Chapter 9: Texture Mapping

Of all the techniques we will see in this book, texture mapping offers the best results for creating realistic and exciting images. In this chapter we will look at the concept of texture mapping with 1D, 2D, and 3D textures and will see several examples the effects you can create with 1D and 2D texture maps. We will also consider how textures can be created from both natural and synthetic images. Finally, we will look at how the OpenGL graphics API works with texture maps and give some working examples of code fragments for using texture maps in graphics programming. In order to benefit from this chapter, you need to have an understanding of the geometry of polygons in 3-space and the concept of how values in one space can map linearly to values in another space.

Introduction

Texture mapping involves applying textures to a graphical object to achieve a more interesting image or add information to the image without computing additional geometry. This is a significant addition to the capabilities of computer graphics and is a very useful technique for you to master. Texturing is a rich topic and we will not try to cover it in all the depth that is possible, but we will describe the most useful set of capabilities as we develop the subject in a way that is compatible with current graphics APIs. This will allow you to use texture mapping effectively in your work.

The key idea of texture mapping is to provide additional visual information for your images as the geometry is computed and displayed. In the API environment we are assuming, the geometry is primarily based on polygons, and as the pixels of the polygon are computed, the color of each pixel is calculated not only from a simple lighting model, but by including information from an array of values called a texture map. This array can represent colors from natural or synthetic images, as is usually done, but can also represent other kinds of information such as transparency or intensity.

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 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.

Creating texture maps from images is not, however, the only approach that can be used in texture mapping. 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 cover in depth 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.

In order to reconcile the two spaces used in texture mapping, and to have the information needed to develop the texture information to apply to each fragment, the system must be given values for many individual parameters. You can think of this parameter setting as a binding operation, and it will be done by several API functions. Among the things you will need to set are

- the name (usually a small integer) given to the texture in internal texture memory,
- the dimensions of the texture map, the format of the information the texture map contains,
- what the texture map represents (a texture map may represent more than simply color),
- the way the texture and fragment information are to be combined when a fragment is rendered with the texture,
- how the texture is to be treated if the texture coordinates go outside the basic texture space,
- how the texture aliasing is to be handled when the texture is applied to each fragment, and
- whether the texture has a border.

You should look for all these bindings, and quite likely others, when you consider how your graphics API handles texture mapping.

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. The graphics API provides ways to interpolate the vertices from the texture array and to compute the value for
the pixel based on the colors at the interpolated vertices. These range from choosing the nearest point in the texture array to averaging the 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. Another view of this could be that the image is on an elastic sheet and it is tacked onto the surface by pinning certain points of the sheet onto the vertices of the 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 1.1, 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 references [SHR] [WOO]. A useful visual examination of 3D textures is found in [WO00]. 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.

As you define your geometry, you associate a point in texture space with each vertex. This is similar to the way you associate a normal with each vertex when you set up lighting with smooth shading. This now has even more information associated with each point: the geometry of the vertex by its coordinates, the color of the vertex or the normal for the vertex that allows the color to be computed, and the coordinates of the texture point that’s associated with the vertex. The vertex coordinates are used in the rendering pipeline to determine the pixel coordinates for the vertex, and the color, normal, and texture information are used to determine the appearance of the pixels within the vertex geometry in the fragment processing step of the rendering pipeline.

Depending on the way your graphics API works, you may either associate each vertex with actual texel coordinates of the texture point or a point with real coordinates, usually each in [0, 1], that represents a point by its proportional location in the texture map. The latter real-number approach is preferable because it allows your design for the use of textures to be independent of the actual texture map size.

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.

**Other meanings for texture maps**

Texture maps can describe other things besides an image that is to be mapped onto an object. A texture map can be used to change the appearance of a polygon by modifying the alpha value, luminance, or intensity of the pixels in the polygon based on the values in the texture map. This gives you a number of ways you can alter the appearance of a polygon by changing the way it is presented. This can be especially effective if it is used as part of multitexturing.

**Texture mapping in the scene graph**

A texture is basically an appearance property for a geometry object, so it should be part of the appearance part of a geometry node. However, the geometry node itself must represent the texture coordinates that correspond to each vertex, so texture must be involved in the geometry node.

The most straightforward way to achieve this is consider all the details of the texture map, such as are used by OpenGL and are discussed below, in the appearance part of the node. This is typically expressed before the geometry of the node is defined for the API, as would be done with the material and shading definitions as part of the appearance node. Then the geometry node would be expressed, but would need to include texture coordinates for each vertex, just as using a lighting model would need the geometry node to express the normal for each vertex of the geometry. As the scene graph is traversed, either automatically or manually, the code that expresses the appearance would then be placed before the code that expresses the geometry. This is relatively straightforward and should pose few problems for the student.

**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 at this time, all such textures are 2D textures because there are no generally-accepted formats for compressed 3D images.

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 [MUR], and some formats include compression techniques that require a great deal of computation when you re-create the image from the file. 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. However, many implementations of graphics APIs are starting to include the ability to read images in various formats. Unless you have such functions available, 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. If you want to use an image that you have in a compressed file format, probably the simplest approach is to open the image in a highly-capable image manipulation tool such as Photoshop, which can read images in most formats, and then re-save it in a simplified form such as interlaced raw RGB.

A sample image that we will use as a texture map, a picture of a group of African penguins created from one of the author’s photographs, is shown in Figure 9.1. Graphics APIs are likely to have restrictions on the dimensions of texture maps (for example, the OpenGL standard requires all dimensions, not including borders, to be a power of 2) 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, and the process of using an image file as a texture map is described in a code example later in this chapter.

Figure 9.1: an image that will be used as a texture map in several examples below

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. For example, if we want to build a 64x64 texture array, we 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. This kind of texture map is often used as a texture-mapped image because it shows problems easily, and such a texture map applied to two rectangles in space is shown in Figure 9.2.

A particularly useful kind of computed texture involves using 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. There are a number of ways to create such functions, and we will not begin to explore them all, but we will take one relatively simple approach to defining a noise function and use it to generate a couple of texture maps.

Instead of starting with a noise function itself, let’s look at a simple texture map that has some of the properties of a noise function: no correlation for rotation or translation. If we generate a random number between 0 and 1 at each point of the texture map, then the nature of random numbers would give us the lack of correlation we need. But there is also no correlation between nearby elements of the texture map, so the purely random texture map is not satisfactory for many uses.

To give us a smoother, but still uncorrelated, random texture, we can apply the kind of filter function that we saw in the examples of diffusion processes in the science examples chapter. Recall that this filter replaces the value of each pixel with a weighted sum of the values of the pixels near that pixel, creating a weighted average of these values. If we start with a random texture and apply the filter process several times, we get a smoother texture that will still have the uncorrelated properties we want. This filtering can be done as often as you like, and the more often it is applied, the smoother the resulting texture. These processes are straightforward, and the results of 2D versions of the processes are shown in Figure 9.3. The texture can be created in grayscales, as shown, or colored textures can be created by creating similar textures for each of the RGB components in the color.
The random textures defined above are examples of noise functions that would be called *white noise*, with random values between 0 and 1. However, many other kinds of noise are possible, and one important one for texture maps is called *1/f noise*. This is a noise function which is built from a linear combination of white noise functions $f_N$ at various frequencies $N$, with the amplitude of $f_N$ defined to be $1/N$. If we take these functions at frequencies $2^N$ for positive values of $N$, then the amplitude of the sum of the functions is $\sum (1/N)$, with the sum taken over all powers of two; this sum is 1, so the combined function also takes values from 0 to 1, just as the individual functions did. A texture map made from this technique has both large-scale properties (low frequencies) and small-scale details (high frequencies) and can be used to model some natural phenomena.

To see how this works, let’s work our way through a simple 1D example. A noise function will be thought of as a piecewise linear function with values in $[0,1]$, defined on the interval $[0,1]$. The *frequency* of a noise function on an interval can be defined to be the number of separate linear segments the function has over the interval. So if a function is defined by values at 0.0, 0.5, and 1.0, its frequency is 2, while if the function is defined by values at 0.0, 0.25, 0.5, 0.75, and 1.0, its frequency is 4. It should be clear how to get functions with frequencies 8, 16, or any other power of 2: for frequency $2^N$, the function $f_N$ could be piecewise linear with values defined at $x=il2^N$ for all values of $i$ from 0 to $2^N$. Figure 9.4 shows a very simple case (a piecewise linear function of one variable from 0 to 16, which could be used to create a 1D texture map 16 pixels wide) where the left-hand column is the graphs of the individual $f_N$ functions for $N = 2, 4, 8, \text{ and } 16$, the center column is the functions $f_N/2^N$, and the right-hand side is the sum of the functions in the center column. Obviously this is a very simple example and a more useful noise function—or noise texture map—would be defined over a larger interval and would involve more piecewise linear functions.

![Figure 9.4: noise functions of various frequencies (left), multiplied by the reciprocals of their frequencies (center), and summed to create a single noise function (right).](image)

In most cases, the function or texture map would use functions on 2D or 3D space instead of functions on 1D space. In the 2D case, you could use polygons defined by vertices with regular $(x,y)$ coordinates and random $z$-coordinates, giving a set of points on a regular grid just as we described for graphing a 2D function in the chapter on science examples. In the 3D case you would need to compute values across a cubic region of space defined by eight coordinates in 3D.
space, which is much more difficult to visualize. But you can still work with functions of varying
frequencies and sum them with varying weights, producing $1/f$ noise functions in 2D or 3D space.

In practice, the approach that is generally used is much more sophisticated than we have described.
This approach uses gradient interpolation as discussed by Peachy in [Ebert]. This is also the kind
of noise function used in the Renderman™ shader system. We will not describe this in detail, but
you are encouraged to work through the sample code for such noise functions provided by Mike
Bailey that is included in the supplementary materials for this book. Figure 9.5 is an example of a
texture built with such a process. As an overview, let’s consider this in the 3D noise case.

The general process is to start with a 3D mesh of the right frequency for a particular function $f_N$.
We will treat the coordinates of each 3D mesh point as $x$, $y$, and $z$-components of a point in the
noise function domain, and the process computes a unit vector of three random components that
represents the gradient at that point. These three components are the direction and amplitude of the
gradient. 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. You are encouraged to read the discussions in
[Ebert] to understand this process and some of the issues in using it effectively and efficiently.

![Figure 9.5: a texture from a 1/f noise function](image)

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-dimensional space that is always rotated to face the viewer and
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. The geometric principles of billboards are discussed a bit later in this chapter.

Because natural images that might be used for billboards do not come with an alpha channel, you
might have some work to do in creating a texture to be used as a billboard. We suggest that you
edit any such image to have a background color and then adjust the alpha values as you read in the
image: if a pixel is not the background color, leave the pixel’s RGB alone and set the alpha to 1.0
for maximum opacity. If the pixel is the background color, set the alpha to 0.0 so the color of the
pixel will be ignored in blending. This is similar to the green-screen technique for television or
film compositing.

Including multiple textures in one texture map

In many graphics APIs, as we will see in the discussion of texture maps in OpenGL below, you
can have several texture maps resident in your system and switch between them to use different
textures in different parts of your image. However, sometimes you may want to have more textures available than you would be able to handle separately. An example of this situation could be billboard ing, where you might want to create labels with billboards but have a number of different labels in your scene. In this case you could create a single texture map with all the labels you would use, and then select the individual label by choosing your texture coordinates to include only the area of the texture map with that particular content.

Selecting only a part of an image for a texture map can be useful in other contexts as well. If you have a non-rectangular part of a photo that you want to use as a texture, for example, you can read the whole photo into texture memory and use texture coordinates to select only the part that is important to you. This lets you get around the problem of only having rectangular textures.

**Interpolation for texture maps**

A 2D texture-mapped polygon is created by interpolating the texture coordinates for each pixel in the polygon from the texture coordinates of the polygon vertices in the rendering pipeline. As we noted in the chapter on that pipeline, if the scene uses a perspective projection, the highest quality texturing is done if the interpolation takes the perspective into account by back-projecting the 2D coordinates for each pixel into the original modeling space before doing the interpolation. If this perspective correction is not done, the texture in each polygon follows a linear pattern based on the polygon boundaries, which can cause awkward artifacts at polygon boundaries. In Figure 9.6, we see a quad defined as two triangles with a checkerboard texture, and we see that in the triangle at lower left, all the lines in the texture are parallel to the left or bottom edges while all the lines in the upper right triangle are parallel to the right or top edges. But because the quad is shown with a perspective projection, the left and right sides are not parallel, leading to problems with the texture where the triangles meet. This illustrates the difference that perspective-corrected interpolation can make for texture mapping. Similar problems can occur for 1D or 3D texture maps in a perspective projection situation when perspective correction is not used. This figure also shows how readily the checkerboard texture shows problems in textured images, as we suggested earlier in the chapter.

![Figure 9.6: a planar rectangular region defined by two triangles without (left) and with (right) perspective correction for the texture mapping](image)

The technique for doing perspective-corrected texture interpolation is simply a special case of the perspective-corrected interpolation discussed in the chapter on the graphics pipeline. Of course, this problem will not occur if you use an orthogonal projection because with this projection, the linear relation between pixels in the screen space is exactly mirrored by the linear relation between the original points in model space.
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 (handled by a magnification filter), or the selection of color from widely separated texels if the texture is very fine relative to the object (handled by a minification filter). Magnification tends to lead to blocky textures as a texel’s effect is felt over many pixels. Minification tends to lead to gaps in the texture where some texels will not be included in coloring pixels. Magnification and minification filters are used to minimize these effects.

Because textures may involve aliasing, it can be useful to have antialiasing techniques with texturing. For magnification filtering, you will find yourself with pixel coordinates often having two adjacent points within the same texel. You can choose to use the nearest filter to determine the color of a pixel: the color is set to the color of the nearest texel vertex. This can alias a number of pixels to the color of a single texel vertex, and can give you a blocky image. Another approach is to use linear filtering, where each pixel’s color is determined by a weighted average of the texel vertices around it, with the weight being determined by how close the pixel is to each texel vertex. Other, more sophisticated kinds of antialiasing techniques are also possible, but graphics APIs tend to keep things simple. 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.

MIP Mapping

We saw that when there is only a single texture map available, the graphics system must sometimes use some sort of antialiasing process to choose the color of a pixel from the colors of pixels in the original map. If the pixel space of a polygon is larger than that of the texture map, there is no way to get individual texel information for each pixel and techniques such as linear filtering are needed. But as a polygon gets small, the pixel space gets smaller than the texture space and you will find that pixels that are near each other in the polygon have colors that are not near each other in the texture space. As these polygons move, the colors can jump around unpredictably, causing unwanted effects.

A solution to this problem can be provided by giving your system a hierarchy of texture maps of different sizes, and having the system select the map that best fits the size of your polygon. One technique for doing this is MIP mapping (MIP means multum in parvo, or “many things in a small place.”) With this approach, you provide your texture map in many resolutions so you can control the versions of the texture that will be seen at each level. This set of different resolution maps is all held the same texture memory and the proper one is selected depending on the size of the polygon to be presented.

MIP mapping can be seen as a level-of-detail process (see the chapter on high-performance graphics) but it is used less for performance reasons than for quality reasons. Thus we believe it fits best in the discussion here of providing quality texture mapping.
Multitexturing

Multitexturing is a rendering technique in which two or more textures are applied to a single surface in the rendering process, applying more than one texture environment to the model as shown in Figure 9.7. For example, one texture might be a wood surface and a second texture might be a light map. The combination of the two textures would produce a texture of a lighted wooden surface. This use of surface and light maps is a common technique in games programming. Other examples might include combining aerial photographs, GIS (geographic information systems) locator symbols, and elevation contour lines to produce a map that combines realistic terrain, points of interest, and elevation information.

![Figure 9.7: texturing in the graphics pipeline: single texture (left) and multitexture (right)](image)

With standard texturing capabilities you might be able to achieve similar results. It may be possible to combine separate texture maps into a single texture map if the separate maps are of the same size and share the same texture coordinates on the target to be textured. You would simply read the values in the individual texture arrays and combine them with appropriate operations into a new texture array that you could use as your single texture. However, this is very restrictive, because texture data usually does not come in such nice packages, and separate texture maps may be of different sizes and with different orientations relative to the surface to be textured. So having a multitexture capability allows you to use each texture with its own individual properties, and the combination you need will be applied in the rendering process.

At this point, multitexturing is not part of many graphics API standards, but it is common in some development systems and can be expected to become part of either API standards or API extensions. This is primarily driven by two forces: graphics programmers who want to use more sophisticated texturing than is available with current standards, and the fact that graphics boards are now coming out with multitexturing capabilities.

In general, using multitextures is almost exactly like using several individual textures. You need to create individual texture maps from whatever sources you need, and you need to specify that you will be using a number of textures and bind the particular texture maps to each and enable them. When you specify your geometry, you will need to define the texture coordinate for each of the individual textures that corresponds to each vertex point. This is not especially difficult, and in the case of the OpenGL multitexturing extension, this is discussed later in the chapter.

![Figure 9.8: showing multitexturing in use](image)
As an example of the kind of result that multitexturing can give you, Figure 9.8 (due to Bryan McNett; permission has not yet been sought) shows a surface texture and a light map separately and the results when the textures are applied together to a polygon.

**Using billboards**

Because we mentioned billboards in this chapter, it seems like a good time to describe how to build them and use them in your programs. The reason for using a billboard is to show the viewer an image of a 3D object with 2D techniques, giving the illusion of 3D actions in the scene.

Billboards are generally used in situations where the eye and billboards are placed absolutely in the scene, that is, are not places relative to other geometric objects. Another way to say this is that the eye and billboards are at the top level of the scene graph. This makes it easier to handle the transformations needed to orient the billboard in the scene.

The key point for a billboard is that it must point in the direction of the view. This is relatively straightforward, because you can get the view direction from the definition of viewing. If your eye point is at \((x_1, y_1, z_1)\) and your view reference point is at \((x_2, y_2, z_2)\), then the direction of the view is given by the vector \(d = \langle x_2-x_1, y_2-y_1, z_2-z_1 \rangle\), suitably normalized. You can then express the direction of \(d\) in polar coordinates as \((1, \theta, \phi)\), using the techniques described in the chapter on mathematics for modeling, and rotate the billboard by an angle of \(\theta\) around the vertical (if you want the vertical component of the billboard to remain fixed) or by both angles if you want the entire billboard to face in the view direction.

If you need to have billboards or the eye defined hierarchically in the scene, it is more difficult to orient the billboard to the view because of other transformations in the hierarchy. To accomplish this, we would use the scene graph; there will be a set of transformations that set the position and orientation of the billboard’s rectangle as it was initially placed in the scene. You must take these transformations and invert them, as discussed in the chapter on modeling with scene graphs, giving a transformation that would place them at the root of the scene graph. Then you must add any transformations that are used to orient the eye point. This transformation, then, is the final modeling transformation defined before the billboard is drawn, and will always make the billboard point toward the viewer.

**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.

One of the details you need to understand is which texture-related functions are used to define which aspects of texture mapping. While there are not a large number of these functions, you must use them carefully and in an appropriate sequence in order to have your textures work properly. A list of the primary texture-related functions in OpenGL is given here, and later in the chapter you will see more details of their use. There are some simple considerations for ordering these function calls that we will describe as we discuss examples.

- **glEnable(...)** enable texture mapping as needed in your program; also **glDisable(...)** to disable texture mapping when no longer needed
• **glGenTextures(…)** generates one or more names (integers) that can be used for textures
• **glBindTexture(…)** binds a texture name (generated in `glGenTextures`) to a texture target
  (GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D)
• **glTexParameteri*(…)** defines how antialiasing, wrapping, and similar functions are to be applied to the texture
• **glTexImage*(…)** binds values to most of the parameters that are used to define a texture, such as the number of color coordinates and the internal format of the texture data, how the texture is to be interpreted, the size of the texture map, and the like.
• **glTexCoord*(…)** associates a texture coordinate to a vertex of a graphic object
• **glTexImage*(…)** controls the automatic generation of vertex coordinates for an object
• **glDeleteTextures(…)** deletes one or more textures that had been generated by `glGenTextures`

There are four main effects of these functions. The first three functions (`glEnable`, `glGenTextures`, `glBindTextures`) set the OpenGL system so that it is prepared to use texture mapping. The next two functions (`glTexEnv`, `glTexParameter`) define how the texture is to be applied to a graphics object as it is rendered by the system. The next function, `glTexImage`, identifies the data array that is to be used by the texture and how it is to be interpreted as the texture is loaded into texture memory. The next function, `glTexCoord`, allows you to identify the texture coordinates that are to be associated with the geometric coordinates for your modeling. The order in which these functions are usually applied is given later in examples of texture mapping.

### Associating vertices and texture points

You define your geometry in OpenGL by using the basic primitives as we described earlier in these notes. Within a `glBegin(…) ... glEnd()` pair, you are used to including both `glVertex*()` and `glNormal*()` functions; you can also include `glTexCoord*()` functions to define the texture coordinate for each vertex. As always, these functions come in several varieties, depending on what kind of coordinates you use, including

```c
    glTexCoord1f(float)
    glTexCoord2f(float, float)
    glTexCoord3f(float, float, float)
    glTexCoord1fv(float[1])
    glTexCoord2fv(float[2])
    glTexCoord3fv(float[3])
```

When you specify the texture coordinates, you must do so before you give the `glVertex()` function that defines the point because the state of vertex is set at that point.

The actual texture coordinates represent the real number that represents the position of the texture point within the texture map, so coordinates in the range [0, 1] give you points within the texture space. Coordinates outside this range are interpreted according to your choice of texture wrap or clamp, as described later.

### 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)` function determines the color buffer from
which subsequent buffer reads are to be done, usually the front buffer if you are using single buffering or the back buffer if you are using double buffering. The `glReadPixels(...)` function, used with the RGB or RGBA symbolic format, can then copy the values of the elements in that buffer into a target array. This function can do much more, however; it can save the values of any one color channel, of the depth buffer, or of the luminance, among others. As such it gives you the ability to retrieve a number of different kinds of information from an image. We will not go into more detail here but refer the student to the manuals for the use of these functions.

The array returned by the `glReadPixels(...)` function may be written to a file for later use, or may be used immediately in the program as the texture array. If it is saved to a file, it will probably be most useful if it is saved in a very raw format, holding nothing but the values read from the buffer, but you may want to add extra information to allow it to be used more readily. For example, if you start the file with the width and height of the image, your file will resemble the .ppm format that can be used by many image manipulation programs. If you capture a stream of images into files with names that include sequential numbers, it may be possible to write scripts that will pick these up and make them into a digital movie to display your images as an animation. We refer you to the chapter on animation for more details.

**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 $C_fC_t$, replacing the color by the product of the colors.
- **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 $(1-A_f)C_f+A_fC_t$, and the alpha channel in the pixel is $A_f$.
- **GL_MODULATE**: the color of the pixel is $C_fC_t$ 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 \( C_f \times L_t \), 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 \( C_f \times I_t \), and the alpha channel in the pixel is \( A_f \times I_t \).
- **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, as shown in Figure 9.9, and as shown, these can be applied separately to the horizontal and vertical texture behavior. 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:

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

Figure 9.9 a quad with a texture that is wrapped in one direction and clamped in the other

If you will be using repeating textures you will effectively be tiling your polygons, and it worth thinking a moment about what makes a good tiling texture. A tiling figure needs to have the same colors and overall textures at both the left and right hand side of the figure as well as at the top and bottom side. This is typically not easy to find, so there are techniques to make good tiling figures. One is to use a tool such as Photoshop and rotate the figure so that the former edges of the figure
are in the middle of the new image and are adjacent to each other. Using the Photoshop tools, the middle of the figure is blurred or manipulated so that the line formerly in the middle is not visible. When the picture is then rotated back so that the edges are back where they started, its left and right sides will tile correctly. A similar operation applied to the top and bottom figures completes the process of making the tile figure.

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 `glTexParameteri*(...)` 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:

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

![Figure 9.10: the penguin head texture with the GL_NEAREST (left) and GL_LINEAR (right) magnification filters](image)

The symbolic values for these filters are `GL_NEAREST` or `GL_LINEAR`. This difference between the two is illustrated in Figure 9.10 in an extreme close-up of the penguin image, and it is easy to see that choosing `GL_NEAREST` for the magnification filter gives a much coarser image than does the `GL_LINEAR` filter. 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. Your choice will depend on the relative importance of speed and image quality in your work.
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 their 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 `glTexImage*D(...)` function has one of the more complex parameter lists. These parameters 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 $2^N + 2^*(\text{border})$ for some integer $N$, 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 **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.
So the complete function call is

\[
\text{glTexImage*D}(\text{target}, \text{level}, \text{internal format}, \text{dimensions}, \\
\quad \text{border, format, type, pixels})
\]

An example of this complete function call can be found below for the 2D texture on the surface of a cube.

Note that the \text{glTexImage*D}(\ldots) function simply defines how the texture array is stored and what it is taken to mean. It does not say anything about the source of the image; if you want to use a compressed image format to store your image outside the file, it would have to be translated in order to put the content into the \text{pixels} array.

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 \text{glTexture*(\ldots)} 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 \text{glTexGen*(\ldots)} function, which allows you to specify the details of the texture generation operation.

The \text{glTexGen*(\ldots)} function takes three parameters. The first is the texture coordinate being defined, which is one of \text{GL_S}, \text{GL_T}, \text{GL_R}, or \text{GL_Q} with \text{S}, \text{T}, \text{R}, and \text{Q} being the first, second, third, and homogeneous coordinates of the texture. The second parameter is one of three symbolic constants: \text{GL_TEXTURE_GEN_MODE}, \text{GL_OBJECT_PLANE}, or \text{GL_EYE_PLANE}. If the second parameter is \text{GL_TEXTURE_GEN_MODE}, the third parameter is a single symbolic constant with value \text{GL_OBJECT_LINEAR}, \text{GL_EYE_LINEAR}, or \text{GL_SPHERE_MAP}. If the second parameter is \text{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 \text{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 \text{GL_TEXTURE_GEN_MODE} and the third parameter is \text{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 \text{GL_OBJECT_LINEAR} mode and the \text{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 interpolation

As we noted earlier in the chapter, the scanline interpolation done to render a polynomial needs to take perspective into account to get the highest possible texture quality if the image projection is perspective. In the rasterization step of the rendering pipeline, both the endpoints of each scanline and the internal pixels of the polygon itself are filled by an interpolation process based on the...
vertex coordinates. For each scan line of the frame buffer that meets the polygon, the endpoints of
the scan are interpolated from appropriate vertices and each pixel between them has its color,
depth, and optionally texture coordinates computed to create what is called a fragment. These
interpolated points are affected by the quality you specify with the OpenGL hint function
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, hint).
Here the hint may be GL_DONT_CARE (take the system default), GL_NICEST (perform the
perspective correction to get the best image), or GL_FASTEST (don’t perform the perspective
correction to maximize speed). These fragments are then passed to the per-fragment operations

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
   gluXXX(myQuadric, ...);
   glPopMatrix();

Multitextures

Multitexturing is an optional part of OpenGL 1.2 but not of earlier versions of the OpenGL API.
The ARB_multitexture extension, approved by the OpenGL Architecture Review Board, is a
relatively common OpenGL extension for earlier versions of the API. However, the extension is
currently deprecated in favor of version 1.2 when it includes multitexturing.

Multitexturing operates by defining multiple texture objects with the OpenGL function
   glGenTextures(N, texNames). There is no guaranteed minimum number of textures that
are supported, but you may inquire that number of your system. For each of the texture objects,
you define the properties of the texture through the functions glTexImage*() and
   glTexParameteri() in the same way you would for a single texture. You then define texture
units for each of your textures with the glBindTexture() and glTexEnvi() functions,
giving you a set of textures that will be applied in the order of their name indices. When an object
is rendered with these textures, the first texture will be applied first, the second texture to the object
that is the result of the first texture mapping, and so on.
In actually applying the textures to an object, you must assign the texture coordinates for each of your textures to the each of the vertices of the object. This is illustrated in the multitexturing code example later in this chapter.

**Some examples**

We saw earlier that textures can be applied in several different ways with the function

```c
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 and a point far from the eye to the blue end of the ramp, you get a result like that shown in Figure 9.11. 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.

![Figure 9.11: a Chromadepth-colored image of a mathematical surface](image)

This image was created by using a white object with standard lighting, and adding the texture with the `GL_MODULATE` texture environment. This preserves the shape information presented by the
lighting and adds the texture information. This technique can be used whenever you want to have both shape and texture information in an image.

The code for this is presented in the 1D color ramp example below. We define a color ramp in much the same way we did when creating a pseudocolor ramp in the discussion of visual communication. We associate that ramp with a 1D texture through the glTexImage1D() function and then set up the other texture environment and parameters needed for a 1D texture. Finally we use the glTexGen*() functions to generate an eye-linear automatic texture that is applied to the surface as it is generated. See the example for more details.

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 9.12, the penguin image was used as the texture map on one face of a cube. This texture could also have been created by saving the frame buffer into a file in the program that created the texture map. 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.

Figure 9.12: a 3D cube with the penguin texture map on one face

Sample code for this is in three parts. In the first, we have the data declarations that establish the internal texture map (texImage) and the set of texture names that can be used for textures (texName), and in the init() function we have the glEnable() that allows the use of 2D textures. In the second, we read a file into a texture array, while in the second we set up the OpenGL functions that define how the texture map is to be applied and in the third we draw the face of the cube with the texture map applied.

```c
#define TEX_WIDTH 512
#define TEX_HEIGHT 512
static GLubyte texImage[TEX_WIDTH][TEX_HEIGHT][3];
static GLuint texName[1]; // parameter is no. of textures used

void init() {
  ...
  glEnable(GL_TEXTURE_2D); // allow 2D texture maps
  ...
}
```
void setTexture(void) // read file into RGB8 format array
{
    FILE * fd;
    GLubyte ch;
    int i, j, k;

    fd = fopen("penguin.512.512.rgb", "r");
    for (i=0; i<TEX_WIDTH; i++) {
        for (j=0; j<TEX_HEIGHT; j++) {
            for (k=0; k<3; k++) {
                fread(&ch, 1, 1, fd);
                texImage[i][j][k] = (GLubyte) ch;
            }
        }
    }
    fclose(fd);
}

// enable textures for the last face
glEnable(GL_TEXTURE_2D);
GenTextures(1, texName); // define texture for sixth face
BindTexture(GL_TEXTURE_2D, texName[0]);
TexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
TexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, TEX_WIDTH, TEX_HEIGHT,
           0, GL_RGB, GL_UNSIGNED_BYTE, texImage);

GlBegin(GL_QUADS); // sixth quad: negative X face
    GlNormal3fv(normals[1]); // single normal; flat shading
    TexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
    TexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
    TexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
    TexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
GlEnd();
GlDeleteTextures(1, texName);

In this example we saw a typical ordering of the texture functions in OpenGL. First an array is defined and is loaded with texture data, either by reading a file or by creating a synthetic texture. Then the sequence
    enable the texture
    generate a texture for a texture name
    bind the texture to a texture type (here GL_TEXTURE_2D)
    set the texture environment
    define the texture parameters
    create the texture image
sets up the OpenGL environment for texturing. The order is important for some of these functions but not for all; setting the texture environment and defining the texture parameters can be done in any order. But it is probably easiest to find a sequence that works for you, and to use that sequence consistently in your work.
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 9.13, we see an example of this as a photograph of Hong Kong that has been modified in Photoshop with a spherical 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 9.13: the original texture for an environment map (left) and the map on a surface (right)](image)

Many parts of this example are handled just as any other 2D texture would be, but the texture is automatically generated by the `glTexGeni()` function and the texture coordinates for the surface are generated by using the normal vectors at each vertex.

As we saw with the ChromaDepth™ example earlier, this example uses lighting on a white surface and adds the texture in GL_MODULATE mode to preserve the shape information from lighting and the texture information from the environment map.

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**

**A 1D color ramp:** Sample code to use texture mapping in the Chromadepth™ example is shown below. The declaration set up the color ramp, define the integer texture name, and create the array of texture parameters.

```c
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.

```c
makeRamp();
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexParameteri(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(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);
```

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.

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

An environment map: 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 computational cost, of course. But the images in Figure 9.13 suggest that it might well be worth the cost.

    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 );

Using multitextures

When we introduced the way OpenGL defines multitextures, we hinted at the kinds of changes you would need to make to use multitextures. Here we will give that code in some detail for a case using two textures so you may see what it would look like.

The declaration of the textures[] array would be:

    int textures[2];

In an initialization function we might find the following definitions of the texture objects:

    // load and bind the textures
    glGenTextures(2, &textures);

    // load the first texture data into a temporary array
    file.open("tex0.raw");
    file.read(textureData, 256*256*3);
    file.close();
// build the first texture
glBindTexture(GL_TEXTURE_2D, texture[0]);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
             GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256, 256, GL_RGB,
                   GL_UNSIGNED_BYTE, textureData);

// load the second texture data into a temporary array
file.open("tex1.raw");
file.read(textureData, 256*256*3);
file.close();

// build the second texture
glBindTexture(GL_TEXTURE_2D, texture[1]);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
gTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
             GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256, 256, GL_RGB,
                   GL_UNSIGNED_BYTE, textureData);

In display(), we might find the following definitions of the texture units:
// set the texture to the first one then bind the texture
gActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[0]);
gTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// set the texture to the second one then bind the texture
gActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[1]);
gTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

And in display() or another function that actually implements the geometry of your model, you might find the following code that associates both sets of texture coordinates to each vertex:

 glBegin(GL_TRIANGLE_STRIP);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 0.0);
  glVertex3f(-5.0, -5.0, 0.0);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 1.0);
  glVertex3f(-5.0, 5.0, 0.0);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 0.0);
  glVertex3f(5.0, -5.0, 0.0);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 1.0);
  glVertex3f(5.0, 5.0, 0.0);
 glEnd();
Summary

We have seen that texture mapping is a straightforward concept that allows you to add a great deal of extra information to an image in several different ways. You should understand how to create textures from photographic and synthetic sources, and how to associate texture coordinates to geometric coordinates as you model a scene. We have discussed how texture mapping is done with the OpenGL graphics API, so you should now be able to write graphics programs that use texture maps and to create texture maps from both natural and synthetic sources. We have not gone into any detail on how texture mapping is carried out in creating an image; this is covered in the later chapter on the rendering pipeline. Most of this work is managed by simple linear interpolation, but you should see the value of perspective adjustment for textures.

Questions

1. The sequence of OpenGL function calls needed for texture mapping has some functions that need to come first, and then other functions can come later and in arbitrary order. Discuss why this sequence is required.

2. What work is needed to use image files in formats such as GIF or JPEG as sources for texture maps? Does your graphics API have any texture loaders that will allow you to use any of these file formats?

3. Think about how you would achieve certain effects if you had multitexturing. Begin with some simple effects, such as distressed wood, bullet holes in metal, or water spots on a surface, and then think up some new effects and decide how you would create them. If you have multitexturing capability with your graphics API, create the individual textures needed and then implement your ideas.

Exercises

4. Make a number of texture maps from easily-available sources. Use as many of the following techniques as possible: (a) digital photographs, (b) scanned photographs, (c) screen captures, and (d) capturing the contents of the frame buffer from an OpenGL program. In each case, be sure the texture map is saved in a format that can be read into a texture array for your programs.

5. Implement a set of labels by writing them on separate lines in one image using the text functions in an application such as Photoshop. Identify the points in the image that separate the individual words or phrases, and write a small graphics program that maps these labels to different parts of an image.

6. Consider a triangle in 2D space with coordinates \( V_0 = ( , ) \), \( V_1 = ( , ) \), and \( V_2 = ( , ) \) counterclockwise, and take the point \( P = ( , ) = \alpha V_0 + \beta V_1 + \delta V_2 \) for some different values of the coefficients \( \alpha \), \( \beta \), and \( \delta \) with \( \alpha + \beta + \delta = 1 \). If the texture coordinates for the triangle’s vertices are \( T_0 = ( , ) \), \( T_1 = ( , ) \), and \( T_2 = ( , ) \) for \( V_0 \), \( V_1 \), and \( V_2 \) respectively, calculate the texture coordinates for the point \( P \). For the choices of coefficients for which these coordinates are not integers, discuss how you would calculate the actual color for point \( P \) in terms of colors for nearby points in texture space and relate these to the filtering options for textures in OpenGL.

7. Besides scanning photos, you can paint synthetic textures as a way to create textures for your images. In this exercise we will draw on one of the world’s great visual cultures, the Ndebele...
of South Africa. Below are four small photographs of Ndebele homes with their extraordinary patterns, and your task is to use an image creation program or to write appropriate procedures to create patterns similar to those in these Ndebele images, and then save your work in a format that is appropriate to use in texture maps. (These figures are available as JPEG images in the resource materials for this chapter.)

8. As an example of a procedural synthetic texture, create a “straw thatch” texture by drawing many parallel lines within a rectangle, each having a given start point and end point, and each having a color that is within the general range of brown to tan that one would find in straw. This texture can also be used to simulate wood grain, though it does not have the more regular behavior usually found there. An example of this texture is shown with this project. You can choose to generate the line segments with known coordinates, or you can make the overall process random. Of course, you need to draw enough of these lines to cover the space, which cannot be guaranteed with a modest number of line segments, so be sure to specify a generous number.
9. Create a different synthetic texture from the one shown in Figure 9.2. For example, you could choose a set of random points in 2D integer space and a sequence of random colors, and for each point in texture space, you could give that point a color that depends on the point to which it is nearest. Or you could use some sort of 2D function pseudocoloring as was discussed in the chapter on science applications. Then use this synthetic texture as we did the checkerboard texture and see what visual results you get.

Experiments

10. Consider the various ways texture can be applied to a surface: GL_BLEND, GL_DECAL, GL_MODULATE, and GL_REPLACE. Create a scene with a texture map, and try these different kinds of texture environments; record the results.

11. Using the 1D texture map concept introduced for the ChromaDepth™ process, define 1D texture maps that could be used to show the elevation on a height field, as introduced in the science examples chapter; extend this to include contour mapping.

12. Use a generally-available program such as POVRay and experiment with the texture options it provides. The goal is to get a sense of how these textures look, especially textures built on the noise function.

13. Make a texture map suitable for use as a billboard by taking a natural image and editing it so that the background is in a unique color, different from any color in the part of the image you want to keep. Save the edited image as a raw RGB image file. Then modify the function in the chapter that reads in a raw RGB image file so that an alpha channel is created as described in the chapter, and use this image with an RGBA texture to see how the alpha blending will work.

14. Pick a texture map that allows you to see the fine details of the texture, such as a checkerboard texture with relatively small squares, and map it onto the GLU quadric objects. Look for points where the texture map on the surface behaves unusually, and see if you can identify those points with any particular geometry on the surface.

15. Take some of the example textures you created in the first exercise above and apply them to the other faces of the cube in Figure 9.13 whose code was given above.

16. Use an artificial texture such as a black-and-white checkerboard pattern to experiment with the use of texture mapping on intensity, luminance, or blending. The results should show the checkerboard pattern, but it should be visible only in the effects on the polygon color instead of by the checkerboard itself being visible.

17. Take an image and create a second image by doing a “fisheye” conversion of the first using a tool such as Photoshop. Apply both images to some smooth geometry using environment map techniques and discuss the results.

Projects

18. (The small house) Take the texture maps you created from the Ndebele houses above and apply them to the exterior and interior walls of the small house you designed earlier, and see what you get when you walk through the resulting house. Do not be surprised if your have a much slower refresh rate for the walkthrough than you did earlier, because the computer is working much harder to rebuild all the texture maps as you re-use them.
19. (A scene graph parser) Add texture mapping to your scene graph parser as described in this chapter, and have the parser write appropriate OpenGL code to include this feature in the `display()` function it writes.