Finite planes

The planes that we have previously used have all been of unbounded size. We can also define a finite or bounded plane class. In this class the *point* field used in an unbounded plane represents the location of the "lower left" corner of the bounded plane. To support bounded planes in all directions we also must supply a vector *xdir[]* which *lies in the the plane* and represents the direction of increase of the *x* coordinate in the plane. This vector is analogous to the grid direction in the tiled plane.

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

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

protected:
   double          lasthit[2];
   double          width;
   double          height;

private:
   double          xdir[3];
};
```

The *hits()* method of the *fplane_t* class works as follows:

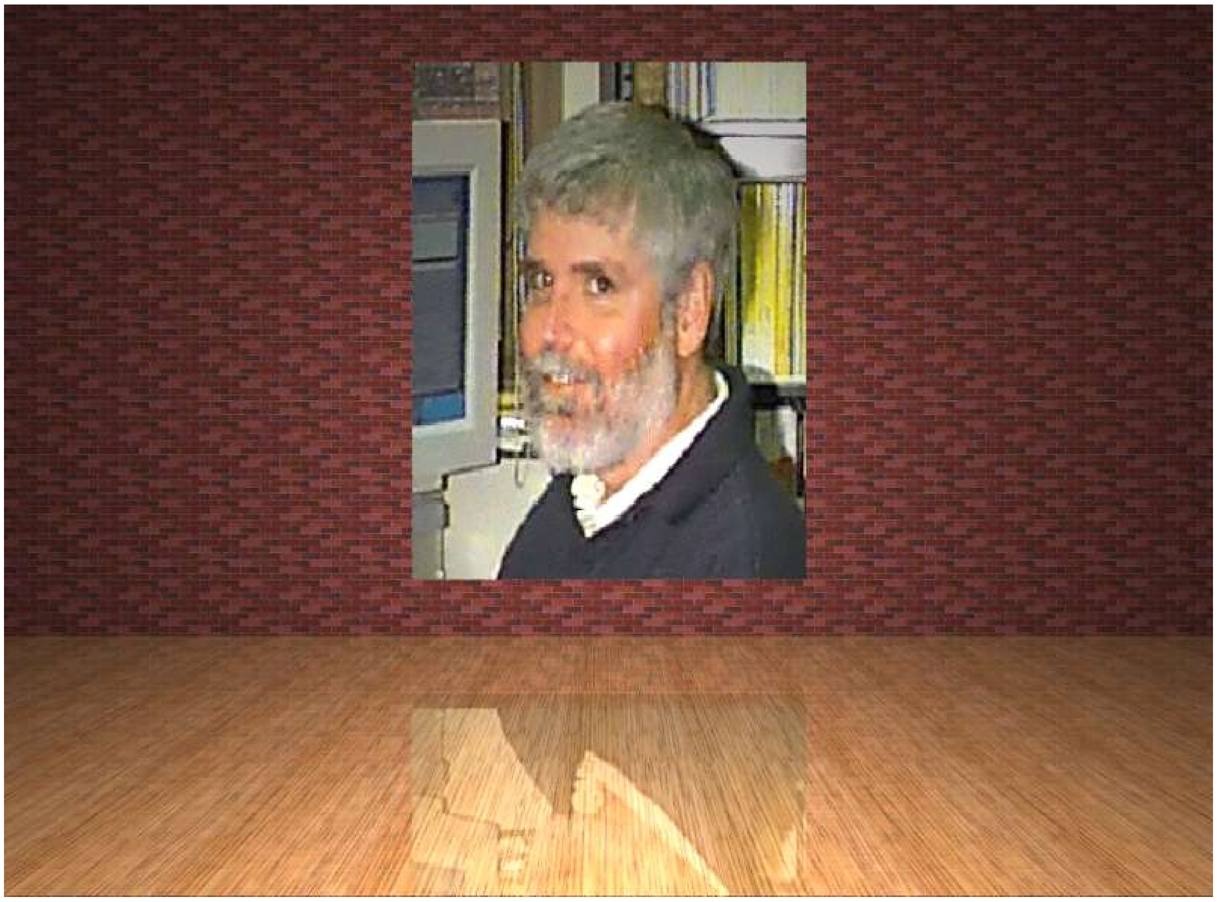
> Call *plane_t::hits()* to find *if / where* the ray hits the infinite plane. If it misses *return(-1).*
>
> Construct a rotation matrix that rotates the plane's normal into the *z-axis,* the *xdir[]* vector into the *x-axis.*
>
> Make a copy of the *hitloc[]*. Call the copy *testhit[].*
>
> Subtract the *base point* of the plane from *testhit[]* and apply the rotation to the result.
>
> Now if *0 <= testhit[0] <= width* and *0 <= testhit[1] <= height* a hit on the finite plane has occured and *t* is returned. Otherwise *-1* is returned.
>
> Before returning the *x* and *y* offsets of the hit are saved as *lasthit[0] = testhit[0], and lasthit[1] = testhit[1].*

Texture mapping

While the *fplane_t* class can be used as a basis for building rectangular solids such as cubes or bricks, a more interesting application is *texture mapping* in which an arbitrary image is mapped onto the finite plane. There are a large number of possible ways in which textures may be applied to finite planes. We will consider a small subset. The first of these is the issue of *CLAMPING* versus *REPEATING.*

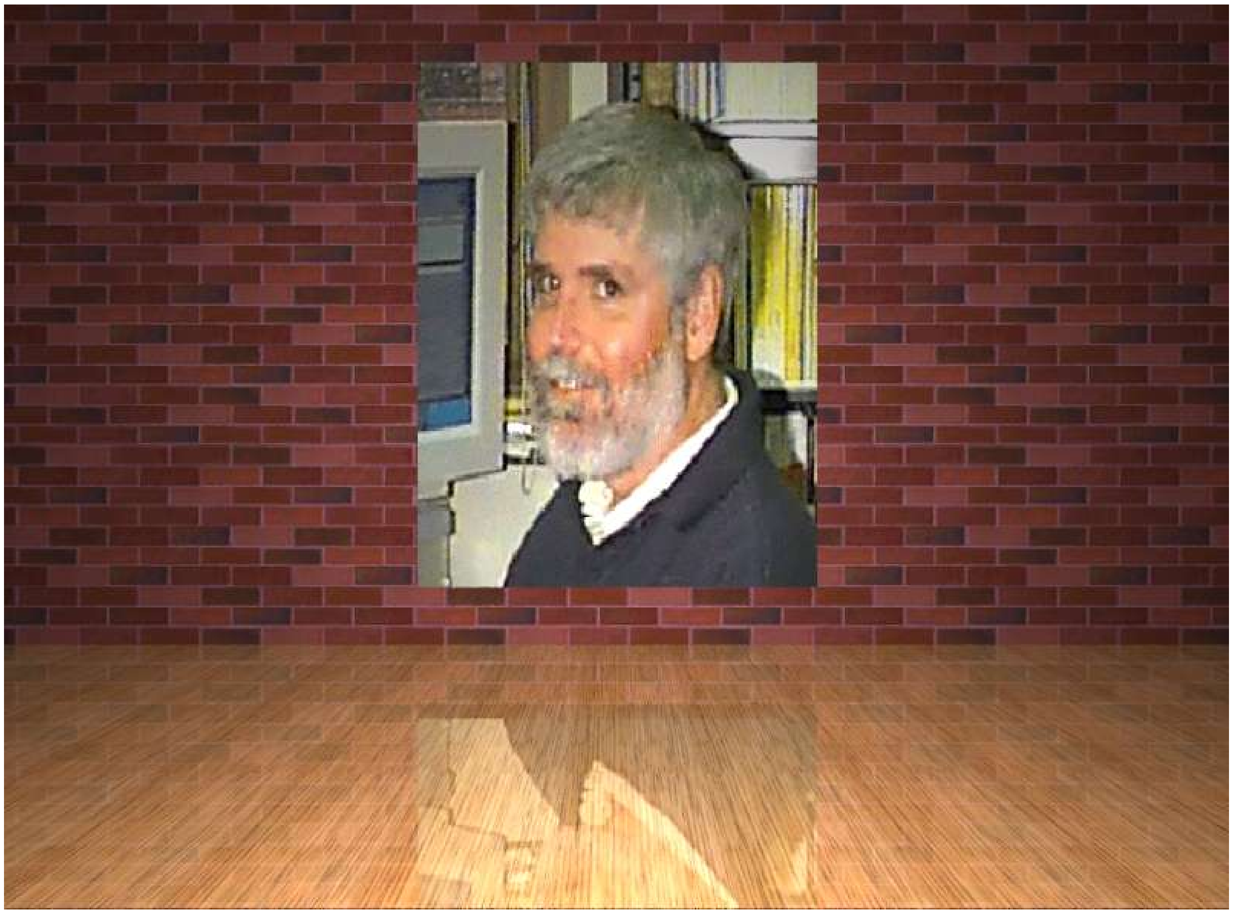
The image shown below consists of three finite planes. The floor and back wall are intentionally designed so that they fill the full viewable space. The *oak.ppm* and *brick4.ppm* texture files has size 128 x 128 pixels but the full image is 800 x 600.

To obtain the effect shown, these two textures are *REPEATED* or tiled across the respective finite planes. The portrait is *CLAMPED.* Clamping will stretch the texture image to exactly fit the dimensions finite plane. This stretching effect may cause distortion an will cause loss of image quality.



REPEATING the texture may also cause undesirable artifacts. The individual bricks in the image just shown look too small. But if the brick texture were *CLAMPED* to the large plane they would be much too large and with significant loss of quality. The easiest way to correct this situation is use *xv* to edit the *brick* texture and enlarge it.

In the more realistic looking image shown below the brick texture has been enlarged to 384x384 pixels.



Lighting of textured planes

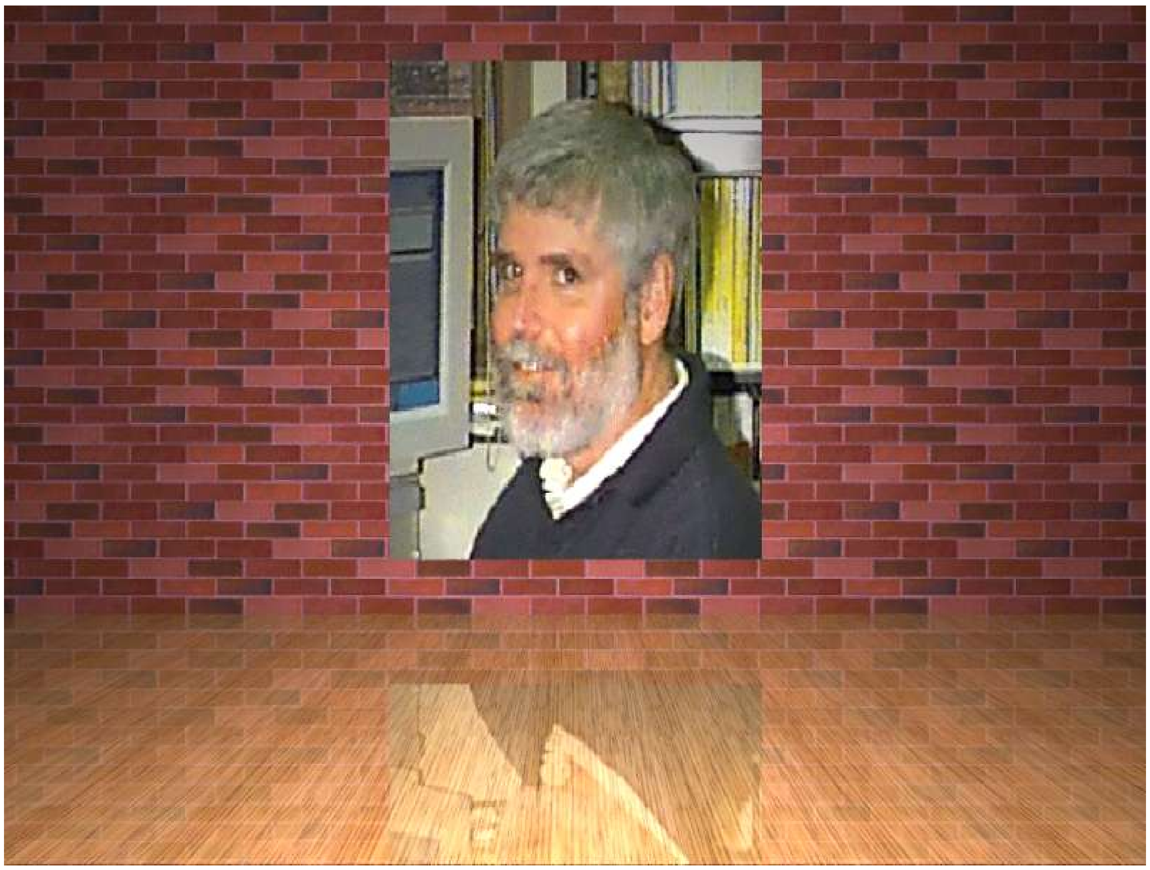
The lighting of textured planes is similar to the lighting of non-textured planes. For the images shown here the *(r, g, b)* components of a textured pixel are computed by *getamb()* and *getdiff()* functions. Recall that the default *getamb()* and *getdiff()* functions simply return the *ambient* and *diffuse* reflectivity of the item being lit.

For a *texplane_t* the *getamb()* function works as follows.

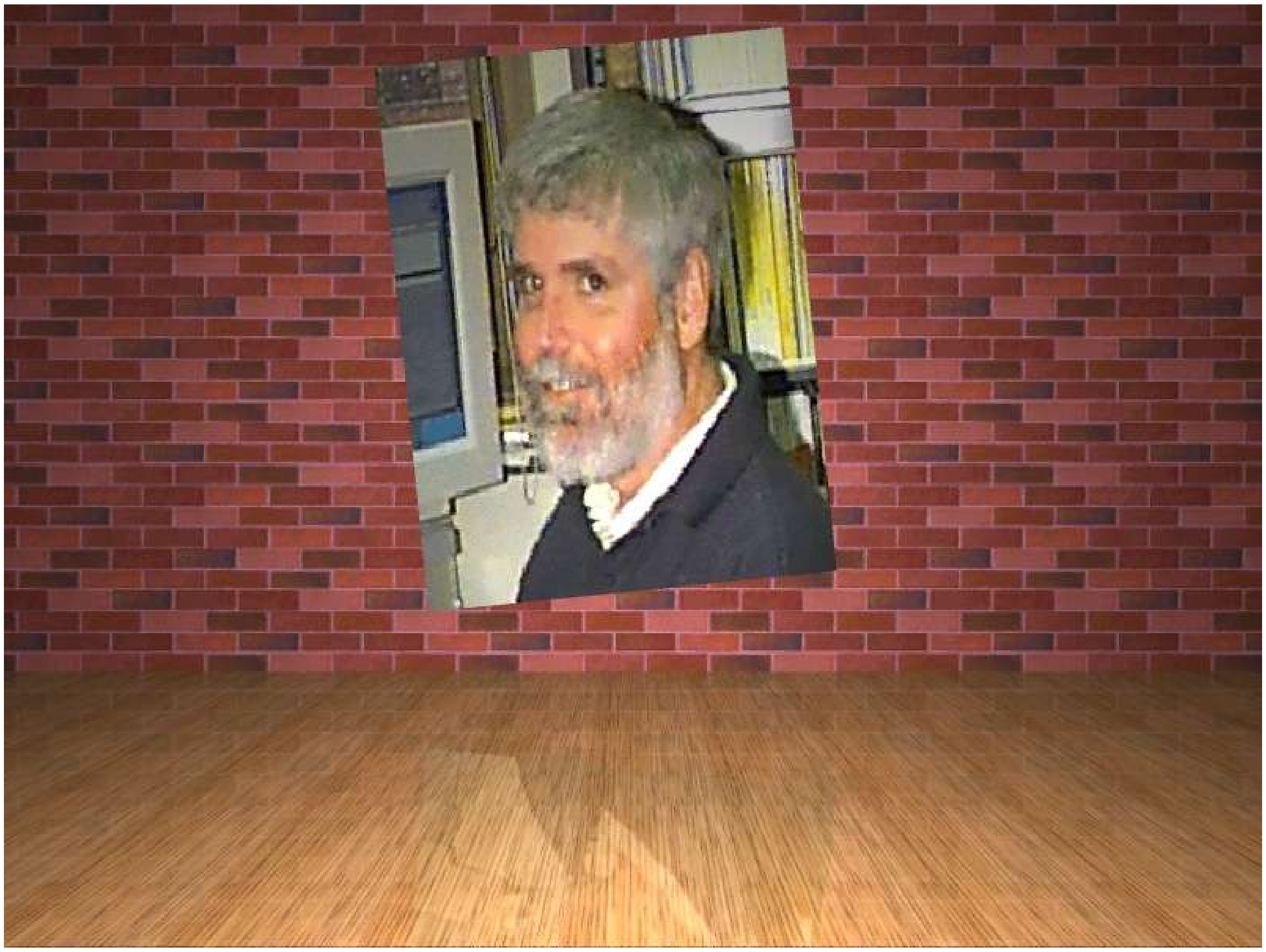
> *ambval[3] = the ambient reflectivity of the plane*
> *texval[3] = the value of the texture pixel or texel scaled to [0.0, 1.0]*

The value returned by *getamb* is the componentwise product of *ambval[]* and *texval[]*. Because of the multiplication, textured surfaces will general appear darker than non-textured ones. Boosting the ambient reflectivity can provide a more realistic and consistent appearance.

However, The diffuse component should not be completely nullified though as it provides the realistic change in illumination as a function of the distance from light sources. The image shown below has a 50% increase in the *ambient* component of the surface onto which the bricks are mapped over the one on the previous page.



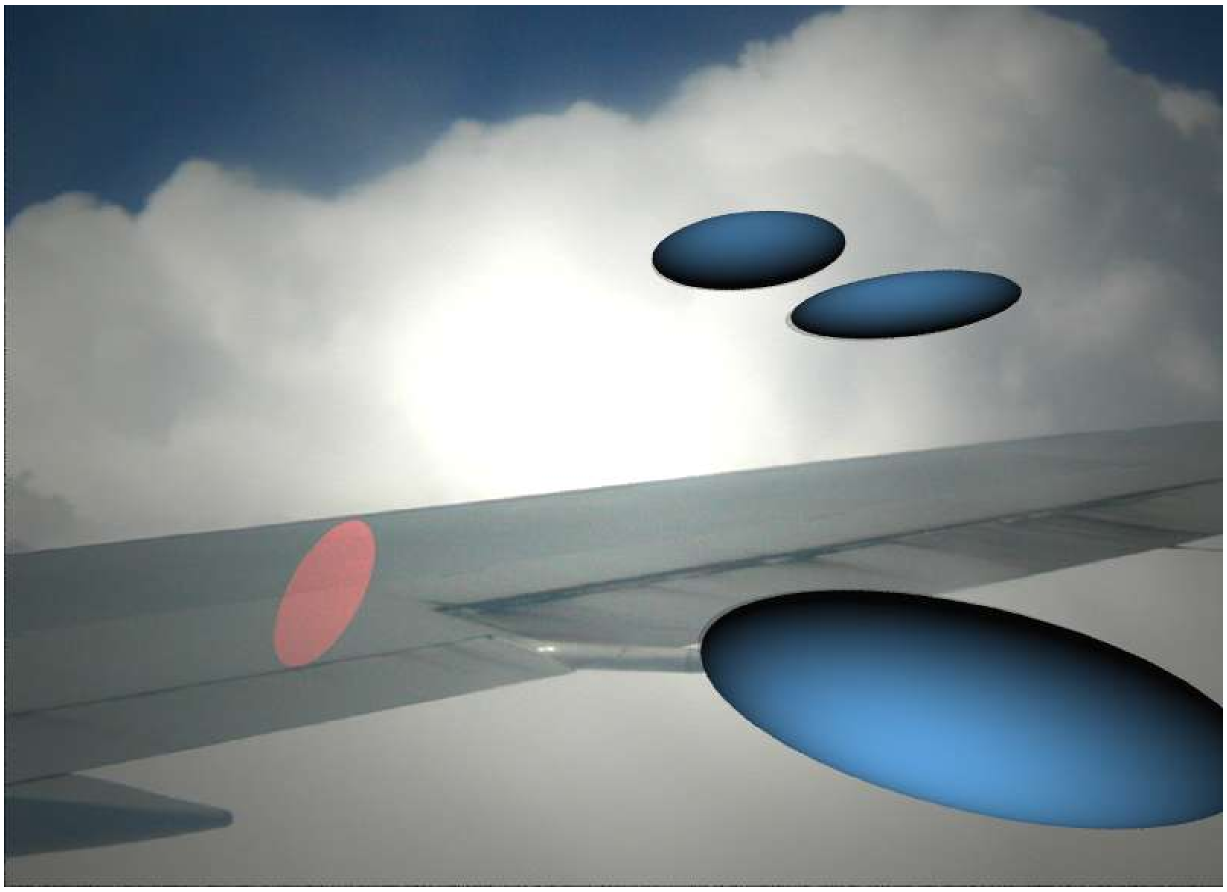
The brightening of the brick wall combined with the 0.4 specular reflectivity on the oak floor makes the floor look too “bricky”. No pictures are ever hung perfectly level either.   So...



It would also be nice to have another oak textured finite plane behind the portrait to serve as a frame but that will be left as an exercise!

Three Dimensional effects with Two Dimensional Textures

The floor, wall, and portrait textures are used to impart a realistic appearance to a flat surface. It is also possible to “trick” the viewer into perceiving a two dimensional texture as a three dimensional environment into which synthetic objects may be embedded. Here we see three flying saucers from Jupiter's moon Titan trailing a Boeing 767-300 at 36,000 feet over Iowa during the summer of 2003.



The texture is simply a “back wall”. All three saucers are necessarily physically in front of the wall, but the visual effect is that wing is closer to the viewer than are the two outer saucers. The shadows gray blurring around the leading edges of each saucer is the shadow that the saucer is casting on the back wall. Because the aircraft and the clouds are not truly three-d all such shadows will be anatomically incorrect. They should be eliminated by disabling the *light_visible()* test for class *texplane_t.* One unrealistic aspect of the picture is the the surface of the saucers is *too perfect* when compared with the wing of the seven-six which contains realistic dirt streaks and stains. In cases such as this it may be necessary to texture *everything* to obtain a better effect.

Details of the texture mapping procedure

The *texture_t* class is responsible for reading a texture into memory and retrieving the *(r,g, b)* components of a *texel* from relative *x, y* coordinates in [0.0, 1.0).

```
class texture_t
{
public:
   texture_t();
   texture_t(FILE *, char *);
   void  mapdiff(double relx, double rely, double *);
   void  getdim(int *x,int *y);
private:
   int              xdim;     // pixel dimensions of texture
   int              ydim;
   unsigned char *imagebuf;
};

class texplane_t: public fplane_t
{
public:
   texplane_t();
   texplane_t(FILE *, FILE *, int otype, int oid);
   virtual void  dump_obj(FILE *);
   virtual void  getdiff(double *w);
   virtual void  getamb(double *w);
           void set_pixsize(double xs,double ys);
           void texture_map(double *, double *);

private:
   texture_t     *texture;
   char          imagename[64];
   double        pix_x_size;   // size of a pixel in world coords
   double        pix_y_size;
   int           mode;         // CLAMP or REPEAT
};
```

Accessing a texel in *CLAMPED* mode.

The *getdiff* and *getamb* methods are simple wrappers for the *texture_function* which does the real work.

```
void texplane_t::getdiff(
double *value)
{
   double work1[3] = {0, 0, 0};
   matdiff(work1);
   texture_map(work1, value);
}

void texplane_t::getamb(
double *value)
{
   double work1[3] = {0, 0, 0};
   matamb(work1);
   texture_map(work1, value);
}
```

The *texture_map()* function divides the *lasthit[]* values which give the *x and y* offsets of the last hit in world coordinates by the *width* and *height* of the texture also in world coordinates to obtain the relative location of the *hit* within the texture.

```
void texplane_t::texture_map(
double *reflect,        /* Amb or Diff refectivity */
double *pixel)          /* output pixel value      */
{
   double work1[3] = {0, 0, 0};

   if (mode == TEX_CLAMP)
   {
     texture->gettexel(lasthit[0]/width, lasthit[1]/height, work1);
     vl_mult3(work1, reflect, pixel);
   }
```

The *gettexel* function

Given a relative offset within the texture, it is the mission of this function to return the *(r,g,b)* value of the pixel at that relative position within the image. This is done by multiplying the relative offset by the dimension of the texture.

```
void texture_t::gettexel(
double xrel,
double yrel,
double tex[3])
{
   unsigned char *texloc;
   int xloc = (int)(xrel * xdim);
   int yloc = (int)((1.0 - yrel) * ydim);

   if (yloc == ydim)
      yloc = ydim - 1;

   texloc = imagebuf + 3 * xdim * yloc + 3 * xloc;

   tex[0] = *texloc / 255.0;
   tex[1] = *(texloc+1) / 255.0;
   tex[2] = *(texloc+2) / 255.0;
}
```

Computing the relative offset in *REPEAT* mode

This is slightly more complicated than in *CLAMPED* mode. Recall that *lasthit[]* contains the offset of the last hit from the lower left corner of the finite plane in World coordinates. The first step is to convert this to Pixel coordinates. This is done by dividing by *pix_x_size* and *pix_y_size.* Then the value

*((1.0 * (pix_x_offset % texture::xdim)) / texture::xdim)*

is the relative offset of the desired texel in the texture.