferred indexed	*_e[r5]
*s[k]	static	absolute deferred indexed	*L10[r5]
*p[k]	parameter	byte displacement deferred indexed	*4(ap)[r5]
*l[k]	local	byte displacement deferred indexed	*-4(fp)[r5]

Table 12. Addressing modes (continued).

2.6. Conditions

```
primary-condition   ::=   operand
                          operand "==" operand |
                          operand "!=" operand |
                          operand "<" operand |
                          operand "<=" operand |
                          operand ">" operand |
                          operand ">=" operand .
condition           ::=   primary-condition |
                          "!" condition |
                          condition "&&" condition |
                          condition "||" condition .
```

The operators have the same meaning as in C.

The type of a *primary-condition* is the type of the single operand if there is only one, otherwise it is the type of the left hand operand. The type of the right hand operand is ignored during translation.

2.7. Statements

```
simple-statement      ::=   goto-statement |
                            break-statement |
                            assignment-statement |
                            function-call |
                            return-statement |
                            save-statement |
                            restore-statement |
                            empty ";" .
structured-statement  ::=   if-statement |
                            switch-statement |
                            while-statement |
                            repeat-statement |
                            for-statement .
statement             ::=   label statement |
                            simple-statement |
                            structured-statement |
                            explicit-assembler |
                            block .
```

2.7.1. Labeled Statements

```
label           ::=   identifier ":" |
                      "case"  case-constant ":" |
                      "default" ":" .
case-constant   ::=   byte-constant |
                      word-constant |
                      longword-constant .
```

"Case" and "default" labels may only occur in the contained statement of a *switch-statement*. They are discussed below. An identifier label is transferred unchanged to the compiler output.

```
forever:    ++i;  | forever:   incl   -4(fp)
```

2.7.2. Goto Statements

goto-statement ::= "goto" *identifier* ";".

A goto statement is translated into a single jbr instruction.

```
forever:    ++i;          | forever:   incl   -4(fp)
            goto forever; |            jbr    forever;
```

2.7.3. Break Statements

break-statement ::= "break" ";".

A break statement is also translated into a single jbr instruction. The destination of the jump is a unique label located immediately after the smallest enclosing while, repeat, for loop, or switch statement.

```
for( i=0 to 10 )        |       clrl    -4(fp)
  {                     | L1:   .data
  printf( "testing" );  |       .align  2
  break;                | L3:   .ascii  "testing  "
  }                     |       .text
                        |       pushl   $L3
                        |       calls   $1,_printf
                        |       jbr     L2
                        |       aobleq  $10,-4(fp),L1
                        | L2:
```

2.7.4. Assignment Statements

```
assignment-statement   ::=   variable "=" operand ";" |
                             variable "=" "-" operand ";" |
                             variable "=" "~" operand ";" |
                             variable "=" "&" operand ";" |
                             variable "=" operand "+" operand ";" |
                             variable "+=" operand ";" |
                             variable "=" operand "-" operand ";" |
                             variable "-=" operand ";" |
                             variable "=" operand "*" operand ";" |
                             variable "*=" operand ";" |
                             variable "=" operand "/" operand ";" |
                             variable "/=" operand ";" |
                             variable "=" operand " |" operand ";"|
                             variable " |=" operand ";"|
                             variable "=" operand "^" operand ";" |
                             variable "^=" operand ";" |
                             variable "=" operand "&~" operand ";" |
                             variable "&~=" operand ";" |
                             variable "=" operand "<<" operand ";" |
                             variable "<<=" operand ";".
```

The "&~" operator forms the bitwise and of the left hand operand with the complement of the right hand operand. The other operators have meaning as in C. To do a right shift, use the "<<" operator with a negative second operand.

Assignment statements are designed to be translated into one unique instruction, depending on the operator and the type of the right hand side. The type of the right hand side is given by the type of the left-most *operand* if there are two, otherwise it is the type of the single *operand.*

The only case where the type of the variable is considered is when the syntax is

variable "=" operand

The following table refers to the declarations below

```
blarg()
{
  block(20) i,j,k;
}
```

If the type of a variable is inconsequential, then it is not given.

syntax	code	generated
i=0B;	clrb	-20(fp)
i=0W;	clrw	-20(fp)
i=0;	clrl	-20(fp)
i=0.0;	clrf	-20(fp)
i=0D0;	clrd	-20(fp)
i@byte=j@byte	movb	-40(fp),-20(fp)
i@byte=j@word	cvtwb	-40(fp),-20(fp)
i@byte=j@longword	cvtlb	-40(fp),-20(fp)
i@byte=j@float	cvtfb	-40(fp),-20(fp)
i@byte=j@double	cvtdb	-40(fp),-20(fp)
i@word=j@byte	cvtbw	-40(fp),-20(fp)
i@word=j@word	movw	-40(fp),-20(fp)
i@word=j@longword	cvtlw	-40(fp),-20(fp)
i@word=j@float	cvtfw	-40(fp),-20(fp)
i@word=j@double	cvtdw	-40(fp),-20(fp)
i@longword=j@byte	cvtbl	-40(fp),-20(fp)
i@longword=j@word	cvtwl	-40(fp),-20(fp)
i@longword=j@longword	movl	-40(fp),-20(fp)
i@longword=j@float	cvtfl	-40(fp),-20(fp)
i@longword=j@double	cvtdl	-40(fp),-20(fp)
i@float=j@byte	cvtbf	-40(fp),-20(fp)
i@float=j@word	cvtwf	-40(fp),-20(fp)
i@float=j@longword	cvtlf	-40(fp),-20(fp)
i@float=j@float	movf	-40(fp),-20(fp)
i@float=j@double	cvtdf	-40(fp),-20(fp)
i@double=j@byte	cvtbd	-40(fp),-20(fp)
i@double=j@word	cvtwd	-40(fp),-20(fp)
i@double=j@longword	cvtld	-40(fp),-20(fp)
i@double=j@float	cvtfd	-40(fp),-20(fp)
i@double=j@double	movd	-40(fp),-20(fp)
i=j@10	movc3	$10,-40(fp),-20(fp)
i=-j@byte	mnegb	-60(fp),-40(fp),-20(fp)
i=-j@word	mnegw	-60(fp),-40(fp),-20(fp)
i=-j@longword	mnegl	-60(fp),-40(fp),-20(fp)
i=-j@float	mnegf	-60(fp),-40(fp),-20(fp)
i=-j@double	mnegd	-60(fp),-40(fp),-20(fp)
i=-j@10	error	

Table 13. Translations of assignment statements.

syntax	code	generated
i=~j@byte	mcomb	-60(fp),-40(fp),-20(fp)
i=~j@word	mcomw	-60(fp),-40(fp),-20(fp)
i=~j@longword	mcoml	-60(fp),-40(fp),-20(fp)
i=~j@float	mcomf	-60(fp),-40(fp),-20(fp)
i=~j@double	mcomd	-60(fp),-40(fp),-20(fp)
i=~j@10	error	
i=&j@byte	movab	-60(fp),-40(fp),-20(fp)
i=&j@word	movaw	-60(fp),-40(fp),-20(fp)
i=&j@longword	moval	-60(fp),-40(fp),-20(fp)
i=&j@float	movaf	-60(fp),-40(fp),-20(fp)
i=&j@double	movad	-60(fp),-40(fp),-20(fp)
i=&j@10	error	
i=j@byte+k	addb3	-60(fp),-40(fp),-20(fp)
i=j@word+k	addw3	-60(fp),-40(fp),-20(fp)
i=j@longword+k	addl3	-60(fp),-40(fp),-20(fp)
i=j@float+k	addf3	-60(fp),-40(fp),-20(fp)
i=j@double+k	addd3	-60(fp),-40(fp),-20(fp)
i=j@10+k	error	
i+=j@byte	addb2	-40(fp),-20(fp)
i+=j@word	addw2	-40(fp),-20(fp)
i+=j@longword	addl2	-40(fp),-20(fp)
i+=j@float	addf2	-40(fp),-20(fp)
i+=j@double	addd2	-40(fp),-20(fp)
i+=j@10	error	
i=j@byte-k	subb3	-60(fp),-40(fp),-20(fp)
i=j@word-k	subw3	-60(fp),-40(fp),-20(fp)
i=j@longword-k	subl3	-60(fp),-40(fp),-20(fp)
i=j@float-k	subf3	-60(fp),-40(fp),-20(fp)
i=j@double-k	subd3	-60(fp),-40(fp),-20(fp)
i=j@10-k	error	
i-=j@byte	subb2	-40(fp),-20(fp)
i-=j@word	subw2	-40(fp),-20(fp)
i-=j@longword	subl2	-40(fp),-20(fp)
i-=j@float	subf2	-40(fp),-20(fp)
i-=j@double	subd2	-40(fp),-20(fp)
i-=j@10	error	
i=j@byte*k	mulb3	-60(fp),-40(fp),-20(fp)
i=j@word*k	mulw3	-60(fp),-40(fp),-20(fp)
i=j@longword*k	mull3	-60(fp),-40(fp),-20(fp)
i=j@float*k	mulf3	-60(fp),-40(fp),-20(fp)
i=j@double*k	muld3	-60(fp),-40(fp),-20(fp)
i=j@10*k	error	

Table 13. Translations of assignment statements (continued).

syntax	code	generated
i*=j@byte	mulb2	-40(fp),-20(fp)
i*=j@word	mulw2	-40(fp),-20(fp)
i*=j@longword	mull2	-40(fp),-20(fp)
i*=j@float	mulf2	-40(fp),-20(fp)
i*=j@double	muld2	-40(fp),-20(fp)
i*=j@10	error	
i=j@byte/k	divb3	-60(fp),-40(fp),-20(fp)
i=j@word/k	divw3	-60(fp),-40(fp),-20(fp)
i=j@longword/k	divl3	-60(fp),-40(fp),-20(fp)
i=j@float/k	divf3	-60(fp),-40(fp),-20(fp)
i=j@double/k	divd3	-60(fp),-40(fp),-20(fp)
i=j@10/k	error	
i/=j@byte	divb2	-40(fp),-20(fp)
i/=j@word	divw2	-40(fp),-20(fp)
i/=j@longword	divl2	-40(fp),-20(fp)
i/=j@float	divf2	-40(fp),-20(fp)
i/=j@double	divd2	-40(fp),-20(fp)
i/=j@10	error	
i=j@byte|k	bisb3	-60(fp),-40(fp),-20(fp)
i=j@word|k	bisw3	-60(fp),-40(fp),-20(fp)
i=j@longword|k	bisl3	-60(fp),-40(fp),-20(fp)
i=j@float|k	error	
i=j@double|k	error	
i=j@10|k	error	
i|=j@byte	bisb2	-40(fp),-20(fp)
i|=j@word	bisw2	-40(fp),-20(fp)
i|=j@longword	bisl2	-40(fp),-20(fp)
i|=j@float	error	
i|=j@double	error	
i|=j@10	error	
i=j@byte^k	xorb3	-60(fp),-40(fp),-20(fp)
i=j@word^k	xorw3	-60(fp),-40(fp),-20(fp)
i=j@longword^k	xorl3	-60(fp),-40(fp),-20(fp)
i=j@float^k	error	
i=j@double^k	error	
i=j@10^k	error	
i^=j@byte	xorb2	-40(fp),-20(fp)
i^=j@word	xorw2	-40(fp),-20(fp)
i^=j@longword	xorl2	-40(fp),-20(fp)
i^=j@float	error	
i^=j@double	error	
i^=j@10	error	

Table 13. Translations of assignment statements (continued).

syntax	code	generated
i=j@byte&~k	bicb3	-60(fp),-40(fp),-20(fp)
i=j@word&~k	bicw3	-60(fp),-40(fp),-20(fp)
i=j@longword&~k	bicl3	-60(fp),-40(fp),-20(fp)
i=j@float&~k	error	
i=j@double&~k	error	
i=j@10&~k	error	
i&~=j@byte	bicb2	-40(fp),-20(fp)
i&~=j@word	bicw2	-40(fp),-20(fp)
i&~=j@longword	bicl2	-40(fp),-20(fp)
i&~=j@float	error	
i&~=j@double	error	
i&~=j@10	error	
i=j@byte<<k	ashl	-60(fp),-40(fp),-20(fp)
i=j@word<<k	ashl	-60(fp),-40(fp),-20(fp)
i=j@longword<<k	ashl	-60(fp),-40(fp),-20(fp)
i=j@float<<k	error	
i=j@double<<k	error	
i=j@10<<k	error	
i<<=j@byte	ashl	-40(fp),-40(fp),-20(fp)
i<<=j@word	ashl	-40(fp),-40(fp),-20(fp)
i<<=j@longword	ashl	-40(fp),-40(fp),-20(fp)
i<<=j@float	error	
i<<=j@double	error	
i<<=j@10	error	

Table 13. Translations of assignment statements (continued).

2.7.5. Function Calls

function-call	::=	*identifier "(" actual-parameters ")" ";"* |
		variable "=" identifier "(" actual-parameters ")" ";".
actual-parameters	::=	*actual-parameter {"," actual-parameter}* |
		empty .
actual-parameter	::=	*operand* |
		"&" operand .

The actual parameters (if any) are placed on the stack, in reverse order, by the code given in the table below. This is followed by a calls instruction.

parameter syntax	code	generated
i@byte	cvtbl	-20(fp),(sp)-
i@word	cvtwl	-20(fp),(sp)-
i@longword	pushl	-20(fp)
i@float	cvtfd	-20(fp),(sp)-
i@double	movd	-20(fp),(sp)-
i@20	subl2	20,sp
	movc3	$20,-20(fp),(sp)
&i@byte	pushab	-20(fp)
&i@word	pushaw	-20(fp)
&i@longword	pushal	-20(fp)
&i@float	pushaf	-20(fp)
&i@double	pushad	-20(fp)
&i@20	error	

Table 14. Code to push parameters onto the stack.

```
block(32) b;      | pushaw   -34(fp)
word w;           | subl2    $32,sp
ext(0,b,&w);      | movc3    $32,-32(fp),(sp)
                  | pushl    $0
                  | calls    $10,_ext
```

If the value returned by the function is to be assigned to a variable, this is followed by precisely the code which would be generated for an *assignment-statement*, the left hand side of which was the contained variable of the *function-call*, and the right hand side of which was a variable named r0 whose type is the declared type of the function (or longword if it's not declared).

```
extern flag();  | calls   $0,_flag
byte  b;        | cvtlb   r0,-1(fp)
b=flag();       |
```

2.7.6. Return Statements

```
return-statement    ::=   "return" ";" |
                          "return" "(" return-expression ")" ";".
return-expression   ::=   operand |
                          "-" operand |
                          "~" operand |
                          "&" operand .
```

The first form of the return statement is translated directly into the ret instruction.

The second form results in precisely the code which would be generated for an *assignment-statement*, the left hand side of which was a variable named r0 whose type is the type of the function currently being defined, and the right hand side of which was the *return-expression*. This is followed by a ret instruction.

```
byte toByte( d )          .data
double d;                 .text
```

```
{                        |             .align   1
   return( d );          |             .globl   _toByte
}                        |  _toByte:   .word    0
                         |             cvtdb    4(ap),r0
                         |             ret
```

2.7.7. Save Statements

save-statement ::= "save" "(" *register-list* ")" ";".
register-list ::= *identifier* {"," *identifier*}

The purpose of the save and restore statements is to preserve register values. Each identifier in the *register-list* must be bound to a register, or registers if the identifier is type double. Save pushes these registers onto the stack with a pushr instruction.

```
save(r0,r1); | pushr    $3
```

2.7.8. Restore Statements

restore-statement ::= "restore" "(" *register-list* ")" ";".

Restore retrieves the named registers from the stack with a popr instruction.

```
restore(r0,r1); | popr    $3
```

2.7.9. If Statements

if-statement ::= "if" "(" *condition* ")" *statement* |
"if" "(" *condition* ")" *statement* "else" *statement* .

Compound if statements are potentially the most complex construction that the Monol compiler has to handle. Their translation is described in detail below.

Firstly, the scope of all "!" signs is reduced to a single *primary-condition* by the use of DeMorgan's laws. "!" is then absorbed into those *primary-conditions* which involve relational operators by switching the sense of the operator ("==" becomes "!=", "<" becomes ">=", etc). Thus, the only remaining "!"s are those which sit in front of single operands.

So, consider the following single operand conditions:

syntax	code generated		
if(i@byte)		tstb	-20(fp)
statement		jeql	false
		...statement...	
	false:		
if(i@word)		tstw	-20(fp)
statement		jeql	false
		...statement...	
	false:		
if(i@longword)		tstl	-20(fp)
statement		jeql	false
		...statement...	
	false:		
if(i@float)		tstf	-20(fp)
statement		jeql	false
		...statement...	
	false:		
if(i@double)		tstd	-20(fp)
statement		jeql	false
		...statement...	
	false:		

Table 15. Translations of if statements.

Translation of the negated condition, and if-then-else statements are equally simple. In the following table, only @byte is shown. Other types are translated similarly, with tstb replaced as appropriate.

syntax	code generated		
if(!i@byte)		tstb	-20(fp)
statement		jneq	false
		...statement...	
	false:		
if(i@byte)		tstb	-20(fp)
statement1		jeql	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		
if(!i@byte)		tstb	-20(fp)
statement1		jneq	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		

Table 15. Translations of if statements (continued).

Next, consider if statements whose condition consists of a single relation "==". As stated in Section 2.6, the type of such a *primary-condition* is the type of the left hand operand. The type of the right hand operand is ignored.

syntax	code	generated	
if(i@byte==j)		cmpb	-20(fp),-40(fp)
statement		jneq	false
		...statement...	
	false:		
if(i@word==j)		cmpw	-20(fp),-40(fp)
statement		jneq	false
		...statement...	
	false:		
if(i@longword==j)		cmpl	-20(fp),-40(fp)
statement		jneq	false
		...statement...	
	false:		
if(i@float==j)		cmpf	-20(fp),-40(fp)
statement		jneq	false
		...statement...	
	false:		
if(i@double==j)		cmpd	-20(fp),-40(fp)
statement		jneq	false
		...statement...	
	false:		

Table 15. Translations of if statements (continued).

Translation of the other relational operators, and if-then-else statements is equally simple. In the following table, only @byte is shown. Other types are translated similarly, with cmpb replaced as appropriate.

syntax	code	generated	
if(i@byte!=j)		cmpb	-20(fp),-40(fp)
statement		jeql	false
		...statement...	
	false:		
if(i@byte!=j)		cmpb	-20(fp),-40(fp)
statement1		jeql	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		
if(i@byte<j)		cmpb	-20(fp),-40(fp)
statement		jgeq	false
		...statement...	
	false:		
if(i@byte<j)		cmpb	-20(fp),-40(fp)
statement1		jgeq	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		
if(i@byte<=j)		cmpb	-20(fp),-40(fp)
statement		jgtr	false
		...statement...	
	false:		
if(i@byte<=j)		cmpb	-20(fp),-40(fp)
statement1		jgtr	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		
if(i@byte>j)		cmpb	-20(fp),-40(fp)
statement		jleq	false
		...statement...	
	false:		
if(i@byte>j)		cmpb	-20(fp),-40(fp)
statement1		jleq	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		

Table 15. Translations of if statements (continued).

syntax	code	generated	
if(i@byte>=j)		cmpb	-20(fp),-40(fp)
statement		jlss	false
		...statement...	
	false:		
if(i@byte>=j)		cmpb	-20(fp),-40(fp)
statement1		jlss	false
else		...statement1...	
statement2		jbr	skip
	false:	...statement2...	
	skip:		

Table 15. Translations of if statements (continued).

Finally, consider conditions composed of i@byte==j and i@byte==k joined by "&&" or "||".

syntax	code	generated	
if(i@byte==j&&i@byte==k)		cmpb	-20(fp),-40(fp)
statement		jneq	false
		cmpb	-20(fp),-60(fp)
		jneq	false
		...statement...	
	false:		
if(i@byte==j&&i@byte==k)		cmpb	-20(fp),-40(fp)
statement1		jneq	false
else		cmpb	-20(fp),-60(fp)
statement2		jneq	false
		...statement1...	
		jbr	skip
	false:	...statement2...	
	skip:		
if(i@byte==j||i@byte==k)		cmpb	-20(fp),-40(fp)
statement		jeql	true
		cmpb	-20(fp),-60(fp)
		jneq	false
	true:	...statement...	
	false:		
if(i@byte==j||i@byte==k)		cmpb	-20(fp),-40(fp)
statement1		jeql	true
else		cmpb	-20(fp),-60(fp)
statement2		jneq	false
	true:	...statement1...	
		jbr	skip
	false:	...statement2...	
	skip:		

Table 15. Translations of if statements (continued).

Conjunctions and disjunctions of other *primary-conditions* are translated similarly. Conjunctions and disjunctions of conjunctions and/or disjunctions are broken down into conjunctions and/or disjunctions of *primary-conditions* by a recursive procedure.

One final note: the compiler does change the reference to label L2 into a reference to L3 if it finds itself generating something like

```
L2: jbr L3
```

2.7.10. Switch Statements

switch-statement ::= "switch" "(" *variable* ")" *statement* .

The *variable* must be type longword. A casel opcode is generated, followed by a jump table. The number of entries in this table is determined by the difference between the maximum and minimum *case-constant* in the contained statement. If this difference is 100,000 the compiler will try to generate a jump table with 100,000 entries. Notice that the cases "fall through" as in C. To exit the switch after a case has been processed, use break.

A default label is translated into a single jbr opcode which follows the jump table.

```
switch( i ) {          |        casel   -4(fp),$1,$5
  case 1: i=1;         |  L5:   .word   L2-L5
  case 2: i=2;         |        .word   L3-L5
  case 6: i=6;         |        .word   L1-L5
  default: i=10;       |
  }                    |        .word   L1-l5
                       |        .word   L1-L5
                       |        .word   L4-L5
                       |        jbr     L1
                       |  L2:   movl    $1,-4(fp)
                       |  L3:   movl    $2,-4(fp)
                       |  L4:   movl    $6,-4(fp)
                       |  L1:   movl    $10,-4(fp)
```

2.7.11. While Statements

while-statement ::= "while" "(" *condition* ")" *statement* .

A while statement is translated into precisely the code which would be generated by

```
loop:   if( condition )
          {
          statement;
          goto loop;
          }
```

```
while( i<10 )          L3:   cmpl    -4(fp),$10
  {                          jgeq    L2
printf( "%d ",i );  |          pushl   -4(fp)
++i;                |          .data
}                   |          .align  2
                    |  L4:     .ascii  "%d "
                    |          .text
                    |          pushl   $L4
                    |          calls   $2,_printf
                    |          incl    -4(fp)
                    |          jbr     L3
                    |  L2:
```

2.7.12. Repeat Statements

repeat-statement ::= "repeat" *statement-sequence* "until" "(" *condition* ")" .

A repeat statement is translated into precisely the code which would be generated by

```
loop:   statement-sequence
        if( !condition )
           goto loop;
```

```
repeat                |  L1:   pushl   -4(fp)
   printf( "%d ",i ); |        .data
   ++i;               |        .align  2
until( i==10 )        |  L3:   .ascii  "%d "
                      |        .text
                      |        pushl   $L3
                      |        calls   $2,_printf
                      |        incl    -4(fp)
                      |        cmpl    -4(fp),$10
                      |        jneq    L1
```

2.7.13. For Statements

for-statement ::= "for" "(" *variable "="* *operand1* "to" *operand2* ")" *statement* |
"for" "(" *variable "="* *operand1* "downto" "0" ")" *statement* .

The *variable* must be type longword. Assuming that operand2 is in the proper form, the first *for-statement* ("to") is translated into the same code as would be generated by

```
        variable = operand1
loop:   statement;
        aobleq  operand2,variable,loop;
```

The second form ("downto 0") is translated into precisely the code which would be generated by

```
         variable = operand1
loop:    statement;
         sobgeq   variable,loop;
```

No check is made of the distance between the label and the aobleq/sobgeq opcode. This must be less than -128 bytes. The assembler will occasionaly complain with the message

```
aobleq: Branch too far (-499b)
```

```
for( i=0 to 10 )     | clrl   -4(fp)
   printf( "%d ",i ); | L1:    pushl    -4(fp)
                     |        .data
                     |        .align   2
                     | L3:    .ascii   "%d  "
                     |        .text
                     |        pushl    $L3
                     |        calls    $2,_printf
                     |        aobleq   $10,-4(fp),L1
```

2.8. Explicit Assembler Code

As a catch-all for everything which is otherwise inaccessible, Monol allows in line assembler code in two places, at the global level

program	::=	*program global-declarations* |
		program function-definition |
		program explicit-assembler |
		empty .

and as a executable statement

statement	::=	*label statement* |
		simple-statement |
		structured–statement |
		explicit-assembler |
		block .

The instructions are **not** divided into those allowed on the global and those allowed on the local level. Any opcode can appear anywhere. Pure assembler code is a legal Monol program.

explicit-assember	::=	*opcode explicit-operand-list* .
explicit-operand-list	::=	*explicit-operand* {"," *explicit-operand*} |
		empty .
explicit-operand	::=	*byte-constant* |
		"$" byte-constant |
		word-constant |
		"$" word-constant |
		longword-constant |
		"$" longword-constant |

floating-constant |
double-constant |
character-constant |
string-constant |
variable .

An *opcode* is any one of the explicit opcodes listing in Section 2.2.5. The complete set of VAX instructions is available, except for the EDIT instructions (Chapter 16 in Vax81).

Floating, double and string constants are *not* automatically placed in the bss segment. Byte, word and longword constants do *not* automatically have a "$" prefixed.

However, variables *are* translated exactly as they are in a regular *operand* (Section 2.6.8). Notice however that -4(fp) and 8(ap) are both legal variables, and that if fp and ap have their default definitions, these variables are translated into -4(fp) and 8(ap) respectively.

```
fill( addr,ch,leng )                 |
byte ch;                             |
{                                    |        .data
  /*                                 |        .text
   * This routine fills the leng     |        .align   1
   * bytes starting at addr with ch  |        .globl   _fill
   */                                | _fill: .word    0
  movc5   $0,(r0),ch,leng,*addr;     |        movc5    $0,(r0),8(ap),12(ap),*4(ap)
}                                    |        ret
```

3. Using Monol

The section describes the Monol preprocessor. It offers the details of how to run a Monol program under UNIX, including the various compiler options available, and ends with some informal loop timings.

The descriptions of the compiler options are, for the most part, taken from the UNIX Programmer's Manual [Unix83], and many of the descriptions of the preprocessor command lines are condensed from [Kernighan78]. The #macro control line described in section 3.2.5 is an exclusive feature of Monol.

3.1. Running the Compiler

To invoke the Monol compiler from a UNIX shell, type

mc *options files*

where *options* is any of the various arguments listed below, and *files* contains the program(s) to be compiled. Files whose names end with .m are taken to be Monol source code, files ending in .s are taken to be VAX assembler code, files ending in .o are taken to be object programs produced by an earlier mc, cc, pc or f77 run. These programs are loaded to produce an executable program called a.out.

The following options may appear in any order. Any additional options are passed on to **ld**. For more information on -r, -e, and -i see the section on recursive macros below.

-c

Suppress the loading phase of compilation, converting all .m and .s files into object programs in files whose names are that of the source with .m or .s replaced by .o.

-g

Have the compiler produce additional information for the symbolic debugger **sdb**.

-o *filename*

Name the final output file *filename* instead of a.out.

-p

Have the compiler produce additional information for the an execution profile by **prof**.

-D*name*=*def*

-D*name*

Define *name* to the preprocessor. If *def* is missing, the name is defined to be "1".

-E

Run only the macro preprocessor on the .m files, and send the results to the standard output.

-O

Invoke the C object code improver /lib/c2. It is doubtful if this will in fact improve a carefully coded Monol program, and is in fact against the philosophy of Monol.

-S

Compile the .m files, leaving the assembler-language output on the corresponding .s files.

-r*number*

Make the limit of recursive macro substitution *number*.

-e

Substitute any macro called beyond the recursion limit by "{exit(1);}".

-i

Insert a run-time warning for each macro called beyond the recursion limit.

3.2. The Preprocessor

The Monol preprocessor is a modification of the standard C preprocessor cpp. It accepts multi-line macro definitions and recursive macro definitions, as well as that which cpp accepts.

3.2.1. File Inclusion

Any line which begins with

#include "*filename*"

is replaced by the contents of the file *filename*. #include's may be nested.

3.2.2. Conditional Compilation

A line of the form

#if *constant-expression*

checks whether the constant expression evaluates to zero. A line of the form

#ifdef *identifier*

checks whether the identifier is defined, that is, has been the subject of a #define control line, or a -D compiler option. A line of the form

#ifndef *identifier*

checks whether the identifier is undefined.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by the line

#endif

If the checked condition is false, then all lines between the test and the #endif or the #else are ignored. If the condition is true and an #else is present, then all lines between the #else and the #endif are ignored.

These constructs may be nested.

3.2.3. Line Control

A line of the form

#line *n filename*

causes the compiler to believe, for the purposes of compile time and run time error diagnostics, that it is working on the *n*'th line of the file *filename*.

3.2.4. Defines

A line of the form

#define *identifier rest-of-line*

causes subsequent occurrences of the identifier to be replaced by the rest of the line. A #define may also have arguments.

> #define *identifier*"(" *identifier* {"," *identifier*} ")" *rest-of-line*

Here, subsequent occurrences of the identifier are replace by the rest of the line, where each occurrence of an argument identifier in *rest-of-line* is in turn replaced by the corresponding actual parameter.

3.2.5. Macros

Lines of the form

> #macro *identifier* "{" *one-or-several-lines* "}"

cause subsequent occurrences of the identifier to be replaced by whatever is between the curly brackets. This may be several lines, or nothing at all. The brackets themselves are included in the replacement text. #macro definitions may have arguments as well.

The real advantage of a #macro definition over a multiple line #define (with each line terminated by a backslash), is that the line numbers of compile and run time errors are correctly reported.

3.2.6. Recursive Macros

The preprocessor also handles directly and indirectly recursive macros. You can specify the depth to which you want the macros to be recursively substituted with the compiler option -r. The default is -r20. How a macro which is a depth greater than the limit is handled depends on two other options. If -i is specified then the macro is expanded into a run time warning. If -e is given, the macro is replaced by "{ exit(1); }". -l -e gives you the warning followed by exit. If neither are specified, the macro is expanded into "{ }".

3.2.7. Bugs

You have to be careful about matching left and right brackets in a #macro definition. An extra left bracket can swallow the whole program without a trace. Recursive #macro's or #define's named "write", "macro", "called", "beyond", "recursion", "limit" or "exit" should be avoided. The preprocessor will try to do a macro substitution on the warning message if -l is given, or on exit if -e is given, leading to infinite recursion.

3.2.8. Undefine

A line of the form

> #undef *identifier*

causes any previous definition of the identifier to be forgotten.

3.3. Efficient Loops

Below are some informal timings made on various loop constructions. As always, Monol is on the left, and the resulting assembler is on the right. It is obvious from these results that one should use the for ... downto 0 construct to control the iteration if possible, and avoid the acb instruction.

```
for( r0@int=1000000 downto 0 );  |       movl     $1000000,r0
                                 | L0:   sobgeq   r0,L0
```

Figure 63. A loop which takes 1.050 seconds to execute.

```
r0@int=0;                        |       clrl     r0
repeat                           | L0:   incl     r0
  ++r0@int;                      |       cmpl     r0,$1000000
until( r0@int>=1000000 )         |       blss     L0
```

Figure 64. A loop which takes 2.467 seconds to execute.

```
r0@int=0;                        |       clrl     r0
while( r0@int<1000000 )          | L0:   cmpl     r0,$1000000
  ++r0@int;                      |       bgeq     L1
                                 |       incl     r0
                                 |       jbr      L0
                                 | L1:
```

Figure 65. A loop which takes 2.933 seconds to execute.

```
same as assembler  |       clrl    r0
                   | L0:   acbl    $1000000,$1,r0,L0
```

Figure 66. A loop which takes 4.133 seconds to execute.

```
for( r0@int=0 to 1000000 );  |       clrl     r0
                             | L0:   aobleq   $1000000,r0,L0
```

Figure 67. A loop which takes 2.267 seconds to execute.

```
r1@int=0;                        |       clrl     r1
for( r0@int=1000000 downto 0 )   |       movl     $1000000,r0
  ++r1@int;                      | L0:   incl     r1
                                 |       sobgeq   r0,L0
```

Figure 68. A loop which takes 2.050 seconds to execute.

```
d=0D0;              |        clrd     -8(fp)
repeat              | L0:    .data
  d+=1D0;           |        .align   2
until( d>=1D6 )     | L21:   .double  0d1.00000e+00
                    |        .text
                    |        addd2    L21,-8(fp)
                    |        .data
                    |        .align   2
                    | L20:   .double  0d1.00000e+06
                    |        .text
                    |        cmpd     -8(fp),L20
                    |        blss     L0
```

Figure 69. A loop which takes 9.567 seconds to execute.

```
same as assembler   |        clrd     -8(fp)
                    | L0:    .data
                    |        .align   2
                    | L21:   .double  0d1.00000e+00
                    |        .text
                    |        .data
                    |        .align   2
                    | L20:   .double  0d1.00000e+06
                    |        .text
                    |        acbd     L20,L21,-8(fp),L0
```

Figure 70. A loop which takes 19.284 seconds to execute.

```
d=0D0;                              |        clrd     -8(fp)
for( r0@int=1000000 downto 0 )      |        movl     $1000000,r0
  d+=1D0;                           | L0:    .data
                                    |        .align   2
                                    | L21:   .double  0d1.00000e+00
                                    |        .text
                                    |        addd2    L21,-8(fp)
                                    |        sobgeq   r0,L0
```

Figure 71. A loop which takes 7.033 seconds to execute.

4. A Larger Example

The routine which performs ray/bounding box intersections is by far the most time critical in the ray tracing package. The code for this routine is reproduced below.

```
#define NOINTERSECT          1e10

/*
 * typedef struct {
 *       float          xmin,ymin,zmin;
 *       float          xmax,ymax,zmax;
 *       } bounding_box;
 */
#define boundingboxp             int
#define XMIN(foo)        (foo@float)
#define YMIN(foo)        4(foo@float)
#define ZMIN(foo)        8(foo@float)
#define XMAX(foo)        12(foo@float)
#define YMAX(foo)        16(foo@float)
#define ZMAX(foo)        20(foo@float)

/*
 * typedef struct {
 *       point          origin;
 *       vector         path;
 *       } ray_info;
 */
#define rayinfop             int
#define ORIGIN_X(foo)           (foo@float)
#define ORIGIN_Y(foo)           4(foo@float)
#define ORIGIN_Z(foo)           8(foo@float)
#define DIRECTION_X(foo)             12(foo@float)
#define DIRECTION_Y(foo)             16(foo@float)
#define DIRECTION_Z(foo)             20(foo@float)


double ICube(ray,box)
rayinfop      ray;
boundingboxp  box;
{
  /*
   * Return the ray parameter which corresponds to the closest intersection
   * of ray with any face of the bounding box
   */
  register      thisray(r11),thisbox(r10);
  float         tmin,t,x,y,z;

  thisray@int=ray;
  thisbox@int=box;
  tmin=NOINTERSECT;

  if( DIRECTION_Z(thisray)!=0.0 )
    {
    /*
```

```
       * Test the ray against the front face, which is the plane defined by
       * box->xmin <= x <= box->xmax;
       * box->ymin <= y <= box->ymax;
       * z= box->zmin;
       */
      r0 = ZMIN(thisbox) - ORIGIN_Z(thisray);
      t = r0@float / DIRECTION_Z(thisray);
      if( t<tmin )
        {
        r0 = t * DIRECTION_X(thisray);
        x = ORIGIN_X(thisray) + r0;
        r0 = t * DIRECTION_Y(thisray);
        y = ORIGIN_Y(thisray) + r0;
        if( XMIN(thisbox)<=x && x<=XMAX(thisbox) &&
          YMIN(thisbox)<=y && y<=YMAX(thisbox) )
          tmin=t;
        }

      /*
       * Test the ray against the back face, which is the plane defined by
       * box->xmin <= x <= box->xmax;
       * box->ymin <= y <= box->ymax;
       * z= box->zmax;
       */
      r0 = ZMAX(thisbox) - ORIGIN_Z(thisray);
      t = r0@float / DIRECTION_Z(thisray);
      if( t<tmin )
        {
        r0 = t * DIRECTION_X(thisray);
        x = ORIGIN_X(thisray) + r0;
        r0 = t * DIRECTION_Y(thisray);
        y = ORIGIN_Y(thisray) + r0;
        if( XMIN(thisbox)<=x && x<=XMAX(thisbox) &&
          YMIN(thisbox)<=y && y<=YMAX(thisbox) )
          tmin=t;
        }
      }

    if( DIRECTION_Y(thisray)!=0.0 )
      {
      /*
       * Test the ray against the top face, which is the plane defined by
       * box->xmin <= x <= box->xmax;
       * y= box->ymax;
       * box->zmin <= z <= box->zmax;
       */
      r0 = YMAX(thisbox) - ORIGIN_Y(thisray);
      t = r0@float / DIRECTION_Y(thisray);
      if( t<tmin )
        {
        r0 = t * DIRECTION_X(thisray);
        x = ORIGIN_X(thisray) + r0;
        r0 = t * DIRECTION_Z(thisray);
        z = ORIGIN_Z(thisray) + r0;
        if( XMIN(thisbox)<=x && x<=XMAX(thisbox) &&
       ZMIN(thisbox)<=z && z<=ZMAX(thisbox) )
       tmin=t;
     }

   /*
    * Test the ray against the bottom face, which is the plane defined by
    * box->xmin <= x <= box->xmax;
    * y= box->ymin;
    * box->zmin <= z <= box->zmax;
    */
   r0 = YMIN(thisbox) - ORIGIN_Y(thisray);
   t = r0@float / DIRECTION_Y(thisray);
   if( t<tmin )
     {
     r0 = t * DIRECTION_X(thisray);
     x = ORIGIN_X(thisray) + r0;
     r0 = t * DIRECTION_Z(thisray);
     z = ORIGIN_Z(thisray) + r0;
     if( XMIN(thisbox)<=x && x<=XMAX(thisbox) &&
       ZMIN(thisbox)<=z && z<=ZMAX(thisbox) )
       tmin=t;
     }
   }

 if( DIRECTION_X(thisray)!=0.0 )
   {
   /*
    * Test the ray against the left face, which is the plane defined by
    * x= box->xmin;
    * box->ymin <= y <= box->ymax;
    * box->zmin <= z <= box->zmax;
    */
   r0 = XMIN(thisbox) - ORIGIN_X(thisray);
   t = r0@float / DIRECTION_X(thisray);
   if( t<tmin )
     {
     r0 = t * DIRECTION_Y(thisray);
     y = ORIGIN_Y(thisray) + r0;
     r0 = t * DIRECTION_Z(thisray);
     z = ORIGIN_Z(thisray) + r0;
     if( YMIN(thisbox)<=y && y<=YMAX(thisbox) &&
       ZMIN(thisbox)<=z && z<=ZMAX(thisbox) )
       tmin=t;
     }
   /*
    * Test the ray against the right face, which is the plane defined by
    * x= box->xmax;
    * box->ymin <= y <= box->ymax;
    * box->zmin <= z <= box->zmax;
    */
   r0 = XMAX(thisbox) - ORIGIN_X(thisray);
   t = r0@float / DIRECTION_X(thisray);
   if( t<tmin )
     {
     r0 = t * DIRECTION_Y(thisray);
```

```
        y = ORIGIN_Y(thisray) + r0;
        r0 = t * DIRECTION_Z(thisray);
        z = ORIGIN_Z(thisray) + r0;
        if( YMIN(thisbox)<=y && y<=YMAX(thisbox) &&
          ZMIN(thisbox)<=z && z<=ZMAX(thisbox) )
          tmin=t;
        }
      }

  return( tmin );
}
```

5. Collected Syntax

```
digit                 ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

digit-sequence        ::=  digit {digit} .

digit-not-zero        ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

hexidecimal-digit     ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
                                "A" | "B" | "C" | "D" | "E" | "F" |
                                "a" | "b" | "c" | "d" | "e" | "f" .

decimal-integer       ::=  digit-not-zero {digit}.

octal-integer         ::=  "0" {digit}.

hexidecimal-integer   ::=  ("0x"|"0X") {hexidecimal-digit} .

byte-constant         ::=  decimal-integer ("B" | "b") |
                           octal-integer ("B" | "b") |
                           hexidecimal-integer ("B" | "b") .

word-constant         ::=  decimal-integer ("W" | "w") |
                           octal-integer ("W" | "w") |
                           hexidecimal-integer ("W" | "w") .

longword-constant     ::=  decimal-integer |
                           octal-integer |
                           hexidecimal-integer |
                           longword-constant "+" longword-constant |
                           longword-constant "-" longword-constant |
                           longword-constant "*" longword-constant |
                           longword-constant "/" longword-constant |
                           longword-constant "&" longword-constant |
                           longword-constant "|" longword-constant |
                           longword-constant "~" longword-constant |
                           longword-constant "<<" longword-constant |
                           longword-constant ">>" longword-constant |
                           "(" longword-constant ")" /

floating-exponent     ::=  ("e"|"E") ["+"|"-"] digit-sequence .

floating-constant     ::=  digit-sequence "." |
                           digit-sequence "." digit-sequence |
                           digit-sequence floating-exponent |
                           digit-sequence "." digit-sequence floating-exponent .

double-exponent       ::=  ("d"|"D") ["+"|"-"] digit-sequence .

double-constant       ::=  digit-sequence double-exponent |
                           digit-sequence "." digit-sequence double-exponent .
```

```
letter                    ::=  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
                                  "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
                                  "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
                                  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
                                  "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
                                  "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_" .

identifier                ::=  letter {letter | digit}.

program                   ::=  program global-declarations |
                               program function-definition |
                               program explicit-assembler |
                               empty .

function-definition       ::=  [class] [type] identifier "(" formal-parameter-list ")"
                                  {parameter-declaration} block .

formal-parameter-list     ::=  identifier {"," identifier} |
                               empty .

block                     ::=  "{" {local-declaration} statement-sequence "}"

type                      ::=  "byte" |
                               "word" |
                               "longword" |
                               "int" |
                               "float" |
                               "double" |
                               "block" "(" longword-constant ")" .

global-declaration        ::=  register-declaration |
                               static-declaration |
                               initialized-static-declaration |
                               extern-declaration |
                               initialized-extern-declaration |
                               function-declaration |
                               default-declaration |
                               initialized-default-declaration .

parameter-declaration     ::=  type identifier {"," identifier} ";" .

local-declaration         ::=  register-declaration |
                               static-declaration |
                               initialized-static-declaration |
                               extern-declaration |
                               function-declaration |
                               default-declaration .

register-declaration      ::=  "register" [type] identifier-register
                                  {"," identifier-register} ";" .

identifier-register       ::=  identifier "(" register-number ")" .

register number           ::=  "r0" | "r1" | "r2" | "r3" | "r4" | "r5" | "r6" | "r7" | "r8" |
                                  "r9" | "r10" | "r11" | "ap" | "fp" | "sp" | "pc" .
```

```
static-declaration               ::=   “static” [type] identifier {“,” identifier } “;” .

initialized-static-declaration   ::=   “static” [type] identifier “=” initialization
                                            {“,” identifier “=” initialization} “;” .

extern-declaration               ::=   “extern” [type] identifier {“,” identifier } “;” .

initialized-extern-declaration   ::=   “extern” [type] identifier “=” initialization
                                            {“,” identifier “=” initialization} “;” .

default-declaration              ::=   type identifier {“,” identifier} “;” .

initialized-default-declaration  ::=   type identifier "=" initialization
                                            {“,” identifier “=” initialization} “;” .

function-declaration             ::=   class [type] function-identifier
                                            {“,” function-identifier} “;” |
                                       type function-identifier {“,” function-identifier} “;” .

class                            ::=   “extern” |
                                       “static” .

function-identifier              ::=   identifier “(” “)” .

initialization                   ::=   constant |
                                       "{" constant { “,” constant } “}”.

constant                         ::=   byte-constant |
                                       word-constant |
                                       longword-constant |
                                       floating-constant |
                                       double-constant |
                                       character-constant |
                                       string-constant .

variable                         ::=   typed-identifier |
                                       “(” typed-identifier “)” |
                                       “(” typed-identifier “)+” |
                                       “*(” typed-identifier “)+” |
                                       “-(” typed-identifier “)” |
                                       longword-constant “(” typed-identifier “)” |
                                       “*” longword-constant “(” typed-identifier “)” |
                                       “*” typed-identifier |
                                       “(” typed-identifier “)” “[” identifier “]” |
                                       “(” typed-identifier “)+” “[” identifier “]” |
                                       “*(” typed-identifier “)+” “[” identifier “]” |
                                       “-(” typed-identifier “)” “[” identifier “]” |
                                       longword-constant “(” typed-identifier “)”
                                            “[” identifier “]” |
                                       “*” longword-constant “(” typed-identifier “)”
                                            “[” identifier “]” |
                                       “*” typed-identifier “[” identifier “]” .

typed-identifier                 ::=   identifier |
```

		identifier "@" *type* | *identifier* "@" *longword-constant* .
operand	::=	*constant* | *variable* .
primary-condition	::=	*operand* *operand* "==" *operand* | *operand* "!=" *operand* | *operand* "<" *operand* | *operand* "<=" *operand* | *operand* ">" *operand* | *operand* ">=" *operand* .
condition	::=	*primary-condition* | "!" *condition* | *condition* "&&" *condition* | *condition* "||" *condition* .
simple-statement	::=	*goto-statement* | *break-statement* | *assignment-statement* | *function-call* | *return-statement* | *save-statement* | *restore-statement* | *empty* ";" .

structured-statement	::=	*if-statement* |
		switch-statement |
		while-statement |
		repeat-statement |
		for-statement .

statement	::=	*label statement* | *simple-statement* | *structured-statement* | *explicit-assembler* | *block* .
statement-sequence	::=	{ *statement* } .
label	::=	*identifier* ":" | "case" *case-constant* ":" | "default" ":" .
case-constant	::=	*byte-constant* | *word-constant* | *longword-constant* .
goto-statement	::=	"goto" *identifier* ";" .
break-statement	::=	"break" ";" .
assignment-statement	::=	*variable* "=" *operand* ";" |

```
                               variable "=" "-" operand ";" |
                               variable "=" "~" operand ";" |
                               variable "=" "&" operand ";" |
                               variable "=" operand "+" operand ";" |
                               variable "+=" operand ";" |
                               variable "=" operand "-" operand ";" |
                               variable "-=" operand ";" |
                               variable "=" operand "*" operand ";" |
                               variable "*=" operand ";" |
                               variable "=" operand "/" operand ";" |
                               variable "/=" operand ";" |
                               variable "=" operand " |" operand |
                               variable " |=" operand |
                               variable "=" operand "~" operand ";" |
                               variable "^=" operand ";" |
                               variable "=" operand "&~" operand ";" |
                               variable "&~=" operand ";" |
                               variable "=" operand "<<" operand ";" |
                               variable "<<=" operand ";" .

function-call           ::=    identifier "(" actual-parameters ")" ";" |
                               variable "=" identifier "(" actual-parameters ")" ";" .

actual-parameters       ::=    actual-parameter {"," actual-parameter} |
                               empty .

actual-parameter        ::=    operand |
                               "&" operand .

return-statement        ::=    "return" ";" |
                               "return" "(" return-expression ")" ";" .

return-expression       ::=    operand |
                               "-" operand |
                               "~" operand |
                               "&" operand .

save-statement          ::=    "save" "(" register-list ")" ";" .

register-list           ::=    identifier {"," identifier}

restore-statement       ::=    "restore" "(" register-list ")" ";" .

if-statement            ::=    "if" "(" condition ")" statement |
                               "if" "(" condition ")" statement "else" statement .

switch-statement        ::=    "switch" "(" variable ")" statement .

while-statement         ::=    "while" "(" condition ")" statement .

repeat-statement        ::=    "repeat" statement-sequence
                               "until" "(" condition ")" .

for-statement           ::=    "for" "(" variable "=" operand "to" operand ")"
                                  statement |
```

“for” “(” *variable* “=” *operand* “downto” “0” “)”
statement .

explicit-assember ::= *opcode explicit-operand-list* .

opcode ::= “chmk” | “chme” | “chms” | “chmu” | “prober” |
“probew” | “rei” | “ldpctx” | “svpctx” | “mfpr” |
“mtpr” | “xfc” | “bpt” | “bug” | “halt” |
“movb” | “movw” | “movl” | “movq” | “movo” |
“movf” | “movd” | “movg” | “movh” | “pushl” |
“clrb” | “clrw” | “clrl” | “clrq” | “clro” |
“clrf” | “clrd” | “clrg” | “clrh” | “mnegb” |
“mnegw” | “mnegl” | “mnegf” | “mnegd” | “mnegg” |
“mnegh” | “mcomb” | “mcomw” | “mcoml” |
“mcomf” | “mcomd” | “mcomg” | “mcomh” | “cvtlb” |
“cvtlw” | “cvtbf” | “cvtbd” | “cvtbg” | “cvtbh” |
“cvtwf” | “cvtwd” | “cvtwg” | “cvtwh” | “cvtlf” |
“cvtld” | “cvtlg” | “cvtlh” | “cvtfb” | “cvtdb” |
“cvtgb” | “cvthb” | “cvtfw” | “cvtdw” | “cvtgw” |
“cvthw” | “cvtfl” | “cvtrfl” | “cvtdl” | “cvtrdl” |
“cvtgl” | “cvtrgl” | “cvthl” | “cvtrhl” | “cvtfd” |
“cvtfg” | “cvtfh” | “cvtdf” | “cvtdh” | “cvtgf” |
“cvtgh” | “cvthf” | “cvthd” | “cvthg” | “movzbw” |
“movzbl” | “movzwl” | “cmpb” | “cmpw” | “cmpl” |
“cmpf” | “cmpd” | “cmpg” | “cmph” | “incb” |
“incw” | “incl” | “tstb” | “tstw” | “tstl” | “tstf” |
“tstd” | “tstg” | “tsth” | “addb2” | “addw2” |
“addl2” | “addf2” | “addd2” | “addg2” | “addh2” |
“addb3” | “addw3” | “addl3” | “addf3” | “addd3” |
“addg3” | “addh3” | “adwc” | “adawl” | “subb2” |
“subw2” | “subl2” | “subf2” | “subd2” | “subg2” |
“subh2” | “subb3” | “subw3” | “subl3” | “subf3” |
“subd3” | “subg3” | “subh3” | “decb” | “decw” |
“decl” | “sbwc” | “mulb2” | “mulw2” | “mull2” |
“mulf2” | “muld2” | “mulg2” | “mulh2” | “mulb3” |
“mulw3” | “mull3” | “mulf3” | “muld3” | “mulg3” |
“mulh3” | “emul” | “emodf” | “emodd” | “emodg” |
“emodh” | “divb2” | “divw2” | “divl2” | “divf2” |
“divd2” | “divg2” | “divh2” | “divb3” | “divw3” |
“divl3” | “divf3” | “divd3” | “divg3” | “divh3” |
“ediv” | “bitb” | “bitw” | “bitl” | “bisb2” |
“bisw2” | “bisl2” | “bisb3” | “bisw3” | “bisl3” |
“bicb2” | “bicw2” | “bicl2” | “bicb3” | “bicw3” |
“bicl3” | “xorb2” | “xorw2” | “xorl2” | “xorb3” |
“xorw3” | “xorl3” | “ashl” | “ashq” | “rotl” |
“poly” | “pushr” | “popr” | “movpsl” | “bispsw” |
“bicpsw” | “movab” | “movaw” | “moval” | “movaq” |
“movao” | “movaf” | “movad” | “movag” | “movah” |
“pushab” | “pushaw” | “pushal” | “pushaq” | “pushao” |
“pushaf” | “pushad” | “pushag” | “pushah” | “index” |
“insque” | “remque” | “insqhi” | “insqti” | “remqhi” |
“remqti” | “ffc” | “ffs” | “extv” | “extzv” |
“cmpv” | “cmpzv” | “insv” | “bneq” | “bnequ” |
“beql” | “beqlu” | “bgtr” | “bleq” | “bgeq” |
“blss” | “bgtru” | “blequ” | “bvs” | “bvc” |

"bgequ" | "blssu" | "bcc" | "bcs" | "brb" |
"brw" | "jmp" | "bbs" | "bbc" | "bbss" |
"bbcs" | "bbsc" | "bbcc" | "bbssi" | "bbcci" |
"blbs" | "blbc" | "acbb" | "acbw" | "acbl" |
"acbf" | "acbd" | "acbg" | "acbh" | "aoblss" |
"aobleq" | "sobgeq" | "sobgtr" | "caseb" | "casew" |
"casel" | "bsbb" | "bsbw" | "jsb" | "rsb" |
"callg" | "calls" | "ret" | "movc3" | "movc5" |
"movtc" | "movtuc" | "cmpc3" | "cmpc5" | "scanc" |
"spanc" | "locc" | "skpc" | "matchc" | "crc" |
"movp" | "cmpp3" | "cmpp4" | "addp4" | "addp6" |
"subp4" | "subp6" | "mulp4" | "mulp6" | "divp4" |
"divp6" | "cvtlp" | "cvtpl" | "cvtpt" | "cvttp" |
"cvtps" | "cvtsp" | "ashp" | ".ABORT" | ".line" |
".file" | ".align" | ".data" | ".text" | ".org" |
".space" | ".byte" | ".word" | ".long" | ".quad" |
".float" | ".double" | ".ascii" | ".asciz" | ".comm" |
".lcomm" | ".globl" | ".set" | ".lsym" | ".stab" |
".stabs" | ".stabn" | ".stabd" | "jneq" | "jnequ" |
"jeql" | "jeqlu" | "jgtr" | "jleq" | "jgeq" |
"jlss" | "jgtru" | "jlequ" | "jgequ" | "jlssu" |
"jbss" | "jbcs" | "jbsc" | "jbcc" | "jlbs" |
"jlbc" | "jcc" | "jcs" | "jvs" | "jvc" |
"jbs" | "jbc" | "jbr" .

explicit-operand-list ::= *explicit-operand* {"," *explicit-operand*} |
empty .

explicit-operand ::= *byte-constant* |
"$" *byte-constant* |
word-constant |
"$" *word-constant* |
longword-constant |
"$" *longword-constant* |
floating-constant |
double-constant |
character-constant |
string-constant |
variable .

empty ::= .

References

Barr83

Alan H. Barr: Global and Local Deformations of Solid Primitives, [in *SIGGRAPH'83: State–of–the–Art in Image Synthesis Seminar Notes*], July 1983.

Barsky82

Brian A. Barsky: End Conditions and Boundary Conditions for Uniform B-Spline Curve and Surface Representations, *Computers in Industry 3 (1/2)*, March/June 1982, 17-29.

Barsky83

Brian A. Barsky and John C. Beatty: Controlling the Shape of Parametric B-spline and Beta-spline Curves, *Graphics Interface '83*, May 1983, 223-232.

Bartels83

Richard H. Bartels, John C. Beatty and Brian A. Barsky: An Introduction to the Use of Splines in Computer Graphics, Technical Report CS-83-09, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, 1983, 182 pages.

Carpenter80

Loren Carpenter et. al.: Vol Libre, [animation sequence], SIGGRAPH Video Review Issue #2.

Catmull74

E.E. Catmull: A Subdivision Algorithm for Computer Display of Curved Surfaces, PhD dissertation, University of Utah, 1974.

Cleary83

John G. Cleary, Brian Wyvill, Graham M. Birtwistle and Reddy Vatti: Multiprocessor Ray Tracing, Research Report 83/128/17, Department of Computer Science, University of Calgary, October 1983.

Cook81a

Robert L. Cook and Kenneth E. Torrance: A Reflection Model for Computer Graphics, *Computer Graphics 15 (3)*, August 1981, 307-316.

Cook81b

Robert L. Cook: A Reflection Model for Realistic Image Synthesis, Masters Thesis, Cornell University, 1981.

Cowan83

William Cowan: An Inexpensive Scheme for Calibration of a Colour Monitor in Terms of CIE Standard Coordinates *Computer Graphics 17 (3)*, July 1983, 315-321

Foley82

James D. Foley and Andries van Dam: Fundamentals of Interactive Computer Graphics, Addison Wesley, 1982, 664 pages.

Fournier82

Alain Fournier, Don Fussel and Loren Carpenter: Computer Rendering of Stochastic Models, *Communications of the ACM 25 (6)*, June 1982, 371-384.

Gouroud71

H. Gouroud: Continuous Shading of Curved Surfaces, *EEE Transactions on Computers C-20 (6)*, June 1971, 623-628.

Hall83a

Roy A. Hall and Donald P. Greenberg: A Testbed for Realistic Image Synthesis, *IEEE Computer Graphics and Applications 3 (8)*, November 1983, 10-20.

Hall83b

Roy A. Hall: A Methodology for Realistic Image Synthesis, Masters Thesis, Cornell University, 1983.

Handbook72

American Institute of Physics Handbook, McGraw–Hill, 1972.

Hanrahan83

Patrick Hanrahan: Ray Tracing Algebraic Surfaces, *Computer Graphics 17 (3)*, July 1983, 83-90.

Henrici64

Peter Henrici: Elements of Numerical Analysis, John Wiley and Sons, 1964, pp 105-107.

Kajiya82

James T. Kajiya: Ray Tracing Parametric Patches, *Computer Graphics 16 (3)*, July 1982, 245-254.

Kajiya83a

James T. Kajiya: New Techniques for Ray Tracing Procedurally Defined Objects, *Transactions on Graphics 2 (3)*, July 1983, 161-181.

Kajiya83b

James T. Kajiya: Siggraph'83 Tutorial on Ray Tracing, [in *SIGGRAPH'83: State–of–the–Art in Image Synthesis Seminar Notes*], July 1983.

Kerninghan78

Brian W. Kerninghan and Dennis M. Ritchie: The C Programming Language, Prentice–Hall, 1978, 228 pages.

Krinov47

E. L. Krinov: Spectral Reflectance Properties of Natural Formations, Izdaltel'stvo Akademi, Nauk, U.S.S.R., 1947, 268 pages [available as Technical Translation TT-439, National Research Council of Canada].

Kay79

Douglas S. Kay: Transparency, Refraction and Ray Tracing for Computer Synthesized Images, Masters Thesis, Cornell University, 1979.

Lane80

J.M. Lane, L.C. Carpenter, J.T. Whitted and J.F. Blinn, Scan Line Methods for Displaying Parametrically Defined Surfaces, *Communications of the ACM 21 (1)*, January 1980, 23-34.

Mandelbrot77

Benoit B. Mandelbrot: Fractals: Form, Chance and Dimension, Freeman, 1977, 281 pages.

Max81

Nelson Max: Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset, *Computer Graphics 15*, 3, August, 1981, 317-324.

Newman73

William M. Newman and Robert F. Sproll: Principles of Interactive Computer Graphics, McGraw–Hill, 1973, 607 pages.

Phong75

Bui Tuong Phong: Illumination for Computer Generated Pictures, *Communications of the ACM 18 (6)*, June 1975, 311-317.

Potmesil82

Michael Potmesil and Indranil Chakravarty: Synthetic Image Generation with a Lens and Aperture Camera Model, *ACM Transactions on Graphics 1 (2)*, April 1982, 85-108.

Purdue70

Purdue University: Thermophysical Properties of Matter, Plenum, 1970 [13 volumns].

Reeves83

William T. Reeves: Particle Systems - A Technique for Modeling a Class of Fuzzy Objects, *Transactions on Graphics 2 (2)*, April 1983, 91-108.

Riesenfeld73

Richard F. Riesenfeld: Applications of B-spline Approximation to Geometric Problems of Computer-Aided Design, PhD dissertation, Department of Systems and Information Science, Syracuse University, 1973.

Riesenfeld80

Richard F. Riesenfeld, Elaine Cohen, and Tom Lyche: Discrete B-splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics, *Computer Graphics and Image Processing 14 (2)*, October 1980, 87-111.

Roth80

Scott D. Roth: Ray Casting as a Method for Solid Modeling, General Motors Research Laboratories, GMR−3466 [in *SIGGRAPH '81: Seminar on Solid Modeling Notes*], 1980.

Roth82

Scott D. Roth: Ray Tracing for Modeling Solids, *Computer Graphics and Image Processing 18*, 1982, 109-144.

Rubin80

Steven M. Rubin and Turner Whitted: Three−Dimensional Representation for Fast Rendering of Complex Scenes, *Computer Graphics 14 (3)*, 1980, 110-116.

Schweitzer82

Dino Schweitzer and Elizabeth S. Cobb, Scanline Rendering of Parametric Surfaces, *Computer Graphics 16 (3)*, 1982, 265-271.

Sparrow66

E.M. Sparrow and R.D. Cess: Radiation Heat Transfer, Brooks/Cole, 1966, 322 pages.

Torrance67

Kenneth E. Torrance and E.M. Sparrow: Theory for Off-Specular Reflection from Roughened Surfaces, *J. Op. Soc. Am. 57 (3)*, Sept 1967, 1105-1114.

Ullner83

Michael K. Ullner: Parallel Machines for Computer Graphics, PhD dissertation, California Institute of Technology, 1983.

Unix83

Unix Programmer's Manual, Computer Science Division, Department of Electrical Engineering and Computer Science, Berkeley, California, 1983.

Vax81

DEC Systems Group: VAX Architecture Handbook, 1981, 506 pages.

Whitted80

Turner Whitted: An Improved Illumination Model for Shaded Display, *Communications of the ACM 23 (6)*, June 1980, 343-349.

Whitted82

Turner Whitted: Processing Requirements for Hidden Surface Elimination and Realistic Shading, *IEEE Spring Coupon*, 1982, 245-250.

Whitted83

Turner Whitted: mentioned during the presentation of the *SIGGRAPH'83 Advanced Images Synthesis Seminar*, July 1983.

Wyszecki82

G. Wyszecki and W.S. Stiles: Colour Science, John Wiley and Sons, 1982 [second edition], 950 pages.

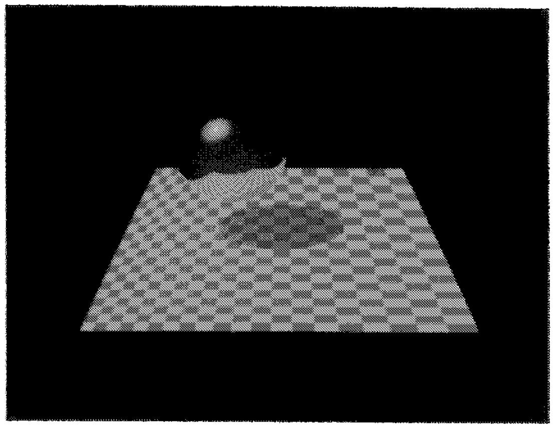


Figure 5. The final image produced (18:30 minutes).

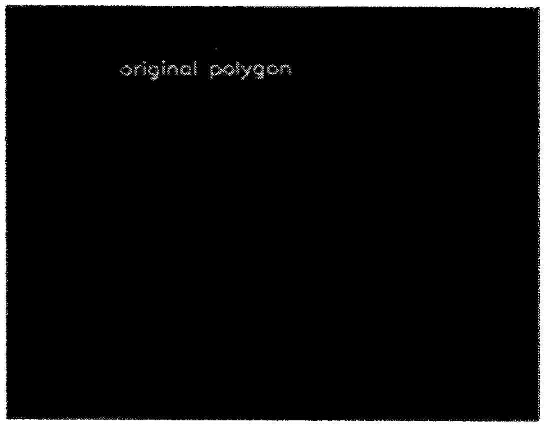


Figure 13. The final triangle (13:49 minutes).

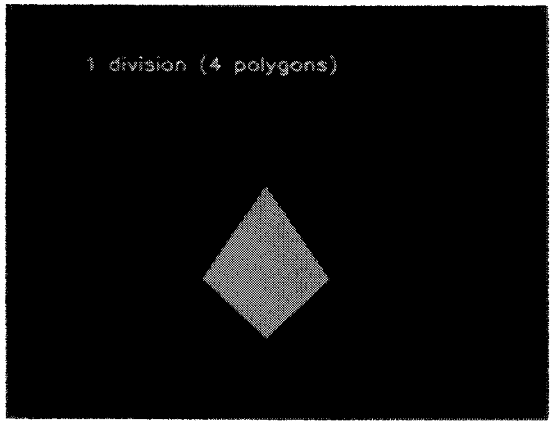


Figure 14. After 1 subdivision (22:45 minutes).

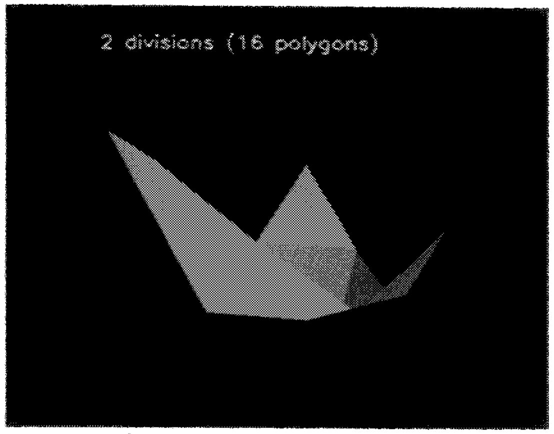


Figure 15. After 2 subdivisions (26:37 minutes).

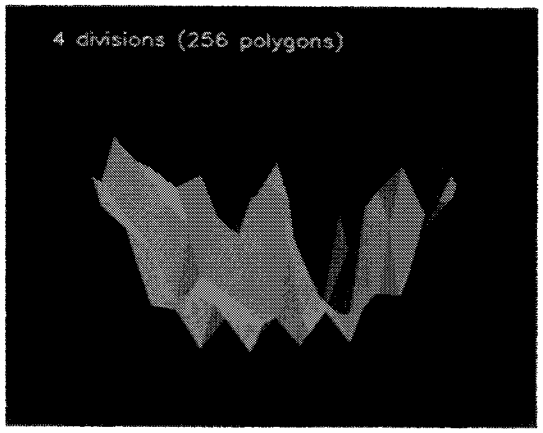


Figure 16. After 4 subdivisions (32:53 minutes).

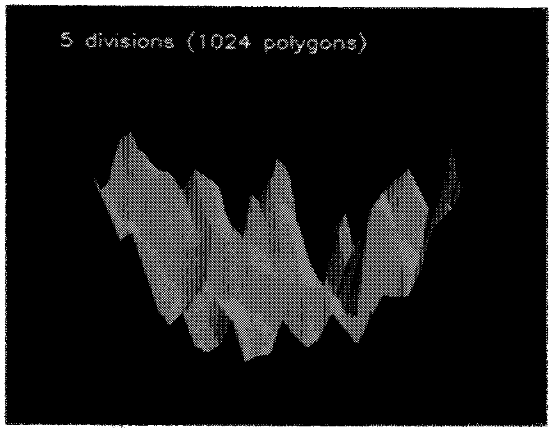


Figure 17. After 5 subdivisions (36:39 minutes).

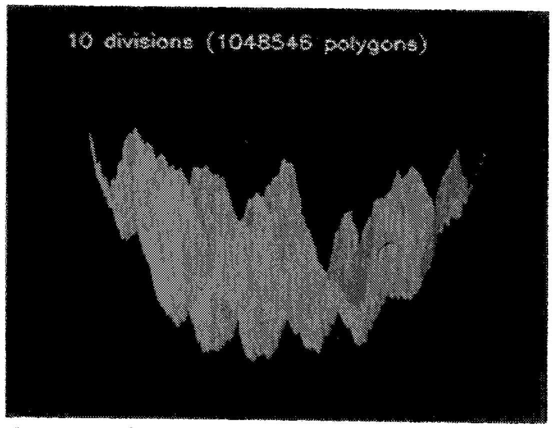


Figure 18. After 10 subdivisions (54:06 minutes).

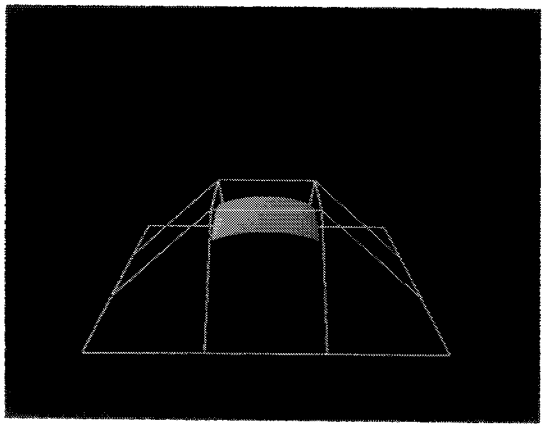


Figure 28. A patch as it would result from single boundary vertices (2:55 minutes).

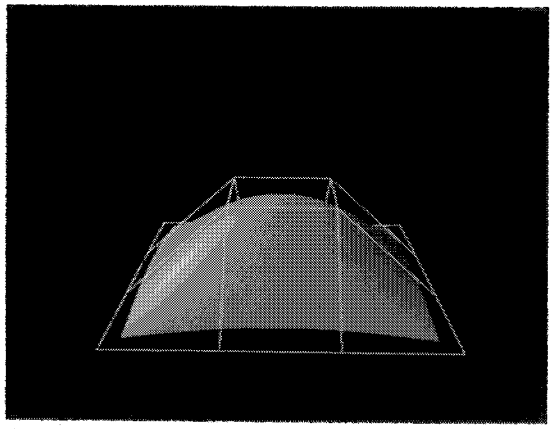


Figure 29. A patch as it would result from double boundary vertices (26:49 minutes).

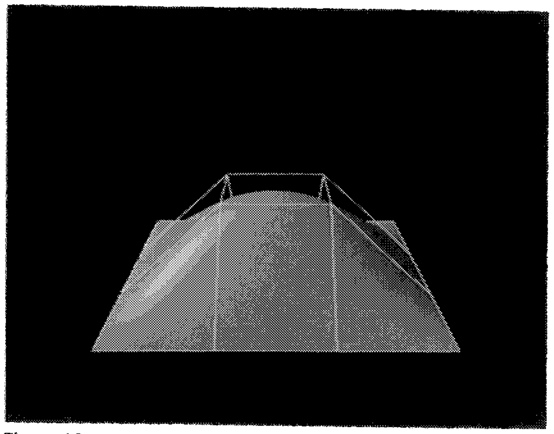


Figure 30. A patch as it would result from triple boundary vertices (32:36 minutes).

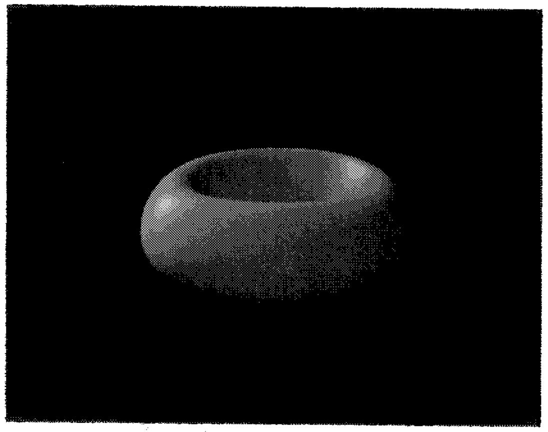


Figure 31. A closed spline donut (93:04 minutes).

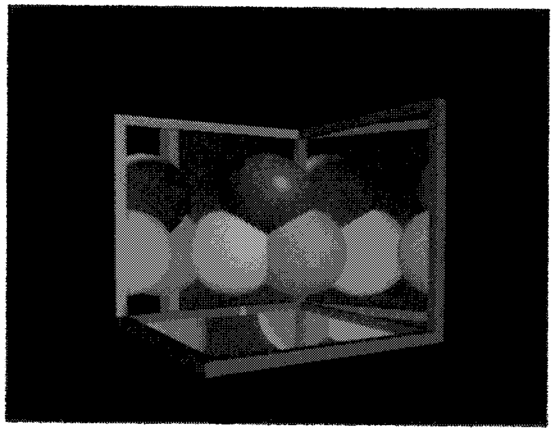


Figure 38. An image with incorrect perspective ($\approx$ 90 minutes).

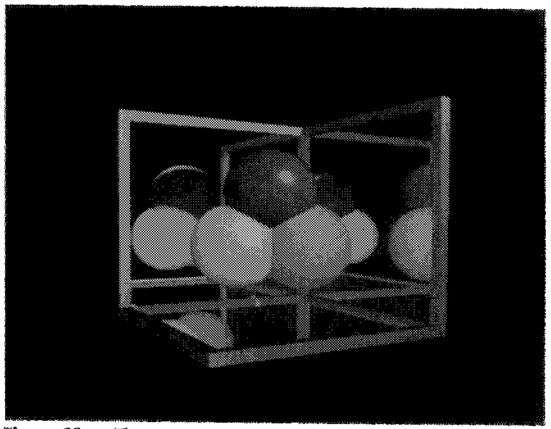


Figure 39. The same image with correct perspective (109:28 minutes).

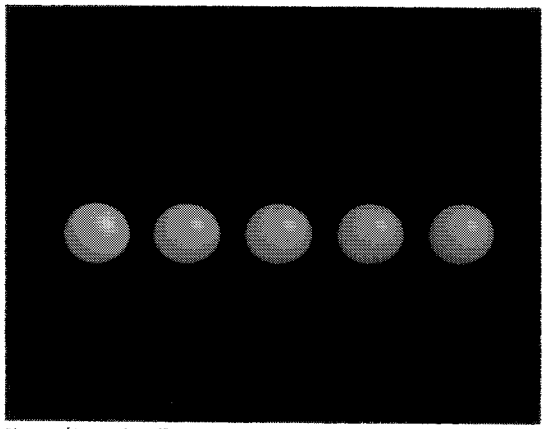


Figure 45. The effects of various amounts of anti-aliasing. From left to right, the balls were generated by: the program not configured to do anti-aliasing (1:01 minutes), **aalevel** = **1** (1:06 minutes), **aalevel** = **2** (1:33 minutes), **aalevel** = **4** (1:55 minutes), **aalevel** = **8** (2:33 minutes). In all cases, the **aathreshold** was set to 40.0.

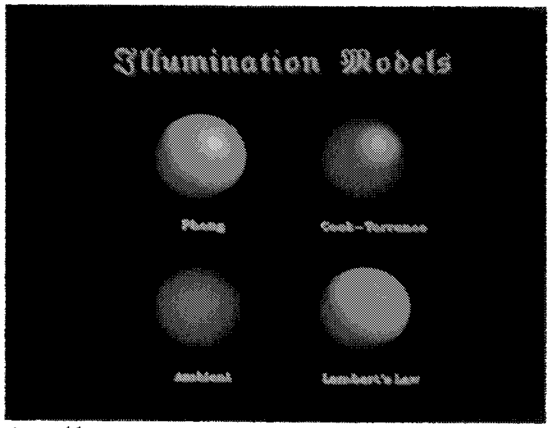


Figure 46. The three illumination models supported, and our ambience function (9:04 minutes).

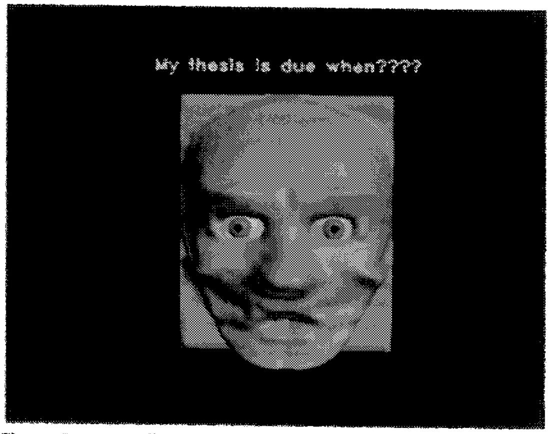


Figure 53. A Halloween mask was cut into strips and digitized, giving the control mesh for this spline surface (72:29 minutes).

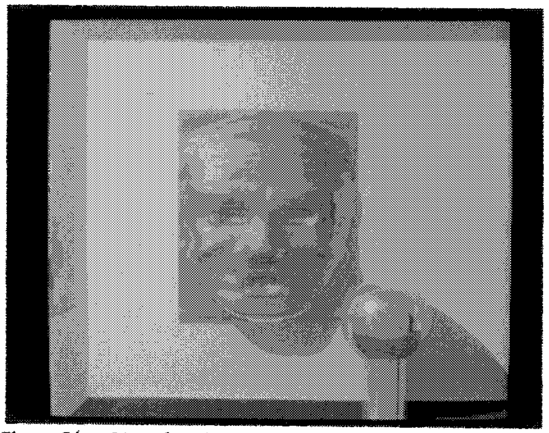


Figure 54. View of a gallery room with a spline mask. The mask is rendered in copper, and the sphere and cylinder are stainless steel. The copper and steel were achieved using the Cook-Torrence illumination model. The sphere and mask reflect each other, as well as a tapestry and doorway directly behind the viewer and pictures on the side walls (≈ 35 hours).

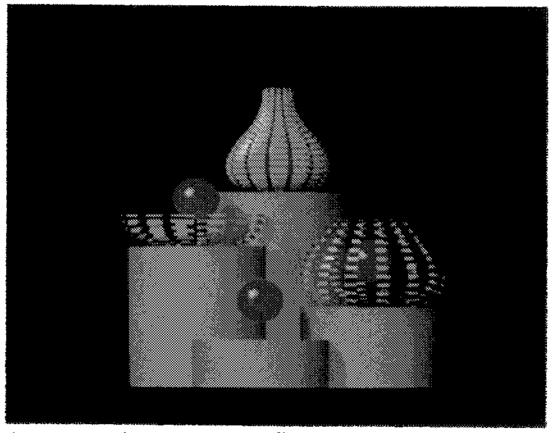


Figure 55. Three vases with insufficient overlap. From the highest to the lowest vases, the overlaps are 0.4, 0.3 and 0.2 (90:32 minutes).

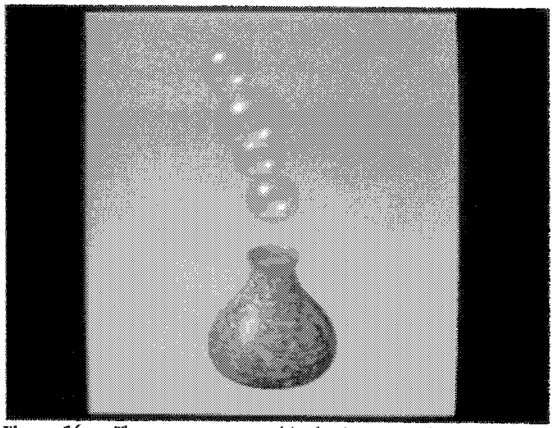


Figure 56. The texture maps used in this image are the work of local artist Karen Fletcher (73:44 minutes).

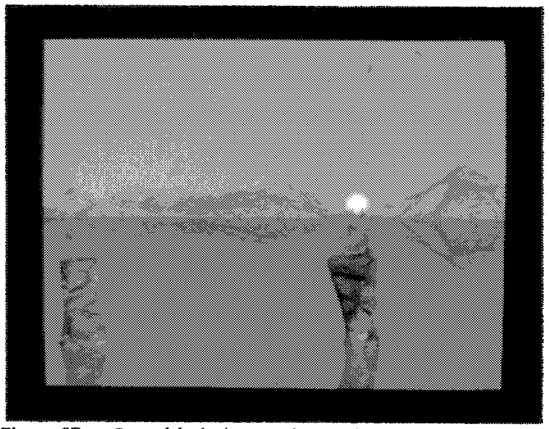


Figure 57. Crater lake looking north towards the inlet. The hills are textured mapped fractals, the columns are spline surfaces, and the water is a reflective rectangle. Blinn's dusty surface lighting model was specially adapted by David Forsey to render the glowing sphere (106:38 minutes).

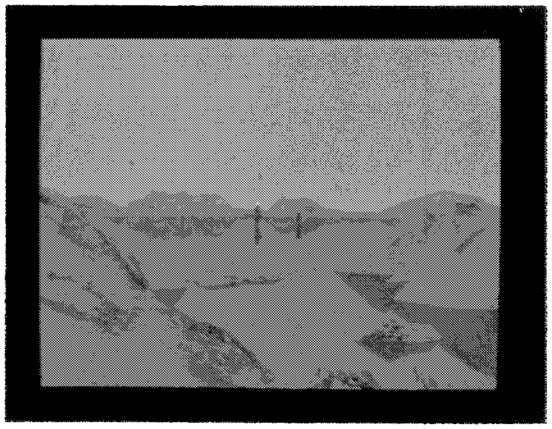


Figure 58. Crater lake as seen from the inlet (228:09 minutes).