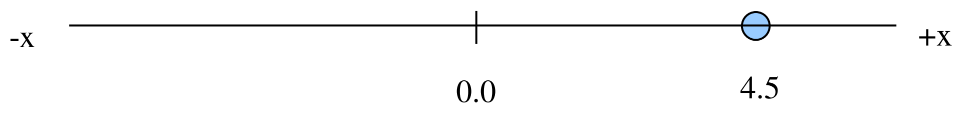
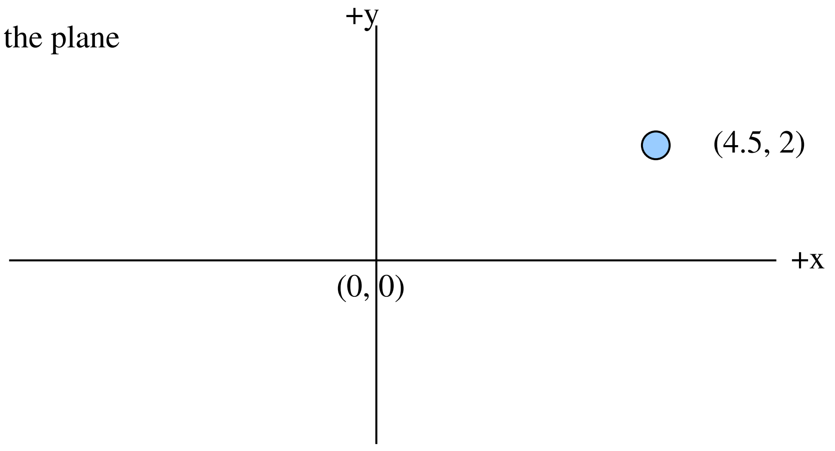
Basic elements of 3-D coordinate systems and linear algebra

Coordinatate systems are used to assign numeric values to locations with respect to a particular frame of reference commonly referred to as the *origin.* The number of dimensions in the coordinate system is equal to the number of perpendicular (orthgonal) axes and is also the number values needed to specify a location with respect to the origin.

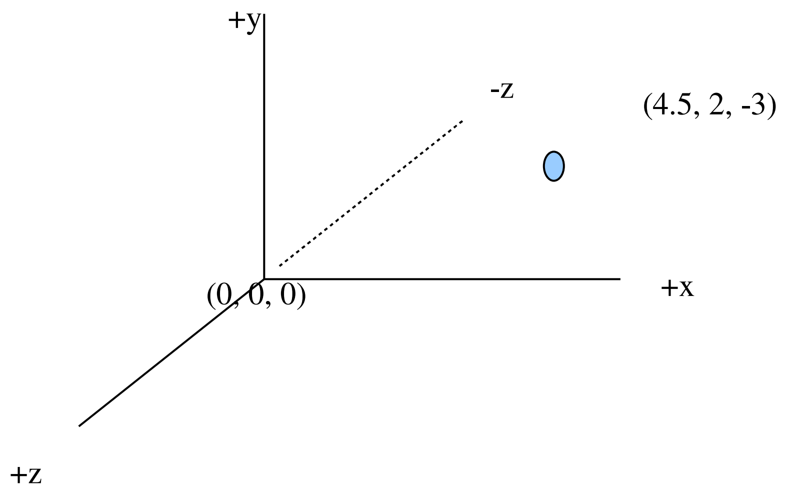
One dimension: the line



Two dimensions: the plane



Three dimensions: the universe as we perceive it (A right handed coordinate system is shown.. In a left handed system the direction of the positive z axis is reversed.)



Points in 3-D space

The location of a *point P* in 3-D Euclidean space is given by a triple (p_x, p_y, p_z)

The *x, y,* and *z* coordinates specify the distance you must travel in directions parallel to the the *x*, *y*, and *z* axes starting from the origin (0, 0, 0) to arrive at the point (p_x, p_y, p_z)

Vectors in 3-D space

A *vector* in 3-D space is sometimes called a *directed distance* because it represents both

- a *direction* and
- a *magnitude* or *distance*

In this context, the triple (p_x, p_y, p_z) can also be considered to represent

- the *direction* from the origin *(0, 0, 0)* to (p_x, p_y, p_z) and
- its length $sqrt(p_x^2 + p_y^2 + p_z^2)$ is the Euclidean (straight line) distance from the origin to (p_x, p_y, p_z)

Points and vectors

Two points in 3-D space implicitly determine a vector pointing from one to the other. Given two points P and Q in 3-D Euclidean space, the *vector*

$$R = P - Q = (p_x - q_x, p_y - q_y, p_z - q_z)$$

represents the direction from Q to P. Its length, as defined above is the distance, between P and Q. Note that the direction is a *signed* quantity. The direction from P to Q is the *negative* of the direction from Q to P. However, the *distance* from P to Q is always the same as the distance from Q to P.

Example: Let $V = (8, 6, 5)$ and $P = (3, 2, 0)$.

Then the vector direction from V to P is : $(3 - 8,\ 2 - 6,\ 0 - 5) = (-5, -4, -5)$

The vector direction from P to V is $(5, 4, 5)$

The distance between V and P is: sqrt(25 + 16 + 25) = sqrt(66) = 8.12.

The geometric interpretation of vector arithmetic

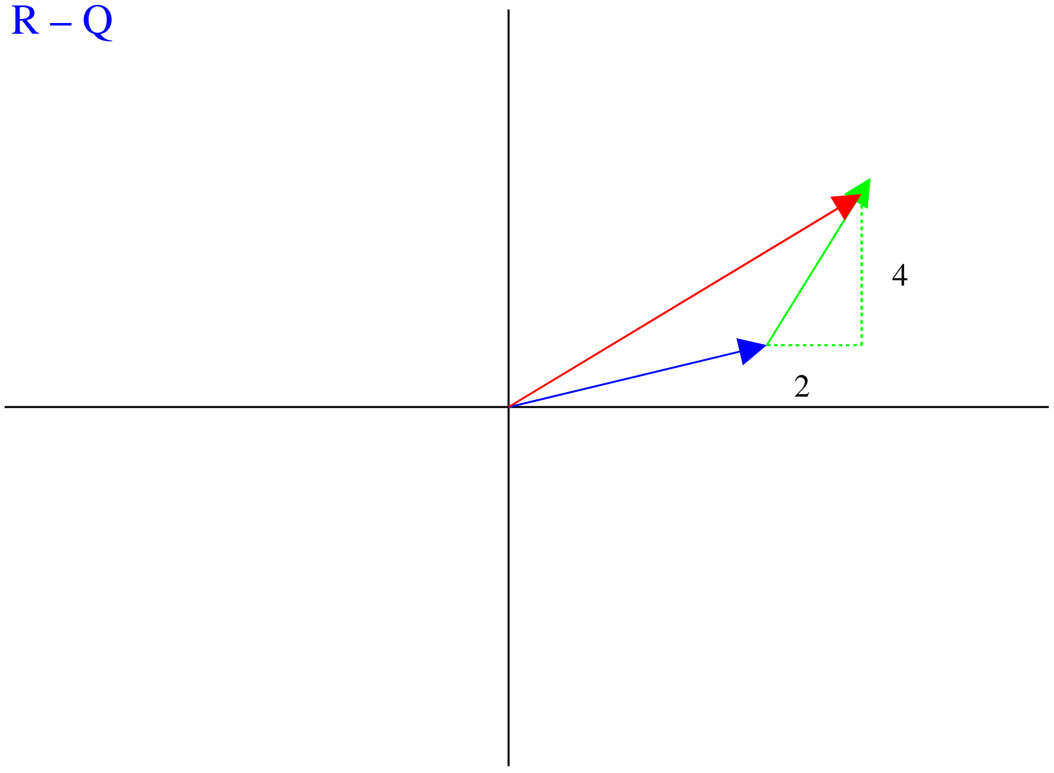
Here we work with 2 dimensional vectors to simplify the visual interpretation, but in 3-d the principles are the same.

P = (5, 1) => +5 in the *x* direction and then +1 in the *y* direction

Q = (2, 4) => +2 in the *x* direction and +4 in the *y* direction.

R = P + Q = (7, 5)

P = R – Q



Useful operations on vectors:

We can define the sum of two vectors *P* and *Q* as follows:

$$R = P + Q = (p_x + q_x,\ p_y + q_y,\ p_z + q_z)$$

$$(3, 4, 5) + (1, 2, 6) = (4, 6, 11)$$

The difference of two vectors is computed as:

$$R = P - Q = (p_x - q_x,\ p_y - q_y,\ p_z - q_z)$$

$$(3, 4, 5) - (1, 2, 6) = (2, 2, -1)$$

We also define multiplication (or *scaling)* of a vector by a scalar number *a*

$$S = aP = (ap_x, ap_y, ap_z)$$

$$3 * (1, 2, 3) = (3, 6, 9)$$

The *length* of a vector *P* is a scalar whose value is denoted:

$$|| P || = sqrt(p_x^2 + p_y^2 + p_z^2)$$

$$|| (3, 4, 5) || = sqrt(9 + 16 + 25) = sqrt(50)$$

A *unit vector* is a vector whose length is 1. Therefore an arbitrary vector *P* may be converted to a unit vector by scaling it by 1 / (its own length). Here *U* is a *unit vector* in the same direction as *P*.

$$U = (1 / || P ||)\ P$$

The *inner product* or *dot product* of two vectors *P* and *Q* is a *scalar number*

$$x = P \text{ dot } Q = (p_x q_x + p_y q_y, + p_z q_z)$$

$$(2, 3, 4) \text{ dot } (3, 2, 1) = 6 + 6 + 4 = 16$$

Thus $|| P || = sqrt(P \text{ dot } P)$

If *U* and *V* are unit vectors and θ is the angle beween them then:

$$cos\ (\theta) = U \text{ dot } V = V \text{ dot } U$$

Representing vectors in C

There are two reasonable ways to represent a vector:

Array based representation:

Use a three element double precision array:

```
double vec[3];
```

Where it is understood that

vec[0] is the x-component (coordinate)
vec[1] is the y-component
vec[2] is the z-component

Structure based representation

Define a structured type in which the elements are explicitly named

```
typedef struct vec_type
{
    double x;
    double y;
    double z;
} vec_t;
vec_t vec;
```

In this representation, it is explicit that

vec.x is the x-component
vec.y is the y-component
vec.z is the z-component

Religious wars have been fought over which is “correct”. We will refuse to engage in the war, but we *will use the structure based approach in this course.*

A library for 3-D vector operations

Since the above operations will be commonly required in the raytracer, you will build a library of functions which we will call *vector.h* to perform them. Here are the function prototypes that must be employed. Because the functions are called many times we will use the *inline* mechanism of *gcc* to improve performance. The *static* qualifier is used to avoid duplicate definition errors at link time when functions are included in *.h* files.

```
/* Scale a 3d vector */

static inline void vec_scale(
double fact,      /* Scale factor  */
vec_t *v1,        /* Input vector  */
vec_t *v2);       /* Output vector */

/* Return length of a 3d vector */

static inline double vec_len(
vec_t *v1);    /* Vector whose length is desired */

/* Compute the difference of two vectors */
/* v3 = v2 - v1                          */

static inline void vec_diff(
vec_t *v1,        /*  subtrahend    */
vec_t *v2,        /*  minuend       */
vec_t *v3);       /*  result        */

/* Compute the sum of two vectors   */

static inline void vec_sum(
vec_t *v1,        /*  addend        */
vec_t *v2,        /*  addend        */
vec_t *v3);       /*  result        */
```

```
/* Return the inner product of two input vectors */

static inline double vec_dot(
vec_t  *v1,        /* Input vector 1  */
vec_t  *v2);       /* Input vector 2  */

/* Copy one vector to another */

static inline void vec_copy(
vec_t *v1,        /* input vector */
vec_t *v2);       /* output vector */

/* Construct a unit vector in direction of input */

static inline void vec_unit(
vec_t *v1,        /* Input vector    */
vec_t *v2);       /* output unit vec */

/* Read in values of vector from file */

static inline void vec_load(
FILE  *in,
vec_t *v1);

/* Print values of vector to file */

static inline void vec_prn(
FILE  *out,        /* output file     */
char  *label,      /* label string    */
vec_t *v1);        /* vector to print */
```

Warning regarding aliased parameters

When parameters are passed using pointers a potentially destructive phenomenon known as *aliasing* may occur. Here the caller of *vec_unit()* is requesting that a vector be converted to a unit vector in place.

```
     vec_unit(v1, v1);
```

Now suppose the implementation of *vec_unit()* is as follows:

```
     static inline void vec_unit(
     vec_t *vin,
     vec_t *vout)
     {
          vout->x = vin->x / vec_len(vin);
          vout->y = vin->y / vec_len(vin);
          vout->z = vin->z / vec_len(vin);
     }
```

This looks correct and (assuming *vec_len()*) is working properly it will work correctly as long as the parameters *vin* and *vout* point to different vectors. However, if they point to the *same vector* incorrect computation will result. If *vin* and *vout* point to the same vector the assignment

```
     vout->x = vin->x / vec_len(vin);
```

also changes *vin->x*   Therefore, in the subsequent steps of the computation

```
     vout->y = vin->y / vec_len(vin);
     vout->z = vin->z / vec_len(vin);
```

vec_len() will *generally (but not always)* return a *different value than in the preceding step.* For the computation to work correctly, *vec_len()* must *always* return the *original length of the input vector.*

The function can be written correctly (and more efficiently) as.

```
     static inline void vec_unit(
     vec_t *vin,
     vec_t *vout)
     {
        double scale = 1.0 / vec_len(vin);
        vec_scale(scale, vin, vout);
     }
```

ALL vector functions should work correctly with aliased parameters.

A sample test driver for vector.h

```c
/* p33.c */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "vector.h"

vec_t v1 = {3.0, 4.0, 5.0};
vec_t v2 = {4.0, -1.0, 2.0};

main()
{
   vec_t v3;
   vec_t v4;
   double v;

   vec_prn(stdout, "v1", &v1);
   vec_prn(stdout, "v2", &v2);
   vec_diff(&v1, &v2, &v3);
   vec_prn(stdout, "v2 - v1 = ", &v3);

   v = vec_dot(&v1, &v2);
   printf("v1 dot v2 is %8.3lf \n", v);

   v = vec_len(&v1);
   printf("Length of v1 is %8.3lf \n", v);

   vec_scale(1 / v, &v1, &v3);
   vec_prn(stdout, "v1 scaled by its 1/ length:", &v3);

   vec_unit(&v1, &v4);
   vec_prn(stdout, "unit vector in v1 direction:", &v4);
}
```

```
v1   3.000   4.000   5.000
v2   4.000  -1.000   2.000
v2 - v1 =    1.000  -5.000  -3.000
v1 dot v2 is   18.000
Length of v1 is    7.071
v1 scaled by its 1/ length:   0.424   0.566   0.707
unit vector in v1 direction:   0.424   0.566   0.707
```

Representing *rgb* data

In the raytracer we will work with three types of *rgb* data:

reflective materials
emissive lights
pixels

We will use *rgb* data for all three, but will use different models for the interaction of lights with materials than for the pixmap itself.

As in CPSC 101, for the pixmap data used in the .ppm file, we use an unsigned character representation where *0 means black and 255 means maximal brightness*.

```
typedef struct irgb_type
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
} irgb_t;
```

For representing lights, reflective materials and their interactions we use:

```
typedef struct drgb_type
{
    double r;
    double g;
    double b;
} drgb_t;
```

In this representation *0.0 means black and 1.0 means maximal brightness*. It is possible to produce values > 1.0 and as in CPSC 101 these must be clamped before converting to *irgb_t*.

Ray tracing introduction

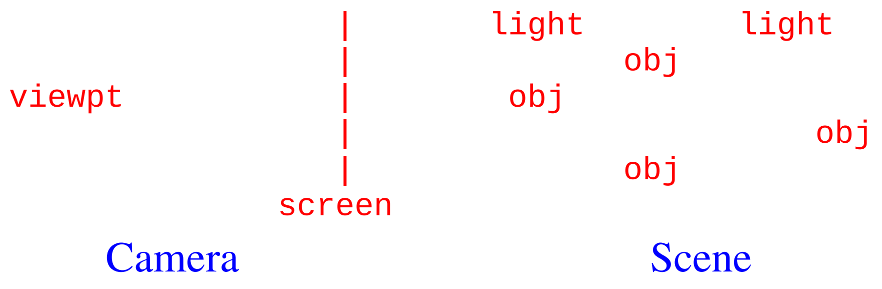
The objective of a ray tracing program is to render a photo-realistic image of a virtual scene in 3 dimensional space. There are two major components in the process:

The virtual camera

1 - *The viewpoint*	This is the location in *3-d* space at which the viewer of the scene is located
2 - *The screen*	This defines a virtual *window* through which the viewer observes the scene. The window can be viewed as a discrete 2-D pixel array (pixmap) . The *raytracing* procedure computes the color of each pixel. When all pixels have been computed, the *pixmap* is written out as a .ppm file

The scene to be viewed

3 - *materials*	One or more material definitions may be associated with each object. The material definition describes how the object interacts with a light. Among other things the material definition defines the color of the object.
4 - *light sources*	Lights themselves are *not* visible, but they do illuminate objects and may be subject to shadowing. Lights may be white or colored.
5 – *visible objects*	Reflective objects that are illuminated by the light sources



World and window coordinate systems

Two coordinate systems will be involved and it will be necessary to map between them:

1 - *Window coordinates*	the coordinates of individual pixels in the virtual window. These are two dimensional (x, y) interger numbers For example, if a 400 col *x* 300 row image is being created the window *x* coordinates range from 0 to 399 and the window *y* coordinates range from 0 to 299. In the raytracing algorithm a ray will be *fired* through each pixel in the window. The color of the pixel will be determined by the color of the object(s) the ray hits.
2 - *World coordinates*	the "natural" coordinates of the scene measured in feet/meters etc. Since world coordinates describe the entire scene these coordinates are three dimensional (x, y, z) floating point numbers.

For the sake of simplicity we will assume that

the screen lies in the *z = 0.0* plane
the lower left corner of the *window* has world coordinates *(0.0, 0.0, 0.0)*
the lower left corner of the *window* has window (pixel) coordinates *(0, 0)*
the location of the *viewpoint* has a positive *z* coordinate
all objects have negative *z* coordinates.
lights may be located in either *positive or negative z* space.

Translating from pixel to world coordinates

Problem: Suppose the window is 640 pixels wide x 480 pixels high, and that the dimension of the window in world coordinates is 8 feet wide by 6 feet high. Find the world coordinates of the pixel at column 100 row 40.

Solution: Compute the fraction or percentage of the complete *x* size that must be traversed to reach column 100. This value is 100/640 = 10 / 64 meaning column 100 is 10/64 of the way across the window. The *x* world coordinate of this location is therefore 10/64 of the total world distance across the window or (10/64)*8 = 10/8 = 1.25. Similarly the world *y* coordinate is (40/480) * 6 = (1 / 12) * 6 = 0.5.

A general formula for the procedure is thus:

*world_x = world_size_x * win_x / (win_size_x)*

Thus the desired world coordinate is (1.25, 0.5, 0). (Recall the screen lies in the *z = 0* plane. Therefore the *z* world coordinate of every point in the window is 0.). WARNING: *Pixel dimensions are stored as integers.* You *must* ensure that the divisions shown above are done in floating point.

An alternative “world view”

If the above approach is used, then the pixel with x coordinate 0 clearly maps to world coordinate 0 as it apparently should. But if we are constructing a 640 pixel image, the maximum pixel coordinate is thus 639. And thus the corresponding world coordinate is:

8 * 639 / 640 = 7.988 instead of 8.

We can fix that by changing

*world_x = world_size_x * win_x / (win_size_x - 1)*

In this way pixel coordinate 0 maps to world coordinate 0 and pixel coordinate 639 maps to world coordinate 8. But then “nice” pixel coordinates such as 40 and 100 now map to really ugly numbers slightly larger than 1.25 and 0.5! Furthermore the image has no “center” pixel that maps to world coordinate (4.0, 3.0, 0.0)!

We can get back our “nice” numbers and our center pixel by using the above strategy but always making the image size 1 more than a “nice size” (e.g. 801 x 601). Since the computer doesn't really care whether a number is ugly or nice, we will use this formulation.

*world_x = world_size_x * win_x / (win_size_x - 1)*

Computing the direction of a ray

Problem: Suppose the viewpoint is at location (4, 3, 6) in world coordinates. Compute a unit length vector from the viewpoint through the pixel at column 100 row 40.

Solution: We saw above that the world coordinates of the pixel is: (1.25, 0.5, 0). From page three we know that two points in 3-D space implicitly determine a vector pointing from one to the other. Given two points P and Q in 3-D Euclidean space, the *vector*

$$R = P - Q = (p_x - q_x, p_y - q_y, p_z - q_z)$$

represents the direction *from Q to P*. Therefore the vector *from* the viewpoint *to* the point on the window is *(point – viewpoint)* or:

$$(1.25,\ 0.5,\ 0) - (4, 3, 6) =$$
$$(1.25 - 4, 0.5 - 3, 0 - 6) = (-2.75, -2.50, -6.00)$$

The length of this vector is 7.06 and so a unit length vector in this direction is:

$$(-0.39,\ -0.35,\ -0.85)$$

If you have computed the direction correctly the *z* component of the vector will always be negative. A good plan is therefore to include the line:

```
assert(direction->z < 0);
```

The assert facility will *abort your program i*f the condition is FALSE and will print the module and line number where the problem happened.

You might be tempted to also do:

```
assert(vec_len(direction)) == 1.0);
```

but because floating point arithmetic is imprecise that would not be a good idea.

The raytracing algorithm

The complete algorithm for the first version of the raytracer is summarized below:

Phase 1: Initialization

```
acquire camera data from the stdin and allocate pixmap data
dump camera data to the stderr

load material, object,  and light descriptions from the stdin
dump material, object and light descriptions to the stderr
```

Phase 2: The raytracing procedure for building the pixmap

```
for each pixel in the window
{
        initialize the color of the pixel to (0.0, 0.0, 0.0)
        compute  the direction in 3-d space of a ray from the viewpoint through the pixel
        identify the first (closest) object hit by the ray
        make a copy of the ambient color of the material associated with the object
        scale the copy of the ambient color by 1.0 / distance_from_viewpt_ to_ hitpt
        add the scaled value to the color of the pixel.
        convert the d_rgb pixel to i_rgb and store it in the pixmap.
}
```

Phase 3: Writing out the pixmap as a .ppm file

```
write .ppm header to stdout
write the image to stdout
```

Example input file and image

```
camera cam1
{
  pixeldim  640 480
  worlddim  8 6
  viewpoint 4 3 6
}

material green
{
   ambient 0  5 0
}

material yellow
{
   diffuse  4  4 0
   ambient  5  4 0
   specular 1  1 1
}

plane leftwall
{
   material green
   normal 3 0 1
   point  0 0 0
}
plane rightwall
{
   material yellow
   normal -3 0 1
   point   8  0 0
}
material gray
{
   ambient 2 2 2
}

plane floor
{
   material gray
   normal 0 1 0
   point 0 -0.2 0
}
```

camera, material, and plane are type names (analogous to *int, char, float).* Only defined type names are legal in this context.

cam1, green, and leftwall are instance names analogous to variable names in C. Any name may be used here.

Reflectivity (i.e. color) of objects is specified as (r, g, b) values. Thus 0 5 0 is green.

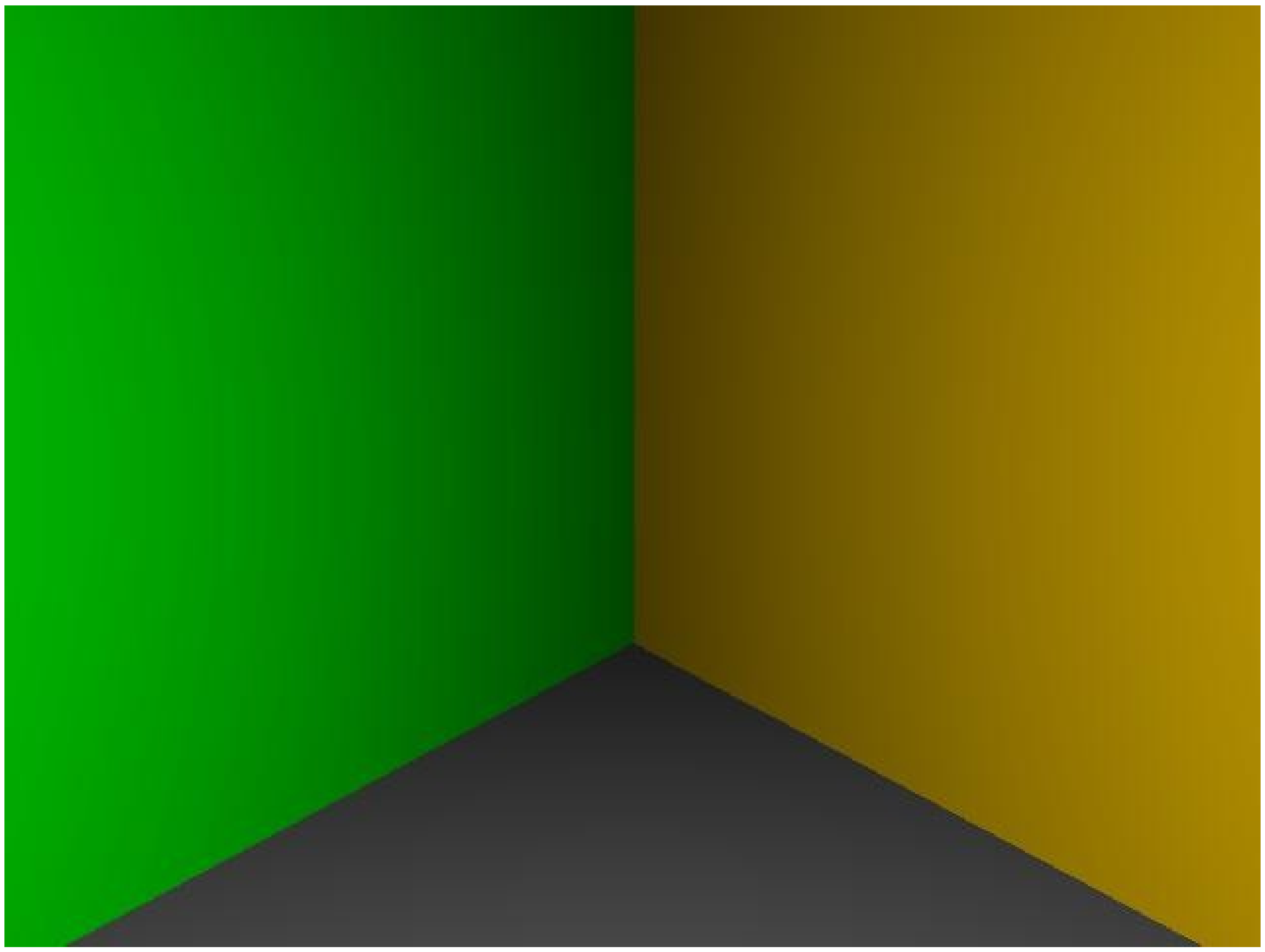
Our lighting model assumes 3 types of reflectivy: ambient, diffuse, and specular. Initially only ambient will be implemented.

Plane definitions *must* contain the three attributes shown. *Materials* must be defined before they are referenced. The value of *point* is the (x, y,z) coordinates of any point on the plane. The value of *normal* is a vector perpendicular to the plane.

The output image

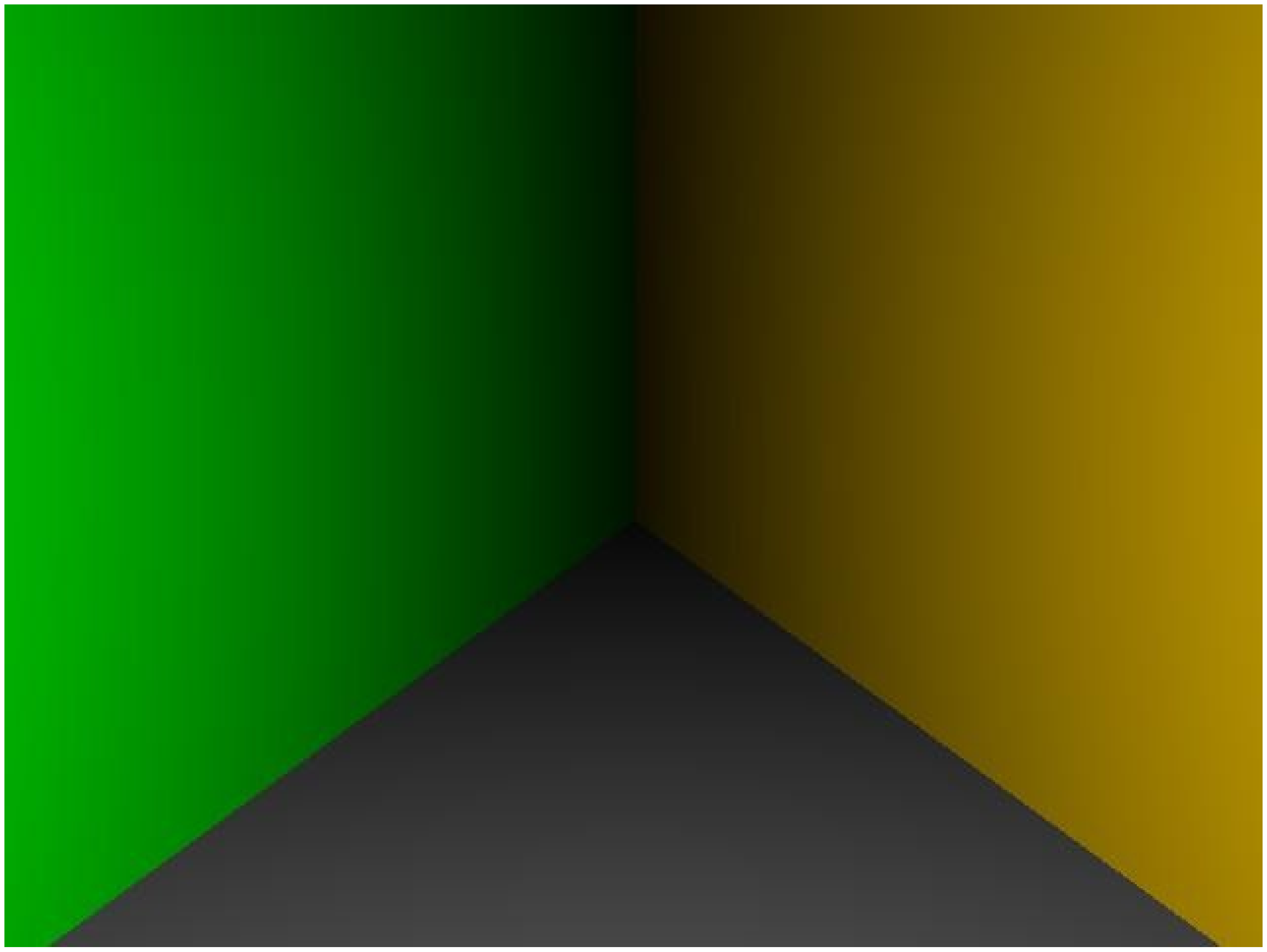
The output produced by the input file on the previous page is shown below. Visible image corruption near the green-gray boundary courtesy of JPEG compression.

The color gradient (which is what provides the “three-D” effect) is achieved by dividing the base ambient reflectivity of the object (0 5 0) by the distance from the view point to the location at which the ray hits the object. Pixels near the green – yellow boundary are more distant from the view point than those near the edges of the images.



We can push the point of intersection of the planes even farther into negative z-space by reducing the z component of the normal from 1 to 0.1.   When we do this, the floor triangle becomes larger, and the intersection of the two plains becomes indistinct.

```
plane leftwall
{
   material green
   normal 3 0 0.1
   point  0 0 0
}
plane rightwall
{
   material yellow
   normal -3 0 0.1
   point   8  0 0
}
```



The camera data structure

The *typedef* facility can be used to create a new identifier for a user defined type. The following example creates a new type name, *cam_t,* which is 100% equivalent to *struct camera_type*.
You may either use or not use *typedef* as you see fit.

A structure of the following type can be used to hold the view point and coordinate mapping data that defines the projection onto the virtual window:

```
#define NAME_LEN   16
#define CAM_COOKIE 23987237

typedef struct camera_type
{
   int    cookie;          /* ID's this as a camera         */
   char   name[NAME_LEN];  /* User selected camera name     */
   int    pixel_dim[2];    /* Projection screen size in pix */
   double world_dim[2];    /* Screen size in world coords   */
   vec_t  view_point;      /* Viewpt Loc in world coords    */
   irgb_t *pixmap;         /* Build image here              */
}  cam_t;
```

The *CAM_COOKIE* value is a completely arbitrary quasi-random identifier that can be used in conjunction with the *assert()* mechanism to detect:

- defective *cam_t* * pointers
- *cam_t* structures that have been corrupted via other pointer errors.

When a camera structure is created the cookie should be initialized

- *cam->cookie = CAM_COOKIE*

When a function is passed an alleged pointer to a *cam_t* it should be verified

- *assert(cam->cookie == CAM_COOKIE);*

If the value of the expression passed to *assert()* is *false()* *the program is aborted* and a message issued which provides the module name and line number at which the error was detected.

Specifying the camera data

The camera configuration and the scene data will be read from the standard input. The camera data *will always be first* and will have the following format.

- The keyword *camera* is an *entity-type* that identifies this as a camera specification.
- Other *entity-type* will include *material, light, plane, sphere, etc.*
- It is followed by user defined and arbitrary *entity-name* (camera name).
- *entity-types* are analogous to data types in C (*int, float, double*)
- *entity-names* are analogous to variable names in C (*max, min, pixel)*
- The attributes of the camera will always be enclosed in { }.
- For the camera, three attributes will always be specified.

    - The *entity-types* and *attribute-types* are *predefined keywords* and will always be spelled as shown.
    - The attribute values will always *follow the attribute type*.
    - The number of values of a particular attribute *will never vary*.
    - The attributes of the *camera* may appear in *any order.*

```
camera cam1
{
  pixeldim  800 600
  worlddim  8 6
  viewpoint 4 4 6
}
```

The attribute values map to the *cam_t* structure in the obvious way:

```
   int    pixel_dim[2];    /* Projection screen size in pix */
   double world_dim[2];    /* Screen size in world coords   */
   vec_t  view_point;      /* Viewpt Loc in world coords    */
```

Parsing the input file

Parsing is a process in which an input file containing "sentences" written in some language is:

- read in from a file
- tokenized
- analyzed

The semantics of the language determine the actions that are taken during the analysis. Some languages (e.g. the C programming language) are quite complex and some formal mechanisms are needed to process them. Our input language is simple enough that informal ad hoc methods suffice.

A *token* is a "word" in the language. In this input:

```
camera cam1
{
  pixeldim  800 600
  worlddim  8 6
  viewpoint 4 4 4
}
```

`camera, cam1, {, pixeldim, 800, 600,` etc are tokens. The individual letters making up the words and the digits making up the numbers are not. If (and only if) the language is structured rigidly enough that the position in a sentence in which *string* values and *numeric* values can be known in advance, then *fscanf()* can be used as a combination reader/tokenizer.

- %s token is a string of 1 or more characters
- %d token is an integer value
- %lf token is a double precision value

This will be the case for the raytracer.

Model description language

In summary, our model description language looks informally like:

```
entity-type entity-name
{
    attribute-type attribute-value(s)
    attribute-type attribute-value(s)
    :
}

entity-type entity-name
{
    attribute-type attribute-value(s)
     :
}

entity-type entity-name
{
    attribute-type attribute-value(s)
      :
}
```

That is,

- the attribute list of each entity definition is *terminated by the } token* and
- the complete model definition is *terminated by end-of-file.*

Each entity-type will provide a constructor function that will know about those attributes that apply to the entity will and be responsible for them.

Entity constructors should use the assert() mechanism to abort the program whenever:

- an unknown attribute type is encountered
- the proper number of attribute values cannot be read in
- required attributes are missing

To make life easier we will also constrain the order in which *entity-types* may appear. The camera must be first.

We will also constrain, to some degree, the order in which *attribute-types* may appear.

An example parser

Suppose a gaming system contains objects of type *aircraft*.

```
#define AC_COOKIE 12345932    // ids an aircraft_t
#define AC_MASK   7           // required item bitmask

typedef struct aircraft_type
{
   int    cookie;          // must have value AC_COOKIE
   int    attrmask;        // attribs found
   char   name[NAME_LEN];  // name of this aircraft
   vec_t  position;        // current location
   vec_t  direction;       // direction of travel
   double speed;           // speed in knots
}  aircraft_t;
```

A similar input language is used so that the input looks like:

```
aircraft F18-1
{
   position  20000 30000 300000
   direction 1 0 1
   speed     720
}
```

Creating and initializing a new aircraft structure.

A new structure of type *aircraft_t* might be created and initialized as follows;

```
aircraft_t *aircraft_init(
FILE *in)
{
   char  buf[256];
   int   count;

/* Create new aircraft structure */

   aircraft_t *aircraft  = malloc(sizeof(aircraft_t));
   assert(aircraft != NULL);
   memset(aircraft, 0, sizeof(aircraft_t));
   aircraft->cookie = AC_COOKIE;

/* Verify the entity name */

   fscanf(in, "%s", buf);
   assert(strcmp(buf, "aircraft") == 0);

/* Acquire the name of the aircraft */

   fscanf(in, "%s", aircraft->name);

/* consume the "{" and verify we found it */

   fscanf(in, "%s", buf);
   assert(buf[0] == '{');
```

Loading attributes

Now the attributes are loaded. They may be specified in arbitrary order. The alleged attribute name is read here but is processed in *aircraft_attr_load();*

```
/* load required attributes */

   count = fscanf(in, "%s", buf);
   while ((count == 1) && (buf[0] != '}'))
   {
      aircraft_attr_load(in, aircraft, buf);
      *buf = 0;
      count = fscanf(in, "%s", buf);
   }

/* verify the closing '}'  */

   assert(buf[0] == '}');

/* Verify all required attributes loaded */

   assert(aircraft->attrmask == AC_MASK);
   return(aircraft);
}
```

Attribute processing

The sanest way to process attributes that may appear in arbitrary order is to make it somewhat table driven. This generic approach is straightforward to transfer to other object types such as *cameras!*

Here we build a table of 3 pointers. Each pointer is initalized to point to a string which is the name of one of the legal attributes. It is important to understand that this is a *table of pointers* not *a table of strings!*

```
/* Table of legit attributes */

static char *ac_attrs[] =
{
   "position",        /* Current location of aircraft */
   "direction",       /* Direction of travel          */
   "speed",           /* Speed in NM/hr               */
};
#define NUM_ATTRS (sizeof(ac_attrs)/sizeof(char *))
```

The following lookup function can be used to identify which if any attribute is found in the input. It should return the index in the attribute table of the string passed in *target*. This should go in *rayfuns.h*.

```
static inline int getndx(
char *attrs[],        /* Table of attribute names  */
int  count,           /* Number of attribute names */
char *target)         /* name from input           */
{
   int code = -1;
   int i;

   for (i = 0; i < count; i++)
   {
      if (strcmp(target, attrs[i]) == 0)
         return(i);
   }
   return(code);
}
```

Processing a single attribute

This function is called as each attribute name is read. The index of the attribute name is looked up in the table and the index is used to determine which loader to call.

```
static inline void aircraft_attr_load(
FILE  *in,
aircraft_t *aircraft,
char *attr)
{
   char buf[18];
   int  ndx;

/* verify structure pointer */

   assert(aircraft->cookie == AC_COOKIE);

/* Perform table lookup and ensure it worked */

   ndx = getndx(ac_attrs, NUM_ATTRS, attr);
   assert(ndx >= 0);

/* Remember which attribute was found */

   aircraft->attrmask |= 1 << ndx;

   if (ndx == 0)
      aircraft_load_position(in, aircraft);
   else if (ndx == 1)
      aircraft_load_direction(in, aircraft);
   else
      aircraft_load_speed(in, aircraft);
}
```

Obtaining the values of a single attribute

This approach is fairly code intensive, requiring a small blob of code for each an every element of each and every structure!   An advantage is that it is straightforward and easy to understand,   but a more clever fully table-driven approach could be constructed that would avoid *ad hoc* functions such as this one entirely.

```
static inline void aircraft_load_position(
FILE       *in,
aircraft_t *ac)
{
   int count;
   assert(ac->cookie == AC_COOKIE);

   count = fscanf(in, "%lf %lf %lf",
                  &ac->position.x, &ac->position.y,
                  &ac->position.z);

/* ensure that the required number of values were found */

   assert(count == 3);
}
```

Ray tracer data structures: the *ray.h* header file

A common technique in building large programs is to consolidate all required header files into a single header file that can be conveniently included by all source modules. The file *ray.h* will contain most of the important data structures of the ray tracer and will also include the header files needed by all of the modules comprising the system.

```
/* ray.h */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <memory.h>
#include <assert.h>

#define NAME_LEN   16      /* max len of entity/attr names */

#define OBJ_COOKIE 12345678  /* quasi-random cookie values */
#define MAT_COOKIE 32456123  /* used to verify that struct */
#define LGT_COOKIE 30492344  /* pointers are what they     */
#define CAM_COOKIE 49495923  /* pretend to be!             */

#include "vector.h"        /* vec_t and vector functions */

typedef struct camera_type
{
   int    cookie;
   char   name[NAME_LEN];
   int    pixel_dim[2];    /* Projection screen size in pix */
   double world_dim[2];    /* Screen size in world coords   */
   vec_t  view_point;      /* Viewpt Loc in world coords    */
   irgb_t *pixmap;         /* Build image here              */
}  cam_t;

:
list_t, object_t, material_t, light_t, drgb_t, irgb_t etc.
:

#include "rayfuns.h"       /* generic inline functions */
#include "rayhdrs.h"       /* prototypes for intermodule calls */
```

An alternative approach

```
/* ray.h */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <memory.h>
#include <assert.h>

#define NAME_LEN   16      /* max len of entity/attr names */

#define OBJ_COOKIE 12345678  /* quasi-random cookie values */
#define MAT_COOKIE 32456123  /* used to verify that struct */
#define LGT_COOKIE 30492344  /* pointers are what they     */
#define CAM_COOKIE 49495923  /* pretend to be!             */

#include "vector.h"        /* vec_t and vector functions */
#include "list.h"
#include "camera.h"
#include "object.h"
#include "plane.h"

/*  Still have to come last !!! */

#include "rayfuns.h"       /* generic inline functions */
#include "camhdrs.h"       /* prototypes for intermodule calls */
#include "listhdrs.h"      /* prototypes for intermodule calls */
#include "objhdrs.h"      /* prototypes for intermodule calls */
```

Neither way is “right” or “wrong”. Factors that influence the choice would be the size of the team working on the program and the personal preference of the programmer.

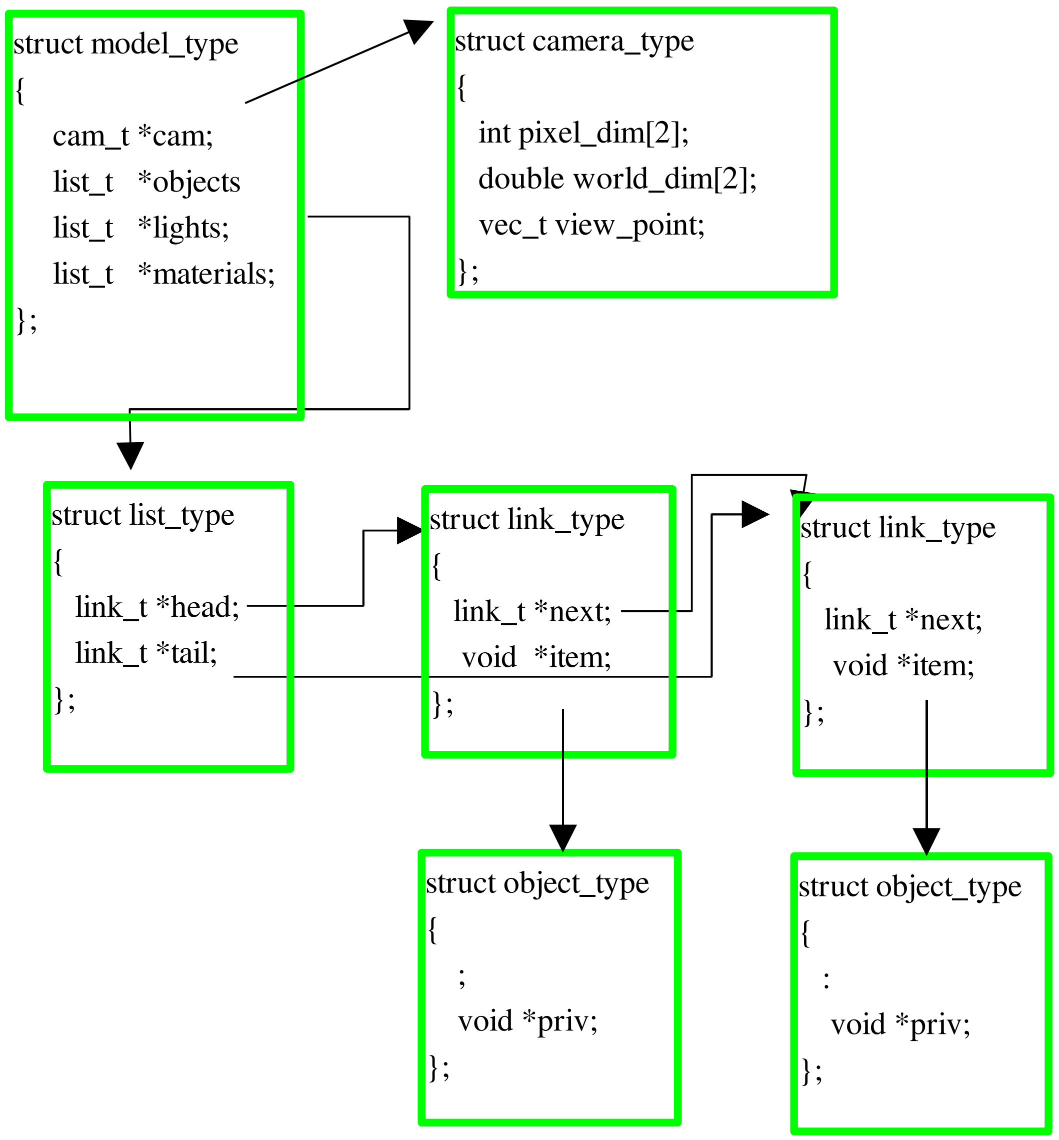
I use the approach shown on the *previous* page because it makes it easy to look at all of my data structures all at one time, but you are free to do it any way you wish.

The one approach I *DO NOT* recommend is including a giant list of header files in *every* source module.   That just makes for unnecessary work when you need to change the list.

Data structures - the big picture

The data structures shown below will all be defined in *ray.h*

WARNING: Some elements of the definitions have been have been abbreviated and or assume the use of the *typedef* construct.   See the examples on other pages for these details.



List management functions

The characteristics of the lists used by the raytracer include the following:

1 - newly created objects are always added to the end of the list
2 - objects are never deleted from the list
3 - lists are always processed sequentially from beginning to end
4 - there is a need for three lists (materials, visible objects, and lights)

Therefore a singly linked list suffices nicely, and a list may be associated with link and list header structures of the type shown below. As usual these definitions go in *ray.h*.

```
typedef struct link_type
{
   link_t    *next; /* next link in the list        */
   void      *item; /* the item (object_t, light_t) */
                    /* that this link owns          */
}  link_t;


typedef struct list_type
{
   link_t   *head; /* pointer to first object in list */
   link_t   *tail; /* pointer to last object in list  */
}  list_t;
```

List management functions

The list management module requires only two functions.   The *list_init()* function is used to create a new list. Its mission is to:

1 - *malloc()* a new *list_t* structure.
2 - set the *head* and *tail* elements of the structure to NULL.
3 - return a pointer to the *list_t* to the caller.

```
list_t *list_init(
void)
{

}
```

The *list_add()* function must add the element pointed to by *new* to the list structure pointed to by *list.* Two cases must be distinguished:

1 - the list is empty (list-> head == NULL)
2 - the list is not empty

The *list_add()* function must *malloc()* a new instance of *link_t,* add it to the tail of the list, ensure the *next* pointer of the new link is NULL and ensure the *next* pointer of the *link_t* that used to be at the end of the list points to the new *link_t*

```
void list_add(
list_t   *list,
void     *new)
{

}
```

The *list_del()* function. This function should process the entire list. For each link in the list, it should invoke the *free()* function to free the *item* the link owns and then its should free the *link_t*. Care must be taken *not to reference* a *link_t* after it has been freed. When all links and items are free the *list* header itself should be freed.

```
void list_del(
list_t    *list,
{

}
```

Processing a list

```
   list_t *elist;
   link_t *elink = elist->head;
   while (elink != NULL)
   {
      eloc = (e_t *)elink->item;
      printf("%s %d \n", eloc->ename, eloc->ecount);
      elink = elink->next;
   }
```

The *model_t* data structure

This structure is a *container* used to reduce the number of parameters that must be passed through the raytracing system. It lives in *ray.h.*

```
typedef struct model_type
{
   cam_t   *cam;    // The camera structure
   list_t  *mats;   // The head of the material list
   list_t  *objs;   // The head of the visible obj list
   list_t  *lgts;   // The head of the light list
}  model_t;
```

The main function

A properly designed and constructed program is necessarily *modular in nature.* Modularity is somewhat automatically enforced in O-O languages, but new C programmers often revert to an ugly pack- it- all- into- one-*main*- function approach.

To discourage this in the *raytracing* program, deductions will be made for:

1 - Functions that are too long (greater than 30 lines)
2 - Nesting of code greater than 2 deep (NO nested loops)
3 - Exception to 2: Its OK to have an *if* inside a loop.
4 - Lines that are too long (greater than 72 characters)

Here is the main function for the *final version* of the ray tracer.

```
/* main.c */

#include "ray.h"

int main(
int argc,
char *argv[])
{
   cam_t   *cam;
   model_t *model;

/* Load and dump camera data */

   cam = cam_init(stdin);
   assert(cam != NULL);
   cam_dump(stderr, cam);

/* Load and dump the model */

   model = model_init(stdin);
   model->cam = cam;
   model_dump(stderr, model);

/* Raytrace the image */

   image_create(model);
   return(0);
}
```

The generic object structure

Even though C is technically not an Object Oriented language it is possible to employ mechanisms that emulate both the inheritence and polymorphism found in true Object Oriented languages.

The *object_t* structure serves as the generic "base class" from which the esoteric objects such as *planes* and *spheres* are derived. As such, it carries only the attributes that are common to the all derived objects. Esoteric attributes of a *plane* are carried by a *plane_t* structure. Esoteric attributes of a *sphere* are carried by a *sphere_t* structure. The *priv* pointer of the *object_t* provides a link to the *plane_t* or *sphere_t* and is thus declared as *void* *.

Polymorphic behavior is achieved by the use of *function pointers* embedded in the *object_t.* These can be initialized to point to functions that provide a *default* behavior but may be overridden as needed when an esoteric object such as a *tiled plane* must substitute its own "method". Fields shown in *blue* are required for the first version of the ray tracer.

```
typedef struct object_type
{
   int     cookie;
   char    objname[NAME_LEN];  /* left_wall, center_sphere */
   char    objtype[NAME_LEN];  /* plane, sphere, ...       */

   double  (*hits)(vec_t *base, vec_t *dir, struct object_type *);
                           /* Finds if/where ray hits object   */
   void    (*dumper)(FILE*, struct object_type *);
                           /* Prints object attributes */

/* Optional plugins for retrieval of reflectivity */
/* useful for the ever-popular tiled floor        */

// void    (*getamb) (struct object_type *, drgb_t *);
// void    (*getdiff)(struct object_type *, drgb_t *);
// void    (*getspec)(struct object_type *, drgb_t *);

/* Surface reflectivity data */

   material_t *mat;      /* Primary color of the object */

   void    *priv;         /* Pointer to type dependent data */
   vec_t   hitloc;        /* Last hit point                 */
   vec_t   normal;        /* Normal at last hit point       */
} object_t;
```

Declaration of derived object types

The esoteric characteristics of specific object types must be carried by structures that are specific to the object type being described. The *priv* pointer of the base class *object_t* is used to connect the generic instance to the esoteric instance. This connection is automatic and invisible in a true OO language but is *manual* and *visible* in C.

Notice that the process of refinement or specialization can continue over multiple levels. The *priv* pointer of the plane structure may point to an *fplane* (bounded rectangular plane) structure.

```
/* This structure carries the attributes */
/* of an infinite (unbounded) plane */

typedef struct plane_type
{
   vec_t   normal;        /* vector perpendicular to plane   */
   vec_t   point;         /* point on the plane              */
   double  ndotq;         /* dot product of normal and point */
   void    *priv;         /* Data for specialized types      */
}  plane_t;


/* Sphere */

typedef struct sphere_type
{
   vec_t   center;
   double  radius;
}  sphere_t;
```

Loading the model description

The raytracer must be able to read model descriptions of the format shown below:

- Camera data will always appear first

Camera data will be loaded by *cam_init()*. This design decision by me is largely motivated by *my* need for *you* to incrementally build the program. (But it could be claimed that the camera is one piece of the puzzle and the scene is logically another).

For the scene specification

- *material, object, and light* data may be intermixed
- *materials* must be defined before being referenced in an object definition
- attributes may appear in any order.
- missing attributes must be set to zero.

```
camera cam1
{
  pixeldim  640 480
  worlddim  8 6
  viewpoint 4 3 6
}

material green
{
   diffuse 0  1 0
   ambient 0  5 0
}

plane leftwall
{
   normal 3 0 0.2
   point  0 0 0
   material green
}
plane rightwall
{
   material green
   point   8  0 0
   normal -3 0 0.2
}
```

The *model_init()* function

This function be contained in the source module *model.c.* It controls the loading of all *material*, *object*, and *light* definitions. When it completes, all material, object and light definitions specified in the input file will have been read, and for each material, object or light definition in the file a new structure of type *material_t, object_t,* or *light_t* must reside on the appropriate list. (If the input file contains the definition of *three* planes, *three object_t's must be on the objs list.)*

```
typedef struct model_type
{
   cam_t    *cam;    // The camera structure
   list_t   *mats;   // The head of the material list
   list_t   *objs;   // The head of the visible obj list
   list_t   *lgts;   // The head of the light list
}  model_t;
```

Implementing the *model_init_function()*

Specifically it must: *malloc()* the *model_t* structure and set it to zero, call *list_init()* three times to set up the *material, object, and light* lists, call an internal *model_load* function to load the model data, and then return the address of the *model_t* structure.

```
/**/
/* Init model data */

model_t *model_init(
FILE *in)
{
   model_t *model = malloc(sizeof(model_t));
   assert(model != NULL);
   memset(model, 0, sizeof(model_t));

/* Create and initialize material structure list */

   model->mats = list_init();
   assert(model->mats != NULL);
```

Step one is to *create* the lists. This step READS NO INPUT DATA.

```
/* Create and initialize visible object structure list */

   ...............

/* Create and initialize light structure list */

   ..............
```

Step two is to *read and store* the model data.

```
/* read in the materials, objects, lights */

   model_load(in, model);
   return(model);
}
```

The design of the model loader

The model loader should operate in a way similar to but not exactly like the camera loader. In fact it is simpler than the camera loader because it depends upon entity specific loaders to

- create object and material structures and
- read and store attribute data

```
static char *entities[] =
{
    "material",
    "plane",
//  "light",
//  "sphere",
};

#define NUM_ENTITIES (sizeof(entities) / sizeof(char *))

static void model_load(
FILE    *in,
model_t *model)
{
   char entityname[16];
   int  count;

   memset(entityname, 0, sizeof(entityname));

/* Here entityname should be one of "material",    */
/* "light", "plane"                                */

   count = fscanf(in, "%s", entityname);
   while (count == 1)
   {

      lookup entity name in entities table
      invoke entity specific initializer (material_init,
           plane_init, light_init, sphere_init)

      count = fscanf(in, "%s", entityname);
   }
}
```

The *model_dump* function

This function just drives the process of producing a nicely formatted version of the contents of the material list, the object list, and the light list.   The entity-type specific functions shown are responsible for the details.

```
/**/
/* dump model data */

void model_dump(
FILE    *out,
model_t *model)
{
   material_dump(out, model);
   object_dump(out, model);
   light_dump(out, model);
}
```

Material definitions

Each generic object must be associated with one *material_type* which defines the way the surface of the object interacts with light in the scene. At the simplest level, the material definition can be thought of as specifying the color of the object in *(r, g, b)* units.

```
typedef struct material_type
{
   int     cookie;          /* ID of a material_t          */
   char    name[NAME_LEN];  /* light_blue for example      */
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
}  material_t;
```

There are three components to the light interaction model:

- *ambient* – specificies how the object reflects light that is present in the scene but is *not* emanating from any particular light source.

- *diffuse* – specifies how the object reflects light that *does* emanate from specific light sources

- *specular* – specifies the degree to which the object acts like a mirror (incoming light is precisely reflected (instead of being diffused) with the angle of incidence being equal to the angle of reflection).

It is possible to create models that are physically unrealizable. We can define an object that reflects ambient light as red and diffuse light as green! But no physical object exists that operates in such a way.

Creating a new material entity

Material definitions define the reflectivity and hence the color of visible objects.

```
/**/
/* Create a new material description */

void material_init(
FILE        *in,
model_t     *model,
int         attrmax)
{
   material_t *mat;
   char attrname[NAME_LEN];
   int count;

   malloc and initialize a material structure
   read material name (lightblue, etc)
   read and verify {

   read attribute name
   while (attribute-read-successfully and
               attrname[0] is not '}')
   {
       material_attr_load(in, mat, attrname);

       read attribute name
   }
   verify '}' was found
   add the material structure to the material list
}
```

Loading material attributes

Legal attribute names are *ambient, diffuse,* and *specular.* At least one attribute must be specified. Each attribute has three values: the red, green, and blue reflectivity.

```
static int material_attr_load(
FILE    *in,
material_t *mat,      /* material to be filled in */
char    *attrname)
{
   lookup attrname in attribute name table
    assert(ndx >= 0);
    if (ndx == 0)
       read ambient values
    else if ....

    else

}
```

Pointers to functions

Pointer variables may also hold the address of a function and be used to invoke the function indirectly:

```
class/215/examples ==> gcc p32.c
class/215/examples ==> cat p32.c

/* p32.c */

#include <stdio.h>
int adder(
int a,
int b)
{
   return(a + b);
}

int main()
{
   int (*ptrf)(int, int);  // declare pointer to function
   int sum;

   ptrf = adder;           // point it to adder (note no &)
                           // is needed (but it doesn't hurt))

   sum = (*ptrf)(3, 4);    // invoke it
   printf("sum = %d \n", sum);
   return(0);
}

class/215/examples ==> a.out
sum = 7
class/215/examples ==>
```

Function pointers as do-it-yourself polymorphism

Recall the the *object_t* structure containes a function pointers:

```
typedef struct object_type
{
   int     cookie;
   char    objname[NAME_LEN];  /* left_wall, center_sphere */
   char    objtype[NAME_LEN];  /* plane, sphere, ...       */

   double  (*hits)(vec_t *base, vec_t *dir, struct object_type *);
                                 /* Hits function.             */
   void    (*dumper)(FILE*, struct object_type *);
    :

}
```

The *hits* pointers must be set in the initialization module. In the *plane_init* function this pointer should be set as follows:

```
   obj->hits = plane_hits;
   obj->dumper = plane_dump;
```

All of the *hits* functions have the same parameters

```
double     plane_hits(
vec_t      *base,     /* Start point of ray  */
vec_t      *dir,      /* MUST be unit vector */
object_t   *obj)      /* Candidate object    */
```

Invoking a hits function

```
    dist = obj->hits(base, dir, obj);
```

Function pointers as a mechanism for simplifying model loading.

There are a number of possible ways to control the loading of model data:

The most straightforward way is:

```
if (strcmp(entity_type, "material") == 0)
   material_init(in, model, 0);
else if (strcmp(entity_type, "plane") == 0)
   plane_init(in, model, 0);
else if (strcmp(entity_type, "light") == 0)
   light_init(in, model, 0);
else if ......
```

A table driven approach

A first cut at an alternative data driven approach might consist of two tables.

- The names of the entities are kept in one
- Pointers to initialization functions are kept in another
- Our friend *getndx()* is used to search the name table.

```
static char *items[] =
{
    "material",
    "plane",
    "light",
    "tiled_plane",
    "sphere",
};
#define NUM_ITEMS (sizeof(items) / sizeof(char *))

static void (*loaders[])(FILE *in, model_t *model, int max) =
{
   material_init,
   (void *)plane_init,
   (void *)light_init,
   (void *)tplane_init,
   (void *)sphere_init,
};

static inline void model_item_load(
FILE    *in,
model_t *model,
char    *itemtype)
{
   int ndx;

   ndx = getndx(items, NUM_ITEMS, itemtype);
   assert(ndx >= 0);
   (*loaders[ndx])(in, model, 0);
   return;
}
```

A refinement to the table driven approach

Note the the table driven approach works only if the two tables are kept "in sync". The address of each entity initializer function must be in the corresponding position to the entity name. We can mitigate the problem somewhat by creating a structure that contains *both* the address of the entity name string and the function that loads it.

```
typedef struct init_type
{
   char *entity_name;
   void (*loader)(FILE *, model_t *, int);
}  init_t;

static init_t items[] =
{
   {"material",      material_init},
   {"plane",        (void *)plane_init},
   {"light",        (void *)light_init},
   {"tiled_plane",  (void *)tplane_init},
   {"sphere",       (void *)sphere_init},
};
#define NUM_ITEMS (sizeof(items) / sizeof(init_t))
```

If we do this, the indirect invocation mechanism remains more or less the same, but note that the newly structured table can no longer be used with the original *getndx()*.

```
static inline void model_item_load(
FILE    *in,
model_t *model,
char    *entitytype)
{
   int ndx;

   ndx = getndxi(items, NUM_ITEMS, entitytype);
   assert(ndx >= 0);

   (*items[ndx].loader)(in, model, 0);

   return;
}
```

The new *getndxi* function.

```
static inline int getndxi(
init_t *inittab,      /* entity initializer table */
int     count,        /* number of entities       */
char *target)         /* candidate entity name    */
{
   int i;
   init_t *ie = inittab;

   int rc = -1;

   for (i = 0; i < count; i++)
   {
      if (strcmp(target, ie->entity_name) == 0)
         return(i);
      ie += 1;
   }
   return(rc);
}
```

The *plane_init()* function

The *plane_init()* function has several responsibilites:

```
object_t *plane_init(
FILE *in,
model_t *model,
int  attrmax)
{
   plane_t  *pln;
   object_t *obj;
```

- invoke *object_init()* to create and initialize an *object_t* structure
- create a *plane_t* structure itself
- load the specialized attributes of the plane_t (*normal* and point)
- set the *priv* pointer in the *object_t* to point to the *plane_t*
- initialize the *hits* and *dumper* function pointers in the *object_t*
- store the object type name ("*plane*") in the *object_t* structure

The *object_init* function

```
object_t   *object_init(
FILE       *in,
model_t    *model)
{
   object_t   *obj;
   material_t *mat;

   char buf[NAME_LEN];
   int count;

/* Create a new object structure and zero it */

   obj = malloc(sizeof(object_t));
   assert(obj != NULL);
   memset(obj, 0, sizeof(object_t));
   obj->cookie = OBJ_COOKIE;

/* Read the descriptive name of the object */
/* left_wall, center_sphere, etc.          */

   count = fscanf(in, "%s", obj->objname);
   assert(count == 1);

/* Consume the delimiter */

   count = fscanf(in, "%s", buf);
   assert(buf[0] == '{');

/* First attribute must be material */

   count = fscanf(in, "%s", buf);
   assert(count == 1);
   count = strcmp(buf, "material");
   assert(count == 0);
   count = fscanf(in, "%s", buf);
   assert(count == 1);

   mat = material_find(model, buf);
   assert(mat != NULL);
   obj->mat = mat;

   list_add(model->objs, (void *)obj);

   return(obj);
}
```

Producing a formatted dump of the object list

```
/**/
void object_dump(
FILE *out,
model_t *model)
{
   For each object_t in the model->objs list
   {
     print the object's objtype and objname
     print the word material and the name of the object's associated material
         plane         rightwall
         material      yellow
     polymorphically print the type specfic attributes of the object via:
              obj->dumper(out, obj);
   }

}
```

Type specific dumpers

Type specific dumpers know what their own attributes are so they just print attribute names and values.

```
void plane_dump(
FILE *out,
object_t *obj)
{
   plane_t *pln;

   Recover the pln pointer using the priv pointer in the object_t
   Print attribute names and values:

         normal        -1.0   0.0   0.2
         point          7.0   0.0   0.0

}
```

A general attribute parser

After writing parsers for the *camera, material,* and *plane,* the typical programmer will find repeatedly rewriting (almost) the same code tiresome and tedious and will seek a better way. It is *not* bad that the *ad hoc* approach was used initially. The very use of the *ad hoc* helps the programmer see common aspects of the problem and develop a more general solution.   As usual we try to make our solution data driven to the extent possible.

To build a general parser we will build upon the capability of the *getndx* mechanism but replace the old table of attribute names with *new tables that contain not only the attribute name but also sufficient information to allow the general parser to load the values.* Specifically, for each attribute, we need the following information:

- How many values must be loaded (e.g. 2 for pixeldim, 3 for viewpoint)
- How many bytes of storage does each value occupy (4 for pixeldim, 8 for viewpoint).
- What format string should be used to read a value (%d for pixeldim, %lf for viewpoint)
- Where should the first value be stored?

Here we make use of the fact that we know adjacent array elements and members of a structure such as a *vec_t* are stored in adjacent memory locations.   For the *vec_t* if the location of the *x* component is memory address *a,* and then the *y* component is at location *a+8,* and the *z* at location *a+16.*

The *struct pparm_type*

Therefore, each entity will employ a table in which each attribute is represented by a structure of the following type.

```
/* the parse parameter structure */

typedef struct pparm_type
{
   char *attrname;      /* Attribute name                */
   int  numvals;        /* Number of attribute values    */
   int  valsize;        /* Size of attribute in bytes    */
   char *fmtstr;        /* Format string to use          */
   void *loc;           /* Where to store 1st attr value  */
}  pparm_t;
```

Building tables of attribute descriptors

For the *camera* entity. We build the structure as follows:

```
static pparm_t cam_parse[] =
{
   {"pixeldim",  2, sizeof(int),    "%d",  0},
   {"worlddim",  2, sizeof(double), "%lf", 0},
   {"viewpoint", 3, sizeof(double), "%lf", 0}
};
#define NUM_ATTRS (sizeof(cam_parse) / sizeof(pparm_t))
```

Items to note:

- *cam_parse* is an array of three elements
- each element is a structure of type *pparm_t*
- initializers should be enclosed in { }
- the location where the first attribute value should be stored will live in the *cam_t* structure that is eventually allocated with *malloc()*. These values can't be set until after the *cam_t* is *malloc'd.*

For the other entitities such as the *material* entity, the structure is analogous.

```
static pparm_t mat_parse[] =
{
    {"ambient", 3, sizeof(double), "%lf", 0},
    {"diffuse", 3, sizeof(double), "%lf", 0},
      :
};
#define NUM_ATTRS (sizeof(mat_parse) / sizeof(pparm_t))
```

The interface to the general attribute parser

The generalized parser has the following prototype:

```
/* Generalized attribute parser */

int parser(
FILE    *in,          /* input file                */
pparm_t *pct,         /* parser control table      */
int      numattrs,    /* number of attrs in table */
int      attrmax);    /* ignore this for now       */
```

Invoking the parser

Each entity initializer, such as the *cam_init* function, must fill in the *loc* element of each attribute descriptor before calling the parser.

```
   cam_t *cam  = malloc(sizeof(cam_t));
   assert(cam != NULL);
   cam->cookie = CAM_COOKIE;

   fscanf(in, "%s", buf);
   assert(strcmp(buf, "camera") == 0);

   fscanf(in, "%s", cam->name);
   fscanf(in, "%s", buf);
   assert(buf[0] == '{');

   cam_parse[0].loc = &cam->pixel_dim;
   cam_parse[1].loc = &cam->world_dim;
   cam_parse[2].loc = &cam->view_point;

   mask = parser(in, cam_parse, NUM_ATTRS, 0);
   assert(mask == 7);
```

```
/**/
/* Generalized attribute parser */
/* It returns a bit mask in which each possible attribute */
/* is represented by a bit on exit the attributes that    */
/* have been found will have their bit = 1                */

int parser(
FILE    *in,
pparm_t *pct,         /* parser control table               */
int      numattrs,    /* number of legal attributes         */
int      attrmax)     /* Quit after this many attrs if not 0 */
{
   char attrname[NAME_LEN];
   int  attrcount = 0;   /* number of attribs loaded  */
   int  mask = 0;        /* loaded attrib bit mask    */
   int  ndx;             /* ndx of this attrib in pct */

/* One trip is made through this loop for every attribute */
/* processed... Exit from the loop is triggered by '}'    */
/* or if the maximum number of attributes is set, when    */
/* the maximum number have been processed                 */

   fscanf(in, "%s", attrname);
   while (strlen(attrname) && attrname[0] != '}')
   {

   /* Process one attribute */

      ndx = parser_load_attr(in, pct, numattrs, attrname);
      mask |= 1 << ndx;
      attrcount++;

   /* See if its quitting time --  */

      if ((attrmax) && (attrcount == attrmax))
         break;
      *attrname = 0;
      fscanf(in, "%s", attrname);
   }

   if (attrmax != attrcount)
      assert(attrname[0] == '}');
   return(mask);
}
```

Loading the values of a single attribute

```
static int parser_load_attr(
FILE    *in,
pparm_t *pct,        /* parser control table       */
int     numattrs,    /* number of legal attributes */
char    *attrname)   /* attribute name             */
{
   pparm_t *pce;     /* Entry corresp to this attribute */
   int     count = 0;
   unsigned char *loc;  /* where to store value.. have to */
   double  *work;       /* use unsigned char for pointer  */
   int     ndx;         /* arithmetic to work correctly   */
   int     i;

/* getndxp is an updated version of getndx that takes a */
/* parse control table pointer as input.                */

   ndx = getndxp(pct, numattrs, attrname);
   assert(ndx >= 0);

/* Point to the proper entry in the table */

   pce = pct + ndx;   // or pce = &pct[ndx];

/* pce->loc points to where the first value must go */

   loc = (unsigned char *) pce->loc;

/* Attributes may have different numbers of attribute values */
/* for example the viewpoint has three but the pixeldim only */
/* has 2 values.  Each iteration consumes one value          */

   for (i = 0; i < pce->numvals; i++)
   {
      count += fscanf(in, pce->fmtstr, loc);
   // work = (double *)loc;
   // fprintf(stderr, "%s %lf \n", pce->attrname, *work);
      loc += pce->valsize;  // point to next spot
   }
   assert(count == pce->numvals);
   return(ndx);
}
```

Exercise: Design a generic getndx function

Avoiding parsing altogether

In building programs it's often useful to employ temporary skeletal modules that facilitate the building and testing of other components but are ultimately *thrown away at the end of the project.* For example, if this were a team project it would be desirable for the ray tracing team to press on in parallel with the parsing team's activities instead of having to wait on a functional parser.

We can view this exerise as transforming or model specification language to C! In fact the C version looks quite similar to the target language.

```
/* modelstat.c */

/* This module provides a statically defined model that can */
/* be used to test the raytracing system.                   */

#include "ray.h"

material_t mat1 =
{
   cookie: MAT_COOKIE,
   name:  "green",
   ambient: {0, 5, 0},
};

material_t mat2 =
{
   cookie: MAT_COOKIE,
   name:  "yellow",
   ambient: {6, 5, 0},
};

material_t mat3 =
{
   cookie: MAT_COOKIE,
   name: "gray",
   ambient: {4, 4, 4},
};
```

Now we define the plane structures. Again the definitions are quite consistent with the target language.

```
plane_t plane1 =
{
   normal: {3, 0, 1},
   point:  {0, 0, 0},
};

plane_t plane2 =
{
   normal: {-3, 0, 1},
   point:  {8, 0, 0},
};

plane_t plane3 =
{
   normal: {0, 1, 0},
   point:  {0, 0, 0},
};
```

The object structure definitions combine elements of the original input language, but more reflect the actions of the program.

```
object_t object1 =
{
   cookie:   OBJ_COOKIE,
   objname:  "leftwall",
   hits:     plane_hits,
   priv:     (void *)&plane1,
   mat:      &mat1,
};

object_t object2 =
{
   cookie:   OBJ_COOKIE,
   objname:  "rightwall",
   hits:     plane_hits,
   priv:     (void *)&plane2,
   mat:      &mat2,
};

object_t object3 =
{
   cookie:   OBJ_COOKIE,
   objname:  "floor",
   hits:     plane_hits,
   priv:     (void *)&plane3,
   mat:      &mat3,
};
```

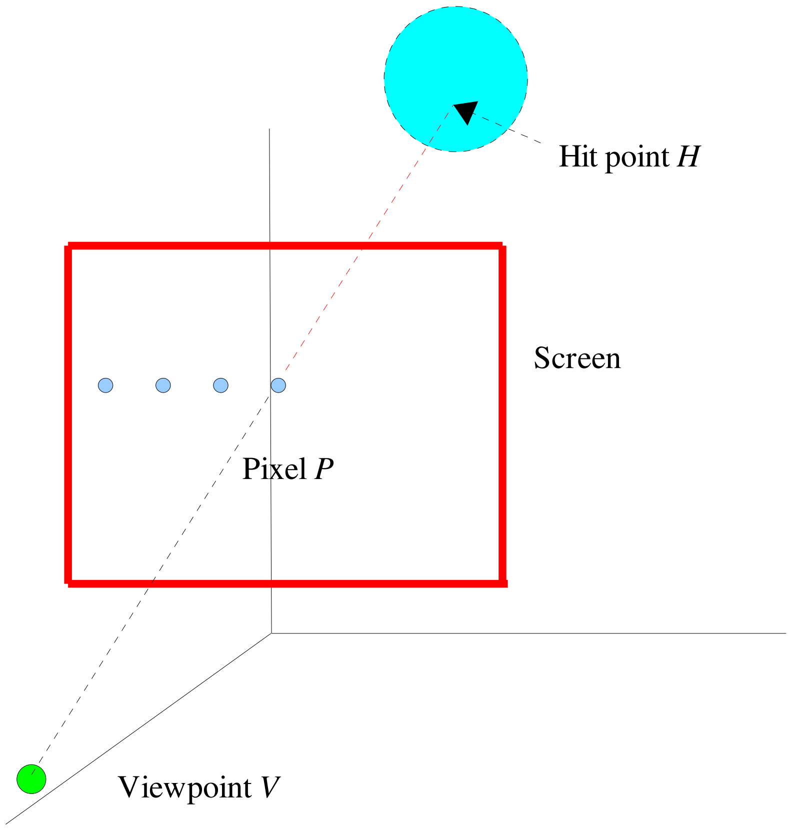
Linking the model together.

The last (and ugliest) piece of the puzzle is to handcraft the object list and put it in the model structure. Note that the *material* list is not necessary because there is on material dumper and no material find needed.

```
link_t link1 =
{
   next: NULL,
   item: (void *)&object2,
};

link_t link2 =
{
   next: &link1,
   item: (void *)&object1,
};

link_t link3 =
{
   next: &link2,
   item: (void *)&object3,
};


list_t list1 =
{
   head: &link3,
   tail: &link1,
};

model_t model =
{
   objs: &list1,
};

model_t *model_init(
FILE *in)
{
   return(&model);
};
```

Hit functions

Given the viewpoint, ray direction and a pointer to an *object_t* the mission of a *hit* function is to determine if the ray hits the object. If it does, the *hit point and the normal vector at the hitpoint* should be stored in the *object_t*.



Given *V*, *D* and an object structure *O* the mission of a hit function is to determine if a ray based at *V* traveling in direction *D* hits *O*.

All points on the ray may be expressed as a function of a single parameter *t* where *t* distance along the ray from the viewpoint. The set of all points on the ray is thus:

$$V + t D \quad \text{for} \quad -\infty < t < \infty$$

Ray direction: $D = (P - V) \;/\; ||P - V||$

Distance to hit point: $t_h = || H - V ||$

Location of hit point: $H = V + t_h D$

Prototype for the hits functions

To determine if a ray hits an object you must add a *hits_objtype* function to your *sphere.c, plane.c etc* modules. These modules should also contain the loading and dumping code for the specific object type. A sample prototype is shown below:

```
double   hits_plane(
vec_t     *base, /* the (x, y, z) coords of origin of the ray   */
vec_t     *dir,  /* unit vector in the (x, y, z) dir of the ray */
object_t *obj); /* the object to be tested for the hit.        */
```

Pointers to the hits' function should be stored in the object structure at the time it is created

```
   obj->hits = hits_plane;
```

General Quadric Surfaces

These surfaces are so named because the variables *x, y, and z* take on at most the power of two. The general equation for the quadric is given below:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz = J$$

We will start with two of the simpler ones:

The sphere:

$$x^2 + y^2 + z^2 = r^2$$

The plane:

$$Gx + Hy + Iz = J$$

where the plane normal is *(G, H, I)* and J is chosen so that the plane passes through the selected point.

Quadric surfaces are “nice” in a raytracing environment because the intersection of a ray with the surface may always be found by solving at worst a quadratic equation.

Determining if a ray hits a plane

This basic strategy will be used in *all* hits functions:

0 - Assume that V represents the start of the ray and D is a *unit* vector in its direction

1 - Derive an equation for an arbitrary point P on the surface of the object.

2 - Recall that all points on the ray are expressed as $V + tD$

3 - Substitute $V + tD$ for P in the equation derived in (1).

4 - Attempt to solve the equation for t.

5 - If a solution t_h can be found, then $H = V + t_h D$.

A plane in three dimensional space is defined by two parameters

A normal vector $N = (n_x, n_y, n_z)$

A point $Q = (q_x, q_y, q_z)$ through which the plane passes.

A point $P = (p_x, p_y, p_z)$ is on the plane if and only if:

$N \text{ dot } (P - Q) = 0$ because, if the two points P, Q lie in the plane, then the vector from one to the other $(P - Q)$ also lies in the plane and thus it is necessarily perpendicular to the plane's normal.

We can rearrange this expression to get:

$$N \text{ dot } P - N \text{ dot } Q = 0$$

$$N \text{ dot } P = N \text{ dot } Q \qquad (1)$$

Note that in this equation *N* and *Q* are known attributes of the plane and P is the unknown. Recall that the the location of any points on a ray based at *V* with direction *D* is given by:

$$V + t\,D$$

Therefore we may replace the *P* in equation (1) by $V + tD$ and get:

$$N \text{ dot } (V + tD) = N \text{ dot } Q \qquad (2)$$

Some algebraic simplification yields allow us to solve this for *t*

$$N \text{ dot } (V + tD) = N \text{ dot } Q \qquad (2)$$
$$N \text{ dot } V + N \text{ dot } tD = N \text{ dot } Q$$
$$N \text{ dot } tD = N \text{ dot } Q - N \text{ dot } V$$
$$t\,(N \text{ dot } D) = (N \text{ dot } Q - N \text{ dot } V)$$
$$t_h = (N \text{ dot } Q - N \text{ dot } V) / (N \text{ dot } D) \qquad (3)$$

The *location of the hitpoint* that should be stored in the *object_t* is thus:

$$H = V + t_h D$$

The *normal at the hitpoint* which must also be saved in the *object_t* is just N

Unlike other quadric surfaces, there is only a single point at which a ray intercepts a plane. Therfore unlike equations we will see later, this one is not quadratic. There *are* some special cases we must consider:

(1) $(N \text{ dot } D) = 0$ In this case the direction of the ray is perpendicular to the normal to the plane. This means the *ray is parallel to the plane.* Either the ray lies in the plane or misses the plane entirely. We will always consider this case a *miss* and return -1. Attempting to divide by 0 will cause your program to either fault and die or return a meaningless value.

(2) $t_h < 0$ In this case the hit lies behind the viewpoint rather than in the direction of the screen. This should also be considered a miss and *-1* should be returned.

(3) The hit lies on the view point side of the screen.

$$H = (h_x, h_y, h_z) \;\; \textit{if} \;\; h_z > 0 \textit{ the hit is on the wrong side}$$

and -1 should be returned.

A makefile for a multi-module program:

The Unix *make* program is a handy utility that can be used to build things ranging from programs to documents. Elements of significance include:

targets	labels that appear in column 1 and are followed by a the character ":" . The *make* command can take a target as an operand as in *make ray.*
dependencies	are files that are enumerated following the name of the target. If any dependency is newer than the target, the target will be rebuilt.
rules	are specified in lines following the target and specify the procedure for building the target. Rules *must* start with a *tab character*. In the example below the tab has been expanded as spaces *but you may not enter spaces.*

The following *makefile* can be used build the executable ray tracer named *ray* (assuming that it requires only the .o files enumerated in the command).

```
a.out: main1.o model.o camera.o list.o material.o plane.o \
       object.o sphere.o \
       vector.h ray.h rayfuns.h rayhdrs.h
        gcc -Wall -g *.o -lm

.c.o: $<
        -gcc -c -Wall -c -g $<  2> $(@:.o=.err)
        cat $*.err
```

The target .c.o: is called a *suffix rule*. It is telling *make* to use the commands that follow whenever it needs to make a .o file from a .c file.

There are a number of predefined macro based names:

```
$@ -- the current target's full name
$? -- a list of the target's changed dependencies
$< -- similar to $? but identifies a single file dependency and is
      used only in suffix rules
$* -- the target file's name without a suffix
```

Another handy macro based facility permits one to change prefixes on the fly. The macro `$(@:.o=.err)` says use the target name but change the .o to .err.

The same result effect may be obtained using `$*.err` as is done in the subsequent *cat* command.

Using macros in *makefiles*

The *makefile* on the previous page is actually broken! All of the *.c* files depend on *ray.h* and should be recompiled if *ray.h* changes, but this will not happen! We could fix this by typing in a collection of other dependencies but the macro facility simplifies that.

Make macros are similar in spirit to Unix *environment variables.* In fact environment variables can be accessed in make files via macro calls. However, it is typically the case that the macros are defined within the *makefile*. Here is a makefile that is used to build a complete raytracer. A macro is defined by using the syntax MACRO-NAME = macro value. Many people use the convention of making names all capital but that is not required.

All of the .o files necessary to build in are defined using the macro name RAYOBJS. The \ character at the end of all but the last line is the standard Unix contuation character. The # character at the start of a line turns the line into a comment.

A macro is invoked using the syntax $(MACRO-NAME). The result of the invocation is that the string $(MACRO-NAME) is replaced by the current value of the macro.

```
RAYOBJS = main1.o model.o camera.o list.o material.o plane.o \
          object.o sphere.o

RAYHDRS = vector.h ray.h rayfuns.h rayhdrs.h

a.out: $(RAYOBJS)
        gcc -Wall -g *.o -lm

$(RAYOBJS): $(RAYHDRS) makefile

.c.o: $<
        -gcc -c -Wall -c -g $<  2> $(@:.o=.err)
        cat $*.err
```

Determining if a ray hits a sphere.

Assume the following:

V = *viewpoint or start of the ray*

D = *a unit vector in the direction the ray is traveling*

C = *center of the sphere*

r = *radius of the sphere.*

The arithmetic is much simpler if the center of the sphere is at the origin. So we start by moving it there! To do so we must make a compensating adjustment to the base of the ray.

$C' = C - C = (0, 0, 0)$ = *new center of sphere*

$V' = V - C$ = *new base of ray*

D *does not change*

A point P on the sphere whose center is (0, 0, 0) necessarily satisfies the following equation:

$$p_x^2 + p_y^2 + p_z^2 = r^2 \qquad (1)$$

All points on the ray may be expressed in the form

$$P = V' + t\,D = (v'_x + td_x,\ v'_y + td_y,\ v'_z + td_z) \qquad (2)$$

where *t* is the Euclidean distance from *V'* to *P*

Thus we need to find a value of *t* which yields a point that satisfies the two equations. To do that we take the *(x, y, z)* coordinates from equation (2) and plug them into equation (1). We will show that this leads to a quadratic equation in *t* which can be solved via the quadratic formula.

$$(v'_x + td_x)^2 + (v'_y + td_y)^2 + (v'_z + td_z)^2 = r^2$$

Expanding this expression by squaring the three binomials yields:

$$(v'^{\,2}_x + 2tv'_x d_x + t^2 d_x^2) + (v'^{\,2}_y + 2tv'_y d_y + t^2 d_y^2) + (v'^{\,2}_z + 2tv'_z d_z + t^2 d_z^2) = r^2$$

Next we collect the terms associated with common powers of t

$$(v'^{\,2}_x + v'^{\,2}_y + v'^{\,2}_z) + 2t\,(v'_x d_x + v'_y d_y + v'_z d_z) + t^2(d_x^2 + d_y^2 + d_z^2) = r^2$$

Now we reorder terms as decreasing powers of t and note that all three of the parenthesized tri-nomials represent dot products.

$$(D \text{ dot } D)t^2 + 2\,(V' \text{ dot } D)\,t + V' \text{ dot } V' - r^2 = 0$$

We now make the notational changes:

$$a = D \text{ dot } D$$
$$b = 2\,(V' \text{ dot } D)$$
$$c = V' \text{ dot } V' - r^2$$

to obtain the following equation

$$at^2 + bt + c = 0$$

whose solution is the standard form of the quadratic formula:

$$t_h = \frac{-b +/- \text{sqrt}(b^2 - 4ac)}{2a}$$

Recall that quadratic equations may have 0, 1, or 2 real roots depending upon whether the *discrimant:*

$$(b^2 - 4ac)$$

is negative, zero, or positive.   These three cases have the following physical implications:

negative	=> ray doesn't hit the sphere
zero	=> ray is tangent to the sphere hitting it at one point (we will consider this a miss).
positive	=> ray does hit the sphere and would pass through its interior (this is the *only* case we consider a *hit).*

Furthermore, the two values of *t* are the distances from the base of the ray to the points(s) of contact with the sphere. We always seek the *smaller* of the two values since we seek to find the "entry wound" not the "exit wound".

Therefore, the *hits_sphere()* function should return

$$t_h = \frac{-b - sqrt(b^2 - 4ac)}{2a}$$

if the discriminant is positive and

$$t_h = -1$$

otherwise.

Determining the coordinates of the hit point on a sphere.

The *hits_sphere()* function should also fill in the coordinates of the *hit* in the *obj_t* structure.

The *(x, y, z)* coordinates are computed as follows.

$$H = V + t_h D$$

Important items to note are:

The actual base of the ray V and not the translated base V' must be used

The vector D must be a *unit vector* in the direction of the ray.

Determining the surface normal at the hit point.

The normal at any point P on the surface of a sphere is a vector from the *center* to the *point*. Thus

$N = P - C$ *(note that N will be a unit vector <==> r = 1)*

Therefore a unit normal may be constructed as follows:

$$N_u = (H - C) / || (H - C) ||$$

Creating an image

We continue to strive to build simple easy to grasp components! Obviously, this could be done by massively nesting loops and building functions 100+ lines long. Even some faculty and professional programmers will do it this way.

But if you do it *my* way you avoid the possibility that one day you will be recreated as a VooDoo doll by those charged with trying to understand and maintain what you wrote!!

```
/**/
/* This function is the driver for the raytracing procedure */

void image_create(
model_t *model)
{
   int      y;
   cam_t   *cam = model->cam;

/* Fire ray(s) through each pixel in the window */

    for (y = 0; y < cam->pixel_dim[1]; y++)
   {
      make_row(model, y);
   }

/* Create ppm image */

   cam_write_image(model->cam);

}
```

Processing a row of pixels

The most common way to mess this up is to forget which element of the *pixel_dim* array represents the horizontal size and which represents the vertical size.

```
static inline void make_row(
model_t *model,
int      y)
{
   int   x;
   cam_t *cam = model->cam;

   for (x = 0; x < cam->pixel_dim[0]; x++)
   {
      make_pixel(model, x, y);
   }
}
```

Building a pixel

This function is called for each pixel in the image. Eventually we will try to minimze the “jaggies” by building a loop in here in which we randomize the ray direction and average the computed pixel values.

```
static inline void make_pixel(
model_t *model,
int      x,
int      y)
{
   vec_t  raydir;
   drgb_t pix = {0.0, 0.0, 0.0};
   cam_t  *cam = model->cam;
   int    i;

/* This function was written previously */

   cam_getdir(cam, x, y, &raydir);

#ifdef DBG_PIX
   fprintf(stderr, "\nPIX %4d %4d - ", y, x);
#endif

/* The ray_trace function determines the pixel color in */
/* d_rgb units.. The last two parameters are used ONLY  */
/* in the case of specular (bouncing) rays which we are */
/* not doing yet.                                       */

   ray_trace(model, &cam->view_point,
                        &raydir, &pix, 0.0, NULL);

/* This function must convert the pixel value from drgb_t */
/* [0.0, 1.0] to irgb_t (0, 255) and to store it in the   */
/* “upside down” location in the pixmap                   */

   cam_setpix(cam, x, y, &pix);

   return;
}
```

The raytrace() function

```
/**/
/* This function traces a single ray and returns the  */
/* composite  intensity of the light it encounters    */

void ray_trace(
model_t  *model,
vec_t    *base,        /* location of viewer or previous hit */
vec_t    *dir,         /* unit vector in direction of object */
drgb_t   *pix,         /* pixel     return location          */
double   total_dist,   /* distance ray has traveled so far   */
object_t *last_hit)    /* most recently hit object           */
{
   object_t *closest;
   double   mindist;
   drgb_t   thisray = {0.0, 0.0, 0.0};

   Ask find_closest_object() to set the closest pointer
   If it returns an object pointer
   {
#ifdef DBG_HIT
   fprintf(stderr, "%-12s HIT:(%5.1lf, %5.1lf, %5.1lf)",
                    closest->objname,
                    closest->hitloc.x, closest->hitloc.y,
                    closest->hitloc.z);
#endif

      copy the object's  ambient reflectivity to “thisray”
   }

   scale the values of  “thisray” by 1 / distance to the closest object
   add the value of “thisray” to pix
#ifdef DBG_DRGB
   fprintf(stderr, "%-12s DRGB:(%5.2lf, %5.2lf, %5.2lf)",
                    closest->objname, pix->r, pix->g, pix->b);

#endif
}
```

Additional notes:

- It is useful to add some inline functions for copying, scaling, and performing component-wise multiplication of *drgb_t's* to *rayfuns.h*

- The use of *thisray* may seem unnecessary here. That's because it is unnecessary here. But if you don't use it you will almost certainly encounter problems in *antialiasing*.

Debugging output

Use the CFLAGS macro to enable precisely those debug aids that you need:

```
CFLAGS = -DAA_SAMPLES=1 -DDBG_PIX -DDBG_HIT
# -DSOFT_SHADOWS

ray: $(RAYOBJS)
        gcc -Wall -o ray -g -pg $(RAYOBJS) -lm

$(RAYOBJS): $(INCLUDE) makefile

.c.o: $<
        -gcc -c -Wall $(CFLAGS) -c -g $<  2> $(@:.o=.err)
        cat $*.err
```

```
PIX   21   16 - leftwall     (  1.3,   1.4,  -4.0)
PIX   22   16 - leftwall     (  1.5,   1.3,  -4.4)
PIX   23   16 - leftwall     (  1.6,   1.2,  -4.8)
PIX   24   16 - leftwall     (  1.8,   1.0,  -5.3)
PIX   25   16 - leftwall     (  2.0,   0.8,  -5.9)
PIX   26   16 - leftwall     (  2.2,   0.6,  -6.5)
PIX   27   16 - leftwall     (  2.4,   0.4,  -7.2)
PIX   28   16 - leftwall     (  2.7,   0.2,  -8.0)
PIX   29   16 - floor        (  3.0,   0.0,  -8.5)
PIX   30   16 - floor        (  3.4,   0.0,  -8.5)
PIX   31   16 - floor        (  3.8,   0.0,  -8.5)
PIX   32   16 - floor        (  4.2,   0.0,  -8.5)
PIX   33   16 - floor        (  4.6,   0.0,  -8.5)
PIX   34   16 - floor        (  5.0,   0.0,  -8.5)
PIX   35   16 - rightwall    (  5.3,   0.2,  -8.0)
PIX   36   16 - rightwall    (  5.6,   0.4,  -7.2)
PIX   37   16 - rightwall    (  5.8,   0.6,  -6.5)
PIX   38   16 - rightwall    (  6.0,   0.8,  -5.9)
PIX   39   16 - rightwall    (  6.2,   1.0,  -5.3)
PIX   40   16 - rightwall    (  6.4,   1.2,  -4.8)
```

Additions to the *camera* module.

The mission of *cam_setpix* is to convert a *drgb_t* pixel encoding to an *irgb_t* pixel encoding and store it in the pixmap.

```
/**/
void cam_setpix(
cam_t    *cam,
int      x,
int      y,
drgb_t   *pix)
{
     int     row;
     int     offset
     irgb_t *maploc;

     assert(cam->cookie == CAM_COOKIE);
```

scale_and_clamp(pix);   // convert to range [0, 255]]

```
#ifdef DBG_IRGB
   fprintf(stderr, " IRGB:(%5.0lf, %5.0lf, %5.0lf)",
                              pix->r, pix->g, pix->b);

#endif
```

Now it is necessary to set the pointer *maploc*. Recall from CPSC 101 that the pixel at location (row, col) in the image had

*offset = row * numcols + col* (1)

in the pixmap. That obviously remains true here.

```
#ifdef DBG_OFFSET
   fprintf(stderr, "OFF: %7d", offset);


#endif
```

It is also true that *x* corresponds to *col*. Unfortunately *y* doesn't exactly correspond to *row*. If you simply assume that it does your image will come out *upside down!*

This occurs because in our coordinate system y = 0 corresponds to the bottom row of the screen. In a .ppm file the first row of pixels in the file appears as the *top row* of the image!

Thus if *y = 0,* the value of *row* should be *cam->pixel_dim[1] – 1* (the last row of the pixmap) and if *y* = *cam->pixel_dim[1] – 1* (the top row of the screen) *row* should be 0 (the first row of the pixmap.

}

Creating the image

```
void cam_write_image(
cam_t *cam)
{

    assert(cam->cookie == CAM_COOKIE);
```

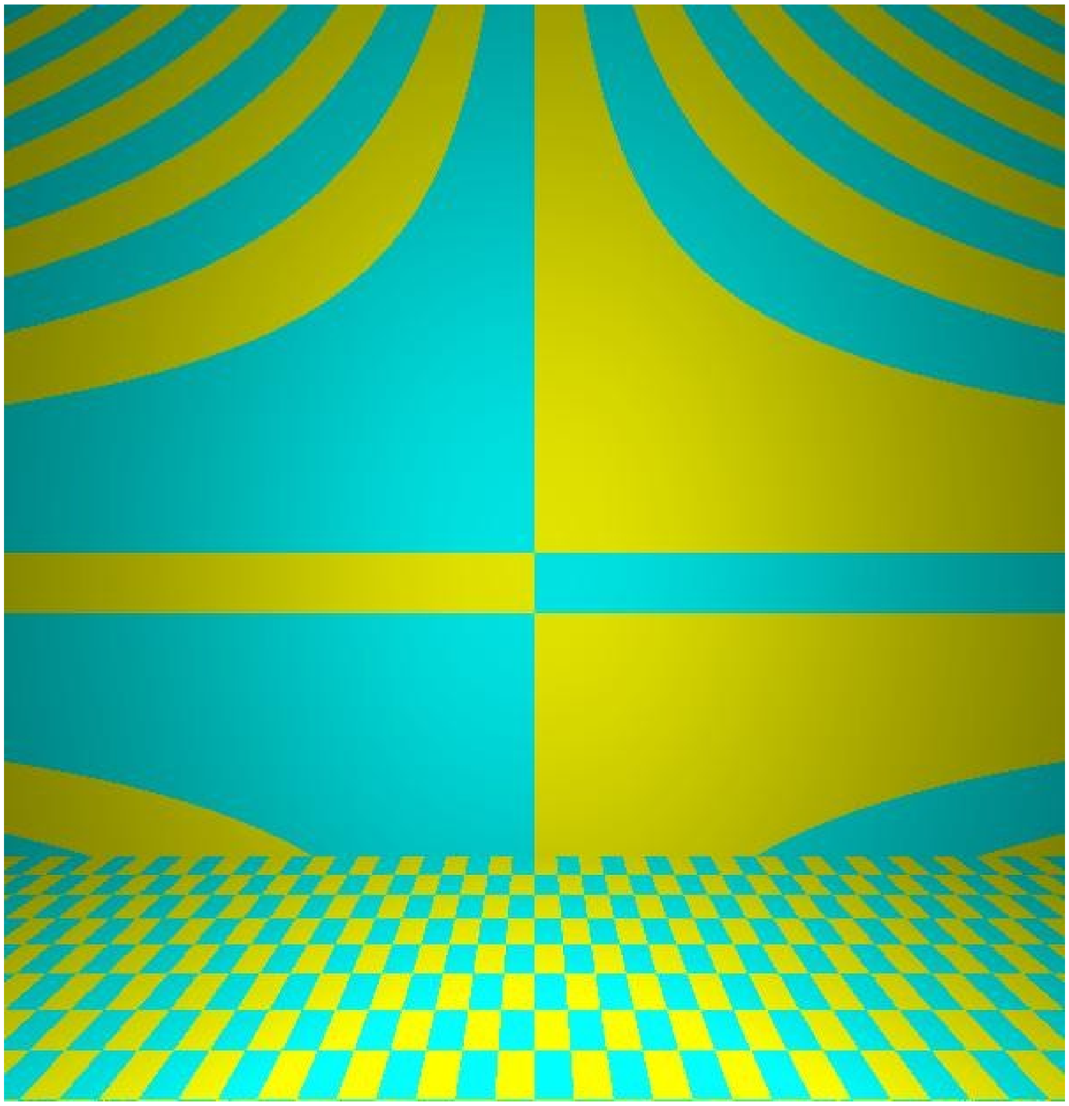
Write the ppm header file (remember width before height)
Write the entire binary pixmap to stdout with a single call to fwrite

```
}
```

Procedural surfaces

Procedural surfaces are those in which an object's reflectivity properties are *modulated* as a function of the location of the hit point on the surface of the object.

There are literally an infinite number of ways to do this. In the next few pages we propose a framework for incorporating procedurally shaded surfaces into raytraced images.



Implementation of procedural shaders

Construction of such shaders is facilitated by the use of both inheritance and polymorphism within a C language framework. The procedurally shaded plane is an lightweight refinement of the *plane_t.*

```
typedef struct pplane_type
{
   int     shader;
}  pplane_t;
```

The distinction between a standard plane and a procedurally shaded plane is made at object *initialization time* by the *pplane_init()* function when it establishes a single function pointer (for ambient only images) that provides the polymorphic behavior.

That function pointer is taken from a table of pointers to programmer provided functions are contained in the module *pplane.c* and perform the procedural shading. These procedural shading functions are passed pointers to the *object_t* structure and to the *dgrb_t intensity vector* whose *(r,g, b)* components are filled in procedurally. Here is an example in which there are three possible shaders.

```
static void (*pplane_shaders[])(object_t *obj, drgb_t *value) =
{
   pplane0_amb,
   pplane1_amb,
   pplane2_amb,
};

#define NUM_SHADERS sizeof(pplane_shaders)/sizeof(void *)
```

Note that:

1. The number of elements in the array is not explicitly specified.
2. The value *NUM_SHADERS* can be computed by dividing the size of the table by the size of a single pointer.

The index of the shader to be used is supplied in the model description as shown below.

```
pplane floor
{
   material gray
   normal   0 1 0
   point    0 0 -8
   shader   0
}

pplane backwall
{
   material gray
   normal   0 0 1
   point    4 3 -8
   shader   1
}
```

The *getamb()* function

Recall that in the ambient only raytracer the last steps of the operation are:

```
add mindist to total_dist
set intensity to the ambient reflectivity of closest object
divide intensity by total_dist
```

The first inclination is to implement the small amount of code in step 2 in the obvious way:

```
 this_pix.r = closest->mat->ambient.r;
 this_pix.g = closest->mat->ambient.g;
 this_pix.b = closest->mat->ambient.b;
```
or

```
pix_copy(&closest->mat->ambient,  &this_pix);
```

However that approach would make it *not easy to override* the *default* behavior. Thus a better approach is to replace the three lines above by:

```
  closest->getamb(obj, &this_pixy);
```

During the *object_init()* object constructor sets the *getamb()* function pointer to the "*default_getamb()*" function which contains the three lines of code we just replaced.

While this adds a slight bit of run time overhead, it also provides us with an easy hook with which we may override the *default_getamb()* with a custom shader.

Modifications to the *object_t* structure

The *getamb* element of the *obj_t()* structure is a pointer to a void function which is passed pointers to the object structure and the *intensity* vector.

```
typedef struct object_type
{
   int     cookie;
   char    objname[NAME_LEN];  /* left_wall, center_sphere */
   char    objtype[NAME_LEN];  /* plane, sphere, ...       */

   double  (*hits)(vec_t *base,vec_t *dir,
              struct object_type *);    /* Hits function.   */
   void    (*dumper)(FILE*, struct object_type *);

/* Optional plugins for procedural reflectivity */

   void    (*getamb) (struct object_type *, drgb_t *);
   void    (*getdiff)(struct object_type *, drgb_t *);
   void    (*getspec)(struct object_type *, drgb_t *);
```

Modifications to *object_init()*

In the *object_init()* function it is necessary to initialize the pointers so that the desired default behavior will be provided.

```
   obj->getamb  = material_getamb;
   obj->getdiff = material_getdiff;
```

Implementation of the default *getter* functions

It is then necessary to implement the default functions in the material.c module.

```
/**/
void material_getamb(
object_t *obj,
drgb_t   *dest)
{

   material_t *mat = obj->mat;
   assert(obj->cookie == OBJ_COOKIE);
   assert(mat->cookie == MAT_COOKIE);
   pix_copy(&mat.ambient, dest);
}
```

The *pplane_init()* function

As shown below the *pplane_init()* function simply invokes the *plane_init()* function to construct the object and then overrides the default *getamb()* function, replacing it with the shader function whose index is provided in the model description files.

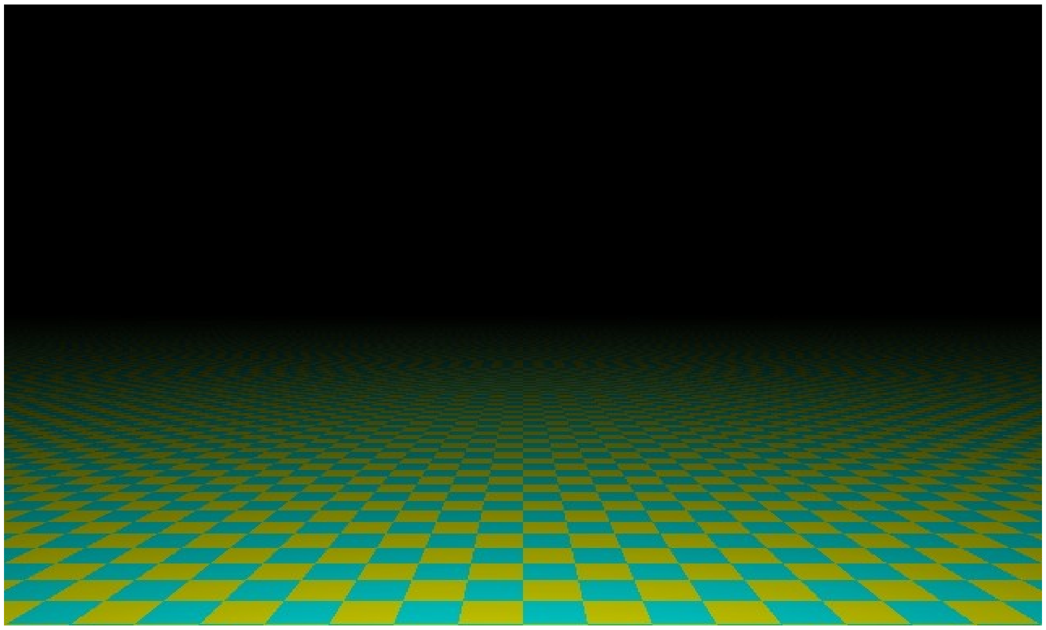
```
/**/
object_t *pplane_init(
FILE    *in,
model_t *model,
int  attrmax)
{
   pplane_t *ppln;
   plane_t  *pln;
   object_t *obj;
   link_t   *link;
   int  mask;

   plane_init(in, model, 2);

   link = model->objs->tail;
   obj  = (object_t *)link->item;
   pln  = obj->priv;

   ppln = (pplane_t *)malloc(sizeof(pplane_t));
   pln->priv = ppln;

   pplane_parse[0].loc = &ppln->shader;
   mask = parser(in, pplane_parse, 1, attrmax);
   assert(mask == 1);

   strcpy(obj->objtype, "pplane");
   obj->getamb = pplane_shaders[ppln->shader];
   obj->dumper = pplane_dump;
}
```

The mysterious attrmax parameter finally gets to do something!

Somewhat crude mechanism for retrieving the *object_t* pointer.

Tiled shading

To produce a tiled "floor" the modulation must be a function of the *x* and *z* coordinates because the *y* coordinate does not vary on the floor. For a "backwall" it would be necessary to modulate *x* and *y,* and for a "sidewall" it would be *y and z.*

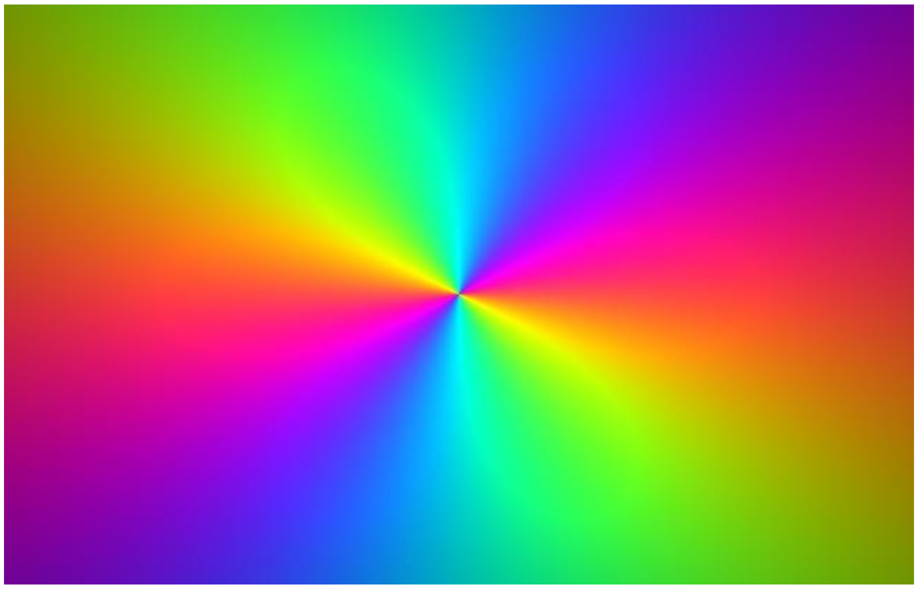


The factor of 2 controls the width of the tiles. The larger the factor the smaller the tile. The value of 1000 is known as Westall's hack for preventing an ugly double wide strip at the origin.

```
void pplane0_amb(
object_t *obj,
drgb_t *value)
{
   int    ix;
   int    iz;

   ix = 2  * obj->hitloc.x  + 1000;
   iz = 2  * obj->hitloc.z  + 1000;
   pix_copy(&obj->mat->ambient, value);

   if ((iz + ix ) & 1)  // test for odd or even sum
   {
      value->r = 0.0;   // make pixel cyan
   }
   else
   {
      value->b = 0.0;    // make pixel yellow
   }
}
```

Continuously modulated shading

The image shown below is produced by a procedural shader that continously modulates the ambient reflectivity.



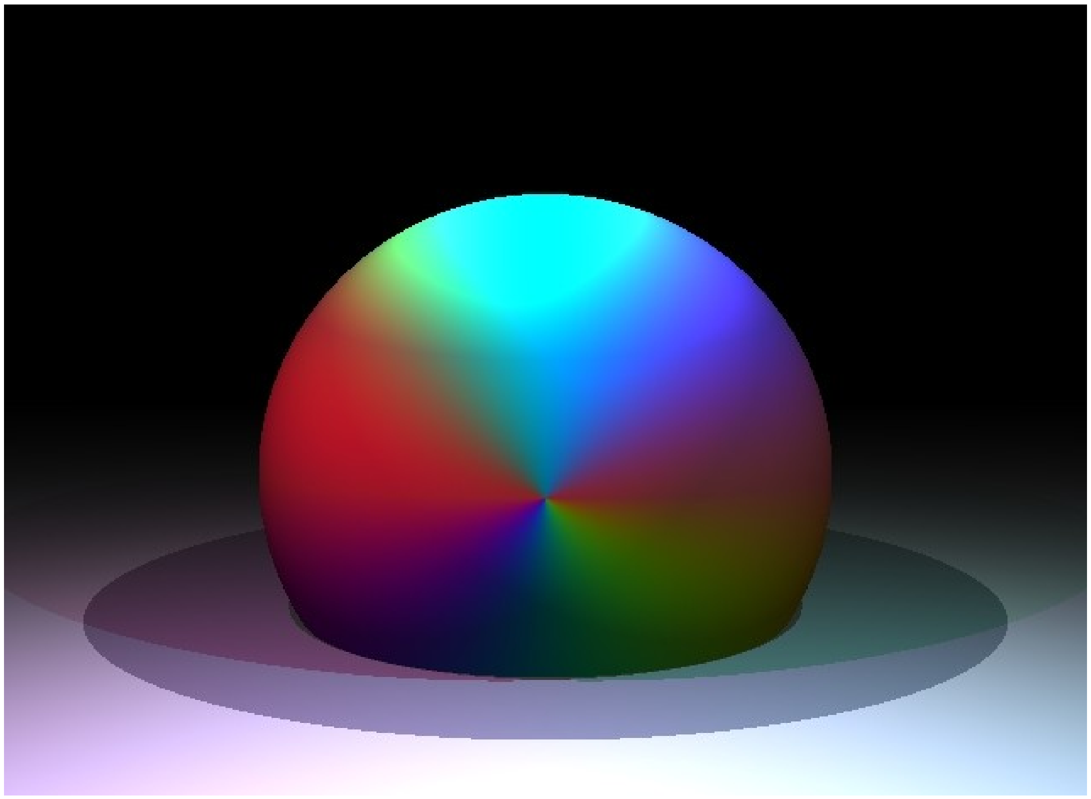
The modulation function is shown below. A vector *V* in the direction from the point defining the plane location to the hitpoint is computed first. Then the angle that the vector makes with the positive *X* axis is computed. Finally the red, green and blue components are modulated using the function $1 + \cos(\omega\, t + \phi)$ where the angular frequency ω is 2 for all three colors, and phase angles ϕ are 0, $2\pi/3$, and $4\pi/3$ respectively. Different effects may be obtained by using different frequencies and phase angles for each color, and it is also possible to combine continuous modulation with striping or tiling.

```
  vec_diff(&p->point, &obj->hitloc, &vec);

   v1 = (vec.x / sqrt(vec.x * vec.x + vec.y * vec.y));
   t1 = acos(v1);

   if (vec.y < 0)
      t1 = 2 * M_PI - t1;

  value->r = 6 * (1 + cos(2 * t1));
  value->g = 6 * (1 + cos(2 * t1+ 2 * M_PI / 3));
  value->b = 6 * (1 + cos(2 * t1+ 4 * M_PI / 3));

}
```

Procedural spheres

It is also easy to invent functions that procedurally color the surface of other objects. Here is one for a sphere.

```
assert(obj->cookie == OBJ_COOKIE);
sph = (sphere_t *)obj->priv;

vec_copy(&obj->hitloc, &coord);
vec_diff(&sph->center, &coord, &coord);

vec_scale(0.5 / sph->radius, &coord,  &coord);
v1 = coord.x / sqrt(coord.x * coord.x + coord.y * coord.y);
t1 = acos(v1);

if (coord.y < 3.0)
   t1 = 2 * M_PI - t1;

dest->r   =  1 * (1 + cos(2 * t1));
dest->g   =  1 * (1 + cos(2 * t1+ 2 * M_PI / 3));
dest->b   =  1 * (1 + cos(2 * t1+ 4 * M_PI / 3));
```



The cross product of two vectors

Given two linearly independent (not parallel) vectors:

$$V = (v_x, v_y, v_z)$$
$$W = (w_x, w_y, w_z)$$

The *cross product* sometimes called *outer product* is a vector which is orthogonal (perpendicular to) both of the original vectors.

$$V \text{ x } W = (v_y w_z - v_z w_y, \quad v_z w_x - v_x w_z , \quad v_x w_y - v_y w_x)$$

```
(1, 1, 1) x (0, -1, 0) = (1, 0, -1)
```

Notes:
The vector (0, -1, 0) is the negative *y axis*. Therefore, any vector that is perpendicular to it must lie in the $y = 0$ plane. The projection of the vector (1, 1, 1) onto the $y = 0$ plane is the vector (1, 0 1). The vector (1, 0, -1) is then perpendicular to this vector and lies in the $y=0$ plane.

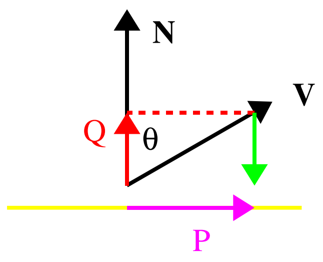
In a *right-handed* coordinate system

X x Y = Z
Y x Z = X
Z x X = Y

You will add the following function to your *vector.h* collection.

```
/**/
/* Compute the outer product of two input vectors */

static inline void vec_cross(
vec_t *v1,          /* Left input vector  */
vec_t *v2,          /* Right input vector */
vec_t *v3)          /* Output vector      */
{

}
```

Projection



Assume that *V and N* are unit vectors. The projection, *Q,* of *V* ***on N*** is shown in red. It is a vector in the same direction as *N* but having length cos(θ). Therefore

$$Q = (N \text{ dot } V)\, N$$

Now assume that *N* is a normal to a plane shown as a yellow line. The projection, *P,* of *V* onto the plane is shown in *magenta* and is given by *V + G* where *G* is the vector shown in green.

Since *G and Q* have clearly *have the same length* but *point in opposite directions*, $G = -Q$

Therefore the projection of a vector *V* onto a plane with normal *N* is given by:

$$P = V - (N \text{ dot } V)\, N$$

or (possibly)

vec_diff(vec_scale(vec_dot(N, V), N), V, P);

In building your new linear algebra routines it is desirable to build upon existing ones where possible but extreme levels of nesting of function calls as shown here *can complicate debugging.*

```
/**/
/* project a vector onto a plane */

static inline void vec_project(
vec_t *n,          /* plane normal     */
vec_t *v,          /* input vector     */
vec_t *w)          /* projected vector */
```

Matrices

Matrix operations are useful in transforming three-dimensional coordinate systems in ways that make it easier to determine if and where an object is hit by an array. There are alternative approaches to creating a 3 x 3 matrix. Probably the most obvious is:

```
double matrix[3][3];
```

but we will build upon our vector type as follows:

```
typedef matrix_type
{
   vec_t row[3];
}  mat_t;

mat_t matrix;
```

To set the the element in the 3rd column of the middle row to fifteen do,

```
matrix.row[1].z = 15.0;
```

To add two rows of a matrix:

```
vec_t sum;
vec_sum(&matrix.row[1], &matrix.row[2], &sum);
```

Suppose **V** ***and W*** are vectors and *a is* a scalar value. A function *F* that maps three-dimensional space to three-dimensional space is a *linear transformation* if and only if

$F(a\mathbf{V} + \mathbf{W}) = a\,F(\mathbf{V}) + \mathbf{W}$ *for any choice of a* **V** and **W**

Furthermore, any linear transformation may be represented by multiplication of a vector by a matrix.

Multiplication of a matrix times a vector.

The product of a 3 x 3 matrix with a 3-d column vector is a 3-d vector. The multiplication rule is as follows:

product[i] = the dot product of the *ith* row of the matrix with the vector.

```
        1.0  1.0  0.0      1.0       1.0
       -1.0  1.0  0.0  x   0.0   =  -1.0
        0.0  0.0  1.0     -2.0      -2.0
```

The *vec_xform()* function should multiply a vector by a matrix.

```
static inline void vec_xform(
mat_t *m,     /* input matrix            */
vec_t *v1,    /* vector to be transformed */
vec_t *v2)    /* output vector            */
{
   vec_t v3;  /* avoid aliasing problems */

/* Perform the transform */


   vec_copy(&v3, v2);
}
```

The transpose of a matrix:

The transpose of a three by three matrix is also three by the matrix. Its elements are given by a simple rule:

transpose[i][j] = original[j][i]

```
                   T
  1.0  3.0  2.0             1.0  1.0 -2.0
  1.0  2.0 -2.0      =      3.0  2.0  0.0
 -2.0  0.0  1.0             2.0 -2.0  1.0
```

Notes:

The diagonal elements of a matrix and its transpose are identical. Off diagonal elements are interchanged in a symmetrical way.

The transpose of a matrix is in general not the same as the inverse of a matrix.

```
static inline void mat_xpose(
mat_t *m1,    /* Input matrix            */
mat_t *m2)    /* Output transpose        */
{
   mat_t m3;  /* Avoid aliasing problems */

   m3.row[0].x = m1->row[0].x;
   m3.row[0].y = m1->row[1].x;
   m3.row[0].z = m1->row[2].x;

   etc....

   copy m3 back to m2..
}
```

Rotation matrices

Rotation matrices are used to rotate coordinate systems in 3-space. They have some special properties:

The three rows are mutually orthogonal unit vectors. That is, the dot product of any pair of rows is 0.

The three columns are also mutually orthogonal unit vectors.

The inverse of a rotation matrix is its transpose.

The 1st row of a rotation matrix is a vector which will be mapped to [1, 0, 0] under the rotation. The 2nd row is a vector will be mapped to [0, 1, 0] and the third row is a vector that will be mapped to [0, 0, 1].

This example shows that the middle row is mapped to (0, 1, 0)

$$\begin{vmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{vmatrix} \begin{vmatrix} r_{1,0} \\ r_{1,1} \\ r_{1,2} \end{vmatrix} = \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix}$$

Constructing rotation matrices

Suppose V and W are orthogonal unit length vectors. Suppose we want to create a rotation matrix *M* that will rotate V into the X-axis and W into the Z-axis.

```
vec_t V;
vec_t W;
mat_t M;

vec_copy(&V, &M.row[0]);   // V will become the X-axis
vec_copy(&W, &M.row[2]);   // W will become the Z-axis

/* The middle row Y = Z x X  */

vec_cross(&M.row[2], &M.row[0], &M.row[1]);
```

Suppose V and W are not necessarily orthogonal unit length vectors.   Suppose we want to create a rotation matrix *M* that will rotate W into the Z-axis and force V to lie in the X=0 plane.

```
vec_t v1 = {1.0, 0.0, 1.0};
vec_t v2 = {1.0, 1.0, 1.0};
vec_t v3;
vec_t v4;
mat_t m1;
mat_t m2;

vec_unit(&v1, &v1);
vec_unit(&v2, &v2);

vec_prn(stderr, "v1 ", &v1);
vec_prn(stderr, "v2 ", &v2);

vec_cross(&v1, &v2, &m1.row[0]);
vec_unit(&m1.row[0], &m1.row[0]);

vec_copy(&v2, &m1.row[2]);
vec_cross(&m1.row[2], &m1.row[0], &m1.row[1]);

vec_prn(stderr, "r0 ", &m1.row[0]);
vec_prn(stderr, "r1 ", &m1.row[1]);
vec_prn(stderr, "r2 ", &m1.row[2]);

vec_xform(&m1, &v1, &v3);
vec_xform(&m1, &v2, &v4);

vec_prn(stderr, "v3 ", &v3);
vec_prn(stderr, "v4 ", &v4);

mat_xpose(&m1, &m2);

vec_xform(&m2, &v3, &v3);
vec_xform(&m2, &v4, &v4);

vec_prn(stderr, "v3 ", &v3);
vec_prn(stderr, "v4 ", &v4);
```

```
v1     0.707   0.000   0.707
v2     0.577   0.577   0.577

r0    -0.707   0.000   0.707
r1     0.408  -0.816   0.408
r2     0.577   0.577   0.577

v3     0.000   0.577   0.816
v4     0.000   0.000   1.000

v3     0.707  -0.000   0.707
v4     0.577   0.577   0.577
```

Introduction to C++

We will spend the remainder of the class exploring aspects of C++. As before, the ray tracer will be used to illustrate the use of language.

The most significant extensions to C are:

- much *stronger type checking*
- the introduction of *true O-O classes*
- a *formal inheritance mechanism* in which derived classes can specialize parent classes
- formal support for *polymorphic behavior* in which a derived class may override a base class method simply by providing a method of the same name.
- support for *function overloading* in which there may be different implementations with a single function name.
- an *operator overloading mechanism* that is analogous to function overloading
- the ability to pass a parameters *by reference* in addition to the standard pass by value.
- yet another *input/output library* that may be used in addtion to standard and low level I/O

The *class* is a generalization of the C *struct*ure and can contain:

Function prototypes or full implementations
Accessibility controls *(friend, public, private, protected)*
Structured and basic data definitions

An example class definition

A class definition creates a type name that can be used in a standalone fashion (like a *typedef)* but the explicit typedef is no longer required. These should replace the comparable structure definitions in *ray.h.*

```
class cam_t
{

public:
          cam_t(){};
          cam_t(FILE *in);

   void   cam_getdir(int x, int y, vec_t *dir);
   void   cam_setpix(int x,int y, drgb_t *pix);
   int    cam_getxdim(void);
   int    cam_getydim(void);
   void   cam_getviewpt(vec_t *view);
   void   cam_dump(FILE *out);
   void   cam_write_image(FILE *out);

private:
   int    cookie;
   char   name[NAME_LEN];
   int    pixel_dim[2];    /* Projection screen size in pix */
   double world_dim[2];    /* Screen size in world coords   */
   vec_t  view_point;      /* Viewpt Loc in world coords    */
   irgb_t *pixmap;         /* Build image here              */
};
```

Example of class *constructors* and function overloading,

Public functions may be called by any entity holding a pointer to a *cam_t*

Private functions or data items are *not accessible* to any entity holding a pointer to a *cam_t*

Function prototypes that appear within a class definition are called *class methods* and are always invoked within the context of an instance of the class. They have *unqualified access to the data structures in the class.* That is, a class method will never refer to *cam->cookie.* It will simply do

```
cookie = CAM_COOKIE;
```

with the understanding that the particular *cookie* is the one that belongs to *this* instance of the class.

Creating an instance of a new class:

In the C language version of the ray tracer a new camera structure was created with a call to *cam_init()*. The *cam_init()* function then proceeded to *malloc()* a *camera_t()* structure and then initialize it.

```
cam = cam_init(stdin);
assert(cam != NULL);
```

Pointers to classes are declared just as pointers to structures are. However, dynamic creation of a new class uses the *new* operator.

```
/* main.c */

#include "ray.h"

int main(
int argc,
char *argv[])
{
   cam_t   *cam;
   model_t *model;

/* Load and dump camera data */

   cam = new cam_t(stdin);
   assert(cam != NULL);
   cam->cam_dump(stderr);
```

The *new* operator may be viewed as somewhat similar to *malloc()* but it *creates an instance of the class before invoking the initializer code (which is referred to as the constructor).*

```
/* Load and dump the model */

   model = new model_t(stdin);
   model->cam = cam;
   assert(model != NULL);
   model->dump(stderr);
```

Alternative ways to create an instance of a class

It is also possible to simply declare an instance of new class and invoke its constructor:

```
     cam_t cam(stdin);
```
or
```
     cam_t cam;
```

The parameters (or lack thereof) determine which constructor is invoked.   The *default* constructor which would be invoked in the second case doesn't actually do anything.

If parameters are supplied that don't match any prototype in the class definition, the C++ compiler responds with an appropriate nastygram.

```
  cam_t cam(stdin, model);

cat main.err
main.c: In function `int main(int, char**)':
main.c:14: error: no matching function for call to
`cam_t::cam_t(_IO_FILE*&, model_t*&)'

ray.h:63: note: candidates are: cam_t::cam_t(const cam_t&)
ray.h:66: note:                 cam_t::cam_t(FILE*)
```

Constructors

1. Are automatically called whenever an instance of the class is created
2. Must *never* have a return type --- not even *void*.
3. May be overloaded.. The function actually invoked is the one whose formal parameters "best" match the actual arguments. Thus when new cam_t(stdin) is invoked this constructor is called.

Although it is possible to implement function bodies inside class definitions, unless the functions are trivially short it leads to a big mess. The scope operator :: is saying this function belongs to the *cam_t* class.

```
cam_t::cam_t(
FILE *in)
{
   char  buf[256];
   int   mask;

   fscanf(in, "%s", buf);
   assert(strcmp(buf, "camera") == 0);

   fscanf(in, "%s", name);

   fscanf(in, "%s", buf);
   assert(buf[0] == '{');

   cookie = CAM_COOKIE;

   cam_parse[0].loc = &pixel_dim;
   cam_parse[1].loc = &world_dim;
   cam_parse[2].loc = &view_point;

   mask = parser(in, cam_parse, NUM_ATTRS, 0);
   assert(mask == 7);

/* Allocate a pixmap to hold the ppm image data */

   pixmap = (irgb_t *)malloc(sizeof(irgb_t) * pixel_dim[0] *
                                              pixel_dim[1]);

}
```

As previously noted, class data members are not and cannot by accessed using the *cam->* prefix. The major part of a C to C++ conversion is eliminating these pointer based references.

Getter and setter functions

One objective of the O-O approach is to *encapsulate* data within instances of classes and thus protect against the uncontrolled access present in standard C. Because of this, it is common for a class to export a collection of *getters* and *setters* that allow other classes access to required data but in a controlled way. For example the *image_create()* mechanism needs the *pixel_dimension* to determine how many times it should iterate on x and y.

The implementation of class methods must employ the structure:

```
return-type   class_type::function_name(parameters)
```

```
/**/
int cam_t::cam_getydim(
void)
{
   return(pixel_dim[1]);
}
```

Private data items are *not accessible* to any entity holding a pointer to a *cam_t,* but the *cam_getydim()* function is *public*. An entity that needs to invoke it *must* hold a valid pointer to a *cam_t,* as shown below.

A C++ program can contain a mix of C++ class methods and traditional C procedures. In fact the *main()* procedure will *always* be a traditional C procedure. A C++ religious warrior may assert that a *real* C++ programmer will never use any other traditional C procedures!

```
static inline void make_row(
model_t *model,
int      y)
{
   int   x;
   int   xdim;
   cam_t *cam = model->cam;

   xdim = cam->cam_getxdim();
   for (x = 0; x < xdim; x++)
   {
```

A C++ setter function

As before the elements belonging to the class are not qualified with a *cam->*

```
/**/
void cam_t::cam_setpix(
int       x,
int       y,
drgb_t    *pix)
{
   int      maprow;
   irgb_t *maploc;

   maprow = pixel_dim[1] - y - 1;
   maploc = pixmap + maprow * pixel_dim[0] + x;

   scale_and_clamp(pix);

   maploc->r = (unsigned char)pix->r;
   maploc->g = (unsigned char)pix->g;
   maploc->b = (unsigned char)pix->b;
}
```

An upside down class

Although it is common for data elements to be private and functions be public, it is sometimes useful to turn the model upside down in a hybrid C/C++ program. Here, the internal procedures do not need to support external reference (pointer) holders. It would be feasible to provide a bunch of *getters* that would return the pointer to the *cam* and the *lists*, but a reasonable person might conclude it is not necessary.

```
class model_t
{
public:
           model_t(FILE *);
   void    dump(FILE *);
   cam_t   *cam;
   list_t  *mats;
   list_t  *objs;
   list_t  *lgts;

private:
   inline void model_item_load(FILE *,char    *);
   inline void model_load(FILE *);

};
```

The *model_t* constructor

As before note that (1) there is no need to *malloc()* the model structure, (2) references to class member data elements (*mats, lgts, objs)* are unqualfied and (3) so are references to class methods *(model_load())*

```
/**/
/* Init model data */

model_t::model_t(
FILE *in)
{

   mats = list_init();
   assert(mats != NULL);

   lgts = list_init();
   assert(lgts != NULL);

   objs = list_init();
   assert(objs != NULL);

   model_load(in);

}
```

The *model_load()* method

This function consumes a single entity name (material, plane, sphere, etc) and then invokes yet another class method to perform the actual loading. Note that even though, this is a C++ method, in a model.cpp module its perfectly legal to use standard C library functions such as *fscanf(), strcmp(), etc.*

I personally find the C++ *iostream* facilty somewhat baroque, but again expect abuse from C++ religious zealots if you employ standard C I/O operations.

```
/* Load model data */

inline void model_t::model_load(
FILE    *in)
{
   char itemtype[16];
   int  count;

   memset(itemtype, 0, sizeof(itemtype));

/* Here itemtype should be one of "material",    */
/* "light", "plane"                             */

   count = fscanf(in, "%s", itemtype);
   while (count == 1)
   {
      model_item_load(in, itemtype);
      count = fscanf(in, "%s", itemtype);
   }
}
```

Creating new entities

Each invocation of *model_item_load()* creates a new instance of a *material, plane, sphere, etc.* In the C language version this was readily done using a table of function pointers. In the C++ version a *switch* or *if/else if /else if* mechanism must be used (I think). The problem is that there is no way (that I know of) to pass a variable value to the *new* operator.

```
        new plane_t(in, this, 0);
```

If we could replace *plane_t* by a variable which could take on the value *plane_t* or *sphere_t* or *material_t* then it would be straightforward to put a table driven approach back in place.

```
static char *items[] =
{
    "material",
    "plane",
};
#define NUM_ITEMS (sizeof(items) / sizeof(char *))

inline void model_t::model_item_load(
FILE    *in,
char    *itemtype)
{
   int ndx;
   ndx = getndx(items, NUM_ITEMS, itemtype);
   assert(ndx >= 0);
   switch (ndx)
   {
   case 0:
      new material_t(in, this, 0);
      break;
   case 1:
      new plane_t(in, this, 0);
      break;
   }
   return;
}
```

Recall that the initializers for *materials and planes* needed a pointer to the *model_t* structure in order to access the *lists*. The implicitly declared variable *this* is always available and points to *this* instance of the class.

Revisiting the *model_t* constructor.

In the code shown below we demonstrate that

- the *this* pointer can be used to provide qualified access to class data members.
- the *this* pointer can be copied to other pointers to provide a more C like representation

Both of these techniques are likely to evoke derision from "real" C++ programmers.

```
model_t::model_t(
FILE *in)
{
   model_t *model = this;

   this->mats = list_init();
   assert(mats != NULL);

   model->lgts = list_init();
   assert(lgts != NULL);

   objs = list_init();
   assert(objs != NULL);

   model_load(in);

}
```

The *material_t* class

The module *material.c* is a hybrid module involving *both standard C procedures and C++ class methods.* The class methods are for the most part typical *getters* providing access to private values.

```
class material_t
{
public:
   material_t(){};
   material_t(FILE *in, model_t *model, int attrmax);
   void  material_getamb(drgb_t *dest);
   void  material_getdiff(drgb_t *dest);
   void  material_getspec(drgb_t *dest);
   char *material_getname();
   inline void material_item_dump(FILE *out);

private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
};
```

Standard C procedures can *not* be declared here

Standard C functions in *material.cpp*

The standard C functions are those in which the entire list of materials must be processed. Since a class method is always invoked within the context of a single instance of the class (i.e. a single material) it is not natural to make them members of the class. These prototypes can *not* appear inside the class definition.

```
/* == material.cpp == */

/* Produce a formatted dump of the material list */

void material_dump(
FILE *out,
model_t *model);

/* Try to locate a material by name */

material_t *material_find(
model_t *model,
char    *name);
```

Initializing a new *material_t*

The only difference between the C and C++ version is the the use of unqualified data names: *cookie, ambient, diffuse*

```
/***/
/* Create a new material description */

material_t::material_t(
FILE        *in,
model_t     *model,
int         attrmax)
{
   char attrname[NAME_LEN];
   int count;
   int mask;

/* Create a new material structure and initialize it */

   cookie = MAT_COOKIE;

/* Read the descriptive name of the material */
/* (dark_red, light_blue, etc.                */

   count = fscanf(in, "%s", name);
   assert(count == 1);

   count = fscanf(in, "%s", attrname);
   assert(*attrname == '{');

   mat_parse[0].loc = &ambient;
   mat_parse[1].loc = &diffuse;
   mat_parse[2].loc = &specular;
   mask = parser(in, mat_parse, NUM_ATTRS, 0);

   assert(mask != 0);

   list_add(model->mats, (void *)this);
}
```

Actual argument here was *this*

Works because *mats* is public.

Interaction of C and C++ components

The mission of the *material_find* function is to find an instance of a *material_t* that has the proper name.   As such it may have to look at *all material_t'*. Therefore it is *not* a *material_t* class method!

```
/**/
/* Try to locate a material by name */

material_t *material_find(
model_t *model,
char    *name)
{
   link_t *link;
   material_t *mat;

   link = model->mats->head;

   while (link != NULL)
   {
      mat = (material_t *)link->item;

      if (strcmp(name, mat->material_getname()) == 0)
         return(mat);
      link = link->next;
   }

   return(NULL);
```

Only works because *mats is public*

Since this function is *not* a class method it cannot touch the *private mat->name.* But it can ask material_getname to return a pointer to it. C++ purists would claim (with some justification) this is a bad design.

This version of the function attempts to access *mat->name* directly. Since *name* is *private,* it does not succeed.

```
/**/
/* Try to locate a material by name */

material_t *material_find(
model_t *model,
char    *name)
{
   link_t *link;
   material_t *mat;

   link = model->mats->head;

   while (link != NULL)
   {
      mat = (material_t *)link->item;

      if (strcmp(name, mat->name) == 0)
         return(mat);
      link = link->next;
   }

   return(NULL);
}
```

Only works because *mats is public*

Since this function is *not* a class method it cannot touch the *private mat->name.*

```
g++ -c -Wall -DAA_SAMPLES=2 -c -g material.cpp  2> material.err
make: [material.o] Error 1 (ignored)
cat material.err
material.cpp: In function ‘material_t* material_find(model_t*,
char*)’:
ray.h:124: error: ‘char material_t::name [16]’ is private
material.cpp:104: error: within this context
```

The *friend* qualifier

Because it is very common for in hybrid C/C++ environments for "helper" functions to need access to private elements of classes. The C++ language provides the *friend* capability.

By using *friend,* one class can give to any functions or class the right to access its private elements directly. Almost *any* use of this facility is frowned upon by O-O purists. Excessive use is frowned upon by everyone.

Nevertheless if we include the following *friend* declaration in *material_t* the error shown on the previous page will go away!

```
class material_t
{

friend material_t *material_find(model_t *, char *);

public:
   material_t(){};
   material_t(FILE *in, model_t *model, int attrmax);
   void  material_getamb(drgb_t *dest);
   void  material_getdiff(drgb_t *dest);
   void  material_getspec(drgb_t *dest);
   char *material_getname();
   inline void material_item_dump(FILE *out);

private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
};
```

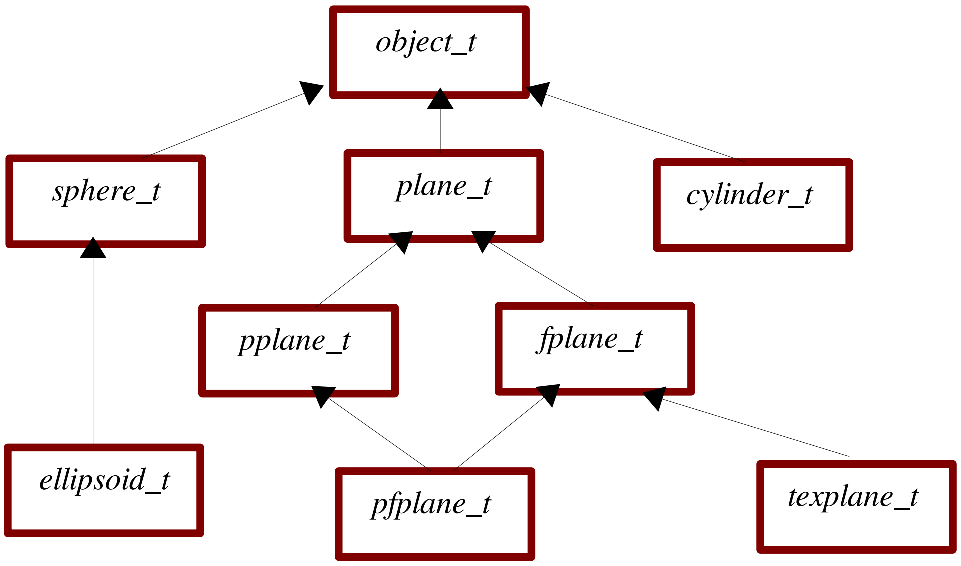
The use of *inheritence*

A C++ class hierarchy is based upon the principle of increased specialization.

- The *base* class carries attributes that are common to all classes and
- virtual functions that may or may not be overridden.

- Attributes that are esoteric to a particular entity *plane/normal* or *sphere/radius* are not defined in the *base class*
- Tthe derived class *inherits* the attributes of classes above it in the class hierarcy

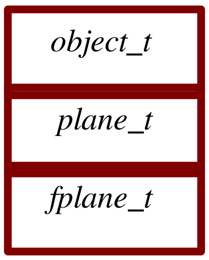
- The specialization can continue over multiple levels
- The amount of "new stuff" required in the implmentation of the derived class can range from trivial (*ellipsoid_t, pfplane_t)* to moderate *(plane_t, sphere_t)* to fairly complex (*texplane_t)*



Derived object creation

When an instance of a new class such as *fplane_t* is created, the *new* facility creates an instance of each class as it proceeds up the hierarchy until it reaches the base class and "glues" them together. The constructors *are always invoked top down*: *object_t() then plane_t() then fplane_t().*

Thus, our attribute loading strategy continues to work as before and model description files that worked in the C version *should work* in the C++ version.



The workings of the binding mechanism are not particulary obvious.

- When a class method at any level is invoked *the "this" pointer* always points to an instance of the level in the hierarchy to which the member function belongs. This guarantees that *plane_t* methods see *plane_t* instances.

- Pointers may also be forcibly *upcast* and *downcast* but this should be a last resort. Downcasting is particularly hazardous because what is *below* an *object_t* varies!!

The *object_t* class

```
class object_t
{
public:

   object_t(){};
   object_t(FILE *in, model_t *model);

   virtual ~object_t(){};

   inline void object_t::object_item_dump(FILE *out);

   virtual  double  hits(vec_t *base, vec_t *dir)
                       {return(-1.0);};
   virtual  void    dumper(FILE *);


   virtual    void    getamb(drgb_t *);
   virtual    void    getdiff(drgb_t *);
   virtual    void    getspec(drgb_t *);


protected:
   vec_t   hitloc;    /* Last hit point              */
   vec_t   hitnorm;   /* Normal at hit point         */

   int     cookie;
   char    objtype[NAME_LEN];  /* plane, sphere, ...      */

/* Surface reflectivity data */
   material_t *mat;

private:
   char   objname[NAME_LEN];  /* left_wall, center_sphere */

};
```

For mysterious reasons, g++ recently began to strongly dislike classes with virtual functions that do not provide a virtual destructor.

inline can still be used

Default behavior for *hits* is a *miss.* The *dumper* function is in object.cpp

Default behaviors for these guys are to punt the problem to *material_t.*

Protected attributes are accessible to derived classes. .

The *object_t* constructor

This function mirrors the operation of the old *object_init()* function. Data elements of the class are now referenced without the *obj->* qualifier.

```
/**/
/* Create a new object description */

object_t::object_t(
FILE        *in,
model_t     *model)
{
   char buf[NAME_LEN];
   int count;

/* Read the descriptive name of the object */
/* left_wall, center_sphere, etc.          */

   count = fscanf(in, "%s", objname);
   assert(count == 1);

/* Consume the delimiter */

   count = fscanf(in, "%s", buf);
   assert(*buf == '{');

   cookie = OBJ_COOKIE;
    :
```

The default reflectivity getters

These functions simply punt the material reflectivity request to the proper handler in the material class.

Excessive punting is often associated with *suboptimal O-O* design. An alternative design that could avoid this punt is to have the *real* objects (plane_t, etc) be derived from *both* material_t and *object_t.* This technique is called *multiple inheritance*.

Punting can also be avoided by copious use of *friend* and *public*, but the downside of that has already been addressed.

```
void object_t::getamb(drgb_t *amb)
{
   material_t *mat;

   mat = this->mat;
   mat->material_getamb(amb);
}

void object_t::getdiff(drgb_t *diff)
{
   material_t *mat;

   etc....
}
```

These functions may also be written more concisely as shown below. There is probably little if any runtime efficiency savings by doing so and the use of multiple pointer chains inside a single statement can make debugging *interesting.*

```
void object_t::getamb(drgb_t *amb)
{
   this->mat->material_getamb(amb);
}
```

Processing the complete object collection

Because an *object_t* member function always runs in the context of a single instance of a class, when it is necessary to process *all* instances of a specific class it must be done:

- in the context of a member function of *another class* (e.g. the model_t) or
- by a "standalone" C language function that is *not a member of any class*

The function that drives the dumping of the object list is implemented here as "case 2". It is a standard C function that is not a member of any class.

```
/**/
void object_dump(
FILE *out,
model_t *model)
{
   link_t  *link;

   object_t *obj;
   link = model->objs->head;

   while (link != NULL)
   {
      obj = (object_t *)link->item;
      obj->dumper(out);
      fprintf(out, "\n");
      link = link->next;
   }
}
```

Dumping the attributes of a single object

While it is *not appropriate* for an *object_t* member function to driver the dumping process, it is appropriate for a member function to deal with dumping a single instance of the class. The *object_item_dump()* function is a class member.

```
inline void object_t::dumper(
FILE     *out)
{

   assert(cookie == OBJ_COOKIE);
   fprintf(out, "%-12s %s \n", objtype, objname);
   fprintf(out, "%-12s %s \n", "material",
                     this->mat->material_getname());

}
```

The *object_item_dump()* function dumps the objects name and material name and then invokes the most specific *dumper* in the hierarchy. The dumping heirarchy is designed so that

- Each level in the hierarchy dumps its own attributes
- The attributes appear in *stderr* in top down order

In its class definition the *fplane_t* declares a *hits* function and a *dumper* function. Thus these virtual functions will *override* the *hits* and *dumper* provided by both the *object_t* and the *plane_t.*

```
class fplane_t: virtual public plane_t
{
public:
   fplane_t();
   fplane_t(FILE *, model_t *, int);

   virtual double  hits(vec_t *base, vec_t *dir);
   virtual void    dumper(FILE *);
```

The *fplane_t dumper()* function.

The *fplane_t dumper* uses the scope operator to invoke the *plane_t's* dumper. This *always* works because each entity *knows* what lies above it in the hierarchy. Traversing the hierarchy the other direction is possible but much nastier and requires specific *downcasting*.

After *plane_t::dumper()* dumps the *normal* and *point* attributes, it will return to the *fplane_t::dumper()* which will dump the *dims* and the *xdir*.

```
/***/
void fplane_t::dumper(
FILE *out)
{
   plane_t::dumper(out);

   fprintf(out,  "%-12s %5.1lf %5.1lf\n", "dims",
                                dims[0], dims[1]);
   fprintf(out,  "%-12s %5.1lf %5.1lf %5.1lf\n", "xdir",
                             xdir.x, xdir.y, xdir.z);
}
```

This scheme is indefininitely continuable. The *pfplane_t* has no attributes of its own but exists to allow the *getamb(), getdiff()* functions of the *pplane_t* to be mated with the *hits* function of the *fplane_t.*

```
/**/
void pfplane_t::dumper(
FILE *out)
{
   pplane_t::dumper(out);
   fplane_t::dumper(out);
}
```

The *find_closest_object()* function.

Since this function must also *process the entire object list* it must not be a member function of the *object_t* class.   As with the *dumper* it is a standalone C function.

```
object_t *find_closest_object(
list_t   *list,         /* Object list       */
vec_t    *base,         /* Base of ray       */
vec_t    *dir,          /* direction of ray  */
object_t *last_hit,     /* object last hit   */
double   *retdist)      /* dist to hit point */
{
   double    dist;
   object_t  *closest = NULL;
   double    mindist;
   object_t  *obj =  NULL;
   link_t    *link = list->head;

   while (link != NULL)
   {
      obj = (object_t *)link->item;

      if (obj == last_hit)
      {
         link = link->next;
         continue;
      }

      dist = obj->hits(base, dir);

      if (dist <= 0)
      {
         link = link->next;
         continue;
      }
```

The *hits* function at the lowest (most specific) point in the hierarchy is the one that is actually used.

```
#ifdef DBG_FIND
      fprintf(stderr, "FND %12s: %5.1lf - \n ",
                         obj->objname, dist);
#endif

      if ((closest == NULL) || (dist < mindist))
      {
          mindist = dist;
          closest = obj;
      }
      link = link->next;
   }

   *retdist = mindist;
   return(closest);
}
```

Creating a derived class

The *plane_t* is derived from the *object_t*

```
class plane_t: public object_t
{
public:
   plane_t(){};
   plane_t(FILE *, model_t *, int);

   virtual double  hits(vec_t *base, vec_t *dir);
   virtual void    dumper(FILE *);

protected:
   vec_t   normal;
   vec_t   point;

private:
   double  ndotq;
};
```

Derivation is specified in the class declaration. If its not here, then this is a standalone class!

For the derived *plane_t* to be functional it is necessary to use a constructor that can read the attributes

Here is where the *plane_t* says that it *will* provide *hits()* and *dumper()* functions that override the defaults

These values may be needed by procedural planes or finite planes and therefore they must not be private.

ndotq is used only in the *hits()* function of the *plane_t* and thus should be made *private.*

Creating a derived class

The *plane_t* is derived from the *object_t*

```
/**/
/* Create a new plane object and initialize it */

plane_t::plane_t(
FILE    *in,
model_t *model,
int      attrmax): object_t(in, model)
{
   int  mask;

   strcpy(objtype, "plane");

/* The parser is fairly generic but the address of where to */
/* put the data must be updated for each new object         */

   plane_parse[0].loc = &point;
   plane_parse[1].loc = &normal;
   mask = parser(in, plane_parse, NUM_ATTRS, attrmax);
   assert(mask == 3);

   vec_unit(&normal, &normal);
   vec_copy(&normal, &hitnorm);
   ndotq = vec_dot(&point, &normal);
}
```

This is a critically important element of inheritance. It specifies *which* constructor of the parent class should be used

A *vector* class

We will use a vector class to illustrate some aspects of C++ programming and in some lab exercises. Use of the vector class and C++ I/O factilities in your ray tracer is *entirely optional.* We put the class definition in file *vec.h*

```
#include <iostream>
using namespace std;

class vec_t
{

public:
   vec_t();
   vec_t(double, double, double);

private:
   double x;
   double y;
   double z;
};
```

These two lines replace *#include <stdio.h>* when using C++ I/O

Default constructor will set vector to (0,0,0). Other on will set it to values provided

The *vec_t* constructors

As with the ray tracing routines, we put the implementations of the class methods in *vec.cpp*.

```
#include "vec.h"

vec_t::vec_t()
{
   cerr << "default constructor" << endl;
   x = y = z = 0.0;
}

vec_t::vec_t(double xi, double yi, double zi)
{
   cerr << "3 parm constuctor" << endl;
   x = xi;
   y = yi;
   z = zi;
}
```

In C++ its sometimes difficult to figure out which overloaded function is called when (or if at all).

The *cerr* function may be passed any number of items of types that it "knows about" separated by << and they will given a default format and sent to the standard error. *cout* can be used to send output to the *stdout.* The counter part of these functions is:

```
   double d;
   cin >> d;
```

It can be used to read a value into a type that it "knows about".

The << and >> operators are used for bit shifting in standard C and are *overloaded* by the *ostream* and *istream* classes. We will see later on how to make << and >> know about the *vec_t* class itself.

Vector operations

In the C++ version of the vector class, our old design can be made to work *if* we make *vec_sum(), vec_diff(), and so on friends of vec_t.* In that way they can remain *standard C functions,* but have access to the private values *x, y, z* of any instance of a vector class to which they hold a reference.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
     :
};
```

Note that the *friend* declaration is mandatory for this function to be able to access the private elements of the *vec_t.* But when declared *friend* the implementation doesn't change.

```
inline void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
   v3->x = v1->x + v2->x;
   v3->y = v1->y + v2->y;
   v3->z = v1->z + v2->z;
}
```

Reference parameters

Standard C supports only *pass by value.* With pass by value, when a structure name is used as a formal parameter/actual argument, the entire structure must be *copied onto the stack*. This becomes increasingly undesirable as structures increase in size. Therefore the standard C solution is to make the formal parameter and actual argument be *pointers to the structure type*. Doing this provides two benefits:

- Only a single word (the pointer) must be pushed onto the stack
- The called function is able to modify elements of the structure.

C++ supports an additional parameter passing technique knowns as *pass by reference.* Use of pass by reference is signified by the use of *& instead of ** in the formal parameters of the function prototype. In C++ *all* function prototypes must have been defined at the point of invocation so there is no ambiguity regarding what technique is in use.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
   :
};
```

In the body of the implementation the reference parameter is *treated as a structure*, and not a pointer to a structure.

```
inline void vec_sum(vec_t &v1, vec_t &v2, vec_t &v3)
{
   v3.x = v1.x + v2.x;
   v3.y = v1.y + v2.y;
   v3.z = v1.z + v2.z;
}

inline void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
   v3->x = v1->x + v2->x;
   v3->y = v1->y + v2->y;
   v3->z = v1->z + v2->z;
}
```

Invoking the overloaded *functions*

On the previous page we created two distinct versions of *vec_sum()*. Fortunately, they produce the same answer here, but there is no requirement that they do so. On this page we look at the problems of how to invoke them and which one gets invoked.

To use pass by reference *the actual argument must be an instance of the class* not a pointer to an instance of the class.

The instance of an overloaded function that is used is the one whose formal parameters (best) match the actual arguments. The matching is straightforward when instances of or pointers to classes are the parameters.

```
int main()
{
   vec_t v1(1.0, 2.0, 3.0);
   vec_t v2(4.0, 5.0, 6.0);
   vec_t v3;


   vec_sum(&v1, &v2, &v3);
   v3.put();


   vec_sum(v1, v2, v3);
   v3.put();

}
```

The parameterized constructor is called here.

The default constructor is called here.

The pointer based implementation is invoked here.

The reference based implementation is invoked here.

A broken function:

We could attempt to add a third implementation of *vec_sum()* which passes parameters by value by copying them onto the stack.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t , vec_t , vec_t );
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
```

To call this verson we would use:

```
   vec_sum(v1, v2, v3);
```

This would produce the following compile time error. Because the calling sequence shown IS the correct way to invoke either of the bottom two prototypes the compiler can't distinguish which one you want.

```
main.cpp: In function ‘int main()’:
main.cpp:24: error: call of overloaded ‘vec_sum(vec_t&, vec_t&,
vec_t&)’ is ambiguous
vec.h:68: note: candidates are: void vec_sum(vec_t&, vec_t&, vec_t&)
vec.h:75: note:                 void vec_sum(vec_t, vec_t, vec_t)
acad/cs102/examples/vec ==>
```

The compile time problem could be "fixed" by removing the implementation that used reference parameters and the resulting program would compile fine but *just would not work.*

Exercise: Why not?

A class based approach

We can also implement yet another instance of *vec_sum()* in which this one is a true member function of the class.

At first glance this one looks a bit odd because *v1* seems to have disappeared!! This occurs because *class methods are always invoked in the context of an instance of the class.* In this case the instance will be *v1*.

The *const* modifier makes it *not* possible for *vec_sum()* to modify *v2*. Obviously *v3* should *not be const.*

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);

   void vec_sum(const vec_t &v2, vec_t &v3);

   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);

};
```

The implementation also looks somewhat asymmetric with *v2* and *v3* being explicitly accessed in contrast to the implicit access to *v1*.

```
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
  v3.x = x + v2.x;
  v3.y = y + v2.y;
  v3.z = z + v2.z;
}
```

The asymmetry is also apparent in the invocation.

```
   v1.vec_sum(v2, v3);
   v3.put();
```

A collection of possible implementations for vec_sum

C++ makes it possible (though not necessarily desirable) to provide implementations that match virtually any parameterization:

```
   void vec_sum(const vec_t &v2, vec_t &v3);
   void vec_sum(const vec_t *v2, vec_t *v3);
   void vec_sum(const vec_t *v2, vec_t &v3);
   void vec_sum(const vec_t  v2, vec_t *v3);
```

Exercises: Identify any possible *usuable* prototypes that we may have missed. Identify a prototype different from any of those above that would cause the compile time problem we saw previously. Identify a prototype that would compile correctly but would not work. Each prototype must have a distinct implementation depending upon whether reference or value pointers are used .

```
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
  v3.x = x + v2.x;
  v3.y = y + v2.y;
  v3.z = z + v2.z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t *v3)
{
  v3->x = x + v2->x;
  v3->y = y + v2->y;
  v3->z = z + v2->z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t &v3)
{
  v3.x = x + v2->x;
  v3.y = y + v2->y;
  v3.z = z + v2->z;
}

void vec_t::vec_sum(const vec_t v2, vec_t *v3)
{
  v3->x = x + v2.x;
  v3->y = y + v2.y;
  v3->z = z + v2.z;
}
```

This is the *only* place an entire vector must be copied onto the stack.

Implementations *returning* instances of *vec_t*

It is possible in *both* C and C++ to define instances of *vec_sum()* that *return* the answer. Unless you have a *real good* reason for doing so, this is generally a bad idea because in both languages it causes a copy on to and copy off of the stack operation. (In the C language function overloading (the use of multiple implementations of the same function name is also illegal).

Here the prototype is declared to *return* an instance of *vec_t*.

```
vec_t vec_sum(const vec_t &v2);
```

The implementation requires a temporary variable in which the sum is computed. It is important to remember to *return(tmp);*

```
vec_t vec_t::vec_sum(const vec_t &v2)
{
  vec_t tmp;
  tmp.x = x + v2.x;
  tmp.y = y + v2.y;
  tmp.x = z + v2.z;
  return(tmp);
}
```

To compute the result v3 = v2 + v1 the following C++ code can be used. The "default" structure assignment mechanism takes care of copying the result off of the stack and into v3. This overhead is realitively minor for a vector but would be seriously bad for a structure containing a large array.

```
 v3 = v2.vec_sum(v1);
```

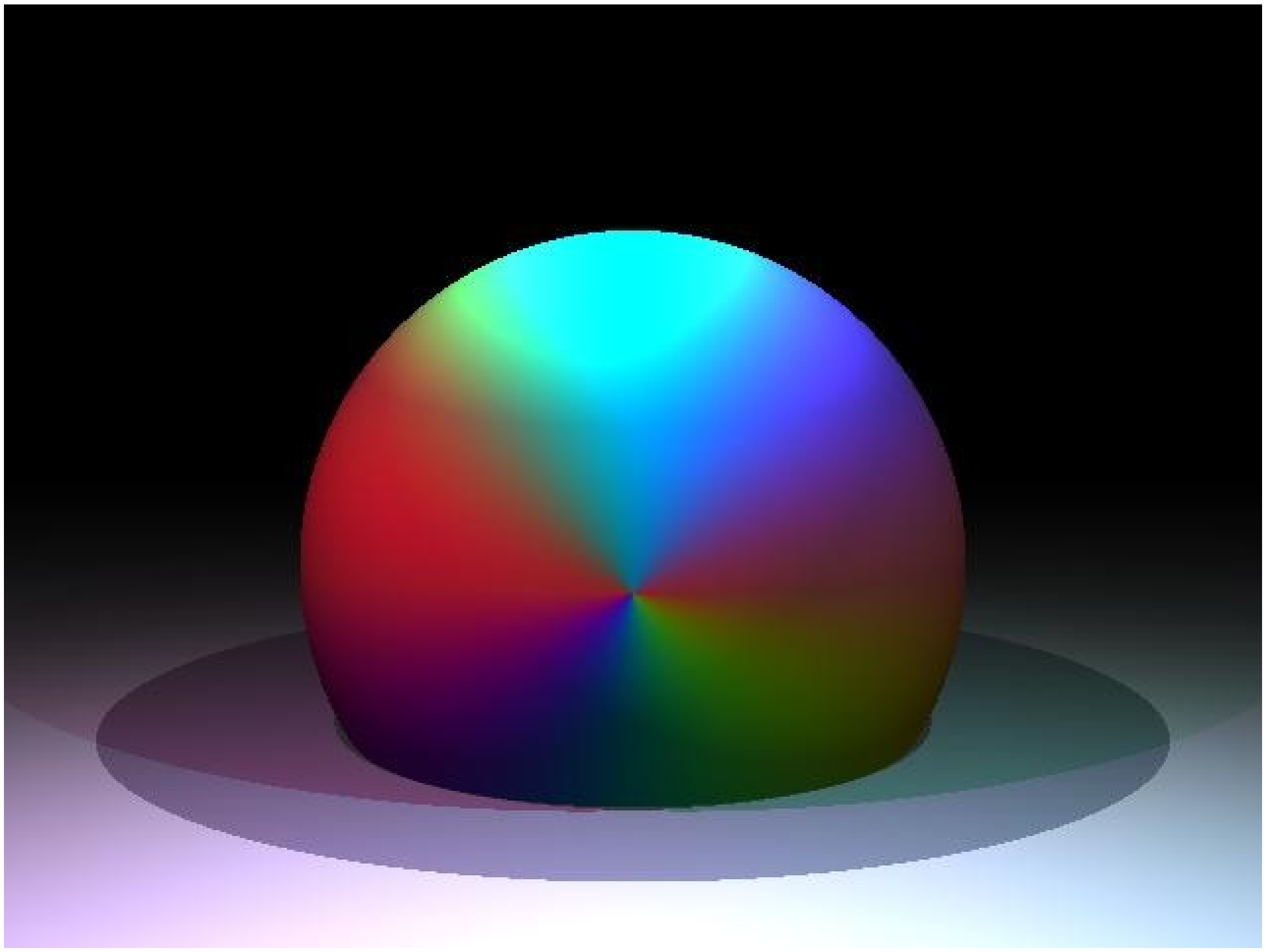
Diffuse illumination

Diffuse illumination is associated with specific light sources but is reflected uniformly in all directions. A white sheet of paper has a high degree of diffuse reflectivity. It reflects light but it also scatters it so that you cannot see the reflection of other objects when looking at the paper.

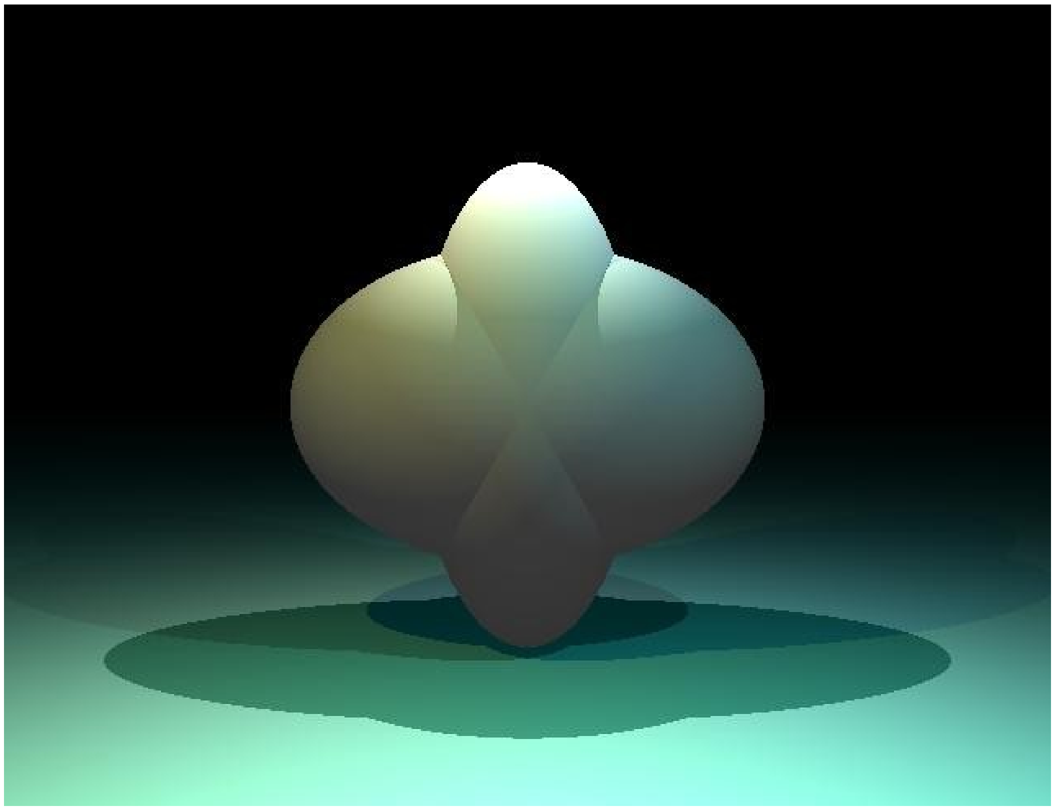
To model diffuse reflectivity requires the presence of one or more light sources. The first type of light source that we will consider is the point light source. This idealized light source emits light uniformly in all directions. It may be located on either side of the virtual screen but is, itself, *not visible.*

In addition to providing additional realism, diffuse illumination also provides (almost) realistic shadows. The "almost" qualifier is necessary because

- real light sources are not points and thus provide "soft" rather than "hard" shadow boundaries
- real objects that reflect light diffusely also illuminate other objects.



Implementin diffuse illumination



A list structure for holding instances of the class *light_t* is already present in the *model_t* class.

```
class model_t
{
public:
           model_t(FILE *);
   void    dump(FILE *);
   cam_t   *cam;
   list_t  *mats;
   list_t  *objs;
   list_t  *lgts;
private:
   inline void model_item_load(FILE *,char    *);
   inline void model_load(FILE *);

};
```

The *light_t* class

```
class light_t
{
public:
   light_t(){};
   light_t(FILE *in, model_t *model, int attrmax);

   virtual ~light_t(){};



   virtual int     vischeck(vec_t  *);
   virtual void    illuminate(model_t *, object_t *, drgb_t *);

   inline  void    light_item_dump(FILE *);


protected:
   vec_t   location;
   drgb_t  emissivity;
   char    name[NAME_LEN];

private:

   int     cookie;
};
```

Invoked for *every hit point and every light.*

Values specified in the model description file

The *light_t* class model description

The description of a light is consistent with the structure of the description of visible object. Each light has a descriptive-name, a location and an emissivity. Emissivity components of white lights are necessarily equal.

```
light red-ceiling
{
   location 4 8 -2
   emissivity 5 1 1
}

light blue-floor
{
   location 2 0 0
   emissivity 1 1 5
}
```

Modifications to *model.cpp*

Two modifications are needed in *model.cpp*

- The *model_item_load()* method must create new instances of *light_t new light_t(in, this, 0);*
- The *dump()* method must invoke the C *function light_dump(out, this)* to dump the light list.

The *light_t* constructor

The *light_t* constructor has the same interface as *object_t* constructors.

```
/**/
/* Create a new light description */
light_t::light_t(
FILE          *in,
model_t       *model,
int            attrmax)
{

- parse the required attribute values
- put the light_t class instance into the light list

}
```

The *light* list dumper.

```
/**/
/* Produce a formatted dump of the light list */

void light_dump(
FILE *out,
model_t *model)
{
   link_t *link;
   light_t *lt;

   link = model->lgts->head;

   for each light_t in the light list
       invoke class method light_item_dump();

}
```

Adding diffuse illumination to the *raytrace()* function

```
/* Hit object in scene, compute ambient and diffuse */
/* intensity of reflected light.                    */

   closest->getamb(&thisray);  // add ambient reflectivity
```

The *add_diffuse()* function drives the diffuse lighting process.

```
   add_diffuse(model, closest, &thisray);
```

The use of the local *this_ray* variable is made necessary by anti-aliasing (which will be described later).   The *scale before add* is ABSOLUTELY NECESSARY for anti-aliasing to work properly!

```
/* Scale intensity by distance of ray travel       */
/* then add the scaled value to the values pointed */
/* to by pix                                       */

   pix_scale(1 / total_dist, &thisray, &thisray);
   pix_sum(&thisray, pix, pix);
```

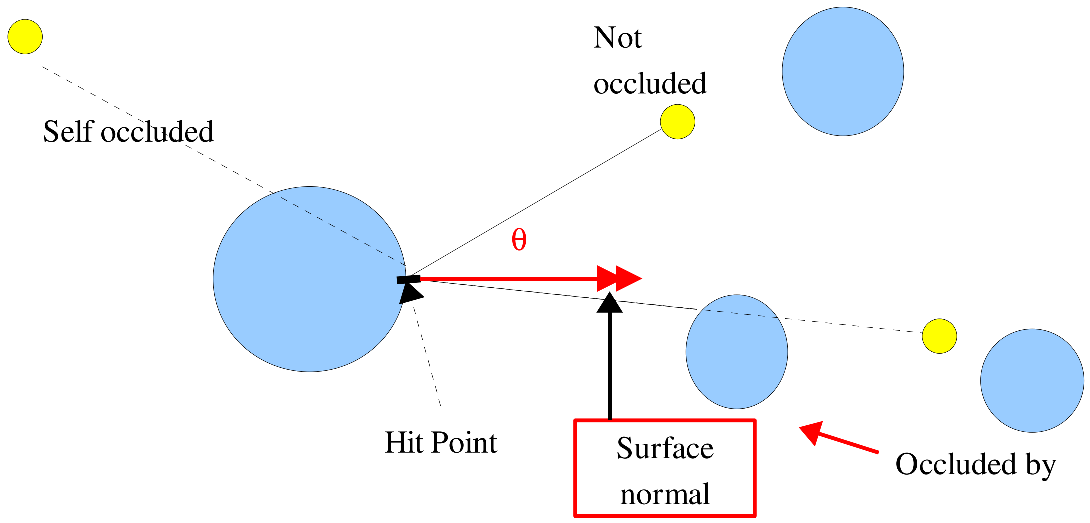
The *add_diffuse()* function

This is a standard C function. As shown on the previous page it is called *every time* an object is hit by a ray. Its mission is to process *the entire light list.* For each light the illumination it provides must be added to **pixel*.

```
void     add_diffuse(
model_t  *model,    /* object list                  */
object_t *hitobj,   /* object that was hit by the ray */
drgb_t   *pixel)    /* where to add intensity       */
{
   light_t *light;
   link_t  *link;

   link = model->lgts->head;

   while (link != NULL)
   {
      light = (light_t *)link->item;
      light->illuminate(model, hitobj, pixel);
      link = link->next;
   }
   return;
}
```

Computing illumination

We use idealized point light sources which are themselves invisible, but do emit illumination. Thus *lights themselves will not be visible in the scene* but the effect of the light will appear. Not all lights illuminate each hit point. The diagram below illustrates the ways in which objects may occlude lights. The small yellow spheres represent lights and the large blue ones visible objects.



We will assume convex objects. An object is *self-occluding* if the angle between the surface normal and a vector from the hit point toward the light is larger than 90 degrees.

A simple test for this condition is that an object is *not* self-occluding if

the *dot product* of a vector from the *hit point* to *that light* with the *surface normal* is *positive*.

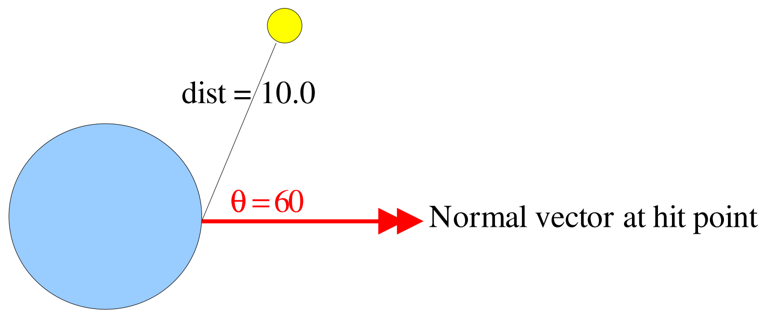
To see if a light is occluded by another object, it is necessary to see if a ray fired from the *hitpoint* to the *light* hits another object before it reaches the light. This can be accomplished via a call to *find_closest_object()* The light is occluded if and only if

(1) *the ray hits some new object*
AND
(2) the distance to the *hit point on the new object* is less than the *distance to the light.*

Computing the illumination (details)

```
void light_t::illuminate(
model_t      *model,
object_t     *hitobj,  /* The object hit by the ray   */
drgb_t       *pixel)   /* add illumination here       */
{

   vec_t    dir;    // unit direction to light
   vec_t    revdir; // unit direction from light to hitpt
   object_t *obj;   // closest object in dir to light
   double   close;  // dist to closest object in dir to light
   double   cos;    // of angle between normal and dir to light
   double   dist;   // to the light from hitpoint
   drgb_t   diffuse = {0.0, 0.0, 0.0};

/* Compute the distance from the hit to the light and a unit */
/* vector in the direction of the light from hitpt           */
   ..........
/* Test the object for self-occlusion and return if occluded */
   ..........
/* Ask find_closest_object() if a ray fired toward the light */
/* hits a "regular" object.  Pass "hitobj" as the "lasthit"  */
/* parameter so it won't be hit again at distance 0.         */
   ..........
/* If an object is hit and the distance to the hit is   */
/* closer to the hitpoint than the light is, return     */
   ............
/* Arriving at this point means the light does illuminate */
/* object. Ask hitobj->getdiff() to compute diffuse       */
/* reflectivity                                           */
   .............

/* Multiply componentwise the diffuse reflectivity by     */
/* the emissivity of the light.                           */
   .............

/* Scale the resulting diffuse reflectivy by cos/dist     */
   ..............
/* Add scaled value to *pixel.                            */
```

An example illumination computation



Suppose the diffuse reflectivity of the object is:	*diffuse = {1, 2, 10}*
Suppose the emissivity of the light is	*emissivity = {8, 8, 2}*
Then the componentwise product is	*diffuse = {8, 16, 20}.*
Since *cos(θ)* = 0.5. The scale factor is	*0.5 / 10 = 1 / 20.0.*
The scaled value of	*diffuse* = {0.4, 0.8, 1.0}

is then added to the current value of **pixel.*

Common *fatal* problems

- failing to use unit vectors when attempting to compute *cos(θ)*
- inadvertently modifying the *emissivity* of the light or the *diffuse reflectivity* of the object instead of modifying the *local variable diffuse* shown on the previous page.

Accessing *hitloc* and *hitnorm*

These are protected members of the *object_t* class. Therefore you must either:

- Declare the *light_t* class to be a *friend* of the *object_t* class. This may produce "circular definition" complications. The easiest way to avoid these is

  ```
  class light_t;

  class object_t
  {
    friend class light_t;
  ```

- Add *getter* functions to the object class that will fill in the *hitloc* and *hitnorm* values.

- Make only the *light_t::illuminate()* function a friend. This complicates the circular definition problem.

Operator overloading in C++

Most operators can be overloaded. The ones that cannot are

`.` `.*` `::` `? :` and `sizeof`

Operator overloading is actually part of the function overloading mechanism. To overload an operator you simply provide a *function of the appropriate name*. For example,

Operator	Function name
`+`	`operator+`
`-`	`operator-`
`*`	`operator*`
`<=`	`operator<=`

Operator overloading is restricted to existing operators. Thus it is *not legal* to try to overload **

```
**       operator**
```

The *operator* functions work "almost" *exactly like "regular"* functions. They can even be invoked using their *operator*+ or *operator*- names. Keeping this fact in mind will remove much of the mystery from how they work and how they must be implemented. The "almost" qualifier above reflects necessity to remember that C operators take either *one or two operands.* Thus an operator function has *at most* two parameters.

The C addition operator takes two operands: *a* + *b*

Therefore the *operator*+ function will have two parameters: the first will represent the *left side operand* and the second the *right side* operand.

The C logical not operator takes one operand: *!value*

Therefore the *operator!* function will have *one* parameter and it will represent the *right side* operand.

Operator overloading *must preserve the "aryness"* (unary or binary) nature of the operator. Thus, the ! operator could be overloaded to compute the length of a *single vector but could not be used to compute a dot product.*

Most operator functions return a value that replaces the operator and its operand(s) in an expression. However, that is not mandatory. For example a *side effect* operator such as ++ may not need to return a value.

Like "regular functions" all operator functions may be defined as "regular C" *friend* functions of the class(es) to which their operands belong.

In *some* but not all cases they may be defined as *class member* functions instead.

Overloads as *friend* functions

We can write a version of the *vec_sum()* function that uses the + operator.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);

   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);

   friend inline vec_t operator+(const vec_t &, const vec_t &);
};
```

The body of the function is written in exactly the same way that a *vec_sum()* function which returned the sum would be written.

```
vec_t inline operator+(const vec_t &v1, const vec_t &v2)
{
  vec_t v3;
  v3.x = v1.x + v2.x;
  v3.y = v1.y + v2.y;
  v3.z = v1.z + v2.z;
  return(v3);
}
```

The function may then be invoked either by standard use of the + operator or by using its *operator+()* name.

```
   vec_t v1(1.0, 2.0, 3.0);
   vec_t v2(4.0, 5.0, 6.0);
   vec_t *vp = &v2;
   vec_t v3;

   v3 = v1 + v2;
   v3 = operator+(v1, v2);
```

If we try to invoke operator + as:

```
   v3 = v1 + vp;
```

```
main.cpp:21: error: no match for ‘operator+’ in ‘v1 + vp’
vec.h:98: note: candidates are: vec_t operator+(const vec_t&, const
vec_t&)
```

This occurs because *vp* is a pointer.   We can fix this in one of two ways.

1 – Write another instance of the operator+() function in which the second parameter is a *vec_t* *
2 – Just invoke the function as:

```
   v3 = v1 + *vp;
```

Overloads as *member* functions

Overloaded operators can be *member functions* instead of friend functions *if the left side operand is an instance of the class or if the right side operand of a unary operator such as ! is a class member.*

```
class vec_t
{
   :
   vec_t operator+(vec_t &rhs);
   vec_t operator+(vec_t *rhs);
   vec_t operator!(void);
   :
};
```

If a binary operator is a class method the left side operand becomes the "this" instance in whose context the function is invoked..

A unary operator which is a class method needs no parameters at all.

```
vec_t vec_t::operator+(vec_t &rhs)
{
   vec_t tmp;

   tmp.x = x + rhs.x;
   tmp.y = y + rhs.y;
   tmp.z = z + rhs.z;

   return(tmp);
}

vec_t vec_t::operator+(vec_t *rhs)
{
   vec_t tmp;

   tmp.x = x + rhs->x;
   tmp.y = y + rhs->y;
   tmp.z = z + rhs->z;

   return(tmp);
}
```

This demonstrates that it is possible to create a second implementation that takes a pointer to a vector instead of an instance of a vector. The invocation below actually works as intended, *but it is really ugly when you think about it!* *What sense does it make to add a vector and a pointer*.

```
 v3 = v1 + vp;
```

Scaling a vector

In the scaling operation we want to multiply the components of a vector by a double precision value. Since the left side operand is conventionally the double precision value *and not a vector* the *friend function method must be used*. Note that it is possible to *write the body of the friend function* within the class definition. Also note that for this weeks lab assignment, this approach is *not allowed.*

```
   friend vec_t operator*(double val, const vec_t &rhs)
   {
      vec_t tmp;
      tmp.x = val * rhs.x;
      tmp.y = val * rhs.y;
      tmp.z = val * rhs.z;
      return(tmp);
   }
```

We can also write a function that computes the component-wise product:

```
   friend vec_t operator*(const vec_t &lhs, const vec_t &rhs)
   {
      vec_t tmp;
      tmp.x = lhs.x * rhs.x;
      tmp.y = lhs.y * rhs.y;
      tmp.z = lhs.z * rhs.z;
      return(tmp);
   }
```

The correct implementation will be chosen by the compiler depending upon the operands.

```
v3 = 5.0 * v2;
v3 = v1 * v2;
```

The following, however, will generate a compile time error:

```
v3 = v2 * 5.0;
```

```
main.cpp: In function ‘int main()’:
main.cpp:38: error: no match for ‘operator*’ in ‘v1 * 5.0e+0’
vec.h:54: note: candidates are: vec_t operator*(double, const vec_t&)
vec.h:63: note:                 vec_t operator*(const vec_t&, const
vec_t&)
```

Further overloading of << and >>

We saw in last week's lab that the overloaded operators << and >> could be used to print and to read numeric and character string values to the *stdout* and *stderr* and from the *stdin.* True to form they can be further overloaded to print and read a complete *vec_t.* We want to be able to do something like

```
cout << v1;
```

where v1 is a vector. Since the left hand side is not a *vec_t,* we must use the *friend* function form. So in *vec.h* we include in the *vec_t* class definition:

```
  friend ostream &operator<<(ostream &out, const vec_t &pvec);
```

We provide the implementation in either *vec.cpp* or inline in the class definition.

```
ostream &operator<<(ostream &out, const vec_t &pvec)
{
   out << pvec.x << ", " << pvec.y << ", " << pvec.z << endl;
   return(out);
}
```

This is the key to cascading application of the << operator

Note that our *new overload just uses the built in overload of* << to output each component of the vector along with punctuation and a '\n'.

Cascading *cout*

Our implementation will also let us print an arbitrary number of vectors:

```
cout << v1 << v2 << v3;
```

This seemingly magic behavior occurs because of two things:

- C++ evaluates a sequence of << operations in *left to right order*
- The operator<< function *returns its left side operand as its result.*

So what REALLY happens is: `cout << v1` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```
cout << v2 << v3;
```

Then `cout << v2` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```
cout << v3;
```

Then `cout << v3` is evaluated and the value it returns is "dropped on the ground", because there is nothing to assign it to.

The finite plane

The finite plane is a rectangular region of finite area within a general plane. It is properly implemented as a derived class of the general plane. It is specified in the following way:

```
fplane origin
{
   material white            object_t constructor

   normal 1 0 1
   point  0 0 0              plane_t constructor

   xdir   1 0 0
   dimensions 4 2            fplane_t constructor
}
```

The three sets of attributes must be specified in the order shown because of the way in which the constructors are executed. The order *within* each set of attributes is arbitrary.

In retrospect a better way to have built the generalized parser would have been to have *built the parse control table dynamically.*

The two new attributes of the *fplane_t* are its *dimensions* in world coordinate units. The first dimension is the size in the *x* direction and the second the size in the *y* direction. The *x* direction is indirectly specified by the *xdir* attribute. The *xdir* attribute is a vector which *when projected into the infinite plane* specifies the *x* direction of the rectangle. The *y direction* is implicitly given by the cross product of the unit plane normal and the unitized, projected *xdir*.

The fplane_t class definition

```
class fplane_t: public plane_t
{
public:
   fplane_t();
   fplane_t(FILE *, model_t *, int);

   virtual double  hits(vec_t *base, vec_t *dir);
   virtual void    dumper(FILE *);

private:

   mat_t  rot;        /* rotation matrix            */
   vec_t  projxdir;   /* projected unitized xdir    */
   vec_t  xdir;       /* input xdir                 */
   double dims[2];    /* input dims in world coords */

};
```

The *fplane_t* constructor:

attrmax = 2 tells plane_t constructor to consume only 2 attributes

```
/**/
fplane_t::fplane_t(
FILE      *in,
model_t *model,
int        attrmax) : plane_t(in, model, 2)
{
```

The constructor should perform the following actions:

- set the *objtype* to "fplane"
- parse the *xdir* and *dimensions* attributes
- project *xdir* onto the plane creating *projxdir.*
- *ensure* that *projxdir* is not {0.0, 0.0, 0.0}
- make *projxdir* unit length.

Next it is necessary to make the *rot* matrix that can rotate the *projxdir* vector into the x-axis and the plane normal into the positive z axis.

- copy *projxdir* to row 0 of *rot*
- copy the plane *normal* to row 2 of *rot* and make it unit length.
- set row 1 of *rot* to row_2 x row_0.

The *fplane_t dumper*:

```
 /***/
void fplane_t::dumper(
FILE *out)
{
```

plane_t::dumper(out);

The dumper should produce a formatted output of the xdir, projxdir, and dimensions.

The *fplane_t hits* function

```
/***/
double fplane_t::hits(
vec_t    *base,      /* ray base              */
vec_t    *dir)       /* unit direction vector */
{
   vec_t newloc;
   double t;
```

In general, determining if a ray hits a rectangular finite plane of arbitrary location and orientation seems like a difficult problem. The *hits* function of the standard plane can certainly be used to determine if and where the ray hits the *infinite* plane in which the rectangular plane lies.

- If the infinite plane is missed then clearly so is the finite plane.
- If the infinite plane is hit, the problem is determining whether or not the hit point is "in bounds" or "out of bounds.

If the lower left corner of the finite plane happened to lie at the origin, and the projected *xdir* happened to lie on the *x-axis,* and the plane normal happened to lie on the z-axis, the problem would be easy. The hit is in bounds if and only if:

0 <= hitloc.x <= dims[0] *and*
0 <= hitloc.y <= dims[1]

We can make it possible to perform this simple test if we *translate* the *point* defining the lower left corner of the plane to the origin and then *rotate* the plane into the proper orientation. Therefore after we apply these operations to the original *hitloc* we can make the simple test above work. So that lighting will still work *we cannot modify the original hitloc.*

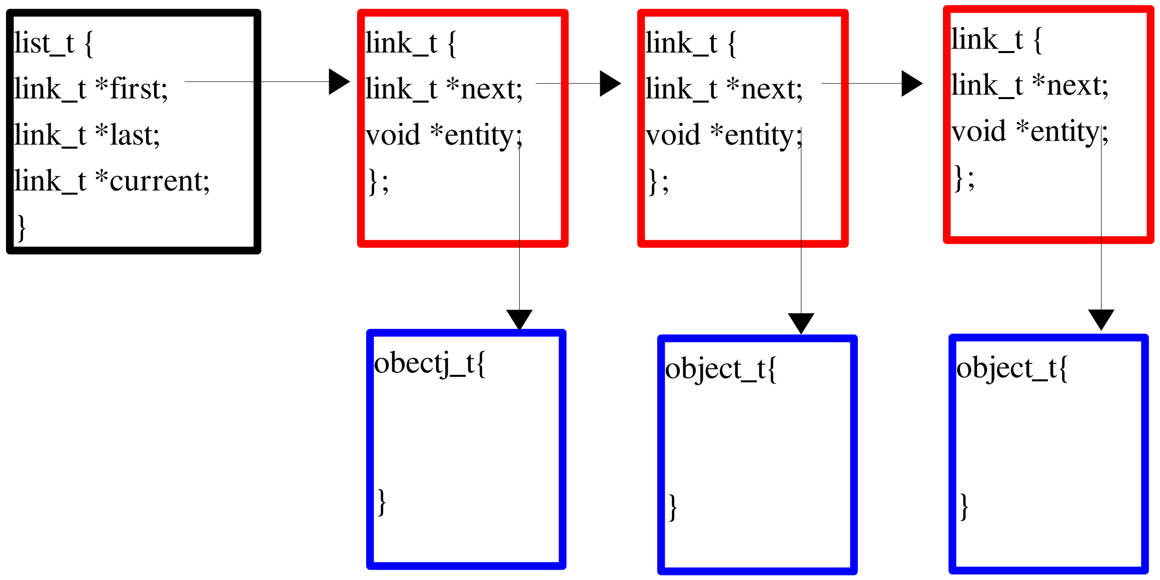
So we set *newloc* to *hitloc – point.* Then we rotate *newloc* by transforming it with the *rot* matrix. Then we can do the test:

0 <= newloc.x <= dims[0] *and*
0 <= newloc.y <= dims[1]

A C++ list structure for object management.

We describe a way in which the list structure we have been using might be converted to C++. In the new structure we will bury some information so that high level users of the list mechanism need to know only about *list_t* methods. The *link_t* objects can remain invisible to the outside world.

The basic structure is very similar to our C language version. The only difference is that the *list_t* structure contains a new element of persistent state: *the current* pointer maintains the *current* location during list traversals. As we shall see this is problematic.



Example linked list classes

The *link* and *list* classes are correctly declared below: Note that the *default* constructor is explicitly provided and that it is overloaded.

Instances of the *link_t* class represent the *static structure* of the list. The values of *next* and *obj* in an instance of the *link_t* class are set by the constructor. The value of *obj* *never* changes thereafter. The value of *next* is initialized to 0, changes to non-zero when the *link* becomes *not* the last element in the list and *never changes again.*

```
class link_t
{
public:
   link_t(void);                     // default constructor
   link_t(void *);                   // overloaded constructor
   ~link_t (void);
   void    set_next(link_t *);       // used in adding new link
   link_t *get_next(void);           // retrieve the next pointer
   void   *get_entity(void);         // retrieve entity pointer

private:
   const link_t *next;               // next link in the list
   void   *entity;                   // entity managed by link
};
```

While a list is being processed the *dynamic state (where we are in the list)* is maintained in the *current* pointer of the *list_t.* This *feature* can be problematic if nested processing of the list is to occur.

```
class list_t
{
public:
   list_t(void);                      // constructor
   ~list_t (void);
   void   list_add(void *);           // add entity to end of list
   void   *list_start(void);          // set current to start of list
   void   *list_next(void);           // advance to next element in list

private:
   link_t *first;                    // first link
   link_t *last;                     // last link
   link_t *current;                  // current link.
};
```

link_t methods

This constructor is passed a pointer to the *obj_t* which this new link will own. It should set the *next* pointer to NULL and set the *entity* pointer to the entity being passed in:

```
link_t::link_t(void *newent)
{


}
```

The destructor should *free* the entity owned by the link. It *does not free the link_t.* That is done by the *delete* facility.

```
link_t::~link_t(void)
{
   fprintf(stderr, "Deleting %p\n", this);
}
```

The *set_next()* method is a typical "set" function that is used to tell the *link_t* to manipulate its own *next* pointer. It is called by the *add* method of the *list_t* class when an item that is not the first item is added to the list. It should set the *next* attribute of the *link_t* to *new_next;*

```
void link_t::set_next(link_t *new_next)
{


}
```

link_t *getter* methods

The *get_next()* method is a typical “get” function that is used as a way to tell the *link_t* to return the value of own *next* pointer.

```
link_t * link_t::get_next()
{


}
```

The *get_object()* method is analogous. It would also work to simply make all of the *next* and *obj* elements *public*. Then any holder of a reference to the *link_t* could simply manipulate them directly... but it would be a *violation* of OO dogma to do so. It should return the *entity* pointer.

```
void * link_t::get_entity()
{


}
```

***list_t* class methods**

The *list_t* class overrides the default constructor with its own constructor with no parameters:

```
    list_t::list_t()
    {
       first = NULL;
       last =  NULL;
       current = first;
    }
```

The *list_t* destructor must *process the entire list* and *delete* each *link_t* instance. As in the C version, it is mandatory that the *address of the next link_t be remembered before the current one is deleted.*

```
list_t::~list_t(void)
{

}
```

Adding a new object to the list

The *list_add()* method creates a new *link_t* and passes its constructer a pointer to the entity to be attached to the *link_t*.   The *link_t* constructor returns a pointer to the new *link_t*.

If this is the *first* item added to an empty list, the *list_add* method should set the *first, last* and *current* pointers to the new link.

Otherwise, the *next* pointer of the existing *last* link should be set to point to the new *link_t* and the *last* pointer of the *list_t* should be set to the new *link_t*.

```
    void list_t::list_add(void *entity)
    {
       link_t *link;




    }
```

Retrieving the *first* element of the list

If the *list* is empty the *list_start()* method should return NULL. Otherwise, the *list_start* method sets the *current* pointer to the first *link* in the list and returns a pointer to the first *entity* in the list.

```
void * list_t::list_start(void)
{

}
```

Retrieving the next *obj_t* in the list.

The *next* method attempts to advance the *current* pointer. If the *current* pointer is already at the end of the list *NULL* will be returned.   The use of the *persistent state variable current* will prove to be something of a pain in nested processing of the list .

```
obj_t * list_t::list_next(void)
{
   link_t *link;


}
```

Using the list_t class

Creating a new *list_t*

```
list_t *list;
list = new list_t();
```

Creating a new object and adding it to the list:

```
sphere = new sphere_t(...);
list->list_add((void *)sphere);
```

Deleting a *list* along with *all* the links and entities associated with the list:

```
delete list;
```

Processing a list

The *start* method is used to set the internal *current* pointer to the internal *first* pointer and returns a pointer to the first object in the list.

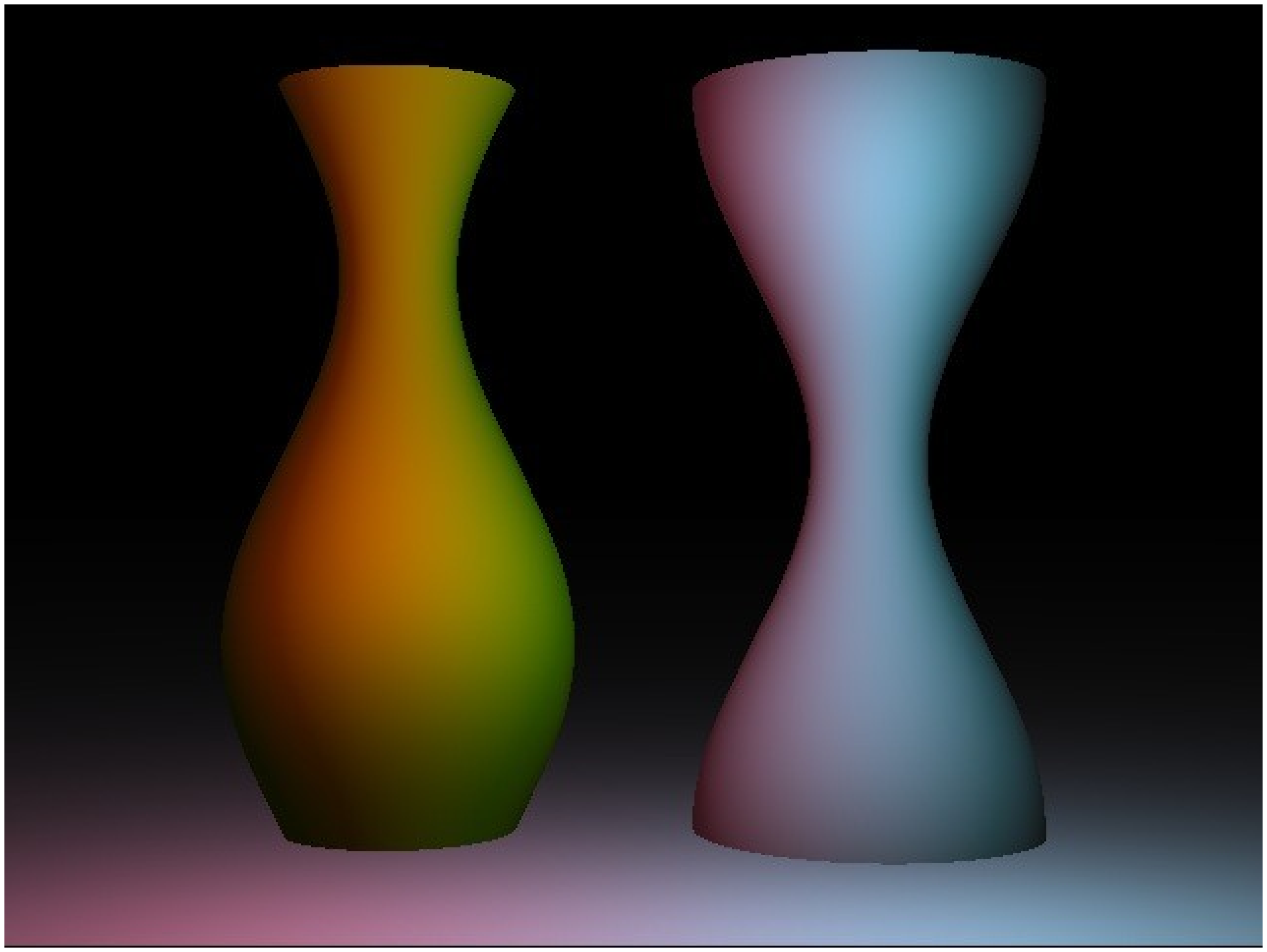
The *next* method is used to advance *current* to point to the next *link_t* and return the *obj_t* pointed to by the new *link_t*. If *current* already points to the end of the list then *NULL* is returned.

```
list_t *list = model->objs;
object_t  *obj;
 :
obj = (object_t *)list->start();
while (obj != NULL)
{
   obj->dumper(out);
   obj = (object_t *)list->next();
}
```

This works well *unless* inside the loop there is a call to an inner function that also need to process the list.   If the inner function uses the same *list_t* as the outer one, it will leave the value of *current* at *last* breaking the caller.   We will return to this issue later.

Surfaces of revolution

Surfaces of revolution greatly expand the range of shapes that we can easily create. The objects shown below are surfaces of revolutions created using *sine* and *cosine* functions. The object on the left is gold in color and the object on the right is gray. The scene is illuminated by three lights. The one on the left has a red tint, the one in the center has a green tint, and the one on the right blue.



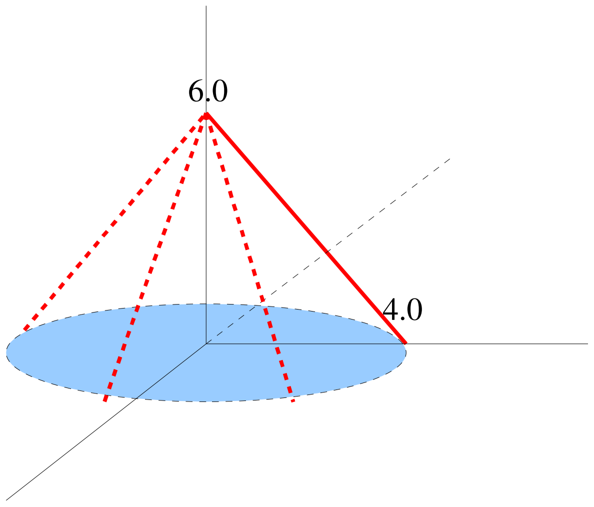
An intuitive motivation

Consider the red line in the diagram below. When $x = 0$, $y = 6.0$ and when $x = 4.0$, $y = 0$. Thus the slope of the line is -1.5 and equation of the red line is given by:

$$y = -6x/4 + 6$$

For reasons that will become apparent we typically express *x* as a function of *y*. It is simple to solve the equation for *x*.

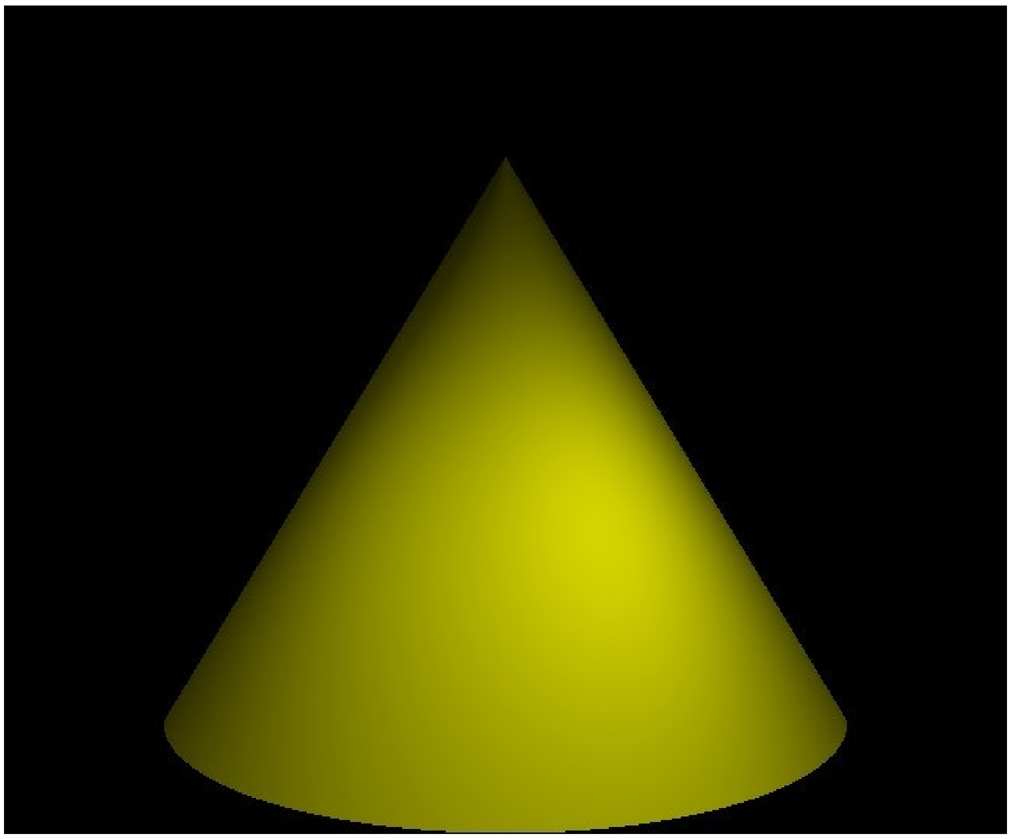
$$x = -4/6\ (y - 6)$$



The term *surface of revolution* is used to describe the surface that is created when the line is dragged around the blue circle (which appears as an ellipse when it is projected upon the "floor"). The surface that is created is a *cone* with vertex at location (0.0, 6.0, 0.0).

Example input and output (*light definitions missing*)

```
camera cam1
{
  pixeldim  640 480
  worlddim  8 6
  viewpoint 4 3 7
}
material gold
{
   ambient 1 1 0
   diffuse 3 3 0
}

revsurf cone
{
   material gold
   surfer 1
   base 4 0 -4
   direction 0 1 0
   height 6
}
```



Determining an equation for the surface

The first step in developing a *hits* function is to develop an equation that describes the surface. Consider the circle that is produced by dragging the point $x = 4$, $y = 0$ around the y axis. This operation produces a circle of radius 4 in the $y = 0$ plane. That is, x *and* z vary but y does not. The equation of this circle is:

$$x^2 + z^2 = 4^2$$

As we move along the red line, the distance of the points along the line from the y axis are given by:

$$x = -4/6\ (y - 6)$$

Thus any point on the cone must satisfy the equation:

$$x^2 + z^2 = (-4/6\ (y - 6))^2$$

Which may be written as:

$$f(x, y, z) = x^2 - (-4/6\ (y - 6))^2 + z^2 = 0$$

This is a quadratic equation. If V is the viewpoint and D the ray direction We can replace (x, y, z) by $v_x + td_x \;\; v_y + td_y \; v_z + td_z$ and solve for t in the usual way. However, we don't want to restrict ourselves to quadratic equations. And we definitely don't want to be limited to surfaces that resolve around the origin.

Mor complex surfaces

For example the gold vase shown earlier was produced by revolving the line:

$$x = 2 + sin(y)$$

Applying the approach of the previous page we see the equation of this surface is:

$$f(x, y, z) = x^2 - (2 + sin(y))^2 + z^2 = 0$$

In general given the equation of any line in the form

$$x = h(y)$$

The equation of the corresponding surface of revolution is:

$$f(x, y, z) = x^2 - (g(y))^2 + z^2 = 0$$

and if a ray this the surface at a point (x, y, z). The normal at the hit point is given by the componentwise (partial) derivatives of the function or:

$$(2x, \; -2(g(y))g'(y), \; 2z)$$

Thus a surface normal on the gold vase at location (x, y, z) is

$$(2x, \; -2(2 + sin(y)) \, cos(y), \; 2z)$$

Determining if a ray hits a surface of revolution.

The "front end" of our approach
Assume the following:

V = *viewpoint or start of the ray*

D = *a unit vector in the direction the ray is traveling*

C = *base point (y = 0) of the surface*

h = *desired height of the surface of revolution.*

$g(y)$ = *the generating function*

The arithmetic is much simpler if the base point of the surface is at the origin (as it was in the preceding examples. So we start by moving it there! To do so we must make a compensating adjustment to the base of the ray.

$$C' = C - C = (0, 0, 0) = \textit{new center of sphere}$$

$V' = V - C$ = *new base of ray*

D *does not change*

A point P on the translated surface with base at (0, 0, 0) necessarily satisfies the following equation:

$$p_x^2 - g(p_y)^2 + p_z^2 = 0 \qquad (1)$$

All points on the ray may be expressed in the form

$$P = V' + t\,D = (v'_x + td_x,\ v'_y + td_y,\ v'_z + td_z) \qquad (2)$$

where t is the Euclidean distance from V' to P

Thus we need to find a value of t which yields a point that satisfies the two equations. To do that we take the (x, y, z) coordinates from equation (2) and plug them into equation (1). In this equation t is the only unknown quantity but unless $g(y)$ is a linear function of y (as was the case with the cone) we do not have a quadratic equation.

$$(v'_x + td_x)^2 - (g(v'_y + td_y))^2 + (v'_z + td_z)^2 = 0$$

Numerical solution of the *hits* equation.

Given an arbitrary function $q(t)$ we need a method for finding a value of t for which $q(t) = 0$.
If we can do that, then we can define $q(t)$ as follows and we are done.

$$q(t) = (v'_x + td_x)^2 - (g(v'_y + td_y))^2 + (v'_z + td_z)^2$$

This is actually a pretty difficult problem because in general there my be multiple values of t. For

$$q(t) = sin(t)$$

setting t any multiple of *pi* will produce $q(t) = 0$.

There may also be no values of t at all as in the case of

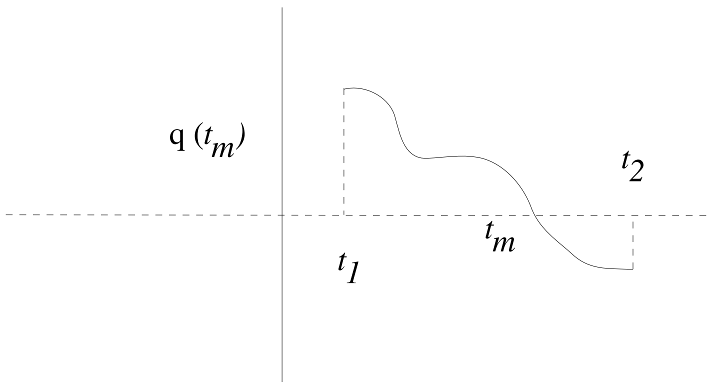
$$q(t) = t^2 + 1$$

where the minimum value of the function is $q(0) = 1$.

Nevertheless we can say that: (1) if $q(t)$ is a continuous function *and* (2) we can find two points t_1 *and* t_2 such that $q(t_1)\, q(t_2) < 0$, then we can find a point t_r such that $q(t_r) \sim= 0.0$.

The method that we will use is based upon the *binary search* technique and is commonly called *bisection.*

The *bisection* algorithm



```
while(t2 – t1) > epsilon)
 {
        compute tm = the midpoint between t2 and t1
        if (q(t1) * q(tm)) < 0)
           replace t2 with tm
        else
           replace t1 with tm
  }
```