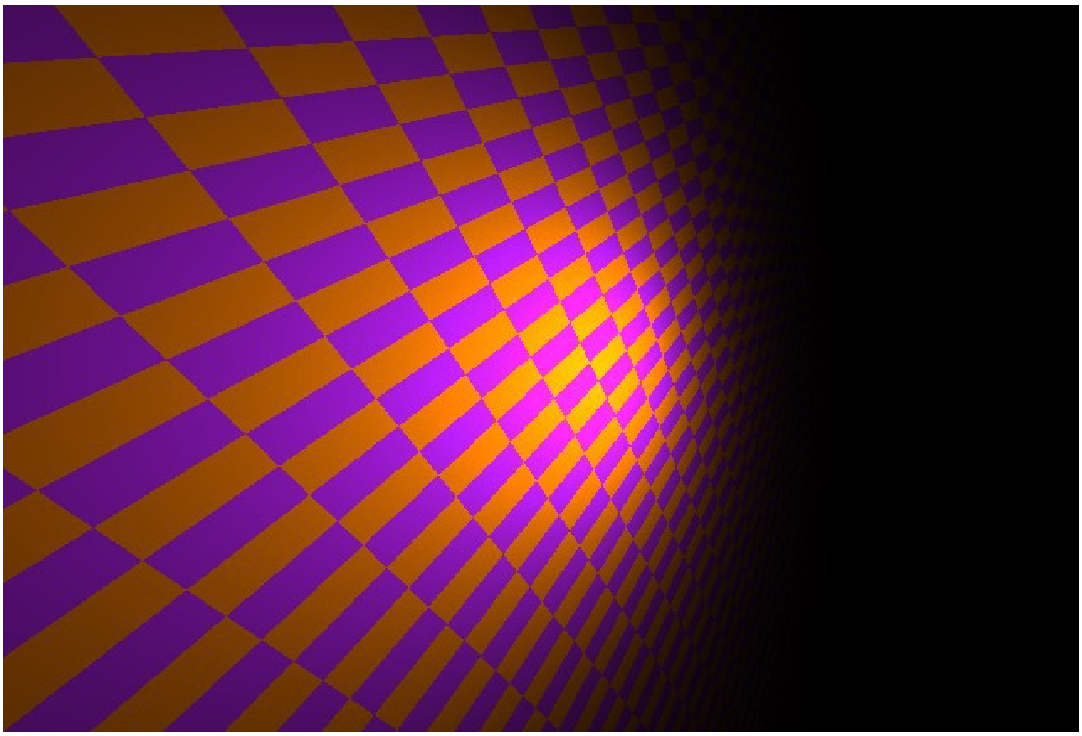
The tiled plane

The *tiled plane* is new object that has characteristics of both the infinite and finite planes. Like the original infinite plane it is of *unbounded size*. Thus, the *plane_hits()* function in your original *plane.c* can be used to determine where the object is hit.

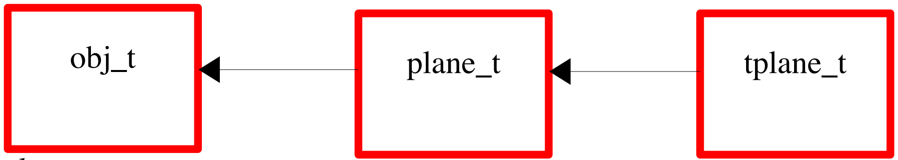
As can be observed, the plane is comprised of a collection of *tiles.* The tiles share characteristics of the *finite_plane*. The tiles have *x* and *y* dimensions and a vector which, when projected onto the plane determines the *x-axis* direction of the tiling.



Unlike both the basic plane and the finite plane, the tiled plane has *two sets of colors.* The *foreground* color may be stored as usual in the *material_t* component of the *obj_t* structure, but we need to save the *background* color in the *tplane_t* structure itself.

Implementation of the tiled plane

Like the finite plane the tiled planed is derived from the infinite plane:



The *tplane_t* structure

```
typedef struct tplane_type {
   double   xdir[3];         /* orientation of the tiling */
   double   size[2];         /* size of the tiling        */
   material_t background;    /* background color          */
} tplane_t;
```

Using the function pointers of the obj_t to provide polymorphic behavior

In the discussion of the procedural plane, it was shown how the *getamb(), getdiff(),* and *getspec()* functions that could optionally be overridden by various objects provided a useful way to emulate the polymorphism of a true object oriented language.

```
typedef struct obj_type {
   struct  obj_type  *next;   /* Next object in list         */
   int     objid;             /* Numeric serial # for debug */
   int     objtype;           /* Type code (14 -> Plane )    */

   double  (*hits)(double *base, double *dir, struct obj_type *);
                              /* Hits function.              */

   /*
    * Optional plugins for retrieval of reflectivity
    * useful for the ever-popular tiled floor
    */
   void    (*getamb)(struct obj_type *, double *);
   void    (*getdiff)(struct obj_type *, double *);
   void    (*getspec)(struct obj_type *, double *);

   material_t material;

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

   void    *priv;        /* Private type-dependent data */
   double  hitloc[3];    /* Last hit point              */
   double  normal[3];    /* Normal at hit point         */
}  obj_t;
```

Invoking the polymorphic methods from *raytrace()* and *illuminate()*

Recall that the main benefit of the polymorphic approach is that it allows us *add new object types* having specialized reflectivty models *without having to modify and junk up existing functions* such as *raytrace()* and *illuminate()* with constructs such as:

```
if (closest->objtype == TILED_PLANE)
  do this
else if (closest->objtype == TEXTURED_PLANE)
  do that
else if (closest->objtype == PROCEDURAL_PLANE)
  do the other thing
else
  provide default behavior
```

Instead the ambient reflectivity should be recovered in *raytrace()* using:

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

closest->getamb(closest, intensity);
diffuse_illumination(lst, closest, intensity);
vl_scale3(1 / total_dist, intensity, intensity);
```

and *illuminate()* as:

```
hitobj->getdiff(hitobj, diffuse);
```

Loading a tiled plane object

As can be seen, the tiled plane specification is comprised of a finite plane specification followed by the alternate tile coloring.

```
16              tiled  plane
0 0 0           r g b ambient (foreground tiles)
6 1 8           r g b diffuse
0 0 0           r g b specular

1   0    1      normal
0   0    0      point is the lower left corner of a foreground tile
1   1   -1      x direction
1.25 0.5        size of a tile

0 0 0           r g b ambient  (background tiles)
8 4 0           r g b diffuse
0 0 0           r g b specular
```

Thus it would be reasonable to either:

> derive the *tplane_t* from the *plane_t* and copy the inner workings of *fplane_init()* to *tplane_init() or*
>
> derive the *tplane_t* from the *fplane_t* and have *tplane_init()* invoke *fplane_init().*

I elected to derive it from *plane_t* and thus my code duplicates the elements of *fplane_t* that are shown in *red.*

```
typedef struct tplane_type {
   double  xdir[3];         /* orientation of the tiling */
   double  size[2];         /* size of the tiling        */
   double  rotmat[3][3];    /* Rotation matrix           */
   material_t background;   /* background color          */
} tplane_t;
```

Loading the tiled plane description

Here is the bulk of what is required to set up a *tplane_t* object.

```
/**/
obj_t   *tplane_init(FILE *in, list_t *lst, int objtype) {
   int      pcount;
   tplane_t *tp;
   plane_t  *p;
   obj_t    *obj;

   /* Invoke "constructor" for "parent class" */

   Need to invoke plane_init() here

   /* Create the tplane_t object and point the plane_t to it */

   Allocate a tplane_t and set the priv pointer in the plane_t

  /* override the default reflectivity functions */

   obj->getamb  = tp_amb;     /* have to write these */
   obj->getdiff = tp_diff;
   obj->getspec = tp_spec;

   /* Load xdir and size fields as done in fplane */

   Have to copy in code from fplane here

  /* Finally load the background material reflectivity */

   Need to call material load here

   return(obj);
}
```

The reflectivity functions *tp_amb, tp_diff,* and *tp_spec*

From a high level perspective, the mission of these fellows is easy:

> Determine if the *obj->hitloc* lies in a "foreground" tile
> If so,
>   copy the reflectivity stored in the *obj_t)*
> If not,
>   copy the "background" reflectivity stored in the *tplane_t*.

The hard part is the *first step* so we abstract that out to the *tp_select()* function which will return 1 for "foreground" and 0 for "background".

```
/**/
void tp_diff(obj_t *obj, double *value){
   plane_t  *pln = (plane_t *)obj->priv;
   tplane_t *tp  = (tplane_t *)pln->priv;

   if (tp_select(obj))
      vl_copy3(obj->material.diffuse, value);
   else
      vl_copy3(tp->background.diffuse, value);
}
```

The *tp_amb()* and *tp_spec()* functions are clearly trivial modifications to *tp_diff()*.

Determining if a foreground or background tile has been hit

First consider the simple case: *obj->hitloc[]*

The plane *normal* is the positive *z-axis*
The plane *point (lower left corner of a foreground tile)* is the origin *obj->hitloc[]* contains the hit point location (note *obj->hitloc[2] == 0)*.

The relative tile number in the *x and y directions* of the tile that contains *obj->hitloc[]* are then

```
relx = (int) obj->hitloc[0] / tp->size[0];
```

and

```
rely = (int) obj->hitloc[1] / tp->size[1];
```

For example

suppose *tp->size = {2, 3}* and *obj->hitloc = {7.2, 6.5, 0.0}*

then

```
relx = 3;
rely = 2;
```

Having done this, the *tp_select()* function simply returns 0 if *relx* + *rely* is even and 1 if *relx* + *rely* is odd.

Complicating factors

While the preceding algorithm was simple it has a couple of holes that remained to be filled.

Suppose *tp->size = {1, 1};*

Consider the two *hitloc's {-0.5, 0.5, 0}* and *{0.5, 0,5, 0.0}*

The two locations are clearly in *different but adjacent* tile squares. The dividing line between the two tiles is the *y-axis.* Points in adjacent squares *must* have different colors.

Unfortunately the algorithm just described will yield *relx = rely = 0* for *both* points.

This will create ugly "double wide" strips of tiles along the *x and y axes.*

There are various hacks that can be used to prevent this. A particularly ugly one is:

```
relx = (int)(10000 + obj->hitloc[0] / tp->size[0]);
```

and

```
rely = (int)(10000 + obj->hitloc[1] / tp->size[1]);
```

Planes of arbitrary orientation

In the general case

the plane normal is not aligned with the z-axis
the xdir vector is not aligned with the x-axis
the point at the base of a tile square is not a the origin.

In this case the solution is analogous to the one used in the *fplane_hits()* function.

Subtract the coordinates of the *plane->point* which defines the lower left corner of a foreground tile square from *obj->hitloc[]* obtaining a translated hit position called *newhit[]*.

Construct a rotation matrix that will rotate

the plane normal into the z-axis
the xdir vector into the x-axis

Apply the rotation to *newhit[]*.

Compute *relx* and *rely* using *newhit[]* and proceed as previously described.

Unions

A union is a structured data that can be used to overlay different data types upon the same storage.

```
union fp_type
{
   unsigned char bvals[4];
   float         fval;
}  x;

main()
{
   x.fval = 34.25;
   printf("%02x %02x %02x %02x \n",
            x.bvals[0], x.bvals[1], x.bvals[2], x.bvals[3]);
}

$ gcc union.c
$ a.out
00 00 09 42
```

The amount of storage allocated for a union is the size of the largest component. In this example both have the same size but that is not necessary.

Unnamed *unions*

Another possible use for a union would be to embed the definition of all specific object types within the obj structure:

```
typedef struct obj_type {
   int objid;
   int objtype;
   union
   {
      sphere_t sphere;
      plane_t  plane;
   };
}  obj_t;
obj_t ob;

main()
{
   ob.sphere.radius = 5;
}
```

Note that the union was not given a name in the above code. We can name it but if we do, the name must be used in referencing the internal objects.

```
typedef struct obj_type
{
   int objid;
   int objtype;
      union
   {
      sphere_t sphere;
      plane_t  plane;

   } u;
}  obj_t;

obj_t ob;

main()
{
   ob.u.sphere.radius = 5;
}
```

Bitfields

Its possible to subdivide words into bitfields having individual names

```
struct bf_type
{
    unsigned int    p1:4;
    unsigned int    p2:8;
    unsigned int    p3:4;
} bf;


main()
{
   bf.p1 = 12;
   bf.p2 = 7;
   bf.p3 = 4;
   printf("%04x\n", bf);
   printf("%04x\n", bf.p1);
   printf("%04x\n", bf.p2);
   printf("%04x\n", bf.p3);

}
```

```
407c
000c
0007
0004
```

Because of endian issues bitfields are inherently not portable.

Pointers to pointers

Pointers to pointers are declared using `**`.

If a variable is declared as

```
item_t **double_p;
```

```
Then
     double_p    is an item_t** which is a pointer to an item_t*
     *double_p   is an item_t*  which is a pointer to item_t
     **double_p  is an item_t
```

In making (incorrect) assignments involving multiple levels of indirection it is common to see compiler warnings regarding “different levels of indirection”. These should not be ignored.

Exercise: given the following declarations which of the following will not generate compiler warnings:

```
item_t **dp;
item_t *sp;
item_t i;

sp = i;
dp = &i;
dp = &sp;
sp = *dp;
i = *sp;
```

Passing the address of a table of pointers

In practice double pointers are occasionally useful in two contexts. One is in *passing the address of a table of pointers as a pointers as a parameter.*

```
int main(int argc, char **argv){
   **argv           is the first character of the program name
   *(*(argv + 1))   is the first character of the first parameter
   *(*(argv + 2) + 1) is the second character of the second
                        parameter

}
```

Because the syntax is a bit on the opaque side you are strongly encouraged to run *offline* tests such as the one shown below to convince yourself you really know what you are doing.

```
main(int argc, char **argv)
{
    printf("%c %c %c \n",
           **argv,
           *(*(argv + 1)),
           *(*(argv + 2) + 1));
}
$a.out Hello Winter
a H i
```

Allowing a function to fill in the address of allocated storage

Double pointers can also be used if a subroutine is used to allocate a data structure and return its address to the caller. Previously far we have recommended the following approach:

```
obj_t * obj_load() {
    obj_t *new;
    new = = malloc(sizeof (obj_t));
    return(new);
}

main(){
    obj_t *newobj;
    newobj = obj_load();
}
```

An alternative approach is:

```
int obj_load(obj_t **new){
    *new = malloc(sizeof (obj_t));
    return(0);
}

main(){
   obj_t *newobj;
   int  rc;
   rc = obj_load(&newobj);
}
```