A description of a simple Ray Tracer

David Malone

December 1995 *

Contents

*Last LaTeX'ed December 6, 2000.

Introduction

This ray tracer has been written as part of the course work for 4D4, a course on computer graphics, it has since been expended for my own amusement. The assignment required the following features to be implemented:

- Simple Objects such as Spheres and Planes.
- Point light sources.
- A simple camera model with no depth of field.
- Phong Illumination model, with simple shadowing.
- Recursive reflection and refraction.
- A simple input language/method.
- Output in `PPM` format.

These will be discussed in the order they were implemented, but first it is necessary to explain the internal working of the code I decided on.

Also, the strengths and weaknesses of the implementation, the model and the features included will be discussed.

1 Implementation

1.1 Overview

1.1.1 Structure of Object information

All the aspects of a scene have one common property, that is they must be read into the program and stored. This suggests having a structure to represent each different aspect and a function which can fill this structure. For this reason I decided to have separate modules for each object type, providing the facility to read and fill the structure.

In the spirit of object orientation the structure should only be accessed directly by a small subset of the code, so for each primitive, functions were included to do intersection and outward normal calculation.

To then allow the use of these functions a table of all object types, with pointer to their read, normal and intersection functions is provided for lookup within the program.

Objects which are not primitives (eg. light sources, the viewer and surface descriptions) have their structures filled by the read function, and then are accessed directly by the ray tracing code. This is reasonable as these are parameters of the ray tracing itself, though it leads to problems of expendability.

1.1.2 Structure of Scene information

When read in, the scene consists of several parts which are all linked to one structure. A list of primitives, their surfaces and types are stored as a linked list. Also the list of light sources are stored as a linked list. The viewer and other information which there is only one occurrence of are just pointed to.

1.1.3 Rays

As rays are rather basic to the whole program a ray structure is used. It consists an origin and a direction, which should be normalised by the creator of the ray. A length is also stored for rays of finite length rays of infinite length are given the value C value `DBL_MAX`.

1.1.4 Spectra

Lights and surfaces have spectra associated with them, and in the simple case the spectra have 3 components - Red, Green and Blue. For the sake of future expansion the number of components has been made a definable value.

1.2 A simple input language

The input language I choose to use is not very human friendly, but quite machine friendly. It is not the job of a ray tracer to pre-process the data it receives, that is the job of a pre-processor (witness `C`).

Each object is assigned an ID at compile time. The input consists of a series of objects, terminated by the end of file. An object is an ID followed by a list of numbers describing the object and if the object is a primitive a surface.

For example, in the following input the first object is a viewer (ID 102), the code for reading a viewer then reads the next 14 numbers. The next object is a point light source (ID 101). The following 6 numbers describe it. Finally the last object is a plane (ID 2), the code for reading planes reads the next 4 numbers. The 100 indicates a the most simple surface type, then the read surface code is called to read 8 numbers.

```
102
0 2 3
0 0 0
1 0 0
0 1 0
256 256

101
-2 5 2
1 1 1

2
0
0 1 0
100
1 1 1
0.0 30
1.0 0 0
```

To understand the descriptions you must read the documentation for them. To ease the writing of a preprocessor a summary of the what the numbers means if defined, eg a sphere might be:

```
radius 1 center 3
```

This means that the description of a sphere consists of first a number (the radius) and then 3 numbers (the center).

1.3 Simple Objects & Primitives

Light sources, viewers, spheres and the like are all considered objects. Geometric objects with which the light can interact are termed primitives. All objects can be read in, primitives can also have rays intersected with them and normals to points calculated.

Surfaces are also implemented as objects, though an error will be produced if they are read in alone. Various surface types provide a way of varying the shading parameters for a depending on a given point. You may also set a default surface and tell an primitive to use the current default object as its surface.

Lights can also vary their spectrum depending on the point they are illuminating - this allows for lights which do not have a r^{-2} effect, spotlights, etc. They may also be sampled several times to allow for non-point light sources.

1.3.1 Reading objects

The read function of any object simply reads some number of doubles from a passed file pointer, and then fills out the structure for the object — possibly doing some useful precalculation first. It returns a pointer to the filled structure.

1.3.2 Normals and Intersections with primitives

For primitives the intersection code takes a ray structure, an object structure (of the appropriate type). It returns how far along the ray the closest intersection occurs. It may disgrade intersections which are further away than the length of the ray. The normal code takes an object structure and a point, and returns the normal to the object at that point.

1.3.3 Surfaces

A surface contains the information for the Phong illumination model, and other parts of the ray tracer. It stores the reflection, transmission and spectral characteristics of an object. A call to a surface is provided with the description of this instance of the surface type and the point that is being illuminated — it returns the Phong illumination parameters for that point.

1.4 Lights

Lights have three parameters — how many times they are to be sampled their position and their spectrum. The sample count function is given the point which it will illuminate and returns how many samples should be taken. The sample function takes the point to be illuminated and returns the position, and spectrum which should be considered for this sample.

1.5 Rasters and Output

The image is stored in a raster during generation, and is outputted to a PPM file when generation is complete. The raster is created, filled and outputted without access directly to the structure. This would allow normalisation/gamma correction of the values output, by monitoring what is written into the raster as it is written.

The raster for an image of resolution X_{RES}, Y_{RES} is stored as an array of $X_{RES} * Y_{RES} *$ (Number of spectral components) doubles. To get sensible memory accesses the spectral components of each pixel are stored consecutively. The raster should be written left ($X = 0$) to right ($X = X_{RES}$) horizontally from the top ($Y = 0$) to the bottom ($Y = Y_{RES}$) of the image for linear memory accesses.

1.6 Camera Model, Viewer and Initial Ray Generation

The specification of the viewer consists of the viewers location, at what point they are looking, two vectors to be used as the horizontal and vertical axes of the image, and horizontal and vertical resolution.

The image is projected onto a plane going through a point at distance 1 along the ray from the viewer to where they are looking. The plane is generated by adding multiples of the two vectors to this point.

As the vectors need not be perpendicular to one another, nor to the ray from the viewer "interesting" effects can be produced by choosing arbitrary vectors.

A function which takes a ray and then calculates the light shining back from that ray is called once the ray has been generated, and the result is written into the raster. This function is also the basis for recursion, and is called to follow reflected and transmitted rays.

1.6.1 Implementation Details

Let $\vec{e}$ be the location of the viewer and $\vec{l}$ be where they are looking at. Let $\vec{x}$ and $\vec{y}$ be our horizontal and vertical vectors. Then the section of plane which is viewed is:

$$\vec{v}(\alpha, \beta) = \vec{e} + \frac{\vec{l} - \vec{e}}{|\vec{l} - \vec{e}|} + \alpha\vec{x} + \beta\vec{y},$$

where $-1 \leq \alpha, \beta \leq 1$. For a point (X, Y) in the raster we we simply take out initial ray to be centered at $\vec{e}$ and to go toward

$$\vec{v}\left(\frac{2X - X_{RES}}{X_{RES}}, -\frac{2Y - Y_{RES}}{Y_{RES}}\right).$$

The minus in the Y direction is to account for computers having the Y on their screens pointing the wrong way.

1.7 Phong Illumination Model

The Phong illutation model gives the basic contribution of light to the scene. It depends on the light sources in the scene, the surface characteristics of the object and the location and normal of the point on the object.

1.7.1 The Model

If there is an ambient light source the contribution for each part of the spectrum is:

$$I_a k_d,$$

where I_a is the intensity of the ambient light source and k_d is the diffuse coefficient of the surface. Then for each light source which the surface faces, and is not blocked by some object we add:

$$I_l \frac{k_d(\vec{d} \cdot \vec{s}) + k_s(\vec{r} \cdot \vec{s})^{k_e}}{l^2},$$

where I_l is the intensity of the light, $\vec{d}$ is a unit normal to the surface, $\vec{s}$ is unit vector toward the light, $\vec{r}$ is a unit vector in the reflected direction, k_s, k_e are spectral characteristics and l is the distance to the light. [1]

[1] The Phong model does not usually contain the $\frac{1}{l^2}$ term.

1.7.2 Implementation Details

The determination of most terms in the model is trivial. Most are known, or provided by the primitive functions. The remaining are calculated as follows.

$$l = |\vec{x_l} - \vec{x}|$$

$$\vec{s} = \frac{\vec{x_l} - \vec{x}}{l}$$

Where $\vec{x_l}$ is the location of the light and $\vec{x}$ is our point. To determine if a surface faces a light source we simply check if:

$$\vec{d} \cdot \vec{s} > 0.$$

The reflected ray is calculated as shown later, we check that $\vec{r} \cdot \vec{s}$, to see if the specular reflection of the light is visible in this direction.

It is assumed that $\vec{d}$ has been chosen to give $\vec{i} \cdot \vec{d} \leq 0$, that is that $\vec{d}$ is the normal which points to where the ray is coming from. (Where $\vec{i}$ is the direction of the incoming ray).

To determine if a light is shadowed, we find the ray from $\vec{x}$ to the light source and look for the an intersection on the correct side of the ray, with distance less than l. If such an intersection is found then we say the light is shadowed. [2]

1.8 Reflection

If we have just found a point of intersection along a ray, we can find a second ray which, if light were traveling along it, would be reflected along the first ray. We call this new ray the reflected ray. Using the fact that the reflected and original rays make the same angle with the normal, it is relatively easy to show the direction of the reflected ray is:

$$\vec{r} = \vec{i} + 2(\vec{i} \cdot \vec{d})\vec{d},$$

where $\vec{i}$ and $\vec{d}$ are the incoming direction and normal to the surface respectively. To add recursive reflection we simply calculate this ray, and evaluate the light coming along it and add this to the illumination of the point (scaled by k_s one of the spectral reflection coefficients).

1.9 Refraction

In a similar manner to reflection, a refracted ray can be calculated using Snell's law, for a ray moving from a material of refractive index n_1 to n_2, giving the following:

$$\eta = \frac{n_1}{n_2},$$

$$\eta\vec{i} + \left(\eta(\vec{i} \cdot \vec{d}) - \sqrt{1 - \eta^2(1 - (\vec{i} \cdot \vec{d})^2)}\right)\vec{d},$$

providing $1 - \eta^2(1 - (\vec{i} \cdot \vec{d})^2) >= 0$. Otherwise total internal reflection occurs. In this case the transmission coefficient is added to the spectral reflection coefficient. [3]

1.9.1 Implementation Details

The only unknown quantity is η, and some assumptions must be made to find it. First the ray tracer assumes the initial rays start in a medium of refractive index 1. Then as the program recurses the refractive index is passed on. When transmission occurs the refractive index of the transmitted ray is set to be 1 if leaving an object and the refractive index of the object otherwise.

[2] This is a bit simplistic if we have transparent objects. More sophisticated shadows can be achieved by tracing rays from the light sources. This is beyond the scope of a simple ray tracer though.

[3] It would be more realistic to implement the Fresnel equations here.

The test for leaving an object is based on $(\vec{i} \cdot \vec{n})$, where $\vec{n}$ is the outward unit normal. (As opposed to $\vec{d}$ which is the normal pointing toward the ray's origin).

As total internal reflection can affect the Phong and reflected shading the transmission should be evaluated first, to determine if the spectral reflection coefficient should be augmented.

1.10 Other Implementation Details

This section contains other small details of implementation which don't seem to fit under the above headings.

1.10.1 Trapped Rays

Sometimes, due to numerical inaccuracy, a newly generated ray will hit the surface it has just left. This produces random speckles on the surfaces, as if the surface has been grated with a cheese grater.

The most simple way to get around this is to reject all hits which are either behind or a small amount in front of the object. This is the solution currently used. A variation on this would be to reject only intersections with the current object which are very close. A different scheme might involve "shunting" the ray in the direction it is going a little before looking for intersections.

1.10.2 When to Stop Recursing

You cannot recurse indefinitely in a general scene, as if you have two parallel mirrors the light may bounce back and forth between them indefinitely. For this reason we need a way out of the recursion. There are 4 reasonable criterion for stopping. The first is if you hit a an object which is either not transparent or not reflective, then it is not necessary to recurse on the the ray which makes zero contribution. Second, if a ray escapes from the scene it can dropped, making either no contribution, or that of the ambient light.

The third and forth reasons are more of a practical nature. Third, if the contribution the ray would make is very small the ray need not be followed. This is implemented by keeping track of the product of reflection and transmission coefficients involved in getting to this stage of the recursion. The problem is to choose a suitable cut off[4]. I have chosen the reciprocals of the number of colours in the output by some factor (currently 0.01).

Finally, if the recursion gets too deep then you can break out. This can produce block spots, or lack of details in your image if you do not recurse to a deep enough level! This depth can be given on the command line and currently defaults to 7.

1.10.3 The Main Program

The main program is really just a skeleton. A more complicated interface could be written if desired, perhaps including reading a series of files and placing the output named files.

```
int main(int argc,char **argv)
{
   struct scene *scene;
   struct raster *raster;

   set_default_options();
   set_arg_options(argc,argv);
   scene = read_scene(Opt.input);
   if( !Opt.read_only )
   {
      raster = ray_trace(scene);
      raster2ppm(raster,Opt.output);
```

[4]This cutoff if called `MIN_WEIGHT` in the code.

```
    }
    rt_unmalloc();
    rt_end("Finished OK!\n");

    return 0;
}
```

2 Primitives

For each primitive in the ray tracer there are three major elements. First the mathematics of the primitive which defines the surface, and allows us to calculate the intersections and normals and also importantly how to define the object. This may suggest some preprocessing of the object's definition to make calculations shorter.

The second is the implementation of the mathematics. It may be quite straight forward to code what has been derived, or maybe not. The code also has to include the third factor, which is how the object will be specified in the input file, down to the order of the inputs.

What follows here is a description of the primitives included with the object, detailing the three above factors.

2.1 Building on Simple Primitives

The most simple type of primitives are geometric objects such as spheres, planes or infinite double cones, which are defined by some simple equation:

$$f(\vec{x}) = 0,$$

where f is some continuous well behaved function. The normal at $\vec{x}_0$ normal can then be found by evaluating:

$$\vec{\nabla}_{\vec{x}} f(\vec{x})\Big|_{\vec{x_0}},$$

although the normal can often be found more easily by geometric arguments.

To make these objects independent of basis we would expect f to depend on $\vec{x} \cdot \vec{x}$ and $\vec{x} \cdot \vec{p}$ where $\vec{p}$ is a vector relating to the primitive concerned (maybe a normal).

By using $\vec{x} = \vec{o} + t\vec{d}$ we make the problem of finding intersections into the problem of solving:

$$0 = F(t) = f(\vec{o} + t\vec{d}).$$

By adding restrictions of suitable values of $\vec{x}$ we can form more primitives. Again the restrictions should depend on $\vec{x} \cdot \vec{x}$ and $\vec{x} \cdot \vec{p}$. For example from a plane it is simple to form a disk by only allowing values of $\vec{x}$ where: [5]

$$(\vec{x} - \vec{p})^2 \leq r^2$$

More complicated primitives can be built from these simpler ones.

2.2 The Sphere Primitive

A sphere is the set of points a given distance r from a center point $\vec{c}$. This is easily expressed in vector form as:

$$(\vec{x} - \vec{c})^2 = r^2$$

where $\vec{x}$ is some point on the sphere.

[5] This actually depends only on scalars and things of the form mentioned if you multiply it out.

Using $\vec{o} + t\vec{d}$ we get a quadratic for t:

$$\begin{aligned}(\vec{o} + t\vec{d} - \vec{c})^2 &= r^2 \\ (\vec{o} - \vec{c})^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{d} + t^2\vec{d}^2 &= r^2 \\ (\vec{o} - \vec{c})^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{d} + t^2\vec{d}^2 &= r^2 \\ t^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{d} + (\vec{o} - \vec{c})^2 - r^2 &= 0\end{aligned}$$

Using the fact that $\vec{d}$ should be normalised. This is easily solved using:

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

Small values should be rejected when choosing the smallest positive value t, as rays which are being transmitted from one side if the sphere to the other may be subject to getting trapped, as explained above.

Also note that r^2 can be calculated at the time the object is read in, saving you a whole multiplication!

The normal to the point $\vec{x}_0$ on a sphere is along the line from the center to this point. All we have to do is normalise and return the surface normal:

$$\vec{n} = \frac{\vec{x}_0 - \vec{c}}{|\vec{x}_0 - \vec{c}|}.$$

A sphere is specified as a radius followed by the components of its center. The unit sphere around the origin would be:

```
1
0 0 0
```

2.3 The Plane Primitive

Suppose you have a plane passing through $\vec{x}_0$ with normal $\vec{n}$. Let $\vec{x}$ be any point in the plane then $\vec{x} - \vec{x}_0$ is a vector along the plane, and so:

$$\begin{aligned}\vec{n} \cdot (\vec{x} - \vec{x}_0) &= 0, \\ \vec{n} \cdot \vec{x} &= \vec{n} \cdot \vec{x}_0, \\ \vec{n} \cdot \vec{x} &= D.\end{aligned}$$

Filling in $\vec{x} = \vec{o} + t\vec{d}$ and solving for t we get:

$$\begin{aligned}\vec{n} \cdot (\vec{o} + t\vec{d}) &= D, \\ t\vec{n} \cdot \vec{d} &= D - t\vec{n} \cdot \vec{o}, \\ t &= \frac{D - t\vec{n} \cdot \vec{o}}{\vec{n} \cdot \vec{d}},\end{aligned}$$

providing that $\vec{n} \cdot \vec{d} \neq 0$. If it is, the line is along the plane, and in this implementation no intersection in returned. The normal is simply $\vec{n}$ everywhere.

A plane is specified by D followed by the three components of $\vec{n}$. For example the following is the plane passing through $(3, 4, 5)$ with normal $(0, 0, 1)$

```
5
0 0 1
```

2.4 The Disk and Annulus Primitives

The disk primitive is just the plane primitive with the extra condition that an intersection only occurs if $\vec{x}$ is within a distance r of some centre. The annulus is the same, but also check the distance is greater than r_{in}.

This actually adds quite a lot of extra work to the task of checking an intersection as:

$$\vec{x} = \vec{o} + t\vec{d},$$

is evaluated. In the respective cases r^2 and r^2, r_{in}^2 may be precalculated at read time.

Disks are described by their centre, normal and radius. Annuli are the same, but with their r_{in} following the radius.

The following is a disk of radius 2.5 with centre $(1, 1, 1)$ with its normal pointing along $(4, 3, -1)$:

```
1 1 1
4 3 -1
2.5
```

And replacing the above with an annulus of inner radius 0.23 and outer radius 4

```
1 1 1
4 3 -1
4 0.23
```

2.5 The Triangle Primitive

A triangle with one vertex at the origin and the other two at $\vec{a}, \vec{b}$ can be thought of as the set of points:

$$\{\alpha\vec{a} + \beta\vec{b} : \alpha, \beta \geq 0 \text{ and } \alpha + \beta \leq 1\}$$

If given a point we with to determine if α, β satisfy the inequalities given.

Given a vector $\vec{x}$ we suppose it it can be written in the form $\alpha\vec{a} + \beta\vec{b}$ (which it can be if it is in the plane containing $\vec{a}$ and $\vec{b}$). Then by taking the dot product with both $\vec{a}$ and $\vec{b}$ we get:

$$\begin{aligned} \vec{a} \cdot \vec{x} &= \alpha\vec{a} \cdot \vec{a} + \beta\vec{a} \cdot \vec{b} \\ \vec{b} \cdot \vec{x} &= \alpha\vec{b} \cdot \vec{a} + \beta\vec{b} \cdot \vec{b} \end{aligned}$$

We can know all these terms but α and β. Writing this in matrix form and inverting:

$$\begin{aligned} \begin{pmatrix} \vec{a} \cdot \vec{a} & \vec{a} \cdot \vec{b} \\ \vec{b} \cdot \vec{a} & \vec{b} \cdot \vec{b} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} &= \begin{pmatrix} \vec{a} \cdot \vec{x} \\ \vec{b} \cdot \vec{x} \end{pmatrix} \\ \begin{pmatrix} \alpha \\ \beta \end{pmatrix} &= \frac{1}{\vec{a}^2\vec{b}^2 - (\vec{a} \cdot \vec{b})^2} \begin{pmatrix} \vec{b} \cdot \vec{b} & -\vec{a} \cdot \vec{b} \\ -\vec{a} \cdot \vec{b} & \vec{a} \cdot \vec{a} \end{pmatrix} \begin{pmatrix} \vec{a} \cdot \vec{x} \\ \vec{b} \cdot \vec{x} \end{pmatrix} \quad \text{or} \\ \begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} &= \begin{pmatrix} \vec{b} \cdot \vec{b} & -\vec{a} \cdot \vec{b} \\ -\vec{a} \cdot \vec{b} & \vec{a} \cdot \vec{a} \end{pmatrix} \begin{pmatrix} \vec{a} \cdot \vec{x} \\ \vec{b} \cdot \vec{x} \end{pmatrix} \end{aligned}$$

and we check $\alpha' + \beta' \leq \vec{a}^2\vec{b}^2 - (\vec{a} \cdot \vec{b})^2$.

So how do we use this?

When reading we are given the three vertices $\vec{v}_0, \vec{v}_1, \vec{v}_2$ of a triangle. We select one of them to be the origin for the calculations, say $\vec{v}_0$. We then let $\vec{a} = \vec{v}_1 - \vec{v}_0$ and $\vec{b} = \vec{v}_2 - \vec{v}_0$.

We easily find the normal to the triangle by taking the cross product $\vec{a} \times \vec{b}$ and normalise, and this with $\vec{v}_0$ gives us a plane. Also at this time we calculate $\vec{a}^2, \vec{a} \cdot \vec{b}, \vec{b}^2$ and $\vec{a}^2\vec{b}^2 - (\vec{a} \cdot \vec{b})^2$.

Now when intersecting ray $\vec{o} + t\vec{d}$ we just do the plane intersection calculation to find t, work out $\vec{x} - \vec{v}_0$ and then by the matrix above α' and β' and test against 0 and $\vec{a}^2\vec{b}^2 - (\vec{a} \cdot \vec{b})^2$.

A triangle is specified by its three vertices. The following is a triangle with vertices at $(1, 0, 0)$,$(0, 1, 0)$ and $(0, 0, 1)$:

```
1 0 0
0 1 0
0 0 1
```

2.6 The Parallelogram Primitive

A parallelogram with one vertex at the origin and the adjacent two at $\vec{a}, \vec{b}$ can be thought of as the set of points:

$$\{\alpha\vec{a} + \beta\vec{b} : 0 \leq \alpha, \beta \geq 1\}.$$

So, if given a point we with to determine if α, β satisfy the inequalities given.

We may determine α and β (or even α', β') in the same way as for the triangle and adapt the inequalities.

A parallelogram is specified by a vertex, and the directions of the two edges leaving that vertex. The following is a square with vertices at $(1, 1, 0)$, $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 0)$:

```
1 1 0
0 -1 0
0 0 -1
```

2.7 The Affine Primitive

Suppose you wish to specify a primitive in another co-ordinate system, say one whose origin is at $\vec{O}$ and whose x, y and z axes appear as $\vec{X}, \vec{Y}, \vec{Z}$ in our co-ordinate system. Let:

$$M = \begin{pmatrix} X_1 & Y_1 & Z_1 \\ X_2 & Y_2 & Z_2 \\ X_3 & Y_3 & Z_3 \end{pmatrix}^{-1} = \begin{pmatrix} \vec{X} & \vec{Y} & \vec{Z} \\ & & \end{pmatrix}^{-1}.$$

Then a ray $\vec{o} + t\vec{d}$ in our old basis will be a ray in out new basis:

$$M(\vec{o} - \vec{O}) + t' \frac{M\vec{d}}{|M\vec{d}|}$$

where $t' = t|M\vec{d}|$. This ray can now be fed to the primitive intersection code of the of the primitive in the new basis giving t' and hence t.

Next consider the normal. What the normal calculation code for any primitive allows us to calculate is:

$$\vec{\nabla}_{\vec{y}} f(\vec{y})\Big|_{\vec{y}_0}.$$

What we want to evaluate is:

$$\vec{\nabla}_{\vec{x}} f(M\vec{x})\Big|_{\vec{x}_0}.$$

By the chain rule we can evaluate this in terms of what we have:

$$\vec{\nabla}_{\vec{x}} f(M\vec{x})\Big|_{\vec{x}_0} = M^T \, \vec{\nabla}_{\vec{y}} f(\vec{y})\Big|_{M\vec{x}_0}$$

Or we may see it directly.

$$\begin{aligned} & \frac{\partial}{\partial \_i} f(M\vec{x}) \\ &= \frac{\partial}{\partial \_i} f(M_{11}x_1 + M_{12}x_2 + M13x_3, M_{21}x_1 + M_{22}x_2 + M23x_3, M_{31}x_1 + M_{32}x_2 + M33x_3) \\ &= \frac{\partial f(\vec{y})}{\partial y_1} M_{1i} + \frac{\partial f(\vec{y})}{\partial y_2} M_{2i} + \frac{\partial f(\vec{y})}{\partial y_3} M_{3i} \end{aligned}$$

Curiously this means if we are given M instead of M^{-1} we do not need the matrix to be invertible. This corresponds to our space appearing flat in the world the primitive lives in!

The implementation of this special primitive is quite straight forward. Multiplication of the vectors in the ray by a matrix and its transpose is standard. The only catch is reading another primitive - which is achieved by reading an object ID, searching the object type table, and then calling the appropriate function.

An affine primitive is specified as $\vec{O}$, then $\vec{X}, \vec{Y}, \vec{Z}$ and then the object ID and description the the embedded primitive.

In the below example we have an affine transformation which stretches a sphere to be twice the length in the y and z directions it normally would. This produces an ellipsoid. [6]

```
        0 0 0
        1 0 0
        0 2 0
        0 0 2
SPHERE_ID
        0.5
        0 0.5 0
```

Similar primitives are provided for the affine distortion of surfaces and lights. However, lights with the $1/r^2$ effect aren't really suited to arbitary affine distortion, as the distance function will depend on direction for a general affine transdorm. If the matrix given just scales distance then all should be well.

2.8 The Cone Primitive

A cone with apex at $\vec{c}$ and axis $\vec{a}$ is the set of all points $\vec{x}$ for which the line from the apex to the point is at some given angle θ to the axis. Normalising $\vec{a}$ this can be written in vector terms:

$$\frac{(\vec{x}-\vec{c})\cdot\vec{a}}{|\vec{x}-\vec{c}|} = \cos(\theta).$$

Let $\alpha = \cos(\theta)$. We can extend this to the double cone:

$$(\vec{x}-\vec{c})\cdot\vec{a} = \pm\alpha|\vec{x}-\vec{c}|,$$

or

$$((\vec{x}-\vec{c})\cdot\vec{a})^2 = \alpha^2(\vec{x}-\vec{c})^2$$

Filling in our ray $\vec{o}+t\vec{d}$:

$$\begin{aligned}\left((\vec{o}-\vec{c})\cdot\vec{a}+t\vec{d}\cdot\vec{a}\right)^2 &= \alpha^2\left((\vec{o}-\vec{c})+t\vec{d}\right)^2\\ ((\vec{o}-\vec{c})\cdot\vec{a})^2+2t\left(\vec{d}\cdot\vec{a}\right)((\vec{o}-\vec{c})\cdot\vec{a})+t^2(\vec{d}\cdot\vec{a})^2 &= \alpha^2\left((\vec{o}-\vec{c})^2+2t(\vec{o}-\vec{c})\cdot\vec{d})+t^2\vec{d}^2\right),\end{aligned}$$

giving the quadratic for t:

$$t^2\left(\alpha^2-(\vec{d}\cdot\vec{a})^2\right)+t2\left(\alpha^2(\vec{o}-\vec{c})\cdot\vec{d}-\left(\vec{d}\cdot\vec{a}\right)((\vec{o}-\vec{c})\cdot\vec{a})\right)+\alpha^2-((\vec{o}-\vec{c})\cdot\vec{a})^2=0,$$

which is solved in the normal manner. By calculating $\vec{x}$ afterwards, dotting with $\vec{a}$ and rejecting certain values it is possible to get only finite cones.

Now to find the normal. Let $\vec{x}_0$ be a point on the cone. We may extend the normal at this point back until it hits the axis, at a distance (say) l from the apex. Then $l\cos(\theta) = |\vec{x}_0-\vec{c}|$ is

[6] The `SPHERE_ID` in the description should be replaced by whatever number has been assigned to the sphere primitive.

the distance from the apex to this point. However the normal, the axis and the line from the apex form a triangle, giving:

$$\vec{n} = (\vec{p} - \vec{c}) - l\vec{a} = \vec{p} - \frac{|\vec{x}_0 - \vec{c}|}{\alpha}\vec{a}$$

This still must be normalised, and must be adjusted for the fact that for one half of the double cone the normal points the other way !

The cone is a rather tough case where the only thing that can be precalculated is α^2, and α must also be stored for computing the normal.

A cone is specified by α, then where the cone starts and finishes along the axis (in terms of the length of the axis), then the 3 components of the apex of the cone, and finally the three components of the axis of the cone.

For example the following cone makes an angle of about 25^o with its axis, starts at the apex and goes to 2 times the length of the normal (in this case for a total of -10 units in the y direction). Its apex is at $(0, 2, 0)$ and its axis points along the y direction.

```
0.9
0 2
0 2 0
0 -5 0
```

2.9 The Cylinder Primitive

A cylinder is the set of points a given distance (the radius r) from a line. If $\vec{c}$ is a point on this line, $\vec{a}$ points along the line and $\vec{x}$ is a point on the cylinder then we note by Pythagoras' theorem (providing $\vec{a}$ has been normalised).

$$\begin{aligned}
(\text{distance from } \vec{c} \text{ to } \vec{x})^2 &= (\text{radius})^2 + (\text{distance along axis from } \vec{c} \text{ to } \vec{x})^2 \\
(\vec{x} - \vec{c})^2 &= r^2 + ((\vec{x} - \vec{c}) \cdot \vec{a})^2 \\
\text{Filling in our ray} & \\
(\vec{o} - \vec{c})^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{d} + t^2\vec{d}^2 &= r^2 + ((\vec{o} - \vec{c}) \cdot \vec{a})^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{a}\vec{d} \cdot \vec{a} + t^2(\vec{d} \cdot \vec{a})^2
\end{aligned}$$

Giving a quadratic for t:

$$t^2\left(1 - (\vec{d} \cdot \vec{a})^2\right) + t2\left((\vec{o} - \vec{c}) \cdot \vec{d} - (\vec{o} - \vec{c}) \cdot \vec{a}\vec{d} \cdot \vec{a}\right) + (\vec{o} - \vec{c})^2 - ((\vec{o} - \vec{c}) \cdot \vec{a})^2 - r^2 = 0.$$

This can be solved in the usual manner, and r^2 may be precalculated. Finite cylinder can be formed by then working out $\vec{x}$ and dotting with $\vec{a}$ and then ensuring this lies within a certain range.

The normal at $\vec{x}_0$ is found by examining this triangle further. One side is the normal, and it is easily seen to be:

$$\vec{n} = (\vec{x}_0 - \vec{c}) - ((\vec{x}_0 - \vec{c}) \cdot \vec{a})\vec{a}.$$

A cylinder is specified by its radius, its upper and lower values for the extent of the cylinder (this cylinder starts at height 2 and goes down to height -8). Then you specify a point on the axis and the direction of the axis.

```
0.5
0 10
0 2 0
0 -1 0
```

3 Assessment

Here is presented a list of the strengths and failings of the model, program structure and implementation.

3.1 Models

3.1.1 Ray Tracing

Ray tracing can produce very convincing looking pictures, but they always seem too clean and shiny if realism is your aim. Ray tracing is a very good model when it comes to smooth, non-refracting objects, none of which contribute much to the general lighting of the scene.

Combining Ray Tracing with the Radiosity method can improve the results if you have a strongly lit surface which contributes largely to the lighting of the scene.

The problem with refracting objects is two fold. First refracting objects, for the most realism, should refract different coloured light in different directions. This means sampling the spectrum at many points and tracing rays for all the different resultant rays. This pushes the computation time up significantly.

Secondly, as light is traced from the eye out, you will not see "rainbows" projected onto things by prisms. If you look into a transparent object you will see rainbows around things, but these will not be visible at all on a diffuse surface. This also means you don't see "caustic" effects when light passes through transparent media.

This second can be partially resolved by tracing rays from the light out and merging the results with that of a traditional ray trace.

3.2 Phong Illumination Model

The Phong Illumination model is a model not based on the physics of the situation, but rather a model which produces reasonable results in reasonable time. As it stands it can be commended, but some simple improvements can be made.

Also the model assumes all lights are "at infinity". This ray tracer goes some way to fix this by adding the inverse square law when evaluating the Phong illumination. A further improvement may be to introduce the inverse square law fully, that is when adding up the illumination from a traced ray to divide the specular part by the square of the total distance travelled since the light left a light source, or a diffuse surface (where the beam becomes unfocused). There is some ambiguity here though which needs to be addressed.

A third improvement would be to allow for objects which absorb light as it passes through them. The standard physical model says light intensity will decay exponentially, with respect to the distance travelled in the object. It is interesting to note that this can be represented by giving objects complex refractive indices.

3.3 The Program

3.3.1 Program Structure

The program was designed with the prospect of adding new primitives in mind. As a result adding a new primitive consists of writing read, intersect and normal code in the same style as the others, and then adding your new object to the object table. This is very straight forward.

In addition to ray tracing several rendering methods have been written. These include shading according to depth (gray level = r^{-1}), shading by pure colour (colour is that of first point hit) and shading by direction of normal (colour is given by dot product of normal and eye ray). None of these methods are very pretty, but might they might help in the debugging of new objects or surfaces.

If each object was given an "in" function which returned if a point was considered to be within the object then it would be possible to implement object intersections. In a similar manner it would be possible to implement a bounding volume or voxel system to improve the programs speed.

The program was not originally designed with expansion of the light types, or surfaces in mind. Both the surface and light code has now been modularised, in the same style as the primitive code.

The surface objects seem quite satisfactory. It is not yet clear if the light objects are sufficiently flexible.

3.3.2 Program Implementation

The program is written in ANSI C, and has been tested with on the following platforms.

Complier	**OS**	**Machine**
gcc 2.7.2	FreeBSD 2.2.7	Pentium PC
egcs 1.1	FreeBSD 2.2.7	Pentium PC
TenDRA-4.1.2	FreeBSD 2.2.7	Pentium PC
gcc 2.7.0	SunOS 4.1.3 U1	SunSparc 5
cc, gcc	RISC/OS 5.01	Mips 4000
cc, gcc	RISC/OS 4.51	Mips 3000
gcc	OSF/1	DEC Alpha
gcc 2.6.3, sc 6.55	AmigaDos 3.1	Amiga 4000
gcc	Linux	Pentium PC

The only compiler errors it has produced (with options `-Wtraditional -W -Wpointer-arith -Wall -Wstrict-prototypes -pedantic -Wmissing-prototypes -Winline -Wmissing-declarations -ansi -Wredundant-decls` for `gcc`) are relating to broken system header files. I would recommend the `comp.lang.c` FAQ to anyone interested in writing good C.

The speed of the implementation is quite good, all the test scenes take in the region of minutes or less to render on all the above machines at a 512 by 512 resolution. A scene of the space shuttle consisting of 1600 triangles takes in the region of half an hour.

It could be improved by making further use of `typedef`, currently some structures are used without defining new types. Some of the code could be better commented.

Memory management is a bit weak[7]. All memory allocated forms a linked list, which can be fully un-allocated. Deallocating part of it is not possible at the moment. This is also linked with the weak error handling. If you hit a problem you call `die` which unallocates all memory and exits.

The code is written as many fairly small bits, for the sake of expand-ability but some of the header files make information available to the rest of the source code which should not be (for example primitive structures can be seen — which should not be the case). With the addition of modular surface code this should be more straight forward.

The object type array should be automatically sorted, and then binary searched - it is currently unsorted and searches are linear. General investigation of compiler optimisation of the code, and its effects on run time should be done. A compiler which can inline floating point square root code speeds up rendering of many scenes significantly.

3.3.3 Preprocessor

The problem here is that there isn't one. I've considered writing some macros for `cpp` (the C pre-processor) or `m4` to make the input language more readable, but the what is really needed is a separate program to preprocess the input. Something written in Perl could be quite suitable. Ideally a proper parser built with yacc/bison/flex/lex should be written.

The preprocessor, a fancy front end and the ray tracer could be merged, however I'd be in favor of keeping these separate.

4 Summary & Conclusion

Ray tracing works. It produces shiny pictures which people go "woooow" at. The rather weird perfection makes their look unreal. Implementing a ray tracer is fiddley in later stages, as it becomes difficult to work out how light should behave in a test scene.

[7] For "a bit weak" read "almost nonexistent".

4.1 Features

- Primitives: Annulus Cone Cylinder Disk Parallelogram Plane Sphere Triangle Torus Polygon
- Special Primitive which allows affine transformation of other primitives.
- Point light sources with and without r^{-2}.
- Ambient light source.
- A simple camera model with no depth of field.
- Phong Illumination model, with simple shadowing and simple inverse square law.
- Surface code for various different types of surface, including texture mapping.
- Recursive reflection and refraction and total internal reflection.
- A simple input language.
- Output in `PPM` format.
- Should be portable to all ANSI C platforms.
- Code for solving polynomials of degree less than 5.
- The ability to turn off parts of the illutation model for testing.
- Gnu `autoconf` used for finding system header files.
- Depth, colour and normal rendering.

4.2 Wish List

- Primitives: Prism, Pyramid - maybe Helix, quadratic forms, paraboloid.
- Special Primitives to allow CSG operations.
- Special Primitives to allow bounding volumes.
- Phong Illumination model, with absorption and proper inverse square law.
- Refraction with complex refractive indices.
- Complex input language and Preprocessor.
- Intersection code that takes advantage of the "best intersection yet".