Introduction to C++

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

The most significant extensions to C are:

- much *stronger type checking* missing casts that produce warnings in C produce errors in C++
- 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 be different implementations with a single function name.
- an *operator overloading mechanism* that is analogous to function overloading
- the ability to pass 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 *struct*ure and can contain:

- Function prototypes or full implementations
- Accessibility controls *(friend, public, private, protected)*
- Structured and basic data definitions

An object oriented list structure

This approach is the C++ version of the *external* list structure we developed in C. It provides a reasonably simple mechanism for us to consider various aspects of C++.

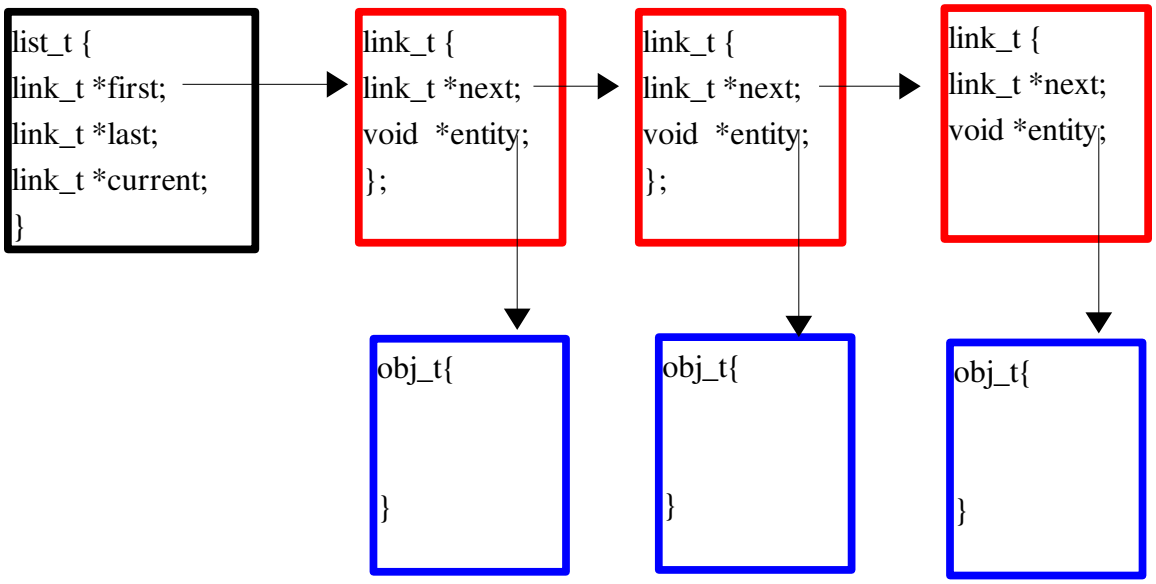
Advantages include:

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

Facilitating the use multiple lists of different kinds of entities. For example, with this techique it is easy to create a list of lights, materials, or visible *without modifying the object structure or the list 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 *entity* in an instance of the *link_t* class are set by the constructor. The value of *entity* *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.*

- The class definitions should go in a file called *list.h.*
- The actual implementations of the functions should go in a filed called list.cpp

```
class link_t
{
public:
   link_t(void);                // default constructor
   link_t(void *);              // constructor with entity ptr
   void     set_next(link_t *); // sets next pointer in this link
   link_t  *get_next(void);     // gets next pointer in this link
   void    *get_entity(void);   // gets entity pointer

private:
   link_t  *next;               // actual next pointer
   void    *entity;             // actual entity pointer

};
```

The *list_t* class.

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.

```
typedef class list_t
{
public:
   list_t(void);                // constructor
   list_t(const list_t *);      // copy constructor
   void   add(void *entity);    // add entity to end of list
   void   *start(void);         // set current to start of list
   void   *get_next(void);      // get next entity in list

private:
   link_t *first;               // first link in the list
   link_t *last;                // last link in the list
   link_t *current;             // current link
} list_t;
```

link_t class methods

This constructor is passed a pointer to the entity which this new link will own. Its mission is to set the *next* pointer to NULL and the *entity* pointer to the new entity being added to the list. Notice that it is not necessary to malloc the new *link_t*. That is done "automagically" within the *new* mechanism.

The :: operator is called the *scope operator*. It is used to declare that the *link_t()* constructor function belongs to the *link_t* class. It is always necessary to use the scope operator when:

- prototypes are used in the class declaration and
- the function body is defined outside the class declaration.

```
link_t::link_t(void *newentity)
{
   next = NULL;
   entity = newentity;
}
```

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)
{
/* set the next element of this link to 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 the next pointer of this link */

}
```

The *get_entity()* method is analogous. It would also work to simply make all of the *next* and *entity* 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.

```
void * link_t::get_entity()
{
/* return the entity pointer of this link */

}
```

list_t class methods

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

```
list_t::list_t()
{
/* Set first, last, and current pointers to NULL */

}
```

Adding a new object to the list

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

```
void list_t::add(void *entity)
{

/* Create a new link passing it the entity pointer */
   link_t *link;
   link = new link_t(entity);

/* Now and the link to the list using generally the */
/* same approach as the C version                   */


}
```

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 entity in the list.

```
void * list_t::start(void)
{

/* Set the current link pointer to the first pointer. */


/* If the list is not empty return the entity pointed */
/* to by the current/first link pointer               */


}
```

Retrieving the next entity 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 .

```
void_t * list_t::get_next(void)
{
   link_t *link;

/* Set link to the next pointer of the current link */


/* If the value of link is NULL  return NULL */


/* Otherwise set current to link and return a pointer */
/* to the entity pointed to by link                   */


}
```

Using the list_t class

Creating a new *list_t*

```
list_t *elist = new list_t;
```

Creating new entities and adding them to the list:

```
/* Read input file consisting of names and id codes
   adding entities to list .. This would not work
   if e_name and e_id were private                */

  while (scanf("%s %d", name, &count) == 2)
  {
     eloc = new e_t;
     strcpy(eloc->e_name, name);
     eloc->e_id = count;
     elist->add((void *)eloc);
  }
```

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 entity in the list.

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

```
/* Now play it back */

  eloc = (e_t *)elist->start();
  while (eloc != NULL)
  {
     printf("%s %d \n", eloc->e_name, eloc->e_id);
     eloc = (e_t *)elist->get_next();
  }
```

This works well *unless* inside the loop there is a call to an inner function that must also 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.

Another example class definition

A class definition creates a type name that can be used in a standalone fashion (like a *typedef)* but the explicit typedef is no longer required. These should replace the comparable structure definitions in *ray.h.*

```
class camera_t
{

public:
          camera_t(){};
          camera_t(FILE *in);

   void   camera_getdir(int x, int y, vec_t *dir);
   void   camera_setpix(int x,int y, drgb_t *pix);
   int    camera_getxdim(void);
   int    camera_getydim(void);
   void   camera_getviewpt(vec_t *view);
   void   camera_dump(FILE *out);
   void   camera_write_image(FILE *out);

private:
   int    cookie;
   char   name[NAME_LEN];
   int    pixel_dim[2];    /* Projection screen size in pix */
   double world_dim[2];    /* Screen size in world coords   */
   vec_t  view_point;      /* Viewpt Loc in world coords    */
   irgb_t *pixmap;         /* Build image here              */
};
```

Example of class *constructors* and function overloading,

Public functions may be called by any entity holding a pointer to a *cam_t*

Private functions or data items are *not accessible* to any entity holding a pointer to a *cam_t*

Function prototypes that appear within a class definition are called *class methods* and are always invoked within the context of an instance of the class. They have *unqualified access to the data structures in the class.* That is, a class method will never refer to *cam->cookie.* It will simply do

```
cookie = CAM_COOKIE;
```

with the understanding that the particular *cookie* is the one that belongs to *this* instance of the class.

Creating an instance of a new class:

In the C language version of the ray tracer a new camera structure was created with a call to *camera_init()*. The *camera_init()* function then proceeded to *malloc()* a *camera_t()* structure and then initialize it.

```
camera_init(stdin, model, 0);
```

Pointers to classes are declared just as pointers to structures are. However, dynamic creation of a new class uses the *new* operator. In the C++ version of the code we will clean things up a bit and not pass the *model* or *attrmax* parameters.

```
camera_t *cam;
model_t  *model;

/* Load and dump camera data */

cam = new camera_t(stdin);
assert(cam != NULL);
```

The *new* operator may be viewed as somewhat similar to *malloc()* but it *creates an instance of the class before invoking the initializer code (which is referred to as the constructor).*

Alternative ways to create an instance of a class

It is also possible to simply declare an instance of new class and invoke its constructor:

```
camera_t camera(stdin);
```

or

```
camera_t camera;
```

The parameters (or lack thereof) determine which constructor is invoked. The *default* constructor which would be invoked in the second case doesn't actually do anything.

If parameters are supplied that don't match any prototype in the class definition, the C++ compiler responds with an appropriate nastygram.

```
  camera_t cam(stdin, model);

cat main.err
main.c: In function `int main(int, char**)':
main.c:14: error: no matching function for call to
`camera_t::camera_t(_IO_FILE*&, model_t*&)'
```

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. Thus when *new camera_t(stdin)* is invoked this constructor is called.

Although it is possible to implement function bodies inside class definitions, unless the functions are trivially short it leads to a big mess. The scope operator :: is saying this function belongs to the *camera_t* class.

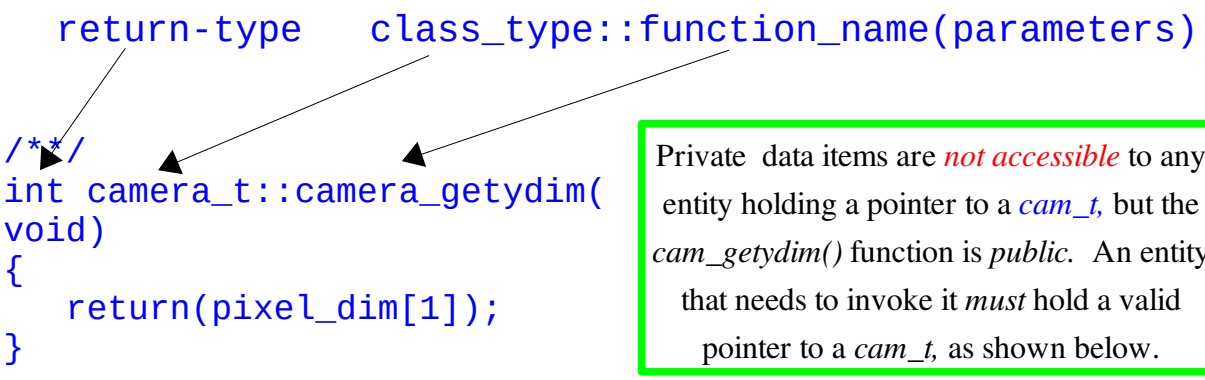
```
camera_t::camera_t(
FILE *in)
{
   char  buf[256];
   int   mask;

   assert (fscanf(in, "%s", name) == 1);

   fscanf(in, "%s", buf);
   assert(buf[0] == '{');

   cookie = CAM_COOKIE;

   camera_parse[0].loc = &pixel_dim;
   camera_parse[1].loc = &world_dim;
   camera_parse[2].loc = &view_point;

   mask = parser(in, camera_parse, NUM_ATTRS, 0);
   assert(mask == 7);

/* Allocate a pixmap to hold the ppm image data */

   pixmap = (irgb_t *)malloc(sizeof(irgb_t) * pixel_dim[0] *
                                              pixel_dim[1]);

}
```

As previously noted, class data members cannot by accessed using the *cam*-> prefix. The major part of a C to C++ conversion is eliminating these pointer based references.

Getter and setter functions

One objective of the O-O approach is to *encapsulate* data within instances of classes and thus protect against the uncontrolled access that is possible in standard C. Because of this, it is common for a class to export a collection of *getters* and *setters* that allow other classes access to required data but in a controlled way. For example the *image_create()* mechanism needs the *pixel_dimension* to determine how many times it should iterate on x and y.

The implementation of class methods must employ the structure:



Private data items are *not accessible* to any entity holding a pointer to a *cam_t,* but the *cam_getydim()* function is *public.* An entity that needs to invoke it *must* hold a valid pointer to a *cam_t,* as shown below.

A C++ program can contain a mix of C++ class methods and traditional C procedures. In fact the *main()* procedure will *always* be a traditional C procedure. A C++ religious warrior may assert that a *real* C++ programmer will never use any other traditional C procedures!

```
static inline void make_row(
model_t *model,
int      y)
{
   int   x;
   int   xdim;
   camera_t *cam = model->cam;

   xdim = cam->camera_getxdim();
   for (x = 0; x < xdim; x++)
   {
```

A C++ setter function

As before the elements belonging to the class are not qualified with a *cam->*

```
/**/
void camera_t::camera_setpix(
int      x,
int      y,
drgb_t   *pix)
{
   int     maprow;
   irgb_t *maploc;

   maprow = pixel_dim[1] - y - 1;
   maploc = pixmap + maprow * pixel_dim[0] + x;

   scale_and_clamp(pix);

   maploc->r = (unsigned char)pix->r;
   maploc->g = (unsigned char)pix->g;
   maploc->b = (unsigned char)pix->b;
}
```

Variables shown in *red* are class member variables that used to be accessed via cam->

An upside down class

Although it is common for data elements to be private and functions be public, it is sometimes useful to turn the model upside down in a hybrid C/C++ program. Here, the internal procedures do not need to support external reference (pointer) holders. It would be feasible to provide a bunch of *getters* that would return the pointer to the *cam* and the *lists*, but a reasonable person might conclude it is not necessary.

```
class model_t
{
public:
            model_t(FILE *);
   void     dump(FILE *);
   camera_t *cam;
   list_t   *mats;
   list_t   *objs;
   list_t   *lgts;

private:
   void model_item_load(FILE *,char    *);
   void model_load(FILE *);

};
```

The *model_t* constructor

As before note that (1) there is no need to *malloc()* the model structure, (2) references to class member data elements (*mats, lgts, objs)* are unqualfied and (3) so are references to class methods *(model_load())*

```
/**/
/* Init model data */

model_t::model_t(
FILE *in)
{

   mats = new list_t;
   assert(mats != NULL);

   lgts = new list_t;
   assert(lgts != NULL);

   objs = new list_t;
   assert(objs != NULL);

   model_load(in);

}
```

The *model_load()* method

This function consumes a single entity name (material, plane, sphere, etc) and then invokes yet another class method to perform the actual loading. Note that even though, this is a C++ method, in a model.cpp module its perfectly legal to use standard C library functions such as *fscanf(), strcmp(), etc.*

I personally find the C++ *iostream* facilty somewhat baroque, but again expect abuse from C++ religious zealots if you employ standard C I/O operations.

```
/* Load model data */

void model_t::model_load(
FILE    *in)
{
   char itemtype[16];
   int  count;

   memset(itemtype, 0, sizeof(itemtype));

/* Here itemtype should be one of "material",    */
/* "light", "plane", "camera"                    */

   count = fscanf(in, "%s", itemtype);
   while (count == 1)
   {
      model_item_load(in, itemtype);
      count = fscanf(in, "%s", itemtype);
   }
}
```

Creating new entities

Each invocation of *model_item_load()* creates a new instance of a *material, plane, sphere, etc.* In the C language version this was readily done using a table of function pointers. In the C++ version a *switch* or *if/else if /else if* mechanism must be used (I think). The problem is that there is no way (that I know of) to pass a variable value to the *new* operator.

```
new plane_t(in, this, 0);
```

If we could replace *plane_t* by a variable which could take on the value *plane_t* or *sphere_t* or *material_t* then it would be straightforward to put a table driven approach back in place.

```
static char *items[] =
{
    "camera",
    "material",
    "plane",
};
#define NUM_ITEMS (sizeof(items) / sizeof(char *))

void model_t::model_item_load(
FILE    *in,
char    *itemtype)
{
   int ndx;
   ndx = table_lookup(items, NUM_ITEMS, itemtype);
   assert(ndx >= 0);
   switch (ndx)
   {
   case 0:
      cam = new camera_t(in);
      break;
   case 1:
      new material_t(in, this, 0);
      break;
   case 2:
      new plane_t(in, this, 0);
      break;
   }
   return;
}
```

Recall that the initializers for *materials and planes* needed a pointer to the *model_t* structure in order to access the *lists*. When such a scenario arises the predifined *this* variable can be used.

Revisiting the *model_t* constructor.

In the code shown below we demonstrate that

- the *this* pointer can be used to provide qualified access to class data members.
- the *this* pointer can be copied to other pointers to provide a more C like representation

Both of these techniques are likely to evoke derision from "real" C++ programmers.

```
model_t::model_t(
FILE *in)
{
   model_t *model = this;

   this->mats = new list_t;
   assert(mats != NULL);

   model->lgts = new list_t;
   assert(lgts != NULL);

   objs = new->list_t;
   assert(objs != NULL);

   model_load(in);

}
```

The *material_t* class

The module *material.c* is a hybrid module involving *both standard C procedures and C++ class methods.* The class methods are for the most part typical *getters* providing access to private values.

```
class material_t
{
public:
   material_t(){};
   material_t(FILE *in, model_t *model, int attrmax);
   void  material_getamb(drgb_t *dest);
   void  material_getdiff(drgb_t *dest);
   void  material_getspec(drgb_t *dest);
   char *material_getname();
   inline void material_item_dump(FILE *out);

private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
};
```

Standard C procedures such as *material_search()* can *not* be declared here

Standard C functions in *material.cpp*

The standard C functions are those in which the entire list of materials must be processed. Since a class method is always invoked within the context of a single instance of the class (i.e. a single material) it is not natural to make them members of the class. These prototypes *can not* appear inside the class definition.

```
/* == material.cpp == */

/* Produce a formatted dump of the material list */

void material_dump(
FILE *out,
list_t *list);

/* Try to locate a material by name */

material_t *material_search(
list_t  *matlist,
char    *name);
```

Initializing a new *material_t*

The only difference between the C and C++ version is the the use of unqualified data names*: cookie, ambient, diffuse*

```
/***/
/* Create a new material description */

material_t::material_t(
FILE        *in,
model_t     *model,
int         attrmax)
{
   char attrname[NAME_LEN];
   int count;
   int mask;

/* Create a new material structure and initialize it */

   cookie = MAT_COOKIE;

/* Read the descriptive name of the material */
/* (dark_red, light_blue, etc.               */

   count = fscanf(in, "%s", name);
   assert(count == 1);

   count = fscanf(in, "%s", attrname);
   assert(*attrname == '{');

   mat_parse[0].loc = &ambient;
   mat_parse[1].loc = &diffuse;
   mat_parse[2].loc = &specular;
   mask = parser(in, mat_parse, NUM_ATTRS, 0);

   assert(mask != 0);
   list->add((void *)this);

}
```

Actual argument here was *this*

Interaction of C and C++ components

The mission of the *material_find* function is to find an instance of a *material_t* that has the proper name. As such it may have to look at *all material_t'*. Therefore it is *not* a *material_t* class method!

```
/**/
/* Try to locate a material by name */
material_t *material_search(
list_t  *list,
char    *name)
{
   material_t *mat = (material_t *)list->start();

   while (mat != NULL)
   {
      if (strcmp(name, mat->material_getname()) == 0)
         return(mat);
      mat = (material_t *)list->get_next();
   }

   return(NULL);
}
```

Since this function is *not* a class method it cannot touch the *private* *mat->name*. But it can ask material_getname to return a pointer to it.

This version of the function attempts to access *mat->name* directly. Since *name* is *private,* it does not succeed.

```
/**/
/* Try to locate a material by name */

material_t *material_search(
list_t  *list,
char    *name)
{
   material_t *mat = (material_t *)list->start();

   while (mat != NULL)
   {

      if (strcmp(name, mat->name) == 0)
         return(mat);
      mat = (material_t *)list->get_next();
   }

   return(NULL);
}
```

Since this function is *not* a class method it cannot touch the *private* *mat->name*.

```
g++ -c -Wall -DAA_SAMPLES=2 -c -g material.cpp  2> material.err
make: [material.o] Error 1 (ignored)
cat material.err
material.cpp: In function ‘material_t* material_find(model_t*,
char*)’:
ray.h:124: error: ‘char material_t::name [16]’ is private
material.cpp:104: error: within this context
```

The *friend* qualifier

Because it is very common for in hybrid C/C++ environments for "helper" functions to need access to private elements of classes. The C++ language provides the *friend* capability.

By using *friend,* one class can give to any functions or class the right to access its private elements directly. Almost *any* use of this facility is frowned upon by O-O purists. Excessive use is frowned upon by everyone.

Nevertheless if we include the following *friend* declaration in *material_t* the error shown on the previous page will go away!

```
class material_t
{

friend material_t *material_search(list_t *, char *);

public:
   material_t(){};
   material_t(FILE *in, model_t *model, int attrmax);
   void  material_getamb(drgb_t *dest);
   void  material_getdiff(drgb_t *dest);
   void  material_getspec(drgb_t *dest);
   char *material_getname();
   inline void material_item_dump(FILE *out);

private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
};
```

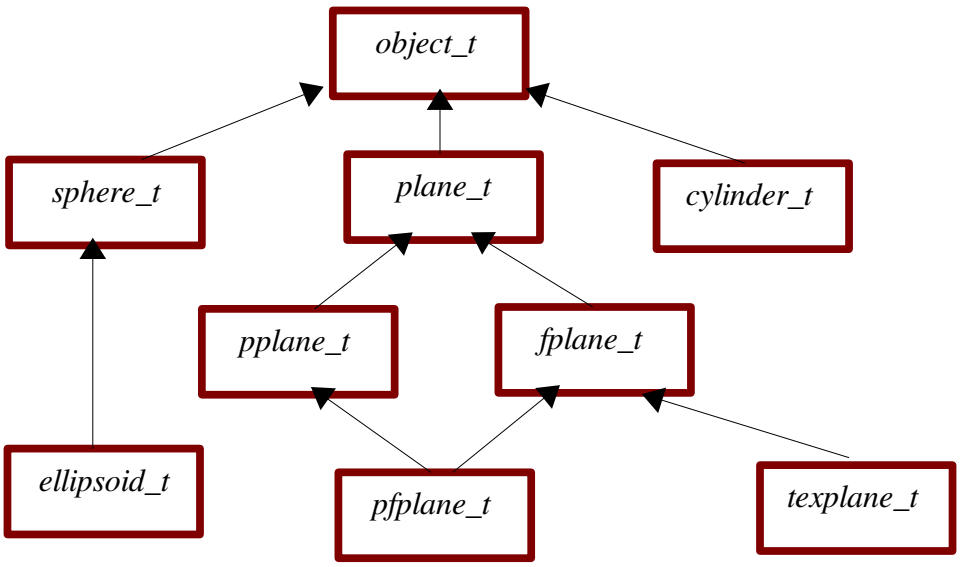
The use of *inheritence*

A C++ class hierarchy is based upon the principle of increased specialization.

- The *base* class carries attributes that are common to all classes and
- virtual functions that may or may not be overridden.

- Attributes that are esoteric to a particular entity *plane/normal* or *sphere/radius* are not defined in the *base class*
- The derived class *inherits* the attributes of classes above it in the class hierarcy

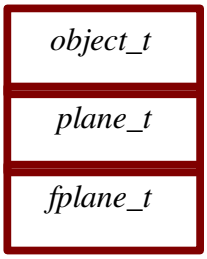
- The specialization can continue over multiple levels
- The amount of "new stuff" required in the implmentation of the derived class can range from trivial (*ellipsoid_t, pfplane_t)* to moderate *(plane_t, sphere_t)* to fairly complex (*texplane_t)*



Derived object creation

When an instance of a new class such as *fplane_t* is created, the *new* facility creates an instance of each class as it proceeds up the hierarchy until it reaches the base class and "glues" them together. The constructors *are always invoked top down*: *object_t() then plane_t() then fplane_t().*

Thus, our attribute loading strategy continues to work as before and model description files that worked in the C version *should work* in the C++ version.



The workings of the binding mechanism are not particulary obvious.

- When a class method at any level is invoked *the "this" pointer* always points to an instance of the level in the hierarchy to which the member function belongs. This guarantees that *plane_t* methods see *plane_t* instances.

- Pointers may also be forcibly *upcast* and *downcast* but this should be a last resort. Downcasting is particularly hazardous because what is *below* an *object_t* varies!!

The *object_t* class

```
class object_t
{
public:

   object_t(){};
   object_t(FILE *in, model_t *model);

   virtual ~object_t(){};

   inline void object_t::object_item_dump(FILE *out);

   virtual  double  hits(vec_t *base, vec_t *dir)
                       {return(-1.0);};
   virtual  void    dumper(FILE *);


   virtual    void    getamb(drgb_t *);
   virtual    void    getdiff(drgb_t *);
   virtual    void    getspec(drgb_t *);

protected:
   vec_t   hitloc;    /* Last hit point              */
   vec_t   hitnorm;   /* Normal at hit point         */

   int     cookie;
   char    objtype[NAME_LEN];  /* plane, sphere, ...       */

/* Surface reflectivity data */
   material_t *mat;

private:
   char   objname[NAME_LEN];  /* left_wall, center_sphere */

};
```

For mysterious reasons, g++ recently began to strongly dislike classes with virtual functions that do not provide a virtual destructor.

inline can still be used

Default behavior for *hits* is a *miss*. The *dumper* function is in object.cpp

Default behaviors for these guys are to punt the problem to *material_t*.

Protected attributes are accessible to derived classes. .

The *object_t* constructor

This function mirrors the operation of the old *object_init()* function. Data elements of the class are now referenced without the *obj->* qualifier.

```
/**/
/* Create a new object description */

object_t::object_t(
FILE       *in,
model_t    *model)
{
   char buf[NAME_LEN];
   int count;

/* Read the descriptive name of the object */
/* left_wall, center_sphere, etc.          */

   count = fscanf(in, "%s", objname);
   assert(count == 1);

/* Consume the delimiter */

   count = fscanf(in, "%s", buf);
   assert(*buf == '{');

   cookie = OBJ_COOKIE;
    :
```

The default reflectivity getters

These functions simply punt the material reflectivity request to the proper handler in the material class.

Excessive punting is often associated with *suboptimal O-O* design. An alternative design that could avoid this punt is to have the *real* objects (plane_t, etc) be derived from *both* material_t and *object_t*. This technique is called *multiple inheritance*.

Punting can also be avoided by copious use of *friend* and *public*, but the downside of that has already been addressed.

```
void object_t::getamb(drgb_t *amb)
{
   mat->material_getamb(amb);
}

void object_t::getdiff(drgb_t *diff)
{

   etc....
}
```

These functions can be written more obscurely as:

```
void object_t::getamb(drgb_t *amb)
{
   material_t *mat;
   mat = this->mat;
   mat->material_getamb(amb);
}
```

Processing the complete object collection

Because an *object_t* member function always runs in the context of a single instance of a class, when it is necessary to process *all* instances of a specific class it must be done:

- in the context of a member function of *another class* (e.g. the model_t) or
- by a "standalone" C language function that is *not a member of any class*

The function that drives the dumping of the object list is implemented here as "case 2". It is a standard C function that is not a member of any class.

```
/**/
void object_dump(
FILE *out,
model_t *model)
{
   link_t  *link;

   object_t *obj;
   link = model->objs->head;

   while (link != NULL)
   {
      obj = (object_t *)link->item;
      obj->dumper(out);
      fprintf(out, "\n");
      link = link->next;
   }
}
```

Dumping the attributes of a single object

While it is *not appropriate* for an *object_t* member function to driver the dumping process, it is appropriate for a member function to deal with dumping a single instance of the class. The *object_item_dump()* function is a class member.

```
inline void object_t::dumper(
FILE     *out)
{

   assert(cookie == OBJ_COOKIE);
   fprintf(out, "%-12s %s \n", objtype, objname);
   fprintf(out, "%-12s %s \n", "material",
                    this->mat->material_getname());

}
```

The *object_item_dump()* function dumps the objects name and material name and then invokes the most specific *dumper* in the hierarchy. The dumping heirarchy is designed so that

- Each level in the hierarchy dumps its own attributes
- The attributes appear in *stderr* in top down order

In its class definition the *fplane_t* declares a *hits* function and a *dumper* function. Thus these virtual functions will *override* the *hits* and *dumper* provided by both the *object_t* and the *plane_t*.

```
class fplane_t: virtual public plane_t
{
public:
   fplane_t();
   fplane_t(FILE *, model_t *, int);

   virtual double  hits(vec_t *base, vec_t *dir);
   virtual void    dumper(FILE *);
```

The *fplane_t dumper()* function.

The *fplane_t dumper* uses the scope operator to invoke the *plane_t's* dumper. This *always* works because each entity *knows* what lies above it in the hierarchy. Traversing the hierarchy the other direction is possible but much nastier and requires specific *downcasting*.

After *plane_t::dumper()* dumps the *normal* and *point* attributes, it will return to the *fplane_t::dumper()* which will dump the *dims* and the *xdir*.

```
/***/
void fplane_t::dumper(
FILE *out)
{
   plane_t::dumper(out);

   fprintf(out,  "%-12s %5.1lf %5.1lf\n", "dims",
                              dims[0], dims[1]);
   fprintf(out,  "%-12s %5.1lf %5.1lf %5.1lf\n", "xdir",
                           xdir.x, xdir.y, xdir.z);
}
```

This scheme is indefininitely continuable. The *pfplane_t* has no attributes of its own but exists to allow the *getamb(), getdiff()* functions of the *pplane_t* to be mated with the *hits* function of the *fplane_t*.

```
/**/
void pfplane_t::dumper(
FILE *out)
{
   pplane_t::dumper(out);
   fplane_t::dumper(out);
}
```

The *find_closest_object()* function.

Since this function must also *process the entire object list* it *must not* be a member function of the *object_t* class. As with the *dumper* it is a standalone C function.

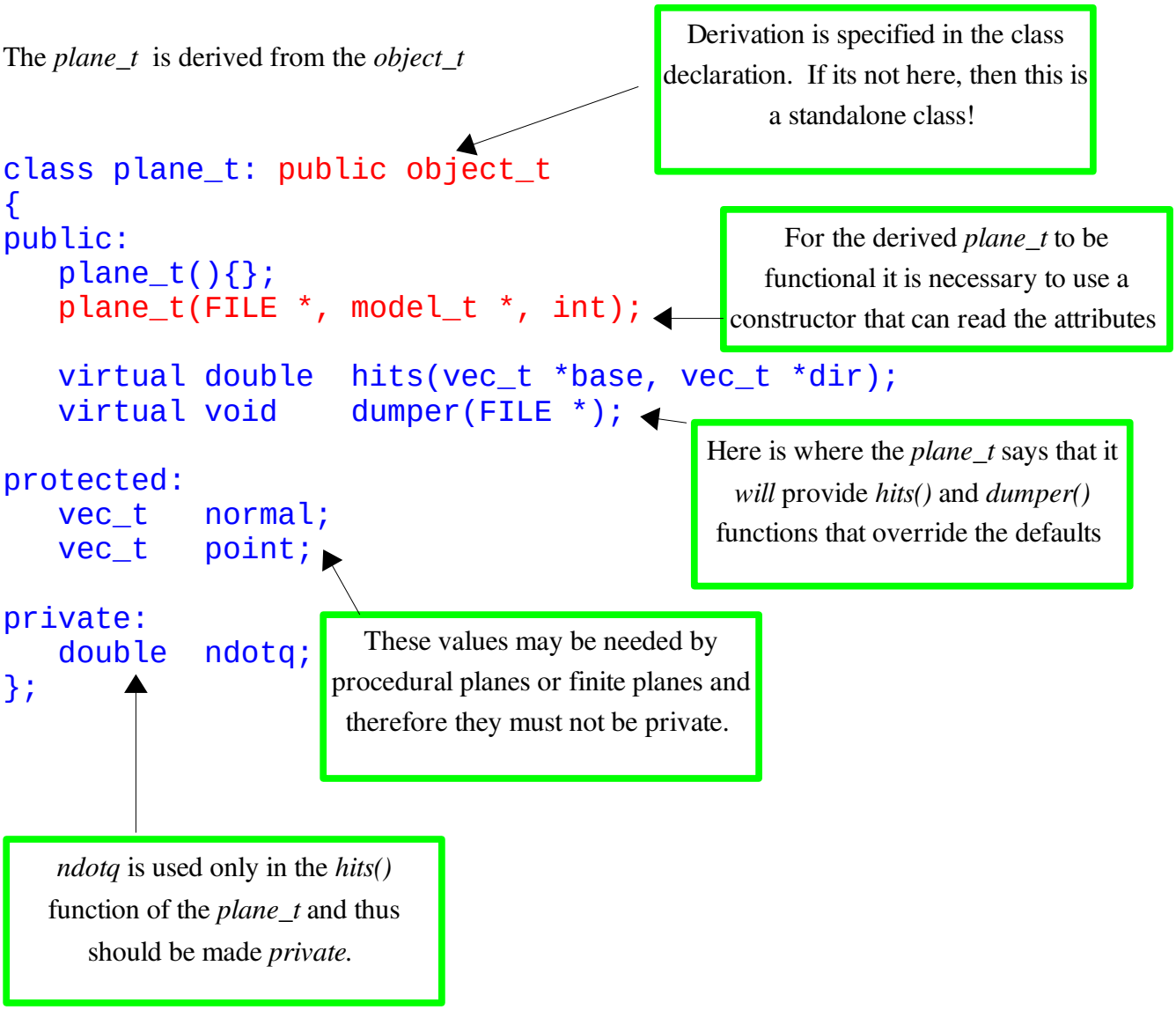
```
object_t *find_closest_object(
list_t   *list,         /* Object list       */
vec_t    *base,         /* Base of ray       */
vec_t    *dir,          /* direction of ray  */
object_t *last_hit,     /* object last hit   */
double   *retdist)      /* dist to hit point */
{
   double    dist;
   object_t  *closest = NULL;
   double    mindist;
   object_t  *obj =  (object_t *)list->start();
   while (obj != NULL)
   {
      if (obj == last_hit)
      {
         obj = (object_t *)list->get_next();
         continue;
      }
      dist = obj->hits(base, dir);

#ifdef DBG_FIND
      fprintf(stderr, "FND %12s: %5.1lf - \n ",
                                    obj->objname, dist);
#endif

   /* see if this is first obj it or closest hit */
   /* and in that case update closest            */

      if ((dist >= 0) && ((closest == NULL) || (dist <
mindist)))
      {
          mindist = dist;
          closest = obj;
      }
      obj = (object_t *)list->get_next();
   }
   *retdist = mindist;
   return(closest);
```

The *hits* function at the lowest (most specific) point in the hierarchy is the one that is actually used.

```
}
```

Creating a derived class

The *plane_t* is derived from the *object_t*



A constructor for a derived class

The *plane_t* is derived from the *object_t*

```
/**/
/* Create a new plane object and initialize it */

plane_t::plane_t(
FILE    *in,
model_t *model,
int      attrmax): object_t(in, model)
{
   int  mask;

   strcpy(objtype, "plane");

/* The parser is fairly generic but the address of where to */
/* put the data must be updated for each new object         */

   plane_parse[0].loc = &point;
   plane_parse[1].loc = &normal;
   mask = parser(in, plane_parse, NUM_ATTRS, attrmax);
   assert(mask == 3);

   vec_unit(&normal, &normal);
   vec_copy(&normal, &hitnorm);
   ndotq = vec_dot(&point, &normal);
}
```

This is a critically important element of inheritance. It specifies *which* constructor of the parent class should be used

A *vector* class

We will use a vector class to illustrate some aspects of C++ programming and in some lab exercises. Use of the vector class and C++ I/O factilities in your ray tracer is *entirely optional.* We put the class definition in file *vec.h*

```
#include <iostream>
using namespace std;

class vec_t
{

public:
   vec_t();
   vec_t(double, double, double);

private:
   double x;
   double y;
   double z;
};
```

These two lines replace *#include <stdio.h>* when using C++ I/O

Default constructor will set vector to (0,0,0). Other one will set it to values provided

The *vec_t* constructors

As with the ray tracing routines, we put the implementations of the class methods in *vec.cpp*.

```
#include "vec.h"

vec_t::vec_t()
{
   cerr << "default constructor" << endl;
   x = y = z = 0.0;
}

vec_t::vec_t(double xi, double yi, double zi)
{
   cerr << "3 parm constuctor" << endl;
   x = xi;
   y = yi;
   z = zi;
}
```

In C++ its sometimes difficult to figure out which overloaded function is called when (or if at all).

The *cerr* function may be passed any number of items of types that it "knows about" separated by << and they will given a default format and sent to the standard error. *cout* can be used to send output to the *stdout.* The counter part of these functions is:

```
   double d;
   cin >> d;
```

It can be used to read a value into a type that it "knows about".

The << and >> operators are used for bit shifting in standard C and are *overloaded* by the *ostream* and *istream* classes. We will see later on how to make << and >> know about the *vec_t* class itself.

Vector operations

In the C++ version of the vector class, our old design can be made to work *if* we make *vec_sum(), vec_diff(), and so on friends of vec_t.* In that way they can remain *standard C functions,* but have access to the private values *x, y, z* of any instance of a vector class to which they hold a reference.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_diff(vec_t *, vec_t *, vec_t *);
     :
};
```

Note that the *friend* declaration is mandatory for this function to be able to access the private elements of the *vec_t.* But when declared *friend* the implementation doesn't change.

```
inline void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
   v3->x = v1->x + v2->x;
   v3->y = v1->y + v2->y;
   v3->z = v1->z + v2->z;
}
```

Reference parameters

Standard C supports only *pass by value.* With pass by value, when a structure name is used as a formal parameter/actual argument, the entire structure must be *copied onto the stack*. This becomes increasingly undesirable as structures increase in size. Therefore the standard C solution is to make the formal parameter and actual argument be *pointers to the structure type*. Doing this provides two benefits:

- Only a single word (the pointer) must be pushed onto the stack
- The called function is able to modify elements of the structure.
- Unlike a pointer, it is not possible to do arithmetic on a reference. Thus some of the more error prone aspects of C programming may be avoided.

C++ supports an additional parameter passing technique knowns as *pass by reference.* Use of pass by reference is signified by the use of *& instead of* * in the formal parameters of the function prototype. In C++ *all* function prototypes must have been defined at the point of invocation so there is no ambiguity regarding what technique is in use.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
   :
};
```

Implementing overloaded functions

The C++ compiler allows even standard C functions to be overloaded. We must provide a function body for every distinct parameterization we wish to allow.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
   :
};
```

In the body of the implementation the reference parameter is *treated as a structure*, and not a pointer to a structure.

```
inline void vec_sum(vec_t &v1, vec_t &v2, vec_t &v3)
{
   v3.x = v1.x + v2.x;
   v3.y = v1.y + v2.y;
   v3.z = v1.z + v2.z;
}

inline void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
   v3->x = v1->x + v2->x;
   v3->y = v1->y + v2->y;
   v3->z = v1->z + v2->z;
}
```

Invoking overloaded *functions*

On the previous page we created two distinct versions of *vec_sum()*. Fortunately, they produce the same answer here, but there is no requirement that they do so. On this page we look at the problems of how to invoke them and which one gets invoked.

To use pass by reference *the actual argument must be an instance of the class* not a pointer to an instance of the class.

The instance of an overloaded function that is used is the one whose formal parameters (best) match the actual arguments. The matching is straightforward when instances of or pointers to classes are the parameters.

```
int main()
{
   vec_t v1(1.0, 2.0, 3.0);
   vec_t v2(4.0, 5.0, 6.0);
   vec_t v3;

   vec_sum(&v1, &v2, &v3);
   v3.put();

   vec_sum(v1, v2, v3);
   v3.put();

}
```

The parameterized constructor is called here.

The default constructor is called here.

The pointer based implementation is invoked here.

The reference based implementation is invoked here.

A broken function:

We could attempt to add a third implementation of *vec_sum()* which passes parameters by value by copying them onto the stack.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);
   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t , vec_t , vec_t );
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
```

To call this verson we would also have to use:

```
   vec_sum(v1, v2, v3);
```

This would produce the following compile time error. Because the calling sequence shown IS the correct way to invoke either of the bottom two prototypes the compiler can't distinguish which one you want.

```
main.cpp: In function ‘int main()’:
main.cpp:24: error: call of overloaded ‘vec_sum(vec_t&, vec_t&,
vec_t&)’ is ambiguous
vec.h:68: note: candidates are: void vec_sum(vec_t&, vec_t&, vec_t&)
vec.h:75: note:                 void vec_sum(vec_t, vec_t, vec_t)
acad/cs102/examples/vec ==>
```

The compile time problem could be "fixed" by removing the implementation that used reference parameters and the resulting program would compile fine but *just would not work.*

Exercise: Why not?

A class based approach

We can also implement yet another instance of *vec_sum()* which is a true member function of the class.

At first glance this one looks a bit odd because *v1* seems to have disappeared!! This occurs because *class methods are always invoked in the context of an instance of the class.* In this case the instance will be *v1*.

The *const* modifier makes it *not* possible for *vec_sum()* to modify *v2*. Obviously *v3* should *not be const.*

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);

   void vec_sum(const vec_t &v2, vec_t &v3);

   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);

};
```

The implementation also looks somewhat asymmetric with *v2* and *v3* being explicitly accessed in contrast to the implicit access to *v1*.

```
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
  v3.x = x + v2.x;
  v3.y = y + v2.y;
  v3.z = z + v2.z;
}
```

The asymmetry is also apparent in the invocation.

```
   v1.vec_sum(v2, v3);
   v3.put();
```

A collection of possible implementations for vec_sum

C++ makes it possible (though not necessarily desirable) to provide implementations that match virtually any parameterization:

```
void vec_sum(const vec_t &v2, vec_t &v3);
void vec_sum(const vec_t *v2, vec_t *v3);
void vec_sum(const vec_t *v2, vec_t &v3);
void vec_sum(const vec_t  v2, vec_t *v3);
```

Exercises: Identify any possible *usuable* prototypes that we may have missed. Identify a prototype different from any of those above that would cause the compile time problem we saw previously. Identify a prototype that would compile correctly but would not work. Each prototype must have a distinct implementation depending upon whether reference or value pointers are used .

```
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
  v3.x = x + v2.x;
  v3.y = y + v2.y;
  v3.z = z + v2.z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t *v3)
{
  v3->x = x + v2->x;
  v3->y = y + v2->y;
  v3->z = z + v2->z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t &v3)
{
  v3.x = x + v2->x;
  v3.y = y + v2->y;
  v3.z = z + v2->z;
}

void vec_t::vec_sum(const vec_t v2, vec_t *v3)
{
  v3->x = x + v2.x;
  v3->y = y + v2.y;
  v3->z = z + v2.z;
}
```

This is the *only* place an entire vector must be copied onto the stack.

Implementations *returning* instances of *vec_t*

It is possible in *both* C and C++ to define instances of *vec_sum()* that *return* the answer. Unless you have a *real good* reason for doing so, this is generally a bad idea because in both languages it causes a copy on to and copy off of the stack operation. (In the C language function overloading (the use of multiple implementations of the same function name is also illegal).

Here the prototype is declared to *return* an instance of *vec_t*.

```
vec_t vec_sum(const vec_t &v2);
```

The implementation requires a temporary variable in which the sum is computed. It is important to remember to *return(tmp);*

```
vec_t vec_t::vec_sum(const vec_t &v2)
{
  vec_t tmp;
  tmp.x = x + v2.x;
  tmp.y = y + v2.y;
  tmp.x = z + v2.z;
  return(tmp);
}
```

To compute the result v3 = v2 + v1 the following C++ code can be used. The "default" structure assignment mechanism takes care of copying the result off of the stack and into v3. This overhead is realitively minor for a vector but would be seriously bad for a structure containing a large array.

```
 v3 = v2.vec_sum(v1);
```

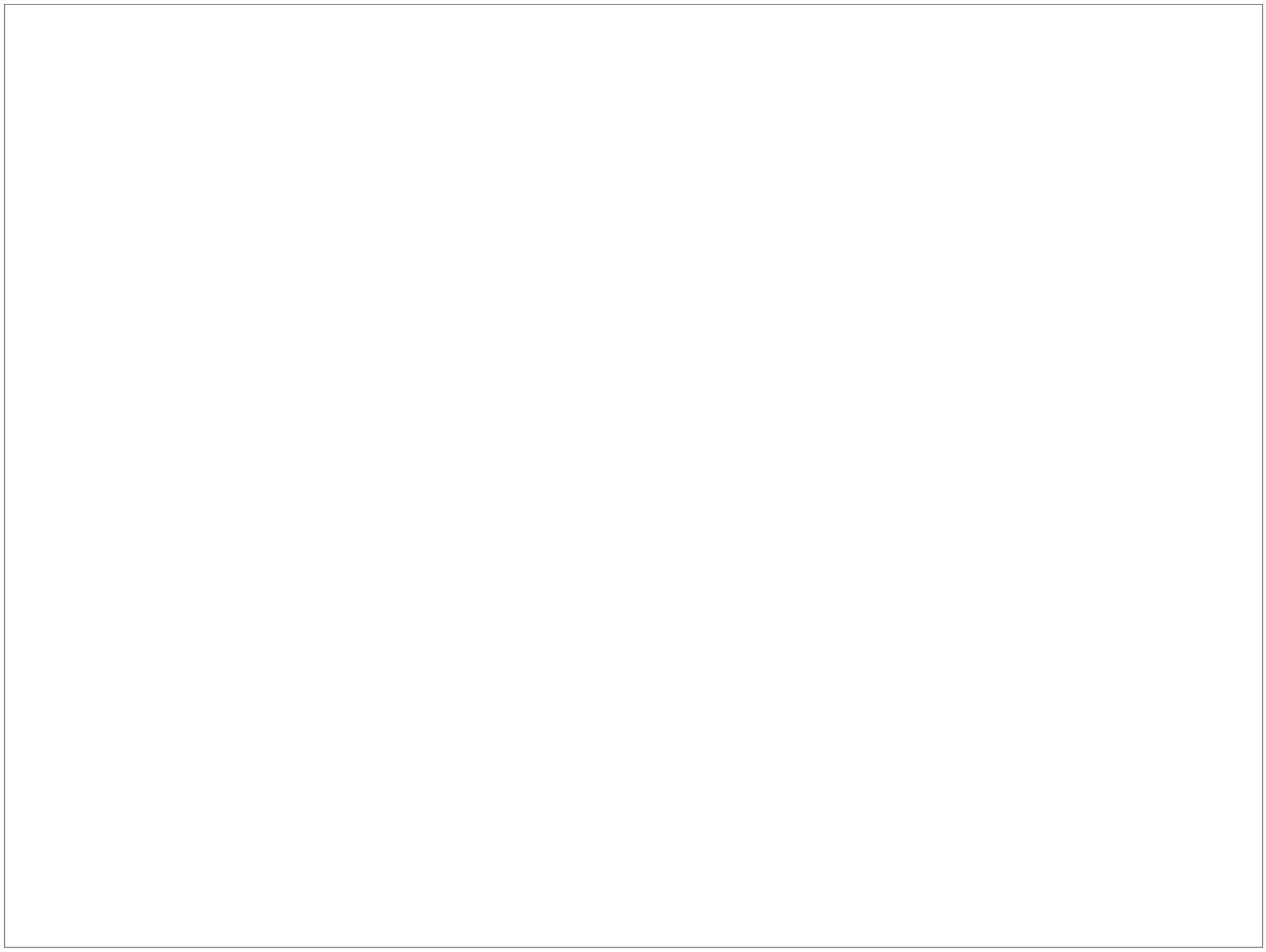
Diffuse illumination

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.

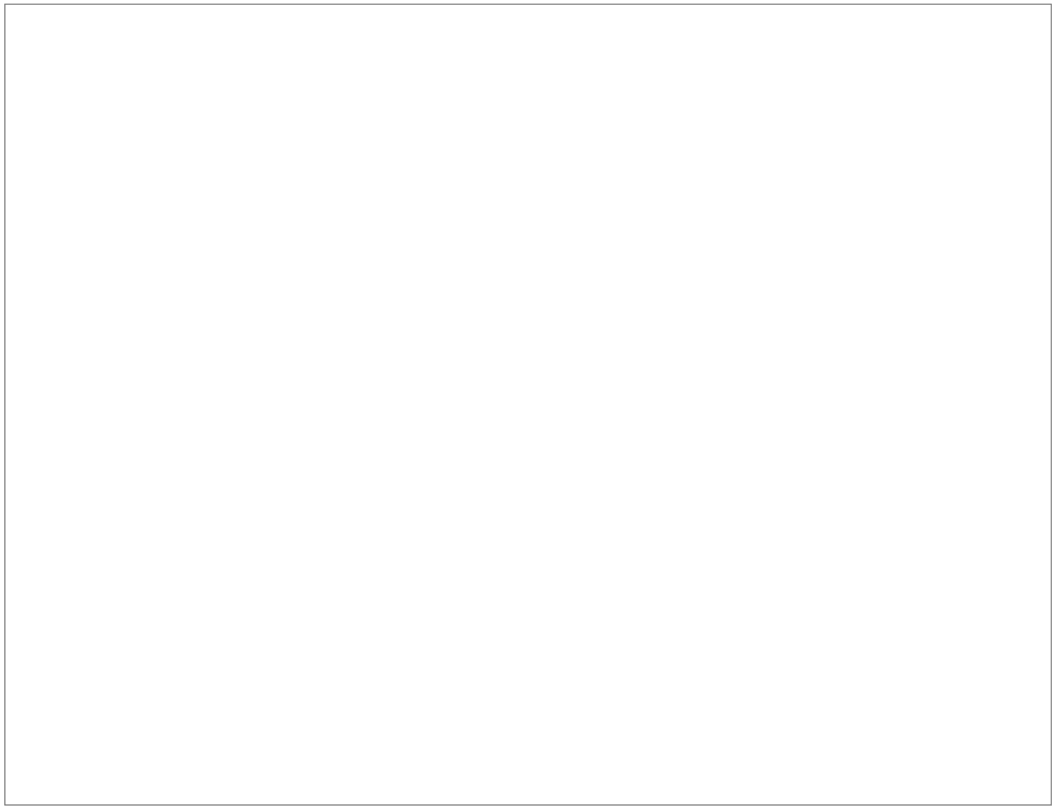
To model diffuse reflectivity requires the presence of one or more light sources. The first type of light source that we will consider is the point light source. This idealized light source emits light uniformly in all directions. It may be located on either side of the virtual screen but is, itself,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

In addition to providing additional realism, diffuse illumination also provides (almost) realistic shadows. The "almost" qualifier is necessary because

- real light sources are not points and thus provide "soft" rather than "hard" shadow boundaries
- real objects that reflect light diffusely also illuminate other objects.



Implementing diffuse illumination



A list structure for holding instances of the class *light_t* is already present in the *model_t* class.

```
class model_t
{
public:
           model_t(FILE *);
   void    dump(FILE *);
   camera_t   *cam;
   list_t  *mats;
   list_t  *objs;
   list_t  *lgts;
private:
   inline void model_item_load(FILE *,char    *);
   inline void model_load(FILE *);

};
```

The *light_t* class

```
class light_t
{
public:
   light_t(){};
   light_t(FILE *in, model_t *model, int attrmax);

   virtual ~light_t(){};


   virtual void    getemiss(drgb_t *)
   virtual int     vischeck(vec_t  *);
   virtual void    illuminate(model_t *, object_t *, drgb_t *);
   virtual void    dumper(FILE *);


protected:
   vec_t   location;
   drgb_t  emissivity;
   char    name[NAME_LEN];

private:
   int     cookie;
};
```

Invoked for *every hit point and every light.*

Values specified in the model description file

The *light_t* class model description

The description of a light is consistent with the structure of the description of visible object. Each light has a descriptive-name, a location and an emissivity. Emissivity components of white lights are necessarily equal.

```
light red-ceiling
{
   location 4 8 -2
   emissivity 5 1 1
}

light blue-floor
{
   location 2 0 0
   emissivity 1 1 5
}
```

Modifications to *model.cpp*

Two modifications are needed in *model.cpp*

- The *model_item_load()* method must create new instances of *light_t new light_t(in, this, 0);*
- The *dump()* method must invoke the C *function light_dump(out, this)* to dump the light list.

The *light_t* constructor

The *light_t* constructor has the same interface as *object_t* constructors.

```
/**/
/* Create a new light description */
light_t::light_t(
FILE          *in,
model_t       *model,
int            attrmax)
{

- parse the required attribute values
- put the light_t class instance into the light list

}
```

The *light* list dumper.

```
/**/
/* Produce a formatted dump of the light list */

void light_dump(
FILE *out,
model_t *model)
{
   list_t  *list;
   light_t *light;

   list = model->lgts;
   light = (light_t *)list->start();

   for each light_t in the light list
       invoke class method light_item_dump();

}
```

Adding diffuse illumination to the *raytrace()* function

```
/* Hit object in scene, compute ambient and diffuse */
/* intensity of reflected light.                    */

   closest->getamb(&thisray);  // add ambient reflectivity
```

The *add_diffuse()* function drives the diffuse lighting process. It must *add* the diffuse contribution of each light that illuminates the hitpoint to *thisray*. *A common error is to compute the diffuse contribution of a light and then store it in the pixel instead of adding* .

```
   add_diffuse(model, closest, &thisray);
```

The use of the local *this_ray* variable is made necessary by anti-aliasing (which will be described later). The *scale before add* is ABSOLUTELY NECESSARY for anti-aliasing to work properly!

```
/* Scale intensity by distance of ray travel       */
/* then add the scaled value to the values pointed */
/* to by pix                                       */

   pix_scale(1 / total_dist, &thisray, &thisray);
   pix_sum(&thisray, pix, pix);
```

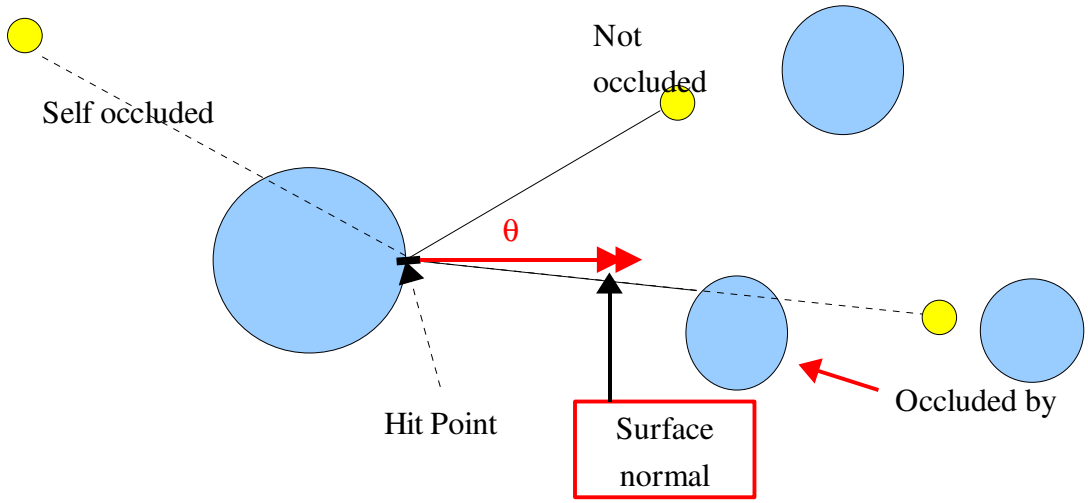
The *add_diffuse()* function

This is a standard C function. As shown on the previous page it is called *every time* an object is hit by a ray. Its mission is to process *the entire light list.* For each light the illumination it provides must be added to **pixel*.

```
void     add_diffuse(
model_t  *model,    /* object list                    */
object_t *hitobj,   /* object that was hit by the ray */
drgb_t   *pixel)    /* where to add intensity         */
{
   light_t *light;
   list_t  *list;

   list = model->lgts;
   light = (light_t *)list->start();

   while (light != NULL)
   {
      light->illuminate(model, hitobj, pixel);
      light = (light_t *)list->get_next();
   }
   return;
}
```

Computing illumination

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. Not all lights illuminate each hit point. 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*.

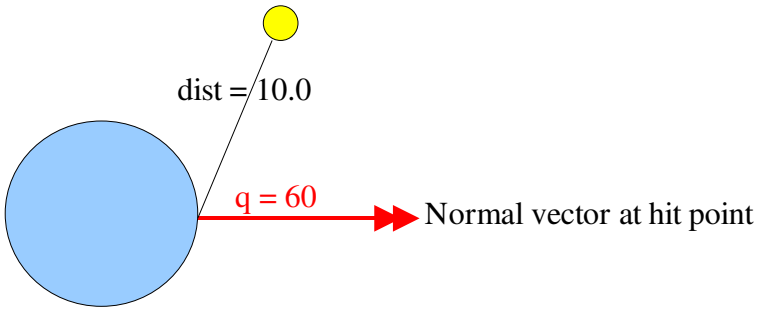
To see if a light is occluded by another object, it is necessary to see if a ray fired from the *hitpoint* to the *light* hits another object before it reaches the light. This can be accomplished via a call to *find_closest_object()* The light is occluded if and only if

(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 (details)

```
void light_t::illuminate(
model_t     *model,
object_t    *hitobj,  /* The object hit by the ray   */
drgb_t      *pixel)   /* add illumination here       */
{

   vec_t    dir;    // unit direction to light from hitpt
   vec_t    revdir; // unit direction from light to hitpt
   object_t *obj;   // closest object in dir to light
   double   close;  // dist to closest object in dir to light
   double   cos;    // of angle between normal and dir to light
   double   dist;   // to the light from hitpoint
   drgb_t   diffuse = {0.0, 0.0, 0.0};

/* Compute the distance from the hit to the light and a unit */
/* vector in the direction of the light from hitpt           */
   ..........
/* Test the object for self-occlusion and return if occluded */
   ..........
/* Ask find_closest_object() if a ray fired toward the light */
/* hits a "regular" object.  Pass "hitobj" as the "lasthit"  */
/* parameter so it won't be hit again at distance 0.         */
   ..........
/* If an object is hit and the distance to the hit is   */
/* closer to the hitpoint than the light is, return     */
   ............
/* Arriving at this point means the light does illuminate */
/* object. Ask hitobj->getdiff() to return  diffuse       */
/* reflectivity                                           */
   ............

/* Multiply componentwise the diffuse reflectivity by     */
/* the emissivity of the light.                           */
   ............

/* Scale the resulting diffuse reflectivy by cos/dist     */
   ............
/* Add scaled value to *pixel.                            */
```

An example illumination computation



Suppose the diffuse reflectivity of the object is: *diffuse = {1, 2, 10}*

Suppose the emissivity of the light is *emissivity = {8, 8, 2}*

Then the componentwise product is *diffuse = {8, 16, 20}.*

Since *cos(q)* = 0.5. The scale factor is *0.5 / 10 = 1 / 20.0.*

The scaled value of *diffuse* = {0.4, 0.8, 1.0}

is then added to the current value of **pixel.*

Common *fatal* problems

- failing to use unit vectors when attempting to compute *cos(q)*
- inadvertently modifying the *emissivity* of the light or the *diffuse reflectivity* of the object instead of modifying the *local variable diffuse* shown on the previous page.

Accessing *hitloc* and *hitnorm*

These are protected members of the *object_t* class. Therefore you must either:

- Declare the *light_t* class to be a *friend* of the *object_t* class. This may produce "circular definition" complications. The easiest way to avoid these is

```
class light_t;

class object_t
{
  friend class light_t;
```

- Add *getter* functions to the object class that will fill in the *hitloc* and *hitnorm* values.

- Make only the *light_t::illuminate()* function a friend. This complicates the circular definition problem.

Operator overloading in C++

Most operators can be overloaded. The ones that cannot are

`.` `.*` `::` `? :` and `sizeof`

Operator overloading is actually part of the function overloading mechanism. To overload an operator you simply provide a *function of the appropriate name*. For example,

Operator	Function name
`+`	`operator+`
`-`	`operator-`
`*`	`operator*`
`<=`	`operator<=`
`->`	`operator->`

Operator overloading is restricted to existing operators. Thus it is *not legal* to try to overload **

`**`	`operator**`

The *operator* functions work "almost" *exactly like "regular"* functions. They can even be invoked using their *operator*+ or *operator*- names. Keeping this fact in mind will remove much of the mystery from how they work and how they must be implemented. The "almost" qualifier above reflects necessity to remember that almost all C operators take either *one or two operands.* Thus an operator function has *at most* two parameters.

The C addition operator takes two operands: *a + b*

Therefore the *operator*+ function will have two parameters: the first will represent the *left side operand* and the second the *right side* operand.

The C logical not operator takes one operand: *!value*

Therefore the *operator!* function will have *one* parameter and it will represent the *right side* operand.

Operator overloading *must preserve the "aryness"* (unary or binary) nature of the operator. Thus, the ! operator could be overloaded to compute the length of a *single vector but could not be used to compute a dot product.* The operators &, *, +, - have both binary and unary versions that may be overloaded separately.

It is not possible to change the precedence or associativity of an overloaded operator.

Most operator functions return a value that replaces the operator and its operand(s) in an expression. However, that is not mandatory. For example a *side effect* operator such as ++ may not need to return a value.

Like "regular functions" all operator functions may be defined as "regular C" *friend* functions of the class(es) to which their operands belong.

In *some* but not all cases they may be defined as *class member* functions instead.

Overloads as *friend* functions

We can write a version of the *vec_sum()* function that uses the + operator.

```
class vec_t
{
public:
   vec_t();
   vec_t(double, double, double);

   friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
   friend inline void vec_sum(vec_t &, vec_t &, vec_t &);

   friend inline vec_t operator+(const vec_t &, const vec_t &);
};
```

The body of the function is written in exactly the same way that a *vec_sum()* function which returned the sum would be written.

```
vec_t inline operator+(const vec_t &v1, const vec_t &v2)
{
  vec_t v3;
  v3.x = v1.x + v2.x;
  v3.y = v1.y + v2.y;
  v3.z = v1.z + v2.z;
  return(v3);
}
```

The function may then be invoked either by standard use of the + operator or by using its *operator+()* name.

```
   vec_t v1(1.0, 2.0, 3.0);
   vec_t v2(4.0, 5.0, 6.0);
   vec_t *vp = &v2;
   vec_t v3;

   v3 = v1 + v2;
   v3 = operator+(v1, v2);
```

If we try to invoke operator + as:

```
v3 = v1 + vp;
```

```
main.cpp:21: error: no match for ‘operator+’ in ‘v1 + vp’
vec.h:98: note: candidates are: vec_t operator+(const vec_t&, const
vec_t&)
```

This occurs because *vp* is a pointer. We can fix this in one of two ways.

1 – Write another instance of the operator+() function in which the second parameter is a *vec_t* *

2 – Just invoke the function as:

```
v3 = v1 + *vp;
```

Overloads as *member* functions

Overloaded operators can be *member functions* instead of friend functions *if the left side operand is an instance of the class or if the right side operand of a unary operator such as ! is a class member.*

```
class vec_t
{
   :
   vec_t operator+(vec_t &rhs);
   vec_t operator+(vec_t *rhs);
   vec_t operator!(void);
   :
};

vec_t vec_t::operator+(vec_t &rhs)
{
   vec_t tmp;

   tmp.x = x + rhs.x;
   tmp.y = y + rhs.y;
   tmp.z = z + rhs.z;

   return(tmp);
}

vec_t vec_t::operator+(vec_t *rhs)
{
   vec_t tmp;

   tmp.x = x + rhs->x;
   tmp.y = y + rhs->y;
   tmp.z = z + rhs->z;

   return(tmp);
}
```

If a binary operator is a class method the left side operand becomes the "this" instance in whose context the function is invoked..

A unary operator which is a class method needs no parameters at all.

This demonstrates that it is possible to create a second implementation that takes a pointer to a vector instead of an instance of a vector. The invocation below actually works as intended, *but it is really ugly when you think about it! What sense does it make to add a vector and a pointer*.

```
v3 = v1 + vp;
```

Scaling a vector

In the scaling operation we want to multiply the components of a vector by a double precision value. Since the left side operand is conventionally the double precision value *and not a vector* the *friend function method must be used*. Note that it is possible to *write the body of the friend function* within the class definition. Also note that for this weeks lab assignment, this approach is *not allowed.*

```
friend vec_t operator*(double val, const vec_t &rhs)
{
   vec_t tmp;
   tmp.x = val * rhs.x;
   tmp.y = val * rhs.y;
   tmp.z = val * rhs.z;
   return(tmp);
}
```

We can also write a function that computes the component-wise product:

```
friend vec_t operator*(const vec_t &lhs, const vec_t &rhs)
{
   vec_t tmp;
   tmp.x = lhs.x * rhs.x;
   tmp.y = lhs.y * rhs.y;
   tmp.z = lhs.z * rhs.z;
   return(tmp);
}
```

The correct implementation will be chosen by the compiler depending upon the operands.

```
v3 = 5.0 * v2;
v3 = v1 * v2;
```

The following, however, will generate a compile time error:

```
v3 = v2 * 5.0;
```

```
main.cpp: In function ‘int main()’:
main.cpp:38: error: no match for ‘operator*’ in ‘v1 * 5.0e+0’
vec.h:54: note: candidates are: vec_t operator*(double, const vec_t&)
vec.h:63: note:                 vec_t operator*(const vec_t&, const
vec_t&)
```

Further overloading of << and >>

We saw in last week's lab that the overloaded operators << and >> could be used to print and to read numeric and character string values to the *stdout* and *stderr* and from the *stdin.* True to form, they can be further overloaded to print and read a complete *vec_t.* We want to be able to do something like

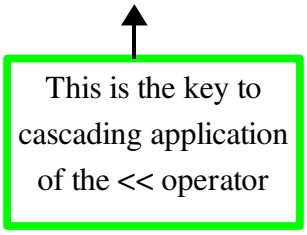
```
cout << v1;
```

where v1 is a vector. Since the left hand side is not a *vec_t,* we must use the *friend* function form. So in *vec.h* we include in the *vec_t* class definition:

```
   friend ostream &operator<<(ostream &out, const vec_t &pvec);
```

We provide the implementation in either *vec.cpp* or inline in the class definition.

```
ostream &operator<<(ostream &out, const vec_t &pvec)
{
   out << pvec.x << ", " << pvec.y << ", " << pvec.z << endl;
   return(out);
}
```



Note that our *new overload just uses the built in overload of* << to output each component of the vector along with punctuation and a '\n'.

Cascading *cout*

Our implementation will also let us print an arbitrary number of vectors:

```
cout << v1 << v2 << v3;
```

This seemingly magic behavior occurs because of two things:

- C++ evaluates a sequence of << operations in *left to right order*
- The operator<< function *returns its left side operand as its result.*

So what REALLY happens is: `cout << v1` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```
cout << v2 << v3;
```

Then `cout << v2` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```
cout << v3;
```

Then `cout << v3` is evaluated and the value it returns is "dropped on the ground", because there is nothing to assign it to.

The finite plane

The finite plane is a rectangular region of finite area within a general plane. It is properly implemented as a derived class of the general plane. It is specified in the following way:

```
fplane origin
{
   material white

   normal 1 0 1
   point  0 0 -6

   xdir   1 0 0
   dimensions 4 2
}
```

object_t constructor

plane_t constructor

fplane_t constructor

The three sets of attributes must be specified in the order shown because of the way in which the constructors are executed. The order *within* each set of attributes is arbitrary.

In retrospect a better way to have built the generalized parser would have been to have *built the parse control table dynamically.*

The value of *point* which was an arbitrary point on the infinite plane specifies the lower left corner of the finite plane.

The two new attributes of the *fplane_t* are its *dimensions* in world coordinate units. The first dimension is the size in the *x* direction and the second the size in the *y* direction. The *x* direction is indirectly specified by the *xdir* attribute. The *xdir* attribute is a vector which *when projected into the infinite plane* specifies the *x* direction of the rectangle. The *y direction* is implicitly given by the cross product of the unit plane normal and the unitized, projected *xdir.*

The fplane_t class definition

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

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

protected:
   vec_t   newloc;  // translated then rotated hitloc

private:

   mat_t  rot;        /* rotation matrix            */
   vec_t  projxdir;   /* projected unitized xdir    */
   vec_t  xdir;       /* input xdir                 */
   double dims[2];    /* input dims in world coords */

};
```

The *fplane_t* constructor:

attrmax = *2* tells plane_t constructor to consume only 2 attributes

```
/**/
fplane_t::fplane_t(
FILE    *in,
model_t *model,
int      attrmax) : plane_t(in, model, 2)
{
```

The constructor should perform the following actions:

- set the *objtype* to "fplane"
- parse the *xdir* and *dimensions* attributes
- project *xdir* onto the plane creating *projxdir.*
- *ensure* that *projxdir* is not {0.0, 0.0, 0.0}
- make *projxdir* unit length.

Next it is necessary to make the *rot* matrix that can rotate the *projxdir* vector into the x-axis and the plane normal into the positive z axis.

- copy *projxdir* to row 0 of *rot*
- copy the plane *normal* to row 2 of *rot* and make it unit length.
- set row 1 of *rot* to row_2 x row_0.

The *fplane_t dumper*:

```
 /***/
void fplane_t::dumper(
FILE *out)
{
```

plane_t::dumper(out);

The dumper should produce a formatted output of the xdir, projxdir, dimensions, and the rotation matrix

The *fplane_t hits* function

```
/***/
double fplane_t::hits(
vec_t    *base,      /* ray base              */
vec_t    *dir)       /* unit direction vector */
{
   vec_t newloc;
   double t;
```

In general, determining if a ray hits a rectangular finite plane of arbitrary location and orientation seems like a difficult problem. The *hits* function of the standard plane can certainly be used to determine if and where the ray hits the *infinite* plane in which the rectangular plane lies.

- If the infinite plane is missed then clearly so is the finite plane.
- If the infinite plane is hit, the problem is determining whether or not the hit point is "in bounds" or "out of bounds.

If the lower left corner of the finite plane happened to lie at the origin, and the projected *xdir* happened to lie on the *x-axis,* and the plane normal happened to lie on the z-axis, the problem would be easy. The hit is in bounds if and only if:

0 <= hitloc.x <= dims[0] *and*
0 <= hitloc.y <= dims[1]

We can make it possible to perform this simple test if we *translate* the *point* defining *the lower left corner of the plane to the origin* and then *rotate* the plane into the proper orientation. Therefore after we apply these operations to the original *hitloc* we can make the simple test above work. So that lighting will still work *we cannot modify the original hitloc.*

So we set *newloc* to *hitloc – point.* Then we rotate *newloc* by transforming it with the *rot* matrix. Then we can do the test:

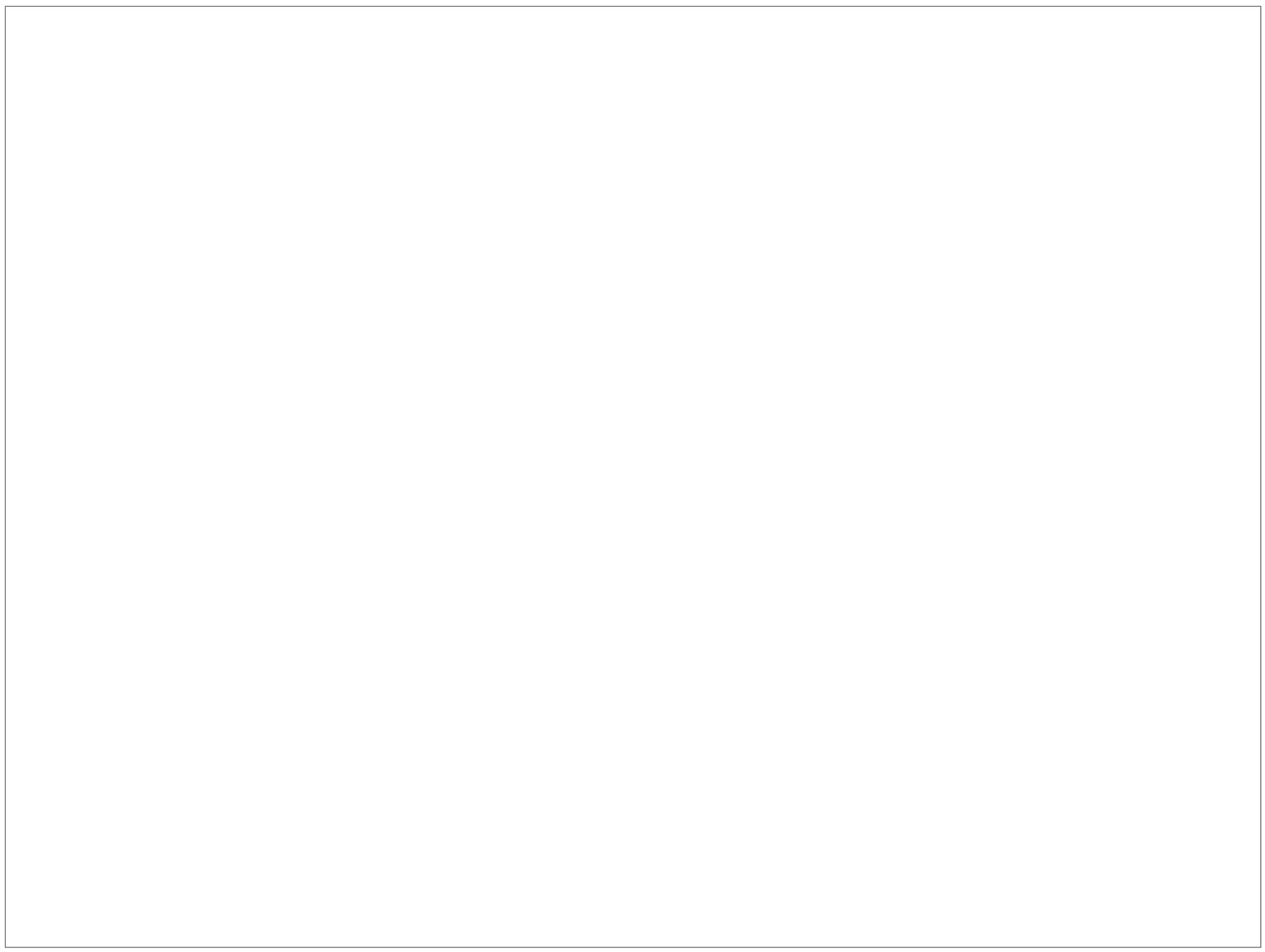
0 <= newloc.x <= dims[0] *and*
0 <= newloc.y <= dims[1]

We return *t* and *newloc* is saved in a protected element of the *fplane_t.*

The textured plane

In CPSC 101 you wrote a program which loaded a *ppm* image and then *fit* it to arbitrary dimensions by shrinking or stretching. Such an image is commonly called a *texture* and the process of mapping it onto a planar surface is called *texture mapping*. Depending upon your 101 instructor you may have also *tiled* a small image by repeating it over a large surface.

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. The specular refection of the wall and the photograph on the floor will be addressed later in the course.

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 *0* means stretch the texture to *fit* the plane. Mapping mode *1* means repeatedly *tile* the texture.

```
texplane oak_floor
{
   material oak_tex
   normal     0   1 0
   point     -6   0 0
   xdir       1   0 0
   dimensions 24  6
   texname    ../images/oak.ppm
   mode       1
}
```

New classes required for texture mapping (take 1)

The *texture_t* class is responsible for managing the texture.

- Its constructor will *malloc()* the *imagebuf* buffer and read a *.ppm* file whose name is specified as a parameter into it.

- The *gettexel()* function takes relative *x,y* offsets in the range *(0.0, 1.0),* looks up the *texel* in the pixmap, converts it from *irgb_t* to *drgb_t* and saves the result in the location pointed to by *texel.*

- The *getdim()* function just returns the dimensions of the texture in pixels.

```
class texture_t
{
public:
   texture_t();
   texture_t(char *);
   void  gettexel(double relx, double rely, drgb_t *texel);
   void  getdim(int *x,int *y);
private:
   int            xdim;
   int            ydim;
   irgb_t         *imagebuf;
};
```

The *texture_t* class methods

The constructor is responsible for processing the *ppm* header and reading the *irgb_t* image data.

```
texture_t::texture_t(
char *name)                 /* name of the .ppm file */
{
   FILE *ppmfile;
```

- *fopen()* the .ppm file and verify that ppmfile != NULL
- read the .ppm header and extract *xdim* and *ydim*
- *malloc* the buffer for the *irgb_t* data and save address in *imagbuf*
- *fread* the *irgb_t* data and verify correct amount read

```
}
```

The *getdim* method just supplies the dimensions of the texture

```
void texture_t::getdim(
int *x,     /* return location for xdim
int *y)     /* return location for ydim
{
```

- fill in **x* and **y*

```
}
```

The *gettexel()* method converts relative offsets to actual and converts the texel from *irgb_t* to *drgb_t*

```
void texture_t::gettexel(
double xrel,   /* relative x location (0, 1) */
double yrel,   /* relative y location (0, 1) */
drgb_t *tex)   /* return drgb_t texel here   */
{
```

- convert relative *x* and *y* offset to pixel *x* and *y* offsets
- convert pixel *x* and y offset to *imagebuf* offset
- convert texel from *irgb_t* to *drgb_t* and save in **tex*

}

The *texplane_t* class methods

```
texplane_t::texplane_t(
FILE    *in,
model_t *model,
int      attrmax) : fplane_t(in, model, 2)
{
   int  mask;
```

- parse *texname* and *mode* attributes
- acquire pixel dimensions from *cam->get_pixdim()*
- create new texture_t and save pointer

}

Necessary modifications to the *fplane_t* class

The *texplane_t* will need to access the location of the translated and rotated hitpoint that is computed in performing the *fplane* hits test. Thus we put it and the dimensions in the *protected* section of the *fplane_t*. It also needs the *dims* of the plane.

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

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

protected:
   vec_t  newloc;
   double dims[2];

private:

   mat_t  rot;        /* rotation matrix        */
   vec_t  xdir;
   vec_t  projxdir;

};
```

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 *texplane_t::getdiff()* function.

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.

```
void texplane_t::getdiff(
drgb_t *value)      // where to store final texel color
{
   drgb_t matdiff;  // diffuse reflectivity of material
   drgb_t  texel;
```

- Acquire the diffuse reflectivity *mat* of the underlying *object_t*
- Ask *texture_map(&texel)* to return value of the texel hit.
- store component-wise product of *matdiff* and *texel* in **value*

}

The *texture_t::getamb( )* function works in an analogous way

The *texture_map* function

This mission of this function is to determine the *texel* which maps to the most recent hit point on the finite plane object. In general this would 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 "*newloc*" data that should be stored with the "*dims"* in the *protected* section of the *fplane_t.*

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.

```
void texplane_t::texture_map(
drgb_t *texel)          /* output texel value      */
{
   drgb_t texel;
```

- compute relative location of the texel in the texture
- ask *texture_t::get_texel* to return *drgb_t* value in *texel*

}

This is a *poor design* computing relative location of the texel in the texture should be part of the texture class!!

Computing the relative location of the texel

Two procedures are necessary. The easier one supports *fit* mode. *Tiled* mode is a little more complicated.

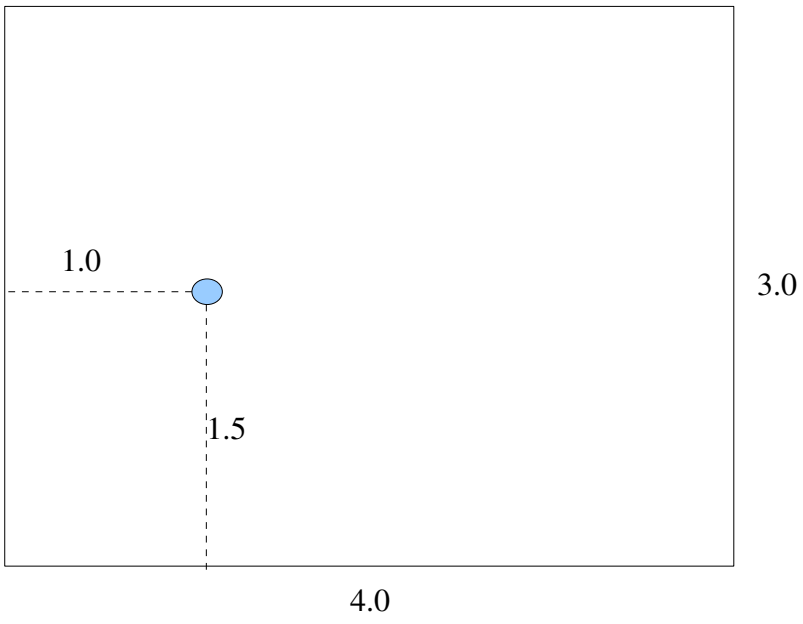
fit mode

- The relative *x* location is simply the *x* component of *newloc / dims[0]*

Example: Given the following dimensions:

- The texplane is size 4 x 3 in world coordinates.
- The texture is size 800 x 600 in pixels.
- The *newloc* location is (1.0, 1.5, 0.0)

The the relative location *newloc* with respect to the *texplane* is (1.0 / 4.0, 1.5 / 3.0) = (0.25, 0.50) and the pixel coordinates of the target *texel* are (0.25 * 800, 0.50 * 600) = (200, 300)



tile mode

- Convert *newloc.x* from world to pixel coords by dividing by pixel size in world coords and save in *int pix_x.*
- Compute the remainder when the *x* dimension of the *texture* is divided by *pix_x.*
- Divide *in floating point* this remainder by the *x* dimension of the texture.
- This quotient is the relative *x* location.

Example: Suppose the dimensions of the fplane are 4 x 3 in world coordinates.

- The image to be produced is 640 x 480 pixels.
- The image is size 12 x 9 in world coordinates
- The texplane is size 4 x 3 in world coordinates.
- The texture is size 80 x 60 in pixels.
- The *newloc* location is (1.0, 2.0)

The pixel size in world coordinates is (12/640, 9/480)

(pix_x, pix_y) = *newloc* in pixel coordinates is thus (640 * 1.0 / 12, 480 * 2.0 / 9) = (53, 106)

Compute (53 mod 80) = 53 and (106 mod 60) = 46. So the actual texel coordinates are (53, 46).

However *get_texel()* wants relative coordinates so (53/80, 46/60) must be passed to *get_texel.*

New classes required for texture mapping (take 2)

The *texture_t* class is responsible for managing the texture.

- Its constructor will *malloc()* the *imagebuf* buffer and read a *.ppm* file whose name is specified as a parameter into it. You should have code from CPSC 101 that can do this. It should also obtain the pixel size in world coordinates from the *camera_t.*

- The *texture_fit()* method is passed relative (x, y) coordinates. It converts these to absolute pixel coordinates and then uses *gettexel()* to retrieve the texel.

- The *texture_tile()* method is passed the world coordinates of the hit point on the textured plane in the translated and rotated coordinate system in which the z-coordinate disappears. It converts these to absolute pixel coordinates and then uses *gettexel()*

- The *gettexel()* function takes absolute pixel *(x y)* coordinates , looks up the *texel* in the pixmap, converts it from *irgb_t* to *drgb_t* and saves the result in the location pointed to by *texel.*

```
class texture_t
{
public:
   texture_t();
   texture_t(char *, model_t *model);
   void  texture_fit(double relx, double rely, drgb_t *);
   void  texture_tile(double worldx, double worldy, drgb_t *);
private:
   void  gettexel(int, int, drgb_t *);
   double         pix_x_size;   // pixel size in world coords
   double         pix_y_size;
   int            xdim;         // of the texture
   int            ydim;
   irgb_t         *imagebuf;
};
```

The *texture_t* class methods

The constructor is responsible for processing the *ppm* header and reading the *irgb_t* image data.

```
texture_t::texture_t(
char *name)                 /* name of the .ppm file */
{
   FILE *ppmfile;
```

- *fopen()* the .ppm file and verify that ppmfile != NULL
- read the .ppm header and extract *xdim* and *ydim*
- *malloc* the buffer for the *irgb_t* data and save address in *imagbuf*
- *fread* the *irgb_t* data and verify correct amount read
- obtain the dimensions of a pixel in world coordinates from the camera
- optionally put *this* texture in the texture list.

```
}
```

The *gettexel()* method converts relative offsets to actual and converts the texel from *irgb_t* to *drgb_t*

```
void texture_t::gettexel(
int    xpix,     /* pixel coordinates relative */
double ypix,     /* to lower left origin...    */
drgb_t &texel)   /* return drgb_t texel here   */
{
```

- convert pixel *x* and y offset to *imagebuf* offset dealing with the upside down problem
- convert texel from *irgb_t* to *drgb_t* and save in *texel*

}

Mapping hit location to texture location.

Two strategies and corresponding routines are used depending upon whether *fit* mode or *tile* mode is in effect.

```
void  texture_fit(
double relx,
double rely,
drgb_t *texel);
{
```

- convert relative x and y coordinates to absolute pixel coordinates
- ask *gettexel()* to retrieve the texel.

}

```
void  texture_tile(
double worldx,
double worldy,
drgb_t *texel);
{
```

- Convert world hit coordinates to plane pixel coordinates by dividing pixel pixel size
- Convert plane pixel coordinates to texel coordinates by mod-ing with texture dimension
- ask *gettexel()* to retrieve the texel.

```
}
```

The *texplane* class

This class is derived from the *fplane_t* class and as such is also a derivative of *plane_t* and *object_t*. Note that it *does not* provide a virtual *hits* function. Therefore, the *hits* function of the *fplane_t* is used. It does override the default *getdiff()* and *getamb()* functions.

These methods return the component-wise product of the *material* associated with the *texplane* and the *texel* associated with the pixel.

The *texture_map()* function is a private method. Its mission is to compute the relative location of the texel in the pixmap, ask *get_texel()* to provide the *drgb_t* representation of the texel and then do a componentwise multiplication of the texel with the material.

```
class texplane_t: public fplane_t
{
public:
   texplane_t(FILE *, model_t *, int);
   texplane_t();
   virtual void  dumper(FILE *);
   virtual void  getdiff(drgb_t *pix);
   virtual void  getamb(drgb_t  *pix);

private:
   texture_t     *texture;    // pointer to associated texture
   char          texname[64]; // name of texture file
   int           mode;        // 0 -> fit  1 -> tile
};
```

The *texplane_t* class methods

```
texplane_t::texplane_t(
FILE    *in,
model_t *model,
int      attrmax) : fplane_t(in, model, 2)
{
   int  mask;
```

- parse *texname* and *mode* attributes
- acquire pixel dimensions from *cam->get_pixdim()*
- create new texture_t (or lookup in list) and save pointer

}

Necessary modifications to the *fplane_t* class

The *texplane_t* will need to access the location of the translated and rotated hitpoint that is computed in performing the *fplane* hits test. Thus we put it and the dimensions in the *protected* section of the *fplane_t*. It also needs the *dims* of the finite plane.

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

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

protected:
   vec_t  newloc;
   double dims[2];

private:

   mat_t  rot;        /* rotation matrix       */
   vec_t  xdir;
   vec_t  projxdir;

};
```

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 *texplane_t::getdiff()* function.

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.

```
void texplane_t::getdiff(
drgb_t *value)      // where to store final texel color
{
   drgb_t matdiff;  // diffuse reflectivity of material
   drgb_t  texel;
```

- Acquire the diffuse reflectivity *mat* of the underlying *object_t*
- Depending on mode ask *texture->texture_fit()* or *texture->texture_tile()* to return value of the texel hit.
- store component-wise product of *matdiff* and *texel* in **value*

}

The *texture_t::getamb( )* function works in an analogous way

Computing the values to be passed to texture_fit or texture_tile()

Two procedures are necessary.

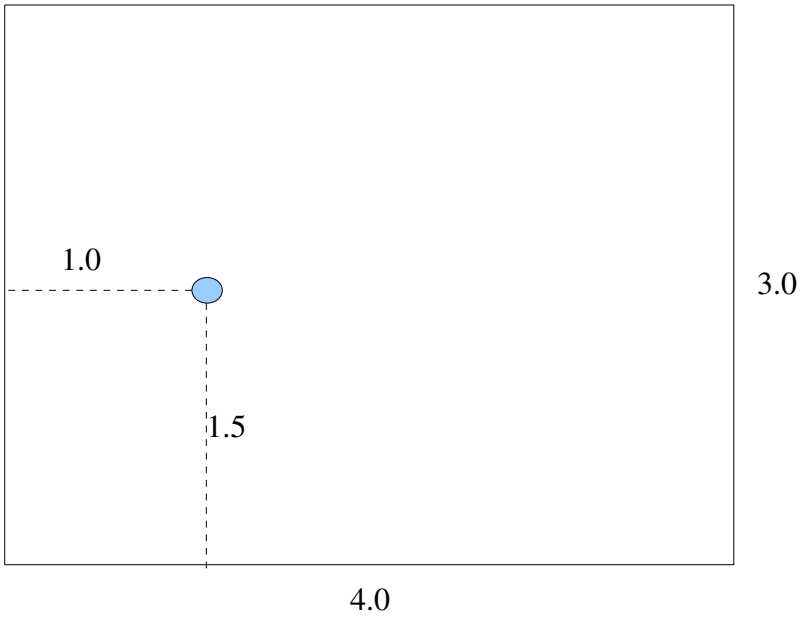
fit mode

- The relative *x* location is simply the *x* component of *newloc / dims[0]*

Example: Given the following dimensions:

- The texplane is size 4 x 3 in world coordinates.
  The *newloc* location is (1.0, 1.5, 0.0)

The the relative location *newloc* with respect to the *texplane* is (1.0 / 4.0, 1.5 / 3.0) = (0.25, 0.50)



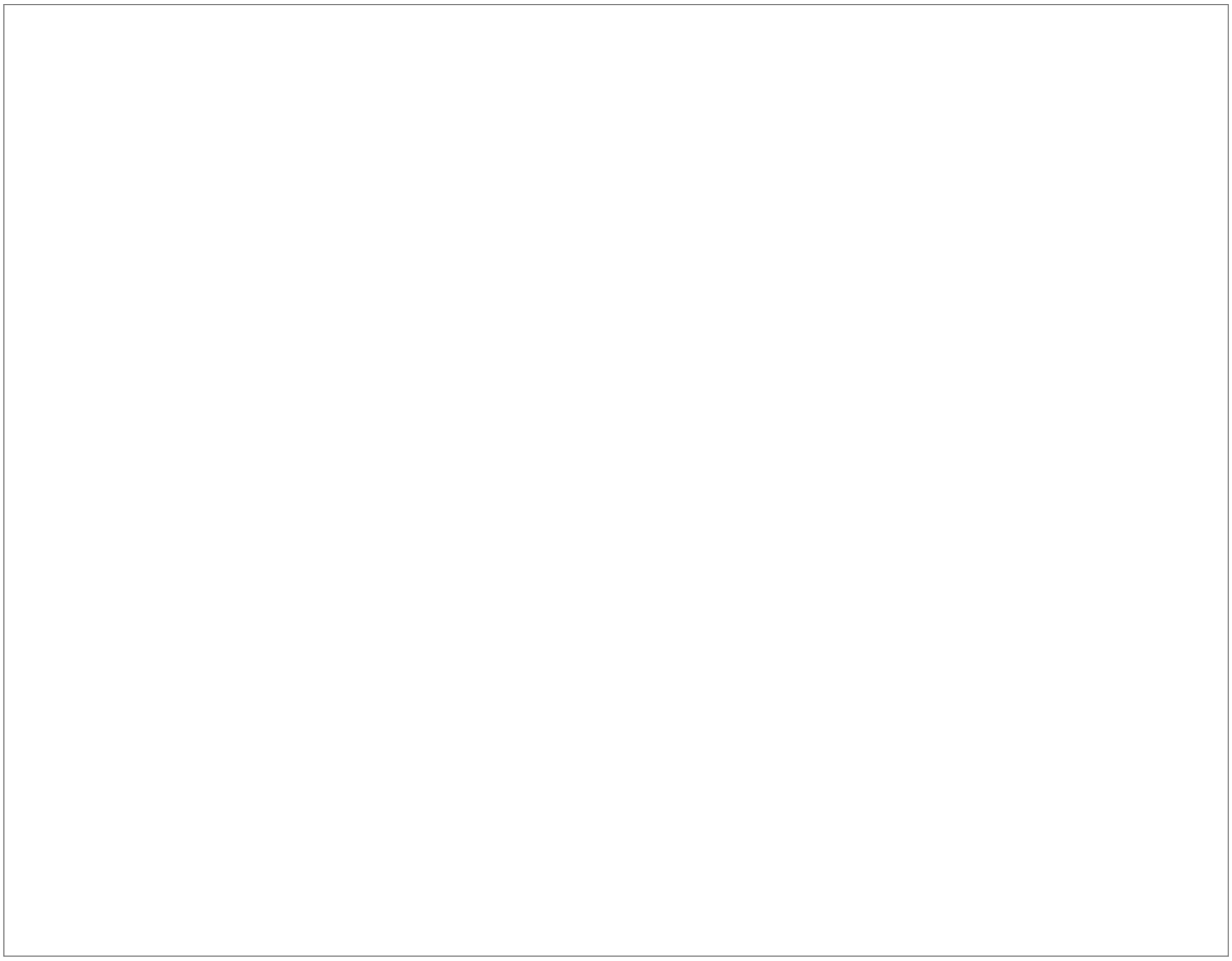
tile mode:

- just pass (1.0, 1.5) to *texture_tile()*

Using a texture as a source of illumination

Consider the traditional color slide projector. A bright light is placed behind the translucent slide and directed toward the screen. We can consider the light source as casting rays which pass through the slide and illuminate the screen. In passing through the color slide the ray is filtered in such a way that it takes on the color of the slide element through which it passes.

We can model this process by placing a light source behind a translucent textured plane. The illumination it provides is model as follows. For each *hitpoint* on a reflective object a ray is fired from the illumination source toward the hit point. If the ray hits the textured plane, then the projector potentially illuminates the hitpoint. The emissivity of the projector at the hitpoint is modeled as the component-wise product of the emissivity of the light source with the texel through which the ray passes.



Characteristics of the projector and multiple inheritance

The projector has characteristics of both the *light_t* and the *texplane_t.* In fact the code in *projector.cpp* that is used to glue them together is very small.

```
camera cam1
{
  pixeldim  800 600
  worlddim  8 6
  viewpoint 4 3 6
}
material white
{
   ambient 0 0 0
   diffuse 9 9 9
}
sphere earth
{
   material white
   center 4 3 -8
   radius 5
}
material dummy
{
   ambient 0 0 0
}

projector front
{
   emissivity 17 17 17
   location   4 3 8

   material dummy
   point      2 1.5 3
   normal     0 0 1

   xdir       1 0 0
   dimensions 4 3

   texname ../images/sky.ppm
   mode 0
}
```

Projector doesn't need a material. But the object_t *dumper* does :-(

Attributes of the *light_t*

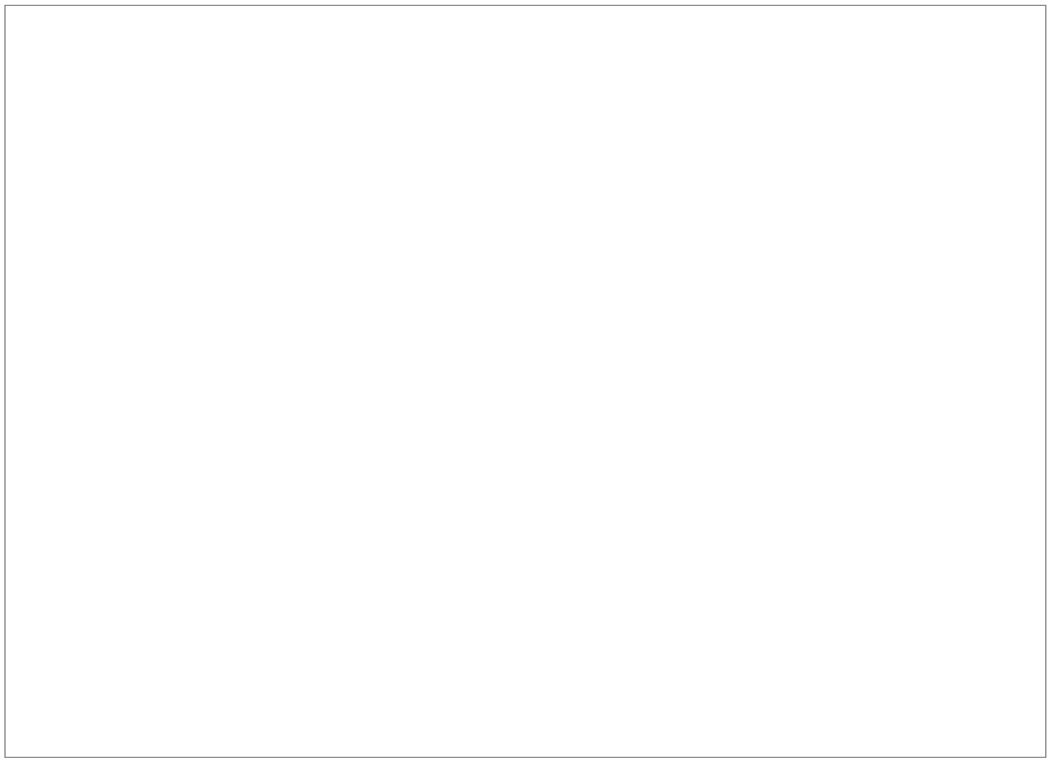
Attributes of the *texplane_t*

The *projector_t* class

The class definition is shown below. Relevant aspects of the definition are noted below.

- No variables are needed
- Like other lights we want the projector to be invisible. We can make this happen by overriding the default *fplane_hits* function with one that always declares a miss.
- The *vischeck* function determines if a ray from the *location* of the projector light source to a *hit location* passes through the textured plane. If not the projector doesn't illuminate the object.
- The *getemiss()* function returns the component-wise product of the emissivity of the projectors light source with the texel through which the ray from the light source to the hit location passes.

```
class projector_t: public light_t, public texplane_t
{
public:
   projector_t(FILE *, model_t *, int);
   virtual int     vischeck(vec_t *);
   virtual void    dumper(FILE *);
   virtual void    getemiss(drgb_t  *);
   virtual double  hits(vec_t *base, vec_t *dir)
         { return(-1);}
};
```



Projector class methods

Creating a new projector_t creates instances of *light_t, object_t, plane_t, fplane_t,* and *texplane_t.* The *projector_t* overrides only the *getemiss()* and *vischeck()* methods of the *light_t* and the *hits()* method of the *object_t.*

```
/**/
/* Create a new projector description */
projector_t::projector_t(
FILE     *in,
model_t  *model,
int      attrmax):light_t(in, model, 2),
                        texplane_t(in, model, 0)
{
```

- Store the string "projector" in *objtype*

```
}
```

Operational class methods

```
int projector_t::vischeck(
vec_t *hitloc)              // hit location on the object
{
    vec_t dir;
    double t;
```

- Compute *dir* vector from *location* to *hitloc*
- Ask *fplane_t::hits()* if ray fired from *location* in direction *dir* hits the *fplane* that is part of this projector. (The normal test for occlusion means we don't have to worry about intervening objects here.
- If so, return 0 else return 1

```
}
void projector_t::getemiss(
drgb_t *emiss)
{
   drgb_t texel;
```

- Ask *texture->texture_fit() or texture->texture_tile()* to return the *texel* that the ray from the light location to the *hitloc* passes through. Note that *this works only because of the work done in vischeck().* The value stored in *newloc* when *vischeck()* called fplane_t::hits() will be used here.
- Store component-wise product of *texel* and *emissivity* in return value *emiss*

```
}
```

Patching it all together

Even though the *projector_t* requires minimal new code, making it all work requires some back patching of existing methods. Two of them hit the *light_t* class. We need to add and write default instances of the two virtual functions that the *projector_t* needs to override.

- The default behavior of *getemiss()* must be to simply copy the *emissivity* of the light to the location provided.
- The default behavior of *vischeck()* is to always return 0 (visible). Note that *vischeck* has *nothing to do with occlusion by self or other objects*. Those tests *must remain*. The mission of *vischeck()* is to determine if the *hitloc* is illuminated by *directional* lights such as the projector and the spotlight.

```
class light_t
{
public:
   light_t(){};
   light_t(FILE *in, model_t *model, int attrmax);

   virtual ~light_t(){};


   virtual void    getemiss(drgb_t *)
   virtual int     vischeck(vec_t  *);
   virtual void    illuminate(model_t *, object_t *, drgb_t *);
```

Invoked for *every hit point and every light.*

Patches to *illuminate*

```
void light_t::illuminate(
model_t     *model,
object_t    *hitobj,  /* The object hit by the ray   */
drgb_t      *pixel)   /* add illumination here       */
{

   vec_t    dir;    // unit direction to light
   vec_t    revdir; // unit direction from light to hitpt
   object_t *obj;   // closest object in dir to light
   double   close;  // dist to closest object in dir to light
   double   cos;    // of angle between normal and dir to light
   double   dist;   // to the light from hitpoint
   drgb_t   diffuse = {0.0, 0.0, 0.0};

/* Compute the distance from the hit to the light and a unit */
/* vector in the direction of the light from hitpt           */
   ..........
/* Ask vischeck to determine if the light can illuminate the */
/* hitloc in the absence of self or other occlusion          */
   ..............
/* Test the object for self-occlusion and return if occluded */
   ..........
/* Ask find_closest_object() if a ray fired toward the light */
/* hits a "regular" object.  Pass "hitobj" as the "lasthit"  */
/* parameter so it won't be hit again at distance 0.         */
   ..........
/* If an object is hit and the distance to the hit is   */
/* closer to the hitpoint than the light is, return     */
   ............
/* Arriving at this point means the light does illuminate */
/* object. Ask hitobj->getdiff() to compute diffuse       */
/* reflectivity                                           */
   ............
/* Ask getemiss() to return the emissivity of the light   */
   ............
/* Multiply componentwise the diffuse reflectivity by     */
/* the emissivity of the light.                           */
   ............
/* Scale the resulting diffuse reflectivy by cos/dist     */
   ............
/* Add scaled value to *pixel.                            */
```

Patches to *plane_t*

- Its often useful to place the projector in *positive* z space.
- The *hits* function of the *plane_t* is suppose to return -1 when the hit occurs in +*z* space.
- This test must be disabled if the *objtype* is a *projector*.

The *BIG UGLY* .... parsing woes

Because of the multiple inheritance when a *projector_t* is created constructors for all of the parent classes execute in a top down fashion. Because the *projector_t* is more of a light than a visible object it must go on the *light* list but *need not* go on the object list. Because the *light_t* parent is specified before the texplane_t, the *light_t* constructor *will* run first.

```
class projector_t: public light_t, public texplane_t
```

Because the constructors drive the parsing, this means that the *light* related attributes, *location* and *emissivity* must come first in the definition. So the *model_t* constructor will consume `projector`. Then the *light_t* constructor will consume `front {` and the attributes. Then the *object_t* constructor will be invoked. Its first action will be to try to consume the object name `front` and `{` , but they are long gone so the *object_t* constructor will abort.

```
projector front
{
   emissivity 17 17 17     Attributes of the light_t
   location   4 3 8

   material dummy
   point      2 1.5 3
   normal     0 0 1        Attributes of the texplane_t
   xdir       1 0 0
   dimensions 4 3
   texname ../images/sky.ppm
   mode 0
}
```

Possible "solutions" most ugly ... some unspeakbly so.

We basically have three options here

- Provide *object_t* what its looking for by patching it into the entity description.

```
projector front
{
  emissivity 17 17 17
  location   4 3 8
  ugly_hack {
  material dummy
  point      2 1.5 3
```

- Update every constructor in the *object_t* hierarchy to carry a parameter that can be used to tell *object_t not to look* for the `object_name {` because this is a *virtual object.)*

- Provide overloaded *texplane_t, fplane_t, plane_t, and object_t* constructors giving an alternate path for the *projector_t.*

- Provide a global variable *int dontlook* = 0; in object.cpp that can be accessed using *extern int dontlook;* in model.cpp. Then in *model_t* do

```
case PROJECTOR:
   dontlook = 1;
   new projector_t(in, this, 0);
   dontlook = 0;
   break;
```

  and in the *object_t* constructor if *dontlook == 1* then just return after setting the object cookie avoiding the parsing, material lookup, and object list operations.

Eliminating the useless *material_t*

With the last two approaches we can also obviate the useless *material* definition. But its important to remember make the *dumper* not try to print material descriptions when *this->mat == NULL!!*

```
/**/
/* Create a new object description */

object_t::object_t(
FILE       *in,
model_t    *model,
int        dontlook)
{
   char objtype[NAME_LEN];
   int count;

   cookie = OBJ_COOKIE;

   if (dontlook)
      return;

......... rest of object loading ......
}
```

The final (so far) entity description

```
projector front
{
   emissivity 17 17 17
   location   4 3 8

   point      2 1.5 3
   normal     0 0 1

   xdir       1 0 0
   dimensions 4 3

   texname ../images/sky.ppm
   mode 0
}
```

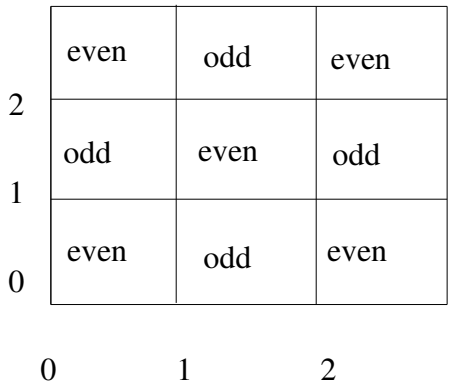
Attributes of the *light_t*

Attributes of the *texplane_t*

Another possible optimization is to add specific object affinity to a projector. This will allow it to illuminate only a single object and not anything that might be visible near the object.

The general tiled plane

Tiled floors are common elements in raytracing systems primarily because they create interesting reflections on specular spheres! The simplest tiled planes lie in x-z space (the normal is (0, 1, 0)) and have unit tile spacing in world coordinates.



To determine the color of a particular tile convert the *x* and *z* coordinates to *int* and then note that choosing a color based upon whether the sum of the integerized x and z is even or odd produces the desired tiling effect.

While this algorithm works correctly in the example shown above, it does have a nasty effect that is not shown. Consider what happens when we extend to negative *x* space. If we simply integerize -0.5 we get 0... the same as when we integerize +0.5.

Thus we end up with an ugly "double -wide" strip of tiles centered along the line $x = 0$ to prevent this we can add -1 to negative numbers before integerizing or add some big positive number to all values before conversion to integer. This trick effectively translates the ugly stripe out of the visible area.

We will describe a slightly more sophisticated approach in which:

- the plane may have any normal vector (not just (0, 1, 0) or (0, 0, 1) etc)
- the tiles may have any rectangular dimensions
- the tiles may be laid out with any orientation

The *tiled plane* object

The tiled plane class, *tplane_t,* like the *fplane_t* illustrates multilayer inheritence. Since it is *explicitly* derived from *plane_t,* it is *implicitly* derived from *object_t* as well. Thus whenever a *tplane_t* is created an instance of a *plane_t,* and an *object_t* will automatically be created as well.

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 *, model_t *, int);

   virtual void    dumper(FILE *);
   virtual void    getamb(drgb_t  *);
   virtual void    getdiff(drgb_t *);
// virtual void    getspec(drgb_t *);


private:
   int   select(void);  /* 0-> forgrnd 1 -> back */
   double dims[2];      /* tile dimensions       */
   material_t *altmat;  /* background material   */
   mat_t  rot;          /* rotation matrix       */
   vec_t  xdir;         /* x direction of tiling */

};
```

Also note that there is no *getspec().* What this means is that in Dr. Westall's present implemenation the alternate tiles *literally* ineherit the 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 *raytrace.c* code in which these reflectivities are accessed.

The *tplane_t* model description

```
camera cam1
{
  pixeldim  640 480
  worlddim  8 6
  viewpoint 4 3 8
}
material white
{
   diffuse 2  2 2
   ambient 1  1 1
}
material brown
{
   diffuse 3  3 0
   ambient 1  1 1
}
light center
{
   location 4 3  0
   emissivity 10 10 10
}
tiled_plane  left
{
   material white
   normal   6.93 0 4
   point    0 0 0
   xdir     1 1 0
   dimensions 1 2
   altmaterial brown
}

tiled_plane   right
{
   material white
   normal  -6.93  0 4
   point    4 0 -6.93
   xdir     1 1 0
   dimensions 1 2
   altmaterial brown
}
```

The *tplane_t* constructor

```
tplane_t::tplane_t(
FILE    *in,
model_t *model,
int      attrmax) : plane_t(in, model, 2)
{
```

- Parse the three required parameters
- Ask *material_find()* to return a pointer to the alternate (background) material
- Project *xdir* into the plane and make it a unit vector
- Build a rotation matrix that rotates the plane normal into the z-axis and the projected *xdir* into the x-axis.

```
}
```

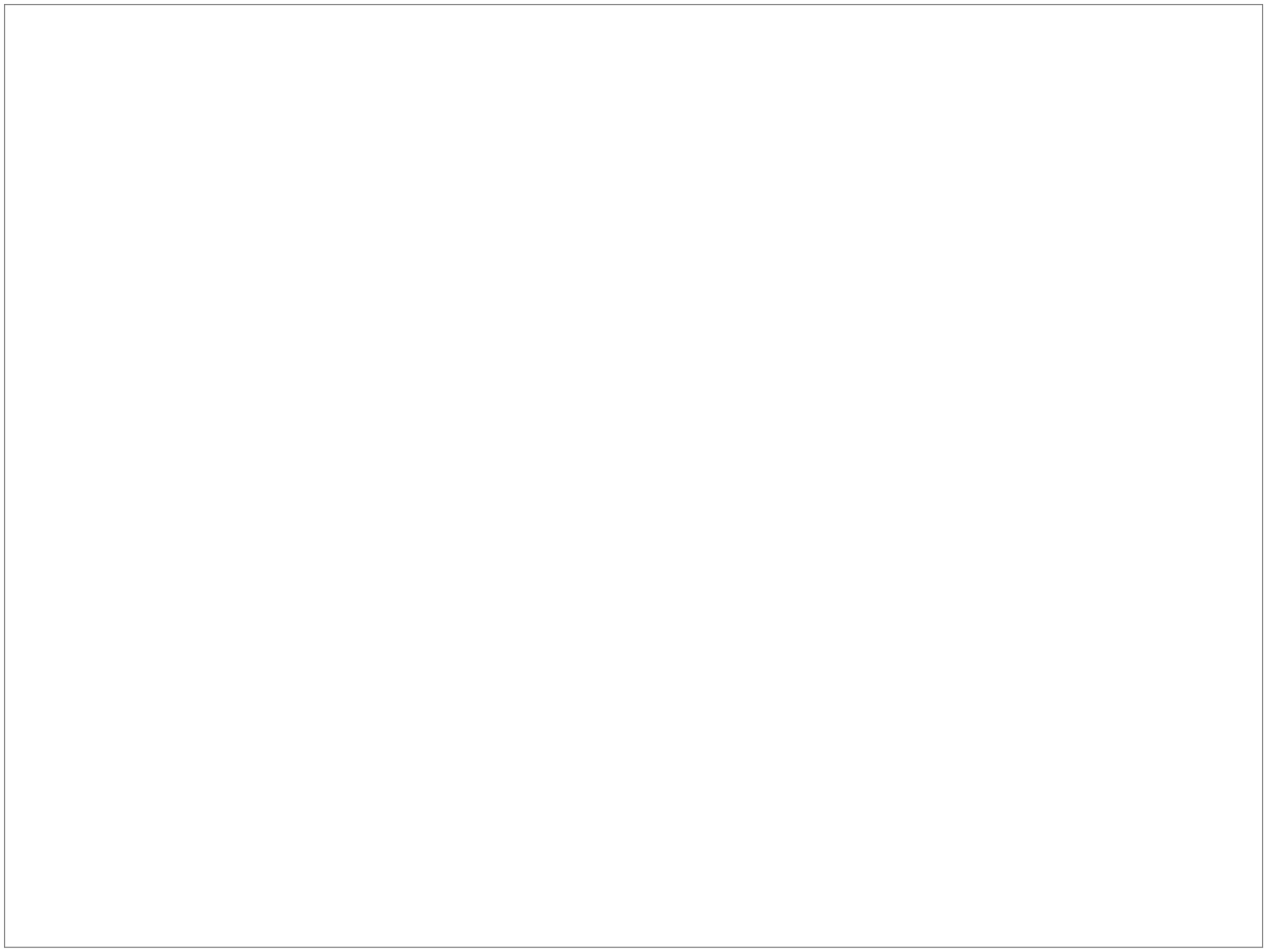
The *getdiff and getamb* methods

These methods are just wrappers that call other methods to retrieve the appropriate reflectivity based upon the value returned by *select()*

```
void  tplane_t::getdiff(
drgb_t *value)
{
  if (select() == 0)
        ask the default (object level) getdiff() to fill in value
  else
        ask material_getdiff() to fill in the diffuse reflectivity of altmat.
}
```

The *select()* method

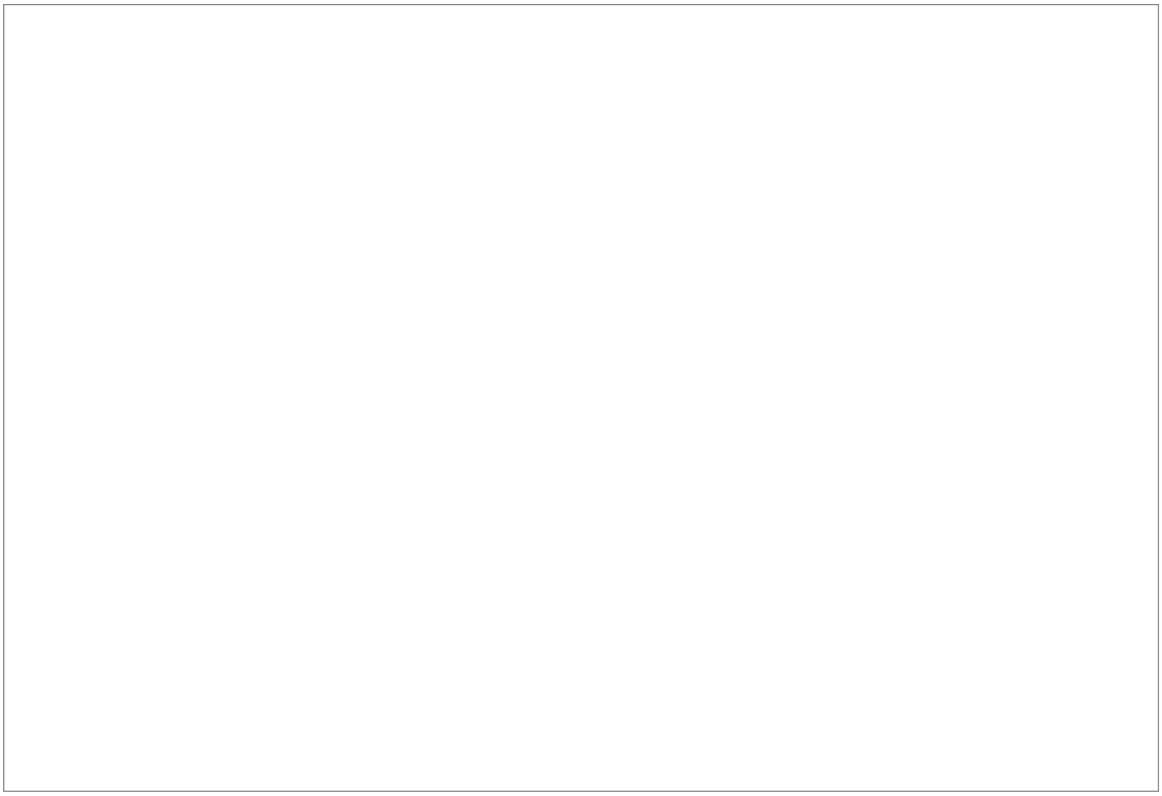
- Apply the *rot* matrix to *hitloc* to rotate it into the plane having normal (0, 0, 1) with the projected xdir of the tiling parallel to the x-axis. Because the plane is infinite, you don't need to translate the *point* defining the plane to the origin. If you choose to do that, your tile pattern will be unchanged but the tiles may appear shifted in the output image.
- Add 100000 to *newloc.x* and *newloc.y* (westall's hack to avoid ugly doublewide stripe at origin).
- Divide *newloc.x* by *dims[0]* and *newloc.y* by *dims[1]* to compute relative tile number in each direction and use the sum of these values to determine if the tile is forground or background.



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

It turns out that those two measures are inconvienient for humans to deal with. So our *spotlight_t* will transform the human-centric measures to computer-centric measures in the constructor.

The spotlight class and constructor

The spotlight class needs only three methods: a constructor, a dumper, and a visibility checker that decides whether or not a hitpoint is in the spotlight's cone. Unlike the projector, it is a pure derivative of the *light_t.*

```
class spotlight_t: public light_t
{
public:
   spotlight_t(FILE *, model_t *, int);
   virtual void    dumper(FILE *);
   virtual int     vischeck(vec_t *hitloc);

private:
   double  theta;      // half angle in degrees
   vec_t   point;      // point the centerline hits
   vec_t   dir;        // unit vec centerline direction
   double  costheta;   // cosine of the cone's half angle

};
```

A sample input looks like

```
spotlight centre_red
{
   location 4 3 1
   emissivity 5 0 0

   point    4 0 -2
   theta    20
}
```

The constructor must parse the *point* and *theta* parameters. It must compute a unit vector in the centerline direction and store it in *dir* and convert *theta* to radians and store its cosine in *costheta.* The C math library supplies a *cos()* function that can be used:

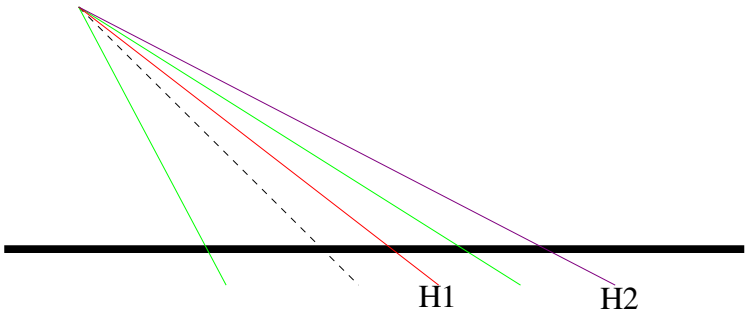
```
costheta = cos(theta);
```

but theta *must* be in radians not degrees .

The *vischeck()* method of the *spotlight_t*

A spotlight can illuminate the *hitloc* if and only if a vector from the *location* of the spotlight (light) to the *hitloc* lies inside the spot cone. Therefore, it is necessary to incorporate such a test in the *vischeck()* method of the *spotlight_t* class.

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

Recursive functions

Recursive functions are those that directly (or sometimes indirectly) invoke themselves.

On the *positive side* these functions can produce amazingly succinct solutions to problems the are extremely difficult to solve in other ways. In fact the "other ways" often involve emulation of recursions or transformation of the problem in to a more complicated representation.

On the *challenging side* the algorithms can be quite subtle and much more difficult to develop than any of the algorithms that we have seen so far.

The classes of problem for which recursion is most well suited are searches through trees and even general graphs.

Applications include:

- parsing of languages that support nested structures (e.g. evaluation of arithmetic expressions in which parentheses may be nested;
- searching for any path or the shortest path through a maze type structure
- generation of subsets of a larger set of elements

Recursion also provides a convenient way to deal with reflected rays in a raytracing environment. However, it is reasonably easy to do this in a non-recursive way as well.

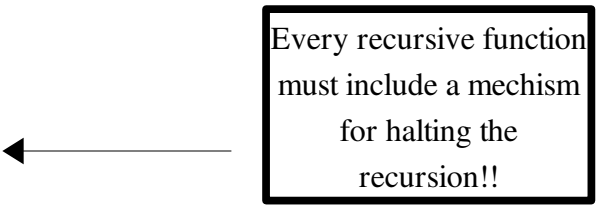
Specifically, when confronted with a multiple alternative path type problem recursion is often a useful approach.

A simple, but not useful, application

The "traditional" example of recursion involves computing mathematical recurrences, commonly the factorial function. The factorial (and other recurrences) are trivial to calculate in a single *for* loop in a way that is much more CPU and memory efficient!

Nevertheless, the simple example provides a useful first step in understanding the approach.

```
#include <stdio.h>

int fact(
int val)
{
   if (val == 1)
      return(1);

   val = val * fact(val - 1);
   return(val);
}

int main()
{
   int fv;

   fv = fact(5);
   printf("%d \n", fv);
}
```



If the input value is 1, then *fact* returns 1. Otherwise *fact()* invokes itself and multiplies the value returned by the input value.

The crucial aspect of the procedure is that *no multplications occur* until 1 is returned.

As described in CPSC 101 each time a function is invoked its parameters and local variables are allocated on the stack. Thus by the time the *return(1)* is reach there are 5 copies of the parameter *val* on the stack. The values returned by the *return(val);* are 2, 6, 24, 120

A version that permits better insight

Because recursive algorithms are more subtle than iterative ones, it is always a good idea to instrument them so that a better understanding of what is transpiring may be obtained. *NEVER* hesitate to introduce new local variables that can help you see how the computation is progressing... as Professor Brooks said in another context "you will anyway".

```
#include <stdio.h>

int fact(
int val)
{
   int newval;
   int retval;

   if (val == 1)
      return(1);

   newval = fact(val - 1);
   retval = val * newval;

   printf("val = %d newval = %d retval = %d \n",
           val, newval, retval);

   return(retval);
}

int main()
{
   int fv;

   fv = fact(5);
   printf("%d \n", fv);
}
```

```
acad/cs102/examples/fact ==> a.out
val = 2 newval = 1 retval = 2
val = 3 newval = 2 retval = 6
val = 4 newval = 6 retval = 24
val = 5 newval = 24 retval = 120
120
```

The proper way to compute a mathematical recurrence

The easy and efficient way requires only a single simple loop! So don't go looking for ways to use recursion unnecessarily!

```
#include <stdio.h>

int main()
{
   int fv;
   int i;

   fv = 1;
   for (i = 2; i <= 5; i++)
      fv = fv * i;

   printf("%d \n", fv);
}
```

A more challenging problem

Many problems that are quite easy to state and easy to solve manually can be very difficult to solve with a computer program. Here is one:

Write all the sequences of the first n letters of the alphabet taken k letters at a time.

It is well know that the number of combinations of *n* items taken *k* at a time is *n! / ((n – k)! k!)*

For example let *n = 5* and *k = 3*. Here the number of combinations is 5!/ (2! 3!) = (5 x 4) / 2 = 10

Being clever humans we can easily enumerate them:

We start by writing all the different combinations that start with *ab*
abc
abd
abe

Then we replace the *b* with *c* and add
acd
ace

Next *d* replaces *c* and we get
ade

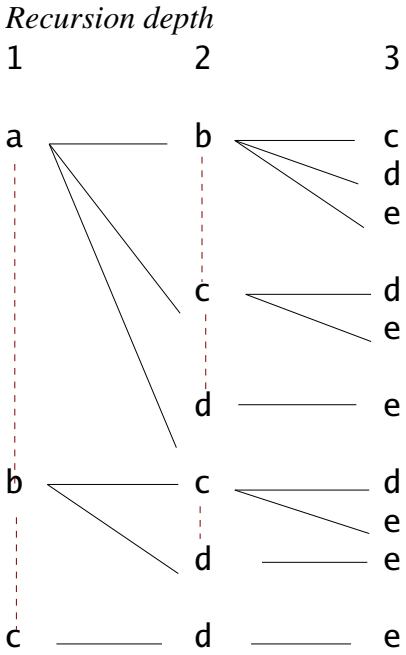
That is all there are that contain *a* so now we start with *b*
bcd
bce

then
bde

which leaves us with
cde

Exercise: Try to write a program that will generate these sequences one letter at a time given *n=5* and *k = 3*.

A graphical view of the solution



Lines shown in - - - - - - - - - - - denote operations done within a loop in the context of a single invocation of the recursive functions.

Lines shown in ———————— denote a recursive call. Information that must be passed in the recursive call includes the current length of the string and the base character that the lower layer must start with. When the current length == *k*, the recursion is finished.

The recursive solution

```
#include <stdio.h>

/* build string here */

char string[27];
char *alpha = "abcdefghijklmnopqrstuvwxyz";

int n;   // total number of characters to work with
int k;   // length of each string

int combos(
int base,     // index in current string
int len)      // length of current string
{
   int left = k - len;  // remaining items to add

   if (left == 0)
   {
      printf("%s\n", string);
      return(0);
   }
```

```
/* A single activation of combos will store a character */
/* only in one spot in the string being constructed.    */
/* That spot is given by len.. However the character    */
/* it actually stores in its spot will move one spot    */
/* down the alphabet for each iteration of the loop.    */

   while ((base + left) <= n)
   {
      string[len] = alpha[base];
      string[len + 1] = 0;
      fprintf(stderr, "%3d   %3d   %3d   %3d  %s \n",
                      len + 1, base, len, left, string);
      combos(base + 1, len + 1);
      base = base + 1;
   }
   return(0);
}
```

```
Depth  Base  Len   Left   String
  1     0     0     3     a
  2     1     1     2     ab
  3     2     2     1     abc
  3     3     2     1     abd
  3     4     2     1     abe
  2     2     1     2     ac
  3     3     2     1     acd
  3     4     2     1     ace
  2     3     1     2     ad
  3     4     2     1     ade
  1     1     0     3     b
  2     2     1     2     bc
  3     3     2     1     bcd
  3     4     2     1     bce
  2     3     1     2     bd
  3     4     2     1     bde
  1     2     0     3     c
  2     3     1     2     cd
  3     4     2     1     cde
```

```
int main()
{

   string[k] = 0;
   fscanf(stdin, "%d %d", &n ,&k);
   combos(0, 0);
}

acad/cs102/examples/combos ==> time a.out | wc -l
26 13
10400600

real    0m5.627s
user    0m1.376s
sys     0m0.152s

int combos(
int base,     // index in current string
int len)      // length of current string
{
   int left = k - len;  // remaining items to add

   while ((base + left) <= n)
   {
      string[len] = alpha[base];
      if (len == (k - 1))
          printf("%s\n", string);
      else
         combos(base + 1, len + 1);
      base = base + 1;

   }
   return(0);
}
```

A maze problem

Consider the following two dimensional array

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

We can view this as a maze in which passage through locations have a value of 0 is permitted but passage through locations having a value of 1 is not. We add the additional constraint that movement is permitted between adjacent locations in the array in *only vertical and horizontal (not diagonal)* directions. As is conventional, the value *maze[0][4] represents row 0 column 4*.

Our mission will be given a start (row, col) and a target (row, col), print a path (if one exists) beween the start and the finish. For the maze shown here given a start (0, 0) and a target (5, 5) the path is:

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

Sample input and output

The input will be given in the following format:

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

Maze data. We will always work with 6 x 6 arrays

```
0 0
```

start (row, col)

```
5 5
```

target (row, col)

The path should be printed in the following format. The first 0 (0, 0)

```
 1 (0, 1 )
 2 (0, 2 )
 3 (0, 3 )
 4 (0, 4 )
 5 (0, 5 )
 6 (1, 5 )
 7 (2, 5 )
 8 (2, 4 )
 9 (2, 3 )
10 (2, 2 )
11 (2, 1 )
12 (2, 0 )
13 (3, 0 )
14 (4, 0 )
15 (4, 1 )
16 (4, 2 )
17 (4, 3 )
18 (5, 3 )
19 (5, 4 )
20 (5, 5 )
```

```
/* 2-d maze search program */

#include <iostream>
using namespace std;

#define X_DIM 6
#define Y_DIM 6

struct coord_type
{
   int y;
   int x;
}  path [Y_DIM * X_DIM];

int maze[Y_DIM][X_DIM];
int visited[Y_DIM][X_DIM];

int starty, startx;      // (y, x) start coordinates
int targety, targetx;    // (y, x) target coordinates
int truelen;             // length of the final path

void read_maze(
void)
{
   int i = 0;
   int *loc = maze[0];

   while (i < X_DIM * Y_DIM)
   {
      cin >> *loc;
      loc += 1;
      i += 1;
   }
   cin >> starty  >> startx;
   cin >> targety >> targetx;
}
```

Because recursion involves a considerable amount of function call/return activity, overhead can be minimized considerably by making global those variables not required for the recursion to work.

The *legal_move()* function

Since we have to evaluate the possibilty of moving East, South, West, and North, it is best to write a single function that will return *true* if a potential move is legal and *false* if it is not.

```
int legal_move(
int desty,     // potential destination for next step
int destx)
{
```

if ((desty, destx) is

- *in the maze and*
- *hasn't been previously visited*
- *has a value of 0*
- `return(1);`

else

```
  return(0);  // false -> not legal move

}
```

The *build_path()* function

The *build_path()* function is the recursive function that actually solves the problem. The variable *pathlen* carries the current *depth* of the recursion which is also the current length of the *path.*

Because these algorithms are very subtle it is even for even an experience *gdb* user to have trouble keeping track what is going on and going wrong. Thus it is a good idea to put diagnostic prints at all entry and exit points of the recursive routine.

Return values are key to making this work. The function must return 0 if

- it discovers that the end of the path is reached or
- it receives a 0 return from a recursive call.

It must return -1 if

- it reaches a point where all moves are illegal
- it is returned -1 all attempts to make legal moves

```
int build_path(
int pathlen,     // current length of path
int y,           // (y, x) coordinates of this location
int x)
{
     int rc = -1;
    remember this location has been visited
     fprintf(stderr, "visiting %d %d with pathlen %d \n", y, x,
                      pathlen);
     if this location is the target
     {
       fprintf(stderr, "found the target at %d %d \n", y, x);
       remember current pathlen in truelen
       store this location in the path
        return(0);
     }
```

```
/* Haven't reached the target so need to press onward */
```

try to build_path() east;
if that doesn't work try to build_path() south;
if that doesn't work try to build_path() west;
if that doesn't work try to build_path() north;

Each of these is a potential recursive call depending upon whether east, south, west, or north is a legal move.

If the target was found, then *build_path()* returns a non-negative number and the search is over. *Note that the path is built backward as the recursion unwinds.*

```
    {
       fprintf(stderr, "adding point %d %d with pathlen %d \n",
                                y, x, pathlen);
        store this location in  the path
       return(0);

    }

/* Hit a dead end... back up one spot */

   if nothing worked
   {
     fprintf(stderr, "stuck at %d %d backing up \n", y, x);
     return(-1)
}

int main()
{
   int pathlen = 0;

   read_maze();

   pathlen = build_path(pathlen, starty, startx);
   if (pathlen)
      print_path(truelen);
   else
      printf("can't get there from here! \n");

}
```

```
0 0 0 0 0 0
1 1 0 1 1 1
1 0 0 0 0 0
0 0 1 1 1 1
0 1 0 0 0 1
0 0 0 1 0 0

5 4
0 5

visiting 5 4 with pathlen 0
visiting 5 5 with pathlen 1
stuck at 5 5 backing up
visiting 4 4 with pathlen 1
visiting 4 3 with pathlen 2
visiting 4 2 with pathlen 3
visiting 5 2 with pathlen 4
visiting 5 1 with pathlen 5
visiting 5 0 with pathlen 6
visiting 4 0 with pathlen 7
visiting 3 0 with pathlen 8
visiting 3 1 with pathlen 9
visiting 2 1 with pathlen 10
visiting 2 2 with pathlen 11
visiting 2 3 with pathlen 12
visiting 2 4 with pathlen 13
visiting 2 5 with pathlen 14
stuck at 2 5 backing up
stuck at 2 4 backing up
stuck at 2 3 backing up
visiting 1 2 with pathlen 12
visiting 0 2 with pathlen 13
visiting 0 3 with pathlen 14
visiting 0 4 with pathlen 15
visiting 0 5 with pathlen 16
found the target at 0 5
```

Specular lighting

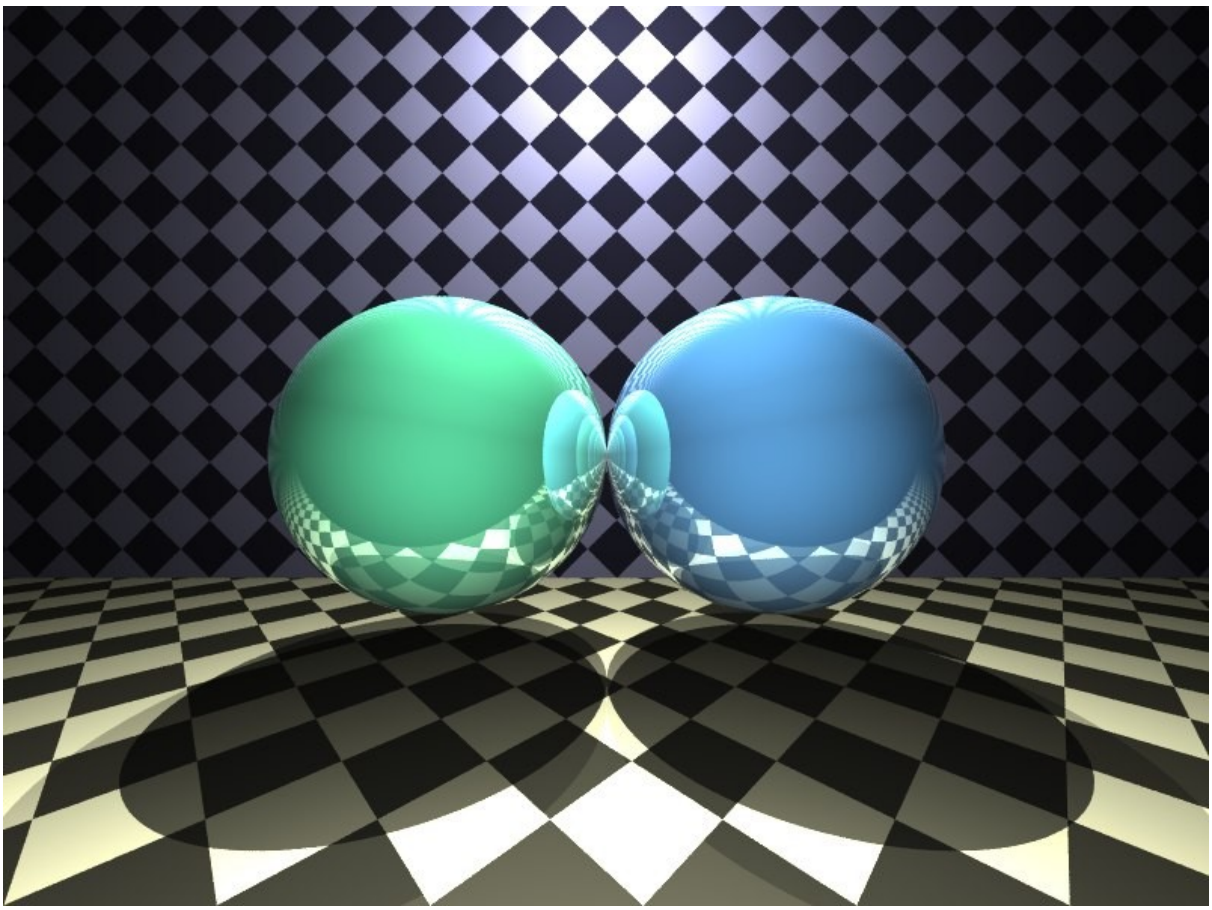
Specular light is which is coherently reflected *without scattering*. The best example of an object with no ambient or diffuse reflectivity but high specular reflectivity is a mirror.

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:

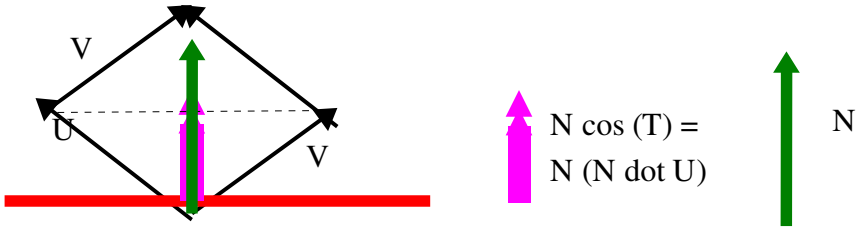
hits no object
hits an object with no specular reflectivity.
travels so far that the effect of further bounces is negligible



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

```
vec_reflect(
vec_t *unitin,      /* unit vector in incoming direction of the ray      */
vec_t *unitnorm,    /* outward surface normal                            */
vec_t *unitout);    /* unit vector in outgoing direction reflected ray   */
```



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:

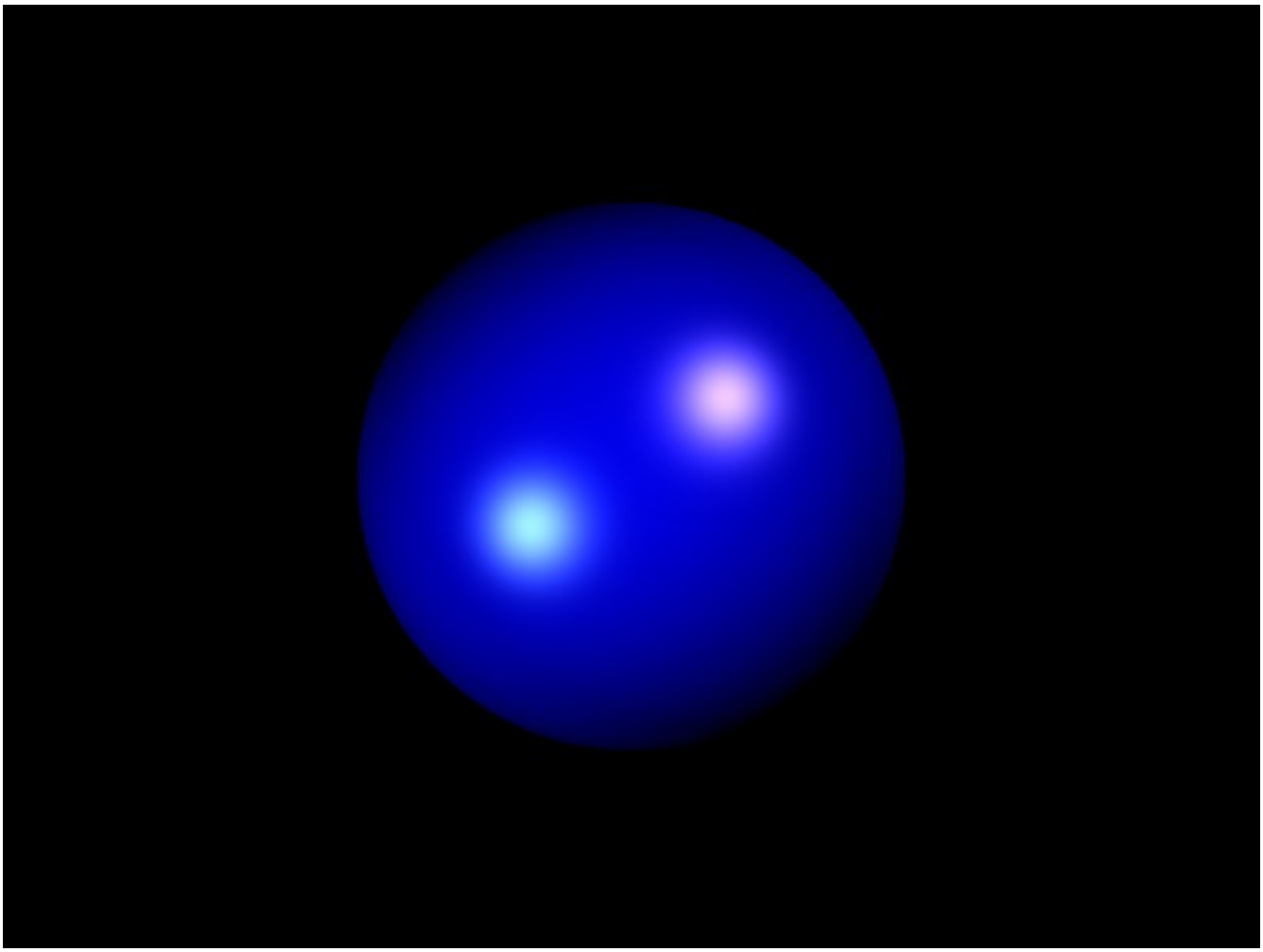
```
void ray_trace(
model_t  *model,
vec_t    *base,         /* location of viewer or previous hit */
vec_t    *dir,          /* unit vector in direction of object */
drgb_t   *pix,          /* pixel     return location          */
double   total_dist,    /* distance ray has traveled so far   */
object_t *last_hit)     /* most recently hit object           */
{
   object_t *closest;
   drgb_t   specref = {0.0, 0.0, 0.0};

   double   mindist;
   drgb_t   thisray = {0.0, 0.0, 0.0};

   if (total_dist > MAX_DIST)
       return;

      Find the closest object that the ray hits, and if there is a hit:
            Add the distance from base of the ray to the hit point to total_dist
            and do ambient and diffuse lighting as before (dividing by total_dist)

      closest->getspec(specref);   /* see if object has specular reflectivity */
      if (specref is not 0)
      {
            drgb_t specint = {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 specint
            add specint to thisray
      }
      add thisray to *pix
}
```

The *hitloc* lies on the surface of *closest.* We can't allow another hit at distance 0

Specular glints

Glints are another form of specular reflection. A glint occurs when a *specific light* is reflected from the surface to the *viewpoint.* In the following example we see glints produced by a greenish light in the lower left and a redish light at the upper right.

The center of the glint is the location at which a ray arriving from the light is reflected about the normal directly into the eye of the viewer. We will provide a model that allows us to control how tightly the reflection is focused.



Modifications to ray tracing data structures

The only modification necessary is the addition of the *shininess* exponent to the *material_t*. It is a single double precision value and must be added to the parser and dumper of the *material_t*. If the shininess value is not specified it should be set to 0.0. The *material_getshine()* function retrieves the *shininess* value.

```
friend material_t *material_find(model_t *, char *);

public:
   material_t(){};
   material_t(FILE *in, model_t *model, int attrmax);
   void  material_getamb(drgb_t *dest);
   void  material_getdiff(drgb_t *dest);
   void  material_getspec(drgb_t *dest);
   void  material_getshine(double *shiny);
   char *material_getname();
   inline void material_item_dump(FILE *out);

private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;         /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
   double  shininess;
};
```

For consistency and to permit polymorphic behavior it is also a good idea to implement a corresponding *getshine()* method in the *object_t*.

```
/* Optional plugins for retrieval of reflectivity */
/* useful for the ever-popular tiled floor        */

   virtual    void    getamb(drgb_t *);
   virtual    void    getdiff(drgb_t *);
   virtual    void    getspec(drgb_t *);
   virtual    void    getshine(double *);
```

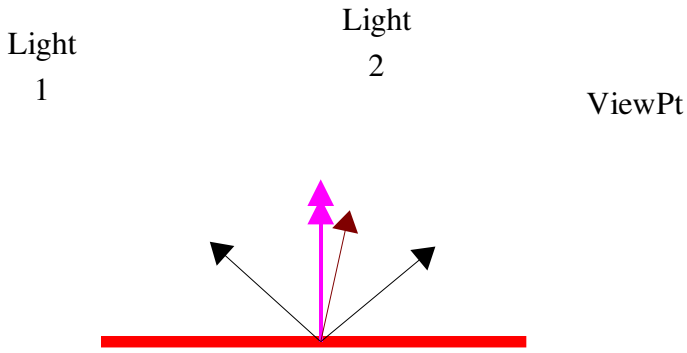
A sample input file

The input file used to produce the blue sphere with the two glints is shown below. The *larger* the value of *shininess* the *more focused (smaller)* the glint will be. Visually realistic values tend to be large.

```
camera cam1
{
  pixeldim  800 600
  worlddim  8 6
  viewpoint 4 3 4
}

material blue
{
   diffuse  0 0 8
   specular 1 1 1
   shininess  50.0
}

light pinkfront
{
   emissivity 6 5 5
   location 10 8 4
}

light cyanfront
{
   emissivity 4 6 5
   location -2 0  4
}

sphere shadowmaker
{
   material blue
   center  4 3 -6
   radius  4
}
```

The model for producing glints

The model is closely related to the one underlying specular reflection. Because the angle of incidence is equal to the angle of reflection, an incoming light ray from *light 1* will be directed toward the view point. In contrast, an incoming ray from light 2 will be reflected on the other side of the surface normal from the ViewPt.



Notice that if the light is reflected directly toward the view point, then the *sum of the vector pointing toward the light and the vector pointing toward the viewpoint (more precisely the source of the ray!) is a vector that is perfectly aligned with the surface normal.*

Therefore the algorithm works as follows:

- Compute a *unit* vector from the hitpoint to the light.
- Compute a *unit* vector from the hitpoint to the source of the ray.
- Take the sum of these two vectors and convert the sum to a *unit* vector.
- Compute the dot product of the *unit* sum with the *unit* normal at the hitpoint. This is our base measure of how close the line to the viewpoint is to the actual direction the light is reflected.
- Raise the dot product to the power of *shininess.* Since the dot product is <= 1.0, *raising it to a large power will tend to reduce it.* This is why a large value of shininess produces a more focused glint.
- Compute the componentwise product of the *emissivity* of the light with the *specular* reflectivity of the hit object's material.
- Scale this value by the dot product raised to the power of *shininess*. Use the *pow()* function here.
- Add the scaled value to *pixel.*

Where does the glint code go?

The *glint* effect must be computed for *every hitpoint and every light.* This is also true of diffuse illumination. Thus, the most reasonable place to put it is:

- at the end of the illuminate method
- or (better still) in a new method that is invoked just after *illuminate().*

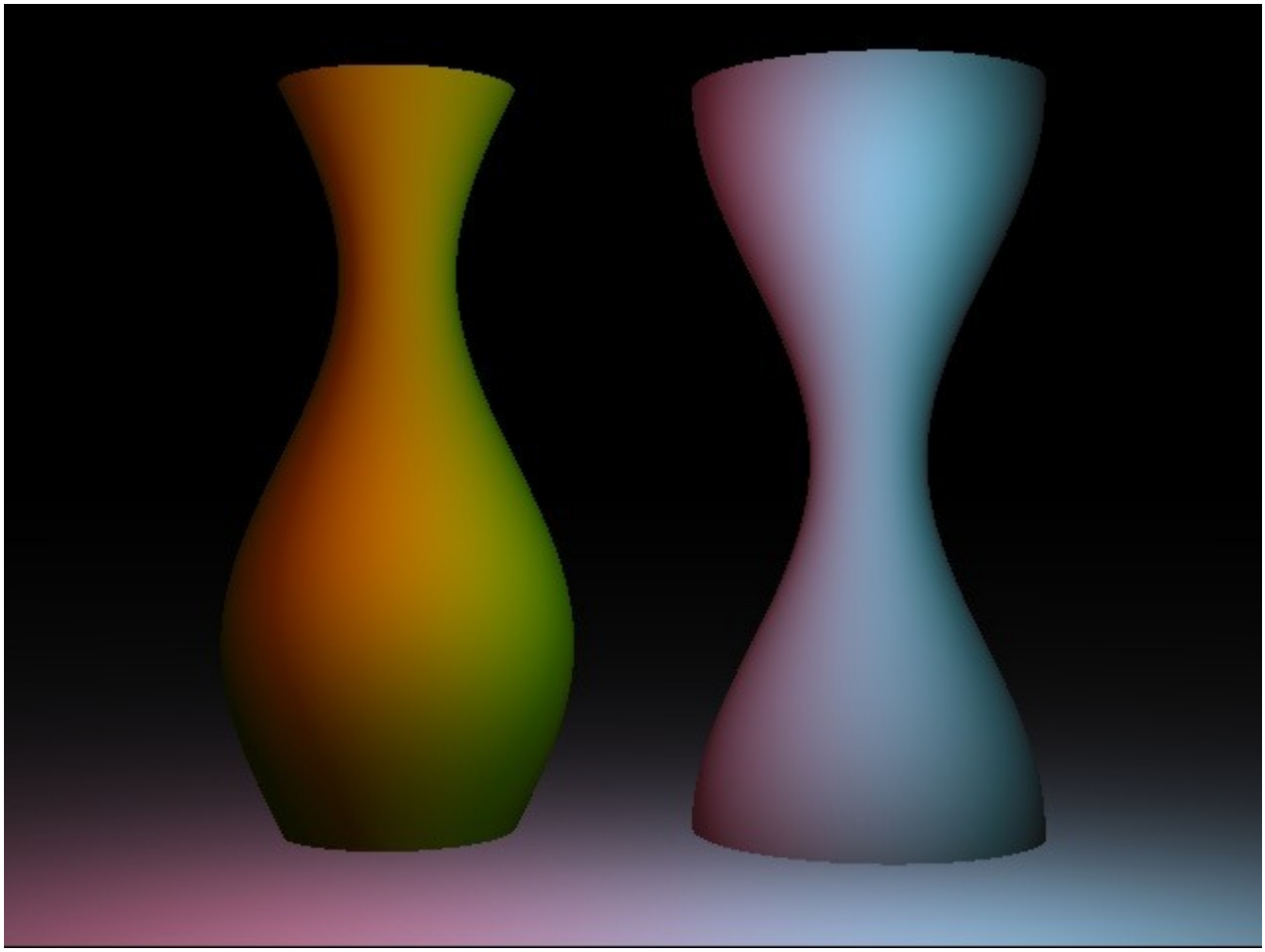
(remember NOT to do the computation if the *shininess* of the material is 0).

How do we access the *base* of the ray??

We must pass it from raytrace, through add diffuse to illuminate :-(

Surfaces of revolution

Surfaces of revolution greatly expand the range of shapes that we can easily create. The objects shown below are surfaces of revolutions created using *sine* and *cosine* functions. The object on the left is gold in color and the object on the right is gray. The scene is illuminated by three lights. The one on the left has a red tint, the one in the center has a green tint, and the one on the right blue.



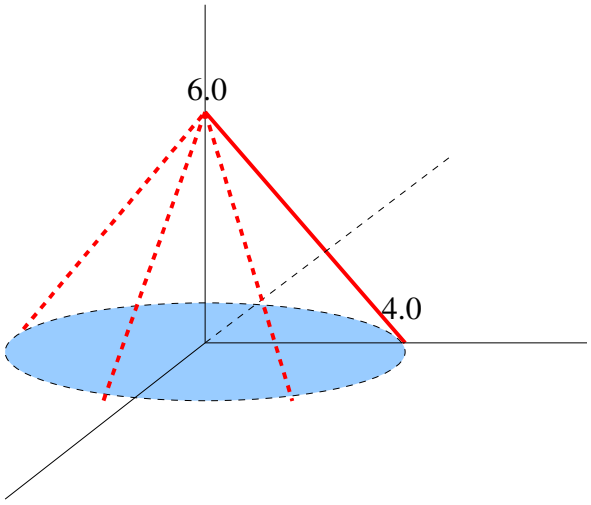
An intuitive motivation

Consider the red line in the diagram below. When $x = 0$, $y = 6.0$ and when $x = 4.0$, $y = 0$. Thus the slope of the line is -1.5 and equation of the red line is given by:

$$y = -6x/4 + 6$$

For reasons that will become apparent we typically express x as a function of y. It is simple to solve the equation for x.

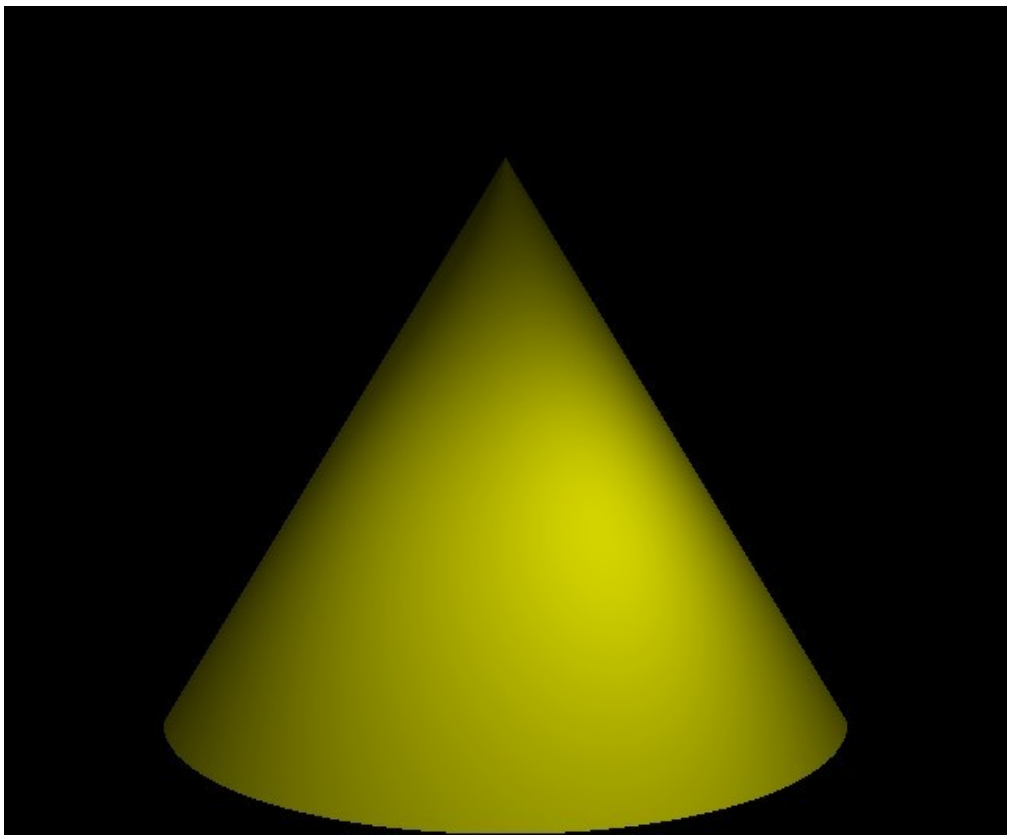
$$x = -4/6\ (y - 6)$$



The term *surface of revolution* is used to describe the surface that is created when the line is dragged around the blue circle (which appears as an ellipse when it is projected upon the "floor"). The surface that is created is a *cone* with vertex at location (0.0, 6.0, 0.0).

Example input and output (*light definitions missing*)

```
camera cam1
{
  pixeldim  640 480
  worlddim  8 6
  viewpoint 4 3 7
}
material gold
{
   ambient 1 1 0
   diffuse 3 3 0
}
revsurf cone
{
   material gold
   surfer 0
   base 4 0 -4
   direction 0 1 0
   height 6
}
```



Determining an equation for the surface

The first step in developing a *hits* function is to develop an equation that describes the surface. Consider the circle that is produced by dragging the point $x = 4, y = 0$ around the y axis. This operation produces a circle of radius 4 in the $y = 0$ plane. That is, x *and* z vary but y does not. The equation of this circle is:

$$x^2 + z^2 = 4^2$$

As we move along the red line, the distance of the points along the line from the y axis are given by:

$$x = -4/6\ (y - 6)$$

Thus any point on the cone must satisfy the equation:

$$x^2 + z^2 = (-4/6\ (y - 6))^2$$

Which may be written as:

$$f(x, y, z) = x^2 + z^2 - (-4/6\ (y - 6))^2 = 0$$

This is a quadratic equation. If V is the viewpoint and D the ray direction We can replace (x, y, z) by $v_x + td_x \quad v_y + td_y \; v_z + td_z$ and solve for t in the usual way. However, we don't want to restrict ourselves to quadratic equations. And we definitely don't want to be limited to surfaces that resolve around the origin.

More complex surfaces

For example the gold vase shown earlier was produced by revolving the line:

$$x = 2 + sin(y)$$

Applying the approach of the previous page we see the equation of this surface is:

$$f(x, y, z) = x^2 + z^2 - (2 + sin(y))^2 = 0$$

In general given the equation of any line in the form

$$x = g(y)$$

The equation of the corresponding surface of revolution is:

$$f(x, y, z) = x^2 + z^2 - (g(y))^2 = 0$$

For points OUTSIDE the surface $x^2 + z^2 > (g(y))^2$ and so f(x, y, z) > 0. For points inside the surface the reverse is true and f(x, y, z) < 0. A point lies on the surface if and only if f(x, y, z) = 0.

The normal at the hit point is given by the componentwise (partial) derivatives of the function or:

$$(2x, -2(g(y))g'(y), 2z)$$

Thus a surface normal on the gold vase at location (x, y, z) is

$$(2x, -2(2 + sin(y)) cos(y), 2z)$$

Determining if a ray hits a surface of revolution.

The "front end" of our approach
Assume the following:

V = *viewpoint or start of the ray*

D = *a unit vector in the direction the ray is traveling*

C = *base point (y = 0) of the surface*

h = *desired height of the surface of revolution.*

$g(y)$ = *the generating function*

The arithmetic is much simpler if the base point of the surface is at the origin (as it was in the preceding examples). So like we did with the sphere, we start by moving it there! To do so we must make a compensating adjustment to the base of the ray.

$C' = C - C = (0, 0, 0) =$ *new center of sphere*

$V' = V - C$ = *new base of ray*

D *does not change*

A point P on the translated surface with base at (0, 0, 0) necessarily satisfies the following equation:

$$p_x^2 - g(p_y)^2 + p_z^2 = 0 \qquad (1)$$

All points on the ray may be expressed in the form

$$P = V' + t\,D = (v'_x + td_x,\ v'_y + td_y,\ v'_z + td_z) \qquad (2)$$

where t is the Euclidean distance from V' to P

Thus we need to find a value of t which yields a point that satisfies the two equations. To do that we take the (x, y, z) coordinates from equation (2) and plug them into equation (1). In this equation t is the only unknown quantity but unless $g(y)$ is a linear function of y (as was the case with the cone) we do not have a quadratic equation.

$$(v'_x + td_x)^2 - (g(v'_y + td_y))^2 + (v'_z + td_z)^2 = 0$$

Numerical solution of the *hits* equation.

Given an arbitrary function *q(t)* we need a method for finding a value of *t* for which *q(t) = 0.*
If we can do that, then we can define *q(t)* as follows and we are done.

$$q(t) = (v'_x + td_x)^2 - (g(v'_y + td_y))^2 + (v'_z + td_z)^2$$

This is actually a pretty difficult problem because in general there my be multiple values of *t.* For

$$q(t) = sin(t)$$

setting *t* any multiple of *pi* will produce *q(t) = 0.*

There may also be no values of *t* at all as in the case of

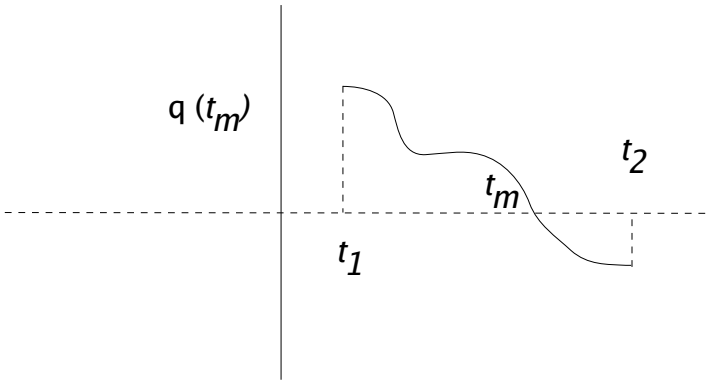
$$q(t) = t^2 + 1$$

where the minimum value of the function is *q(0) = 1.*

Nevertheless we can say that: (1) if *q(t)* is a continuous function *and* (2) we can find two points t_1 *and* t_2 such that $q(t_1)\, q(t_2) < 0$, then we can find a point t_r such that $q(t_r) \sim= 0.0$.

The method that we will use is based upon the *binary search* technique and is commonly called *bisection.*

The *bisection* algorithm



while(*t2 – t1) > epsilon)*
{
compute *tm* = the midpoint between *t2* and *t1*
if (q(t1) * q(tm)) < 0)
replace *t2* with *tm*
else
replace *t1* with *tm*
}

Determining the start and end points for bisection:

We begin by translating the base of the revsurf to the origin and then rotating it so that its centerline is aligned with the *y* axis. The rotation step can be omitted if we assume *all* objects are so aligned.

```
   vec_copy(raybase, &s_base);
   vec_diff(&base, &s_base, &s_base);
   vec_xform(&rot, &s_base, &s_base);

/* Also need to make a local copy of the direction */

   vec_copy(dir, &s_dir);
   vec_xform(&rot, &s_dir, &s_dir);
```

The value *t=0* will alwaysproduce a positive value of *q(t).* A value of *t* inside the surface will produce a negative one. How do we find such a value???

One obvious (but wrong) way to try to do this is to compute the distance to where the ray intersects the *z = 0* plane.

```
/* Here we need to find how far a ray fired from s_base       */
/* in direction s_dir needs to travel before reaching         */
/* the z = 0 plane.  Points on the ray are given by           */
/*         (b.x + t*d.x, b.y + t*d.y, b.z+t*d.z               */
/* Hence the "t" value that makes pt.z = 0 depends only on    */
/* b.z and d.z                                                */

    c = -s_base.z / s_dir.z;
```

Ooops #1

```
/* The above argument is not actually correct.. It is possible */
/* that if the object is located near the edge of visible      */
/* and also near the viewplane that a ray could pass through   */
/* the translated object but not enter it in positive z space  */
/* It also doesn't work AT ALL for specular and shadows...     */
/* :-(.  An alternative is to fire a ray long enough to reach  */
/* the origin.  In practice it seems to be quite difficult to  */
/* produce a scenario in which the different approaches        */
/* produce different images.                                   */

   d = vec_len(&s_base);
   c = d;
```

Ooops #2

```
/* Now I think what we REALLY want for max t is the distance   */
/* from the translated viewpoint to the centerline of the      */
/* translated object... Since the centerline is z = x = 0 this */
/* is easy to compute                                          */

   c = sqrt(s_base.z * s_base.z + s_base.x * s_base.x);
```

Oops#3 ... closer but not there yet

```
/* Not so fast... Also probably need y deflection in there */

   d = c * s_dir.y;          // compute y deflection
   c = sqrt(c * c + d * d);  // get length of hypotenuse
```

Success at last?

Our previous attempt included the *y* deflection but failed to include the x-z deflection. We can do the x-z deflection in first and then compute the *y* deflection as follows. This actually provides the correct answer!!!

```
   vec_copy(&s_dir, &dir_proj);
   dir_proj.y = 0;

/* s_base      y_axis (points out toward you)
     -------------
     \   vc_len  |
       \         |
         \       | target plane
 dir_proj  \     |
       dp_len\   |
               \ |

   vc_len = dp_len * cos(theta)

*/

   vec_unit(&dir_proj, &dir_proj);
   dp_len = vc_len / vec_dot(&view_to_cl, &dir_proj);

/* Finally we account for the y deflection  */

   c = dp_len / vec_dot(&dir_proj, &s_dir);
```

A more straightforward way to the answer

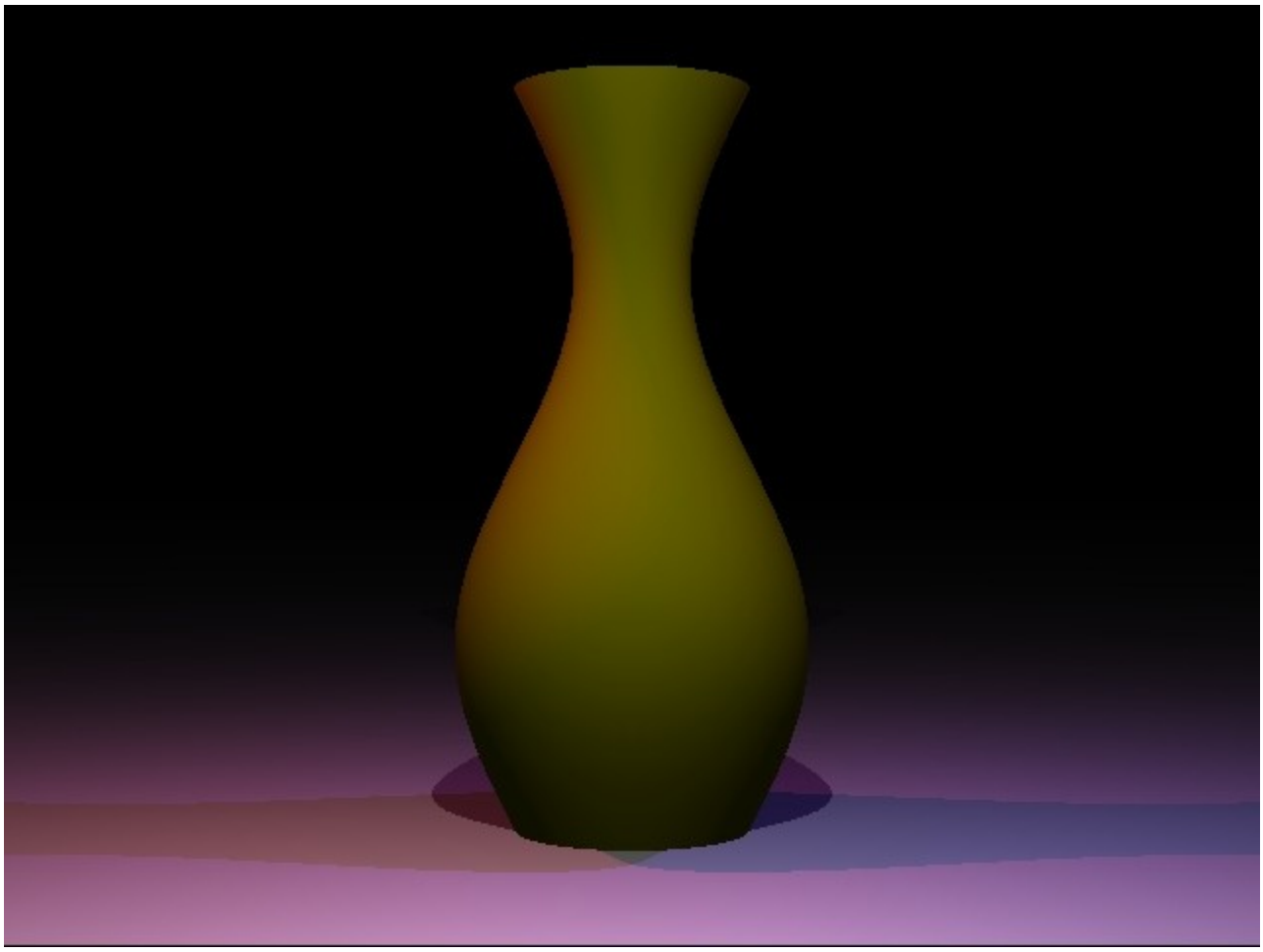
We began this nightmarish quest by trying to find where the ray intersected a plane. That was actually the correct way to look at the problem. We just used the *wrong plane*. The correct plane is one that contains the *y axis* (just like z = 0 does) but whose normal is the reverse of the shortest line from the viewpoint to the *y-axis*.

```
/* Now in retrospect all of that was an unnecessary effort */
/* When we think about this problem correctly, it reduces  */
/* to a hits plane problem in the following way.           */

/* Somewhat like before draw the shortest line connecting  */
/* the y-axis to s_base. Now consider the plane for which  */
/* that line is the normal.  Thus the normal is            */
/* (s_base.x, 0, s_base.z) and the pt is (0, s_base.y, 0)  */

/* We seek the distance to where a ray fired in direction  */
/* s_dir hits the plane.  Recall the standard plane hits   */
/* equation   where q is the point on the plane, b is the  */
/* base of the ray, n is the normal and d is the direction */

/* t = (ndotq - ndotb) / ndotd;

   here q =  (0,         0,        0)
        n =  (s_base.x,  0,        s_base.z)
        b =  (s_base.x,  sbase_y,  s_base.z)
        d =  (s_dir.x,   sdir_y,   s_dir.z)

   and ndotq  == 0  so  t = - ndotb / ndotd
*/
```

One might be tempted to try to craft a "bogo" plane_t object and just use the plane_t hits function. While that could possibly be made to work, it would be a bad idea. This code must execute for every ray fired. *We don't want to be created and deleting 100,000,000 plane_t objects.* So instead we just include the guts of the *hits* code.

```
   vec_copy(&s_base, &normal);
    normal.y = 0;
    vec_unit(&normal, &normal);
    c = -vec_dot(&normal, &s_base) / vec_dot(&normal, &s_dir);
```

This produces somewhat more believable shadows than the previous plane.



Finding the hitpoint:

Your *bisect()* routine *must* test for the condition in which both initial values of the function lie on the same side of the x axis and return -1 if that is true. This will be the case if the ray completely misses the *revsurf.*

```
   iflag = bisect(evaluators[surfer], &b, &c, ae);
   if (iflag != 0)
      return(-1);

/* Determine the hitloc in translated rotated space */
```

The value *b* is the distance to the hitpoint in translated rotated space (with the base of the object at the origin and the centerline on the y axis. Thus if we scale *s_dir* by *b* and add it to *s_base* we obtain the location of the hitpoint in translated rotated space. We call this *xlhit.*

Now its necessary to make sure we are within the vertical limits of the object.

```
    if (xlhit.y < 0)
      return(-1);
   if (xlhit.y > height)
      return(-1);
```

If within the limits we have a hit!

Ask the correct normer to supply the normal at the hitpoint.

```
   (*normers[surfer])(&xlhit, &hitnorm);
```

- Rotate the normal by *irot* if necessary
- Rotate *xlhit* by *irot* if necessary
- Translate *xlhit* by adding the base location of the object.

Evaluators and normers

The easiest way to access these is by a table of function pointers in which the attribute *surfer (0, 1,..)* is used as in index into the table.

```
 iflag = bisect(evaluators[surfer], &b, &c, abserr);

static double (*evaluators[])(double t) =
{
   f0,
   f1,
   f2,
   f3,
   f4,
};
#define NUM_SURFS (sizeof(evaluators) / sizeof(double *))


static void (*normers[])(vec_t *, vec_t *) =
{
   n0,
   n1,
   n2,
   n3,
   n4

};
```

An evaluator for the cylinder

The *init_sum* function computes the common part of the general equation which involves the x and z terms .

```
/* revsurfaces all have the same general equation */
/* x**2 + z**2 - f(y) = 0                         */
/* this piece computes the common part            */

static inline double init_sum(
double t)
{
   double v;
   double sum = 0.0;

/* Compute x-coord for given "t" and square it */

   v  = s_base.x + t * s_dir.x;
   v *= v;
   sum += v;

/* Add squared z-coord for given "t" */

   v  = s_base.z + t * s_dir.z;
   v *= v;
   sum += v;

   return(sum);
}
```

The evaluators *f0, f1,* ... use add in the y specific part after calling *init_sum.*

The equation for a cylinder is *sqrt(x^2 + z^2) = r.* Thus our *g(y)* function is *g(y) = r,* and our *f(x, y, z) = x^2 + z^2 – g^2(y) = x^2 + z^2 – r^2*

```
/* The mission of this function is to evaluate   */
/* f(x, y, z) = x**2 + z**2 - g(y)**2            */

/* The function computed here is:           */
/*    x**2 + z**2 -  p0^2                   */
```

The f0 function only computes the y-component of the sum which is $-r^2$ where *r* is the radius. The variable *p0* the value of the attribute *surferp0.* This parameter allows us to compute cylinders of different radii with a single *f0* evaluator.

```
double f0(
double t)  /* A t value in the search range */
{
   double sum = 0.0;    // f(x, y, z)
   double v;

   sum = init_sum(t);

   v = p0;
   v *= v;

   sum -= v;
   return(sum);
}
```

Where does *p0* come from??? We get a bit more flexibility if we make things like the radius of our cylinder parameterizable.

```
static pparm_t revsurf_parse[] =
{
   {"surfer",    1, 4, "%d",  0},
   {"base",      3, 8, "%lf", 0},
   {"height",    1, 8, "%lf", 0},
   {"direction", 3, 8, "%lf", 0},
   {"surferp0",  1, 8, "%lf", 0},
   {"surferp1",  1, 8, "%lf", 0}
};
```

The *normer* function.

```
/* In the event there is a hit the normal at the hit  */
/* which is the del f(x,y,z) must be compute          */
/* The hitloc value passed in MUST be the hitloc      */
/* WHEN THE base is translated to the origin          */

/* The function computed here is:                     */
/*    x**2 + z**2 – r**2  So the component wise       */
/*  derivatives are:
/*    2x, 0, 2z                                       */

void   n0(
vec_t  *hitloc,
vec_t  *normloc)
{
   normloc->x = 2 * hitloc->x;
   normloc->z = 2 * hitloc->z;
   normloc->y = 0;
   vec_unit(normloc, normloc);
}
```

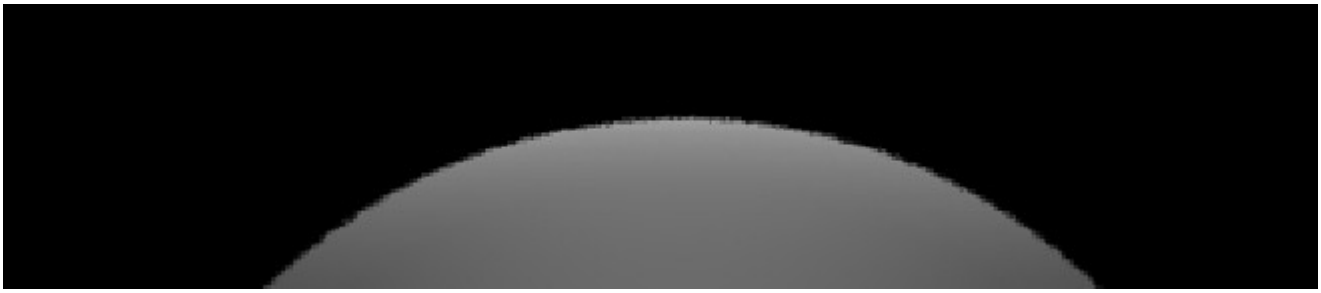
Antialiasing with sub-pixel sampling.

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 value actually stored in the image is the *average* of all the pixel values computed.

Magnified view without antialiasing

Magnified veiw with 16 pixel averaging



Jittering the ray direction

The easiest way to jitter the direction of the ray is to jitter the coordinates of the pixel through which it passess. This is easy to do

```
/**/
void camera_t::camera_getdir(
int      x,
int      y,
vec_t   *dir)
{
   vec_t world;
   double dx = x;
   double dy = y;

   if (AA_SAMPLES > 1)
   {
      dx = randomize(dx);
      dy = randomize(dy);
   }

/* Compute direction using (dx, dy)  */
```

Can't jitter an int!!

The *randomize()* function

Most systems provide library functions that generate streams of *pseudo* random numbers. The *random()* function is the one we will use. It returns an integer value between 0 and 0x7fffffff. To perform pixel randomization the value must be converted to a *double* in the range [-0.5, 0.5] which is then added to the input pixel coordinate. In the transformations, *DO NOT USE THE SYMBOL* RAND_MAX. Be sure that your randomized pixel locations are >0 and < pixel_dim

Modifications to *image_create()*

No modifications are necessary to the *make_pixel()* function provided in the baseline C++ sample code. But for this to work, it is necessary that *ray_trace()* *add to and not set* the *contents of the pixel pointer* passed to it.

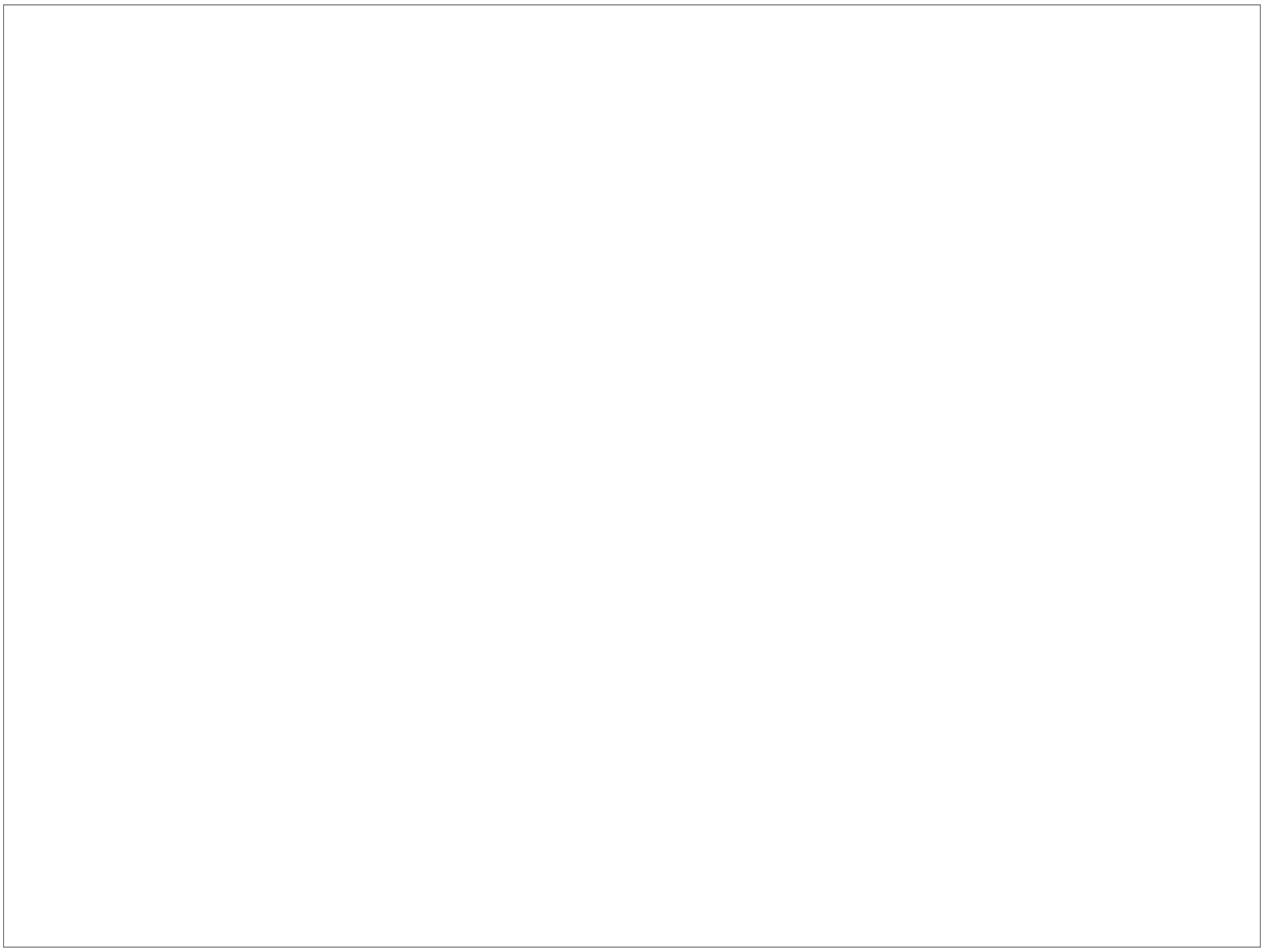
```
static inline void make_pixel(
model_t *model,
int      x,
int      y)
{
   vec_t  raydir;
   vec_t  viewpt;
   drgb_t pix = {0.0, 0.0, 0.0};
   camera_t  *cam = model->cam;
   int    i;

   cam->camera_getviewpt(&viewpt);
   for (i = 0; i < AA_SAMPLES; i++)
   {
      cam->camera_getdir(x, y, &raydir);
      ray_trace(model, &viewpt, &raydir, &pix, 0.0, NULL);
   }

   cam->camera_setpix(x, y, &pix);
   return;
}
```

If all of this is done properly then in *scale_and_clamp()* it suffices to *scale* the *drgb_t* by (255.0 / AA_SAMPLES)

Partial transparency

Partial transparency of objects is also easy to implement. An optional transparency factor is added to the *material_t*. Its default value is 0.0 which means not transparent at all.

```
private:
   int     cookie;
   char    name[NAME_LEN];
   drgb_t  ambient;      /* Reflectivity for materials  */
   drgb_t  diffuse;
   drgb_t  specular;
   double  shininess;
   double  transparency; /* 0.0 -> opaque : 1.0 invisible */
};
```



Modifications

The following modifications are needed to support transparency

- Material constructor must parse transparency attribute
- New *material_t::material_gettrans()* and *object_t::gettrans()* methods
- New recursive call to raytrace to pass the ray through the object
- Blending of object color with color returned by recursive call
- Modifications to *light_t::illuminate()* to implement *partial occlusion.*

In the image on the previous page this is the material definition for the gold vases.

```
material gold
{
   ambient 1 1 0
   diffuse 4 4 0
   transparency 0.2
   specular 0.4 0.4 0.4
}
```

This is the definition for the see through sphere.

```
material see_thru
{
   diffuse 4 4 4
   transparency 0.6
}
```

Refraction

The transparency model just presented does not account for refraction. Real objects that pass light bend it about the surface normal according to Snell's Law. The coefficients *n1* and *n2* are known as the

idices of refraction. The index for space is 1.0 and water is about 1.33.

As light enters a slow medium it bends toward the surface normal. As it exits the slow medium it bends away from the surface normal.

The most straightfoward way to compute the direction of the refracted way is to:

- Build a matrix that rotates the surface normal into the y-axis and the incoming ray into the x-y plane. This can be done by rotating the cross product of the incoming ray and the surface normal into the *z* axis.
- Apply basic trigonometry to compute the direction of the refracted ray in the rotated coordinate system.
- Invert the matrix constructed in step 1.
- Apply the inverse matrix to the vector computed in step 2.

Note that when passing from a slow medium to a faster one, it is possible for *sin(theta2)* to exceed 1. If this occurs then the incoming ray has exceeded the *critical angle* and it will be reflected instead of refracted.