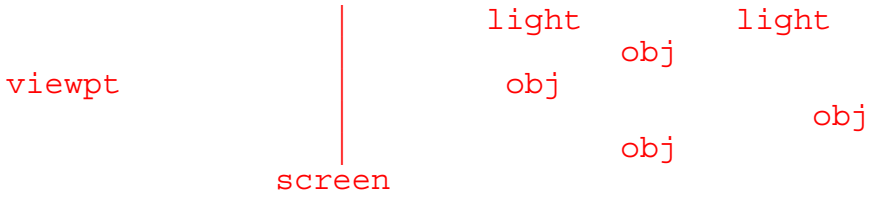
Ray tracing introduction

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 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

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 *viewpoint* has a positive *z* coordinate
- all objects have negative *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
{
   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 coordintate a function such as the following could be used:

```
void map_pix_to_world(
proj_t *proj,     /* projection definition */
int    x,         /* x and y pixel         */
int    y,         /*   coordinates         */
double *world)    /* pointer to 3 doubles  */
{
   *(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 0,
the world x coordinate of the pixel with x pixel coordinate 0 is -10.
the world x coordinate of the pixel with x pixel coordinate 799 is 10.0

The raytracing algorithm

The complete raytracing algorithm is summarized below:

Phase 1: Initialization

acquire output ppm filename and window column pixel dimension from command line
read world coordinate dimensions of the window from the stdin
read world coordinates of the view point from stdin
*compute window row pixel dimension as (world_height/world_width)*window_columns*
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

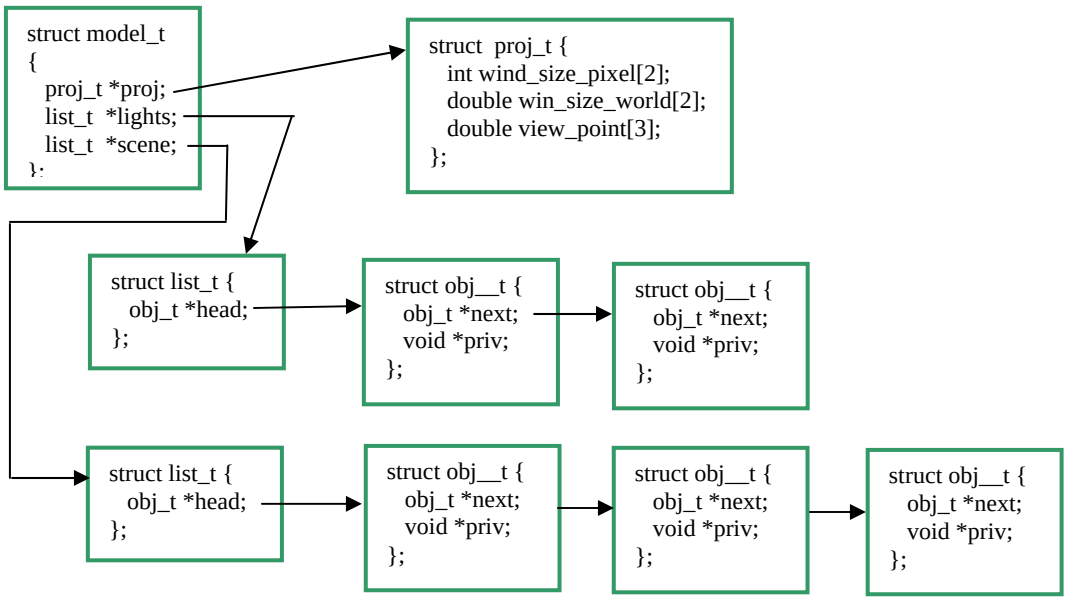
for each pixel in the window
{
    compute the world coordinates of the pixel
    compute the direction in 3-d space of a ray from the viewpt through the pixel
    compute the color of the pixel based upon the illumination of the object(s) hit by the ray
}

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

write .ppm header to output ppm file
write the image to output ppm file

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 beginning 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 */
} 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* element 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 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;

   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(0);
}
```