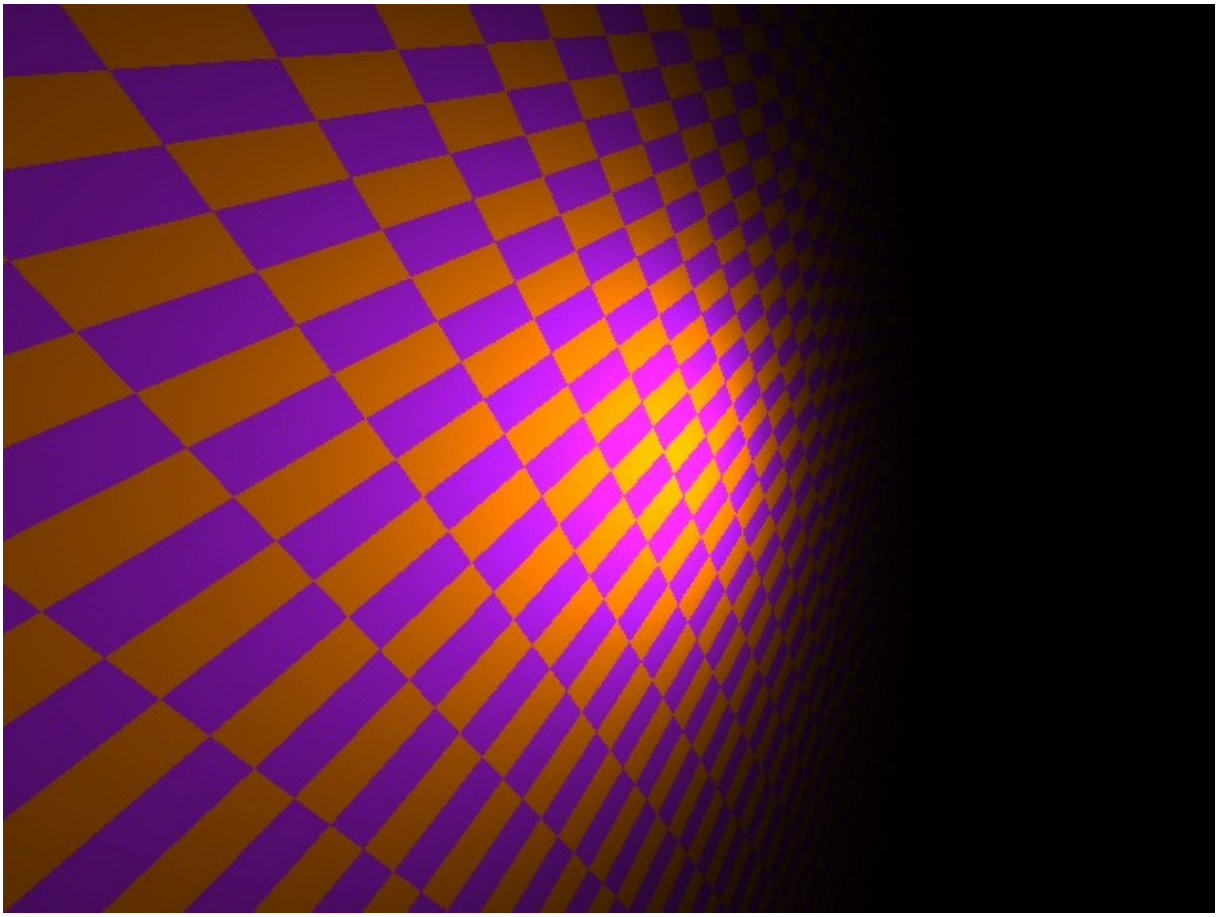
The tiled plane

So far we have generated objects with solid color surfaces. A popular alternative is the *tiled surface*. We will create a new scene object type called a *tiled plane* that will be derived from the infinite plane object.

We will be able to use the existing *plane_hits()* function, but the question for a filed plane is "which color tile is a given hitloc[] in"? We will find that this question shares many of the same issues that we encounted for a finite plane.

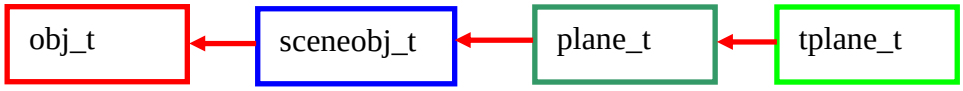
As can be observed, the plane is comprised of a collection of *tiles.* The tiles share characteristics of the *finite_plane*. The tiles have *x* and *y* dimensions and a vector which, when projected onto the plane determines the *x-axis* direction of the tiling.



Unlike both the infinite 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 *sceneobj_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:



Using the function pointers of the sceneobj_t to provide polymorphic behavior

In the sceneobj_t() structure for a scene object there are three function pointers *getamb*, *getdiff* and *getspec* that we have not used yet. These functions will be used to retrieve the red, green and blue values for the ambient, diffuse and specular lighting values for a given object. In the default case these would simply retrieve the intensity values stored in the *material_t* structure of the object. For tiled planes we will override the default pointers and set these three pointers to point to functions that return the color of the tile the hit location is in.

```
/** Scene object category **/
typedef struct sceneobj {
   /* Pointer to type dependent data */
   void    *priv;        /* Private type-dependent data           */

/* Reflectivity for reflective objects */
   material_t material;

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

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

/* "Hit" information for object */
   double  hitloc[3];    /* Last hit point                       */
   double  normal[3];    /* Normal at hit point                  */
   double  distance;     /* Distance from "base" to last hit point */
} sceneobj_t;
```

Modifications to material_init()

Since we will be modifying ray_trace() and diffuse_illumination() to use function pointers for calls to retrieve the reflectivity values of the hit point, we need to set the default functions pointers *getamb*, *getdiff* and *getspec* in the *sceneobj_t* structure to point to functions that simply return the ambient, diffuse and specular fields of the object. This is logically done in material_init(). The default routines *getamb()*, *getdiff()* and *getspec()* can also be defined in the *material.c* module.

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 *process_light()* 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 something else still
else
  provide default behavior
```

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

```
/* Hit something not a light */
   sceneObjPtr->getamb(closest, intensity);
   diffuse_illumination(model, closest, intensity);
   vl_scale3(intensity, intensity, 1.0/total_dist);
```

and *process_light()* as:

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

We can now add the tiled plane, and other scene object types that have surfaces that are not a solid color, by writing functions that compute the ambient, diffuse and specular lighting values based upon the hit location and the characteristics of the object.

Loading a tiled plane object

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

```
tplane          # 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 forground tile

# tplane specific data
1    1   -1     # tile orientation
1.25 0.5        # size of a tile

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

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 *tplane_t* from *plane_t* and thus my code duplicates the elements of *fplane_t* that are shown in *red*.

```
typedef struct tplane_type
{
   double  orient[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

The tplane_init() function incorporates code that is similar to fplane_init() and material_init(). A general pseudo-code description is:

```
obj_t *tplane_init(...)
{
   obj = plane_init(...);
   sptr = obj->catpriv;
   pln = sptr->priv;
   allocate new tplane_t;
   link it to pln->priv;
   read orient, size;
   project orient onto infinite plane
   compute required rotation matrix
   read background tile ambient, diffuse and specular data
   /* Override default pointers to surface color routines */
   sptr->getamb  = tp_getamb;     /* have to write these */
   sptr->getdiff = tp_getdiff;
   sptr->getspec = tp_getspec;

   return(obj);
}
```

The reflectivity functions *tp_getamb*, *tp_getdiff*, and *tp_getspec*

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

> Determine if the *sceneObjPtr->hitloc* lies in a "foreground" tile
> If so,
> copy the reflectivity stored in the *sceneobj_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_getamb(
obj_t *obj,
double *intensity)
{
   sceneobj_t *sPtr = (sceneobj_t *)obj->catpriv;
   plane_t  *pln = (plane_t *)sPtr->priv;
   tplane_t *tp  = (tplane_t *)pln->priv;

   if (tp_select(obj))
      vl_copy3(intensity, sPtr->material.ambient);
   else
      vl_copy3(intensity, tp->background.ambient);
}
```

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

Determining if a foreground or background tile has been hit

Much as we did with the simple case for finite plane, first consider the simple case:

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

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

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

and

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

For example

> suppose *tp->size = {2, 3}* and *sptr->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. This condition may be expressed as:

```
(relx + rely) % 2;
```

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 *mus t* have different colors.

Unfortunately the algoithm 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 (Westall's hack) just moves the hit point well away from the center line, i.e.:

```
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 at 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 *sPtr->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[]*. Note: if *newhit[2] != 0* at this point something is fatally flawed!!

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