Structured data types

A more elegant way to deal with the *rgb* image is to employ *structured data types*. A Java *class* is a generalization of a C *struct*. A *structure* is a aggregation of basic and structured data elements possibly including pointers, but unlike a class it contains no embedded *methods* (functions).

A C structure is declared as follows:

```
struct pix_type{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};
```

As with Java it is important to be aware that *struct pix_type is the name of a user defined structure type.* It is *not* the name of a variable.

To declare an *instance* (*variable*) of type *struct pix_type* use:

```
struct pix_type pixel;
```

`struct pix_type` is the name of the *type*
`pixel` is the name of a *variable* of type *struct pix_type*

To set or reference components of the *pixel* use:

```
pixel.r = 250;  // make Mr. Pixel yellow
pixel.g = 250;
pixel.b = 0;
```

Pointers to structures:

To declare a *pointer* to a *struct pix_type* use:

```
struct pix_type *pixptr;
```

Before using the pointer we must always make it point to something:

```
pixptr = (struct pix_type *)malloc(sizeof(struct pix_type));
```

To set or reference components of the *pix_type* to which it points use:

```
(*pixptr).r = 250;  // make Mr. *pixptr magenta
(*pixptr).g = 0;
(*pixptr).b = 250;
```

Warning: the C compiler is very picky about the location of the parentheses here.

Perhaps because of the painful nature of the above syntax, an alternative "short hand" notation has evolved for accessing elements of structures through a pointer:

```
pixptr->r = 0;      // make Mr. pixptr-> cyan
pixptr->g = 250;
pixptr->b = 250;
```

This shorthand form is almost universally used.

Revisiting the color image

Space for a color image in binary *rgb* format can be allocated by:

```
struct pix_type *pixptr;
      :
pixptr = (struct pixptr *)malloc(sizeof(struct pix_type)  *
                                    numrows * numcols);
```

To read the image data

```
pixcount = fread(pixptr, sizeof(struct pix_type), numrows *
                 numcols, stdin);

if (pixcount != numrows * numcols){
   fprintf(stderr, "pix count err - wanted %d got %d \n",
               numrows * numcols, pixcount);
   exit(1);
}
```

To access a specific pixel

```
red   = *(pixptr + row * numcols + col).r;
green = *(pixptr + row * numcols + col).b;
blue  = *(pixptr + row * numcols + col).g;

or

red   = (pixptr + row * numcols + col)->r;
green = (pixptr + row * numcols + col)->g;
blue  = (pixptr + row * numcols + col)->b;

or even (but not in class assignments)

red   = pixptr[row * numcols + col].r;
```

Structures and Arrays

An element of a structure may be an *array:*

```
struct img_type{
   int numrows;
   int numcols;
   unsigned char pixval[1024][768];
};

struct img_type image;
```

Elements of the array are accessed in the usual way:

```
image.pixval[5][10] = 125;
```

It is also possible to create an array of structures

```
struct pixel_type{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pixel_type pixmap[100];
```

To access an individual element of the array place the subscript next to the name it indexes

```
pixmap[15].r = 250;
```

It is also possible to create a array of structures containing arrays

```
struct img_type images[20];
```

Elements of the array are accessed in the usual way:

```
image[4].pixval[5][10] = 125;
```

Structures containing structures

It is common for structures to contain elements which are themselves structures or arrays of structures. In these cases *the structure definitions should appear in "inside-out" order.*

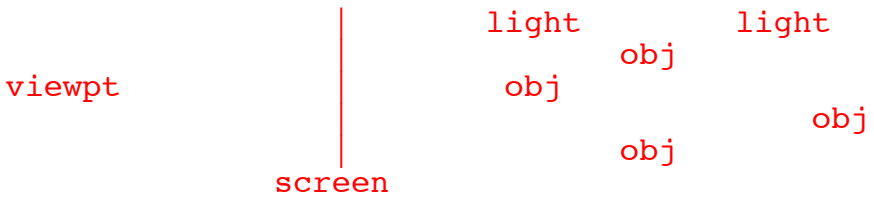
This is done to comply with the usual rule of not referencing a name before it is defined.

```
struct pixel_type{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct img_type{
   int numrows;
   int numcols;
   struct pixel_type pixdata[1024][768];
};

struct img_type image;

image.pixdata[4][14].r = 222;
```

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 three major elements involved in the process:

1 - *The viewpt* 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
3 - *The scene* The scene consists of objects and light sources



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 window. These are two dimensional (x, y) integer numbers For example, if a 400 cols *x* 300 rows image is being created the window *x* coordinates range from 0 to 399 and the window *y* coordinates range from 0 to 299.

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 center 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 *viewpt* has a negative *z* coordinate (this is subject to change)
all objects have **positive** *z* coordinates.

The projection data structure

The *typedef* facility can be used to create a new identifier for a user defined type. The following example creates a new type name, *proj_t,* which is 100% equivalent to *struct projection_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:

```
typedef struct projection_type {
   int    win_size_pixel[2];  /* Projection screen size in pix */
   double win_size_world[2];  /* Screen size in world coords   */
   double view_point[3];      /* Viewpt Loc in world coords    */
}  proj_t;
```

To map a pixel coordinate to a world coordinate a function such as the following could be used:

```
/*
 * map_pix_to_world – map a pixel coordinate to a world coord.
 * Parameters:      proj – pointer to a project definition
 *                  x,y  - x and y pixel coordinates
 *                  world – pointer to three doubles
 */
void map_pix_to_world(proj_t *proj, int x,int y, double *world){
   *(world + 0) = (double)x /(proj->win_size_pixel[0] - 1) *
                  proj->win_size_world[0];
   *(world + 0) -=  proj->win_size_world[0] / 2.0;

   *(world + 1) = ???;
   *(world + 2) = 0.0;
}
```

Example:

Suppose win_size_pixel[0] = 800 pixels
Suppose win_size_world[0] = 20 units

Then:

the world x coordinate of the pixel with x pixel coordinate 400 is approximately ___,
the world x coordinate of the pixel with x pixel coordinate 0 is ___.
the world x coordinate of the pixel with x pixel coordinate 799 is ___.

The raytracing algorithm

The complete raytracing algorithm is summarized below:

Phase 1: Initialization

acquire window pixel dimensions from command line
read world coordinate dimensions of the window from the stdin
read world coordinates of the view point from stdin
print projection data to the stderr

load object and light descriptions from the stdin
dump object and light descriptions to the stderrr

Phase 2: The raytracing procedure for building the pixmap

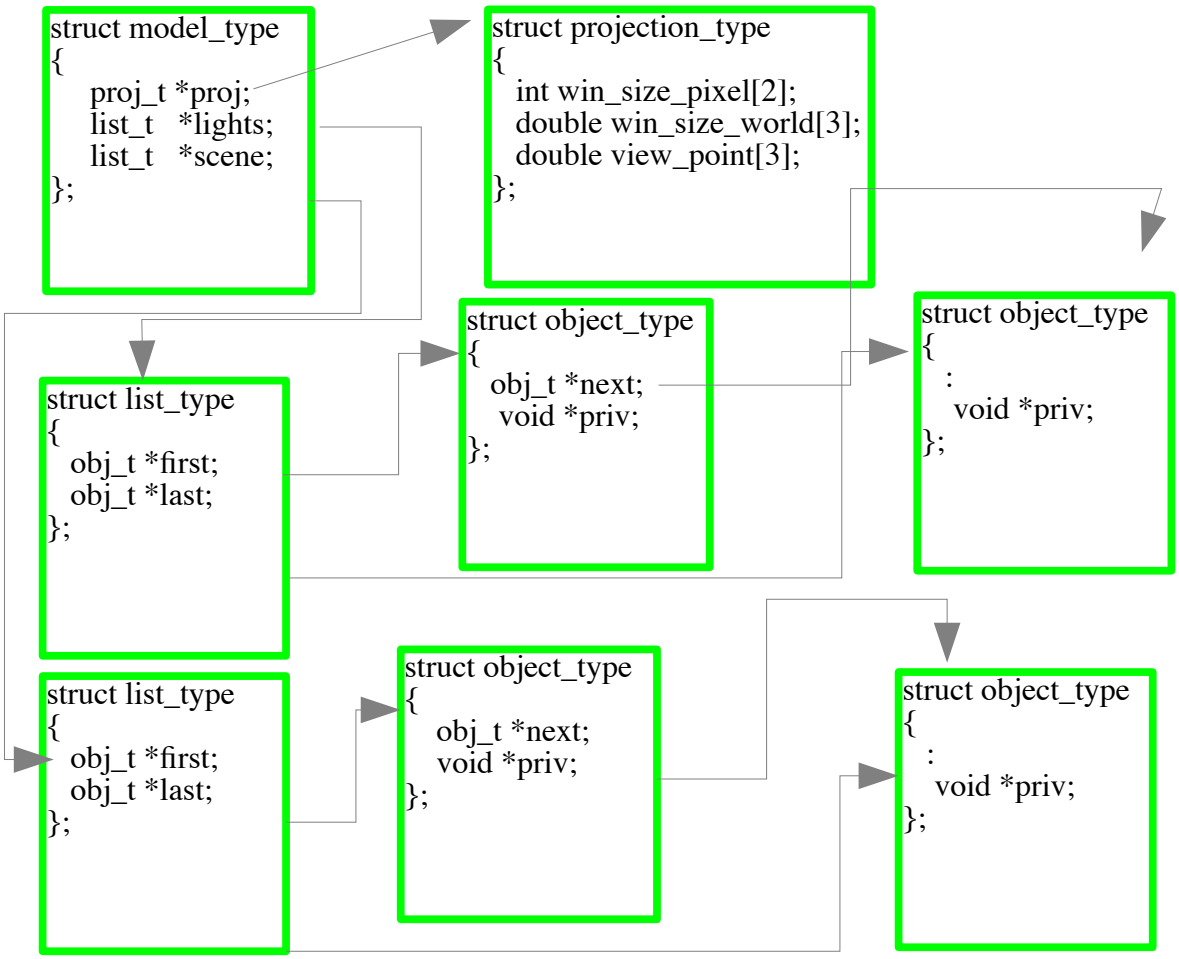
```
for each pixel in the window {
   compute the world coordinates of the pixel
   compute  the direction in 3-d space of a ray from the viewpt through the pixel
   compute the color of the pixel based upon the illumination of the object(s) hit by the ray
}
```

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

write .ppm header to stdout
write the image to stdout

Data structures - the big picture

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 object 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

Because of these restrictions a singly linked list suffices nicely, and a list may be associated with a list header structure of the type shown below.

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

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

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

```
list_t *list_init(void) {

}
```

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

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

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

}
```

The *model* data structure

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

```
typedef struct model_type {
   proj_t  *proj;
   list_t  *lights;
   list_t  *scene;
}  model_t;
```

The main function

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

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

1 - Functions that are too long (greater than 30 - ish lines)
2 - Nesting of code greater than 2 deep
3 - Lines that are too long (greater than 80 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;

   model->proj = projection_init(argc, argv, stdin);
   projection_dump(stderr, model->proj);

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

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

   if (rc == 0)
      make_image(model);

   return(EXIT_SUCCESS);
}
```

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 the esoteric objects such as *planes* and *spheres* are derived. As such, it carries only the attributes that are common to the derived objects.

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

```
typedef struct obj_type {
   struct  obj_type  *next;   /* Next object in list        */
   int     objid;             /* Numeric serial # for debug */
   int     objtype;           /* Type code (14 -> Plane )   */

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

   /* 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 *);

   /* Reflectivity for reflective objects */
   material_t material;

   /* These fields used only in illuminating objects (lights) */
   void    (*getemiss)(struct obj_type *, double *);
   double  emissivity[3]; /* For lights                  */
   void    *priv;         /* Private type-dependent data */

   double  hitloc[3];     /* Last hit point              */
   double  normal[3];     /* Normal at hit point         */
} obj_t;
```