The *getamb()* function

Recall that in the ambient only raytracer the last steps of the operation are:

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

The first inclination is to implement the small amount of code in step 2 in the obvious way:

```
intensity[0] = closest->material.ambient[0];
intensity[1] = closest->material.ambient[1];
intensity[2] = closest->material.ambient[2];
```

However that approach would make it *not easy to override* the *default* behavior. Thus a better approach is to replace the three lines above by:

```
obj->getamb(obj, intensity);
```

During the *object_init()* object constructor sets the *getamb()* function pointer to the *default_getamb()* function which contains the three lines of code we just replaced.

While this adds a slight bit of run time overhead, it also provides us with an easy hook with which we may override the *default_getamb()* with a custom shader.

Examples of procedural shaders

Alternating stripes of various shapes appear on the back wall in the figure previously shown. The algorithm used to generate that part of the image is given here. A vector V pointing from the point defining the location of the plane to the hit point is computed first. Then the following function of V is computed.

$$f(V) = 1000 + v_x v_y^2 / 100 + v_x v_y / 100$$

The striping effect is obtained by converting the value to integer and then altering the color value that is returned based upon whether or not the integer is even or odd.

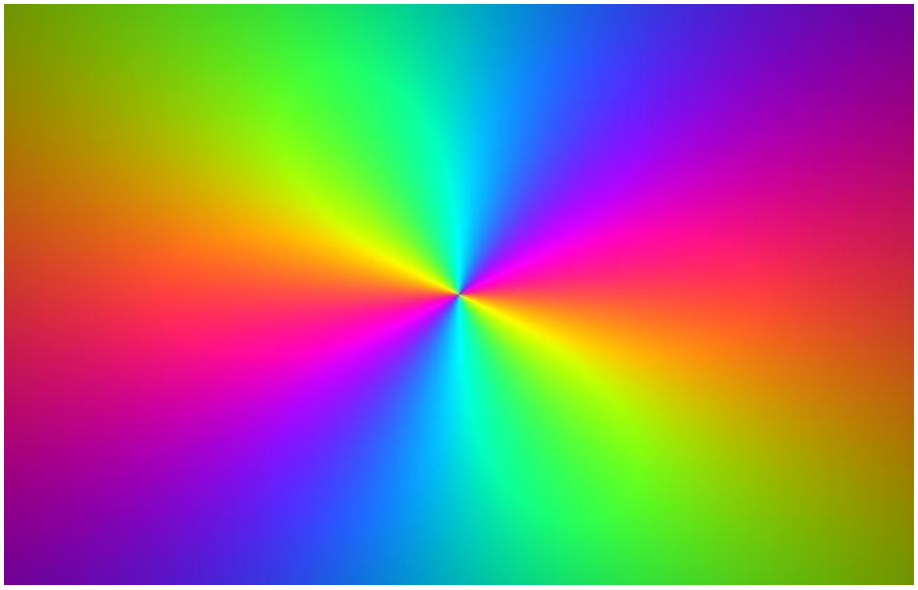
The constant value 100 affects the width of the stripe. The value 1000 is a "hack" used to avoid a nasty "double-wide" stripe where the value of the function ranges between [-1 and +1] since all values in that range are integerized to 0. Note that this procedure is designed specifically for a back wall as the *x* and *y coordinates* of the hit point vary but the *z* coordinate does not. Applying it to a floor would produce a solid color (why)??

```
void pplane1_amb(obj_t *obj,double *value){
   double vec[3];
   plane_t *p = (plane_t *)(obj->priv);
   int    isum;
   double sum;
   vl_copy3(obj->material.ambient, value);
   vl_diff3(p->point, obj->hitloc, vec);
   sum = 1000 + vec[0] * vec[1] * vec[1] / 100 +
                                  vec[0] * vec[1] / 100;
   isum = sum;
   if (isum & 1)
      value[0] = 0;    // zap red
   else
      value[2] = 0;    // zap blue
}
```

The tiling effect seen on the floor can be achieved (on the back wall) by computing separate functions $f_x(v_x)$ and $f_y(v_y)$, integerizing them separately, adding the integers and selecting color based upon whether the *sum* is even or odd.

Continuously modulated shading

The image shown below is produced by a procedural shader that continuously modulates the ambient reflectivity.



The modulation function is shown below. A vector *V* in the direction from the point defining the plane location to the hit point is computed first. Then the angle that the vector makes with the positive *X* axis is computed. Finally the red, green and blue components are modulated using the function $1 + \cos(\omega t + \phi)$ where the angular frequency ω is 2 for all three colors, and phase angles ϕ are 0, $2\pi/3$, and $4\pi/3$ respectively. Different effects may be obtained by using different frequencies and phase angles for each color, and, as shown in the example images, it is also possible to combine continuous modulation with striping or tiling.

```
vl_diff3(p->point, obj->hitloc, vec);

v1 = (vec[0] / sqrt(vec[0] * vec[0] + vec[1] * vec[1]));
t1 = acos(v1);

if (vec[1] < 0)          // acos() returns values in [0,PI]
   t1 = 2 * M_PI - t1;   // extend to [0, 2PI] here

value[0] *= (1 + cos(2 * t1));
value[1] *= (1 + cos(2 * t1+ 2 * M_PI / 3));
value[2] *= (1 + cos(2 * t1+ 4 * M_PI / 3));
```

Tables of function pointers

The model dumping and loading operation provides another example in which tables of function pointers can replace a big messy switch construct and simplify the adding of new object types. The following example

1 - declares an array of pointers to object constructor functions

2 - initializes the pointers to point to the appropriate functions

```
obj_t *dummy_init(FILE *in, int objtype){  return(NULL); }

obj_t *(*obj_loaders[])(FILE *in, int objtype) =
{
   dummy_init,        /* placeholder for type 10 (light) */
   dummy_init,        /* placeholder for type 11         */
   dummy_init,        /* placeholder for type 12         */
   sphere_init,       /* object type 13                  */
   plane_init         /* object type 14                  */
};

int model_init(FILE *in, model_t *model){
   char buf[256];
   int  objtype;
   obj_t *new;

   /* now load the objects in the scene */
   while (fscanf(in, "%d", &objtype) == 1){
      fgets(buf, 256, in); /* consume rest of line */
      if ((objtype >= FIRST_TYPE) && (objtype <= LAST_TYPE)){
         if ((new = (*obj_loaders[objtype - FIRST_TYPE])(in, objtype))
                  == NULL)
            return(-2);
         /* add new object to proper list (scene or lights) */
      } else {
           /* invalid object id */
          fprintf(stderr, "Invalid object type %d \n", objtype);
          return(-1);
      }
   }
}
```

Malloc'd objects, garbage collection, and memory leaks

The Java language provides for "automagic" garbage collection in which storage for an object is magically reclaimed when all references to an object have gone out of existence.

C provides no such mechanism.

A *memory leak* is said to have occurred when:

1. the last pointer to a malloc'd object is reset or
2. the last pointer to a malloc'd object is a local variable in a function from which a return is made,

In these cases the *malloc'd* memory is no longer accessible. Excessive leaking can lead to poor performance and, in the extreme, *program failure*.

Therfore C programmers must recognize when last pointer to *malloc'd* storage is about to be lost and use the *free()* function call to release the storage before it becomes impossible to do so.

Several examples of incorrect pointer use and memory leaking have been observed in student programs

Problem 1: The instant leak.

This is an example of an *instant leak*. The memory is allocated at the time *temp* is declared and leaked when *temp* is reassigned the address of the first object in the list.

```
obj_t *temp = malloc(sizeof(obj_t));

temp = list->head;
while (temp != NULL)
  -- process list of objects --
```

Two possible solutions:

Insert *free(temp)* before *temp = list->head;*

This eliminates the leak, but what benefit is there to allocating storage and instantly freeing it???

Change the declaration to *obj_t *temp = NULL;*

This is the correct solution.

A rational rule of thumb is *never malloc memory unless you are going to write into it!*

Another good rule of thumb is to *never declare a pointer without initializing it.*

Problem 2: The traditional leak

Here storage is also allocated for *univec* at the time it is declared.

The call to *vec_unit3()* writes into that storage.

If the storage is not *malloc'd,* then *univec* will not point to anything useful and the call to *vl_unitvec3()* will produce a segfault or will overwrite some other part of the program's data. So this *malloc()* is necessary.

```
{
    double *univec = malloc(3 * sizeof(double));

    vec_unit3(dir, univec);
           :
    more cool stuff involving univec
           :
    return;
 }
```

However, the instant the return statement is executed, the value of *univec* becomes no longer accessible and the memory has been leaked.

Here the correct solution is to add

```
  free(univec);
```

just before the return;

A rational rule of thumb is: *malloc'd storage must be freed before the last pointer to it is lost.*

Problem 3: Overcompensation

The concern about leakage might lead to an overcompensation. For example, an object loader might do the following:

```
{
     obj_t *new_obj;
          :
     new_obj = malloc(sizeof(obj_t));
          :
     if (list->head == NULL)
     {
          list->head = list->tail = new_obj;
     }
     else
     {
          list->tail->next = new_obj;
          list->tail = new_obj;
     }
     free(new_obj);

     return(0);
}
```

This problem is the reverse of a memory leak. A live pointer to the object exists through the *list* structure, but the storage has been *freed.*

The results of this are:

1. Usually attempts to reference the freed storage will succeed.
2. The storage will eventually be assigned to another object in a later call to *malloc().*
3. Then "both" objects will occupy the same storage.

Rational rule of thumb: *Never free an object while live pointers to the object exist.* Any pointers to the freed storage that exist after the return from *free()* should be set to NULL.

To fix this problem the *free(new_obj) must be deleted from the code*. If the objects in the object list are to be freed, *it is safe to do so only at the end of the raytrace.*

It is not imperative to do so at that point because the Operating System will *reclaim all memory* used by the process when the program exits.

Problem 3b: Overcompensation revisited

The *free()* function must be used *only* to free memory previously allocated by *malloc()*

```
unsigned char buf[256];
:
:
free(buf);
```

is a fatal error.

The *free()* function must be not be used to free the same area twice.

```
buf = (unsigned char *)malloc(256);

:
free(buf);

:
free(buf);
```

is also fatal**.**

The general solution: Reference counting

For programs even as complicated as the raytracer it is usually easy for an experienced programmer to know when to *free()* dynamically allocated storage.

In programs as complicated as the Linux kernel it is not.

A technique known as *reference counting* is used.

```
typedef struct obj_type
{
   int refcount;
   :
  :
} obj_t;
```

At object creation time:

```
new_obj = malloc(sizeof(obj_t));
new_obj->refcount = 1;
```

When a new reference to the object is created

```
my_new_ptr = new_obj;
my_new_ptr->refcount += 1;
```

When a reference is about to be reused or lost

```
my_new_ptr->refcount -= 1;
if (my_new_ptr->refcount == 0)
   free(my_new_ptr);
my_new_ptr = NULL;
```

In a multithreaded environment such as in an OS kernel it is *mandatory* that the testing and update of the reference counter be done *atomically*.