An Introduction to Ray Tracing[1]

Roman Kuchkuda
University of North Caroline
roman@cs.unc.edu

Abstract

This paper is a practical guide to ray tracing for those familiar with graphics. It consists of a conceptual model of ray tracing, C code for a basic system, and an explanation of how and why the code works.

Key Words & Phrases
Ray tracing, C code, Conceptual Model, Spheres, Boxes, Triangles, Superquadrics, Reflection, Shadows, Transparency, LEX and YACC.

1 INTRODUCTION

Ray tracing is one of the most popular algorithms for rendering high quality computer graphics images. It is conceptually simple; rendering is reduced to finding the intersection of a line with an object and the shading the point of intersection of a line with an object and the shading the point of intersection. It is powerful; almost any type of object can be rendered. It is extensible; adding a variety of new effects is easy and natural.

For all its conceptual simplicity, it can be a difficult subject to learn. The classic computer graphics text only mention it. Current technical journals contain article about advanced features and special extensions. Article on the basics were printed many years ago and can be dificult to find.

This paper attempts to fill the need for a begginer's guide to ray tracing. It consists of three parts:

- Part1 present a conceptual model of ray tracing and discusses the translation of the model into code.
- Part 2 presents C code for a basic ray tracing system. Each section of code is accompanied by an explanation of how and why it works. Figure 1 shows the image generated by the basic system, using the default image parameters.
- Part 3 presents extensions to the basic system. Figures 2-4 show images generated by the full system.

All of the code necessary to build a simple ray tracer is included. Figures 1-4 were produced using code which was filtered from the text of this document. Only the standard features of C are used. In some cases, execution speed has been sacrificed in the interest of clarity.

1.1 Cautions

The chief drawback of the ray tracing is that it is slow. Complicated images take HOURS of CPU time. Be kind to your fellow users; run your ray tracer in the background at low priority.

Image files are large. Keeping many images on disk will quickly use a lot of space.

Finally, beware -- ray tracing is addictive. The beauty os your images will entrance you. There is always one more effect that can be added to produce a new and more spectacular picture.

1.2 Conceptual model

The fundamental idea behind ray tracing is to follow rays of light from a light source. Their paths and intensity/color are computed as they reflect off and refract through objects.

Imagine a camera on a tripod sitting in a room filled with a variety of objects. The room is lit by a single light bulb. Rays of light from the light bulb reflect off objects. Some of the light rays travel through the cameras lens and hit the film, creating an image. This is the process which ray tracing attempts to simulate.

Everything in the room can be represented mathematically. The light bulb has an x,y,z position and a brightness. The objects can be modeled as geometric primitives, such as spheres or polygons, or as collections of primitives. The camera can be described by its position, orientation, and the field of view of the lens.

In the program, lights rays are send out from the light bulb in all directions. The angles of reflection and refraction of the light rays are computed as they bounce off objects. The surface properties (color, shiness, etc.) of the objects change the color and intensity of the rays. Some of the rays eventually strike the fill. The color and intensity of those that do are recorded.

[1] This article was published in *Theoretical Foundations of Computer Graphics and CAD*, edited by R. A. Earnshaw, Springer-Verlag, 1988, 1039-1060.

To produce a clear images, many rays must be strike the fill. Because only a small percentage reach the fill, a tremendous number of rays must be traced. It is incredibly expensive, but the images should be nearly perfect if the model of light interaction is good.

The key to practical ray tracing is a slight adjustment to this process. Rather than trace rays from the light source, with most of the rays missing the film, rays are traced from points on the film out of the camera into the room. One ray is traced for each spot on the film, with the color of the object hit recorded as the color for that spot. On the graphics display, each pixel correspond to a spot on the film.

1.3 History

The earliest reference in computer graphics to ray tracing is [Appel68]. [Goldstein71] described image production software at MAGI. [Whitted80] extended ray tracing by adding reflection and refraction. SInce this early work, ray tracing has benefitted from the efforts of many researchers. Some areas of development are described in Section 3.9.

2 FIRST PROGRAM

The first ray tracing program is designed to allow you to quickly produce some simple images. It also provides a solid base for further development.

A ray tracing system includes routines for many types of objects primitives. The first program include only code for spheres. In the full system, the code for boxes, triangles, and superquadrics is presented. As you develop your ray tracer, you can add more primitives.

The shading model in the first program is fairly sophisticated. It includes ambient, diffuse, and specular lighting. Several point light sources are allowed. In the full system shadows, reflections and transparency will be added.

In a flexible system many values are specified by the user at run time. The code to obtain user input is fairly long. So, the values are hard coded in the basic system. In Section 3.6, user input routines are presented. For now, run the program and see what the images looks like. Try changing some of the hard code values and see how the images change.

2.1 Main Routine

The main routine determines a color value for each pixel in the image and writes this value to the pixel output file.

Initializations are performed by a call to **setup**. **viewing** is called to calculate the direction and spacing of the rays to be traced. The pixel output file is opened and the image size written to it in **startpic**.

The actual ray trace is the performed. The rays are generated, in order, from left to right, top to bottom, by two nested loops. The outer loop is 1 to n, where n is the number of lines in the image. The inned loop is 1 to m, where m is the numbers of pixels across each line. Inside the inner loop, a ray is send into the world by a call to **intersect**.

intersect computes the intersections of a ray with all objects. It determines the point of closest intersection and returns the color of the object at that point. This color value is stored. If the ray doesn't hit any object, the background color is stored.

When the colors of all of the pixels across a line have been determined, the line is output through a call to **linepic**. Every tenth line a status message is sent to stdout. When all of the lines in the image have been completed, **endpic** closes the output file.

```
/***  main.c ***/
#include <stdio.h>
#include "typedefs.h"
#include "maindecl.h"

main()
{
  int line_y, pixel_x;
  t_3d scrnx,scrny,firstray,ray;
  t_color color;
  double dis,line[SCRENNWIDTH][3];

  setup();
  viewing(&scrnx,&scrny,&firstray);
  startpic(outfilename,sizey,sizex);
  for(line_y=0;line_y<sizey;line_y++)
  {
    for(pixel_x=0;pixel_x<sizex;pixel_x++)
    {
      ray.x = firstray.x + pixel_x*scrnx.x
              - line_y*scrny.x;
      ray.y = firstray.y + pixel_x*scrnx.y
              - line_y*scrny.y;
      ray.z = firstray.z + pixel_x*scrnx.z
              - line_y*scrny.z;
      normalize(&ray);

      /* actual ray trace */
      dis = intersect(-1,&eyep,&ray,&color);
      if( dis > 0 )
      {      /* ray intersected object */
        line[pixel_x][0] = color.r;
        line[pixel_x][1] = color.g;
        line[pixel_x][2] = color.b;
      }
      else
      {      /* use background color */
        line[pixel_x][0] = background.r;
        line[pixel_x][1] = background.g;
        line[pixel_x][2] = background.b;
      }
    }
    /* Output line of pixels */
    linepic(line);
    if (line_y % 10 == 0)
    {
      printf( "\nDone line %d", line_y );
      fflush( stdout );
    }
  }
  endpic();
}
```

2.2 Include Files

Include files define structure types, global variables, constants, and functions types. Their use simplifies later modification of the program.

The first include file, **typedefs.h**, defines the structures for the basic ray tracer. Structures allow conceptual grouping of data and simplify information passing between routines. New attributes can easily be added to structures.

Spheres, boxes, triangles, and superquadrics are defined. Each object is decribed in two structures. The first structure has the fields which are common to all objects. The second has the fields that are unique to a given object.

o_sphere describes spheres. It has fields for the sphere's radius and its x,y,z center position.

The box object is defined by the **o_box** structure. It has x,y,z center position, sizes for the x, y, and z sides and sidehit. Sidehit is used in the computation of the surface normal of the box.

Triangles are described by the **o_triangle** structure. It include the triangle's plane description, three edge vectors, and three edge constants.

The superquadric object structure, **o_superq**, has all of the values of the box structure, plus some new values. Thes values are described in Section 3.5.

The **t_object** structure contains the fields which are common to all objects. It has object id, which is unique for each object, surface type number, which specifies the color of the surface, object type, and object pointer, which points to one of the object type structures defined above.

The **t_light** type defines a point light source. It has an x,y,z position in space and a brightness value.

The **t_surface** structure describes the properties of an object's surfaces. The color is specified by red, green, and blue components for each of ambient, diffuse, and specular lighting. The specular coefficient controls the size and intensity of the specular highlight. It is a positive integer and the higher the coeficient, the smaller and brighter the specular highlight. The reflectivity of the surface is specified by a number from zero to one, with zero being no reflection, one being a perfect mirror. Transparency is specified in a similar manner, with zero being opaque, one completely transparent. Reflections and transparency are not included in the basic program; they are added later.

Three-dimensional vectors are represented by the **t_3d** structure. It is used throughout the program for both positions and vectors in 3 space.

Lastly, **t_color** structure specifies a red, green, blue color trio. Color values are passed throughout the program in this form.

```
/*** typedefs.h ***/
typedef struct
{
  double x,y,z;
} t_3d;

typedef struct
{
  double r;          /* radius */
  double x,y,z; /* position    */
} o_sphere;

typedef struct
{
  double sidehit;/* side intersected */
  double xs,ys,zs; /* size of sides  */
  double x,y,z;  /* center position */
} o_box;

typedef struct
{
  t_3d  nrm;         /* triangle normal */
  double d;           /* plane constant */
  t_3d  e1,e2,e3;    /* edge vectors  */
  double d1,d2,d3;/*plane constants  */
} o_triangle;

typedef struct
{
  int sidehit; /* side intersected   */
  double xs,ys,zs;    /* size of sides */
  double x,y,z; /* center position */
  double pow;      /* n in formula */
  double a,b,c,r;  /* coefficients */
  double err;     /* error measure */
} o_superq;

typedef struct
{
  int id;       /* object number     */
  int objtyp;     /* object type */
  int surfnum; /* surface number */
  union
  {
    o_sphere   *p_sphere;
    o_box      *p_box;
    o_triangle *p_triangle;
    o_superq   *p_superq;
  } objpnt;    /* object pointer */
} t_object;

typedef struct
{
  double x,y,z;  /* position  */
  double bright;
} t_light;

typedef struct
{
  double ar,ag,ab; /* ambient r,g,b  */
  double dr,dg,db; /* diffuse r,g,b  */
  double sr,sg,sb;/* specular r,g,b  */
  double coef;     /* specular coef */
  double refl;    /* reflection 0-1 */
  double transp;/* transparency 0-1   */
} t_surface;
```

```
typedef struct
{
  double r,g,b; /* red,green,blue    */
} t_color;
```

constants.h contains a variety of constants definitions. First, the sizes of the light, object, and surface type arrays are defined. The array sizes must be increased as you render images with more objects (or lights or surfaces types) in them. Only the mais program should required recompilation when these sizes are changed.

Next the screen size is defined. It should not change unless you move to a frame buffer with a different resolution or aspect ratio. The gamma correction constant is likewise frame buffer dependent.

Finally, object type constants are specified. These constants relate the object type to the routines used to find intersections and normals. The constants (0-n) correspond to the order of the intersection and normal routines in the **objint** and **objnrm** arrays in **maindecl.h**.

```
/*** constants.h ***/
#define LIGHTS            4
#define OBJECTS          50
#define SURFACES         50
#define SCREENWIDTH       512
#define SCREENHEIGHT    512
#define ASPECTRATIO       1.0
#define GAMMA             1.8
#define OTYPSPHERE        0
#define OTYPBOX           1
#define OTYPTRIANGLE      2
#define OTYPSUPERQ        3
```

maindecl.h contains the *declaration* of the global variables and is included in the main program.

Global information has been minimized. This tends to lessen unforseen interactions among routines. The global variables consist mainly of values which define the ray tracing world.

First, the object intersection routine and object normal routine pointer arrays are declared. The **objint** array has pointers to the ray-object intersection routine for each type of object. The **objnrm** array has pointers to the surface normal routine for each object type.

Next, the arrays for objects, lights, and surfaces types are declared. A maximum size for each array is kept, along with a count of how many itens in each array are currently being used.

The viewing parameters are declared next. They include the size of the image to be produced, the eye point, look point, up vector, and horizontal and vertical fields of view. Section 2.5 describes these parameters.

The current level and maximum level of reflection are used in the full system to limit the number of levels of reflection which are rendered.

The string for the pixel output file name is then declared. Finally, the background color is defined. It is used when a ray does not hit any object.

```
/*** maindecl.h ***/
#include "constants.h"
#include "funcdefs.h"

/* intersection routines */
double (*objint[])()={intsph,intbox,
                      inttri,intsup};
/* normal routines */
int (*objnrm[])()={nrmsph,nrmbox,
                   nrmtri,nrmsup};
/* global variables */
int nlight;                /* presently in use */
int lightlim=LIGHTS;       /* maximum declared */
t_light light[LIGHTS];      /* array of lights */
int nobject;               /* presently in use */
int objectlim=OBJECTS;     /* maximun declared */
t_object object[OBJECTS]; /* array of objects */
int nsurface;              /* presently in use */
int surfacelim=SURFACES;  /* maximum declared */
t_surface surface[SURFACES];  /* array surfaces */
int sizex,sizey;                /* image sizes */
t_3d eyep,lookp,up;         /* view definition */
double hfov,vfov;             /* field of view */
int level,maxlevel;       /* reflection levels */
char *outfilename[];        /* pixel file name */
t_color background;        /* background color */
```

globalvar.h contains the *definition* of the global variables and is included in routines which access the global variables.

```
/*** globalvar.h ***/
extern double     (*objint[])();
extern int        (*objnrm[])();
extern int        nlight,lightlim;
extern t_light    light[];
extern int        nobject,objectlim;
extern t_object   object[];
extern int        nsurface,surfacelim;
extern t_surface  surface[];
extern int        sizex,sizey;
extern t_3d       eyep,lookp,up;
extern double     hfov,vfov;
extern int        level,maxlevel;
extern char       *outfilename[];
extern t_color    background;
```

funcdefs.h defines the names and types for all of the routines in the ray tracer. It is included in all routines which have calls to other routines.

```
/*** funcdefs ***/
double brightness();
int    crossp();
double dotp();
int    endpic();
int    gammacorrect();
double intersect();
int    lightray();
int    linepic();
double normalize();
int    setup();
int    shade();
int    startpic();
int    viewing();
int    yyparse();
int    maksph();
double intsph();
int    nrmsph();
int    makbox();
double intbox();
int    nrmbox();
int    maktri();
double inttri();
int    nrmtri();
int    maksup();
double intsup();
int    nrmsup();
```

2.3 Setup Routine

setup initializes global variables and sets defaults for user specified values. **yyparse** is called to get user input.

```
/*** setup.c ***/
#include "typedefs.h"
#include "globalvar.h"

int setup()
{
  /* set defaults */
  nlight = 0;
  nobject = 0;
  nsurface = 0;
  level = 0;
  sizex = 512;
  sizey = 512;
  hfov  = 50;
  vfov = 50;
  eyep.x = 100.0;
  eyep.y = 0.0;
  eyep.z = 0.0;
  lookp.x = 0.0;
  lookp.y = 0.0;
  lookp.z = 0.0;
  up.x  = 0.0;
  up.y = 1.0;
  up.z = 0.0;
  strcpy (outfilename, "raytrace.pix" );
  yyparse (); /* parse user input */
}
```

2.4 First Program Stubs

The basic system contains stubs for routines which will be defined in the full system. **yyparse** and the intersection normal routines for boxes, triangles, and superquadrics are stubbed.

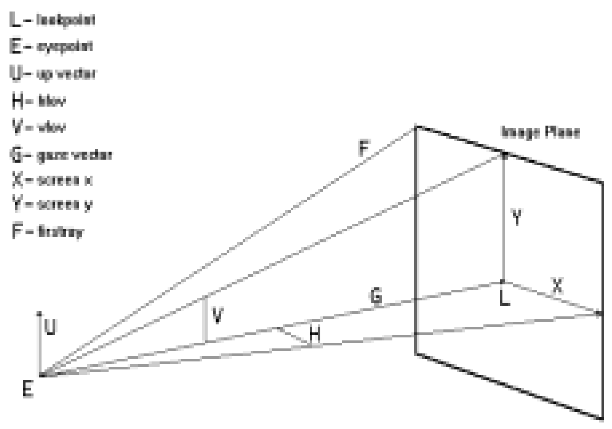
The intersection and normal routines are empty, returning zero when called. There is no user input in the basic system; **yyparse** consist of a series of hard coded assignment statments.

```
/*** stubs.c ***/
#include "typedefs.h"
#include "globalvar.h"

/* intersection and normal routines */
int makbox(){return;}
int maktri(){return;}
int maksup(){return;}
double intbox(){return(0.0);}
double inttri(){return(0.0);}
double intsup(){return(0.0);}
int nrmbox(){return;}
int nrmtri(){return;}
int nrmsup(){return;}

int yyparse()
{
  nlight = 1;         /* light source */
  light[0].x = 100;
  light[0].y = 300;
  light[0].z = 0;
  light[0].bright = 1.0;
  /* sphere - surface type 0, radius 40 */
  /* center at 5,30,40                  */
  maksph( 0,40.0,5.0,30.0,40.0 );
  nobject++;
  /* sphere - surface type 1, radius 30 */
  /* center at -10,-10,-40              */
  maksph( 1,30.0,-10.0,-10.0,-40.0 );
  nobject++;
  nsurface = 2;
  surface[0].ar = 30; /* surface 0 - red shiny */
  surface[0].ag = 0;
  surface[0].ab = 0;
  surface[0].dr = 90;
  surface[0].dg = 0;
  surface[0].db = 0;
  surface[0].sr = 190;
  surface[0].sg = 120;
  surface[0].sb = 120;
  surface[0].coef = 15;
  surface[1].ar = 0;/* surface 1- green duller */
  surface[1].ag = 50;
  surface[1].ab = 0;
  surface[1].dr = 0;
  surface[1].dg = 100;
  surface[1].db = 0;
  surface[1].sr = 30;
  surface[1].sg = 40;
  surface[1].sb = 30;
  surface[1].coef = 2;
  background.r = 100;     /* background color */
  background.g = 200;     /* light blue       */
  background.b = 250;
  return(0);
}
```

2.5 Viewing Routine

To produce an image, a ray is sent into the world for each pixel. The **viewing** routine calculates the direction and the spacing of the rays. It uses either the viewing parameters specified by the user or the defaults defined in **setup**.

The viewing definition is complicated. It is described using the camera analogy introduced earlier. The point, vectors, and angles involved are shown in the following diagram.



The eyepoint (E) describes the camera position. The lookpoint (L) describes the camera is pointing. It is the center of the image plane. Together, the eyepoint and the lookpoint define the gaze vector (G). The up vector (U) describes the orientation of the camera about the gaze vector.

Field of view refers to the length of the camera's lens, from wide angle to telephoto. Both horizontal (H) and vertical (V) fields of view are defined. For the usual rectangular display, the horizontal field of view is somewhat larger than the vertical. For a square display, they are equal.

Sfter the view has been defined, several values are derived from it. The screen x (X) and screen y (Y) vectors describe the image plane. They are scaled to the width and height of one pixel on the image plane. Firstray (F) describes the ray from de eyepoint to the pixel in the top left corner of the image plane. Screen x, screen y, and firstray are used in the main program to generate the individual rays for each pixel.

```
/***** viewing.c *****/
#include <math.h>
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"
#define DEGREETORADIAN (3.1415926/180.)


int viewing( scrnx, scrny, firstray )
     t_3d *scrnx, *scrny, *firstray;
{
  int i,j;
  t_3d gaze;
  double dist, magnitude;

  gaze.x = lookp.x - eyep.x;
  gaze.y = lookp.y - eyep.y;
  gaze.z = lookp.z - eyep.z;
  dist = normalize( &gaze );

  /* scrnx = gaze cross up */
  crossp( scrnx, &gaze, &up );
  normalize( scrnx);

  /* scrny = scrnx cross gaze */
  crossp( scrny, scrnx, &gaze );

  dist *= 2.0;
  magnitude = dist * tan( hfov * DEGREETORADIAN )
            / sizex;
  scrnx->x *= magnitude;
  scrnx->y *= magnitude;
  scrnx->z *= magnitude;
  magnitude = dist * tan( vfov * DEGREETORADIAN )
            / sizey;
  scrny->x *= magnitude;
  scrny->y *= magnitude;
  scrny->z *= magnitude;

  firstray->x = lookp.x - eyep.x;
  firstray->y = lookp.y - eyep.y;
  firstray->z = lookp.z - eyep.z;
  firstray->x += sizey / 2 * scrny->x -
                 sizex / 2 * scrnx->x;
  firstray->y += sizey / 2 * scrny->y -
                 sizex / 2 * scrnx->y;
  firstray->z += sizey / 2 * scrny->z -
                 sizex / 2 * scrnx->z;
}
```

2.6 Intersection Routine

intersect finds the closest intersection between a ray and any of the objects in the world. The source of the ray, the ray vector, and the start position are input. The distance to the closest intersection and the color of the object at the point of intersection are returned. If the ray doesn't hit any object, a value of zero is returned.

For each object:

- The intersection routine associated with it is called. The distance to intersection is returned. Zero is returned if the ray misses the object.
- If the intersection is closer than any found previously, the object identity and distance to intersection are saved.

After the closest intersection is found, the point of intersection is computed. Letting Xp,Yp,Zp be the ray origin, Xr,Yr,Zr the ray direction, and s the distance to the intersection: the parametric equations for the ray give the point of intersection.

$$X = Xp + s \cdot Xr$$

$$Y = Yp + s \cdot Yr$$

$$Z = Zp + s \cdot Zr$$

The normal at the point of intersection is found by calling the normal routine for the object hit by the ray. The the shading routine is called to calculate the color at the point of intersection.

```
/***** intersect.c *****/
#include <math.h>
#include "typedefs.h"
```

```
#include "globalvar.h"
#define FAR_AWAY 99.99E+20

double intersect( source, pos, ray, color )
     int source;
     t_3d *pos, *ray;
     t_color *color;
{
  int objhit, objtry;
  double s, ss;
  t_3d hit, normal;

  objhit = -1;
  ss = FAR_AWAY;

  /* check ray intersection with all objects */
  for( objtry=0; objtry<nobject; objtry++ )
  {

    /* special check used for reflections */
    if( objtry != source ) /* don't try source */
    {
       s = (*objint[ object[ objtry ].objtyp ] )
                   ( pos, ray, &object[objtry] );
       /* keep track of closest intersection */
       if( ( s > 0.0 ) && ( s <= ss ) ){
          objhit = objtry;
          ss = s;
       }
    }
  }

  if( objhit < 0 )
     return(0); /* ray hit no objects */

  /* find point of intersection */
  hit.x = pos->x + ss * ray->x;
  hit.y = pos->y + ss * ray->y;
  hit.z = pos->z + ss * ray->z;

  /* find normal */
  (*objnrm[ object[ objhit ].objtyp ] )
     ( &hit, &object[objhit], &normal );

  /* find color at point of intersection */
  shade( &hit, ray, &normal, &object[objhit],
         color );
  return(ss);
}
```

2.7 Sphere Routines

The **maksph** routine allocates the space for a sphere description and fills it with the values passed to the routine.

intsph computes the intersection between a ray and a sphere. The source position of the ray, the ray vector, and the object, a sphere, are passed to the routine. The distance to the nearest intersection is returned. A value of zero is returned if no intersection is found.

The equation of a sphere is $X^2 + Y^2 + Z^2 = R^2$.

The equations for the ray in parametric form, where Xp,Yp,Zp is the ray source and Xr,Yr,Zr is the ray vector, are

$$X = Xp + s \cdot Xr$$

$$Y = Yp + s \cdot Yr$$

$$Z = Zp + s \cdot Zr$$

Substituing the parametric form in the sphere equation gives

$$s^2 \cdot Xr^2 + 2 \cdot Xp \cdot Xr + Xp^2 +$$

$$s^2 \cdot Yr^2 + 2 \cdot Yp \cdot Yr + Yp^2 +$$

$$s^2 \cdot Zr^2 + 2 \cdot Zp \cdot Zr + Zp^2 - R^2 = 0$$

This is a quadric of the form $A \cdot s^2 + B \cdot s + C = 0$

$$A = Xr^2 + Yr^2 + Zr^2$$

$$B = 2 \cdot (Xp \cdot Xr + Yp \cdot Yr + Zp \cdot Zr)$$

$$C = Xp^2 + Yp^2 + Zp^2 - R^2$$

Therefore, there are solutions

$$S = \frac{-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}}{2 \cdot A}$$

Several simplifications can be made. Since the ray is normalized with lenght one:

$$A = Xr^2 + Yr^2 + Zr^2 = 1$$

Let $B = 2 \cdot D$ with

$$D = Xp \cdot Xr + Yp \cdot Yr + Zp \cdot Zr$$

This produces

$$S = \frac{-2 \cdot D \pm \sqrt{4 \cdot D^2 - 4 \cdot C}}{2}$$

Which simplifies to

$$S = -D \pm \sqrt{D^2 - C}$$

This form of the equation is solved in the **intsph** routine. Since there are two solutions; the smallest one greater than zero is returned. If no solution is found, zero is returned.

nrmsph finds the surface normal at a point on the sphere's surface. If the point of intersection is P, the sphere center C, and its radius R, then the normal vector is

$$Nx = (Px - Cx) / R$$

$$Ny = (Py - Cy) / R$$

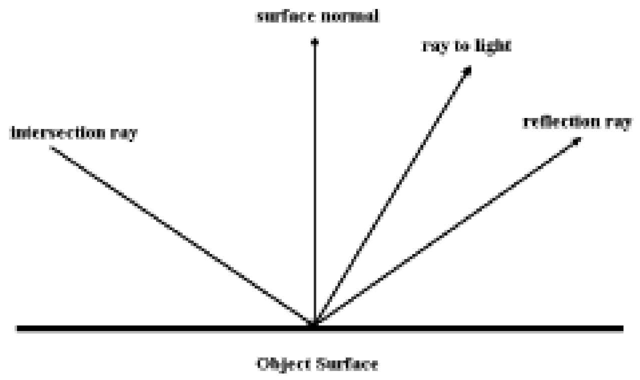
$$Nz = (Pz - Cz) / R$$

```
/***** sphere.c *****/
#include <math.h>
#include "constants.h"
#include "typedefs.h"
#include "globalvar.h"

/* create sphere object */

int maksph( surf, r, x, y, z )
     int surf;
     double r, x, y, z;
{
  int size;
  o_sphere *sphere;
  size = sizeof( o_sphere );
  sphere = ( (o_sphere *) malloc( size ) );
  object[nobject].id = nobject;
  object[nobject].objtyp = OTYPSPHERE;
  object[nobject].surfnum = surf;
  object[nobject].objpnt.p_sphere = sphere;
  sphere->r = r;
  sphere->x = x;
  sphere->y = y;
  sphere->z = z;
}

double intsph( pos, ray, obj )
     t_3d *pos;          /* origin of ray */
     t_3d *ray;          /* ray vector */
     t_object *obj;      /* sphere description */
{
  double b, t, s;
  double xadj, yadj, zadj;
  o_sphere *sph;

  sph = obj->objpnt.p_sphere;

  /* translate ray origin to object's space */
  xadj = pos->x - sph->x;
  yadj = pos->y - sph->y;
  zadj = pos->z - sph->z;

  /* solve quadratic equation */
  b = xadj*ray->x + yadj*ray->y + zadj*ray->z;
  t = b*b - xadj*xadj - yadj*yadj - zadj*zadj +
         sph->r*sph->r;
  if( t < 0 ) return( 0.0 );
  s = -b - sqrt( t ); /* try smaller solution */
  if( s > 0 ) return( s );
  s = -b + sqrt( t ); /* try larger solution */
  if( s > 0 ) return( s );
  return( 0.0 );  /* both solutions <= zero */
}


int nrmsph( pos, obj, nrm )
     t_3d *pos;       /* point of intersection */
     t_object *obj;   /* sphere description */
     t_3d *nrm;       /* return surface normal */
{
    o_sphere *sph;

    sph = obj->objpnt.p_sphere;
    nrm->x = ( pos->x - sph->x ) / sph->r;
    nrm->y = ( pos->y - sph->y ) / sph->r;
    nrm->z = ( pos->z - sph->z ) / sph->r;
}
```

2.8 Shading Routine

The shading routine calculates the color of a point on an object's surface. It models ambient light as well as diffuse and specular lighting from multiple point light sources. White light is assumed for all light sources.

The point os intersection, surface normal at the point, intersection ray, distance to intersection, and object pointer are passed to the shading routine.



Using the rays shown above and the object's surfaces properties, the color of the point being shaded is determined.

The reflection ray from the surface is calculated from the intersection ray and the surface normal. The angle between the reflection ray and the surface normal is equal to the angle between the surface normal and the intersection ray.

The initial color for the point being shaded is the ambient color.

For each light, the difuse color and the specular highlight color are computed and added to the color of the point being shaded.

- The ray to the light source is calculated bu the **lightray** routine. The brightness of the light is found by call to the **brightness** routine.

- The strength of the difuse lighting is given by dot product of the surface normal and the ray to the light. If it is less than zero, the surface is facing away from the light. Thus, the light does nor contribute to the color of the surface. If it is greater than zero, the surface faces the light. The dot product is multiplied by both the brightness of the light and the diffuse surface color.

- The specular highlight is computed as the dot product of the reflection ray and the ray to the light. The dot product, if greater than zero, is raised to the power of k, the surface specular coefficient. This resul is multiplied by the specular surface color.

At each step, lighting components are added to the color of the surface. The surface becomes brihter as each succeeding term is processed. This matches what you would expect. An object lit only by ambient light is fairly dark and evenly shaded. If the surface is lit from a light source, it is brighter and shows more shape. If it is shiny, bright specular highlights appear.

A more complicated shading model, which includes reflection, transparency and shadows is presented in sections 3.1 and 3.2.

```
/***** shade.c *****/
#include <math.h>
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"


int shade( pos, ray, nrm, obj, color )
     t_3d *pos, *ray, *nrm;
     t_object *obj;
     t_color *color;
{
  int lnum;
  double k,bright,spec,diffuse;
  t_surface *surf;
  t_3d refl,ltray;

  /* ambient light contribuition */
  surf = &surface[obj->surfnum];
  color->r = surf->ar;
  color->g = surf->ag;
  color->b = surf->ab;

  /* calculate reflected ray */
  k = -2.0 * dotp( ray, nrm );
  refl.x = k * nrm->x + ray->x;
  refl.y = k * nrm->y + ray->y;
  refl.z = k * nrm->z + ray->z;

  for( lnum=0; lnum < nlight; lnum++ )
  {
    /* get ray to light */
    lightray( lnum, pos, &ltray );
    diffuse = dotp( nrm, &ltray );
    if( diffuse > 0.0 )
    {
       /* object faces light, add diffuse */
       bright = brightness( obj->id, lnum,
                           pos, &ltray );
      diffuse *= bright;
      color->r += surf->dr * diffuse;
      color->g += surf->dg * diffuse;
      color->b += surf->db * diffuse;

      spec = dotp( &refl, &ltray );
      if( spec > 0.0 )
      {
         /* highlight is here, add specular */
         spec = bright * pow( spec, surf->coef );
         color->r += surf->sr * spec;
         color->g += surf->sg * spec;
         color->b += surf->sb * spec;
       }
    }
  }
}
```

2.9 Light Routine

The lighting model supports multiple point light sources of varyng brightnesses. It is implemented using two routines.

The **lightray** routine takes the light number and the surface point as input. It computes the ray from the surface to the light. This ray is used to determine whether the point on the object is facing towards or away from the light. The ray is normalized (legth = 1.0) and returned.

The **brightness** routine takes as input the object number, the surface point, the light ray, and the light number. The brightness of the light as seen from the object surface is returned. Later, this routine will be extended so that it can determine whether the surface is in shadow.

```
/***** light.c *****/
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"

int lightray( lnum, objpos, lray )
     int lnum;
     t_3d *objpos,*lray;
{
  lray->x = light[lnum].x - objpos->x;
  lray->y = light[lnum].y - objpos->y;
  lray->z = light[lnum].z - objpos->z;
  normalize( lray );
}

double brightness( source, lnum, pos, ray )
     int source, lnum;
     t_3d *pos, *ray;
{
  return( light[lnum].bright );
}
```

2.10 Output Routine

Once the image is generated, it must be stored and displayed. This is the one part of a ray tracing system which will vary greatly from site to site. Each site has different frame buffers and programs to load them.

A very simple version os pixel output is presented here. You may have to modify it to suit your equipament. It consists of four routines. The first, **startpic**, opens a file and writes a header. The header consists of the image height and width, each an integer.

The second routine, **linepic**, outputs one line of pixels per call. Its input is an array of pixel r, g, b values. The pixel values are gamma corrected and converted to integer by the **gammacorrect** routine. The shading routine assumes a linear brightness response. However, frame buffers have nonlinear brightness response. Gamma correction is the process which adjusts brightness values to account for nonlinear monitor response. The corrected pixel values are stored in a buffer. After the entire line of pixels has been processed, the buffer is written to the output file.

The final routine **endpic**, which simply closes the output file. If you are using a more sophisticated output algorithm, such as pixel run length encoding, you may want to print the number of runcodes or other performance information.

```
/***** outputp.c *****/
#include <stdio.h>
```

```
#include <math.h>
#include "constants.h"
#include "funcdefs.h"


int width;
FILE * outfile;
int linesize;

int startpic( fname, y, x )
     char *fname[];
     int y, x;
{
  int header[2];

  outfile = creat( fname, 0666 );
  if( outfile == -1 )
  {
    fprintf( stderr, "\nERROR CREATING FILE\n" );
    fflush( stderr );
    abort();
  }
  width = x;
  linesize = 3*width*sizeof( unsigned char );
  header[0] = y;
  header[1] = x;
  write( outfile, header, 2*sizeof(int) );
}

int linepic( pixels )
    double pixels[SCREENWIDTH][3];
{
  unsigned char buffer[SCREENWIDTH][3];
  int i, r, g, b;
  double dr, dg, db;

  for( i=0; i<width; i++ )
  {
    r = gammacorrect( pixels[i][0] );
    g = gammacorrect( pixels[i][1] );
    b = gammacorrect( pixels[i][2] );
    buffer[i][0] = r;
    buffer[i][1] = g;
    buffer[i][2] = b;
  }
  write(outfile, buffer, linesize);
}

int endpic()
{
  close( outfile );
}

int gammacorrect( intensity )
     double intensity;
{
  int ival;
  double dval;

  /* scale to 0-1 range */
  dval = intensity / 255.0;
  if( dval > 1.0 ) dval = 1.0;
  if( dval < 0.0 ) dval = 0.0;

  /* do gamma correction */
  dval = exp( log( dval ) / GAMMA );

  /* convert to integer, range 0-255 */
  dval *= 255.0;
  ival = (int) ( dval + 0.5 );
  return( ival );
}
```

2.11 Ray Functions

Several functions operating on rays have been used throughout the basic system.

The **normalize** routine scales a ray to have length one. The ray's original length is also returned. The **dotp** routine returns the dot product os two rays. The **crossp** routine returns the cross product os the two rays.

```
/***** raymath.c *****/
#include <math.h>
#include "typedefs.h"

double normalize( a )
     t_3d *a;
{
  double d;

  d = sqrt( a->x*a->x + a->y*a->y + a->z*a->z );
  a->x /= d;
  a->y /= d;
  a->z /= d;
  return( d );
}

double dotp( a, b )
     t_3d *a, *b;
{
  double d;

  d = (a->x*b->x) + (a->y*b->y) + (a->z*b->z);
  return( d );
}

int crossp( o, a, b )
     t_3d *o, *a, *b;
{
  double d;

  o->x = ( a->y * b->z ) - ( a->z * b->y );
  o->y = ( a->z * b->x ) - ( a->x * b->z );
  o->z = ( a->x * b->y ) - ( a->y * b->x );
  d = sqrt( o->x*o->x + o->y*o->y + o->z*o->z );
  o->x /= d;
  o->y /= d;
  o->z /= d;
}
```

3 SYSTEM DEVELOPMENT

By now you should have produced several ray traced images. It's time to extend the capabilities of the first program into a full scale system.

The **shading** routine was adequate; now it is extended to produce shadows, reflections, and transparency. Shadow generation can be added by itself. It is easiest to add reflections and transparency together because the **shade** routine is modified for each of them.

Spheres are ok of test images. For real images, many differents primitives are useful. Each new primitive can be added by itself. For each primitive, four things are needed: a 'make' routine, and intersection routine, a normal routine, and a structure definition. Routines and structures to handle boxes, triangles, and superquadrics are presented.

3.1 Shadows

Shadows are an extension to the basic shading routine. They give very trong depth cues and add to the realism of images. In discussing shadows, it is important to differentiate between two terms. Shading is the process of finding the color at a point on an object's surface. Shadowing refers to blocking the light which would have fallen on the object's surface.

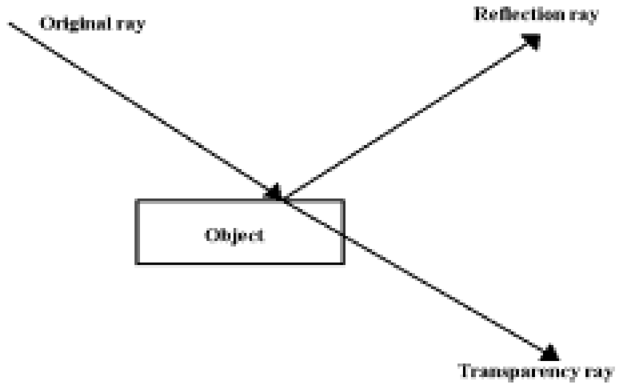
Two routines are used to determine whether the point being shaded is in shadow. **lightray** calculates a shadow ray between the point and the light source, saving the distance between the two.

brightness returns zero if the point is in shadow. Othewise, it returns the light's brightness value. **brightness** is very similar to **intersect**, calculating intersections between the shadow ray and all objects. It stops as soon as it finds one object between the point being shaded and the light source.

```
/***** light2.c *****/
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"

double s_litdis;

int lightray( lnum, objpos, lray )
     int lnum;
     t_3d *objpos, * lray;
{
  lray->x = light[lnum].x - objpos->x;
  lray->y = light[lnum].y - objpos->y;
  lray->z = light[lnum].z - objpos->z;
  s_litdis = normalize( lray );
}

double brightness( source, lnum, pos, ray )
     int source, lnum;
     t_3d *pos, *ray;
{
  int objtry;
  double s;

  for( objtry=0; objtry < nobject; objtry++ )
  {
    if( objtry != source ) /* don't try source */
    {
      s = (*objint[ object[ objtry ].objtyp ])
                  (pos, ray, &object[objtry] );
      if( ( s > 0.0 ) && ( s <= s_litdis ) )
         return( 0.0 );  /* object in sahdow */
    }
  }
  /* object not in shadow */
  return( light[lnum].bright );
}
```

3.2 Reflection & Transparency

Reflection and transparency are extensions to the shading routine. Each adds a new component to the color of the point being shaded.



A reflection ray is sent from the point being shaded by a call to **intersect**. **intersect** finds the closest intersection with another object. Then it calls **shade** to determine the color at the point of intersection and returns the color. This color is combined with the color already calculated for the point being shaded.

$$Color_p = Color_p + R \cdot Color_r$$

where

Color$_p$ is the color already calculated for the point.

R is the reflectivity of the surface being shaded, which ranges from 0.0 for nonreflecting to 1.0 for a perfect mirror.

Color$_r$ is the color of the object hit by the reflection ray. The background color is used if the reflection ray didn't hit any object.

Adding transparency is similar to adding reflection. THe transparency ray is sent from the point being shaded by a call to **intersect**. The color returned is mixed with the color already calculated for the point being shaded.

$$Color_p = (1-T) \cdot Color_p + T \cdot Color_t$$

where

Color$_p$ is the color already calculated for the point, including reflection.

T is the transparency of the surface being shaded, which ranges from 0.0 for opaque to 1.0 for completely transparent.

Color$_t$ is the color of the object hit by the transparency ray. The background color is used if the transparency ray didn't hit any object.

It is important to understand that shading is now defined recursibely. **shade** calls **intersect** which calls **shade** .... A count is used to limit the depth of the recursive calls.

The net result is that reflections can be seen in reflections.

```
/***** shade2.c *****/
```

```
#include <math.h>
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"


int shade( pos, ray, nrm, obj, color )
     t_3d *pos, *ray, *nrm;
     t_object *obj;
     t_color *color;
{
  int lnum;
  double k,dis,bright,spec,diffuse;
  t_surface *surf;
  t_3d refl,ltray;
  t_color newcol;

  /* calculate reflected ray */
  k = -2.0 * dotp( ray, nrm );
  refl.x = k * nrm->x + ray->x;
  refl.y = k * nrm->y + ray->y;
  refl.z = k * nrm->z + ray->z;

  /* ambient light contribuition */
  surf = &surface[obj->surfnum];
  color->r = surf->ar;
  color->g = surf->ag;
  color->b = surf->ab;

  for( lnum=0; lnum < nlight; lnum++ )
  {
    /* get ray to light */
    lightray( lnum, pos, &ltray );
    diffuse = dotp( nrm, &ltray );
    if( diffuse > 0.0 )
    {
      /* object faces light, add diffuse */
      bright=brightness(obj->id,lnum,pos,&ltray);
      diffuse *= bright;
      color->r += surf->dr * diffuse;
      color->g += surf->dg * diffuse;
      color->b += surf->db * diffuse;

      spec = dotp( &refl, &ltray );
      if( spec > 0.0 )
      {
        /* highlight is here, add specular */
         spec = bright * pow( spec, surf->coef );
         color->r += surf->sr * spec;
         color->g += surf->sg * spec;
         color->b += surf->sb * spec;
      }
    }
  }

  /* reflection */
  k = surf->refl;
  if( ( k > 0.0 ) && ( level < maxlevel ) )
  {
    level++;
    dis = intersect( obj->id,pos,&refl,&newcol );
    if( dis > 0 )
    {
       color->r += newcol.r * k;
       color->g += newcol.g * k;
       color->b += newcol.b * k;
    }
    else
    {
       color->r += background.r * k;
       color->g += background.g * k;
       color->b += background.b * k;
    }
    level--;
  }

  /* transparency */
  k = surf->transp;
  if( k > 0.0 )
  {
    color->r *= ( 1-k );
    color->g *= ( 1-k );
    color->b *= ( 1-k );
    dis = intersect( obj->id,pos,ray,&newcol );
    if( dis > 0 )
    {
       color->r += newcol.r * k;
       color->g += newcol.g * k;
       color->b += newcol.b * k;
    }
    else
    {
       color->r += background.r * k;
       color->g += background.g * k;
       color->b += background.b * k;
    }
  }
}
```

3.3 Boxes

The **makbox** routine allocates the space for a box description and fills is with values which were passed to the routine.

intbox computes the intersection between a ray and a box. The source position of the ray, the ray vector, and the object, a box, are passed to the routine. The distance to the nearest intersection is returned. A value of zero is returned is no intersection is found.

The ray can hit any of the six sides of the box. Therefore, the six possible intersections must be checked.

Esch intersection calculations has two parts. First, the intersection of the ray with the plane of the box side is found. Then the point of intersection with the plane is checked to determine whether the it lies within the box side.

Let Xp,Yp,Zp be the ray origin, Xr,Yr,Zr the ray, Xc,Yc,Zc the box center, and Xs,Ys,Zs the sizes of the box sides. The intersection calculation for the plus X side os the box is

$$S = ((Xc + Xs) - Xp) / Xr$$

This gives the point of intersection with the plane

$$X = Xc + Xs$$

Then, the y and z cordinates of the point are found.

$$Y = Yp + S \cdot Yr$$

$$Z = Zp + S \cdot Zr$$

If Y is within Ys of Yc and Z is within Zs of Zc, the point is on the box.

This calculation is performed for all six sides. The distance to the colsest valid intersection is returned. The code is fairly long, but simple.

The normal is found by keeping track of which side os the box is intersected. The normal is 1.0 in direction of that side of the box.

```
/***** box.c *****/
#include <math.h>
#include "constants.h"
#include "typedefs.h"
#include "globalvar.h"
#define FAR_AWAY 99.99E+20

/* create box object */

int makbox( surf, x, y, z, xs, ys, zs )
     int surf;
     double x, y, z;
     double xs, ys, zs;
{
  int size;
  o_box *box;

  size = sizeof( o_box );
  box = ( ( o_box * ) malloc( size ) );
  object[nobject].id = nobject;
  object[nobject].objtyp = OTYPBOX;
  object[nobject].surfnum = surf;
  object[nobject].objpnt.p_box = box;
  box->x = x;
  box->y = y;
  box->z = z;
  box->xs = xs;
  box->ys = ys;
  box->zs = zs;
}

/* intersection calculation for ray and box */

double intbox( pos, ray, obj )
     t_3d *pos;          /* origin of ray */
     t_3d *ray;          /* ray vector */
     t_object *obj;      /* box description */
{
  double s, ss, xhit, yhit, zhit;
  double xadj, yadj, zadj;
  o_box  *box;

  box = obj->objpnt.p_box;
  ss = FAR_AWAY;
  /* translate ray origin to object's space */
  xadj = pos->x - box->x;
  yadj = pos->y - box->y;
  zadj = pos->z - box->z;

  if( ray->x != 0 )   /* check x faces */
  {
    s = ( box->xs - xadj ) / ray->x;
    if( ( s > 0 ) && ( s < ss ) )
    {
       yhit = fabs( yadj + s * ray->y );
       zhit = fabs( zadj + s * ray->z );
       if(( yhit<box->ys ) && ( zhit<box->zs ))
      {
         box->sidehit = 0;
         ss = s;
       }
    }
    s = ( -box->xs - xadj ) / ray->x;
    if( ( s > 0 ) && ( s < ss ) )
    {
       yhit = fabs( yadj + s * ray->y );
       zhit = fabs( zadj + s * ray->z );
       if(( yhit<box->ys ) && ( zhit<box->zs ))
      {
         box->sidehit = 1;
         ss = s;
       }
    }
  }

  if( ray->y != 0 )   /* check y faces */
  {
    s = ( box->ys - yadj ) / ray->y;
    if( ( s > 0 ) && ( s < ss ) )
    {
       xhit = fabs( xadj + s * ray->x );
       zhit = fabs( zadj + s * ray->z );
       if(( xhit<box->xs ) && ( zhit<box->zs ))
      {
         box->sidehit = 2;
         ss = s;
       }
    }
    s = ( -box->ys - yadj ) / ray->y;
    if( ( s > 0 ) && ( s < ss ) )
    {
       xhit = fabs( xadj + s * ray->x );
       zhit = fabs( zadj + s * ray->z );
       if(( xhit<box->xs ) && ( zhit<box->zs ))
      {
         box->sidehit = 3;
         ss = s;
      }
    }
  }

  if( ray->z != 0 )   /* check z faces */
  {
    s = ( box->zs - zadj ) / ray->z;
    if( ( s > 0 ) && ( s < ss ) )
    {
       xhit = fabs( xadj + s * ray->x );
       yhit = fabs( yadj + s * ray->y );
       if(( xhit<box->xs ) && ( yhit<box->ys ))
      {
         box->sidehit = 4;
         ss = s;
       }
    }
    s = ( -box->zs - zadj ) / ray->z;
    if( ( s > 0 ) && ( s < ss ) )
    {
       xhit = fabs( xadj + s * ray->x );
       yhit = fabs( yadj + s * ray->y );
       if(( xhit<box->xs ) && ( yhit<box->ys ))
      {
         box->sidehit = 5;
         ss = s;
      }
    }
  }

  if( ss == FAR_AWAY ) return( 0.0 );
  return( ss );
}

/* normal calculation for box */

int nrmbox( pos, obj, nrm )
     t_3d *pos;       /* point of intersection */
     t_object *obj;   /* box description */
     t_3d *nrm;       /* return surface normal */
{
  o_box *box;

  box = obj->objpnt.p_box;
  nrm->x = 0.0;
  nrm->y = 0.0;
  nrm->z = 0.0;
  switch( box->sidehit )
  {
    case(0): nrm->x = 1.0;
               break;
    case(1): nrm->x = -1.0;
               break;
    case(2): nrm->y = 1.0;
               break;
    case(3): nrm->y = -1.0;
             break;
    case(4): nrm->z = 1.0;
             break;
    case(5): nrm->z = -1.0;
             break;
  }
  return;
}
```

3.4 Triangles

The **maktri** routine allocates the space for a triangle structure and fills the fields. The three corner points os the triangle are passed into **maktri**. In order to make the ray-triangle intersection calculation fast, some values are precumputed. A plane is define by a normal vector and a plane constant. The normal vector and plane constant for the triangle are computed and stored. Each edge of the triangle can be defined by a plane which is perpendicular to the plane of the triangle. The vectors and constants for the triangle's three edges are computed and stored.

The normal vector for the triangle is the cross product os the vectors from the first point to the second and the second to the third. The plane constant is the dot product os the normal vector and any one of the triangle's point.

The normal for an edge is defined by the cross product of the vector from its start point to its end point and the triangle's normal vector. The plane constant for an edge is the dot product of the edge vector and one of the points in the edge.

inttri computes the intersection between a ray and a triangle in two steps. First, it finds the intersection of the ray with the plane of the triangle. Next, it determines whether the point os intersection is inside the triangle. Plane equations are used for each step. Note that the points of the triangle are either in clockwise or counterclockwise order. You must be consistent, because the direction of the triangle's normal is determined by the order os the points. The convention used is that as you look at a triangle, its normal points towards you, and its points are in counterclockwise order.

```
/***** triangle.c *****/
#include "constants.h"
#include "typedefs.h"
#include "globalvar.h"
#include "funcdefs.h"

/* create triangle object */

int maktri( surf, p1, p2, p3 )
     int surf;
     t_3d *p1, *p2, *p3;
{
  int size;
  o_triangle *triangle;
  t_3d vc1, vc2, vc3;

  size = sizeof( o_triangle );
  triangle = ( ( o_triangle * ) malloc( size ) );
  object[nobject].id = nobject;
  object[nobject].objtyp = OTYPTRIANGLE;
  object[nobject].surfnum = surf;
  object[nobject].objpnt.p_triangle = triangle;
  /* edge vectors */
  vc1.x = p2->x - p1->x;
  vc1.y = p2->y - p1->y;
  vc1.z = p2->z - p1->z;
  vc2.x = p3->x - p2->x;
  vc2.y = p3->y - p2->y;
  vc2.z = p3->z - p2->z;
  vc3.x = p1->x - p3->x;
  vc3.y = p1->y - p3->y;
  vc3.z = p1->z - p3->z;
  /* plane of triangle */
  crossp( &triangle->nrm, &vc1, &vc2 );
  triangle->d = dotp( &triangle->nrm, p1 );
  /* edge planes */
  crossp( &triangle->e1, &triangle->nrm, &vc1 );
  triangle->d1 = dotp( &triangle->e1, p1 );
  crossp( &triangle->e2, &triangle->nrm, &vc2 );
  triangle->d2 = dotp( &triangle->e2, p2 );
  crossp( &triangle->e3, &triangle->nrm, &vc3 );
  triangle->d3 = dotp( &triangle->e3, p3 );
}

/* intersection calculation for ray and triangle
*/

double inttri( pos, ray, obj )
     t_3d *pos;        /* origin of ray */
     t_3d *ray;        /* ray vector */
     t_object *obj;    /* triangle description */
{
  double s, k;
  t_3d point;
  o_triangle *triangle;

  triangle = obj->objpnt.p_triangle;
  /* plane intersection */
  k + dotp( &triangle->nrm, ray );
  if( k == 0 ) return( 0.0 );
  s = (triangle->d - dotp(&triangle->nrm,pos))/k;
  if( s <= 0 ) return( 0.0 );

  point.x = pos->x + ray->x *s;
  point.y = pos->y + ray->y *s;
  point.z = pos->z + ray->z *s;

  /* edge checks */
  k = dotp(&triangle->e1, &point) - triangle->d1;
  if( k < 0 ) return( 0.0 );
  k = dotp(&triangle->e2, &point) - triangle->d2;
  if( k < 0 ) return( 0.0 );
  k = dotp(&triangle->e3, &point) - triangle->d3;
  if( k < 0 ) return( 0.0 );

  return( s );
}

/* normal calculation for triangle */

int nrmtri( pos, obj, nrm )
     t_3d *pos;       /* point of intersection */
     t_object *obj;   /* triangle description */
     t_3d *nrm;       /* return surface normal */
{
  o_triangle *triangle;

  triangle = obj->objpnt.p_triangle;
  nrm->x = triangle->nrm.x;
  nrm->y = triangle->nrm.y;
  nrm->z = triangle->nrm.z;
  return;
}
```

3.5 Superquadrics

Superquadrics can be described as boxes with rounded corners. They are specified by equations of the form

$$a \cdot |x|^n + b \cdot |y|^n + c \cdot |z|^n = r^n$$

The **maksup** routine allocates the space for a superquadric description and fills it with the values which were passed to the routine and values which it calculates.

The superquadric structure has all of the fields in the box structure, along with some new fields. The values which correspond to the box structure fields are passed into the routine. The new fields are the A, B, C, R, and N calues in the superquadric equation, along with an error metric. The N value is passed into the routine. The other values are derived from the box values. R is the size of the largest box side. A, B, and C are found by dividing the box sizes by R. The error metric is used in the interative solution of the ray-object intersection. The solution is accurate enough when the measured error is less than this value.

The intersection of a ray with a superquadric cannot be calculated directly because the solution of an order N polynomial is required. Therefore, an interative technique is need. The secant method is used because it is very simple.

The superquadric surface lies entirely within the box used to define it. Any ray which will strike the superquadric will first strike the box. So, the ray-box intersection is used as the initial value for the ray-superquadric intersection. This also gives an inexpensive rejection criteria for most of the rays which would not hit the superquadric.

There is only one special case. If the ray starts within the surrounding box, it can hit the superquadric without hitting the box.

```
/***** superquadric.c *****/
#include <math.h>
#include "constants.h"
#include "typedefs.h"
#include "funcdefs.h"
#include "globalvar.h"
#define ERRCONST 1.05

/* create superquadratic object */

int maksup( surf, x, y, z, xs, ys, zs, power )
     int surf;
     double x, y, z;
     double xs, ys, zs;
     double power;
{
  int size;
  double max;
  o_superq *super;

  size = sizeof( o_superq );
  super = ( ( o_superq * ) malloc( size ) );
  object[nobject].id = nobject;
  object[nobject].objtyp = OTYPSUPERQ;
  object[nobject].surfnum = surf;
  object[nobject].objpnt.p_superq = super;
  super->x = x;
  super->y = y;
  super->z = z;
  super->xs = xs;
  super->ys = ys;
  super->zs = zs;
  super->pow = power;
  max = xs;
  if( ys > max ) max = ys;
  if( zs > max ) max = zs;
  super->a = xs / max;
  super->b = ys / max;
  super->c = zs / max;
  super->r = pow( max, power );
  super->err = pow((ERRCONST * max),power) -
                    super->r;
}

/* intersection calculation for ray and
superquadratic */

double intsup( pos, ray, obj )
     t_3d *pos;          /* origin of ray */
     t_3d *ray;          /* ray vector */
     t_object *obj;      /* superquadratic
description */
{
  double xsiz, usiz, zsiz;
  double s, s1, t;
  double xadj, yadj, zadj;
  double old, result, p;
  o_superq *super;

  /* find box intersection */
  s = intbox( pos, ray, obj );
  if( s == 0 ) return( 0.0 );
  super = obj->objpnt.p_superq;
  xadj = pos->x - super->x;
  yadj = pos->y - super->y;
  zadj = pos->z - super->z;

  /* special case - ray origin within box */
  if( ( fabs( xadj ) < super->xs ) &&
      ( fabs( yadj ) < super->ys ) &&
      ( fabs( zadj ) < super->zs ) ) s = 0;

  /* initial solution */
  p = super->pow;
  result = pow(fabs((xadj+ray->x*s)/super->a),p)+
           pow(fabs((yadj+ray->y*s)/super->b),p)+
           pow(fabs((zadj+ray->z*s)/super->c),p)-
             super->r;

  if( result < super->err ) return( s );

  s1 = s;
  s = s + 0.001;
  /* interactive refinament */
  while( result > super->err )
  {
    old = result;
    result=pow(fabs((xadj+ray->x*s)/super->a),p)+
           pow(fabs((yadj+ray->y*s)/super->b),p)+
             pow(fabs((zadj+ray->z*s)/super->c),p)
           - super->r;
    if( result >= old ) return( 0.0 );
    t = (result * ( s - s1 )) / ( result - old );
    s1 = s;
    s -=t;
  }
  return( s );
}
```

```
/* normal calculation for superquadratic */

int nrmsup( pos, obj, nrm )
     t_3d *pos;    /* point of intersection */
     t_object *obj;/*superquadratic description*/
     t_3d *nrm;    /* return surface normal */
{
  double k;
  o_superq *super;

  super = obj->objpnt.p_superq;
  nrm->x = ( pos->x - super->x ) / super->a;
  nrm->y = ( pos->y - super->y ) / super->b;
  nrm->z = ( pos->z - super->z ) / super->c;
  k = super->pow - 1;
  if( nrm->x > 0 ) nrm->x = pow( nrm->x, k );
  else nrm->x = -pow( -nrm->x, k );
  if( nrm->y > 0 ) nrm->y = pow( nrm->y, k );
  else nrm->y = -pow( -nrm->y, k );
  if( nrm->z > 0 ) nrm->z = pow( nrm->z, k );
  else nrm->z = -pow( -nrm->z, k );
  normalize( nrm );
}
```

3.6 Input Processing

Writing routines to parse and process user input is a large task. In a ray tracing system, many things should be under user control. It should be easy to specify viewing parameters. Large files of object descriptions should be easy to process. Files contaning surface type descriptions should be produced once and used for many different images.

The many and varied requirements of the user interface make it quite complicated. Addicionally, it is very nice to be able to change the interface easily. One of the cleanest solutions is to use parsing tools, in particular, the UNIX tools LEX and YACC. They are simple to use and because they generated C code, are easily added to the ray tracer.

LEX handles character by character input processing. It recognizes key words and can read numbers in just about any form. It produces tokens which are used by YACC.

```
/***** input_lex.l *****/
%{
#include <stdio.h>
#include "y.tab.h"
%}
alpha [a-zA-Z]
special [\.\_]
digit [0-9]
exp [Ee][-+]?{digit}+
string {alpha}({alpha}|{special})*
%start COMMENT
%%
" "                         ;
\t                          ;
\n                          ;
"/*"                        {BEGIN COMMENT;}
<COMMENT>[^/\n]*\n          ;
<COMMENT>[^/\n]+"/"         {if (yytext[yyleng-2]
                                 =='*') BEGIN 0;}
eyep                        {return(TEYEP);}
lookp                       {return(TLOOKP);}
up                          {return(TUP);}
fov                         {return(TFOV);}
screen                      {return(TSCREEN);}
light                       {return(TLIGHT);}
surface                     {return(TSURFACE);}
background                  {return(TBACKGROUND);}
maxlevel                    {return(TMAXLEVEL);}
sphere                      {return(TSPHERE);}
box                         {return(TBOX);}
triangle                    {return(TTRIANGLE);}
superq                      {return(TSUPERQ);}
outfile                     {return(TOUTFILE);}
{string}                    {yylval.c=yytext;
                              return(TSTRING);}
[+-]?{digit}+       {sscanf(yytext,"%d",&yylval.i);
                              return(TINT);}

[+-]?{digit}+"."{digit}*({exp})? |
[+-]?{digit}*"."{digit}+({exp})? |
[+_]?{digit}+{exp}
                    {sscanf(yytext,"%F",&yylval.d);
                              return(TFLOAT);}
%%
yywrap() {return(1);}
```

YACC recognizes patters of tokens and performs user specified actions based on them.

For example, LEX recognizes the letters 'e' 'y' 'e' 'p' as the string "eyep". It recognizes the characters '1' '0' ' ' '0' ' ' '0' as the numbers 10 0 0. For each string of characters, it sends a token to YACC. In this case it sends TEYEP TINT TINT TINT, with each TINT having a numerical value assiciated with it. YACC recognizes that this sequence of tokes specifies an eyepoint position. User written C code actions then process the eyepoint position.

```
/*** input_yacc.y ***/
%{
#include <stdio.h>
#include "typedefs.h"
#include "globalvar.h"

#define YYDEBUG  1
int number;
t_3d p1, p2, p3;
%}
%union {
       char *c;
       int i;
       double d;
       }
%token <i> TINT
%token <d> TFLOAT
%token <c> TSTRING
%token TEYEP TLOOKP TUP TFOV TSCREEN TMAXLEVEL
%token TLIGHT TSURFACE TSPHERE TBOX TTRIANGLE
TSUPERQ
%token TBACKGROUND TOUTFILE
%type <d> Fnumber
%type <c> String
%%
File            : File Item
                | Item
                ;
Item            : Eyep
                | Lookp
                | Up
                | Fov
                | Screen
                | Maxlevel
                | Background
                | Light
                | Surface
                | Sphere
                | Box
                | Triangle
                | Superq
                | Outfile
                ;
Eyep            : TEYEP Fnumber Fnumber Fnumber
                { eyep.x = $2;
                  eyep.y = $3;
                  eyep.z = $4;
                 }
                ;
Lookp           : TLOOKP Fnumber Fnumber Fnumber
                { lookp.x = $2;
                  lookp.y = $3;
                     lookp.z = $4;
                 }
                ;
Up              : TUP Fnumber Fnumber Fnumber
                { up.x = $2;
                  up.y = $3;
                     up.z = $4;
                 }
                ;
Fov             : TFOV Fnumber Fnumber
                { hfov = $2; vfov = $3; }
                ;
Screen          : TSCREEN TINT TINT
                { sizey = $2; sizex = $3; }
                ;
Maxlevel        : TMAXLEVEL TINT
                { maxlevel = $2; }
                ;
Background      : TBACKGROUND Fnumber Fnumber
Fnumber
                { background.r = $2;
                  background.g = $3;
                     background.b = $4;
                 }
                ;
Light           : TLIGHT Fnumber
                     Fnumber Fnumber Fnumber
                { if (nlight == lightlim)
                      yyerror("too many lights");
                     light[nlight].bright = $2;
                     light[nlight].x = $3;
                     light[nlight].y = $4;
                     light[nlight].z = $5;
                     nlight++;
                 }
                ;
Surface         : TSURFACE TINT
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                { number = $2;
                   if (number >= surfacelim)
                    yyerror("surface # too big");
                   surface[number].ar = $3;
                   surface[number].ag = $4;
                   surface[number].ab = $5;
                   surface[number].dr = $6;
                   surface[number].dg = $7;
                   surface[number].db = $8;
                   surface[number].sr = $9;
                   surface[number].sg = $10;
                   surface[number].sb = $11;
                   surface[number].coef = $12;
                   surface[number].refl = $13;
                   surface[number].transp = $14;
                   nsurface++;
                 }
                ;
Sphere          : TSPHERE TINT Fnumber
                      Fnumber Fnumber Fnumber
                { if (nobject == objectlim)
                     yyerror("too many objects");
                   maksph($2,$3,$4,$5,$6);
                   nobject++;
                   }
                ;
Box             : TBOX TINT
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                { if (nobject == objectlim)
                     yyerror("too many objects");
                   makbox($2,$3,$4,$5,$6,$7,$8);
                   nobject++;
                   }
                ;
Triangle        : TTRIANGLE TINT
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                { if (nobject == objectlim)
                     yyerror("too many objects");
                   p1.x=$3; p1.y=$4; p1.z=$5;
                   p2.x=$6; p2.y=$7; p2.z=$8;
                   p2.x=$9; p3.y=$10; p3.z=$11;
                   maktri($2,&p1,&p2,&p3);
                   nobject++;
                   }
                ;
Superq          : TSUPERQ TINT
                     Fnumber Fnumber Fnumber
                     Fnumber Fnumber Fnumber
                     Fnumber
                { if (nobject == objectlim)
                     yyerror("too many objects");
                 maksup($2,$3,$4,$5,$6,$7,$8,$9);
                   nobject++;
                   }
                ;
Outfile         : TOUTFILE String
                { strcpy(outfilename, $2); }
                ;
Fnumber         : TFLOAT
                { $$=$1; }
                | TINT
                { $$=$1; }
                ;
String          : TSTRING
                { $$ = $1; }
                ;
%%
yyerror(s)
  char *s;
  {
  fprintf( stderr,"%s\n",s);
  }
```

3.7 Sample Input Files

An image description consists of a series of keywords and parameters read from standard input. The keywords can occur in any order with spaces and C-like comments interspersed.

basic.pds contains the image description to create the default picture produced by the basic system (figure 1). The command **tracer2.go < basic.pds** runs the full system, to produce this image.

```
/*** basic.pds ***/
    screen 512 512
    fov     50  50
    eyep   100   0   0
    lookp    0   0   0
    up       0   1   0
    maxlevel 0
    outfile basic.pix
    light  /* brightness */ 1.0
           /* position   */ 100.0 300.0 0.0
    sphere /* color  */ 0
           /* radius */ 40
           /* center */ 5 30 40
    sphere /* color  */ 1
           /* radius */ 30
           /* center */ -10 -10 -40
    surface 0
        30   0   0 /* ambient */
        90   0   0 /* diffuse */
      190 120 120 /* specular */
        15 0.0 0.0 /* coef, transm, refl */
    surface 1
         0  50   0 /* ambient */
         0 100   0 /* diffuse */
        30  40  30 /* specular */
         2 0.0 0.5 /* coef, transm, refl */
    background /* color */ 100 200 250
```

all.pds contais the image description for figure 2. Each of the four object types as well as reflection and transparency is shown in this image.

```
/*** all.pds ***/
    screen 512 512
    fov     20  20
    eyep   100 150  90
    lookp    0   0   0
    up       0   1   0
    maxlevel 1
    outfile  all.pix
    light    /* brightness */ 1.0
              /* position */ 100.0 300.0 0.0
    superq   /* color  */ 0
              /* center */ 20 10 10
              /* size   */ 10 20 5
              /* power  */ 5
    sphere   /* color  */ 1
              /* radius */ 10
              /* center */ -10 25 -20
    triangle /* color  */ 2
              /* point  */ -30  30  0
              /* point  */ -34  15 19
              /* point  */ -34 -12 -9
    box      /* color  */ 3
             /* center */ -17 -10  7
             /* size   */  12   5 25
    surface 0
         0  50   0 /* ambient */
         0 100   0 /* diffuse */
       30  60   0 /* specular */
         6 0.0 0.2 /* coef, transm, refl */
    surface 1
        90  90   0 /* ambient */
       230 230   0 /* diffuse */
      110  90  30 /* specular */
         2 0.0 0.0 /* coef, transm, refl */
    surface 2
         0  90  90 /* ambient */
         0 230 230 /* diffuse */
       30 110  90 /* specular */
         8 0.0 0.0 /* coef, transm, refl */
    surface 3
        60   0  90 /* ambient */
       210   0 230 /* diffuse */
        90  30 110 /* specular */
        15 0.2 0.0 /* coef, transm, refl */
    background /* color */ 0 0 250
```

3.8 UNIX Makefile

UNIX provides a very convenient toll for building large programming system: the makefile. It specifies the routines needed to build the system, dependencies between the routines, and the process for building each routine.

If you are not on a UNIX system, you should be able to build command files to compile and link the basic system and the full system.

```
#::: makefile :::#
# C compile command for all routines #
.c.o:;cc -O -c $*.c

# build basic system #
basic: main.o setup.o viewing.o intersect.o \
       shade.o light.o raymath.o stubs.o    \
       sphere.o outputp.o
cc -O -o tracer1.go \
        main.o setup.o viewing.o intersect.o \
        shade.o light.o raymath.o stubs.o    \
        sphere.o outputp.o
       -lm

# build full system #
system: main.o setup.o viewing.o intersect.o   \
         shade2.o light2.o raymath.o input_lex.o\
         input_yacc.o sphere.o box.o triangle.o \
         superquadric.o outputp.o
cc -O -o tracer2.go \
         main.o setup.o viewing.o intersect.o   \
         shade2.o light2.o raymath.o input_lex.o\
         input_yacc.o sphere.o box.o triangle.o \
         superquadric.o outputp.o
         -lm

# YACC and LEX for full system #
input_yacc.c: input_yacc.y
yacc -d input_yacc.y
mv y.tab.c input_yacc.c

input_lex.c: input_lex.l input_yacc.c
lex -t input_lex.l > input_lex.c
```

3.9 Further Suggestions

This article described only a small subset of what can be accomplished with ray tracing. A few important topics are listed below.

Supersampling & Antialiasing: Sending only one ray per pixel leads to 'jaggies' along the edges of objects. Supersampling is the process of sending multiple rays per pixel. Antialiasing deals with which rays to send for a pixel and how to combine the resulting colors from the rays. [Whitted80] suggest sending rays at the corners of each pixel. Adapting the number of rays sent into a pixel based of the complexity of the image is suggested in [Lee85].

Object Types: Many different types of objects have been ray traced. Ray tracing prism, surfaces of revolution, and fractals is presented in [Kajiya83].

Bounding Volumes: Chacking every ray for intersection with every object becomes prohibitively expensive for large databases. One solution [Rubin80] is to place bounding volumes around complex objects. Simple objects such as spheres or boxes are used as bounding volumes. Intersection with the complex object is only checked after a ray strikes the bounding object.

Octree & Other Space Subdivision: Algorithms for automatic generation of bounding volumes led to the idea of space subdivision. [Glassner84] used octrees to subdivide space; [Kay86] and [Fujimoto86] present other subdivision algorithms. Objects are checked for intersection with a ray only when the ray and the object are in the same region of space.

Shading Models: Many different shading models have been proposed. The goal of each is to more accurately describe the real world. [Cook81] and [Hall84] present two particularly good models.

Patters & Textures: Addings patterns and textures is an efficient way to increase the realism os images. [Blinn76] and [Blinn78] discuss patters and textures in general. Procedurally defined marble, soap films, water droplets and waves are presented in [Perlin85]. Procedurally defined wood texture is discussed in [Peachey85].

Distributed Ray Tracing: [Cook84] and [Dippe85] introduced extensions to ray tracing which produce antialiased images with depth of field, motion blur, surfaces gloss, shadow penumbra, and other wonderful effects.

References

[Appel68] Appel, Arthur. Some Techniques for Shading Machine Rendering of Solids. *AFIPS Spring Joint Computer Conference* 32 (1968) 37-45.

[Blinn76] Blinn, James F. Texture and Reflection in Computer Generated Images. *Communications of the ACM* 19,10 (October 1976) 542-547.

[Blinn78] Blinn, James F. Simulation of Wrinkled Surfaces. *Communications of the ACM* 12,3 (August 1976) 286-292.

[Cook81] Cook, Robert L. and Torrance, Kenneth E. A Reflectance Model for Computer Graphics. *Computer Graphics* 15,3 (August 1981) 307-316.

[Cook84] Cook, Robert L., Porter, Thomas and Carpenter, Loren. Distributed Ray Tracing. *Computer Graphics* 18,3 (July 1984) 137-145.

[Dippe85] Dippe, Mark A. and Wold, Erling Henry. Antialiasing Through Stochastic Sampling. *Computer Graphics* 19,3 (July 1985) 69-78.

[Foley82] Foley, James D. and Van Dam, Andries. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass. 1982.

[Fujimoto86] Fujimoto, Akira, Tanaka, Takyuki and Iwata, Kansei. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications* 6,4 (April 1986) 16-26.

[Glassner84] Glassner, Andrew S. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications* 4,10 (October 1984) 15-22.

[Goldstein71] Goldstein, Robert A. and Nagel, Roger. 3-D Visual Simulation. *Simulation* 16 (January 1971) 25-31.

[Hall84] Hall, Roy A. A Testbed for Realistic Image Synthesis. *IEEE Computer Graphics and Applications* 3,8 (November 1983) 10-20.

[Kajiya83] Kajiya, James T. New Techniques for Ray Tracing Procedurally Defined Objects. *Computer Graphics* 17,3 (July 1983) 91-102.

[Kay86] Kay, Timothy L. and Kajiya, James T. Ray Tracing Complex Scenes. *Computer Graphics* 20,4 (July 1986) 269-278.

[Lee85] Lee, Mark E., Redner, Ricahrd A. and Uselton, Samuel P. Statistically Optimized Sampling for Distributed Ray Tracing. *Computer Graphics* 19,3 (July 1985) 61-67.

[Newman73] Newman, Willian M and Sproul, Robert F. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York 1973.

[Peachey85] Peachey, Darwyn R. Solid Texturing of Complex Surfaces. *Computer Graphics* 19,3 (July 1985) 279-286.

[Perlin85] Perlin, Ken. An Image Synthesizer. *Computer Graphics* 19,3 (July 1985) 287-296.

[Roth82] Roth, Scott D. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing* 18 (1982), 109-144.

[Rubin80] Rubin, Steven M. and Whitted, Turner. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics* 14,3 (June 1980) 110-116.

[Whitted80] Whitted, Turner. An Improved Illumination Model for Shaded Display. *Communications of the ACM* 23,6 (June 1980) 343-349.