Texture Mapping

Prerequisites

An understanding of the geometry of polygons in 3-space, a concept of interpolation across a polygon, and the concept of how values in an array can map linearly to values in another space.

Introduction

Texturing — applying textures to a graphical object to achieve a more interesting image or add information to the image without computing additional geometry — is a significant addition to the capabilities of computer graphics. Texturing is a rich topic and we will not try to cover it in depth, but we will discuss something of what is possible as we develop the subject in a way that is compatible with current graphics APIs and that you can implement in your work.

The key idea is to apply additional information to your images as your geometry is computed and displayed. In our API environment that primarily works with polygons, as the pixels of the polygon are computed, the color of each pixel is not calculated from a simple lighting model but from a texture map, and this chapter will focus on how that texture mapping works. Most of the time we think of the texture as an image, so that when you render your objects they will be colored with the color values in the texture map. This approach is called *texture mapping* and allows you to use many tools to create visually-interesting things to be displayed on your objects. There are also ways to use texture maps to determine the luminance, intensity, or alpha values of your objects, adding significantly to the breadth of effects you can achieve.

Texture mapping is not, however, the only approach that can be used in texturing. It is also possible to compute the texture data for each pixel of an object by procedural processes. This approach is more complex than we want to include in a first graphics programming course, but we will illustrate some procedural methods as we create texture maps for some of our examples. This will allow us to approximate procedural texturing and to give you an idea of the value of this kind of approach, and you can go on to look at these techniques yourself in more detail.

Texture maps are arrays of colors that represent information (for example, an image) that you want to display on an object in your scene. These maps can be 1D, 2D, or 3D arrays, though we will focus on 1D and 2D arrays here. Texture mapping is the process of identifying points on objects you define with points in a texture map to achieve images that can include strong visual interest while using simpler geometry.

The key point to be mastered is that you are dealing with two different spaces in texture mapping. The first is your modeling space, the space in which you define your objects to be displayed. The second is a space in which you create information that will be mapped to your objects. This information is in discrete pieces that correspond to cells in the texture array, often called *texels.* In order to use texture maps effectively, you must carefully consider how these two spaces will be linked when your image is created — you must include this relationship as part of your design for the final image.

There are many ways to create your texture maps. For 1D textures you may define a linear color function through various associations of color along a line segment. For 2D textures you may use scanned images, digital photos, digital paintings, or screen captures to create the images, and you may use image tools such as Photoshop to manipulate the images to achieve precisely the effects you want. Your graphics API may have tools that allow you to capture the contents of your frame buffer in an array where it can be read to a file or used as a texture map. This 2D texture world is the richest texture environment we will meet in these notes, and is the most common texture

context for most graphics work. For 3D textures you may again define your texture by associating colors with points in space, but this is more difficult because there are few tools for scanning or painting 3D objects. However, you may compute the values of a 3D texture from a 3D model, and various kinds of medical scanning will produce 3D data, so 3D textures have many appropriate applications.

Most graphics APIs are quite flexible in accepting texture maps in many different formats. You can use one to four components for the texture map colors, and you can select RGB, RGBA, or any single one of these four components of color for the texture map. Many of these look like they have very specialized uses for unique effects, but an excellent general approach is to use straightforward 24-bit RGB color (8 bits per color per pixel) without any compression or special file formats — what Photoshop calls "raw RGB."

Finally, texture mapping is much richer than simply applying colors to an object. Depending on the capabilities of your graphics API, you may be able to apply texture to a number of different kinds of properties, such as transparency or luminance. In the most sophisticated kinds of graphics, texturing is applied to such issues as the directions of normals to achieve special lighting effects such as bump mapping and anisotropic reflection.

Definitions

In defining texture maps below, we describe them as one-, two-, or three-dimensional arrays of colors. These are the correct definitions technically, but we usually conceptualize them a little more intuitively as one-, two-, or three-dimensional spaces that contain colors. When texture maps are applied, the vertices in the texture map may not correspond to the pixels you are filling in for the polygon, so the system must find a way to choose colors from the texture arrays. There are ways to filter the colors from the texture array to compute the value for the pixel, ranging from choosing the nearest point in the texture array to averaging values of the colors for the pixel. However, this is usually not a problem when one first starts using textures, so we note this for future reference and will discuss how to do it for the OpenGL API later in this chapter.

1D texture maps: A 1D texture map is a one-dimensional array of colors that can be applied along any direction of an object — essentially as though it were extended to a 2D texture map by being replicated into a 2D array. It thus allows you to apply textures that emphasize the direction you choose, and in our example below it allows us to apply a texture that varies only according to the distance of an object from the plane containing the eye point.

2D texture maps: A 2D texture map is a two-dimensional array of colors that can be applied to any 2D surface in a scene. This is probably the most natural and easy-to-understand kind of texture mapping, because it models the concept of "pasting" an image onto a surface. By associating points on a polygon with points in the texture space, which are actually coordinates in the texture array, we allow the system to compute the association of any point on the polygon with a point in the texture space so the polygon point may be colored appropriately. When the polygon is drawn, then, the color from the texture space is used as directed in the texture map definition, as noted below.

3D texture maps: A 3D texture map is a three-dimensional array of colors. 3D textures are not supported in OpenGL 1.0 or 1.1, but were added in version 1.2. Because we assume that you will not yet have this advanced version of OpenGL, this is not covered here, but it is described in the OpenGL Reference Manual and the OpenGL Programmers Guide for version 1.2. A useful visual examination of 3D textures is found in Rosalee Wolfe's book noted in the references. The 3D texture capability could be very useful in scientific work when the 3D texture is defined by an array of colors from data or theoretical work and the user can examine surfaces in 3-space, colored by the texture, to understand the information in the space.

The relation between the color of the object and the color of the texture map

In a texture-mapping situation, we have an object and a texture. The object may be assumed to have color properties, and the texture also has color properties. Defining the color or colors of the texture-mapped object involves considering the colors both the object and the texture map.

Perhaps the most common concept of texture mapping involves replacing any color on the original object by the color of the texture map. This is certainly one of the options that a graphics API will give you. But there are other options as well for many APIs. If the texture map has an alpha channel, you can blend the texture map onto the object, using the kind of color blending we discuss in the color chapter. You may also be able to apply other operations to the combination of object and texture color to achieve other effects. So don't assume simply that the only way to use texture maps is to replace the color of the object by the color of the texture; the options are much more interesting than merely that.

Texture mapping and billboards

In the chapter on high-performance graphics techniques we introduce the concept of a *billboard* — a two-dimensional polygon in three-space that has an image texture-mapped onto it so that the image on the polygon seems to be a three-dimensional object in the scene. This is a straightforward application of texture mapping but requires that the color of the polygon come entirely from the texture map and that some portions of the texture map have a zero alpha value so they will seem transparent when the polygon is displayed.

Creating texture maps

Any texture you use must be created somehow before it is loaded into the texture array. This may be done by using an image as your texture or by creating your texture through a computational process. In this section we will consider these two options and will outline how you can create a texture map through each.

Getting an image as a texture map

Using images as texture maps is very popular, especially when you want to give a naturalistic feel to a graphical object. Thus textures of sand, concrete, brick, grass, and ivy — to name only a few possible naturalistic textures — are often based on scanned or digital photographs of these materials. Other kinds of textures, such as flames or smoke, can be created with a digital paint system and used in your work. All the image-based textures are handled in the same way: the image is created and saved in a file with an appropriate format, and the file is read by the graphics program into a texture array to be used by the API's texture process. And we must note that such textures are 2D textures.

The main problem with using images is that there is an enormous number of graphics file formats. Entire books are devoted to cataloging these formats, and some formats include compression techniques that require a great deal of computation when you re-create the image from the file. We suggest that you avoid file formats such as JPEG, GIF, PICT, or even BMP, and use only formats that store a simple sequence of RGB values. Using compressed images directly requires you to use a tool called an RIP — a raster image processor — to get the pixels from your image, and this would be a complex tool to write yourself. If you want to use an image that you have in a compressed file forma, probably the simplest approach is to open your image in a highly-capable image manipulation tool such as Photoshop, which can perform the image re-creation from most formats, and then re-save it in a simplified form such as raw RGB. Graphics APIs are likely to have restrictions on the dimensions of texture maps (OpenGL requires all dimensions to be a

power of 2, for example) so even if the format is so low-level that it does not include the dimensions, they can be recalled easily. We suggest that you include the dimension as part of the file name, such as `ivy.128x64.rgb` so that the size will not be something that must be recorded. The process of using an image file as a texture map is described in the second code example in this chapter.

Generating a synthetic texture map

Because a texture map is simply an array of color, luminance, intensity, or alpha values, it is possible to generate the values of the array by applying a computational process instead of reading a file. Generating a texture computationally is a very powerful technique that can be very simple, or it may be relatively complex. Here we'll describe a few techniques that you might find helpful as a starting point in creating your own computed textures.

One of the simplest textures is the checkerboard tablecloth. Here, we will assume a 64x64 texture array and can define the color of an element `tex[i][j]` as red if `((i%4)+(j%4))%2` has value zero and white if the value is one. This will put a 4x4 red square at the top left of the texture and will alternate white and red 4x4 squares from there, thus creating a traditional checkerboard pattern.

A particularly useful instance of a computed texture involves the use of a *noise function*. A noise function is a single-valued function of one, two, or three variables that has no statistical correlation to any rotation (that is, does not seem to vary systematically in any direction) or translation (does not seem to vary systematically across the domain) and that has a relatively limited amount of change in the value across a limited change in the domain. When one does procedural texturing, one can calculate the value of a noise function for each pixel in a textured object and use that value to determine the pixel color in several different ways There are a number of ways to create such functions, and we will not begin to explore them all, but we will take one approach to defining a noise function and use it to generate a couple of texture maps.

The usual technique to define a noise function is to use (pseudo)random numbers to determine values at a mesh of points in the domain of the function, and then use interpolation techniques to define the values of the function between these. The interpolation techniques give you a smoother function than you would have if you simply used random values, producing a more effective texture. We will focus on 2D texture maps here by defining functions that calculate values for a 2D domain array, but this is all readily extended to 1D or 3D noise. You may compute a single noise function and use it to generate a noise texture map so you can work with the same texture map to have a consistent environment as you develop your image, or you may calculate a new noise function and new texture map each time you execute your program and achieve a constantly-new effect. We suggest that you use the consistent texture map for development, however, so you may distinguish effects caused by the texture map from effects caused by other graphics issues.

The approach we will take to develop a noise function is the gradient interpolation discussed by Peachy in [Ebert et al] and is chosen to give us a simple interpolation process. This is also the kind of noise function used in the Renderman™ shader system, which you may want to investigate but which we do not cover in these notes. For each point of the 3D mesh, whose indices we will treat as x-, y-, and z-components of the noise function domain, we will compute a unit vector of three components that represents the gradient at that point. We will then assume a height of 0 at each grid point and use the gradients to define a smooth function to be our basic noise function. Other functions will be derived from that. A 3D noise is a real-valued function of three variables, and we can obtain 2D or even 1D noise functions by taking 2D or 1D subspaces of the 3D noise space. But we will retain the basic 3D approach because 3D textures are difficult to get through non-computational means. We will also re-phrase the code in terms of multi-dimensional arrays to

make it easier to read, but you should note that writing in terms of arrays will create slower execution, which is a critical problem for real procedural textures. You are encouraged to read the discussions of efficiency in [Ebert] to understand these issues.

Generating the gradients for each grid point depends on generating random numbers between zero and one. We assume that you have available, or can easily create, such a function, which we will call `random()`, and that your local laboratory support will help you work with any issues such as initializing the random number environment. Note that the computation `z = 1.–2.*random()` will produce a random number between -1 and 1. Then the function below, to be executed in the initialization section of your program, sets values in the table that will be interpolated to produce the noise function:

```
float gradTab[TABSIZE][TABSIZE][TABSIZE][3];

void gradTabInit(void) {
   float z, r, theta;
   int i;

   for (i = 0; i < TABSIZE; i++) {
      z = 1. - 2.*random();
      r = sqrt((1. - z*z));
      theta = 2.0*PI*random();
      gradTab[i][0] = r*cos(theta);
      gradTab[i][1] = r*sin(theta);
      gradTab[i][2] = z;
   }
}
```

In addition to the table, we build an array of permutations of [0..255] so that we can interpolate the gradient data from distributed points, giving us the uncorrelated information we need for the noise function.

```
// the table is a permutation of 0..255; it could as easily be generated
// with a permutation function if one wished
static unsigned char perm[TABSIZE] = {
   225,155,210,108,175,199,221,144,203,116, 70,213, 69,158, 33,252,
     5, 82,173,133,222,139,174, 27,  9, 71, 90,246, 75,130, 91,191,
   169,138,  2,151,194,235, 81,  7, 25,113,228,159,205,253,134,142,
   248, 65,224,217, 22,121,229, 63, 89,103, 96,104,156, 17,201,129,
    36,  8,165,110,237,117,231, 56,132,211,152, 20,181,111,239,218,
   170,163, 51,172,157, 47, 80,212,176,250, 87, 49, 99,242,136,189,
   162,115, 44, 43,124, 94,150, 16,141,247, 32, 10,198,223,255, 72,
    53,131, 84, 57,220,197, 58, 50,208, 11,241, 28,  3,192, 62,202,
    18,215,153, 24, 76, 41, 15,179, 39, 46, 55,  6,128,167, 23,188,
   106, 34,187,140,164, 73,112,182,244,195,227, 13, 35, 77,196,185,
    26,200,226,119, 31,123,168,125,249, 68,183,230,177,135,160,180,
    12,  1,243,148,102,166, 38,238,251, 37,240,126, 64, 74,161, 40,
   184,149,171,178,101, 66, 29, 59,146, 61,254,107, 42, 86,154,  4,
   236,232,120, 21,233,209, 45, 98,193,114, 78, 19,206, 14,118,127,
    48, 79,147, 85, 30,207,219, 54, 88,234,190,122, 95, 67,143,109,
   137,214,145, 93, 92,100,245,  0,216,186, 60, 83,105, 97,204, 52
   };
```

Once the gradient and permutation tables have been computed, a smoothed linear interpolation of the nearest mesh points is computed and is returned as the value of the function. The indices in the gradient table are taken as values in the 2D domain, and this interpolation uses the gradient values

to determine the function's values between the integer points in the grid. Note that in this code we introduce an alternative to the standard C `floor()` function, an interpolation function that calculates a value that lies a proportion `p` between `x0` and `x1` with `0 <= p < =1`, a smoothstep function of a value between 0 and 1 that provides a smooth transition between those values rather than a simple linear transition, and a function that calculates an address in the gradient table based on values from the permutation table.

```
#define FLOOR(x) ((int)(x) - ((x) < 0 && (x) != (int)(x)))
#define INTERP(p,x0,x1) ((x0)+(p)*((x1)-(x0)))
#define SMOOTHSTEP(x) ((x)*(x)*(3. - 2.*(x)))
#define PERM(x) perm[(x)&(TABSIZE-1)]
#define ADDR(i, j, k) (PERM((i) + PERM((j) + PERM((k)))));

float noise(float x, float y, float z) {
   int ix, iy, iz;
   float fx0, fx1, fy0, fy1, fz0, fz1,
         wx, wy, wz, vx0, vx1, vy0, vy1, vz0, vz1;

   ix = FLOOR(x);
   fx0 = x - ix;
   fx1 = fx0 - 1.;
   wx = SMOOTHSTEP(fx0);

   iy = FLOOR(y);
   fy0 = y - iy;
   fy1 = fy0 - 1.;
   wy = SMOOTHSTEP(fy0);

   iz = FLOOR(z);
   fz0 = z - iz;
   fz1 = fz0 - 1.;
   wz = SMOOTHSTEP(fz0);

   vx0=gradTab[ADDR(ix,iy,iz)][0]*fx0+
         gradTab[ADDR(ix,iy,iz)][1]*fy0+
         gradTab[ADDR(ix,iy,iz)][2]*fz0;
   vx1=gradTab[ADDR(ix+1,iy,iz)][0]*fx1+
         gradTab[ADDR(ix+1,iy,iz)][1]*fy0+
         gradTab[ADDR(ix+1,iy,iz)][2]*fz0;
   vy0=INTERP(wx, vx0, vx1);
   vx0=gradTab[ADDR(ix,iy+1,iz)][0]*fx0+
         gradTab[ADDR(ix,iy+1,iz)][1]*fy1+
         gradTab[ADDR(ix,iy+1,iz)][2]*fz0;
   vx1=gradTab[ADDR(ix+1,iy+1,iz)][0]*fx1+
         gradTab[ADDR(ix+1,iy+1,iz)][1]*fy1+
         gradTab[ADDR(ix+1,iy+1,iz)][2]*fz0;
   vy1=INTERP(wx, vx0, vx1);
   vz0=INTERP(wy, vy0, vy1);

   vx0=gradTab[ADDR(ix,iy,iz+1)][0]*fx0+
         gradTab[ADDR(ix,iy,iz+1)][1]*fy0+
         gradTab[ADDR(ix,iy,iz+1)][2]*fz1;
   vx1=gradTab[ADDR(ix+1,iy,iz+1)][0]*fx1+
         gradTab[ADDR(ix+1,iy,iz+1)][1]*fy0+
         gradTab[ADDR(ix+1,iy,iz+1)][2]*fz1;
   vy0=INTERP(wx, vx0, vx1);
   vx0=gradTab[ADDR(ix,iy+1,iz+1)][0]*fx0+
         gradTab[ADDR(ix,iy+1,iz+1)][1]*fy1+
```

```
            gradTab[ADDR(ix,iy+1,iz+1)][2]*fz1;
        vx1=gradTab[ADDR(ix+1,iy+1,iz+1)][0]*fx1+
            gradTab[ADDR(ix+1,iy+1,iz+1)][1]*fy1+
            gradTab[ADDR(ix+1,iy+1,iz+1)][2]*fz1;
        vy1=INTERP(wx, vx0, vx1);
        vz1=INTERP(wy, vy0, vy1);

        return INTERP(wz, vz0, vz1);
    }
```

The noise function will probably be defined initially on a relatively small domain with a fairly coarse set of integer vertices, and then the values in the full domain will be computed with the smoothing operations on the coarser definition. For example, if we define the original noise function N_0 on an 8x8x8 grid we could extend it to a function N_1 for a larger grid (say, 64x64x64) by defining $N_1(x,y,z) = N_0(x/8,y/8,z/8)$. We can thus calculate the values of the function for whatever size texture map we want, and either save those values in an array to be used immediately as a texture, or write those values to a file for future use. Note that this new function has a frequency 8 times that of the original, so given an original function we can easily create new functions with a larger or smaller frequency at will. And if we know that we will only want a 1D or 2D function, we can use a limited grid in the unneeded dimensions.

The noise function we have discussed so far is based on interpolating values that are set at fixed intervals, so it has little variation between these fixed points. A texture that has variations of several different sizes has more irregularity and thus more visual interest, and we can create such a texture from the original noise function in several ways. Probably the simplest is to combine noise functions with several different frequencies as noted above. However, in order to keep the high-frequency noise from overwhelming the original noise, the amplitude of the noise is decreased as the frequency is increased. If we consider the function $N_2 = A*N_0(Fx,Fy,Fz)$, we see a function whose amplitude is increased by a factor of A and whose frequency is increased by a factor of F in each direction. If we create a sequence of such functions with halved amplitude and doubled frequency and add them,

$$N(x,y,z) = \Sigma_k \, N_0(2^k x, 2^k y, 2^k z)/2^k$$

we get a function that has what is called 1/f noise that produces textures that behave like many natural phenomena. In fact, it is not necessary to compute many terms of the sum in order to get good textures; you can get good results of you calculate only as many terms as you need to get to the final resolution of your texture. Another approach, which is very similar but which produces a fairly different result, is the turbulence function obtained when the absolute value of the noise function is used, introducing some discontinuities to the function and table.

$$T(x,y,z) = \Sigma_k \, abs(N_0(2^k x, 2^k y, 2^k z))/2^k$$

And again, note that we are computing 3D versions of the 1/f noise and turbulence functions, and that you can take 2D or 1D subspaces of the 3D space to create lower-dimension versions. The images in Figure 11.1 show the nature of the noise functions discussed here.

So far, we have created noise and turbulence functions that produce only one value. This is fine for creating a grayscale texture and it can be used to produce colors by applying the single value in any kind of color ramp, but it is not enough to provide a full-color texture. For this we can simply treat three noise functions as the components of a 3D noise function that returns three values, and use these the RGB components of the texture. Depending on your application, either a 1D noise function with a color ramp or a 3D noise function might work best.

Figure 11.1a: a 2D graph of the original noise function (left) and of the 1/f noise derived from it (right)

Figure 11.1b: the surface representation of the original noise function

Antialiasing in texturing

When you apply a texture map to a polygon, you identify the vertices in the polygon with values in texture space. These values may or may not be integers (that is, actual indices in the texture map) but the interpolation process we discussed will assign a value in texture space to each pixel in the polygon. The pixel may represent only part of a texel (texture cell) if the difference between the texture-space values for adjacent pixels is less than one, or it may represent many texels if the difference between the texture space values for adjacent pixels is greater than one. This offers two kinds of aliasing — the magnification of texels if the texture is coarse relative to the object being texture mapped, or the selection of color from widely separated texels if the texture is very fine relative to the object.

Because textures may involve aliasing, it can be useful to have antialiasing techniques with texturing. In the OpenGL API, the only antialiasing tool available is the linear filtering that we discuss below, but other APIs may have other tools, and certainly sophisticated, custom-built or research graphics systems can use a full set of antialiasing techniques. This needs to be considered when considering the nature of your application and choosing your API. See [Ebert] for more details.

Texture mapping in OpenGL

There are many details to master before you can count yourself fully skilled at using textures in your images. The full details must be left to the manuals for OpenGL or another API, but here we will discuss many of them, certainly enough to give you a good range of skills in the subject. The details we will discuss are the texture environment, texture parameters, building a texture array, defining a texture map, and generating textures. We will have examples of many of these details to help you see how they work.

Capturing a texture from the screen

A useful approach to textures is to create an image and save the color buffer (the frame buffer) as an array that can be used as a texture map. This can allow you to create a number of different kinds of images for texture maps. This operation is supported by many graphics APIs. For example, in OpenGL, the `glReadBuffer(mode)` and `glReadPixels(...)` functions will define the buffer to be read from and then will read the values of the elements in that buffer into a target array. That array may then be written to a file for later use, or may be used immediately in the program as the texture array. We will not go into more detail here but refer the student to the manuals for the use of these functions.

Texture environment

The a graphics API, you must define your texture environment to specify how texture values are to be used when the texture is applied to a polygon. In OpenGL, the appropriate function call is

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, *)
```

The meaning of the texture is determined by the value of the last parameter. The options are `GL_BLEND`, `GL_DECAL`, `GL_MODULATE`, or `GL_REPLACE`.

If the texture represents *RGB color*, the behavior of the texture when it is applied is defined as follows. In this and the other behavior descriptions, we use C, A, I, and L for color, alpha, intensity, and luminance respectively, and subscripts f and t for the fragment and texture values.

`GL_BLEND`: the color of the pixel is $C_f(1-C_t)$.

`GL_DECAL`: the color of the pixel is C_t, simply replacing the color by the texture color.

`GL_MODULATE`: the color of the pixel is C_f*C_t,replacing the color by the subtractive computation for color.

`GL_REPLACE`: same as GL_DECAL for color.

If the texture represents *RGBA color*, then the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-C_t)$, and the alpha channel in the pixel is A_f*A_t.

`GL_DECAL`: the color of the pixel is $(1-A_t)C_f+A_tC_t$, and the alpha channel in the pixel is A_f.

`GL_MODULATE`: the color of the pixel is C_f*C_t, as above, and the alpha channel in the pixel is A_f*A_t.

`GL_REPLACE`: the color of the pixel is C_t and the alpha channel in the pixel is A_t.

If the texture represents the *alpha channel*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is C_f, and the alpha channel in the pixel is A_f.

`GL_DECAL`: the operation is undefined

`GL_MODULATE`: the color of the pixel is C_f, and the alpha channel in the pixel is A_f*A_t.

`GL_REPLACE`: the color of the pixel is C_f and the alpha channel in the pixel is A_t.

If the texture represents *luminance*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-L_t)$, and the alpha channel in the pixel is A_f.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is C_f*L_t, and the alpha channel in the pixel is A_f.

`GL_REPLACE`: the color of the pixel is L_t and the alpha channel in the pixel is A_f.

If the texture represents *intensity*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-I_t)$, and the alpha channel in the pixel is $A_f(1-I_t)$.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is C_f*I_t, and the alpha channel in the pixel is A_f*I_t.

`GL_REPLACE`: the color of the pixel is I_t and the alpha channel in the pixel is I_t.

Texture parameters

The texture parameters define how the texture will be presented on a polygon in your scene. In OpenGL, the parameters you will want to understand include texture wrap and texture filtering. Texture wrap behavior, defined by the `GL_TEXTURE_WRAP_*` parameter, specifies the system behavior when you define texture coordinates outside the [0,1] range in any of the texture dimensions. The two options you have available are repeating or clamping the texture. Repeating the texture is accomplished by taking only the decimal part of any texture coordinate, so after you go beyond 1 you start over at 0. This repeats the texture across the polygon to fill out the texture space you have defined. Clamping the texture involves taking any texture coordinate outside [0,1] and translating it to the nearer of 0 or 1. This continues the color of the texture border outside the region where the texture coordinates are within [0,1]. This uses the `glTexParameter*(...)` function to repeat, or clamp, the texture respectively as follows:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
```

Mixing the parameters with horizontal repetition and vertical clamping produces an image like that of Figure 11.2.

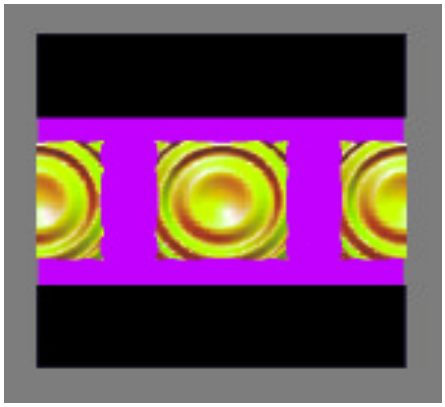

Figure 11.2: a polygon face with a texture that is wrapped in one direction and clamped in the other

Another important texture parameter controls the filtering for pixels to deal with aliasing issues. In OpenGL, this is called the minification (if there are many texture points that correspond to one pixel in the image) or magnification (if there are many pixels that correspond to one point in the texture) filter, and it controls the way an individual pixel is colored based on the texture map. For any pixel in your scene, the texture coordinate for the pixel is computed through an interpolation

across a polygon, and rarely corresponds exactly to an index in the texture array, so the system must create the color for the pixel by a computation in the texture space. You control this in OpenGL with the texture parameter `GL_TEXTURE_*_FILTER` that you set in the `glTexParameter*(...)` function. The filter you use depends on whether a pixel in your image maps to a space larger or smaller than one texture element. If a pixel is smaller than a texture element, then `GL_TEXTURE_MIN_FILTER` is used; if a pixel is larger than a texture element, then `GL_TEXTURE_MAG_FILTER` is used. An example of the usage is:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

The symbolic values for these filters are `GL_NEAREST` or `GL_LINEAR`. If you choose the value `GL_NEAREST` for the filter, then the system chooses the single point in the texture space nearest the computed texture coordinate; if you choose `GL_LINEAR` then the system averages the four nearest points to the computed texture coordinate with weights depending on the distance to each point. The former is a faster approach, but has problems with aliasing; the latter is slower but produces a much smoother image. This difference is illustrated in Figure 11.3 in an extreme close-up, and it is easy to see that the `GL_NEAREST` filter gives a much coarser image than the `GL_LINEAR` filter. Your choice will depend on the relative importance of speed and image quality in your work.

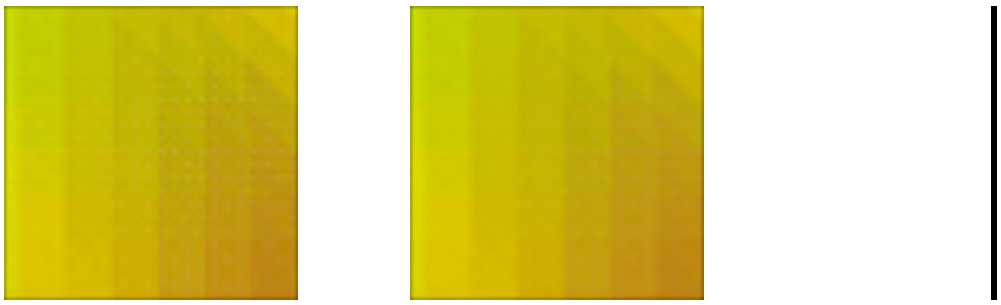

Figure 11.3: a texture zoomed in with `GL_NEAREST` (left) and `GL_LINEAR` (right) filter

Getting and defining a texture map

This set of definitions is managed by the `glTexImage*D(...)` functions. These are a complex set of functions with a number of different parameters. The functions cover 1D, 2D, and 3D textures (the dimension is the asterix in the function name) and have the same structure for the parameters.

Before you can apply the `glTexImage*D(...)` function, however, you must define and fill an array that holds your texture data. This array of unsigned integers (`GLuint`) will have the same dimension as your texture. The data in the array can be organized in many ways, as we will see when we talk about the internal format of the texture data. You may read the values of the array from a file or you may generate the values through your own programming. The examples in this chapter illustrate both options.

The parameters of the `glTexImage*D(...)` function are, in order,

- the *target*, usually `GL_TEXTURE_*D`, where * is 1, 2, or 3. Proxy textures are also possible, but are beyond the range of topics we will cover here. This target will be used in a number of places in defining texture maps.
- the *level*, an integer representing level-of-detail number. This supports multiple-level MIP-mapping.

- the *internal format* of the texture map, one of the places where an API such as OpenGL must support a large number of options to meet the needs of a wide community. For OpenGL, this internal format is a symbolic constant and can vary quite widely, but we will list only a set we believe will be most useful to the student. Most of the other options deal with other organizations that involve a different number of bits per pixel of the component. Here we deal only with formats that have eight bits per component, and we leave the others (and information on them in manuals) to applications that need specialized formats.
  - `GL_ALPHA8`
  - `GL_LUMINANCE8`
  - `GL_INTENSITY8`
  - `GL_RGB8`
  - `GL_RGBA8`
- the *dimensions* of the texture map, of type `GLsizei`, so the number of parameters here is the dimension of the texture map. If you have a 1D texture map, this parameter is the *width*; if you have a 2D texture map, the two parameters are the width and *height*; if you have a 3D texture map, the three parameters are width, height, and *depth*. Each of these must have a value of `2N+2(border)` where the value of border is either 0 or 1 as specified in the next parameter.
- the *border*, an integer that is either 0 (if no border is present) or 1 (if there is a border).
- the *format*, a symbolic constant that defines what the data type of the pixel data in the texture array is. This includes the following, as well as some other types that are more exotic:
  - `GL_ALPHA`
  - `GL_RGB`
  - `GL_RGBA`
  - `GL_LUMINANCE`

  The format indicates how the texture is to be used in creating the image. We discussed the effects of the texture modes and the texture format in the discussion of image modes above.
- the *type* of the pixel data, a symbolic constant that indicates the data type stored in the texture array per pixel. This is usually pretty simple, as shown in the examples below which use only `GL_FLOAT` and `GL_UNSIGNED_BYTE` types.
- the pixels, an address of the pixel data (texture array) in memory.

You will be creating your textures from some set of sources and probably using the same kind of tools. When you find a particular approach that works for you, you'll most likely settle on that particular approach to textures. The number of options in structuring your texture is phenomenal, as you can tell from the number of options in some of the parameters above, but you should not be daunted by this broad set of possibilities and should focus on finding an approach you can use.

Texture coordinate control

As your texture is applied to a polygon, you may specify how the texture coordinates correspond to the vertices with the `glTexture*(...)` function, as we have generally assumed above, or you may direct the OpenGL system to assign the texture coordinates for you. This is done with the `glTexGen*(...)` function, which allows you to specify the details of the texture generation operation.

The `glTexGen*(...)` function takes three parameters. The first is the texture coordinate being defined, which is one of `GL_S`, `GL_T`, `GL_R`, or `GL_Q` with S, T, R, and Q being the first, second, third, and homogeneous coordinates of the texture. The second parameter is one of three symbolic constants: `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. If the second parameter is `GL_TEXTURE_GEN_MODE`, the third parameter is a single symbolic constant with value `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, or `GL_SPHERE_MAP`. If the second parameter is `GL_OBJECT_PLANE`, the third parameter is a vector of four values that defines the plane from which an object-linear texture is defined; if the second parameter is `GL_EYE_PLANE`, the third parameter is a vector of four values that defines the plane that contains

the eye point. In both these cases, the object-linear or eye-linear value is computed based on the coefficients. If the second parameter is GL_TEXTURE_GEN_MODE and the third parameter is GL_SPHERE_MAP, the texture is generated based on an approximation of the reflection vector from the surface to the texture map.

Applications of this texture generation include the Chromadepth™ texture, which is a 1D eye-linear texture generated with parameters that define the starting and ending points of the texture. Another example is automatic contour generation, where you use a GL_OBJECT_LINEAR mode and the GL_OBJECT_PLANE operation that defines the base plane from which contours are to be generated. Because contours are typically generated from a sea-level plane (one of the coordinates is 0), it is easy to define the coefficients for the object plane base.

Texture mapping and GLU quadrics

As we noted in the chapter on modeling, the GLU quadric objects have built-in texture mapping capabilities, and this is one of the features that makes them very attractive to use for modeling. To use these, we must carry out three tasks: load the texture to the system and bind it to a name, define the quadric to have normals and a texture, and then bind the texture to the object geometry as the object is drawn. The short code fragments for these three tasks are given below, with a generic function readTextureFile(...) specified that you will probably need to write for yourself, and with a generic GLU function to identify the quadric to be drawn.

```
readTextureFile(...);
glBindTexture(GL_TEXTURE_2D, texture[i]);
glTexImage2D(GL_TEXTURE_2D,...);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

myQuadric = gluNewQuadric();
gluQuadricNormals(myQuadric, GL_SMOOTH);
gluQuadricTexture(myQuadric, GL_TRUE);
gluQuadricDrawStyle(myQuadric, GLU_FILL);

glPushMatrix();
  // modeling transformations as needed
  glBindTexture(GL_TEXTURE_2D, texture[i]);
  gluXXX(myQuadric, ...);
glPopMatrix();
```

Some examples

Textures can be applied in several different ways with the function

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode )
```

One way uses a decal technique, with mode GL_DECAL, in which the content of the texture is applied as an opaque image on the surface of the polygon, showing nothing but the texture map. Another way uses a modulation technique, with mode GL_MODULATE, in which the content of the texture is displayed on the surface as though it were colored plastic. This mode allows you to show the shading of a lighted surface by defining a white surface and letting the shading show through the modulated texture. There is also a mode GL_BLEND that blends the color of the object with the color of the texture map based on the alpha values, just as other color blending is done. In the examples below, the Chromadepth image is created with a 1D modulated texture so that the underlying surface shading is displayed, while the mapped-cube image is created with a 2D decal texture so that the face of the cube is precisely the texture map. You may use several different textures with one image, so that (for example) you could take a purely geometric white terrain

model, apply a 2D texture map of an aerial photograph of the terrain with `GL_MODULATE` mode to get a realistic image of the terrain, and then apply a 1D texture map in `GL_BLEND` mode that is mostly transparent but has colors at specific levels and that is oriented to the vertical in the 3D image in order to get elevation lines on the terrain. Your only limitation is your imagination — and the time to develop all the techniques.

The Chromadepth™ process: using 1D texture maps to create the illusion of depth. If you apply a lighting model with white light to a white object, you get a pure expression of shading on the object. If you then apply a 1D texture by attaching a point near the eye to the red end of the ramp (see the code below) and a point far from the eye to the blue end of the ramp, you get a result like that shown in Figure 11.4 below.

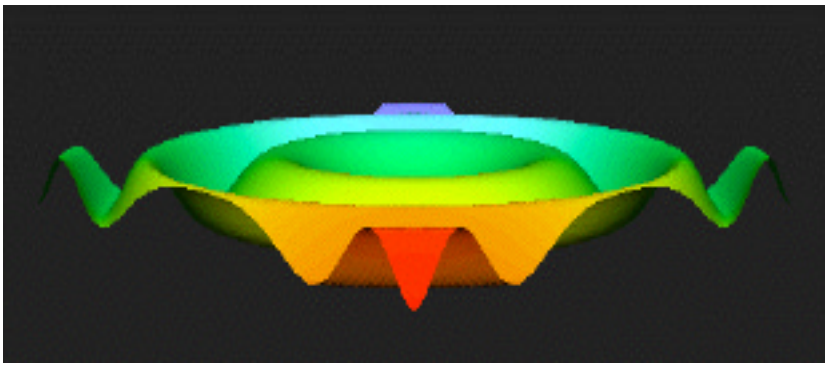

Figure 11.4: a Chromadepth-colored image of a mathematical surface

This creates a very convincing 3D image when it is viewed through Chromadepth™ glasses, because these glasses have a diffraction grating in one lens and clear plastic in the other. The diffraction grating bends red light more than blue light, so the angle between red objects as seen by both eyes is larger than the angle between blue objects. Our visual system interprets objects having larger angles between them as closer than objects having smaller angles, so with these glasses, red objects are interpreted as being closer than blue objects.

Using 2D texture maps to add interest to a surface: often we want to create relatively simple objects but have them look complex, particularly when we are trying to create models that mimic things in the real world. We can accomplish this by mapping images (for example, images of the real world) onto our simpler objects. In the very simple example shown in Figure 11.5, a screen capture and some simple Photoshop work created the left-hand image, and this image was used as

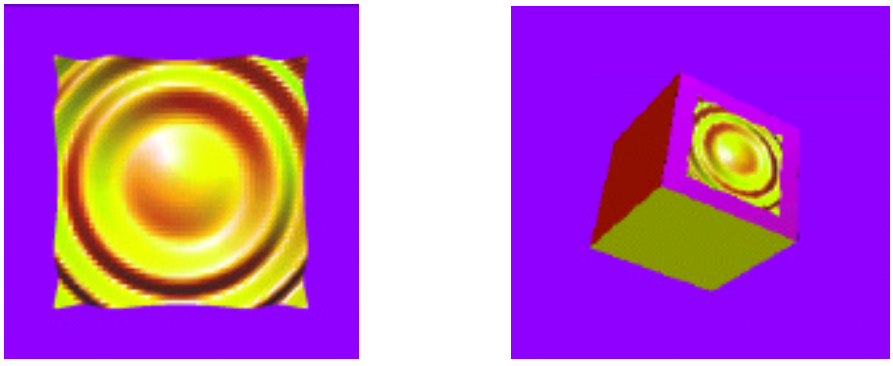

Figure 11.5: a texture map (left) and a 3D cube with the texture map placed on one face (right)

the texture map on one face of the cube in the right-hand image. (The boundary around the function surface is part of the texture.) This texture could also have been created by saving the frame buffer into a file in the program that created the mathematical surface. This created a cube that has more visual content than its geometry would suggest, and it was extremely simple to connect the square image with the square face of the cube.

Environment maps

Environment maps allow us to create the illusion that an object reflects images from a texture that we define. This can provide some very interesting effects, because realistic reflections of real-world objects is one of the visual realism clues we would expect. With environment maps, we can use photographs or synthetic images as the things we want to reflect, and we can adapt the parameters of the texture map to give us realistic effects. One of the easy effects to get is the reflection of things in a chrome-like surface. In Figure 11.6, we see an example of this as a photograph of Hong Kong that has been modified with a very wide-angle lens filter is used as a texture map on a surface. The lens effect makes the environment map much more convincing because the environment map uses the surface normals at a point to identify the texture points for the final image.

Figure 11.6: the original texture for an environment map (left) and the map on a surface (right)

A word to the wise...

Texture mapping is a much richer subject than these fairly simple examples have been able to show. You can use 1D textures to provide contour lines on a surface or to give you the kind of color encoding for a height value we discussed in the module on visual communication. You can use 2D textures in several sophisticated ways to give you the illusion of bumpy surfaces (use a texture on the luminance), to give the effect of looking through a varigated cloud (use a fractal texture on alpha) or of such a cloud on shadows (use the same kind of texture on luminance on a landscape image). This subject is a fruitful area for creative work.

There are several points that you must consider in order to avoid problems when you use texture mapping in your work. If you select your texture coordinates carelessly, you can create effects you might not expect because the geometry of your objects does not match the geometry of your texture map. One particular case of this is if you use a texture map that has a different aspect ratio than the space you are mapping it onto, which can change proportions in the texture that you might not have expected. More serious, perhaps, is trying to map an entire rectangular area into a quadrilateral that isn't rectangular, so that the texture is distorted nonlinearly. Imagine the effect if you were to try to map a brick texture into a non-convex polygon, for example. Another problem can arise if you texture-map two adjacent polygons with maps that do not align at the seam between

the polygons. Much like wallpaper that doesn't match at a corner, the effect can be disturbing and can ruin any attempt at creating realism in your image. Finally, if you use texture maps whose resolution is significantly different from the resolution of the polygon using the texture, you can run into problems of aliasing textures caused by selecting only portions of the texture map. We noted the use of magnification and minification filters earlier, and these allow you to address this issue.

In a different direction, the Chromadepth™ 1D texture-mapping process gives excellent 3D effects but does not allow the use of color as a way of encoding and communicating information. It should only be used when the shape alone carries the information that is important in an image, but it has proved to be particularly useful for geographic and engineering images, as well as molecular models.

Code examples

First example: Sample code to use texture mapping in the first example is shown below. The declaration set up the color ramp, define the integer texture name, and create the array of texture parameters.

```
float D1, D2;
float texParms[4];
static GLuint texName;
float ramp[256][3];
```

In the `init()` function we find the following function calls that define the texture map, the texture environment and parameters, and then enables the texture generation and application.

```
makeRamp();
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexImage1D( GL_TEXTURE_1D, 0, 3, 256, 0, GL_RGB, GL_FLOAT, ramp );
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_1D );
```

The `makeRamp()` function is defined to create the global array `ramp[]` that holds the data of the texture map. This process works with the HSV color model in which hues are defined through angles (in degrees) around the circle which has saturation and value each equal to 1.0. The use of the number 240 in the function comes from the fact that in the HSV model, the color red is at 0 degrees and blue is at 240 degrees, with green between at 120 degrees. Thus an interpolation of fully-saturated colors between red and blue will use the angles between 0 and 240 degrees. The RGB values are calculated by a function `hsv2rgb(...)` that is a straightforward implementation of standard textbook color-model conversion processes. The Foley et al. textbook in the references is an excellent resource on color models (see Chapter 13).

```
void makeRamp(void)
{
   int i;
   float h, s, v, r, g, b;

// Make color ramp for 1D texture: starts at 0, ends at 240, 256 steps
   for (i=0; i<256; i++) {
      h = (float)i*240.0/255.0;
      s = 1.0; v = 1.0;
```

```
      hsv2rgb( h, s, v, &r, &g, &b );
      ramp[i][0] = r; ramp[i][1] = g; ramp[i][2] = b;
   }
}
```

Finally, in the `display()` function we find the code below, where `ep` is the eye point parameter used in the `gluLookAt(...)` function. This controls the generation of texture coordinates, and binds the texture to the integer name `texName`. Note that the values in the `texParms[]` array, which define where the 1D texture is applied, are defined based on the eye point, so that the image will be shaded red (in front) to blue (in back) in the space whose distance from the eye is between D1 and D2.

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
D1 = ep + 1.0; D2 = ep + 10.0;
texParms[0] = texParms[1] = 0.0;
texParms[2] = -1.0/(D2-D1);
texParms[3] = -D1/(D2-D1);
glTexGenfv( GL_S, GL_EYE_PLANE, texParms);
glBindTexture(GL_TEXTURE_1D, texName);
```

Second example: Sample code to use texture mapping in the second example is shown in several pieces below. To begin, in the data declarations we find the declarations that establish the internal texture map (`texImage`) and the set of texture names that can be used for textures (`texName`).

```
#define TEX_WIDTH 512
#define TEX_HEIGHT 512
static GLubyte texImage[TEX_WIDTH][TEX_HEIGHT][3];
static GLuint texName[1]; // parameter is the number of textures used
```

In the init function we find the `glEnable` function that allows the use of 2D textures.

```
glEnable(GL_TEXTURE_2D);  // allow 2D texture maps
```

You will need to create the texture map, either through programming or by reading the texture from a file. In this example, the texture is read from a file named `myTexture.rgb` that was simply captured and translated into a raw RGB file, and the function that reads the texture file and creates the internal texture map, called from the init function, is

```
void setTexture(void)
{
   FILE * fd;
   GLubyte ch;
   int i,j,k;

   fd = fopen("myTexture.rgb", "r");
   for (i=0; i<TEX_WIDTH; i++)     // for each row
   {
      for (j=0; j<TEX_HEIGHT; j++) // for each column
      {
         for (k=0; k<3; k++) // read RGB components of the pixel
         {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
         }
      }
   }
   fclose(fd);
```

```
}
```

Finally, in the function that actually draws the cube, called from the `display()` function, we first find code that links the texture map we read in with the texture number and defines the various parameters of the texture that will be needed to create a correct display. We then find code that draws the face of the cube, and see the use of texture coordinates along with vertex coordinates. The vertex coordinates are defined in an array `vertices[]` that need not concern us here.

```
glGenTextures(1, texName);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, TEX_WIDTH, TEX_HEIGHT,
             0, GL_RGB, GL_UNSIGNED_BYTE, texImage);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glBegin(GL_QUADS);
   glNormal3fv(normals[1]);
   glTexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
   glTexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
   glTexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
   glTexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
glEnd();
glDeleteTextures(1,texName);
```

Third example: The third example also uses a 2D texture map, modified in Photoshop to have a fish-eye distortion to mimic the behavior of a very wide-angle lens. The primary key to setting up an environment map is in the texture parameter function, where we also include two uses of the `glHint(...)` function to show that you can define really nice perspective calculations and point smoothing — with a computation cost, of course. But the images in Figure 11.5 suggest that it might be worth the cost sometimes.

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
...
// the two lines below generate an environment map in both the S and T
// texture coordinates
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

References

Chromadepth information is available from
Chromatek Inc
1246 Old Alpharetta Road
Alpharetta, GA 30005
888-669-8233
http://www.chromatek.com/

Chromadepth glasses may be ordered from
American Paper Optics
3080 Bartlett Corporate Drive
Bartlett, TN 38133
800-767-8427, 901-381-1515
fax 901-381-1517

Ebert, David et al., *Texturing and Modeling: a Procedural Approach*, second edition, Academic Press, 1998

Foley, James D. et al., *Computer Graphics Principles and Practice*, second edition, Addison-Wesley, 1990

Murray, James D. and William vanRyper, *Encyclopedia of Graphics File Formats*, second edition, O'Reilly & Associates, 1996

Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3), Proceedings of SIGGRAPH 85, July 1985, 287-296

Wolfe, R. J., *3D Graphics: A Visual Approach*, Oxford University Press, 2000