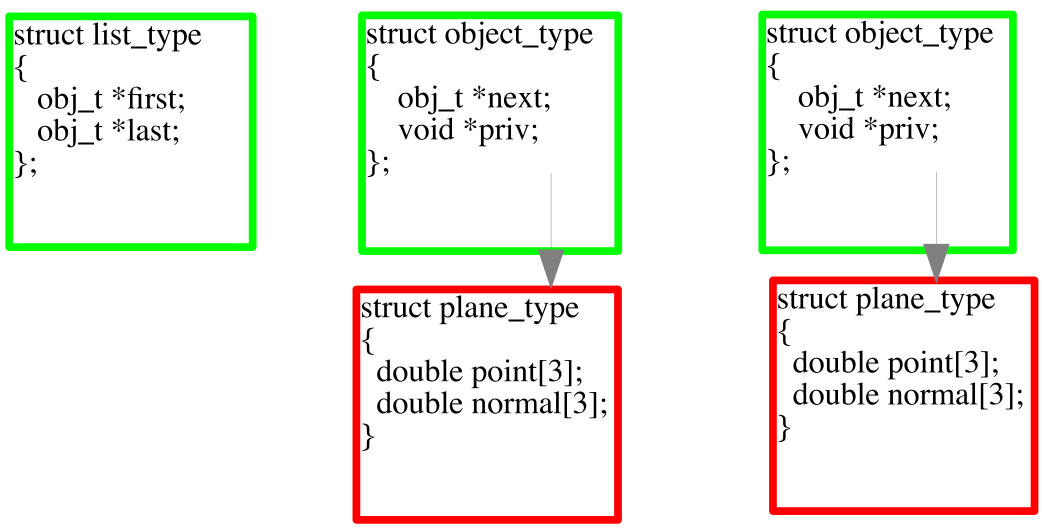
Declaration of derived object types

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 *obj_t* is used to connect the base class instance to the derived class instance. This connection is automatic and invisible in a true OO language but is *manual* and *visible* in C.

```
/* Infinite plane */
typedef struct plane_type {
   double  point[3];  /* A point on the plane    */
   double  normal[3]; /* A normal[2] vector to the plane */
}  plane_t;

/* Sphere */
typedef struct sphere_type {
   double  center[3];
   double  radius;
}  sphere_t;

/* Point light source */
typedef struct light_type {
   double  location[3];
}  light_t;
```



2 The term *normal vector* is used to refer to a vector perpendicular to the surface of an object.

Loading the model description

The raytracer must be able to read model descriptions of the format shown below. This format is designed for *easy* digestion. All numeric values are readable with *scanf()*. After reading the required values from each line *fgets(buff, 256, stdin);* should be called to consume the descriptive text.

Model definitions will begin with the projection data as previously described.

Following the view point will be an arbitrary and unknown number of object descriptions. Each object description will begin with an object type (10 = light and 14 = plane, 15 = sphere...) other new types will follow. The remainder of the parameters will be dependent upon the type of object being loaded. *Therefore you must create a separate object loader (and dumper) for each object type.* Here you will need routines *plane_init(), plane_dump(), sphere_init(), sphere_dump()*.

```
8 6             world x and y dims
0 0   3         viewpoint (x, y, z)

14              plane
5 5 2           r g b ambient
0 0 0           r g b diffuse
0 0 0           r g b specular

 1 0   1        normal
-4 -1 0         point

14              plane
5 2 5           r g b ambient
0 0 0           r g b diffuse
0 0 0           r g b specular

-1 0   1        normal
 4 -1 0         point

13              sphere
2 5 2           r g b ambient
0 0 0           r g b diffuse
0 0 0           r g b specular

0 1 -3          center
1.5             radius
```

The *model_init* function

The *model_init()* function should consist of a single loop that reads an *object type code* from the standard input and then invokes an object type specific function to read in the data describing the *sphere, plane, or light.* This function should abort the program by calling exit if errors are encountered in the input.

Associating numeric identifiers with symbolic names

When numeric identifiers are used in a C program they should always be equated to a symbolic name and *only the symbolic name* should be used in executable code.

```
/* Object types (Values subject to change)*/

#define FIRST_TYPE   10
#define LIGHT        10
#define SPOTLIGHT    11
#define PROJECTOR    12
#define SPHERE       13
#define PLANE        14
```

```
int model_init(FILE *in,  model_t *model){
   char  buf[256];       /* comment me!!! and get rid of 256 */
   int   objtype;        /* me two         */
   int   rc    = 0;      /* me three       */
   obj_t *new  = NULL;   /* and me!        */

   <SNIP>
}
```

This is not the best way to accomplish our goal, but will suffice for now.

Object creation and initialization

In true object oriented languages instances of derived classes are "automagically" bound to an instance of the base class at the time the object is created. In C it will be necessary to manually invoke a constructor for the *derived* class. The constructor for the *sphere_t* is:

```
obj_t *sphere_init(FILE *in, int objtype);
```

Derived class constructors must:

1. Explicitly invoke the constructor for the *obj_t* base class.
2. *malloc()* an instance of the structure describing the derived class
3. Fill in the attributes of the instance of the derived class.
4. Fill in required function pointers in the *obj_t* structure
5. Link the *obj_t* structure to the derived class structure using the *obj->priv* pointer in the *obj_t*

The *sphere_init(), plane_init()* functions are responsible for creating the required structures and reading in attribute data from the model definition file.

```
obj_t *sphere_init(FILE *in, int objtype){
   obj_t      *obj = NULL;
   sphere_t   *new = NULL;
   int        pcount = 0;
```

All object-specific loaders begin by creating the generic object type.

```
   obj = object_init(in, objtype);
```

allocate a sphere_t structure on the heap
link it to the obj_t structure

read the location of the center and the radius into the sphere_t structure

```
}
```

Initialization of the *obj_t*

The *obj_t* constructor *object_init()* is responsible for allocating an instance of the *obj_t* and initializing it. The reflective properties of visible objects in the scene are carried in the *material_t* structure that is embedded within the *obj_t.*

The *material_t* structures carry the *red, green,* and *blue* reflectivity of the object to *ambient, diffuse and specular light.* Larger values make the object brighter. An ambient reflectivity of (5, 0, 0) makes the object appear as *red,* while (5, 5, 0) is yellow, and (5, 5, 5) white. For the first milestone only the *ambient* reflectivity will be used but we will go ahead and read in all components.

// Note: You should remove magic numbers from example code.

```
typedef struct material_type{
   double  ambient [3];    /* Reflectivity for materials  */
   double  diffuse [3];
   double  specular[3];
}  material_t;


obj_t *object_init(FILE *in, int objtype){
   obj_t *obj = NULL;

   malloc a structure of type obj_t
   fill in objtype and objid fields

   if (the object is not a light){
      call material_load to read ambient, diffuse, specular reflectivity
   }
   return(obj);
}
```

Object dumpers

For each object type you must also provide an object dumper that provides a reasonably formatted report describing the input data. The following example is acceptable:

```
Dumping object of type Plane
Material data -
Ambient  -    5.000   5.000   2.000
Diffuse  -    0.000   0.000   0.000
Specular -    0.000   0.000   0.000

Plane data
normal -    1.000   0.000   1.000
point  -   -4.000  -1.000   0.000
```

The recommended form of a object dumper is shown below. All should reply on a common *material_dump()* function rather than each embedding its own material dumper.

```
int plane_dump(FILE *out, obj_t *obj){
   plane_t *plane = NULL;

   material_dump(out, &obj->material);

   print plane specific data
}
```

Ray tracer designed (continued)

Now we are finally ready to build an image. This will be a very crude image because it will support ambient lighting only. Nevertheless this is a significant milestone because the 3-D geometry problem must be addressed.

Overview of the *make_image()* function

The *make_image()* function should live in a separate module named *image.c*

```
void make_image(model_t *model){
    unsigned char *pixmap = NULL;

    compute size of output image and malloc() pixmap.

    for y = 0 to  window size in pixels {
        for x = 0 to window size in pixels {
            make_pixel(model, x, y, pixmap_location);
        }
    }
    write .ppm P6 header
    write pixmap
}
```

The *make_pixel* function

This function is responsible for driving the construction of the *(r, g, b)* components of a single pixel. Within the ray tracing process pixel colors are represented as

1. double precision values in the range *[0.0, 1.0]* where
2. 0.0 represents black and 1.0 the brightest level of the corresponding color.

However, depending upon input values its possible for the raytracing algorithm *to compute intensities that exceed 1.0.* When this happens this module must *clamp* them back to the allowable range [0.0, 1.0].

```
/*
 * make_pixel -      ??
 * Parameters:       model – I am something?
 *                   x, y – pixel x and y coordinate
 *                   pixval – to (r,g,b) in pixmap
 */
Be careful using alloca(), use it where appropriate (note, it is
really not needed anymore)!

void make_pixel(model_t *model, int x,int y,
                unsigned char *pixval){
   double *world = alloca(3 * sizeof(double));
   double *intensity = alloca(3 * sizeof(double));

   map_pix_to_world(x, y, world);
```

initialize intensity to (0.0, 0.0, 0.0)

compute unit vector dir: in the direction from the *view_point* to *world;*

```
   ray_trace(model, model->proj->view_point, dir, intensity,
             0.0, NULL);
```

clamp each element of intensity to the range [0.0, 1.0]

*set (r, g, b) components of vector pointed to by pixval to 255 * corresponding intensity*

```
}
```

The *ray_trace* function

The *ray_trace* function is responsible for tracing a single ray. It should reside in *ray.c*

```
/*
 * This function traces a single ray and returns the composite
 * intensity of the light it encounters   It is recursive and
 * so the start of the ray cannot be assumed to be the viewpt
 * Recursion will not be involved until we take on specular light
 *
 *
 * Parameters:      model – pointer to the model container
 *                  base[] - location of viewer, or previous hit
 *                  dir[] - unit vector in the direction of object
 *                  intensity[] - intensity return location
 *                  total_dist – distance ray has traveled so far
 *                  last_hit – obj that reflected this ray or NULL
 */
void ray_trace(model_t *model, double base[3], double dir[3],
               double intensity[3], double total_dist,
               obj_t  *last_hit) {
```

The ray trace function should rely upon *find_closest_object* to identify the nearest object that is hit by the ray. If none of the objects in the scene is hit, NULL is returned. The distance from the base of the ray to the nearest hitpoint is returned in *mindist.*

```
   closest = find_closest_obj(model->scene, base, dir, NULL,
                              &mindist);

   if (closest == NULL)
      return;
```

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

```
}
```