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:

```plaintext
fplane origin {
    material white
    normal 1 0 1
    point 0 0 -6
    xdir 1 0 0
    dimensions 4 2
}
```

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 \texttt{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 printer(FILE *);
    // might need getter functions for dims and rothit

  protected:
    vec\_t rothit; /* translated then rotated last\_hit */
    double dims[2]; /* input dims in world coords */

  private:
    mtx\_t rot; /* rotation matrix */
    vec\_t projxdir; /* projected unitized xdir */
    vec\_t xdir; /* input xdir */
};
The **fplane_t** constructor:

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

    // The constructor should perform the following actions:
    // 1. set the obj_type to "fplane"
    // 2. parse the xdir and dimensions attributes
    // 3. project xdir onto the plane creating projxdir
    // 4. ensure that projxdir is not {0.0, 0.0, 0.0}
    // 5. 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.
    // 1. copy projxdir to row_0 of rot
    // 2. copy the plane normal to row_2 of rot and make it unit length.
    // 3. set row_1 of rot to the cross product of row_2 with row_0.

    attrmax = 2 tells plane_t constructor to consume only 2 attributes
}
```

The **fplane_t** printer:

```cpp
void fplane_t::printer(FILE *out) {
    plane_t::printer(out);

    // The printer should produce a formatted output of the xdir, projxdir, dimensions, and the rotation matrix
}
```

The **fplane_t** hits function

```cpp
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 first step is to invoke the hits function of the standard plane to determine if and where the ray hits the infinite plane in which the rectangular plane lies. The plane_t::hits() will return either -1 on a miss or a distance t to the hitpoint on the infinite plane.

    // • 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:

\[
0 \leq \text{last_hit}.x \leq \text{dims}[0] \quad \text{and} \\
0 \leq \text{last_hit}.y \leq \text{dims}[1]
\]

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 \( \text{last_hit} \) we can make the simple test above work. So that lighting will still work we cannot modify the original \( \text{last_hit} \).

The translation step is accomplished by setting \( \text{newloc} \) to \( \text{last_hit} - \text{point} \). (That is, you MUST compute \( \text{newloc} \) by subtracting the \textit{point} attribute of the \( \text{plane}_t \) from the \( \text{last_hit} \) object of the \( \text{object}_t \).

Then we rotate \( \text{newloc} \) by transforming it with the \( \text{rot} \) matrix. After the rotation we have effectively transformed the original \( \text{last_hit} \) into the "easy location" in which the lower left corner is at the origin and the \( \text{fplane} \) lies in the \( x\)-\( y \) plane. Then we can do the test:

\[
0 \leq \text{newloc}.x \leq \text{dims}[0] \quad \text{and} \\
0 \leq \text{newloc}.y \leq \text{dims}[1]
\]

We return \( t \) and \( \text{newloc} \) is saved in a protected element of the \( \text{fplane}_t \).

**Note:** A C language statement such as:

\[
\text{if } (0 \leq \text{newloc}.x \leq \text{dims}[0]) \\
\text{do-something}
\]

will compile correctly but will not do what you intend.
The general tiled plane

Tiled floors are common elements in raytracing systems primarily because they create interesting reflections on reflective 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 inheritance. 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 `tplane` has `getdiffuse()` and `getambient()` methods which will override the `getdiffuse` supplied in the `object_t` when `obj->getdiffuse` is invoked on a `tplane_t` object.

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

    virtual void printer(FILE *);
    virtual void getambient(drgb_t *);
    virtual void getdiffuse(drgb_t *);

  private:
    int select(void);    /* 0-> forgrnd 1 -> back */
    double dims[2];      /* tile dimensions       */
    material_t *altmat;  /* background material   */
    mtx_t rot;           /* rotation matrix       */
    vec_t projxdir;      /* proj / unitized xdir  */
    vec_t xdir;          /* x direction of tiling */
};
```
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) {
  1. Parse the three required parameters
  2. Ask material_getbyname() to return a pointer to the alternate (background) material
  3. Project xdir into the plane and make it a unit vector
  4. Build a rotation matrix that rotates the plane normal into the z-axis and the projected xdir into the x-axis.
}

The getdiffuse and getambient 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::getdiffuse(drgb_t *value) {
  if (select() == 0)
    ask the default (object level) getdiffuse() to fill in value
  else
    ask getdiffuse() to fill in the diffuse reflectivity of altmat
}
The `select()` method

1. Apply the `rot` matrix to `last_hit` to rotate it into the plane having normal (0, 0, 1) with the projected `xdir` of the tiling parallel to the x-axis. This should create a new vector called `newloc` whose z-component should be 0. 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.
2. Add 100000 to `newloc.x` and `newloc.y` (westall's hack to avoid ugly doublewide stripe at origin).
3. 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 foreground or background.