Introduction to C++

We will spend the remainder of the class exploring aspects of C++. As before, the raytracer will be used to illustrate the use of language.

The most significant extensions to C are:

much *stronger type checking* and

the introduction of *true O-O classes.*

a *formal inheritance mechanism* in which derived classes can specialize parent classes

formal support for *polymorphic behavior* in which a derived class may override a base class method simply by providing a method of the same name.

support for *function overloading* in which there may bedifferent implementations with a single function name.

an *operator overloading mechanism* that is analogous to function overloading

the ability to pass a parameters *by reference* in addition to the standard pass by value.

yet another *input/output library* that may be used in addtion to standard and low level I/O

The class is a generalization of the C structure and can contain:

Function prototypes or full implementations

Accessibility controls *(friend, public, private, protected)*

Structured and basic data definitions

Class definitions

Class *type definitions* operate more like C language *typedef'd* structures than pure C structure definitions in that you may use *model_t* as a type name without using the *typedef* mechanism. . You *may* use *typedef* as before though if you wish.

```
class model_t
{
public:

   model_t(void);
   model_t(FILE *in, FILE *out, int cols, int rows);
```

Example of class *constructors* and function overloading,

```
   proj_t *get_proj();
   list_t *get_list();
```

Public functions that may be called by any entity holding a reference to *model_t*

```
private:
   class proj_t *proj;
   class list_t *list;
// proj_t *proj;
// list_t *list;
};
```

Omitting the *class* here works fine too.

The *main()* function

Each C++ program must have at least one function that is not a class method. The C++ and C *main()* functions are invoked in identical ways. Here is a *main()* function used in a C++ raytracer.

As we shall see, almost all the work of the C++ raytracer is done via constructer functions.

```
/* main.c */

#include "ray.h"

int main(
int argc,
char **argv)
{
   int cols;
   int rows;

   if (argc < 3)
   {
      fprintf(stderr, "Usage: ray numcols numrows < in.txt \n");
      exit(1);
   }

   cols = atoi(argv[1]);
   rows = atoi(argv[2]);
```

The code below illustrates two ways in which class instances may be created.:

```
// model_t  *model = new model_t(stdin, stderr, rows, cols);
   model_t  model(stdin, stderr, rows, cols);
   pixmap_t pixmap(&model, rows, cols);
// delete model;
}
```

The *new* facility creates an instance of a class and returns a pointer to it. As such it is somewhat analogous to *malloc*. Unlike *malloc()* creating an object with *new* causes the *construtor* for the class to be invoked and parameters may be passed to the constructor.

Constructors

1. Are automatically called whenever an instance of the class is created
2. Must *never* have a return type --- not even *void*.
3. May be overloaded.. The function actually invoked is the one whose formal parameters "best" match the actual arguments.

Note that **proj* and **list* are *pointers* to classes, and not instances. Thus when the *model_t* is created its necessary to create the *proj_t* and the *list_t.*

```
model_t::model_t(
FILE *in,
FILE *out,
int  cols,
int  rows)
{
```

Dynamic creation of a class instance. The parameters here:
1- Determine *which constructo*r is used
2 - and are *passed to* the constructor.

```
   proj = new proj_t(in, out, cols, rows);
   list = new list_t();
```

The *new* language construct:
dynamically creates instance of the specified class
is somewhat analogous to to *malloc()*
but allows the caller to pass parameters to the constructor

An alternate construction of the *model_t*

```
class model_t
{
public:

   model_t(void);
   model_t(FILE *in, FILE *out, int cols, int rows) :
                             proj_t(in, out, cols, rows);

   proj_t *get_proj();
   list_t *get_list();

private:
   class proj_t proj;
   class list_t list;
};
```

Constructors for inner classes are specified in this way.

Here *proj* and *list* are instances rather than pointers.

Factors that determine your choice of approach here include:

the *order in which you want the constructors executed* and
the *parameters to be passed* to the constructor of the inner classes

If you nest classes as shown here, the constructors *will* be executed in the order *proj_t, list_t, model_t*. Dynamically creating the *proj_t* and the *list_t* provides additional flexibility in execution order.

Protection attributes:

public	Any holder of a pointer to an instance of the *class* may invoke any *public* method or modify any *public* data item.
private	Private methods may be invoked only from other class methods and private variables may be accessed only by methods belonging to the class.
protected	Protected methods/variables may be access be derived classes but not others.

The *proj_t* class

```
class proj_t
{

   friend class pixel_t;
   friend class model_t;

public:

   proj_t(void);
   proj_t(FILE *in, FILE *out, int cols, int rows);
   void   print_proj(FILE *);


private:
   int    load_proj(FILE *);  // could be in the constructor

   int    win_x_size;
   int    win_y_size;

   double pix_x_size;
   double pix_y_size;

   double world_x_size;
   double world_y_size;

   double projpt[3];
};
```

The *friend* directive allows *pixel_t* and *model_t* methods to access the *private* elements of *proj_t*

The *friend* directive

- May be used to permit other classes to access private data of *this* class.
- Is *not* liked by O-O purists!
- Who claim that its a *slacker's* way to compensate for bad design (which is often true!!)
- But is more efficient than relying on *get/set* functions for frequently accessed data
- Is *not reflexive.* Here *proj_t* has *no access* to private data of *model_t* or *pixel_t*

Implementations of class methods

It is common (and recommended) that implementations of class functions *not be included* in the class definition (unless the implementation is a *one-liner).* Thus the methods shown below should reside in a source module named *proj.cpp*

The :: operator is called the *scope* operator. Here it binds the *load_proj* and *print_proj* functions to the *proj_t* class.

```
#include “ray.h”

int proj_t::load_proj(void)
{
   :
   return(0);
}

int proj_t::print_proj(void)
{
   :
   return(0);
}
```

The following shows one effect of the stronger typing in C++. Because *print_proj()* was declared as returning *void* in the class definition the compiler complains (in a way that appears odd at first) when we incorrectly declare it as *int* here.

```
g++ -c work2.cpp
work2.cpp:20: prototype for `int proj_t::print_proj()' does not
match any in class `proj_t'
work2.cpp:5: candidate is: void proj_t::print_proj()
work2.cpp:20: `int proj_t::print_proj()' and `void
proj_t::print_proj()' cannot
   be overloaded
```

The message is a bit on the opaque side because *some forms* of function overloading are possible in C++.

Overloaded functions

A class may have several methods that have the *same name*. This is referred to as *function overloading*. There must be sufficient differences in *their formal parameters* for the compiler to determine which one to use based upon the *actual arguments* provided at compile time.

This *proj_t* class has three separate *load_proj()* functions.

```
class proj_t
{
public:
   int    load_proj(int);
   void   load_proj(char);
   void   load_proj(void);
   void   print_proj(void);
private:
   int    win_x_size;
   int    win_y_size;
   double world_x_size;
   double world_y_size;
   double projpt[3];
};
```

The following class definition will *not* compile, because *return* values are not used in making the binding decisions.

```
class proj_t
{
public:
   int    load_proj(int);
   void   load_proj(int);
   void   print_proj(void);
private:
   int    win_x_size;
   int    win_y_size;
   double world_x_size;
   double world_y_size;
   double projpt[3];
};
```

Return type may *not* be used to distinguish overloaded functions

Creating class instances and invoking class methods

The following example shows one way to create an instance of the *proj_t* class and to invoke the *load_proj()* method. Notes:

> In the context of a class method, class variables don't require explicit scope identification. For example, there is *no* use of *p->world_x_size.* As in Java, class variables are accessed directly. The use of a *structure instance* or *pointers* is necessary and correct only when accessing elements of *another instance* of "this" class. Because of this you should take great care to ensure that *you don't create local variables that have the same name as class variables.*
>
> *All* standard C library functions including I/O functions remaine available (but numerous C++ only library routines have been added).

```
proj_t::proj_t(FILE *in, FILE *out, int rows, int cols)
{
   win_x_size = cols;
   win_y_size = rows;

   load_proj(in);

   pix_x_size = world_x_size / win_x_size;
   pix_y_size = world_y_size / win_y_size;
   print_proj(out);
}
```

Scope operators are optional when one class method invokes another.

```
int proj_t::load_proj(FILE *in)
{
   int vc;
   char buf[256];

   vc = fscanf(in, "%lf %lf", &world_x_size, &world_y_size);
   fgets(buf, 256, in);
   vc += fscanf(in, "%lf %lf %lf", projpt, projpt + 1,projpt + 2);
   fgets(buf, 256, in);

   if (vc == 5)
       return(0);
   else
       return(-1);
}
```

A new list structure for object management.

This approach is called an *external* list. It provides a reasonably simple mechanism for us to consider various aspects of C++.

Advantages include:

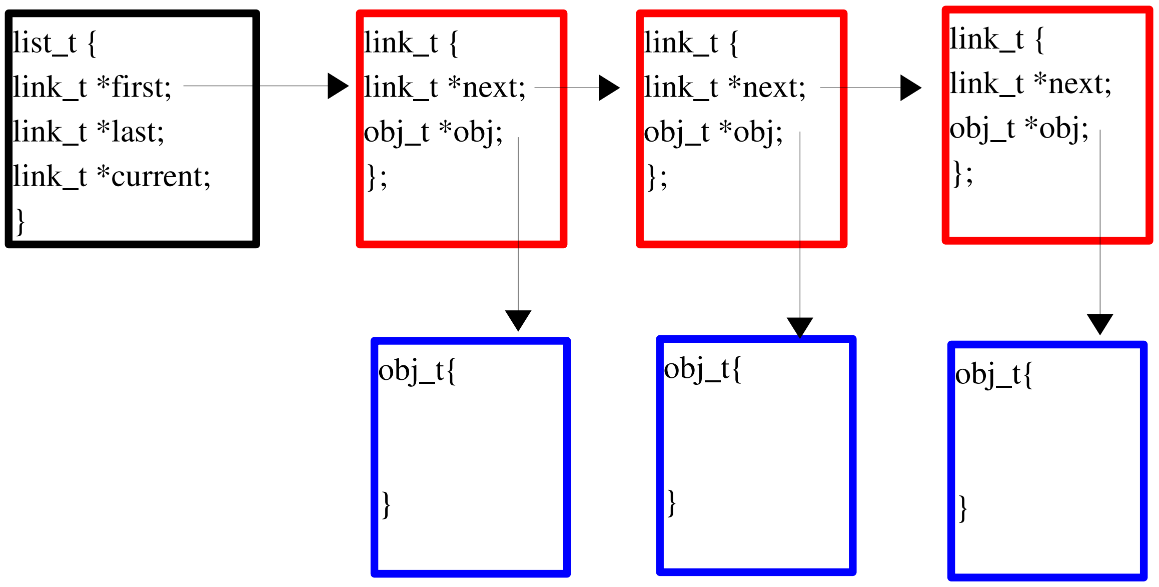
Decoupling the object from the list itself

The *link_t* objects can remain invisible to the outside world.

Facilitating the use multiple lists of objects. For example, with this techique it is easy to create a list of lights or a list of non-lights to supplement the basic object list *without modifying the object structure.*

Disadvantages include:

How to safely process the list when a new function is called by a function already in the middle of processing the list.



Example linked list classes

The *link* and *list* classes are correctly declared below: Note that the *default* constructor is explicitly provided and that it is overloaded.

Instances of the *link_t* class represent the *static structure* of the list. The values of *next* and *obj* in an instance of the *link_t* class are set by the constructor. The value of *obj* *never* changes thereafter. The value of *next* is initialized to 0, changes to non-zero when the *link* becomes *not* the last element in the list and *never changes again.*

```
class link_t
{
public:
   link_t(void);                   // default constructor
   link_t(obj_t *);                // overloaded constructor
   void    set_next(link_t *);     // used in adding new link
   link_t *get_next(void);         // retrieve the next pointer
   obj_t  *get_object(void);       // retrieve object pointer

private:
   link_t *next;
   obj_t  *obj;
};
```

While a list is being processed the *dynamic state (where we are in the list)* is maintained in the *current* pointer of the *list_t.* This *feature* can be problematic if nested processing of the list is to occur.

```
class list_t
{
public:
   list_t(void);              // constructor
   list_t (const list_t *)    // copy constructor
   void   add(obj_t *obj);    // add object to end of list
   obj_t  *start(void);       // set current to start of list
   obj_t  *next(void);        // advance to next element in list

private:
   link_t *first;             // first link
   link_t *last;              // last link
   link_t *current;           // current link.
};
```

***link_t* methods**

This constructor is passed a pointer to the *obj_t* which this new link will own:

```
link_t::link_t(obj_t *newobj)
{
   next = NULL;
   obj  = newobj;
}
```

The *set_next()* method is a typical "set" function that is used to tell the *link_t* to manipulate its own *next* pointer. It is called by the *add* method of the *list_t* class when an item that is not the first item is added to the list.

```
void link_t::set_next(link_t *new_next)
{
   next = new_next;
}
```

The *get_next()* method is a typical "get" function that is used as a way to tell the *link_t* to cough up the value of own *next* pointer.

```
link_t * link_t::get_next()
{
   return(next);
}
```

The *get_object()* method is analogous. It would also work to simply make all of the *next* and *obj* elements *public*. Then any holder of a reference to the *link_t* could simply manipulate them directly... but it would be a *violation* of OO dogma to do so.

```
obj_t * link_t::get_object()
{
   return(obj);
}
```

***list_t* class methods**

The *list_t* class overrides the default constructor with its own constructor with no parameters:

```
list_t::list_t()
{
   first = NULL;
   last = NULL;
   current = first;
}
```

Adding a new object to the list

The *add()* method creates a new *link_t* and passes its construct a pointer to the *obj_t.*
It would clearly also be possible to have a *set_obj()* method in the *link_t* class as well.

```
void list_t::add(obj_t *obj)
{
   link_t *link;
   link = new link_t(obj);
```

The *link_t* is added to the end of the *list_t* here. What would happen if the *order* of the statements in the *else* block were interchanged?? Why do we not try to set *link->next* = *NULL* here??

```
   if (first == NULL)
   {
      first = last = link;
   }
   else
   {
      last->set_next(link);
      last = link;
   }
}
```

Retrieving the *first* element of the list

The *start* method sets the *current* pointer to the first element in the list and returns a pointer to the first object in the list.

```
obj_t * list_t::start(void)
{
   if (first == NULL)  // check for empty list
       return(NULL);

   current = first;
   return(current->get_object());
}
```

Retrieving the next *obj_t* in the list.

The *next* method attempts to advance the *current* pointer. If the *current* pointer is already at the end of the list *NULL* will be returned. The use of the *persistent state variable current* will prove to be something of a pain in nested processing of the list .

```
obj_t * list_t::next(void)
{
   link_t *link;

   link = current->get_next();

   if (link != NULL)
   {
      current = link;
      return(link->get_object());
   }
   else  // current must be at end of list
   {
       return(NULL);
   }
}
```

Using the list_t class

Creating a new *list_t*

```
list_t *list;
list = new list_t();
```

Creating a new object and adding it to the list:

```
sphere = new sphere_t(in, out, objtype, objid);
list->add(sphere);
```

Note that a *sphere_t* instead of an *obj_t* is passed to *list->add* here. This works only because the *sphere_t* is a derived class of the *obj_t.*   When a new *sphere_t* is created, a new *obj_t* is automagically created at the same time and the instance of the *sphere_t* and the *obj_t* are also automagically bound together in a way the makes it possible to use a pointer to the *sphere_t.*

Processing a list

The *start* method is used to set the internal *current* pointer to the internal *first* pointer and returns a pointer to the first object in the list.

The *next* method is used to advance *current* to point to the next *link_t* and return the *obj_t* pointed to by the new *link_t.* If *current* already points to the end of the list then *NULL* is returned.

```
list_t *list = model->list;
obj_t  *obj;
 :
obj = list->start();
while (obj != NULL)
{
   obj->dump_obj(out);
   obj = list->next();
}
```

This works well *unless* inside the loop there is a call to an inner function that also need to process the list. If the inner function uses the same *list_t* as the outer one, it will leave the value of *current* at *last* breaking the caller. We will return to this issue later.

Data structures for the C++ raytracer

We have previously discussed the list managment classes *list_t* and *link_t* that are used to manage object lists and have introduced the *model_t* and *proj_t* classes.   We now complete this discussion. A raytrace model can be projed as consisting of two collections of components:

1 - Characteristics of the proj point, world coordinate system, and pixmap.
2 - The collection of objects and lights making up the scene

Management of these two collections is combined in the *model_t* class.   As noted previously it would be possible to embed *instances of* rather than *pointers to* the object list and the proj data.

```
class model_t
{
public:
   model_t(void);
   model_t(FILE *in, FILE *out, int cols, int rows);
   proj_  *get_proj();
   list_t *get_list();
private:
   proj_t *proj;
   list_t *list;
};
```

The *model_t* class is instantiated here as a *local* variable on the stack of *main*

```
#include "ray.h"

int main(
int argc,
char **argv)
{
   int cols;
   int rows;

   cols = atoi(argv[1]);
   rows = atoi(argv[2]);

   model_t  model(stdin, stderr, rows, cols);
   pixmap_t pixmap(&model, rows, cols);

}
```

This structure *will* produce a nastygram from g++ to the effect that *pixmap* is not used.
The *model_t and pixmap_t* classes could also be allocated on the heap via: (note change to *pixmap invocation*).

```
     model_t *model = new model_t(stdin, stderr, row, cols);
     pixmap_t *pixmap = new pixmap_t(model, rows, cols);
     delete model;
     delete pixmap;
```

This approach will eliminate the nastygram.

The *model_t* constructor

The *model_t* constructor contains a messy list of declarations of every object type. In the C language version this was unnecessary, it may (or may not) be unnecessary in the C++ version.

```
/**/
/* Constructor for the model_t class. */
model_t::model_t(
FILE *in,
FILE *out,
int  cols,
int  rows)
{
   int          objtype;
   sphere_t    *sphere;
   light_t     *light;
   plane_t     *plane;
   tplane_t    *tplane;
     :
   spotlight_t *spotlight;
   int          objid = 0;
   obj_t       *obj;
   char          buf[256];
```

The *model_t* constructor begins my creating new *proj_t* and *list_t* classes and therefore invokes indirectly constructors for both. Since this is the constructor for the *model_t* class, the variables *proj* and *list* are implicitly know to be member elements of the class being initialized. They *must not* be declared as local variables.

```
   proj = new proj_t(in, out, cols, rows);
   list = new list_t();
```

Completing the model loading:

In the C language version the loading of object descriptions was *table driven.*

```
while (fscanf(in, "%d", &objtype) == 1)
{
   fgets(buf, 256, in);
   if ((objtype >= FIRST_TYPE) &&(objtype <= LAST_TYPE))
   {
      rc = (*obj_loaders[objtype - FIRST_TYPE])(in, lst,
                                              objtype);
                 :
   }
}
```

Loading the object list

But in the *model_t* constructor the object list is constructed using a big ugly switch. Each individual object description is loaded by that objects constructor. But C++ does allow us to use the generic *list->add()* method to insert the object descriptions in the list. There may or may not be a way to table drive this. The problem is that the item which distinguishes each *case* is a *type (sphere_t, ellipse_t, ...).*

```
  list = new list_t();

   while (fscanf(in, "%d", &objtype) == 1)
   {
      fgets(buf, 256, in);
      switch (objtype)
      {
      case SPHERE:
      /* fprintf(stderr, "Loading sphere \n"); */
         sphere = new sphere_t(in, out, objtype, objid);
         list->add(sphere);
         break;
      case ELLIPSE:
      /* fprintf(stderr, "Loading ellipse \n"); */
         ellipse = new ellipse_t(in, out, objtype, objid);
         list->add(ellipse);
         break;
      case PLANE:
      /* fprintf(stderr, "Loading plane \n"); */
         plane  = new plane_t(in, out, objtype, objid);
         list->add(plane);
         break;
      case TILED_PLANE:
```

The *obj_t* class

The *obj_t* is the *base class* from which more specific classes are *derived*.

Note that the *void *priv* pointer that was necessary to provide “pseudo-polymorphic” behavior in C is not required at all here.   Derived classes are transparently bound to the base class in C++.

```
class obj_t
{
   friend class ray_t;
public:
           obj_t();            // constructors
           obj_t(int, int);
   virtual void   dump_obj(FILE *);
   virtual double hits(double *base, double *dir){return(-1.0);};

   virtual void   getloc(double *loc){};
   virtual void   getamb(double *){};
   virtual void   getdiff(double *){};
   virtual void   getspec(double *){};
   virtual void   getemiss(double *){};

protected:   // so they can be set by derived classes
   double  hitloc[3];     /* Last hit point              */
   double  normal[3];     /* Normal at hit point         */

private:
   int     objid;
   int     objtype;
};
```

These functions do nothing! A derived class that wants them to do something must override.

Items of interest in the definition of *obj_t* include:

The *ray_t* class is declared as a *friend* so that *ray_t* member functions can directly *read* the *hitloc, normal,* and *objtype.*

Both *hitloc* and *normal* are declared as *protected* so that the *hits* functions of the derived classes can set them.

The *virtual* functions provide the basis for *polymorphism* in C++. Each derived class may provide its own instance of these function. If it does so, the one defined in the derived class will override the definition in the base class. Each specific object type will provide its own *dump_obj* and *hits* function. The *getloc()* function is implemented only in *light_t* and *spotlight_t.*

The other *get()* functions allow objects such as tiled and procedural planes to provide their own coloring schemes.

Note that the *virtual* functions are *full implementations --- not just prototypes.* Prototypes will permit successful complilation but lead to ugly, hard to diagnose errors at link time.

Example of *link* time error caused by converting *getamb()* to prototype.

```
g++ -o ray -g -pg illum.o   light.o  list.o    main.o
material.o model.o   obj.o    sphere.o  veclib3d.o  ray.o pixmap.o
pixel.o  proj.o    spotlight.o plane.o tplane.o  ellipse.o cyl.o
cone.o -lm

light.o(.gnu.linkonce.d._ZTV7light_t+0x18):/local/westall/acad/cs2
15/examples/mwraypp11/light.cpp:10: undefined reference to
`obj_t::getamb(double*)'

obj.o(.gnu.linkonce.d._ZTV5obj_t+0x18):/local/westall/acad/cs215/e
xamples/mwraypp11/obj.cpp:10: undefined reference to
`obj_t::getamb(double*)'

spotlight.o(.gnu.linkonce.d._ZTV11spotlight_t+0x18):/local/westall
/acad/cs215/examples/mwraypp11/spotlight.cpp:10: undefined
reference to `obj_t::getamb(double*)'

collect2: ld returned 1 exit status
make: *** [ray] Error 1
```

These errors could also be corrected by adding *getamb()* function bodies to each of the classes *obj_t, light_t* and *spotlight_t* but the correct approach to provide the complete function body in the *obj_t* class definition.

The *material_t* classs

The *material_t* class manages the reflectivity properties of an object. Its methods load, dump and retrieve the ambient, diffuse, and specular reflectivity. The *material_load* method is a private method invoked only by the *constructor* and so could well be placed inline in the constructor

```
class material_t
{
public:
           material_t();
           material_t(FILE *);
   void    matamb(double *);   // return reflectivity
   void    matdiff(double *);
   void    matspec(double *);
   void    material_print(FILE *);

private:
   int     material_load(FILE *);
   double  ambient[3];     /* Reflectivity for materials  */
   double  diffuse[3];
   double  specular[3];
};
```

Standard objects will have their *getamb, getdiff, getspec* functions just call these methods.

Exercise: Why can't the virtual functions, *getamb()* and friends just invoke these?

Inheritance and polymorphism

Each derived class should be a *specialization* of a base class. As examples of this we will use consider the infinite plane object *plane_t* which is derived from the *obj_t* and the tiled plane *tplane_t* which is derived from the *plane_t.*

```
class plane_t: public obj_t, public material_t
{
public:
   plane_t();
   plane_t(FILE *, FILE *, int otype, int oid);

   virtual double  hits(double *base, double *dir);
   virtual void    dump_obj(FILE *);
   virtual void    getamb(double *w) {matamb(w);};
   virtual void    getdiff(double *w){matdiff(w);};
   virtual void    getspec(double *w){matspec(w);};

private:
   double  normal[3];
   double  point[3];
   double  ndotq;   // compute once use many times
};
```

Single and multiple inheritence

The *plane_t* class is derived from two base classes: *obj_t* and *material_t.* This is known as *multiple inheritence*. Some in O-O circles seem to think multiple inheritence is a bad thing. When only single inheritence is allowed, the object structure of a program becomes a forest of *inheritance trees*. With multiple inheritance a complex graph of interconnections can result.

Public inheritance

The type of inheritence specified here is *public.* Use of public inheritance permits *derived* class objects to also be treated as *base* class objects. We saw this being done when the *list->add()* method was invoked.

Constructors

When a new *plane_t* is created the constructors for *plane_t, obj_t,* and *material_t* will all be invoked in the order *obj_t(), material_t(), plane_t()*

Polymorphic behavior

Since the *plane_t* derived class provides its own implmentation of *hits, dump_obj, getamb/diff/spec()* when *obj->hits()* is called for a *plane_t* object the methods contained in *plane.cpp* and not the default stubs declared in the *obj_t* will be activiated. However, if *obj->getloc()* were to be invoked on a *plane_t* object the stub defined in the *obj_t* class definition would be used. Thus *hits(),* and *dump_obj()* are true *polymorphic* functions in that *they have different implementations in different classes.*

Here the *get()* functions simply reflect *get* calls for material properties to the *material* class. However, in the *tiled* plane class this will no longer be the case. Note that these “reflectors” cannot be placed in the *obj_t.* Since the *obj_t* is not derived from *material_t* an *obj_t* any attempt to reference *matamb()* from an *obj_t* would cause a syntax error of the form:

```
ray.h:62: `matamb' undeclared (first use this function)
```

Multiple layers of inheritence

The tiled plane class, *tplane_t,* illustrates multilayer inheritence. Since it is *explicitly* derived from *plane_t,* it is *implicitly* derived from *material_t* and *obj_t* as well. Thus whenever a *tplane_t* is created an instance of a *plane_t,* a *material_t,* and an *obj_t* will automatically be created as well. Constuctors are invoked in the order: *obj_t(), material_t(), plane_t(), tplane_t.*

All four of the instances will be internally bound together. Any entity that holds a pointer to anyone of them can cast the pointer to any other.

Note that the *t_plane* has a *getdiff()* method which will override the *getdiff* supplied in the *plane_t* when *obj->getdiff* is invoked on a *tplane_t* object. Dr. Westall's slack implementation of the tiled plane only supported diffuse illumination here.

```
class tplane_t: public plane_t
{
public:
   tplane_t();
   tplane_t(FILE *, FILE *, int otype, int oid);

   virtual double  hits(double *base, double *dir);
   virtual void    dump_obj(FILE *);
   virtual void    getdiff(double *w);

private:
   material_t      *altmat;
   double          gridsize[2];
   double          gridbase[3];  // the xdir
};
```

Also note that there is no *getamb()* and no *getspec().* What this means is that in Dr. Westall's present implemenation the alternate tiles *literally* ineherit the ambient and specular reflectivity of the foreground tiles. At some point in the future Dr. Westall can easily fix this by just adding new methods here. He won't have to touch the *ray.cpp* code in which these reflectivities are accessed.

```
   closest->getamb(ity);
   comp_diffuse(closest);
   vl_scale3(1.0 / total_dist, ity, ity);
```

The *tplane_t* constructor

We will consider the construction process in some detail. A *plane_t* will be implicitly constructed whenever a *tplane_t* is created. *Unless explicitly overridden as is shown below in red,* the *default* constructor for the *plane_t* would be used Note that only those attributes of the *tplane_t* that are specific to the *tplane_t* need be read by its constructor. The other items will be read by the constructors of *plane_t* and *material_t.*

Therefore the *structure of* C++ dicates the order in which we must supply input to the program: *reflectivity; plane data; tiled plane data.*

Selecting the proper parent class constructor

```
tplane_t::tplane_t(
FILE *in,
FILE *out,
int  otype,
int  oid) : plane_t(in, out, otype, oid)
{
   int pcount;
   char buf[256];

/* All of this code runs after the constructor for */
/* plane_t returns                                 */

/* Load alternate tile material description */

   altmat = new material_t(in);

   pcount = fscanf(in, "%lf", &gridsize);
   fgets(buf, 256, in);

   pcount += fscanf(in, "%lf %lf %lf",
                gridbase, gridbase + 1, gridbase + 2);

   fgets(buf, 256, in);
   if (pcount != 4)
      --- bad ----

}
```

The plane_t constructor

Recall that the *plane_t* class is a derived from both *obj_t* and *material_t.*

Therefore whenever an instance of a *plane_t* is created and *obj_t* and a *class_t* will also be created. The order in which they are created is the order specified *here:*

```
class plane_t: public obj_t, public material_t
{
public:
   plane_t();
   plane_t(FILE *, FILE *, int otype, int oid);
```

As was the case with *tplane_t* it is necessary to specify parameters to be passed to *obj_t* and *material_t* constructors here. These constructors will be called *before* the body of the *plane_t* constructor is allowed to procede. Thus in the C++ raytracer all material specifications must procede other specifications in the description file.

Identifies parameters to be passed but does not govern order of invocation.

```
plane_t::plane_t(
FILE *in,
FILE *out,
int  otype,
int  oid) : obj_t(otype, oid), material_t(in)
{
   int pcount;
   char buf[256];

   pcount = fscanf(in, "%lf %lf %lf",
                     normal + 0, normal + 1, normal + 2);
   fgets(buf, 256, in);
   pcount += fscanf(in, "%lf %lf %lf",
                     point + 0, point + 1, point + 2);
   fgets(buf, 256, in);

   if (pcount != 6)
   {
      fprintf(stderr, "Load plane expected 6 values got %d \n",
                        pcount);
   }
   vl_unitvec3(normal, obj_t::normal);
   ndotq = vl_dot3(normal, point);
 }
```

The *obj_t* constructor

The *obj_t* constructor just files away the *objtype* and *objid.* The *objid* is useful in debugging messages but is otherwise not used.

```
/* Absence of this fellow creates havoc at link */
/* time <=> inheritence is in play.             */

obj_t::obj_t()
{
   fprintf(stderr, "obj_t null constructor \n");
}

obj_t::obj_t(
int otype,
int oid)
{
   objid = oid;
   objtype = otype;
}
```

Dumping the object list

It would be possible to have each constructor invoke a dumper for each newly created object. This approach would be undesirable because it would not verify the integrity of the object list. Therefore when the *model_t* constructor encounters end of file it does the following:

```
   obj = list->start();
   while (obj != NULL)
   {
      obj->dump_obj(out);
      obj = list->next();
   }
```

When the object is actually a *t_plane,* the *dump_obj()* function of the *t_plane* class will be invoked. Unlike the case with constructors, the *dump_obj()* functions of the base class(es) will not be automatically invoked. They may be invoked explicitly by use of the the scope operator :: for true base classes or via references to classes for which this object holds a pointer.

```
void tplane_t::dump_obj(
FILE *out)
{
   fprintf(out, "Tiled plane object \n");

   plane_t::dump_obj(out);

   fprintf(out, "Tiled plane data \n");

   altmat->dump_material(out);

   fprintf(out, "Gridsize %6.2lf \n", gridsize);
   fprintf(out, "Gridbase %6.2lf %6.2lf %6.2lf \n",
                    gridbase[0], gridbase[1], gridbase[2]);

   vl_unitvec3(gridbase, gridbase);

}
```

The *dump_obj* method of the *plane_t*

As with *t_plane* it uses the scope operator to invoke the *dump_obj* function of the *obj_t*.
Since it is derived from *material_t* it may directly ask *dump_material()* to dump the base material.

```
void plane_t::dump_obj(
FILE *out)
{
   obj_t::dump_obj(out);
   dump_material(out);

   fprintf(out, "\nPlane data \n");

   fprintf(out, "Normal \n");
   fprintf(out, "%6.2lf %6.2lf %6.2lf \n",
           normal[0], normal[1], normal[2]);

   fprintf(out, "Point  \n");
   fprintf(out, "%6.2lf %6.2lf %6.2lf \n",
           point[0], point[1], point[2]);

   vl_unitvec3(normal, normal);
}
```

The *hits* method of the tiled plane.

It might appear that it would also be appropriate to do the following:

```
double tplane_t::hits(
double  *base,
double  *dir)
{
   return(plane_t::hits(base, dir));
}
```

This will work, but it is actually *completely unnecessary*. Instead we should just trust C++ to do its job and pick the *hits* function of the *plane_t* automatically.

Antialiasing with sub-pixel sampling.

Aliasing is an effect in which edges that should appear smooth actually appear jagged because of the finite size of pixels. One approach to anti-aliasing is to artificially induce an intensity gradient near any edge. One way to do this is via *random sub-pixel sampling*.

In this approach, the *world* coordinate space is partitioned into a collection of non-overlapping squares in which the actual pixel associated at the square is located at the center. Multiple rays are fired at each pixel with the direction of the ray randomly jittered in a way that ensures in passes through the proper square.

The easiest way to do this is via a simple modification to *map_pix_to _world.*

```
void map_pix_to_world(
proj_t *v,
int x,
int y,
double *world)
{
   double rx;  // These values MUST BE DOUBLE -- not int!!!!
   double ry;
   rx = randpix(x);
   ry = randpix(y);
   *(world + 0) = (1.0 * rx / v->win_x_size) * v->world_x_size;
   *(world + 1) = (1.0 * ry / v->win_y_size) * v->world_y_size;
   *(world + 2) = 0.0;
}
```

The *double randpix(int p)* function should

compute a *double precision* random number, *rv1,* in [0.0, 1.0] (see *man random*).
subtract 0.5 from *rv1* giving a random number, *rv2,* in [-0.5, 0.5]
return the sum of *rv2* and the input integer *p*
(see man *random*)

To perform anti-aliasing we insert the following code in the *make_pixel()* function *image.c*.

```
intensity[0] = intensity[1] = intensity[2] = 0.0;

for (k = 0; k < AA_SAMPLES; k++)
{
   map_pix_to_world(proj, j, i, world);
   vl_diff3(proj->projpt, world, dir);
   vl_unitvec3(dir, dir);
   ray_trace(lst, proj->projpt, dir, intensity, 0.0, NULL);
}
vl_scale3(255.0 / AA_SAMPLES, intensity, intensity);
```

now clamp intensity to the range [0, 255] and assign to pixmap

Specular lighting

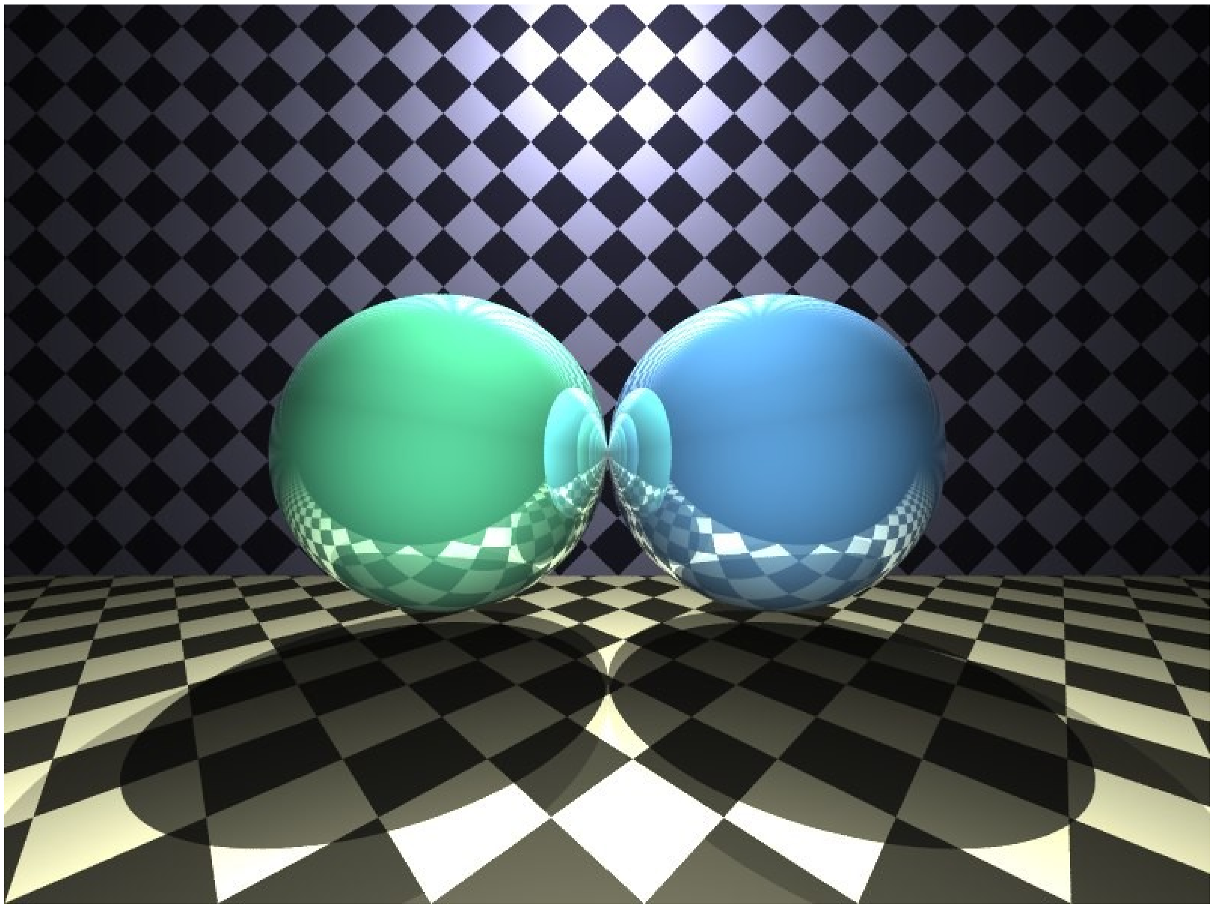
Specular light is which is coherently reflected *without scattering*. The best example of an object with no ambient or diffuse reflectivity but high specular reflectivity is a mirror.

When you look into a mirror, what you see is the reflection of light that has previously been reflected or emitted by other objects.

Therefore in a raytracing system, if a ray hits an object with a non-zero specular reflectivity it is necessary to reflect or *bounce* the ray to see what it hits next. If that object also has a non-zero specular reflectivity it is necessary to bounce the ray again.

This process continues until the bounced ray:

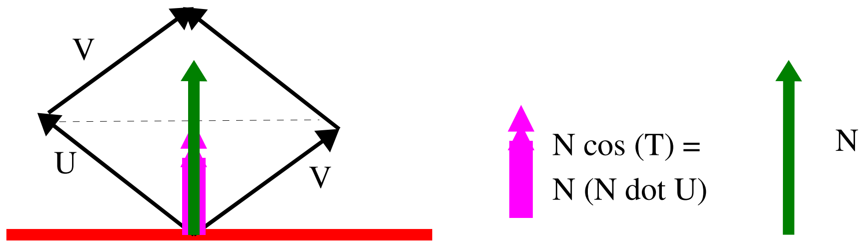
hits no object
hits an object with no specular reflectivity.
travels so far that the effect of further bounces is negligible



Relfecting a ray

Basic physics says: The angle of incidence (the angle the incoming ray makes with the normal at the hitpoint) is equal to the angle of reflection

```
vl_reflect3(
double *unitin,          /* unit vector in incoming direction  */
double *unitnorm,        /* outward surface normal             */
double *unitout);        /* unit vector in direction of bounce */
```



Let

U = -*unitin*

N = *unitnorm*

Then

U + *V* = 2 *N cos(T)* where *T* is the angle between *U* and *N*

cos(T) = *U dot N*

so

U + *V* = 2 *N (U dot N)*

and

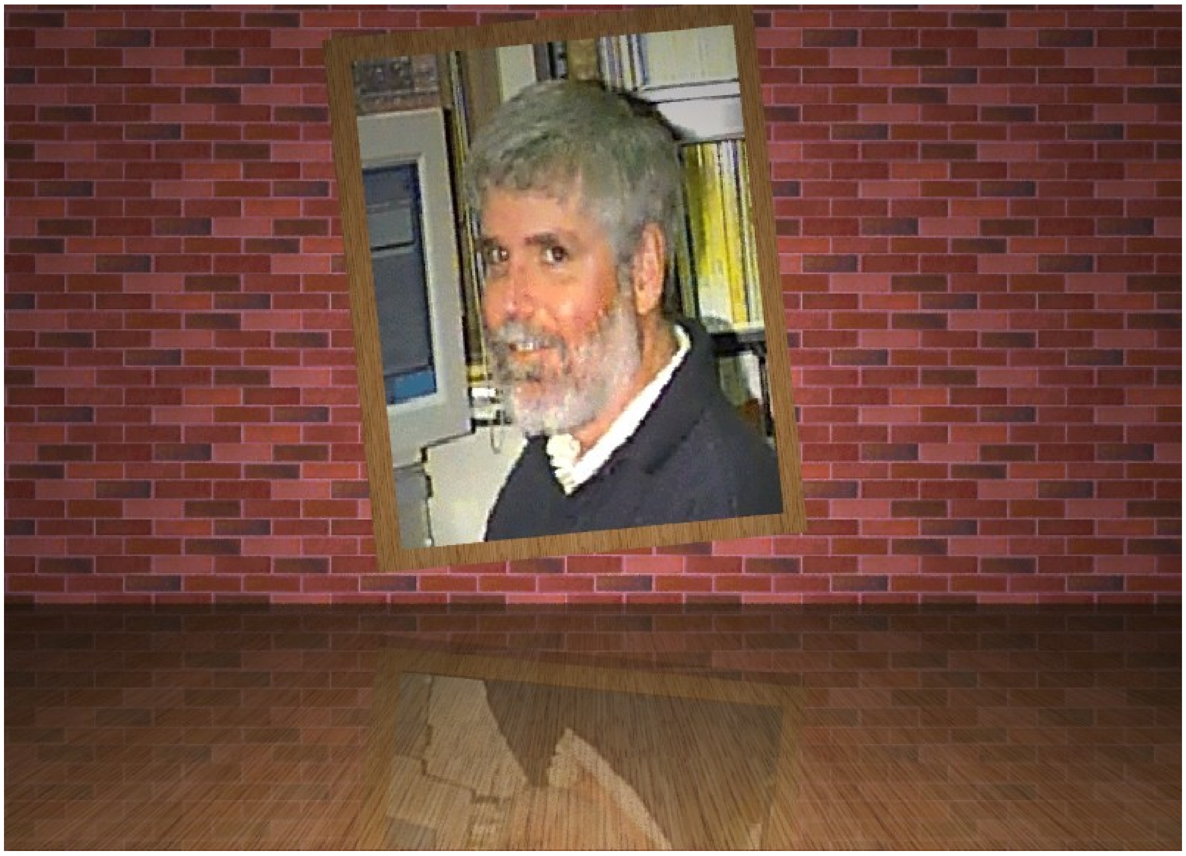
V = 2 *N (U dot N)* - *U*

The updated raytrace function:

```
void ray_trace(
list_t *lst,
double *base,        /* location of projer or previous hit */
double *dir,         /* unit vector in direction of object */
double *intensity,   /* intensity return location          */
double *total_dist,  /* distance ray has traveled so far   */
obj_t  *last_hit)    /* last obj hit if recursive call     */
{
     obj_t  *closest;
     double  mindist;
     double specref[3] = {0.0, 0.0, 0.0};

     if (total_dist > 30.0)
           return;

     Set  "closest" to point to the closest object hit by the rayt.
      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 intesity vector
     Add the diffuse reflectivity of the object at the hitpoint to the intensity vector
     Scale the intensity vector by 1 / total_dist.

     closest->getspec(specref);   /* see if object has specular reflectivity */
     if (vl_dot3(specref, specref) > 0)
     {
          double specint[3] = {0.0, 0.0, 0.0};
          compute direction, ref_dir,  of the reflected ray.
          ray_trace(model, closest->hitloc, ref_dir, specint, total_dist, closest);
          multiply specref by specint leaving result in specref
     }
     vl_sum3(intensity, specref, intensity);
} .
```

Ensure that *specref[]* is a *local copy* of the specular reflectivity! You must not corrupt the reflectivity in the material structure!!

Specular lighting may also be used in combination with other effects such as texturing.   In applications such as the *specref[]* values may be used to tune the blending of the base texture with the reflected image.



Finite planes.

The planes that we have previously used have all been of unbounded size. We can also define a finite or bounded plane class. In this class the *point* field used in an unbounded plane represents the location of the "lower left" corner of the bounded plane. To support bounded planes in all directions we also must supply a vector *xdir[]* which *lies in the the plane* and represents the directions of increase of the *x* coordinate in the plane. This vector is analogous to the grid direction in the tiled plane.

```
class fplane_t: public plane_t
{
public:
   fplane_t();
   fplane_t(FILE *, FILE *, int otype, int oid);

   virtual void    dump_obj(FILE *);
   virtual double  hits(double *base, double *dir);

protected:
   double         lasthit[2];
   double          width;
   double          height;

private:
   double          xdir[3];
};
```

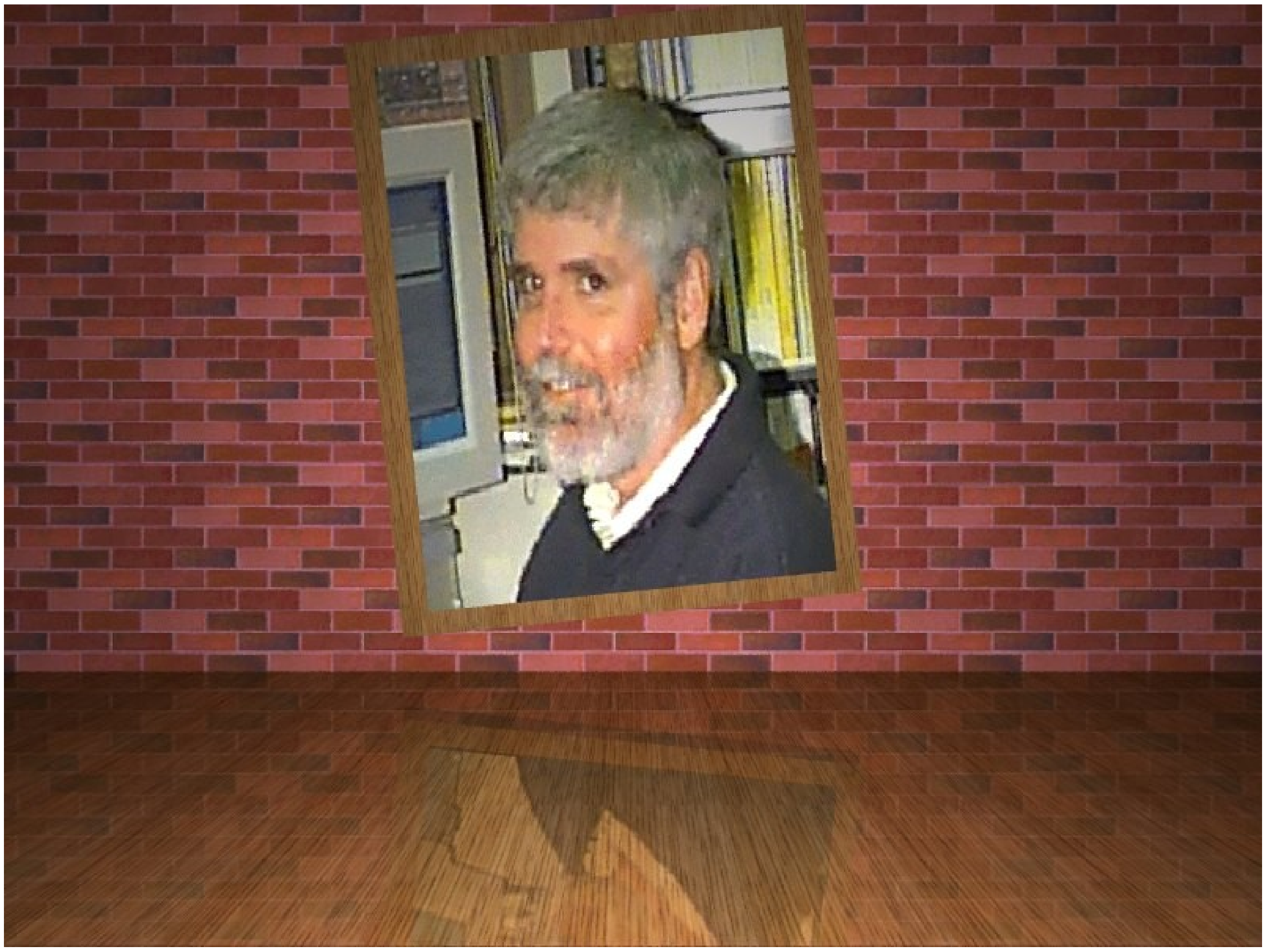
The *hits()* method of the *fplane_t* class works as follows:

> Call *plane_t::hits()* to find *if / where* the ray hits the infinite plane. If it misses *return(-1).* Construct a rotation matrix that rotates the plane's normal into the *z-axis,* the *xdir[]* vector into the *x-axis.* Make a copy of the *hitloc[].* Call the copy *testhit[].* Subtract the *base point* of the plane from *testhit[]* and apply the rotation to the result.
>
> Now if *0 <= testhit[0] <= width* and *0 <= testhit[1] <= height* a hit on the finite plane has occured and *t* is returned. Otherwise *-1* is returned.
>
> Before returning the *x* and *y* offsets of the hit are saved as *lasthit[0] = testhit[0], and lasthit[1] = testhit[1].*

The textured plane

In assignment one you wrote a program which loaded a *ppm* image and then mapped it onto a rectangular plane. Such an image is commonly called a *texture* and the process of mapping it onto a planar surface is called *texture mapping*.

We can incorporate this approach into the raytracer. The *textured plane* is simply a *finite plane* onto which a texture has been mapped. Both *tiled* and *fit* mode may be employed as shown in the following example.



The image above includes 4 textured planes. For the brick wall, the oak floor, and the frame on which the photograph is mounted, the texture is mapped in *tiled* mode. The photograph itself is mapped in *fit* mode. Note specular refection of the wall and the photograph.

The textured plane definition

The textured plane definition requires two items of information beyond that of the finite rectangular plane: the name of the file containing the texture and the mapping mode. Mapping mode *1* means stretch the texture to *fit* the plane. Mapping mode *2* means repeatedly *tile* the texture.

```
15             textured plane
6.0 6.0 6.0    amb, diffuse, spec
4.0 4.0 4.0    amb, diffuse, spec
0.0 0.0 0.0    amb, diffuse, spec
0 0 1          normal
-8 -6 -4.2     point

1 0 0          xdir
24 18          width height

brick8.ppm     name of the texture file
2              tile mode (1 means fit mode)
```

The *tex_plane* inheritance structure

```
typedef struct obj_type
{
   void    *priv;

} obj_t;

typedef struct plane_type
{
   double  normal[3];
   double  point[3];
   double  ndotq;
   void    *priv;         /* Data for specialized types  */
}  plane_t;

typedef struct fplane_type
{
   double  xdir[3];
   double  size[2];
   double  lasthit[2];
   void    priv;           /* data for specialized types */
} fplane_t;

typedef struct texplane_type
{
   int         texmode;     /* 1 -> fit and 2-> tile     */
   char        texname[40]; /* name of .ppm file         */
   texture_t  *texture;     /* pointer to texture struct */
}  texplane_t;
```

Loading a textured plane

The textured plane management functions *texplane_load, texplane_dump, texplane_amb, texplane_diff,* and *texplane_spec* should reside in a new module called *texplane.c*

```
/**/
obj_t   *texplane_load(
FILE    *in,
list_t  *lst,
int     objtype)
{
   texplane_t *texp;
   fplane_t   *fp;
   plane_t    *p;
   obj_t      *obj;
   int        pcount;
   char       buf[256];

/* Create fplane_t, plane_t, and obj_t structures */

   obj = fplane_load(in, lst, objtype);
   if (obj == NULL)
      return(NULL);

   recover pointer to fplane_t structure
   malloc new texplane_t and link it to fplane_t

   set obj->hits to fplane_hits

   set obj->getamb/getdiff to texplane_getamb/getdiff

   read in texture file name and texture file mode.

   if (texture_load(texp))
      return(NULL);

   return(obj);
}
```

Loading the texture

Each texture will be represented by a structure of the following type

```
typedef struct texture_type
{
   int            size[2];   /* x dim – y dim */
   unsigned char *texbuf;    /* pixmap buffer */
}  texture_t;
```

The *texture_load()* and *texture_map()* functions should live in a new module named *texture.c*

```
int texture_load(
texplane_t *tp)
{
   attempt to fopen() the image file and return(-1) on error

   read and parse the .ppm header returning (-1) if it's not a P6 .ppm image

   malloc a texture_t structure and link it to the texplane_t

   save texture dimensions in the texture_t

   malloc a texture buffer to hold the pixel data and save its address in the texture_t

   read in the pixel data from the pixmap.

   return(0) on success and -1 on failure

}
```

Determining the *diffuse* pixel color of the *textured* plane.

As was the case with the *tiled* plane, the real action occurs in *getamb/getdiff*. We will describe the action of the *getdiff()* function *texplane_diff()*.

The basic idea is that the intensity returned will be the *product* of the object's reflectivity with the texel that maps to the hit point. The *texture_map()* function in *texture.c* is responsible for determining what *texel* maps to the hit point.

```
/**/
void texplane_diff(
obj_t *obj,
double *value)
{
   plane_t  *pln = (plane_t *)obj->priv;
   fplane_t *fp  = (fplane_t *)pln->priv;

   double texel[3];

   texture_map(fp, texel);
   vl_mult3(obj->material.diffuse, texel, value);

}
```

The *texture_map* function

This mission of this function is to determine the *texel* which maps to the most recent hit point on the object. In general this would appear to be a difficult thing to do but, as with the finite plane, its not so hard if the textured plane is based at (0, 0, 0), has *x* direction (1, 0, 0), and normal (0, 0, 1).

Thus, to simulate this condition, we can simply use the "newhit" data that was computed in *fplane_hits* and stored in *fp->lasthit[]* at the time the *hit* was found:

```
  fp->lasthit[0] = newhit[0];
  fp->lasthit[1] = newhit[1];
```

Acquiring the value of a texel is easy when the *x* and *y* offsets of the hit are expressed as a fractional percentage of the *x* and *y* dimensions of the texture. This is the main mission of the *texture_map()* function.

```
/**/
int texture_map(
fplane_t *fp,
double *texel)
{
    recover pointers to the texplane_t and the texture_t

    if the texture mode is TEX_FIT
    {
       compute the fractional x-offset, xfrac, of the texel as fp->lasthit[0] / fp->size[0]
       pass xfrac and yfrac to texel_get()
    }
   else  /* mode is TEX_TILE */
   {
       This mode is slightly more complicated because the size of the texture must be considered.
       There are two possible ways to do this:  (1) convert the size of the texture to world
       coordinates or (2) map the fp->lasthit to pixel coordinates.

       Since we already have a map_pix_to_world() function it might seem easier to use approach
       (1) but  I recommend using approach (2) because it corresponds better to the approach
       suggested in assn 1.  To do this you will need to build a map_world_to_pix() function.

       Assuming you have done map_world_to_pix(fp->lasthit,  pixhit);

       Then xfrac = (double)(pixhit[0] % texture->size[0])/ texture_size[0];
       pass xfrac and yfrac to texel_get()
   }
```

The *texel_get()* function

This functions mission is to copy the contents of the texel at fractional position (xrel, yrel) into the texel passed as a parameter.

```
void texel_get(
texture_t *tex,
double xrel,      /* relative offset [0.0, 1.0) of texel */
double yrel,      /* within the texture                  */
double texel[3])
{
   unsigned char *texloc;
   int xtex;      /* x coordinate of the texel */
   iny ytex;      /* y coordinate of the texel */

   multiply x rel and y rel by respective texture dimensions getting xtex and ytex;
   ensure 0 <= xtex < x texture size and 0 < = ytex < y texture size

   compute offset, texloc, of the (ytex, xtex) in texture in the usual way

   texel[0] = *texloc / 255.0;
   texel[1] = *(texloc+1) / 255.0;
   texel[2] = *(texloc+2) / 255.0;

}
```

Mapping world to pixel coordinates

You will note that information in the *proj_t* is required to map between pixel and world coordinates. Furthermore the *proj_t* structure is not passed down the diffuse and ambient lighting paths.

There are several ways (all somewhat ugly) to work around this:

(1) pass a pointer to the *proj_t* from *mk_image()* through *raytrace()* through *diffuse_illumination()....*

(2) in module *proj.c* where the mapping functions should reside (I am aware some have wandered off in the direction of *veclib3d.c*), create a *static proj_t *p;* that is *not* in the body of any function. Initialize it to point to the *proj_t* structure in *projection_init().* Then use this pointer to access the *proj_t* structure in *map_world_to_pix().*

(3) Add *pix_x_size* and *pix_y_size* elements to the *fplane_t* structure. Then hack a patch into *model_load()* (which does hold a reference to the *proj_t),* Each time a *texplane_t* is loaded, the patch can compute the values of *pix_x_size* and *pix_y_size.*

The third method is the one I used in my C++ version since I already had to have the *case* structure in place anyway.

```
    case TEX_PLANE:
    /* fprintf(stderr, "Loading finite pln\n"); */
       texplane  = new texplane_t(in, out, objtype, objid);
       texplane->set_pixsize(proj->pix_x_size, proj->pix_y_size);
       list->add(texplane);
       break;
```

In my C version I decided that option (2) was the easiest and least intrusive.

The *pixmap* class

Recall the *main()* function of the C++ raytracer:

```
int main(
int argc,
char **argv)
{
   int cols;
   int rows;

   if (argc < 3)
   {
      fprintf(stderr, "Usage: ray numcols numrows < in.txt \n");
      exit(1);
   }

   cols = atoi(argv[1]);
   rows = atoi(argv[2]);

   model_t  model(stdin, stderr, rows, cols);
   pixmap_t pixmap(&model, rows, cols);
// delete model;
}
```

The constructor for the pixmap class is the *only* element of the class and drives the raytrace procedure.

```
class pixmap_t
{
public:
   pixmap_t(void);
   pixmap_t(model_t *, int rows, int cols);
};
```

An alternative design is to avoid the class structure and use C structure and just call the *make_image()* function as done previously.

The *pixmap* constructor

The *pixmap* constuctor is similar in structure to *make_image()* procedure. Instead of invoking *make_pixel* for each pixel in the image it creates a new instance of the *pixel_t* which drives the *raytrace* and then *deletes* it after retrieving the pixel value.

```
/* pixmap.cpp */

#include "ray.h"

int grow;  // globals used for debugging...
int gcol;  // other modules may access via extern int gcol;

pixmap_t::pixmap_t(
model_t *model,
int      rows,
int      cols)
{
   pixel_t *pix;
   int      j, k;
   unsigned char *pixmap;

   pixmap = (unsigned char *)malloc(3 * rows * cols);

   for (j = 0; j < rows; j++)
   {
      for (k = 0; k < cols; k++)
      {
          grow = j;
          gcol = k;
          if ((j == 150) && (k == 200))
              fprintf(stderr, "here");
          pix = new pixel_t(model, j, k);
          pix->getpix(pixmap + 3 * ((rows - 1 - j) * cols + k));
          delete pix;
      }
   }
   fprintf(stdout, "P6\n%3d %3d\n 255\n", cols, rows);
   fwrite(pixmap, 3, rows * cols, stdout);
}
```

Why not embed the actual pixmap as a data element of the pixmap_t class??

Row 150 column 200 is near the center of a 400 x 300 image. This is an easy way to use gdb to test that pixel only

The pixel_t class

The constructor for the *pixel_t* drives the ray trace. The *getpix* method handles the scaling by 255.0 and clipping to the range [0, 255]. The *map_pix_to_world()* function is in the *wrong class*.

```
class pixel_t
{
public:
   pixel_t(void);
   pixel_t(model_t *model, int row, int col);
   void getpix(unsigned char *);
   void map_pix_to_world(proj_t *proj,int x,int y,double *world);

private:
   int    row;          /* row and column location in map */
   int    col;
   double rgb[3];       /* rgb intensity from raytrace */
};
```

The *map_pix_to_world()* function: an example of *bad design*

This function produces a randomized world coordinate suitable for use in anti-aliasing. This alternative approach to antialiasing works in world space, requires that the world dimimensions of the pixel be known, and *should not be used in your raytracer*! The references shown in *red* work only because the *proj_t* class declared the *pixel_t* class to be a *friend*.

If the designer of this system had been more clever he would have seen that this method *belonged in the proj_t class!*

```
void pixel_t::map_pix_to_world(
proj_t *v,
int     y,
int     x,
double *world)
{
   int    rv;
   double off;

   *(world + 0) = (1.0 * x / v->win_x_size) * v->world_x_size;
   *(world + 1) = (1.0 * y / v->win_y_size) * v->world_y_size;

   rv = random();
   off = 1.0 * rv / RAND_MAX - 0.5;
   off *=  v->pix_x_size;
   *(world + 0) += off;

   rv = random();
   off = 1.0 * rv / RAND_MAX - 0.5;
   off *=  v->pix_y_size;
   *(world + 1) += off;

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

The pixel constructor

This function works similarly to the pixmap constructor. For each anti-aliasing sample it creates a new instance of *ray_t* and then destroys it. The persistent state of the *ray* object reduces the number of parameters that must be passed to the *trace* method.

```
/**/
pixel_t::pixel_t(
model_t *model,
int      row,
int      col)
{
   double screen[3];
   double dir[3];
   int    hit;
   ray_t  *ray;
   proj_t *proj = model->get_proj();
   int    i;

   vl_zero3(rgb);
   for (i = 0; i < AA_SAMPLES; i++)
   {
      map_pix_to_world(proj,  row, col, screen);
      vl_diff3(proj->projpt, screen, dir);
      vl_unitvec3(dir, dir);

      ray = new ray_t(model, proj->projpt, dir);
      hit = ray->trace(0.0, NULL);

      if (hit)
      {
        ray->getity(rgb);

      }
      delete ray;
   }
   vl_scale3(1.0 / SAMPLES, rgb, rgb);
}
```

The *ray* class.

The ray class has two “user level” constructors. This is probably OOverkill but the reason is that for each specular “bounce” a new ray is created. There is a fair amount of overhead in this, but it helps to keep clear in everyones head how the total intensity is being accumulated.

One might ask: ``Why not just make the raytrace a class method of the pixel class?". The excessively “classy” organization used here is for illustrative purposes only. It *will* produce poor performance. For example, a pixmap of 1000 x 1000 in size with 10 antialiased samples per pixels and 10 bounces on average per ray would require creating and destroying 100,000,000 rays!!!

Blind loyalty to the O-O paradigm in such a way as this is precisely how O-O earned its reputation for poor performance. For decent performance it is important *not* to create and destroy objects within deeply nested loops with large iteration counts.

```
class ray_t
{

public:
   ray_t();
   ray_t(model_t *model, double *b, double *d);
   ray_t(const ray_t *oldray, double *b, double *d);
   int     trace(double dist, obj_t *last);
   void    getity(double *loc);
   list_t *getlist();

private:
   void    comp_diffuse(obj_t *hit);
   int     light_visible(obj_t *, obj_t *, double *, double *,
                         double);
   list_t *list;
   double  base[3];
   double  dir[3];
   double  ity[3];     /* Intensity       */
};
```

Ray constructors

Both constructors perform the same functions: retrieve a reference to the object list; save the ray's *base* and *direction* and initialize the intensity vector.

```
ray_t::ray_t(
model_t *model,       /* -> model class       */
double *b,            /* base of ray          */
double *d)            /* direction of the ray */
{
    list = model->get_list();
    vl_copy3(b, base);
    vl_copy3(d, dir);
    vl_zero3(ity);
}


ray_t::ray_t(
const ray_t *oldray,    /* -> ray class         */
double *b,              /* base of ray          */
double *d)              /* direction of the ray */
{
    list = oldray->list;
    vl_copy3(b, base);
    vl_copy3(d, dir);
    vl_zero3(ity);
}
```

The *trace* function

Because of the work of the constructor, only two parameters are now passed to *trace*. Furthermore the nature of the recursion is altered, because *trace will not be called recursively* within the context of a single instance of a *ray_t* object.

```
/**/
int    ray_t::trace(
double total_dist,    /* distance ray has traveled so far   */
obj_t  *last_hit)     /* most recently hit object           */
{
   link_t  *link;
   obj_t   *obj;
   double  dist;
   obj_t   *closest = NULL;
   double  mindist;
   double  specular[3] = {0.0, 0.0, 0.0};
   ray_t   *newray;
```

The basic operation is nearly unchanged though. The main difference is disappearance of a number of pointers.

```
   if (total_dist > 30.0)
      return(0);

   closest = find_closest_obj(list, NULL, &mindist);

   if (closest == NULL)
      return(0);
```

Computing ambient, diffuse, and specular reflectivity

This is also virtually unchanged from the approach used in the C version. Here it is assumed that *diffuse_illumination* will *add to* and not *set* the ray's *ity*.

```
/* Hit something not a light */

   closest->getamb(ity);
   diffuse_illumination(closest);
   vl_scale3(1.0 / total_dist, ity, ity);
```

In the specular computation, the vector *specular* refers to the specular reflectivity of the object and *specint* to the visible specular intensity. The recursion will create a *stack* of rays. When either there is not a hit or the total distance becomes too large, the recursion will unwind.

```
   closest->getspec(specular);
   if (specular[0] || specular[1] || specular[2])
   {
      double specint[3] = {0.0, 0.0, 0.0};
      double bounce[3];
      int    hit;

      vl_reflect3(dir, closest->normal, bounce);
      newray = new ray_t(this, closest->hitloc, bounce);
      hit = newray->trace(total_dist, closest);
      if (hit)
      {
         newray->getity(specint);
         vl_mult3(specint, specular, specint);
         vl_sum3(ity, specint, ity);
      }
      delete newray;
   }

   return(1);
```

Dealing with lights

A *light_t* is derived from both an *illum_t* and an *obj_t*. The *illum_t* actually holds the emissivity but is is declared *protected* so an instance of *light_t* can access it.

```
class illum_t
{
public:
           illum_t();
           illum_t(FILE *);
   void    dump_illum(FILE *);
   void    getemiss(double *);
protected:
   double  emissivity[3];     /* Emissivity for lights  */
};
```

The *obj_t* has a virtual *getemiss()* method which is overridden by the *light_t.*

```
class light_t: public obj_t, public illum_t
{
public:
   light_t();
   light_t(FILE *, FILE *, int otype, int oid);

   virtual void    dump_obj(FILE *);
   virtual double  hits(double *base, double *dir);
   virtual void    getloc(double *);
   virtual void    getemiss(double *);
protected:
   double  center[3];
 };
```

The *light_t* class just copies the *emissivity* from the *illum_t.*

```
void light_t::getemiss(
double *where)
{
   vl_copy3(emissivity, where);
}
```

Possible ways of accessing the emissivity

The "normal" way to do this is to use the polymorphic *getemiss()* method of the *obj_t,* but both of the approaches shown disabled below will also work correctly.

They also demonstrate the use of casting in C++. The cast, `static_cast<light_t *>(closest);` is called a *down* cast and will extract a pointer to the *light_t* object from the *obj_t* of which it is a part. Such casts should be used with extreme care as they will "work" even if the *obj_t* is actually a tiled plane.

The cast `static_cast<illum_t *>(light);` is called an *up* cast because it extracts a pointer to a parent class from which the *light_t* is derived. Care must also be taken to ensure that *light* is actually a pointer to a *light_t.*

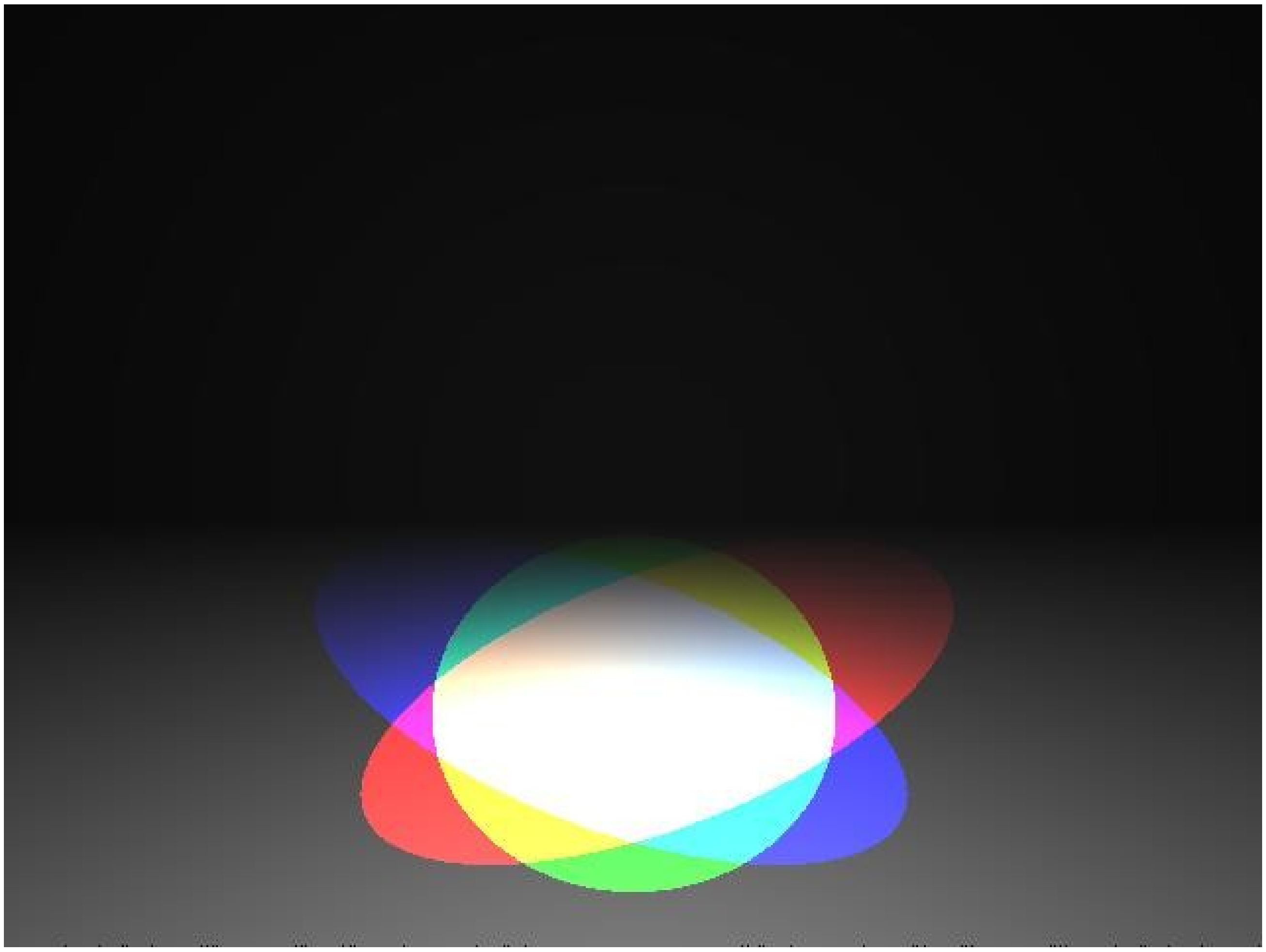
Given a pointer to a class an any level in a class hierarchy, it is possible to navigate the entire inheritance structure using static casts *if you are sure exactly what the true class of the object is.*

```
   obj_t *closest;

   :
   if (closest->objtype == LIGHT)
   {
#if 0
      light_t *light = static_cast<light_t *>(closest);
      light->getemiss(ity);
      illum_t *illum = static_cast<illum_t *>(light);
      illum->getemiss(ity)
#endif
      closest->getemiss(ity);
      vl_scale3(1.0 / total_dist, ity, ity);
      return(1);
   }
```

Spotlights

The *light* object that we have been using radiates light in all directions.

A *spotlight* can be thought of as lying at the base of a *cone* and illuminating only that area which is visible from the base of the cone.



Definining a *spotlight* object requires two additional items of information beyond that required for an omnidirectional light:

> The direction the spotlight is pointing
> The cosine of the angle defining the width of the cone

```
typedef struct spotlight_type
{
   double  direction[3];      /* Cone centerline vector    */
   double  theta;             /* Half-width of cone in deg */
   double  costheta;          /* precomputed cos(theta)    */
}  spotlight_t;
```

It turns out to be much easier for a human to *aim* the spotlight if the human is allowed to specify a point on the centerline of the spotlight instead of the direction it points. The direction is needed for visibility computations and can be computed as:

```
     vl_diff3(spot->center, spot->hits, spot->direction);
```

```
16              spotlight
4  4  3         center
0  32  0        emissivity
4  -4  -4       a point on the floor that the centerline hits
20.0            cone's half-width theta in degrees
```

Loading the spotlight data

Although it seems easy to “hack” the spotlight code into the existing light loader.

```
if (objtype == SPOTLIGHT)
{
   pcount = scanf("%lf %lf %lf",
                  new->hits,
                  new->hits + 1,
                  new->hits + 2);
   fgets(buf, 256, in);
   pcount += scanf("%lf" ,  &theta);

  Compute cos(theta) and save in spotlight structure.
  Recall cos() wants radians not degrees.

}
```

The theory of *you-touched-it-you-broke-it* says ts better not to do that. Instead, the spotlight should be treated as a derived class of the *light.*

The *spotlight_init* function()

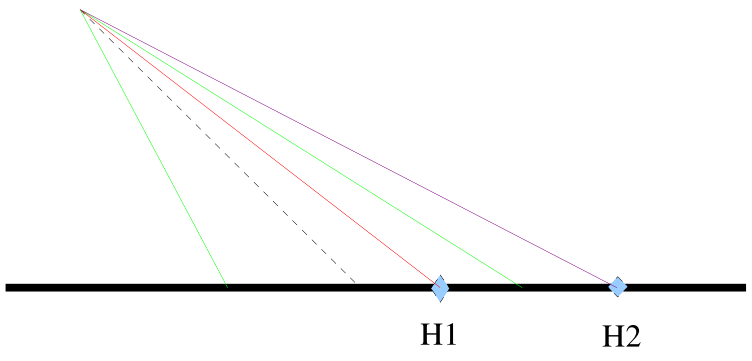
The *spotlight_init()* function is analogous to *fplane_init()*.

```
/**/
obj_t   *spotlight_init(
FILE    *in,
int     objtype)
{
   obj_t          *obj;
   light_t        *light;
   spotlight_t    *spot;
   int            pcount = 0;
   double         theta;
   double         hits[3];

   obj = light_init(in, objtype);
   if (obj == NULL)
      return(obj);

   light = (light_t *)obj->priv;
   spot = malloc(sizeof(spotlight_t));

   light->priv = (void *)spot;
   pcount = vl_get3(in, hits);
   pcount += vl_get1(in, &theta);

   if (pcount != 4)
      return(0);

   Convert hits[3] to spot->direction.

   Compute cos(theta) and save in spotlight structure.
   Recall cos() wants radians not degrees.

   obj->illum_check = spot_visible;

     return(obj);
```

Adjustments to lighting computations

A spotlight can illuminate the *hitloc* if and only if a vector from the *center* of the spotlight to the *hitloc* lies inside the spot cone.   Therefore it is necessary to incorporate such a test in the *process_light()* procedure of the *diffuse illumination* module.

In the diagram below the dashed line is the spotlight centerline and the green lines delimit the spot cone. The *hitloc* lies inside the spot cone if and only the angle between the centerline vector and a vector from the center of the spotlight to the hitpoint is less than *theta* the angle that defined the halfwidth of the spot cone. The point H1 is illuminated by the spotlight but H2 is not.



Therefore to determine if a *hitloc* is illuminated:

1. Compute a *unit* vector *from* the center of the spotlight *to* the *hitloc*
2. Take the dot product of this vector with a *unit* vector in the direction of the centerline.
3. If this value is *greater than* the *costheta* value previously computed, the *hitloc* is illuminated.

Implementing the illumination test

As usual it would be possible to hack the test into the middle of process light.

```
/* If the object is a spotlight ensure the hitpoint is within */
/* the cone.                                                  */

   if (lobj->objtype == SPOTLIGHT)
   {
     if hitobj->hitloc is not in the spot cone
          return(0);
   }

   obj = (obj_t *)lst->head;
   while (obj != NULL)
```

As usual this would be a bad idea.   A better idea is to use a "virtual" or polymorphic function that could be used to test not only *spotlights* but also *projectors* and other conceivable light forms. In the C language it is common to use an *if it exists, call it* approach for implementing such functions. In this way an omni-directional light doesn't need to provide a *default* function that always returns *0*. At the end of *spotlight_init()* the *obj->illum_check* pointer was set to point to the function *spot_visible.*

```
/* If the light is a directional light such as a spotlight or */
/* a projector it may have a special visibility function      */

   if (lobj->illum_check)
   {
      if (lobj->illum_check(lobj, hitobj->hitloc))
         return(0);
   }
```

However, it can also be argued that doing it this way is *dangerous* because some later program maintainer might not realize that *illum_check()* functions were optional and might (fatally) attempt to invoke one via a NULL pointer.