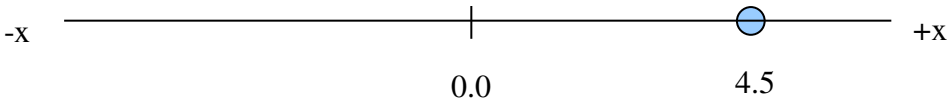
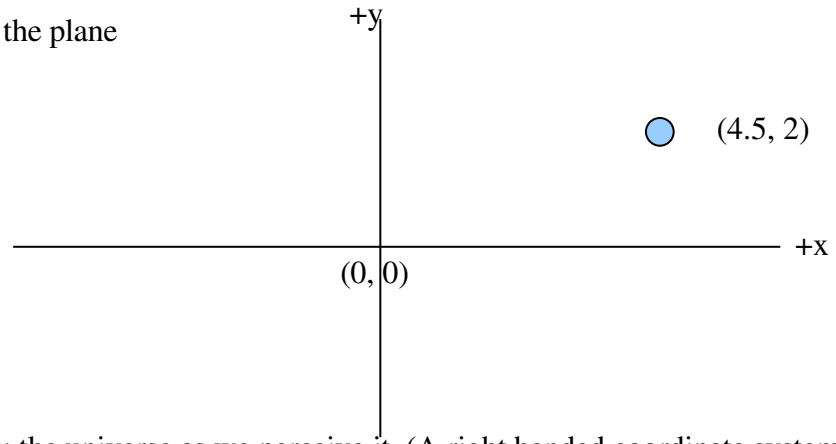
**Basic elements of 3-D coordinate systems and linear algebra**

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.

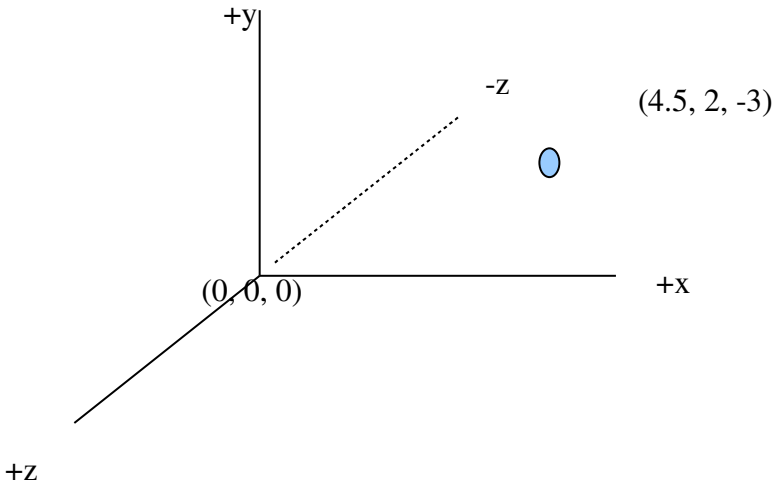
One dimension: the line




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: $\text{sqrt}(25 + 16 + 25) = \text{sqrt}(66) = 8.12.$

**The geometric interpretation of vector arithmetic**

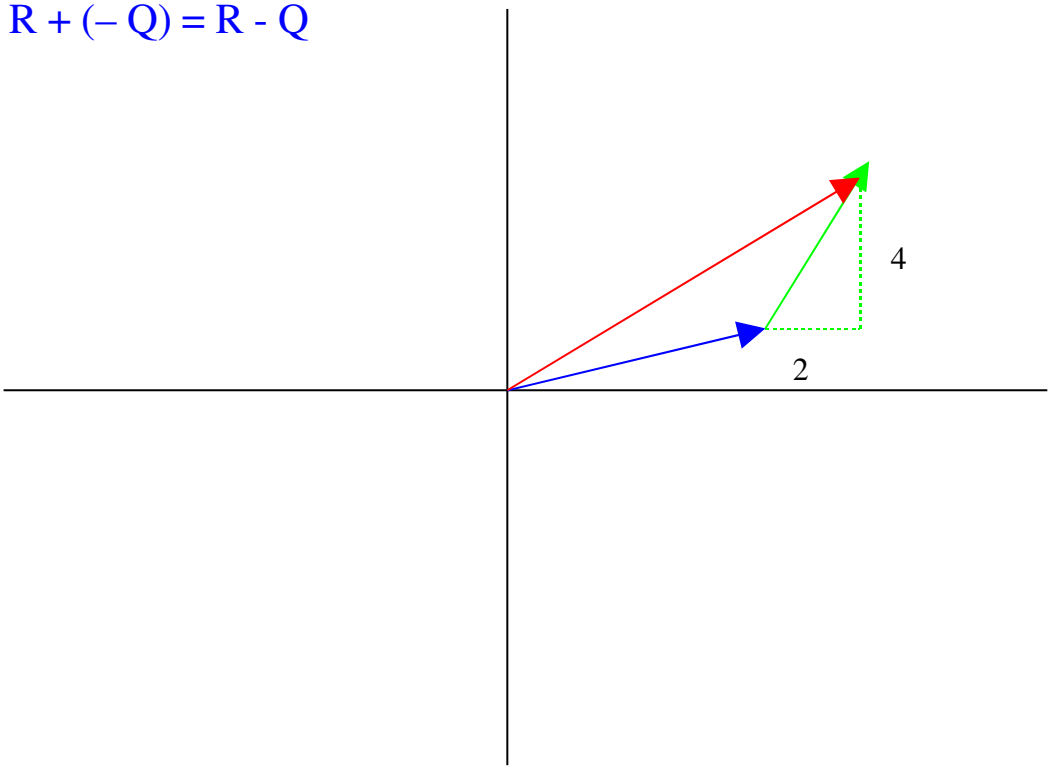
Here we work with 2 dimensional vectors to simplify the visual interpretation, but in 3-d the principles are the same.

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) = R - Q



**Useful operations on vectors:**

We define the sum of two vectors $P$ and $Q$ as the componentwise sums:

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

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

The difference of two vectors is computed as the componentwise differences:

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

*(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.* It is computed by taking the sum of the componentwise products of the two vectors.

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

*(2, 3, 4) dot (3, 2, 1) = 6 + 6 + 4 = 16*

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

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

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

**Representing vectors in C**

There are at least two easy 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.*

Because elements of a structure are guaranteed to be packed into adjacent memory elements its possible to cheat and use either array or structure notation.

```
vec_t  v = {1.0, 2.0, 3.0};
double *w = (double *)&v;

printf("%lf %lf %lf\n",  v.x, v.y, v.z);
printf("%lf %lf %lf\n",  w[0], w[1], w[2]); ;
```

```
1.000000 2.000000 3.000000
1.000000 2.000000 3.000000
```

**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.*

**A correct version of *vec_unit()***

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 ***must*** work correctly with aliased parameters.

**A sample test driver for *vector.h***

```
/* 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

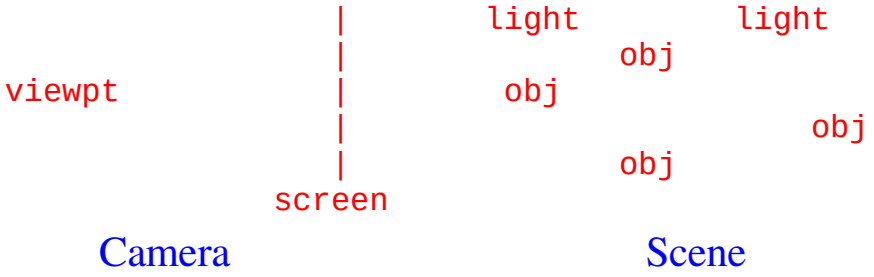
2 - *The screen* This defines a virtual *window* through which the viewer observes the scene. The window can be viewed as a discrete 2-D pixel array (pixmap) . The *raytracing* procedure computes the color of each pixel. When all pixels have been computed, the *pixmap* is written out as a .ppm file

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.

*Possible 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, 0.0). (Recall the screen lies in the $z = 0$ plane. Therefore the $z$ world coordinate of every point in the window is 0.0).

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)*
*world_y = world_size_y * win_y / (win_size_y - 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:

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

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 (malloc()) pixmap data buffer*
*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 double (r, g, b) values. Values must be chosen in an *ad hack* manner.

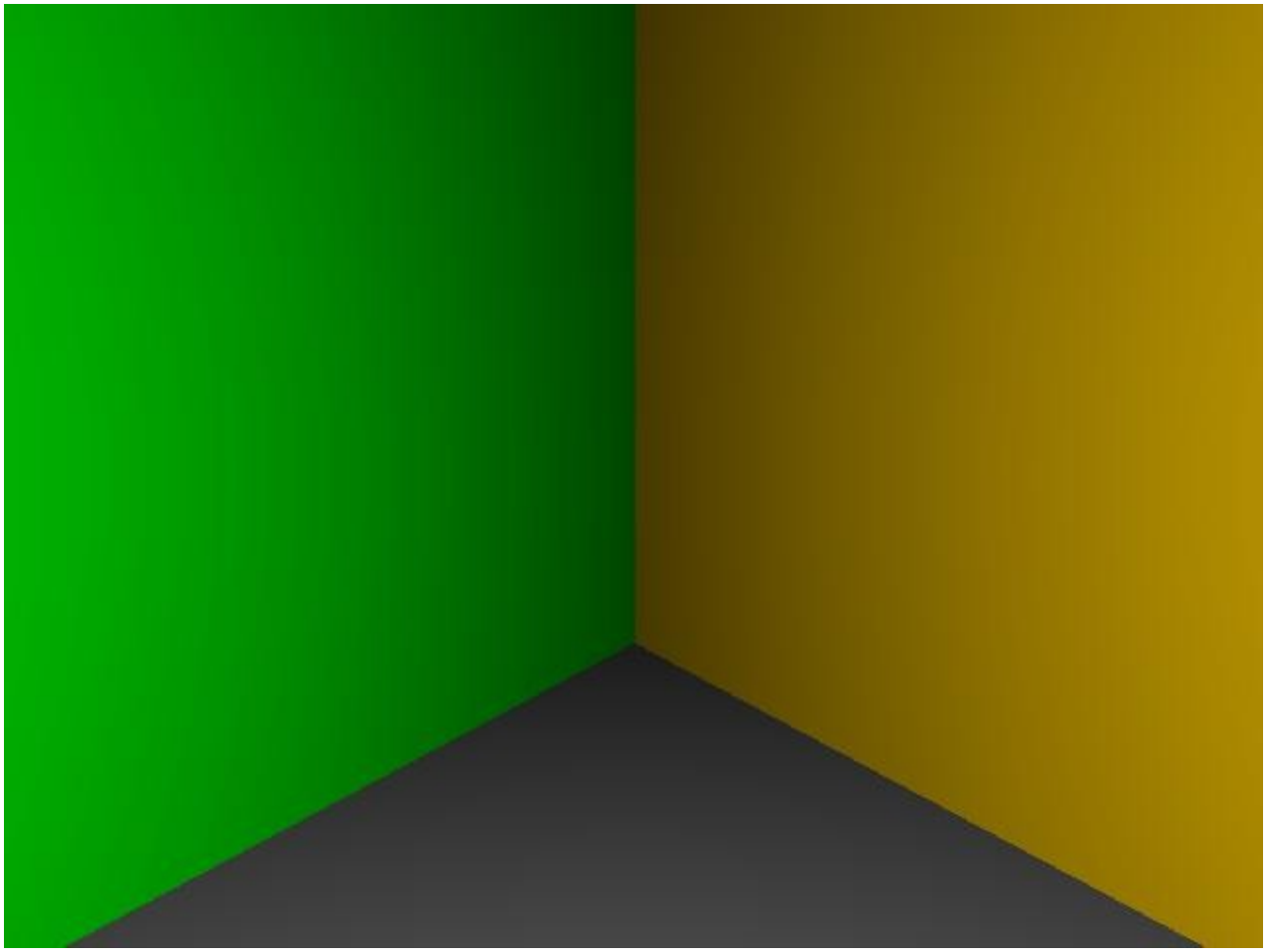
Our lighting model assumes 3 types of reflectivy: ambient, diffuse, and specular. Initially only ambient will be implemented.

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 in the direction of the *viewpoint.*

**The output image**

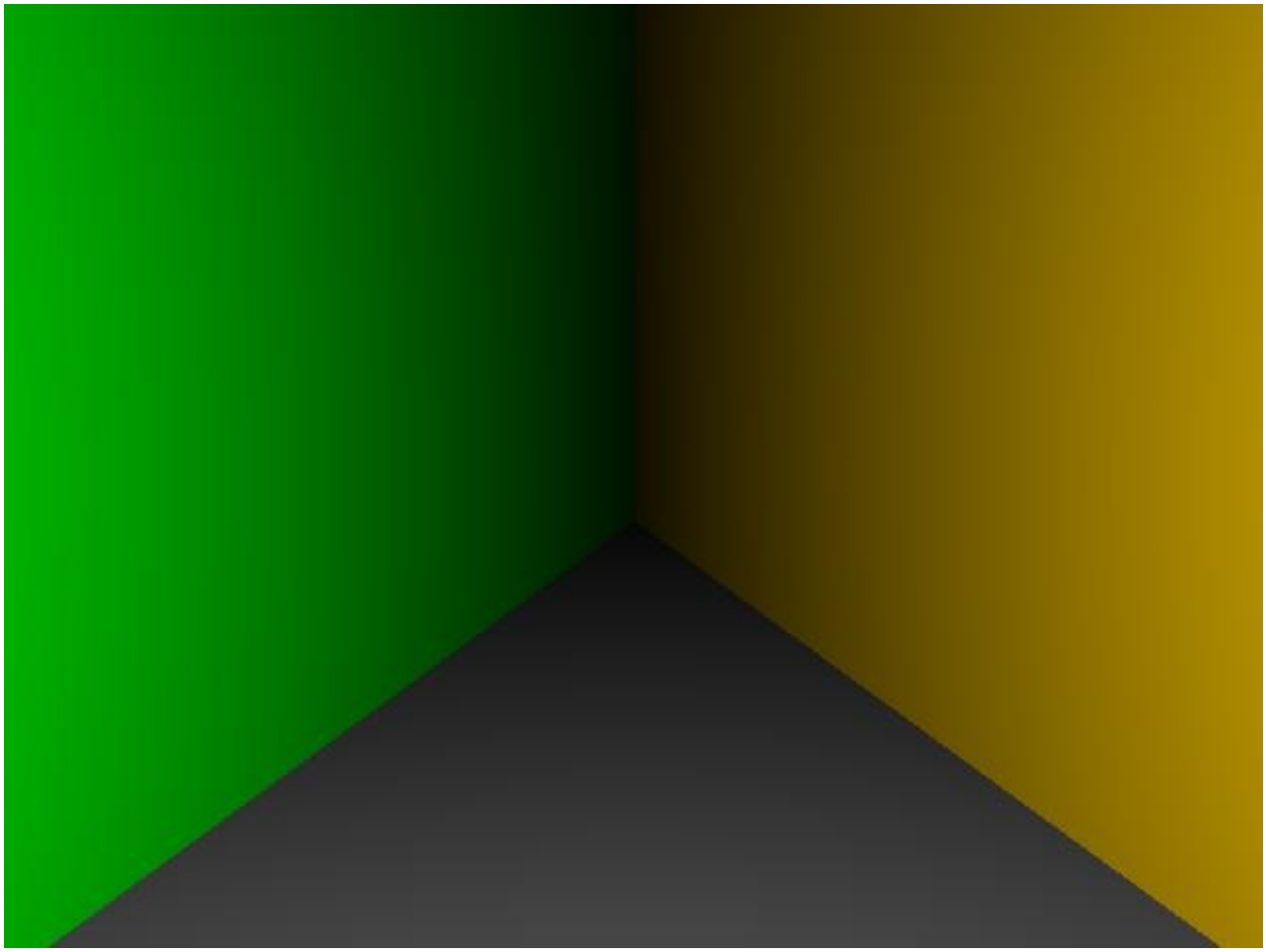
The output produced by the input file on the previous page is shown below. Visible image corruption near the green-gray boundary courtesy of JPEG compression.

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 an 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.

**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**

- Each "sentence" in our language begins with an an *entity-type* identifier.
- Our *entity-types* will include *camera, material, light, plane, sphere, etc.*
- The *entity-type* is followed by user defined and arbitrary *entity-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 *entity-name* is followed by a collection of *entity-attributes* enclosed in { }.
- Each *entity-attribute* consists of an *attribute-type* specifier followed by attribute values.

- 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 any entity may appear in *any order.*

Attribute values map to the data structure associated with a particular entity-type in an obvious way:

```
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    */
```

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)
     :
}

etc.
```

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*.

**Constructors and parsing**

In object oriented languages, each object type has an associated *constructor* function that is called each time a new instance of the object type is created. Therefore, if a raytracing model contains two planes, the plane constructor will be invoked twice.

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

We will 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;

```
/* this function is invoked when the parser driver     */
/* reads an entity-type which has the value "aircraft" */
/* It then calls the aircraft_init() constructor to    */
/* create an instance of the aircraft_t stucture and   */
/* to load the attributes of this aircraft.            */

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;

/* The word "aircraft" has already been read so here we */
/* acquire the name of the aircraft (f18-1) */

   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(ac_attrs[0]))
```

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 table_lookup(
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 = table_lookup(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 14555678  /* quasi-random cookie values */
#define MAT_COOKIE 32434323  /* used to verify that struct */
#define LGT_COOKIE 30434344  /* pointers are what they     */
#define CAM_COOKIE 49324423  /* pretend to be!             */
#define MOD_COOKIE 34321564

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

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

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

/* MUST COME LAST */

#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.

**The *model_t* data structure**

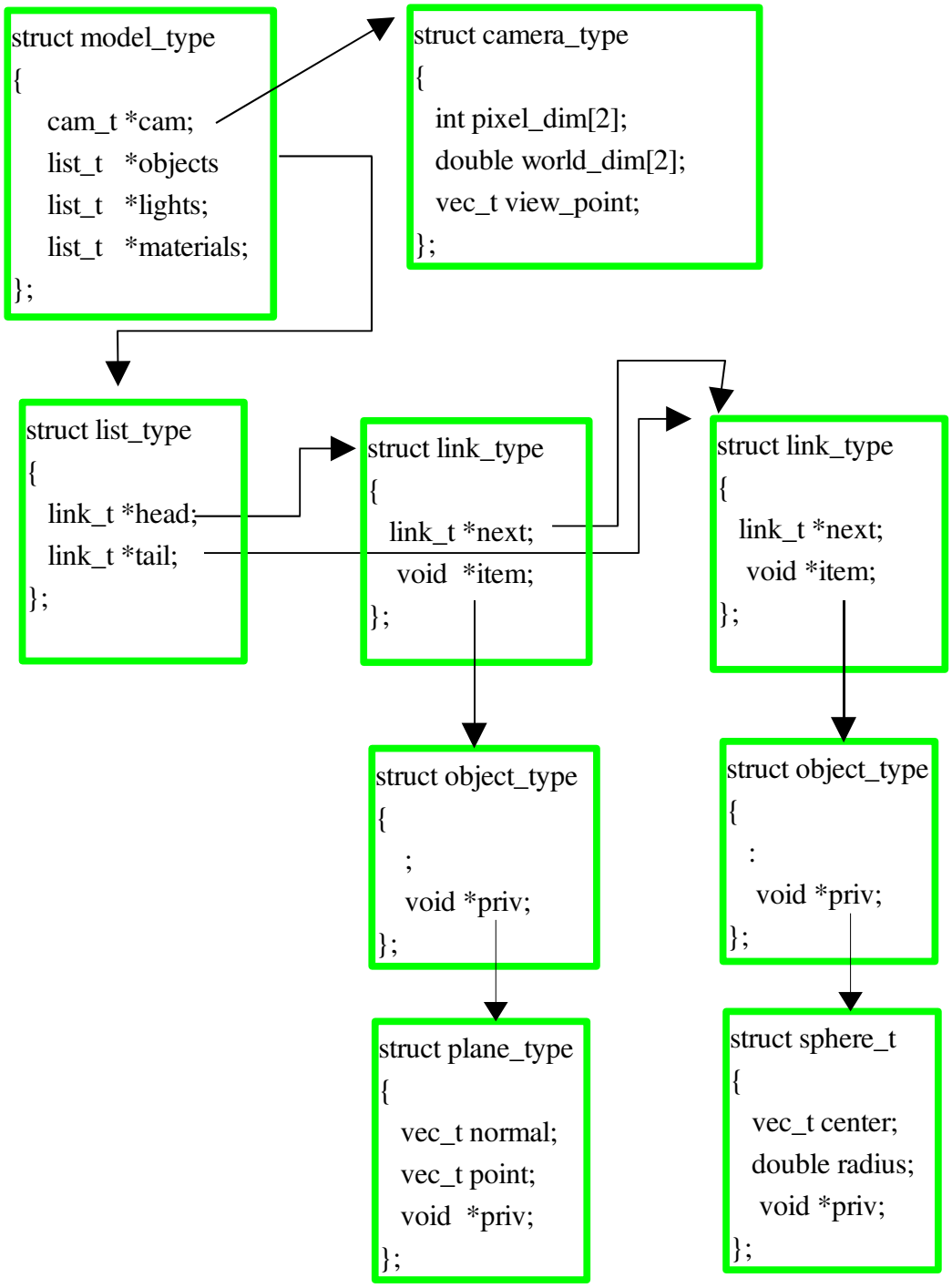
This structure is a *container* used to reduce the number of parameters that must be passed through the raytracing system. It lives in *ray.h*.

```
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;
```

**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).
5 - We want a single generic structure that can manage lists of the three different types.

The two structures shown below are sufficient.

There is a single instance of the *list_t* structure for *each list.*
There is a single instance of the *link_t* structure for *each element in each list.*

```
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**

For now, our list management module (to be constructed in lab) will include three functions.

The *list_init()* function is the constructor that 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.* Its mission is to:

1 - *malloc()* a new instance of *link_t,*
2 - add it to the tail of the list,
3 - ensure the *next* pointer of the new link is NULL and
4 - ensure the *next* pointer of the *link_t* that used to be at the end of the list points to the new *link_t*

Two cases must be distinguished:

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

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

}
```

**Deleting a list**

The *list_del()* function. This function should process the entire list. For each link in the list, it should

1 - invoke the *free()* function to free the *item* the link owns and then
2 - it 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**

This code segment shows

- how to define an arbitrary structure that might be managed by the list
- how to ask list init to create a new list.
- how to process the list from head to tail

```
   typedef struct entity_type
   {
       char e_name[16];
       int  e_id;
   } e_t;

   list_t *elist;
   link_t *elink;

   elist = list_init();

/* Load the list */

   load_my_list(elist);

/* Now traverse the list printing attibutes of the elements */

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

**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)

**The *main*() function**

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 the model */

   model = model_init(stdin);
   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* or *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 (*or its subordinate esoterics.*)* 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 */


/* 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**

For the scene specification

- *material, object, and light* data may be intermixed
- the *camera* definition may appear anywhere
- *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 *camera, material, object*, and *light* definitions. When it completes, the *cam* pointer must point to a complete cam_t structure and 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));
   model->cookie = MOD_COOKIE;

/* 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 camera, materials, objects, lights */

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

**The design of the model loader**

The model loader operates in a way similar to but not exactly like the CPSC 101 parser. In fact the model loader itself is simpler than the CPSC 101 parser *because it depends upon entity specific constructors* to

- create camera, object, light and material structures and
- read and store attribute data

The model loader like the CPSC 101 parser relies upon a table lookup strategy in which the lookup table is defined as shown below:

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

};

#define NUM_ENTITIES (sizeof(entities) / sizeof(entities[0]))
```

**The *model_load()* function**

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

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

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

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

      lookup entity name in entities table
      invoke entity specific initializer (camera_init, 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)
{
   camera_dump(out, model->cam);
   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 *drgb_t (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)
{
    int ndx;
    ndx = lookup attrname in attribute name table
    assert(ndx >= 0);
    if (ndx == 0)
       read ambient values
    else if ....

    else

    return(0);
}
```

**The *material_search()* function**

This function must search the list of materials looking for the color specified in the name parameter:

```
material_t *material_search(
model_t *model,
char    *name)     // e.g. orange
{
   material_t *mat;

    for each mat  in the model->mats list
    {
       if (mat->name matches name)
            return(mat)
    }
    return(NULL);
}
```

**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 *);

    :
   void    *priv;              /* sub-object (plane_t) */
}
```

The *hits* and *dumper* pointers must be set in at initialization time. The *plane_init* function must set these pointers as follows. These pointers allow the object to behave in a way that is called *polymorphically* in an O-O language.

```
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(ray_base, ray_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, "camera") == 0)
   camera_init(in, model, 0);
else 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 ......
```

or

```
ndx = table_lookup(entities, NUM_ENTITIES,
                                    entity_type);
assert (ndx >= 0);
switch (ndx)
{
case 0:
    camera_init(in, model, 0);
    break;
 case 1:
   material_init(in, model, 0);
   break;
case 2:
```

**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 *table_lookup()* is used to search the name table.

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

static void (*loaders[])(FILE *in, model_t *model, int max) =
{
   (void *)camera_init,
   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 = table_lookup(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[] =
{
   {"camera",       (void *)camera_init},
   {"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(items[0]))
```

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 *table_lookup()*.

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

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

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

   return;
}
```

The new *table_lookupi* function.

```
static inline int table_lookupi(
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 responsibilities that are described below. In a true object oriented language, the plane_type will be a specialization of the object_type and when a new instance of plane_type is created the constructors for BOTH the plane and object classes will be AUTOMATICALLY invoked in top down order.

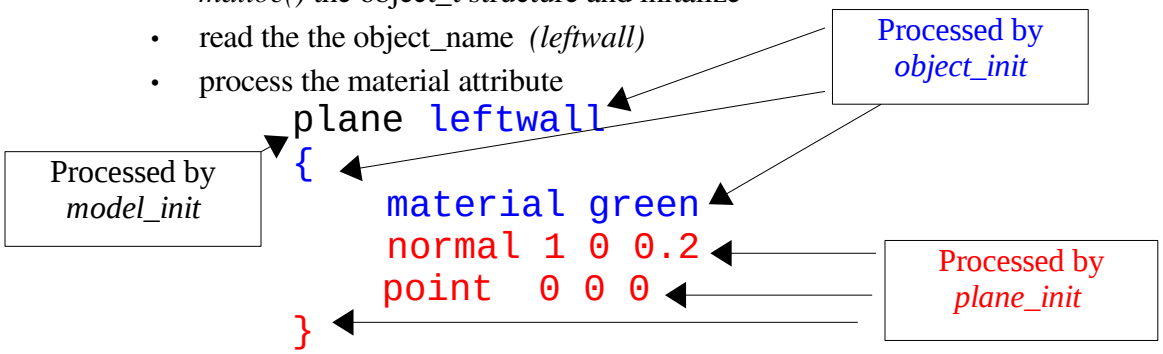
When simulating inheritance with C we must

- explictly invoke the constructors or each element in the hierarchy and
- link the structures that represent them together.

In both C and C++ this will *implicitly require that generic attributes appear first* in the model definition.

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

- invoke *object_init() which will*
  - *malloc()* the object_t structure and initalize
  - read the the object_name *(leftwall)*
  - process the material attribute




- 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_search(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 ask the object to print the type specfic attributes. For the plane type object, *obj->dumper* will point to the actual function *plane_dump*.

```
obj->dumper(out, obj);
```

the *plane_dump()* function will be responsible for printing the specific attributes of a plane.

```
normal 1 0 0.2
point  0 0 0
```

}

```
}
```

**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 *table_lookup* 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.*

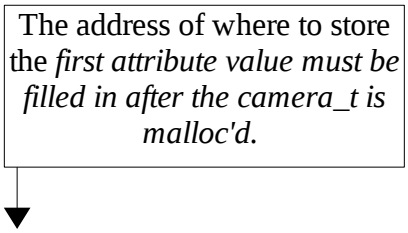
**The *struct pparm_type***

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**

This version of *camera_init()* demonstrates how the use of the general parser can reduce your workload.

```
void camera_init(
FILE *in,
model_t *model,
int  attrmax)
{

/* malloc the camera structure */

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

/* Read camera name and { */

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

/* Store locations where attribute data should be read */

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

/* Invoke the parser */

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

/* verify required attributes read */

   assert(mask == 7);

/* remember address of camera structure */

   model->cam = cam;

}
```

```
/**/
/* 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;

/* table_lookupp is an updated version of table_lookup that */
/* takes parse control table pointer as input.              */

   ndx = table_lookupp(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 table_lookup 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,
};
```

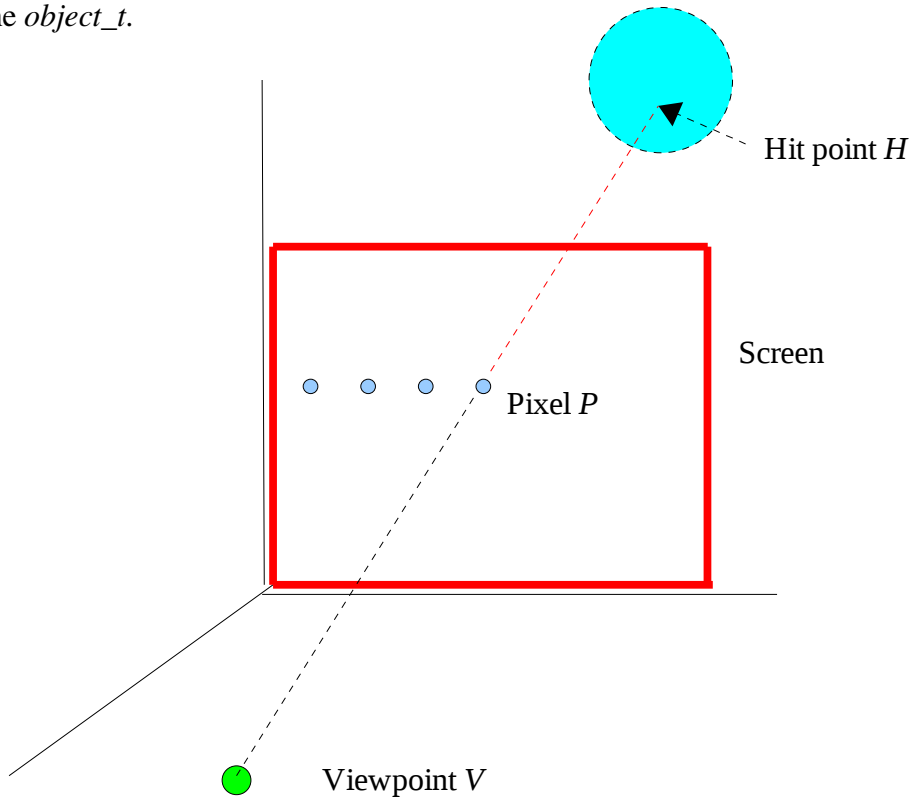
**Linking the model together.**

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  plane_hits(
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 $N$ is $(G, H, I)$ and
- J is chosen so that the plane passes through the specified point Q.

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.

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

and -1 should be returned.

**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.

$$C' = C - C = (0, 0, 0) = \text{new center of sphere}$$

$V' = V - C$ = *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:

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

Next we collect the terms associated with common powers of $t$

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

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:

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

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 */

   camera_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 d_pix = {0.0, 0.0, 0.0};
   cam_t  *cam = model->cam;
   int    i;

/* This function was written previously */

   camera_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, &d_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                   */

   camera_setpix(cam, x, y, &d_pix);

   return;
}
```

**The ray_trace() 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   *dpix,        /* 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**

The raytracer is sufficiently complicated that debugging output may be required for problem resolution. The C-compiler preprocessor *cpp* permits us to conditionally compile or not compile statements into a program. If we include the line:

```
#define DBG_DRGB
```

in the source code then this statement will be compiled and produce debugging output. But if we comment it out the debug output will not be produced.

```
#ifdef DBG_DRGB
   fprintf(stderr, "%-12s DRGB:(%5.2lf, %5.2lf, %5.2lf)",
                     closest->objname, pix->r, pix->g, pix->b);

#endif
```

Instead of having to comment in/out the definition, the C compiler allows you to define a symbol on the command line:

```
   gcc -c -g -DDBG_DRGB raytrace.c
```

**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 user written 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
```

**Definining debug control symbols in the *makefile***

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

```
CFLAGS = -DDBG_PIX -DDBG_HIT

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

$(RAYOBJS): $(INCLUDE) makefile

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

This code is in the *make_pixel()* function:

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

This code is in the *ray_trace()* function.

```
#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
```

Because of how the \n characters are used in the format string they work together to produce this useful output.

```
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 *camera_setpix* is to convert a *drgb_t* pixel encoding to an *irgb_t* pixel encoding and store it in the pixmap.

```
/**/
void camera_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**

Some of the code from the *write_ppm_image()* from you 101 project can be recycled here.

```
void camera_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

}
```

**Polymorphism - II:**

We have already discussed how the *hits* and *dumper* functions provide polymorphic bahavior in the *object* "class". Each specialization (plane, sphere) must provide its own characteristic function. There is no "default" hits function. A failure to provide a hits function will cause an instant segfault when the object is to be tested for a ray intersection. This bad behavior could be avoided by creating a function

```
double  object_hits(
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.        */
{   return(-1); }
```

in object.c and in the *object_init()* funciton setting the object's *hits* function pointer to point to this default function. Then *plane_inits()* could override the default behavior by storing an pointer to *plane_hits()* in *obj->hits*. But an object that "forgot" to override, wouldn't crash the raytracer, (but it would be invisible!)

**The *getamb()* function**

In other cases most specializations *may want to use the default* function. This will be the case with the material color retrieval function *getamb()*.

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 routine. We will see an example of this with the tiled plane object.

**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;
   obj->getspec = material_getspec;
```

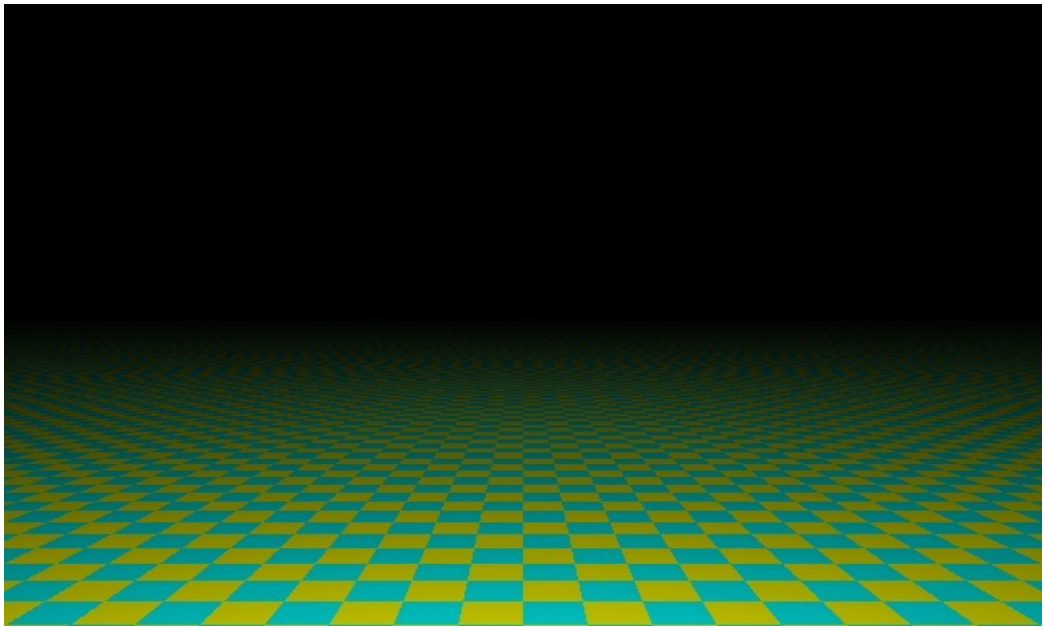
**Implementation of the default *getter* functions**

It is then necessary to implement the default functions in the material.c module. This *material_getamb()* and *material_getspec()* are analogous.

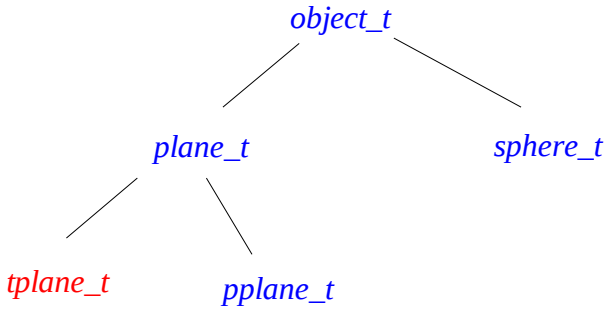
```
/**/
void material_getamb(
object_t *obj,
drgb_t   *dest)
{

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

**Tiled planes**

The tiled plane object is commonly found in ray tracing models.



We will use it to demonstrated how the inheritance heirarchy of specialization can be extended. In Object Oriented terminology the *object_t* structure is called a *base class.* The base class is at the top or root of a class hierarchy. The plane_t and sphere_t are *specializations* of the object_t and are called *derived classes.* The tiled plane *tplane_t* and the procedural plane pplane_t are further specializations of the *plane_t.*



**The tiled_plane model object description**

The tiled plane requires two attributes beyond those of the regular plane.

The dimensions specify the size of the tiling in world coordinates
The *altmaterial* specifies the color of the alternate tiles.

```
tiled_plane floor
{
    material white
    normal    0 1 0
    point     0 0 0
    dimension 2.0 3.0
    altmaterial green
}
```

Processed by object_init

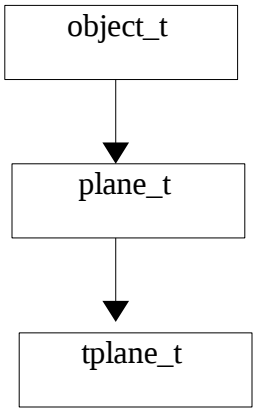
Processed by plane_init

Processed by tplane_init

A specialization *always* requires that *some* aspect of the behavior of the parent class be modified. If no modifications at all were required, we could just use the base class.

A specialization *never* requires that that *all* aspects of the parent's behavior be overridden. In that case it would be appropriate to create a new class.

When a new *tiled_plane* is created three structures must be allocated and linked together:




The object_init() function is called by plane_init(). It mallocs the object_t.

The plane_init() function is called by tplane_init(). It mallocs the *plane_t* and sets the *priv* pointer in the object_t

The tplane_init() function is called by the model loader. It mallocs the *tplane)t* and sets the *priv* pointer in the plane_t.

**The *tplane_t* structure and the *tplane_init()* function.**

The structure is shown below and it must be filled in by *tplane_init()* from the input data

```
/* Tiled plane... descendant of plane */

typedef struct tplane_type
{
   char       matname[NAME_LEN];
   material_t *background;   /* background color    */
   double     dimension[2];  /* dimension of tiles  */
}  tplane_t;
```

The dimension parameter specifies the (x, z) dimensions of a single tile in world coordinates. The *altmaterial* is the name of the material used in alternate tiles. The *tplane_init()* function must ask *material_search()* to lookup the corresponding material_t.

```
tiled_plane floor
{
   material white
   normal   0 1 0
   point    0 0 0
   dimension 2.0 3.0
   altmaterial green
}
```

Processed by plane_init

Processed by tplane_init

**The *tplane_init()* function**

```
object_t *tplane_init(
FILE     *in,
model_t  *model)
```

The function is called from the model loaders. Its missions are to:

- invoke *plane_init()* to create the *plane_t* and *object_t* structures
- malloc a *tplane_t* and set the *priv* pointer in the *plane_t* structure
- parse the parameter data
- set the *getamb* pointer in the *object_t* to point to the *tplane_amb()* function
- set the *dumper* pointer in the *object_t* to point to *tplane_dump*
- set *objtype* field in the *object_t* to "tiled plane".

The *tplane_init()* function need not override the hits function provided *plane_init()*.

**The *tplane_dump()* function**

```
static void tp_dump(
FILE     *out,
object_t *obj)
```

- invoke the *plane_dump()* function
- dump the *tplane* attributes

**The *tplane_amb()* function**

This is the function that actually gives the *tplane* its characteristic behavior.

```
void tplane_amb(
object_t *obj,
drgb_t *value)
{
   int foreground = tplane_foreground(obj);

      if (foreground)
         material_getamb(obj, value);
      else
         copy ambient reflectivity from background material

}
```

**The *tplane_foreground()* function**

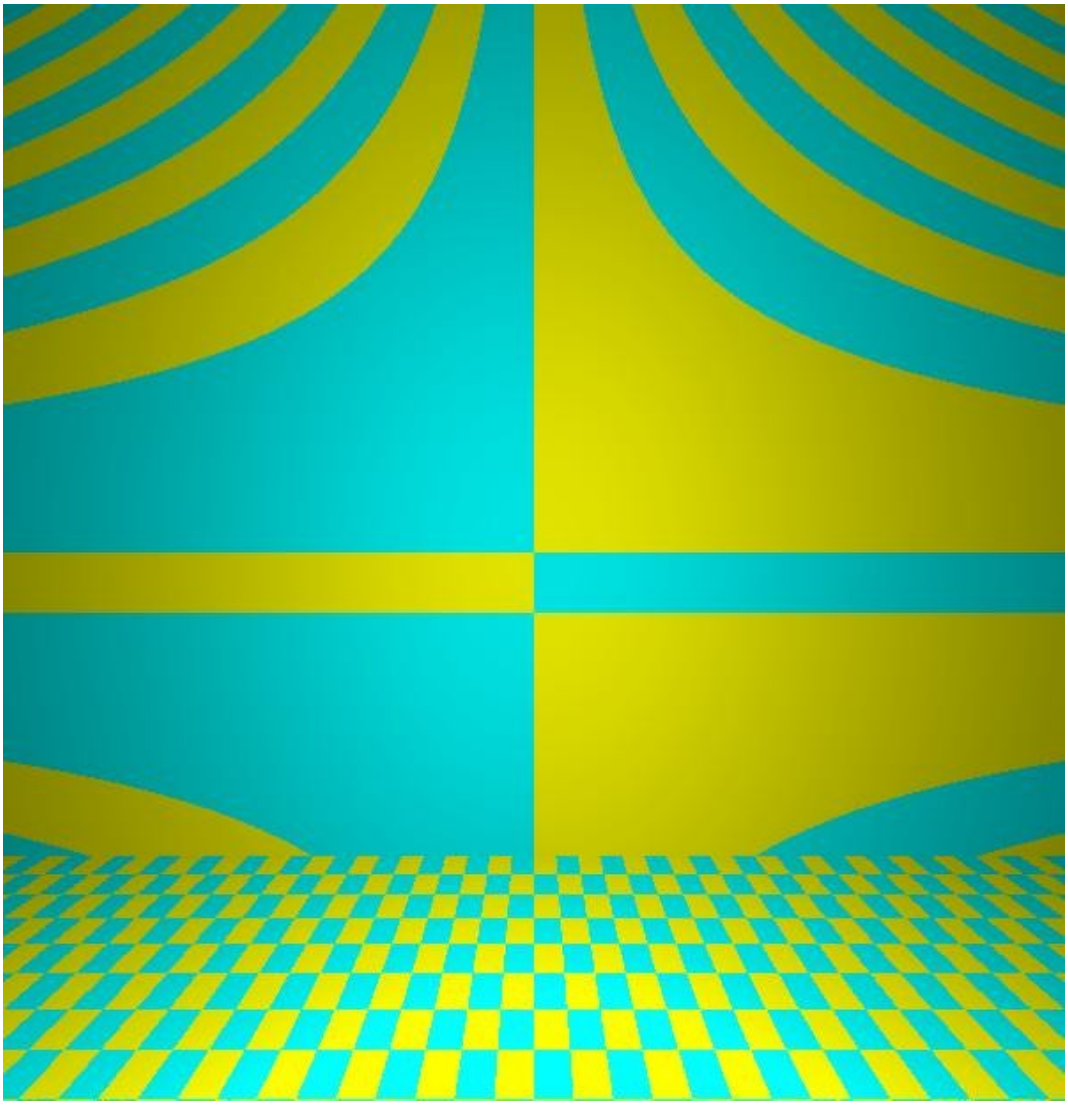
```
void tplane_foreground(
object_t *obj)
{
      Compute x_ndx = tile index of the hitpoint in the x direction
      Compute z_ndx = tile index of the hitpoint in the z direction

       if ((x_ndx + z_ndx) is an even number)
            return(1);
       else
            return(0);
}
```

A tile index is computed by adding 10000 to the x (or z) coordinate of the hitpoint and dividing by the x (or z) dimension of the tile.

**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 *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.

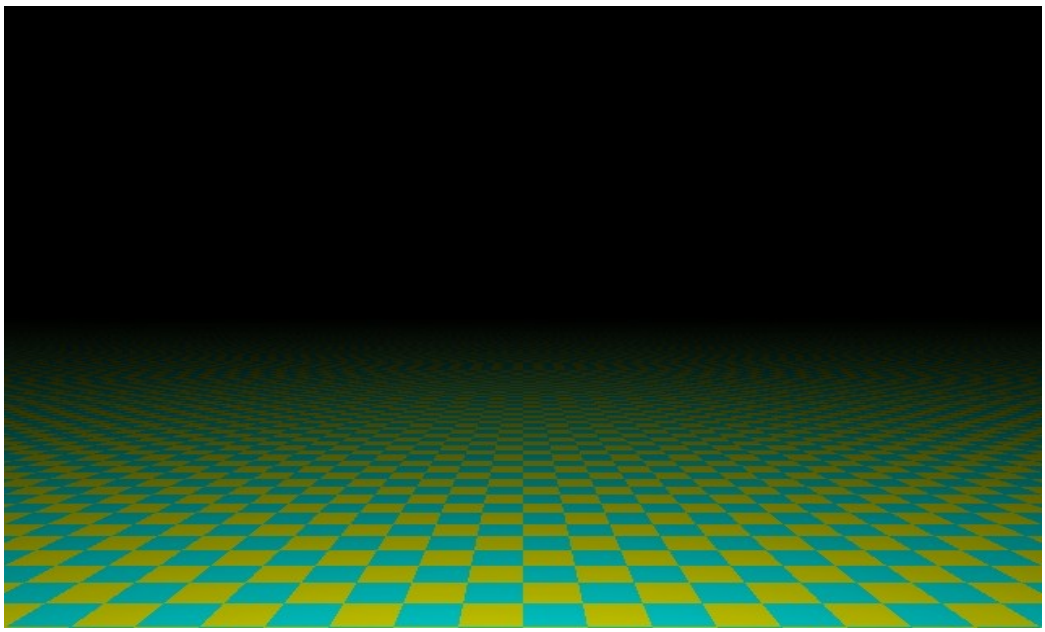
```
/**/
object_t *pplane_init(
FILE    *in,
model_t *model,
int  attrmax)
{
   pplane_t *ppln;
   plane_t  *pln;
   object_t *obj;
   link_t   *link;
   int  mask;

   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.*

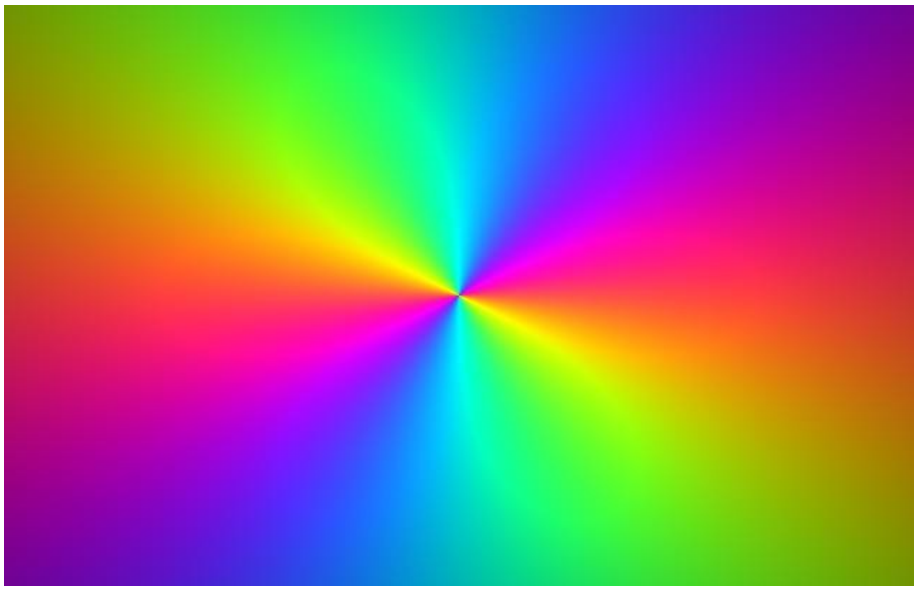


```
void pplane0_amb(
object_t *obj,
drgb_t *value)
{
   int    ix;
   int    iz;

   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
   }
}
```

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.

**Continuously modulated shading**

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



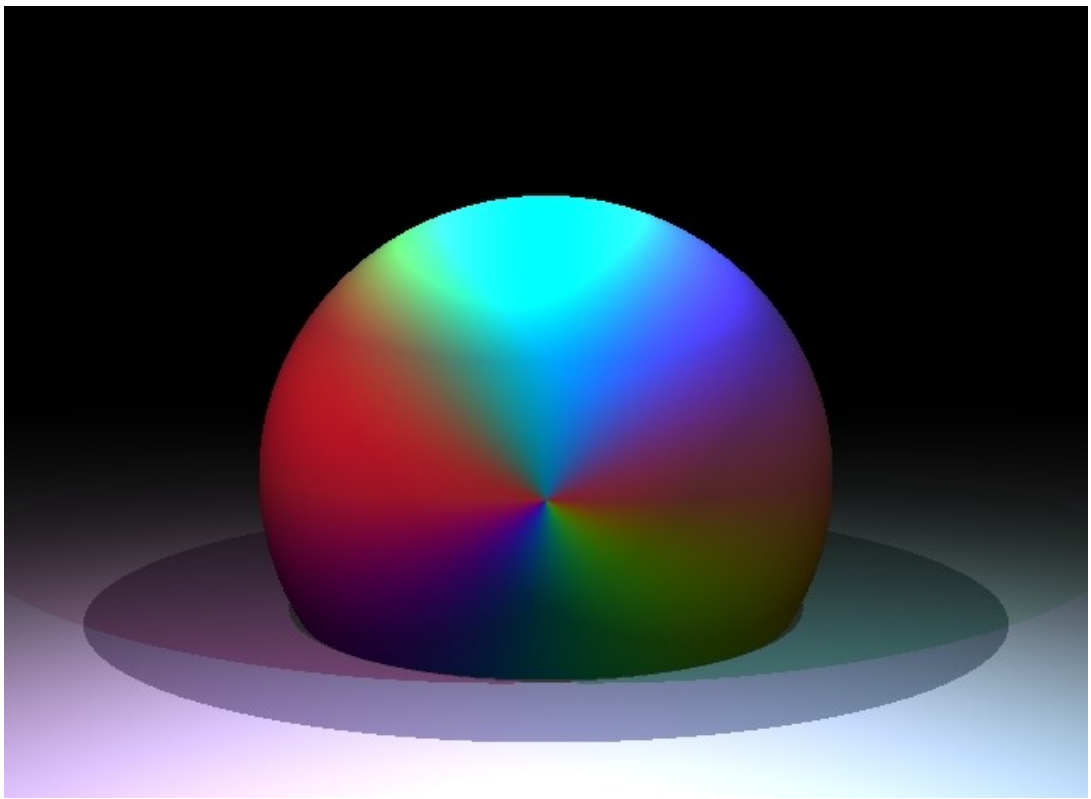
The modulation function is shown below. A vector *V* in the direction from the point defining the plane location to the hitpoint is computed first. Then the angle that the vector makes with the positive *X* axis is computed. Finally the red, green and blue components are modulated using the function $1 + \cos(w\,t + f)$ where the angular frequency w is 2 for all three colors, and phase angles f are 0, $2p/3$, and $4p/3$ 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.

$$V \times W = (v_y w_z - v_z w_y, \quad v_z w_x - v_x w_z, \quad v_x w_y - v_y w_x)$$

$$(1, 1, 1) \times (0, -1, 0) = (1, 0, -1)$$

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

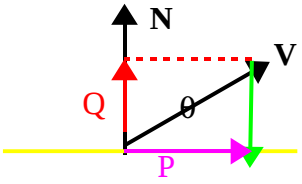
Right thumb x forefinger = middle finger.

**The *vec_cross()* function**

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(q). 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);
```

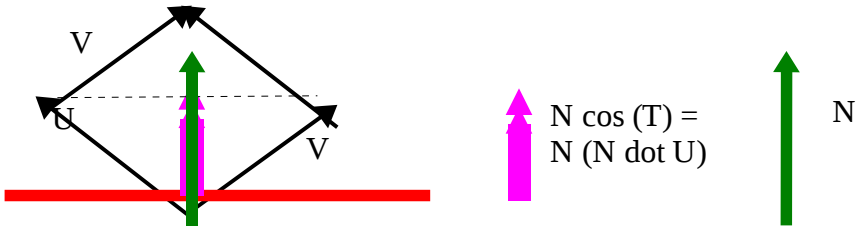
In building your new linear algebra routines it is desirable to build upon existing ones where possible but extreme levels of nesting of function calls as shown here *can complicate debugging.*

```
/**/
/* 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 */
```

**Reflection**

Basic physics says: The angle of incidence (the angle the incoming ray makes with the normal at the hitpoint) is equal to the angle of reflection

```
vec_reflect(
vec_t *unitin, /* unit vector in incoming direction of the ray  */
vec_t *unitnorm,  /* outward surface normal                      */
vec_t *unitout);/* unit vector in outgoing direction ray        */
```




Let

*U = -unitin*

*N = unitnorm*

Then

*U + V = 2 N cos(T)* where *T* is the angle between *U* and *N*

*cos(T) = U dot N*

so

*U + V = 2 N (U dot N)*

and

*V = 2 N (U dot N) - U*

**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 a ray. 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]*

$$\begin{bmatrix} 1.0 & 3.0 & 2.0 \\ 1.0 & 2.0 & -2.0 \\ -2.0 & 0.0 & 1.0 \end{bmatrix}^{T} = \begin{bmatrix} 1.0 & 1.0 & -2.0 \\ 3.0 & 2.0 & 0.0 \\ 2.0 & -2.0 & 1.0 \end{bmatrix}$$

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_transpose(
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]);
```

**Another example**

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 positive Y, Z ( X=0 )plane.

```
vec_t w = {1.0, 0.0, 1.0};
vec_t v = {1.0, 1.0, 1.0};
vec_t v3;
vec_t v4;
mat_t m1;
mat_t m2;
```

All rows of rotation matrices *must* be unit vectors!

```
vec_unit(&v, &v);
vec_unit(&w, &w);

vec_prn(stderr, "v ", &v);
vec_prn(stderr, "w ", &w);
```

If we want W to end up on the +Z access and V to lie in the X = 0 plane, then a vector perpendicular to *both W and V* must end up pointing along the +X axis. Therefore , the vector V x W must be what gets mapped to the X axis. It is important to note that the cross product in not length preserving! Therefore we must renormalize the 1st row of the matrix.

```
vec_cross(&v, &w, &m1.row[0]);
vec_unit(&m1.row[0], &m1.row[0]);
```

Since W is to be mapped to the positive z-axis we just copy it to the bottom row of the matrix.

```
vec_copy(&w, &m1.row[2]);
```

Since the missing middle row must be orthogonal to the other two rows it may be computed via the cross product. The order here is important! Z x X = Y but X x Z = -Y!

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

The matrix is now complete so we print it out.

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

We now apply the matrix to V and W

```
vec_xform(&m1, &v, &v3);
vec_xform(&m1, &w, &v4);
```

and then print the transformed vectors

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

The inverse of a rotation is its transpose.

```
mat_transpose(&m1, &m2);
```

So if we apply the inverse to v3 and v4, we should get back the original V and W

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

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

Normalized V and W vectors

```
v     0.577    0.577    0.577
w     0.707    0.000    0.707
```

The rotation matrix

```
r0     0.707    0.000   -0.707
r1    -0.000    1.000    0.000
r2     0.707    0.000    0.707
```

Transformed V and W. Note that the transformed V lies in the postive Y-Z plane (its x coordinate is 0) and the transformed W is the positive Z axis.

```
v3     0.000    0.577    0.816
v4     0.000    0.000    1.000
```

After applying the inverse transformation, the vectors are transformed back to their original values shown at the top of this page.

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