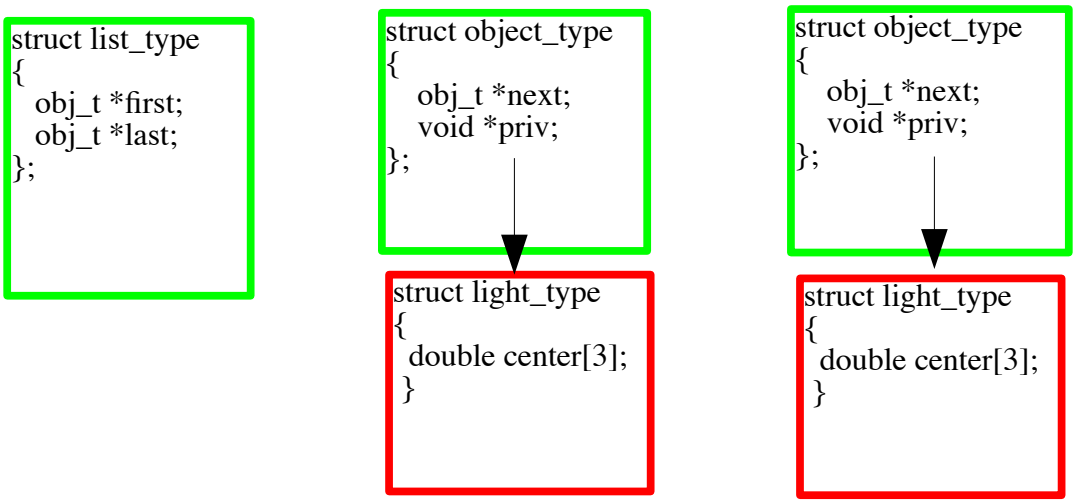
Diffuse illumination

Diffuse illumination is associated with specific light sources but is reflected uniformly in all directions. A white sheet of paper has a high degree of diffuse reflectivity. It reflects light but it also scatters it so that you cannot see the reflection of other objects when looking at the paper.

To model diffuse reflectivity requires the presence of one or more light sources. The first type of light source that we will consider is the point light source. This idealized light source emits light uniformly in all directions. It may be located on either side of the virtual screen but is, itself, *not visible.*

The lights will be managed in a way analogous to the visible objects, but will reside on a *separate list.*

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



Properties of point light sources

The location of the light is carried in the *light_t* structure.

```
typedef struct light_type{
   double  center[3];
}  light_t;
```

The brightness of the light is carried in the *emissivity vector* of the *obj_t*. By using *r, g, b* components of different magnitudes it is possible to create lights of any color. The color of a pixel that is illuminated by a light is the component-wise product of the *emissivity* and the *diffuse reflectivity* of the visible object. Therefore, in a raytraced scene consisting of a single plane and a single light it is *not possible* to determine from the image whether a *red* material is being illuminated by a *white* light or a *white* material is being illuminated by a *red* light! Furthermore, if a pure *red* light illuminates a pure *green* sphere, the result is purely invisible! This effect is counter to our experience because in the "real world" monochromatic lights and objects with pure monochromatic reflectivity are not commonly encountered.

It could be argued that the *emissivity* should properly reside in either the *material_t structure* or the *light_t* structure.

```
typedef struct obj_type{
   struct  obj_type  *next;      /* Next object in list        */
   int     objid;                /* Numeric serial # for debug */

/* Reflectivity for reflective objects */

   material_t material;

/* These fields used only in illuminating objects (lights) */

   double  emissivity[3]; /* For lights                    */
```

Specifying a light in a model description file

```
10             code for light
4 4  4         emissivity (a white light)
8 6 -2         location
```

The diffuse illumination procedure

Diffuse illumination should not be computed in the with *ray_trace()* function. The procedure is sufficiently complicated that it should be implemented in a separate module that we will call *illuminate.c* It will contain a function called *diffuse_illumination()* that will be called by the *ray_trace() function.*

```
/* ray_trace - 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.
 * Parameters: model -   list of objects
 *             base -    location of viewer OR previous hit
 *             dir  -    unit vector in direction of the object
 *             intensity – return location for intensity
 *             total_dist – total distance so far...
 */

void  ray_trace(list_t *model, double *base, double *dir,
                double *pix, double total_dist)
{
```

Test each non-light object to see if it is hit by the ray and
set "closest" to point to the nearest such object hit.
If closest is NULL
return;
Add the distance from base of the ray to the hit point to total_dist
Add the ambient reflectivity of the object to the intensity vector
Add the diffuse reflectivity of the object at the hitpoint to the intensity vector
Scale the intensity vector by 1 / total_dist.

```
}
```

Important notes:

The ambient reflectivity is a property of the *object as a whole.*
The diffuse reflectivity is a function of the *specific hit point.*
The contribution of *all lights* that are visible from the hit point must be summed.

Computing the diffuse reflectivity of an object

A function having the following prototype provides a useful front end for the diffuse computation. It should be called from *ray_trace()* at as shown in the pseudo code on the previous page.

```
 /*
  * diffuse_illumination – Comment me
  * Parameters: model - pointer to the model structure
  *             hitobj – object that was hit by the ray
  *             intensity – where to add the intensity
  */
void diffuse_illumination(model_t  *model, obj_t *hitobj,
                          double *intensity){
     for all lights on the light list {
          process_light()
     }
}
```

Determining if a light illuminates the hit point

We use idealized point light sources which are themselves invisible, but do emit illumination. Thus *lights themselves will not be visible in the scene* but the effect of the light will appear.

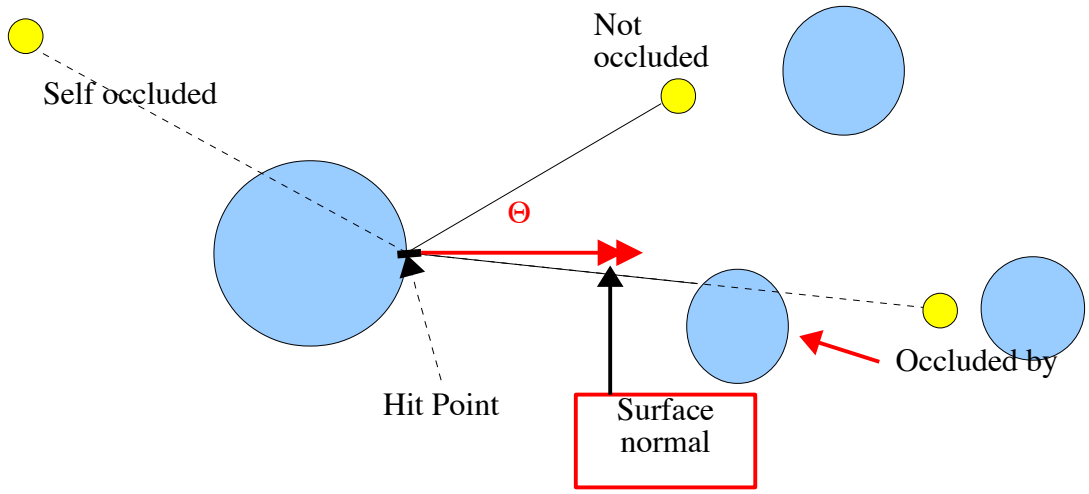
The *process_light()* function determines performs this portion of the algorithm.

```
/*
 * process_light – Comment me
 * Parameters       lst – list of all objects
 *                  hitobj – The object hit by the ray
 *                   lobj -   The current light source
 *                   ivec – [r, g, b] intensity vector
 * Return:          ???
 */
int process_light(list_t *lst, obj_t *hitobj, obj_t  *lobj,
                  double *ivec)
{

      if the hitobj occludes itself
            return
      find_closest_object() along a ray from hitloc to the center of the light
      if one exists and is closer than the light  // the light is occluded by the object
          return
      compute the illumination and add it to *pix;
}
```

Testing for occlusion

The diagram below illustrates the ways in which objects may occlude lights. The small yellow spheres represent lights and the large blue ones visible objects.



We will assume convex objects. An object is *self-occluding* if the angle between the surface normal and a vector from the hit point toward the light is larger than 90 degrees.

A simple test for this condition is that an object is *not* self-occluding if

> the *dot product* of a vector from the *hit point* to *that light* with the *surface normal* is *positive*.

To see if a light is occluded by another object, it is necessary to see if a ray fired from the *hitpoint* to the *light* hits another object before it reaches the light. This can be accomplished via a call to *find_closest_object()* The light is occluded if and only if

> (1) *the ray hits some new object*
> AND
> (2) the distance to the *hit point on the new object* is less than the *distance to the light.*

Computing the illumination

If the light does illuminate the hitpoint, its effect must be *added to* the pixel intensity vector **pix*.

*(ivec + 0) += diffuse[0] * lobj->emissivity[0] * cos(Θ) / dist_from_hit_to_light
*(ivec + 1) += diffuse[1] * lobj->emissivity[1] * cos(Θ) / dist_from_hit_to_light
*(ivec + 2) += diffuse[2] * lobj->emissivity[2] * cos(Θ) / dist_from_hit_to_light

The above computation assumes that the *diffuse[]* vector was filled in by a call to: *hitobj->getdiff(hitobj, diffuse)*. Such a mechanism will allow procedural shading to work properly in conjunction with diffuse illumination.

If procedural shading is not supported then *diffuse[]* should be replaced in the assignments shown above by *hitobj->material.diffuse[]*

Debugging diffuse illumination

Though the procedure is straightforward, there are *many* places where a minor error may render a light completely inoperative. Attempting to do problem diagnosis by looking at an *all black* image is not very productive.

There are so many things going on here, I recommend a different form of debugging than used in the simpler environment earlier. In the *process light* function you should dump relevant elements of the computation.

```
#ifdef DBG_DIFFUSE
   light_t *lt = lobj->priv;
   vl_vecprn1("hit object id was        ", &hitobj->objid);
   vl_vecprn3("hit point was            ", hitobj->hitloc);
   vl_vecprn3("normal at hitpoint       ", hitobj->normal);
   vl_vecprn1("light object id was      ", &lobj->objid);
   vl_vecprn3("light center was         ", lt->center);
   vl_vecprn3("unit vector to light is  ", dir);
   vl_vecprn1("distance to light is     ", &dist);
   vl_vecprn1("cosine is                ", &cos);
#endif
```

When another object occludes the light:

```
#ifdef DBG_DIFFUSE
/* If occluded by another object */
      vl_vecprn1("hit object occluded by ", &obj->objid);
      vl_vecprn1("distance was            ", &close);
#endif
      return(0);
   }
```

When the illumination is finally computed:

```
#ifdef DBG_DIFFUSE
   vl_vecprn3("Emissivity of the light  ", emiss);
   vl_vecprn3("Diffuse reflectivity     ", diffuse);
   vl_vecprn3("Current ivec             ", intensity);
#endif
```

Two dimensional arrays and matrix operations in C

Two dimensional arrays are declared as in Java:

double x[4][5];

The declaration above creates 4 x 5 = 20 double precision values.
The values are stored in the following order:

x[0][0], x[0][1], ... x[0][4], x[1][0], ..., x[3][4]

This is consistent with subscripting techniques commonly used in mathematics in which the first subscript represents a *row* and the second represents a *column.*

Passing two dimensional arrays as parameters

```
/**/
/* multiply two three-D matrices together */
void vl_matmult3(double inleft[][],double inright[][],
                 double out[][]) {
   double x;

   x = inleft[2][1];
}
```

```
gcc -c twod.c
twod.c: In function `vl_matmult3':
twod.c:11: invalid use of array with unspecified bounds
```

The problem here is that the compiler has now way to know *how many columns there are in each row of the matrix.* Recall the way that we implicitly handled two dimensional pixmaps

```
*(imageloc + row * numcols + col)
```

To obtain the offset of the start of a particular row we had to know how many columns were in the pixmap.

Defining arrays passed as parameters

The correct way is to specify the length of the columns (or both rows and columns)

```
void vl_matmult3(double inleft[][3],double inright[][3],
                 double out[3][3])
{
   double x;

   x = inleft[2][1];
}
gcc -c -Wall twod3.c
```

Passing matricies as actual arguments in the calling function

For the parameter passing to work correctly, all that is technically required is for the caller to provide the *address of the first element in the matrix.* As is shown below there are many ways to accomplish this mission but may of them produce undesirable compiler warnings.

```
 3 void vl_matmul3(double inleft[][3], double inright[][3],
                   double out[][3]){
       <SNIP>
10 }

12 int main() {
13     double m1[3][3];
14     double m2[3][3];
15     double *mp;
16     mp = m1[0];

17     vl_matmul3(m1, m1, m2);  <----
18     vl_matmul3(&m1[0][0], m1, m2);
19     vl_matmul3(&m1[0], m1, m2); <----
20     vl_matmul3(mp, m1, m2);
21     vl_matmul3(&m1, m1, m2);
22     return(0);
23 }
```

```
==> gcc -Wall -c twod4.c
twod4.c: In function `main':
twod4.c:18: warning: passing arg 1 of `matmul3' from incompatible
pointer type
twod4.c:20: warning: passing arg 1 of `matmul3' from incompatible
pointer type
twod4.c:21: warning: passing arg 1 of `matmul3' from incompatible
pointer type
```

These examples show that:

For the 2-D array use the name of the first row if you wish to set a pointer to the start of the array

To pass the array as a parameter use either its name or the address of its first row.

However, if the receiving function understands what the structure of the array is, it can in fact *recover from all of the above faux pas* as shown in the following example.

```
void matprn3(char *label,double mat[3][3]);

void idmat3(double mtx[3][3]);

int main(){
    double m1[3][3];
    double *mp;
    double v1[3];
    double *vp;

    vp = v1;
    mp = m1[0];

    vl_idmat3(m1);
    m1[2][1] = 4;
    m1[1][2] = 7;
    matprn3("17 ",m1);
    matprn3("18 ",&m1[0][0]);
    matprn3("19 ",&m1[0]);
    matprn3("20 ",mp);
    matprn3("21 ",&m1);
    return(0);
}
```