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 program, deductions will be made for:

1 - Functions that are too long (greater than 30 lines)
2 - Nesting of code greater than 2 deep
3 - Lines that are too long (greater than 72 characters)

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

```
int main(
int argc,
char **argv)
{
   model_t *model   = (model_t *)malloc(sizeof(model_t));
   int      rc;

   fetch ppm filename and number columns from command line
   open output ppm file as outputFP

   model->proj = projection_init(stdin, columns);
   projection_dump(model->proj);

   model->lights  = list_init();
   model->scene   = list_init();

   rc = model_init(stdin, model);
   model_dump(stderr, model);

   if (rc == 0)
      make_image(outputFP, 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 inheritance and polymorphism found in true Object Oriented languages.

The *obj_t* structure serves as the generic "base class" from which objects in the *light* or *scene object* categories are derived. In turn the esoteric objects such as *planes* and *spheres* are derived from the scene object category. As such, the based class carries only the attributes that are common to the derived objects.

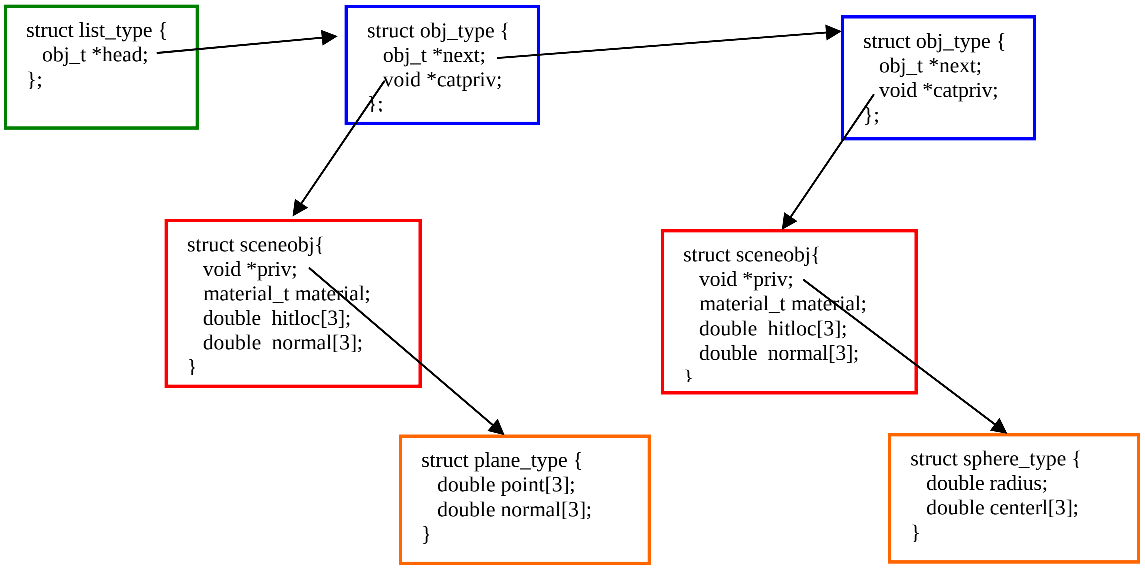
Polymorphic behavior is achieved by the use of *function pointers* embedded in fields related to the specific object type. These can be initialized to point to functions that provide a *default* behavior but may be overridden as needed when an esoteric object such as a *tiled plane* must substitute its own method. We will see examples of this later (note the function pointers in the following description of the object category structures).

```
typedef struct obj_type
{
   struct  obj_type  *next;   /* Next object in list                   */
   int     objid;             /* Numeric serial # for debug            */
   char    *objclass;         /* Object type name (e.g. "plane")       */
   int     objcat;            /* Category code (e.g. 1 -> sceneobject) */

   void    *catpriv;          /* Private category-dependent data       */

} obj_t;
```

Each object will have a pointer that point to data that is specific to a particular instance of an object. The *catpriv* pointer will point to data that is specific to the category type of the object (i.e. light versus scene object). As we will see, the *priv* pointer in the category structure will point to data that is specific to a particular type of light or scene object (e.g. sphere, plane, projector, etc.). The linked list of objects now looks like:



Category Specific Data

Our virtual world will contain two categories of objects: scene objects (e.g. planes, spheres, etc) and lights. These two categories have very distinct needs in terms of the type of data that is stored for the object. For example, for a scene object we need information about the color of the object under various lighting conditions. The *classpriv* pointer of the base class *obj_t* is used to connect the bases class to the derived category class.

The data structure for the category specific information for a *scene object* is (fields shown in *blue* are required for the first version of the ray tracer):

```
typedef struct sceneobj {
      /* Pointer to type dependent data */
         void    *priv;        /* Private type-dependent data            */

      /* Reflectivity for reflective objects */
         material_t material;

      /* Optional plugins for retrieval of reflectivity */
      /* useful for the ever-popular tiled floor        */
         void    (*getamb)(struct obj_type *, double *);
         void    (*getdiff)(struct obj_type *, double *);
         void    (*getspec)(struct obj_type *, double *);

      /* Hits function.            */
         double  (*hits)(double *base, double *dir, struct obj_type *);

      /* "Hit" information for object */
         double  hitloc[3];    /* Last hit point                         */
         double  normal[3];    /* Normal at hit point                    */
         double  distance;     /* Distance from "base" to last hit point */
} sceneobj_t;
```

A light category object requires a different set of data:

```
typedef struct lightcat {
      /* Pointer to type dependent data */
         void    *priv;         /* Private type-dependent data   */

         double  emissivity[3]; /* Light color                   */

      /* Plugin to retrieve light color                          */
         void    (*getemiss)(struct obj_type *, double *);
} lightcat_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 classe *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().* Note that comments start with a "#" and continue through the end-of-line.

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 name ("light", "plane", "sphere", etc … 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 category loader and 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)

# Infinite Plane 1
plane
5 5 2           #r g b ambient
0 0 0           #r g b diffuse
0 0 0           #r g b specular
# Plane specific data
 1 0  1         #normal
-4 -1 0         #point

# Infinite Plane 2
plane
5 2 5           #r g b ambient
0 0 0           #r g b diffuse
0 0 0           #r g b specular
# Plane specific data
-1 0  1         #normal
 4 -1 0         #point

# Sphere 1
sphere
2 5 2           #r g b ambient
0 0 0           #r g b diffuse
0 0 0           #r g b specular
# Sphere specific data
0 1 -3          #center
1.5             #radius
```

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. The category type provides a good opportunity to make use of this:

```
/* Category types */

#define LIGHTCAT      0
#define SCENEOBJ      1
```

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 a category specific and an object type specific function to read in the data describing the category and the object *sphere, plane, or light.* This function should abort the program by calling exit if errors are encountered in the input.

The following is a combination of *pseudo-code* and "C" statements that outline the form of the "model_init()" function:

```
/**/
int model_init(FILE    *in, model_t *model){
   obj_t *new;

   /* Load the objects in the scene */
   while there are more objects in input file
      read class name into objclass
      if objclass is a "sphere"
              new = sphere_init(in, objclass)
      else if objclass is a "plane"
              new = plane_init(in, objclass)
      else
              print "Unknown object type 'objclass'" to stderr
              new=NULL;
       }

       if (new == NULL)
       {
          fprintf(stderr, "failed to load type %s \n",
                            objclass);
          model_dump(stderr, model);
          exit(2);
       }
       else
       {
          list_add(model->scene, new);
       }
     }
```
It will be shown later that it is possible to replace this rather messy construction with a table of function pointers.

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.   For example, the constructor for the *sphere_t* is:

```
obj_t *sphere_init(FILE *in, char *objclass);
```

Derived class constructors must:

1.Explicitly invoke the contstructor for the *sceneobj_t* category 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 *sceneobj_t* structure
5.Link the *sceneobj_t* structure to the derived class structure using the scene*obj->priv* pointer in the *sceneobj_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,
char    *objclass)
{
   obj_t      *obj;
   sphere_t   *new;
   int        pcount = 0;
```

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

```
   obj = sceneobj_init(in, objclass);

   new = malloc'd space for sphere_t structure
   set obj->catpriv->priv to new

   read the location of the center and the radius into the sphere_t pointed to by new

   return(obj)

}
```

Category creation and initialization for scene objects

Class constructors (e.g. sphere_init(), plane_init()) will use a category constructor to create and initialize the category specific data. For example, the constructor for a scene object *sceneobj_t* is:

```
     obj_t *sceneobj_init(FILE *in, char *objclass);
```

The scene object category constructor must:

1. Explicitly invoke the contstructor for the *obj_t* generic class.
2. *malloc()* an instance of the structure describing the category class (e.g. *sceneobj_t*),
3. invoke *material_load()* to process the color data for the object,
4. Link the *sceneobj_t* structure to the *obj_t* structure using the scene*obj->priv* pointer in the *obj_t*

Note that all scene objects share common fields such color information and a place to save hit information.

Note that incorporated into the *sceneobj_t* object is an instance of the *material* structure. 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.

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

The pseudo-code for creating a sceneobj_t object is:

```
obj_t *sceneobj_init(FILE *in, char *objclass) {
  obj = object_init(in, objclass, SCENEOBJ)
  set obj->catpriv to malloc'd space for sceneobj_t object
  call material_load to read ambient, diffuse and specular reflectivity
  return(obj)
}
```

Initialization of the *obj_t*

The *obj_t* constructor *object_init()* is responsible for allocating an instance of the *obj_t* and initializing it. These are the fields that are common to all objects. Note that object_init() calls upon an appropriate category constructor (see next page).

```
obj_t *object_init(
FILE   *in,
char   *objclass
int    *objcat)
{
   obj_t *obj;

   set obj to malloc'd space for an object of  type obj_t
  fill in objclass, objid and objcat  fields

   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. An object dumper should rely upon a common category dumper (e.g. sceneobj_dump()) to print category specific data. In turn the sceneojb_dump() function should rely on a common *material_dump()* function rather than each embedding its own material dumper.

```
void plane_dump(
FILE *out,
obj_t *obj)
{
   plane_t *plane;

   sceneobj_dump(out, obj);

   plane = ((plane_t *)obj->catpriv)->priv;

   print plane specfic data
}


void sceneobj_dump(
FILE *out,
obj_t *obj)
{
   sceneobj_t *sceneobj;

   sceneobj = (sceneobj_t *)obj->catpriv;

   material_dump(out, sceneobj);

}
```