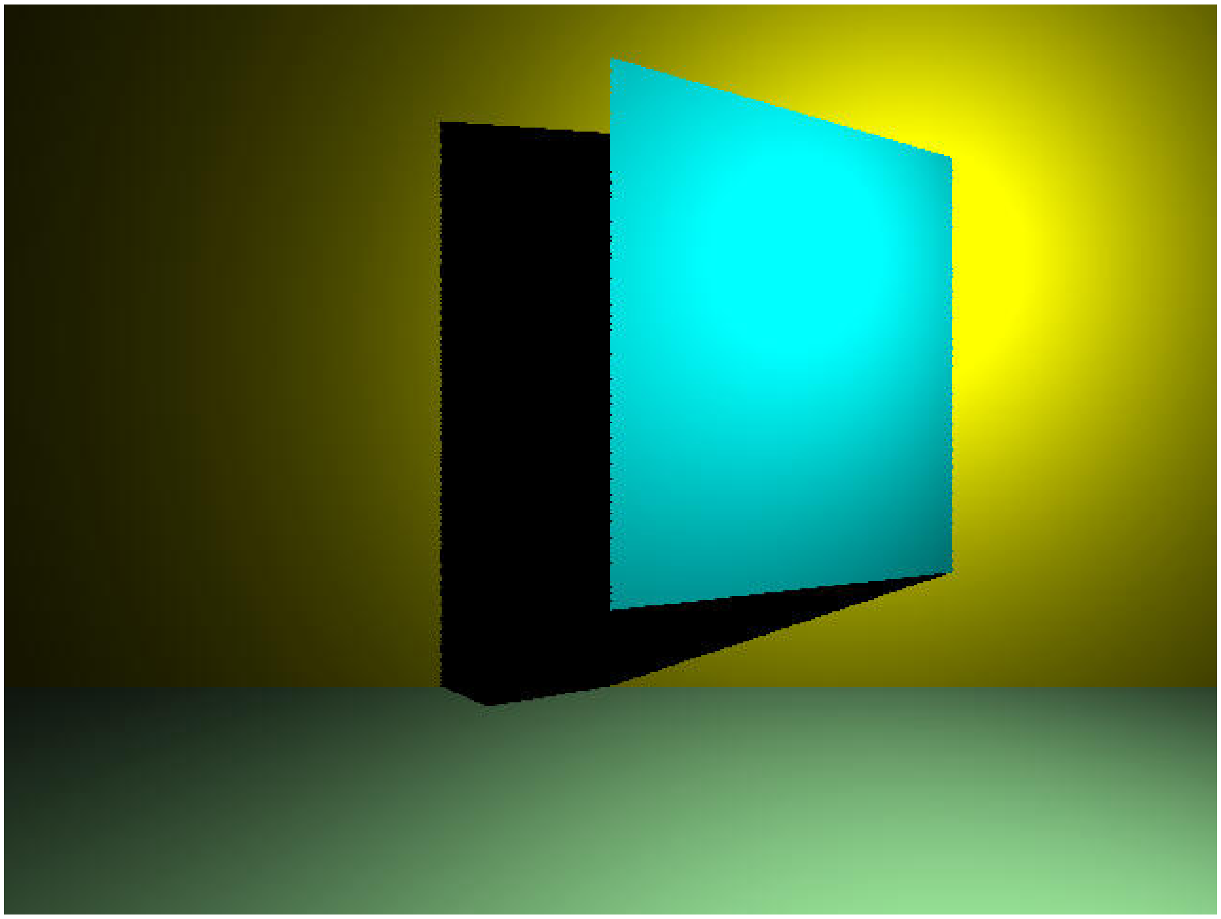
Finite rectangular planes.

The planes that we have previously used have all been of unbounded size. We can also define a finite or bounded plane object.

To define a finite plane, we will use some of the information we already have for an infinite plane, and several new fields that are specific to finite planes.

From the plane_t structure we will use the *point* field used in the definition of an unbounded plane to represent the location of the "lower left" corner of the finite, rectangular plane. The plane *normal* plays its usual role. Additional quantites that are required for a finite plane are:

> The vector *orient[3]* *when projected into the plane* and represents the orientation of the finite plane (i.e. its angle in the picture). The vector should be projected and converted into a unit vector at the time the finite plane definition is loaded.
>
> The vector *size[2]* vector provides the *width* and *height* in world coordinates of the rectangular plane in the *x* and *y* directions respectively.



Data structures for the finite plane

In the C language, there is no “built in” support for derived classes and inheritance, but, as seen earlier, we can build it in ourselves, by adding a *priv* pointer the the *plane type*.

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


typedef struct fplane_type
{
   double  orient[3];      /* plane orientation */
   double  size[2];        /* width x height    */
   double  rotmat[3][3];   /* Rotation matrix   */
   double  lasthit[2];     /* used for textures */
} fplane_t;
```

Alternatively (but more easily) one could just junk up the *plane_t* structure with finite plane and/or textured plane attributes.

Input Data for the finite plane

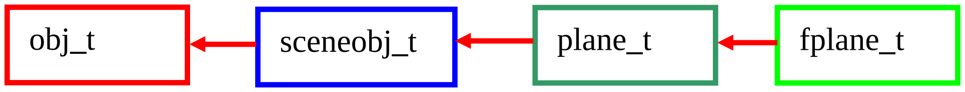
```
# finite plane
fplane
0 0 0         # r g b ambient
0 6 6         # r g b diffuse
0 0 0         # r g b specular

1  0  3       # normal
4  1 -3       # point

# Finite plane specific data
1  2  0       # plane’s orientation
5  5          # size
```

Initializing the *fplane_t*

In a true object oriented language we would have the following inheritance structure



When a new plane_t was created, constructors for *obt_t*, *sceneobj_t*, *plane_t*, and *fplane_t* would be *automatically* activated in that order. As we have seen there are no constructors for C structures, but we can emulate the behavior through explicit calls and *avoid needless duplication of code.*

```
obj_t *obj_init(...)
{
     allocate new obj_t;
     link it into the object list;
     return(obj);
}

obj_t *sceneobj_init(. . .) {
    obj  = obj_init(...);
    allocate new sphereobj_t;
    link it to obj->catpriv;
    load material properties
}

obj_t *plane_init(...)
{
    obj  = sceneobj_init(...);
    allocate new plane_t;
    link it to obj->catpriv->priv;
    load plane geometry
    return(obj);
}

obj_t *fplane_init(...)
{
   obj = plane_init(...);
   sptr – obj->catpriv;
   set sptr->hits to fplane_hits;
   pln = sptr->priv;
   allocate new fplane_t;
   link it to pln->priv;
   read orient, size;
   project orient onto infinite plane
   compute required rotation matrix
   return(obj);
}
```

An analogous strategy can be used in *fplane_dump()*.

The *fplane_hits()* function

If *sptr = obj->catpriv*, the *sptr->hits()* pointer for a finite plane should point to *fplane_hits()*.

```
double  fplane_hits(
double  *base, /* the (x, y, z) coords of origin of the ray */
double  *dir,  /* the (x, y, z) direction of the ray        */
obj_t   *obj)  /* the object to be tested for the hit.      */
{
```

Even though an *fplane_hits()* function is required, it would be a *very bad* idea to paste the internals of *plane_hits()* inline here. Instead, *plane_hits()* should be invoked to determine if and where the ray hits the infinite plane in which the finite plane is contained.

```
     t = plane_hits(base, dir, obj);
     if (t < 0)
          return(t);
```

Arrival here means that the ray hit the infinite plane and that the location of the hit has been stored in *sptrj->hitloc[],* The next task is to determine is *within the bounds of the prescribed rectangular area.*

In general, this seems like a *very difficult* problem. But there is a case for which the answer is simple.   Suppose the base of the rectangle happened to be at *(0, 0, 0)*, the *xdir[]* vector was *(1,0.0)* and the plane normal is *(0, 0, 1).* In that case, the rectangular finite plane is based at the origin and lies in the (x, y) plane. Therefore the following test could be applied (*fp* points to the fplane_t structure and *sptr* points to the scenceobj_t structure).

```
   if ((sptr->hitloc[0] > fp->size[0]) || (sptr->hitloc[0] < 0.0))
        return(-1);

   if ((sptr->hitloc[1] > fp->size[1]) || (sptr->hitloc[1] < 0.0))
      return(-1);
```

Transforming the coordinates of the finite plane.

A two-step coordinate system transformation may be applied to the original *sptr->hitloc[]* to permit use of the simple test on the previous page. The idea is to map the hitloc[] to a new coordinate system where the lower left corner of the plane is at (0,0,0) and the x, y and z axises correspond to the projected orient[] vector, a vector orthogonal (perpendicular) to the orient vector and the plane's normal, and the plane's normal. The two steps are:

1. translate (move) the lower left corner of the finite plane to the origin, and
2. rotate the coordinate system so that the plane normal rotates into the positive Z-axis and the orient[] vector rotates into the X-axis.

Step 1 can be accomplished via a simple:

```
vl_diff3(newhit, sptr->hitloc, pln->point);
```

Constructing the rotation is slightly more complicated (see next section). Once the rotation has been constructed the second step may be accomplished via (assuming *fp* points to the *fplane_t* structure):

```
vl_xform3(newhit, fp->rotmat, newhit);
```

After this is done, the simple test on the previous page may be applied to *newhit.*

Constructing the rotation matrix:

In fplane_init() we construct the rotation matrix *rotmat[][]* from the plane's normal and the projected *orient* vector:

$$\text{rotmat[3][3]} = \begin{vmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{vmatrix}$$

In fplane_init() we start with the inputted *orient[]* vector. We first convert it to a vector that is parallel to the plane using *vl_project3()* and then convert the result to a unit vector. In our rotated coordinate system this will become our x-axis and the first row of rotmat[][], i.e. rotmat[0].

The 3rd row will simply be the plane's normal (converted to a unit vector), and it will become the new coordinate system's z-axis, i.e. **rotmat[2]**.

This leaves the y-axis for the new coordinate system, which needs to be a vector that is orthogonal to both the projected *orient[]* vector (*rotmat[0]*) and the plane's *normal* (*rotmat[2]*). Recall that the cross product (*vl_cross3())* of two vectors is exactly that, a new vector that is orthogonal to both. In this case the cross product of the plane's normal vector and the projected orient vector will give us this third orthogonal vector. So we can use *vl_cross3()* to compute:

```
rotmat[1] = normal x projected_orient
```

or equivalently

```
rotmat[1] = rotmat[2] x rotmat[0]
```

which should also be converted to a unit vector. Now the vl_xform3() function can be used to convert the coordinates of a point to the new coordinate system.