Chapter 18

Textures

This chapter describes the texture capabilities of the IRIS GL. Texture is not available on every system. If you are not using a system that supports texture, you may want to skip to Chapter 19, “Using the GL in a Networked Environment.”

Some systems perform texturing in software, and others have special hardware for texturing. Systems that use software texturing do not exhibit the same level of texture performance as systems that use hardware texturing, so plan carefully when using texture on systems that use software texturing.

- Section 18.1, “Texture Basics,” begins by introducing some terminology that you need to know to understand textures.
- Section 18.2, “Defining a Texture,” tells you how to define a texture and how to optimize image quality.
- Section 18.3, “Using Texture Filters,” describes how to use filters to modify how textures are treated by the system.
- Section 18.4, “Using the Sharpen and DetailTexture Features,” describes two advanced texture features that are available on RealityEngine systems.
- Section 18.5, “Texture Coordinates,” tells you how to assign texture coordinates to define a mapping into object space.
- Section 18.6, “Texture Environments,” tells you how to set the texture environment to modify the color and opacity of textured polygon pixels.
- Section 18.7, “Texture Programming Hints,” presents strategies for achieving the maximum texture mapping performance from the GL.
- Section 18.8, “Sample Texture Programs,” contains two sample programs that illustrate two different ways of using textures.

18.1 Texture Basics

Texture adds realism to an image. You can use texture in a variety of ways to enhance visual information. Use texture to:

- show the material of an object. For example, wrap a wood grain pattern around a rectangular solid to create a block of wood.
- create patterned surfaces such as brick walls and fabrics by repeating textures across a surface.

  See the *brick.c* sample program in Section 18.8, "Sample Texture Programs," for an example of how to create a brick texture.
- simulate physical properties for scientific visualization applications. For example, temperature data represented by color can be mapped onto an object to show thermal gradients.

  See the *heat.c* sample program in Section 18.8 for an example of how to use texture to show a thermal gradient.
- simulate lighting effects such as reflections for photo-realistic images.

Figure 18-1 shows an example of some textures that can be used to represent water, wood planks, and the end of a wood board. The water and wood plank textures are photos. The wood cross-section texture is a synthetically generated collection of light and dark hues.

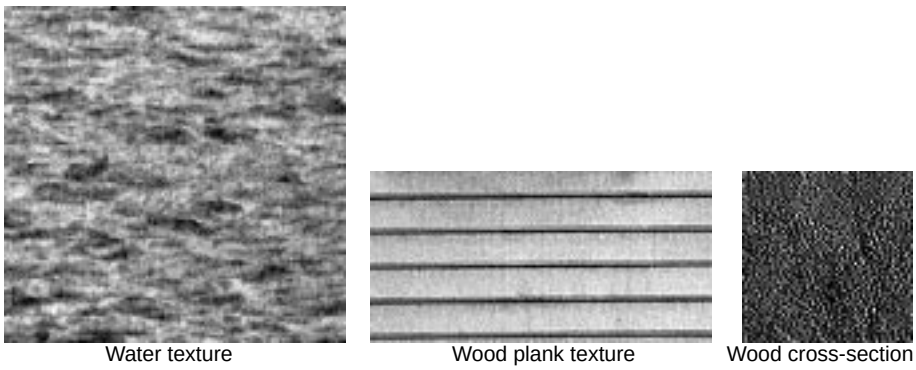

Figure 18-1 Textures for Wood and Water

Figure 18-2 shows how these textures are applied to objects in a 3-D scene.

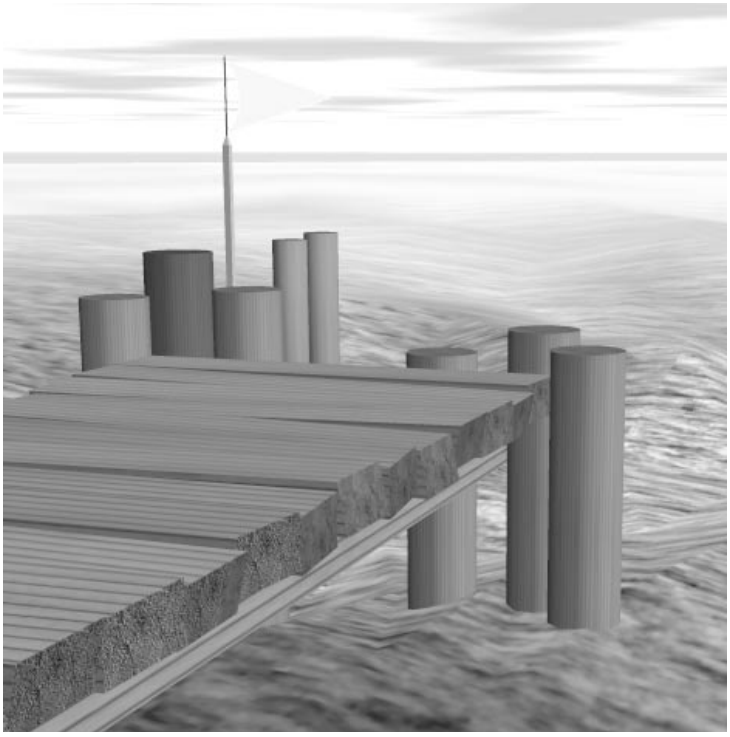

Figure 18-2 Fully Textured Scene

Textures are usually specified in two dimensions for most applications. RealityEngine systems allow three-dimensional textures. Definitions and uses of 2-D and 3-D textures follow.

18.1.1 2-D Textures

Texture mapping is a technique that applies an image to an object's surface as if the image were a decal or cellophane shrink-wrap. The image exists in a coordinate system called the *texture space*. The coordinate axes *S* and *T* define a 2-D texture space. A *texture* is a function that is defined on the interval 0 to 1 along both axes in the texture space. The individual elements of a texture are called *texels*. Texels are indexed with *(s,t)* coordinate pairs.

Figure 18-3 shows how a simple stripe texture is defined in 2-D texture space and mapped to 3-D object space. The mapping describes where the texels are placed in object space. This is not always a one-to-one mapping to screen pixels, as you will see later.

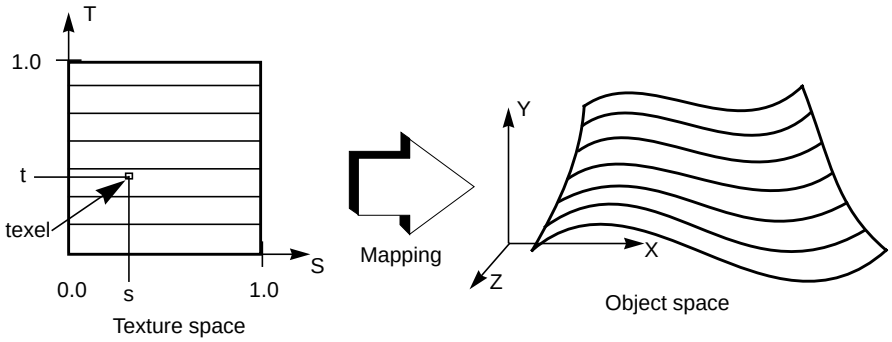

Figure 18-3 Mapping from 2-D Texture Space to 3-D Object Space

18.1.2 3-D Textures

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to section Section 18.1.3, “How to Set Up Texturing,” if you do not have one of these systems.

RealityEngine systems let you specify 3-D textures. Figure 18-4 shows a 3-D texture. Three-dimensional textures can also be thought of as an array of 2-D textures, as illustrated by the diagram on the right of the 3-D texture.

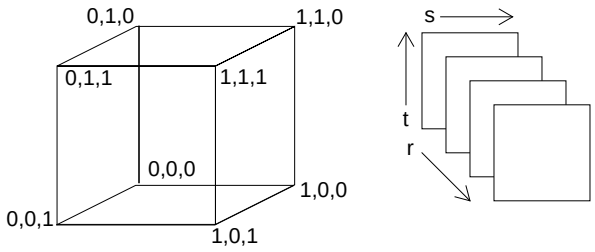

Figure 18-4 3-D Texture

The 3-D texture is mapped into *(s,t,r)* coordinates such that its lower left back corner is (0,0,0) and its upper right front corner is (1,1,1).

3-D textures can be used for

- Volume rendering.
- Examining a 3-D volume one slice at a time.
- Animating textured geometry—for example, people that move.

Texel values defined in a 3-D coordinate system form a texture volume. Textures can be extracted from this volume by intersecting it with a 3-D plane, as shown in Figure 18-5.

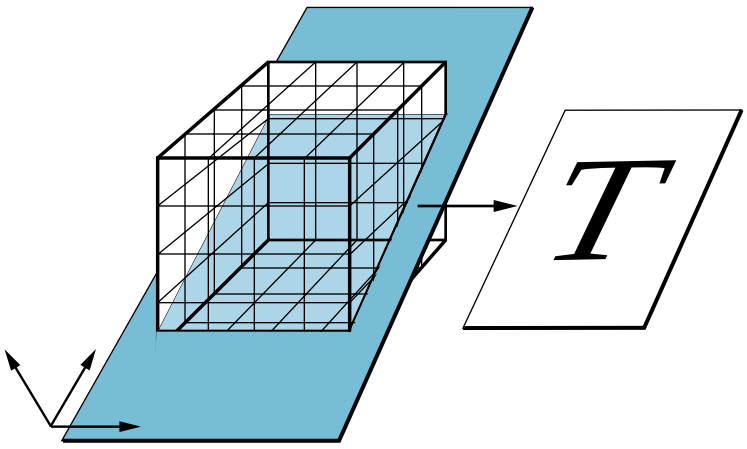

Figure 18-5 Extracting an Arbitrary Planar Texture from a 3-D Texture Volume

The resulting texture, which is applied to a polygon, is the intersection of the volume and the plane. You determine the orientation of the plane by supplying, or by having the GL supply, texture coordinates.

18.1.3 How to Set Up Texturing

This list provides an overview of the steps used to set up texturing.

1. Use `getgdesc(GD_TEXTURE)` to determine whether your system supports texturing.
2. Create a texture by defining a texture image.
3. Specify a set of texture properties that describes the number of components in the texture and how the texture should be filtered.
4. Assign texture coordinates to the vertices of geometric primitives, either explicitly or automatically, to define a mapping from texture space to geometry in object space.
5. Choose a texture environment that specifies how texture values should modify the color and opacity of an incoming shaded pixel. You can use this feature to indicate whether you want the texture to be completely opaque on top of the pixel, let some of the pixel color show through, or mix the pixel color with the texture.

Each of these steps is discussed in detail in the following sections.

18.2 Defining a Texture

A texture function consists of an image defined as an array of texels and a set of parameters that determines how samples are derived from the image. The texture image can be any image that you have constructed, scanned in, or captured from the screen.

Regardless of its dimensions, the texture image is mapped into an *(s,t,*[*r*]) coordinate range such that its lower-left-back corner is (0.,0.,[0.]) and its upper-right-front corner is (1.,1., [1.]).

Note: For a 2-D texture, *r* is always ignored.

18.2.1 Using Texture Components

The elements of the texture array are constructed with one to four components per texel. Table 18-1 lists the texture types and the components for each type.

Texture Type	Components
1-component	Intensity
2-component	Intensity, Alpha
3-component	Red, Green, Blue
4-component	Red, Green, Blue, Alpha

Table 18-1 Texture Components

Intensity is used to show color variations (shades) within the same color value. Alpha is used to indicate the transparency of the color.

Suggestions for choosing a texture type to achieve certain effects follow.

Use a 1-component texture to create subtle variations in surfaces. For example, vary the intensity of a brown shade to turn plain brown hills into hills with shades of light to dark brown. You can also use a 1-component texture with a texture environment to create blended textures, such as a blue-and-white sky. The wood and the water in Figure 18-2 are examples of 1-component textures.

Use a 2-component texture to create surfaces with subtle color variations on geometry that has irregular edges. For example, use a 2-component texture to create a tree with many shades of green. The 2-component texture used for the foliage varies in intensity, creating different shades of green from light to dark. Another 2-component texture is used for the trunk, which has different shades of brown. 1-component and 2-component textures are sufficient for representing many types of objects and are effective because they use less memory than 3- and 4-component textures.

Alpha, the other component of the 2-component texture, is used to indicate how transparent the texture is. Commands that determine how alpha affects the manner in which a texture is drawn include `afunction()`, `blendfunction()`, and on Reality Engine, `msalpha()`.

Note: On RealityEngine, use multisampling with the `msalpha()` feature rather than `afunction()` to define the edges of the tree.

Figure 18-6 shows an example of a tree created with a 2-component texture. The tree is a single rectangular polygon that has a scanned photo of a tree superimposed on it. This polygon can be rotated about the center of the trunk, as shown by the outlines, so that it is always facing the viewer.

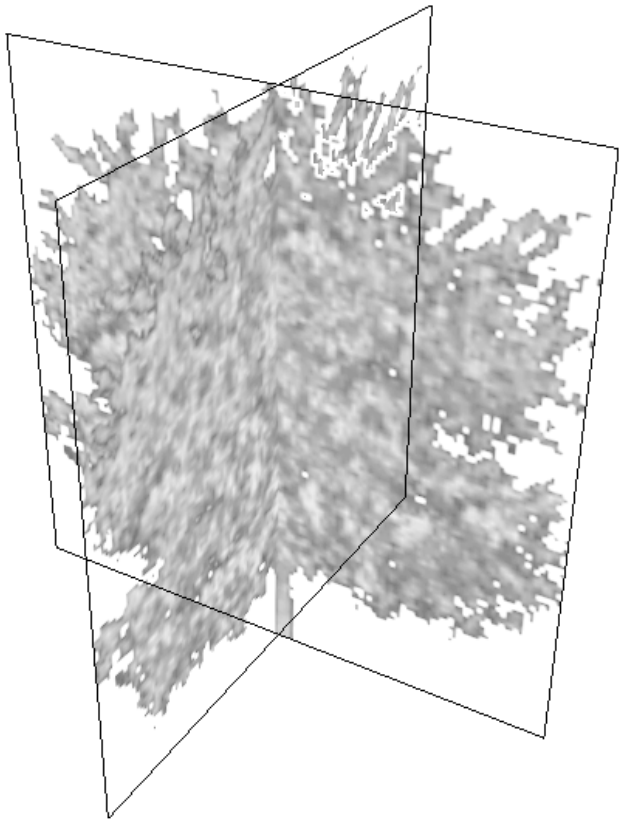

Figure 18-6 Example of a Tree Created with a 2-component Texture

Representing complex surfaces by texturing simple polygons rather than by creating complex geometry with multiple polygonal faces can achieve greater realism and better performance. You can experiment with the performance trade-off between the number of polygons and the use of texturing to get the best possible solution for your application.

18.2.2 Loading a Texture Array

Textures are loaded into a memory array that the system accesses when rendering the textured surface. Figure 18-7 shows how the texture array is constructed for an 8-bit–per-component image.

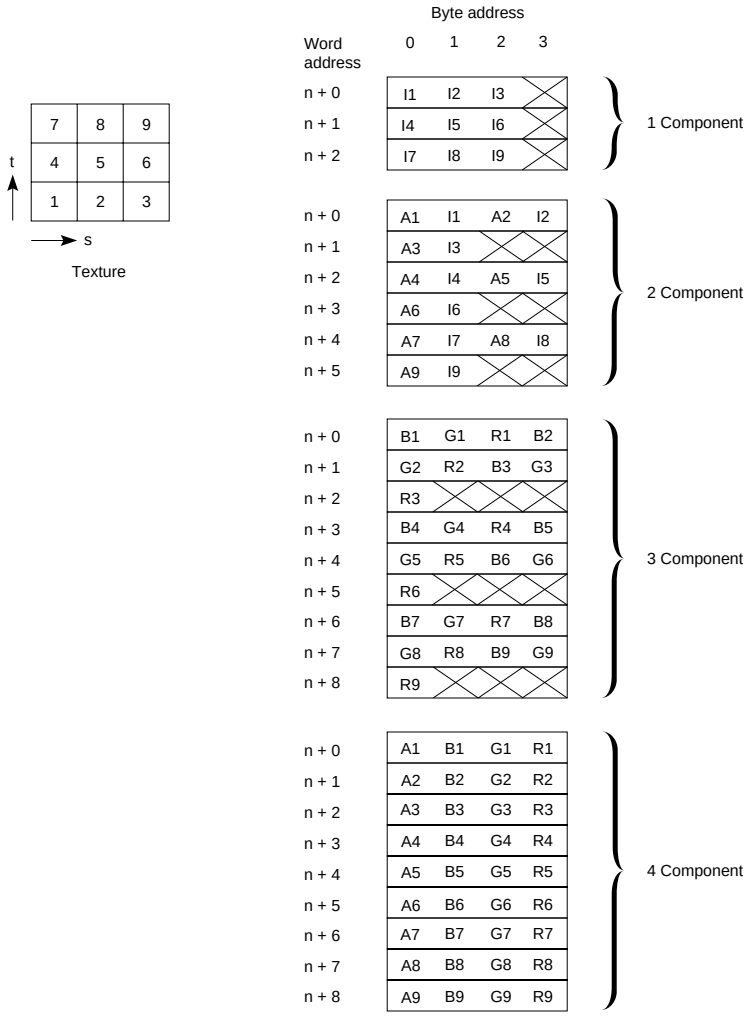

Figure 18-7 Structure of a Texture Array

Figure 18-7 shows a texture consisting of 9 texels, which are numbered 1 through 9. The texels fill the texture from left to right, bottom to top. The component information for each texel is stored as a packed array of unsigned long words. This is the same format used by `lrectread()`.

In Figure 18-7, the boxes represent blocks of memory. A long word is 32 bits, and each byte of texture information requires 8 bits. Therefore, 4 bytes of texel information can fit into each long word. Each row of texel information must be long word-aligned, so the end of the row must be byte-padded to the end of each long word. The diagram shows how the array is packed for an 8-bit–per-component texture of 1-, 2-, 3-, or 4- components, consisting of 9 texels.

Table 18-2 summarizes the relationships between texel component and byte ordering.

Components	Pixel Type	Byte Ordering (low-order to high-order)
1-Component	Intensity	I0, I1, I2, I3, I4,...
2-Component	Intensity-Alpha	A0, I0, A1, I1, A2,...
3-Component	Red, Green, Blue	B0, G0, R0, B1, G1,...
4-Component	Red, Green, Blue, Alpha	A0, B0, G0, R0, A1,...

Table 18-2 Texture Image Array Format

For each polygon pixel to be textured, the texture function generates texture components (color, intensity, alpha) based on the texel type, the texture map coordinates of the pixel's center, and the area in texels onto which the pixel maps. The properties that you specify for the texture function determine how the texture image is sampled and how the texture function is evaluated outside the range (0.,0.,[0.]), (1.,1.,[1.]).

Note: All geometry including polygons, lines, points, and character strings are texture-mapped. Character strings always have the texture coordinates (0.,0.,[0.]).

18.2.3 Defining and Binding a Texture

Textures use the define/bind paradigm that was introduced in Chapter 9.

Use `texdef2d()` or `texdef3d()` to *define* a texture and `texbind()` to *activate* a texture.

Textures can be redefined by calling `texdef2d()` or `texdef3d()` with the index of a previously defined texture. As with materials, only one texture can be active, or bound, at a time. The binding process and defining process are separated for performance reasons—it takes substantially less time to bind a texture than it takes to define one.

The ANSI C specifications for `texdef2d()` and `texdef3d()` are:

```
void texdef2d(long index, long nc, long width, long height,
              unsigned long *image, long np, float props)

void texdef3d(long index, long nc, long width, long height,
       long depth, unsigned long *image, long np, float props)
```

where:

index is a unique index, or name, that identifies the texture. Index 0 is reserved as a null definition, and it cannot be redefined.

nc is the number of components per texel (1, 2, 3, or 4).

width is the width of the texture image in texels.

height is the height of the texture image in texels.

depth is the depth of the texture image in texels.

image is a word-aligned array containing the texel data.

np is the number of symbols and floating point values in the *props* array, including the termination symbol `TX_NULL`. If *np* is zero, it is ignored, but operations over network connections are more efficient when *np* is correctly specified.

props is an array of floating point symbols and values that define how to interpret the texture function. The *props* array contains a sequence of symbols, each followed by the appropriate number of floating point values. The last symbol in the array must be `TX_NULL`, which terminates the array.

The following code fragment illustrates how to use `texdef2d()` to define a 2-D brick texture:

```
float texprops[] = {TX_MINFILTER, TX_POINT,
                    TX_MAGFILTER, TX_POINT,
                    TX_WRAP,TX_REPEAT, TX_NULL};

texdef2d(1, 1, 8,  8, bricks, 7, texprops);
```

The current texture, bricks in this case, is bound using the `texbind()` call:

```
texbind(TX_TEXTURE_0, 1);
```

In this example, the array *texprops* explicitly specifies `TX_MINFILTER`, the filter function that is used for minifying texture when the pixel being textured maps onto an area greater than one texel, and `TX_MAGFILTER`, the filter function that is used when the pixel being textured maps to an area less than or equal to one texel. See Section 18.3, "Using Texture Filters," for a discussion of minification and magnification filters. In the brick example, `TX_MINFILTER` and `TX_MAGFILTER` are both set to use point-sampling filters.

`TX_WRAP`, which specifies what to do when the (*s*,*t*,[r]) coordinates are outside the range 0.0 through 1.0, is set to `TX_REPEAT`. `TX_REPEAT` specifies that only the fractional parts of the texture coordinates are used, thereby creating a repeating pattern. `TX_REPEAT` is the default. By setting `TX_WRAP` to `TX_REPEAT`, the small 8×8 pattern is repeated across the polygon, creating an entire wall of bricks.

You can specify the wrapping behavior per coordinate, rather than globally:

`TX_WRAP_S` specifies the wrapping behavior only for the *s* texture coordinate.

`TX_WRAP_T` specifies the wrapping behavior only for the *t* texture coordinate.

`TX_WRAP_R` specifies the wrapping behavior only for the *r* texture coordinate.

If you replace `TX_REPEAT` with `TX_CLAMP`, you see the brick pattern only once on the polygon, where the *(s, t)* coordinates are in the range (0.,1.). The edges of the texture are smeared across the rest of the polygon. `TX_CLAMP` is useful for preventing wrapping artifacts when mapping a single image onto an object.

`TX_TILE`, a property that is not used in the brick example, supports mapping of high-resolution images with multiple rendering passes. By splitting the texture into multiple pieces, each piece can be rendered at the maximum supported texture resolution. For example, to render a scene with 2× texture

resolution, `texdef2d()` is called four times. Each call includes the entire image, but specifies a different subregion of that image to be converted into a texture.

`TX_TILE` is followed by four floating point coordinates that specify the *x* and *y* coordinates of the lower-left corner of the subregion, then the *x* and *y* coordinates of the upper-right corner of the subregion. The original texture image continues to be addressed in the range 0,0 through 1,1. However, the subregion occupies only a fraction of this space, and pixels that map outside the subregion are not drawn.

To divide the image both horizontally and vertically into quadrants, the corners of the subregions should be (0,0 .5,.5), (.5,0 1,.5), (0,.5 .5,1), and (.5,.5 1,1). The scene is then drawn four times, each time calling `texbind()` with the texture ID of one of the four quadrants. In each pass, only the pixels whose texture coordinates map within that quadrant are drawn.

If the image, or the specified subregion of the image, is larger than what can be handled by the hardware, it is reduced to the maximum supported size automatically, with no indication other than the resulting visual quality. Because subregions are specified independently, they should all be the same size. Otherwise, some subregions may be reduced while others are not.

18.2.4 Selecting the Texel SIze

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.3, “Using Texture Filters,” if you do not have one of these systems.

RealityEngine supports three internal texel sizes: 16-bit, 32-bit, and 64-bit. You can change this internal format to select the texel size that best suits your application needs. There is a trade-off between image quality and speed. The fill rate is inversely proportional to the texel size; thus, the fill rate doubles when the texel size is halved.

The default texel size for 1- and 2-component textures is 16 bits. The default texel size for 3- and 4-component textures is 32 bits.

Each of the texel sizes is available with 12, 8, or 4 bits per component. Table 18-3 shows the configurations possible, and the symbols for selecting those configurations for the different texel sizes.

Texel Size	1-component	2-component	3-component	4-component
16-bit	TX_I12_A4	TX_I12_A4, TX_IA_8	TX_RGB_5	TX_RGBA_4
32-bit		TX_IA_12	TX_RGBA_8	TX_RGBA_8
64-bit			TX_RGB_12	TX_RGBA_12

Table 18-3 Texture Component Configuration for Different Texel Sizes

Use 16-bit texels for the fastest performance and to reduce memory usage. 16-bit texels with TX_RGBA_4 provide high performance and good image quality, but if you don't need alpha, use TX_RGB_5 for even better image quality, because it increases the color resolution.

Use the 64-bit texel size for the highest resolution for color computations, for example, in low-light-level simulations. This format provides 12-bit per component capability for R,G,B,A texture maps. The advantage of 12 bits per component is that it increases the number of color levels for each component from 256 to 4096, greatly enhancing the precision of the color computation.

Use the 32-bit texel size when you want to balance performance halfway between speed and image quality.

The texel size and bit configuration of the texture components are set as internal and external format hints in the *props* array of the `texdef2d()` and `texdef3d()` commands.

Use TX_INTERNAL_FORMAT in the *props* array as a hint to trade image quality for speed. This hint affects the precision used internally in texture function computations. Because the performance of texture function implementations is typically constrained by texel accesses per screen pixel, you can specify a smaller internal texel size and often realize performance gain.

The tokens for TX_INTERNAL_FORMAT are:

TX_I_12A_4 specifies that a 1- or 2-component texture should be computed with at least 12 bits for intensity and 4 bits for alpha. Texel size: 16 bits.

TX_IA_8 specifies that a 2-component texture should be computed with at least 8 bits for intensity and 8 bits for alpha. Texel size: 16 bits.

TX_RGB_5 specifies that a 3-component texture should be computed with at least 5 bits for red and blue and at least 6 bits for green. Texel size: 16 bits.

TX_RGBA_4 specifies that a 4-component texture should be computed with at least 4 bits per component. texel size: 16 bits.

TX_IA_12 specifies that a 2-component texture should be computed with at least 12 bits per component. Texel size: 24 bits; may be rounded up to 32 bits.

TX_RGBA_8 specifies that a 3- or 4-component texture should be computed with at least 8 bits per component.Texel size: 32 bits.

TX_RGBA_12 specifies that a 4-component texture should be computed with at least 12 bits per component.Texel size: 64 bits.

TX_RGB_12 specifies that a 3-component texture should be computed with at least 12 bits per component. Texel size: 48 bits, rounded to 64 bits.

TX_EXTERNAL_FORMAT specifies the size of the image components:

TX_PACK_8 specifies that the image is composed of 8-bit components. This is the default.

TX_PACK_16 specifies that the image is composed of 16-bit components.

When the external format is larger than the internal format, the most significant bits of the external format pixel are used. When the external format is smaller than the internal format, the most significant bits of the external format pixel are replicated in the lower order bits of the internal format. Thus, three 8-bit external format components with the hexadecimal values AB,FF,00 become the three 12-bit internal format components with the hexadecimal values ABA,FFF,000.

The next section describes the filters that can be specified in the *props* array.

18.3 Using Texture Filters

During the texture mapping process, the texture function computes texture values based on the *(s,t,* [*r*]) texture coordinates at the center of the polygon pixel that is being textured and the area in texture space onto which the pixel maps. One of two filtering algorithms is used, depending on the size of this area.

If the area is greater than the area of 1 texel, as shown in Figure 18-8, the texture is *minified* to fit the screen pixel and the texture function's minification filter is used. Specify the minification filter with the `TX_MINFILTER` parameter.

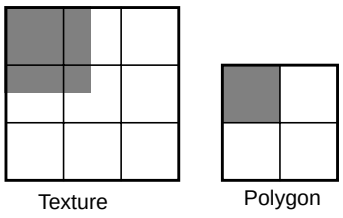

Figure 18-8 Texture Minification

If the area is less than the area of 1 texel, as shown in Figure 18-9, the texture is *magnified* to fill the screen pixel and the texture function's magnification algorithm is used. Specify the magnification filter with the `TX_MAGFILTER` parameter.

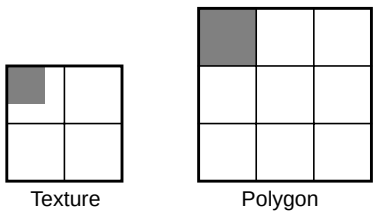

Figure 18-9 Texture Magnification

Minification and magnification filters are discussed in detail in the sections that follow.

18.3.1 Minification Filters

Minification filters are used when multiple texels correspond to a single screen pixel, as shown in Figure 18-10.

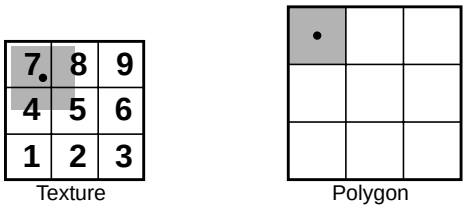

Figure 18-10 Texture Minification

In most cases, the best minification results are obtained by using a MIPmap to minify the texture.

MIPmap Minification Filters

Figure 18-11 shows a MIPmap. MIP comes from a Latin term that means “many things in a small place.” A MIPmap stores an array of prefiltered versions of the texture image.

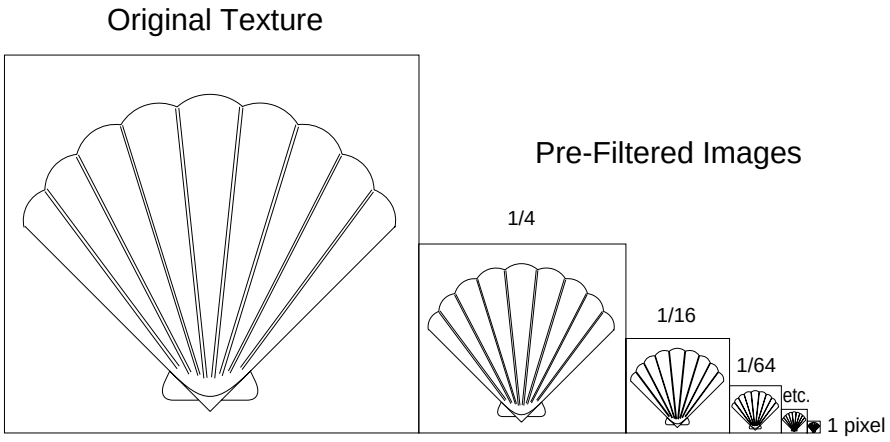

Figure 18-11 MIPmap

Each image in the array has half the resolution of the image before it, but it still maps into the texture coordinate range (0.,0.) to (1.,1.). Thus, the first image in the MIPmap has a 1-to-1 texel-to-pixel correspondence. The second image has a 4-to-1 correspondence, the third image, 16-to-1, and so on.

For any minification factor, there is one image in the MIPmap whose texels map closely to an area in texture space that is less than or equal to the area that the pixel being textured maps into. This image has the appropriate resolution, so samples interpolated from this image do not have undersampling artifacts.

Each of the MIPmap filters works differently. The default minification filter for systems other than RealityEngine is `TX_MIPMAP_LINEAR` or a filter of equal performance, but better quality. Prefiltered versions of the image, when required by the minification filter, are computed automatically by the GL.

RealityEngine uses high-performance trilinear MIPmap filtering by default. Simultaneous parallel memory access allows the eight samples needed for trilinear interpolation to be retrieved with a single memory access.

Trilinear interpolation is one of the highest quality texture functions available. It produces images that look sharp when viewed from close range and that remain stable under all circumstances. In addition, there is no perceptible transition in the image as the textures move relative to the eyepoint.

RealityEngine also performs *quadlinear* MIPmap filtering of 3-D textures. This is effectively a trilinear interpolation of a 3-D texture, automatically generating a series of 3-D volumes, each 1/8 smaller than the one above. The interpolation is performed between the 8 adjacent pixels in the MIPmap from the two closest-bounding volume levels and then blended between the two results, thus achieving a four-way interpolation.

Select the filter to use based on the type of application you are creating and the quality and. performance results you want. Refer to Section 18.7, “Texture Programming Hints,” for additional information on selecting a filter.

Note: Because the high-performance MIPmap filters available on RealityEngine are superior to other MIPmap minification filters, the GL always uses `TX_MIPMAP_TRILINEAR` for MIPmapping 2-D textures and `TX_MIPMAP_QUADLINEAR` for MIPmapping 3-D textures for applications running on a RealityEngine, no matter what filter is specified in the props array.

To select a minification filter, use the token TX_MINFILTER, followed by a single symbol that specifies the minification filter. Values for TX_MINFILTER are listed below, with descriptions of what they do.

Note: Filters marked with an asterisk(*) are currently available only on RealityEngine systems.

TX_MIPMAP_POINT
: chooses a prefiltered version of a 2-D texture, based on the number of texels that correspond to 1 screen pixel. The value of the pixel that is nearest to the (*s*,*t*,*r*) mapping onto that image is used to color the pixel.

TX_MIPMAP_LINEAR
: chooses the two prefiltered versions of a 2-D texture that have the nearest texel-to-screen pixel correspondence. A weighted average of the values of the pixel in each of these images that is nearest to the (*s*,*t*,*r*) mapping onto that image is used to color the pixel.

TX_MIPMAP_BILINEAR
: chooses a prefiltered version of a 2-D texture, based on the number of texels that correspond to 1 screen pixel. The weighted average of the values of the 4 pixels nearest to the (*s*,*t*) mapping onto that image is used to color the pixel.

TX_MIPMAP_TRILINEAR
: chooses the prefiltered version of the 2-D texture whose texel size most closely corresponds to screen pixel size. A weighted average of the values of the pixels nearest to the mapping onto that image is used to color the pixel.

  For 2-D textures, TX_MIPMAP_TRILINEAR chooses the two prefiltered versions of the image that have the nearest texel-to-screen pixel size correspondence. A weighted average of the values of the 4 pixels in each of these images that are nearest to the *(s*,*t*) mapping onto that image is computed. The weighted averages from the two levels are then themselves interpolated.

  For 3-D textures, this filter is analogous to MIPMAP_BILINEAR for the 2-D textures—that is, the filter chooses the prefiltered MIPmap image whose texel size most closely corresponds to screen pixel size and uses the weighted average of the values of the 8 pixels nearest to the *(s*,*t*,*r*) mapping onto that image.

Note: `TX_MIPMAP_TRILINEAR` is available only on SkyWriter, VGXT, and RealityEngine systems.

`TX_MIPMAP_QUADLINEAR*`
chooses the two prefiltered versions of a 3-D texture that have the nearest texel-to-screen pixel size correspondence. A weighted average of the 8 pixels in each of these images that are nearest to the (*s*,*t*,*r*) mapping onto that image is computed. The weighted averages from the two levels are then themselves interpolated.

`TX_MIPMAP_FILTER_KERNEL*`
specifies an 8x8x8 kernel to use as a separable symmetric filter to generate MIPmap levels. Because it is separable and symmetric, only one dimension needs to be specified. The eight floating point values that follow the token specify the kernel. The default that is used for implementations which do not correct for perspective distortion is 0.0, 0.0, 0.125, 0.375, 0.375, 0.125, 0.0, 0.0. The default that is used for implementations which correct for perspective distortion is 0.0, -0.03125, 0.05, 0.48125, 0.48125, 0.05, -0.03125, 0.0. This filter blurs less than the others.

Other Minification Filters

Minification can be performed without MIPmapping. To minify textures without using a MIPmap, select one of these filters:

Note: Filters marked with an asterisk(*) are currently available only on RealityEngine systems.

`TX_POINT` uses the value of the texel, in either a 2-D or 3-D texture, that is nearest to the *(s,t,r)* mapping onto the texture to color the pixel.

`TX_BILINEAR` uses a weighted average of the values of the 4 texels in a 2-D texture that are nearest to the *(s,t)* mapping onto the texture.

`TX_TRILINEAR*` uses a weighted average of the values of the 8 texels of a 3-D texture that are nearest to the *(s,t,r)* mapping onto the texture.

`TX_BICUBIC*` computes a smoothly weighted average of a 4×4 region of texels in a 2-D texture that are nearest to the *(s,t)* mapping onto the texture.

The drawback of using either the TX_POINT or the TX_BILINEAR filter for minification is that only 1, or 4, of the texture pixels that map onto the area of the pixel being textured are considered in the texture value computation. If the texture is mapped so that it is shrunk by a factor greater than two, it may exhibit *scintillation*, a shimmering or swimming motion as if it is not tacked firmly to the surface, or it may appear to have a *moire* pattern on top of it.

Aliasing artifacts such as these result from *undersampling*— not including in the texture value computation the contributions of all of the texture pixels that map onto the pixel being textured. Artifacts caused by undersampling can be alleviated by using one of the MIPmap filters.

To see how MIPmap filtering reduces aliasing and blockiness, change the *texprops* array of the brick texture to:

```
float texprops[] = {TX_MINFILTER, TX_MIPMAP_BILINEAR,
TX_MGFILTER, TX_BILINEAR, TX_WRAP,TX_REPEAT, TX_NULL};
```

Sometimes you may not want the blurring that results from MIPmap filtering, as is frequently the case when texture alpha is used as a geometry approximating template—for example, in defining the outline of a row of trees. In these circumstances, TX_BILINEAR is a good minification filter choice on systems other than RealityEngine. RealityEngine supports a feature called SharpenTexture, described in Section 18.4, “Using the Sharpen and DetailTexture Features,” to maintain the crispness of edges on textured geometry.

18.3.2 Using Magnification Filters

Magnification filters are used when multiple screen pixels correspond to 1 texel, as shown in Figure 18-12.

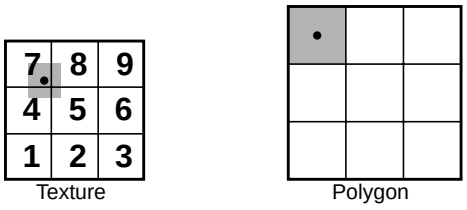

Figure 18-12 Texture Magnification

To select a magnification filter, use the token TX_MAGFILTER, followed by a single symbol that specifies the magnification filter. Values for TX_MAGFILTER are listed below, with descriptions of what they do.

Note: Filters marked with an asterisk (*) are currently available only on RealityEngine systems.

TX_POINT — Used for either 2-D or 3-D textures to select the value of the texel nearest to the (*s*,*t*,*[r*]) mapping onto the screen pixel of the polygon that is being textured. For example, in Figure 18-12, TX_POINT selects texel number 7 for texturing the highlighted polygon pixel.

On systems other than RealityEngine, TX_POINT is generally faster than TX_BILINEAR, but has the drawback that mapped textures can appear boxy because there is not as smooth a transition between the texels as there is with TX_BILINEAR. If the texture image does not have sharp edges, this effect might not be noticeable.

TX_BILINEAR — Used for 2-D textures, to select the weighted average of the values of the 4 texels nearest to the (*s*,*t*) mapping onto the texture. For example, in Figure 18-12, TX_BILINEAR would cause a weighted average of texels 4, 5, 7, and 8 to be used to color the screen pixel.

TX_TRILINEAR* — Used for 3-D textures, to select the weighted average of the values of the 8 texels nearest to the *(s*,*t*,*r*) mapping onto the texture.

TX_BICUBIC* — Used for 2-D textures, to compute a smooth weighted average of a 4×4 region of texels nearest to the *(s*,*t*) mapping onto the texture.

See the *texdef(3G)* man page for the formulas used to compute filter parameters.

See Section 18.4, “Using the Sharpen and DetailTexture Features,” for information on three additional magnification filters— TX_SHARPEN, TX_ADD_DETAIL, and TX_MODULATE_DETAIL—that can be used for enhancing the image quality of magnified textures on RealityEngine systems.

18.4 Using the Sharpen and DetailTexture Features

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.5, “Texture Coordinates,” if you do not have one of these systems.

The appearance of a textured surface can vary, depending on whether it is seen from a distance or close up. For example, from a distance you see the lane markings and reflectors on a road, but close to its surface you see only gravel and tar.

There are two types of problems that occur when the eyepoint is close to a textured surface:

- The texture lacks sufficient detail for close-ups.
- The texture image is out of focus as a result of over-magnification.

RealityEngine provides solutions for these problems with Sharpen and DetailTexture. These two features enable low-resolution textures to be as crisp as high-resolution textures without taking up a lot of texture storage space.

Sharpen works best when the high-frequency information is used to represent edge information. A stop sign is an example of this type of texture—the edges of the letters have distinct outlines. Magnification normally causes the letters to blur, but Sharpen keeps the edges crisp.

DetailTexture works best for a texture with high-frequency information that is not strongly correlated to its low-frequency information. This occurs in images that have a uniform color and texture variation throughout, such as a field of grass or a wood panel with a uniform grain.

18.4.1 Using the Sharpen Feature

Textures must often be magnified for close-up views. However, not all textures can be magnified without looking blurry or artificial. The fine details of a texture, such as the precise edges of letters on a sign, are supplied by high-frequency image data within a high-resolution image. When the high-frequency data is missing, the image is blurred.

Sharpen uses the top two levels of a MIPmap to *extrapolate* high-frequency information *beyond* the texture image in the top level of the MIPmap.

Sharpen lets you use a lower resolution texture map, yet preserve the sharpness of the edges in the original image. This allows you to use less texture storage per texture.

Sharpen maintains edges that bilinear magnification normally blurs. For example, Sharpen works exceptionally well for textures such as the stop sign and for textures whose alpha represents geometry with intricate edges, such as a tree. During the magnification process the edges are extrapolated and they stay crisp.

To use Sharpen, specify the `TX_SHARPEN` token for `TX_MAGFILTER`.

How Sharpen is Computed

The GL computes a Level-of-Detail (LOD) factor at each pixel it textures. LOD is the magnification factor above the base level. LOD *n* is a 2^n magnification. For example, if a 512×512 base texture is LOD 0, its LOD (–1) texture is 256×256.

To produce a sharpened texel *n* LODs above the base texture, the GL adds *n* times the weighted difference between the texel at LOD 0 and LOD (–1) to LOD 0, or

```
LODn = LOD0 + weight(n) * (LOD0 - LOD(-1))
```

where:

n	is the number of levels of extrapolation.
weight(*n*)	is the sharpening multiplier function.
LOD 0	is the base texture.
LOD (–1)	is the texture at half resolution.

By default, the GL uses a linear extrapolation function, where weight(*n*) = *n*.

Customizing the Sharpen Function

Sharpen can cause ringing in some textures when they are magnified too much. The weight can be varied to create a nonlinear LOD extrapolation curve and / or the extrapolation function can be clamped to reduce the ringing.

Figure 18-13 shows LOD extrapolation curves as a function of weight and magnification factors.

The curve on the left is the default linear extrapolation, where weight(n)=1$*n$. The curve on the right is a nonlinear extrapolation, where the weight function is modified to control the amount of sharpening so that less sharpening is applied as the magnification factor increases.

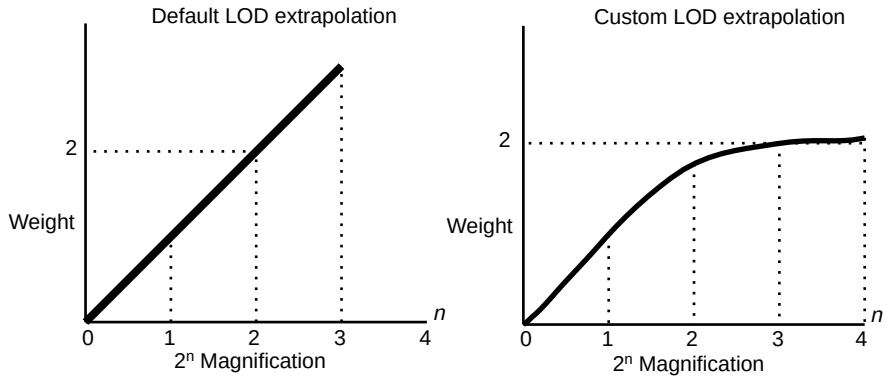

Figure 18-13 LOD Extrapolation Curves

Use `TX_CONTROL_POINT` to specify control points for shaping the sharpen function.The first control point specifies the LOD, and the second control point specifies a weight multiplier for that magnification level.

For example, to gradually ease the sharpening effect—use a nonlinear LOD extrapolation curve, as shown on the right in Figure 18-13—with these control points:

```
TX_CONTROL_POINT, 0., 0.,
TX_CONTROL_POINT, 1., 1.,
TX_CONTROL_POINT, 2., 1.7,
TX_CONTROL_POINT, 4., 2.0,
```

If a texture exhibits ringing when it is magnified with Sharpen—for example, beyond a 6× magnification, you can set the `TX_CONTROL_CLAMP` to clamp at the maximum allowable extrapolation.

Figure 18-14 shows how the default linear extrapolation on the left can be clamped at an arbitrary LOD value, 2.5 in this case, beyond which extrapolation is clamped.

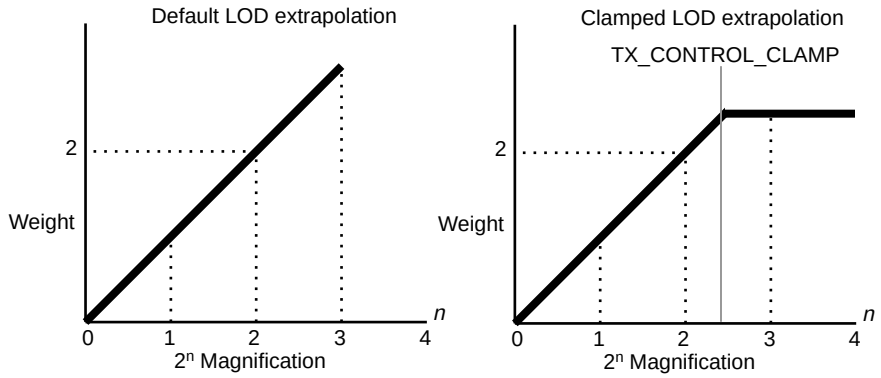

Figure 18-14 Clamping the LOD Extrapolation

Specify a clamp at LOD 2.5 as follows:

```
TX_CONTROL_CLAMP, 2.5
```

You can sharpen the alpha or the color of a texture independently by explicitly setting the magnification filter to use for color and alpha. For example, use the following functions to maintain the precise edges of a geometry described by alpha such as a tree, while allowing the colors to blur:

```
TX_MAGFILTER_ALPHA, TX_SHARPEN,
TX_MAGFILTER_COLOR, TX_BILINEAR,
```

18.4.2 Using DetailTextures

Ideally, you would always use textures that have high enough resolution to allow magnification without bluriness. High-resolution textures maintain realistic image quality for both close-up and distant views. For example, in a high-resolution road texture, both the large features, such as potholes, oil stains, and lane markers that are visible from a distance, as well as the asphalt of the road surface, look realistic no matter where the viewpoint is.

Unfortunately, a high-resolution road texture with that much detail may be as large as 2K×2K, which exceeds the maximum texture storage capacity of the system. Making the image close to or equal to the maximum allowable size still leaves little or no memory for the other textures in the scene.

RealityEngine provides a solution for representing the 2K×2K road texture with the DetailTexture feature.

How DetailTexture Works

The detail elements of a texture, such as the asphalt in a road texture, are the high-frequency components of a high-resolution image. Because the high-frequency detail is virtually the same across a texture such as a road, the high-frequency detail from any portion of the image can be used as the high-frequency detail across the entire image.

Using the same high-frequency detail across the entire image allows the high-resolution image to be represented with the combination of a low-resolution image and a small high-frequency detail image, which is called a DetailTexture. RealityEngine can combine these two images on-the-fly to create an approximation of the high-resolution image.

Creating a DetailTexture and a Low-Resolution Texture

You can convert a high-resolution image into a low-resolution image and a DetailTexture in the following manner:

Make the low-resolution image by shrinking the high-resolution image to the desired resolution. You can then extract the high-frequency detail from the high-resolution image by scaling the low-resolution image back up to the size of the high-resolution image, then subtracting it from the original high-resolution image.

The result is a *difference image* that contains only the high-frequency details of the image. You can use any 256×256 subimage of this difference image as a DetailTexture.

For example, follow these steps to create a 512×512 low-resolution texture, and a DetailTexture from a 2K×2K high-resolution image:

1. Make the low-resolution image as follows:

   Use *izoom* or other resampling program to make the low-resolution image by shrinking the high-resolution image by 2^n. In this example, *n* is 2, so the resolution of the low-resolution image is 512×512. This band-limited image has had the *n* highest frequency bands of the original image removed from it.

2. Make the DetailTexture as follows:

   1. Use *subimage*, or other tool to select a 256×256 region of the original high-resolution image, 2K×2K in this case, whose *n* highest frequency bands are characteristic of the image as a whole.

      For example, rather than choosing a subimage from the lane markings, choose an area in the middle of a lane.

   2. Optionally, you can make this image self-repeating along its edges to eliminate the seams.

   3. Make a blurry version of this 256×256 subimage.

      First, shrink the 256×256 subimage by 2^n, to 64×64 in this case.

      Now, scale the resulting image back up to 256×256.

      This image is blurry because it is missing the two highest frequency bands present in the two highest levels of detail (LOD).

   4. Subtract the blurry subimage from the original subimage. This signed difference image has only the 2 highest frequency bands.

   5. Add a bias to make the image unsigned. If the original image has 8 bits per component, add 128. If the original image has 12 bits per component, add 2048. This is the DetailTexture.

   6. Define and bind the low-resolution texture and the DetailTexture. See “Defining and Binding the DetailTexture” for instructions.

How DetailTexture is Computed

The GL computes the Level-of-Detail (LOD) at each pixel it textures. LOD is the magnification factor above the base level. LOD *n* is a 2^n magnification. In the road example, the 512x512 base texture is LOD 0. The DetailTexture combined with the base texture represents LOD 2, which is called the maximum-detail texture.

When a pixel's LOD is between 0 and 2, the GL linearly interpolates between the texture as it looks at LOD 0 and LOD 2. Linearly interpolating between more than 1 LOD can result in aliasing. To minimize aliasing between the known LODs, the GL lets you specify a nonlinear interpolation curve.

Setting the Detail Control Points

Figure 18-15 shows the default linear interpolation and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

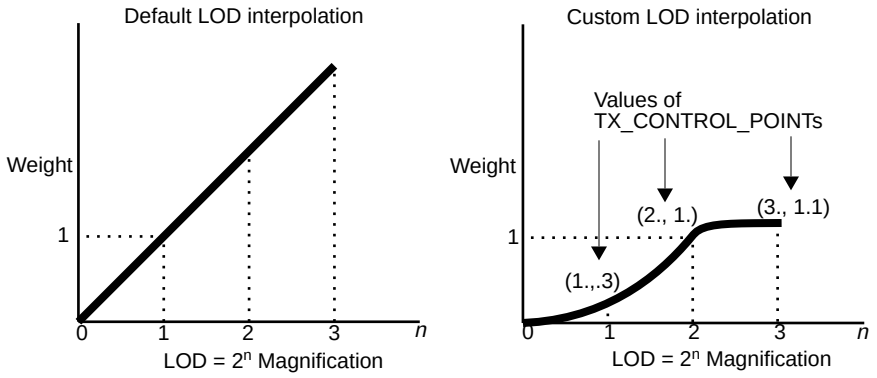

Figure 18-15 LOD Interpolation Curves

The basic strategy is to use very little of the maximum-detail texture until the LOD is within 1 LOD of the maximum-detail texture. More of the information from the maximum-detail texture can be used as the LOD approaches LOD2. At LOD 2, the full amount of detail is used, and the resultant texture exactly matches the high-resolution texture.

Use `TX_CONTROL_POINT` to specify control points for shaping the curve.

The parameters for TX_CONTROL_POINT are LOD and weight, where weight is used in the functions listed in Table 18-4 to control how the DetailTexture is combined with the base texture.

TX_MAGFILTER	Formula
TX_ADD_DETAIL	Factor(n) = weight(n) ∗ DetailTexture
TX_MODULATE_DETAIL	Factor(n) = weight(n) * DetailTexture * base

Table 18-4 Formulas for Computing DetailTexture Filters

The following control points can be used to create a nonlinear interpolation, as shown in Figure 18-15, for the road texture example:

```
TX_CONTROL_POINT, 0.0, 0.0,
TX_CONTROL_POINT, 1.0, 0.3,
TX_CONTROL_POINT, 2.0, 1.0,
TX_CONTROL_POINT, 3.0, 1.1,
```

Notice that making the weight at LOD 3 greater than 1.0 extends the extrapolation beyond the maximum-detail texture, which prevents the texture from blurring beyond a 4× magnification.

Defining and Binding the DetailTexture

For a texture to be used as a DetailTexture, it is bound to the TX_TEXTURE_DETAIL target rather than the familiar TX_TEXTURE_0 target, and used with a texture that has TX_ADD_DETAIL or TX_MODULATE_DETAIL as a magnification filter.

Use TX_DETAIL in the *props* array to define a DetailTexture. TX_DETAIL is followed by five values, *J, K, M, N*, and *scramble*. *J* and *K* must be equal and *M* and *N* must be equal. Currently, *J* and *K* must both be 4 and *scramble* must be zero.

M and *N* describe the mapping of the DetailTexture to the base texture and are given by the following formula:

(EQ 18-1)

$$M, N = \frac{256}{2^{8-n}}$$

where *n* is the number of frequency bands, or LODs, in the DetailTexture.

In the 2K×2K road texture example, the 256×256 detail texture maps to a 64×64 area of the 512×512 low-resolution texture, so the TX_DETAIL parameters for the detail texture are:

```
TX_DETAIL, 4.,4.,64.,64.,0,
```

The magnification filter for the low-resolution texture is:

```
TX_MAGFILTER, TX_ADD_DETAIL
```

or

```
TX_MAGFILTER, TX_MODULATE_DETAIL
```

When a texture is used as a DetailTexture, the properties MINFILTER, MAGFILTER, MAGFILTER_COLOR, MAGFILTER_ALPHA, TX_WRAP, TX_WRAP_S, TX_WRAP_T, TX_WRAP_R, TX_MIPMAP_FILTER_KERNEL, TX_CONTROL_POINT, TX_CONTROL_CLAMP, and TX_TILE have no effect.

Note: The DetailTexture must have the same number of components and the same number of bits per component as the base texture.

The following code fragment provides another example of how to use a DetailTexture:

To define and bind a a DetailTexture, use these properties:

```
TX_DETAIL, 4., 4., 4., 4., 0, TX_NULL
```

To apply a DetailTexture to another texture, use:

```
#define MAX_DETAIL 1.0

TX_MAGFILTER, TX_MODULATE_DETAIL,
TX_CONTROL_POINT, 0., 0.0,
TX_CONTROL_POINT, 0.5, 0.05,
TX_CONTROL_POINT, 2., 0.4,
TX_CONTROL_POINT, 5., MAX_DETAIL,
TX_CONTROL_CLAMP, MAX_DETAIL.

/* a detail texture must be bound and a base texture must be bound */
texbind(TX_TEXTURE_DETAIL, detail_texture_id);
texbind(TX_TEXTURE_0, texture_id);
```

Note: You cannot bind one DetailTexture to another DetailTexture.

18.5 Texture Coordinates

This section describes how to map textures onto object geometry using texture coordinates and how texture coordinates are generated at screen pixels.

To define a texture mapping, you assign texture coordinates to the vertices of a geometric primitive, a process called *parameterization*. You can either assign texture coordinates explicitly with the `t()` subroutines, or let the system automatically generate and assign texture coordinates using the `texgen()` subroutine. You can also use a NURBS texture as described in Chapter 14.

The current texture matrix transforms the texture coordinates. This matrix is set while in `mmode(MTEXTURE)` and is a standard transformation matrix.

The final step generates *(s,t,* [*r, q*]) at every pixel center inside a geometric primitive by interpolating between the vertex texture coordinates as it fills the geometric primitives during scan conversion.

Note: On RealityEngine, a full 3-D projective transformation is supported.

The IRIS-4D / VGX uses hardware to interpolate texture coordinates linearly. Although hardware interpolation is very fast, it is incorrect for perspective projections. The `scrsubdivide()` subroutine improves interpolation—and consequently image quality—for perspective projections on the VGX.

SkyWriter, VGXT, and RealityEngine systems use an enhanced hardware interpolation that does not require the use of `scrsubdivide()`. IRIS Indigo Entry, XS, XS24, and Elan can perform the perspective correction in software if `getgdesc(GD_TEXTURE_PERSP)` = 1.

18.5.1 Assigning Texture Coordinates Explicitly

Use the `t()` subroutines to specify individual texture coordinates explicitly. The argument you specify for `t()` is a 2-, 3-, or 4-element array whose type can be short, long, float, or double. Like vertex coordinates, texture coordinates can be 2-D, 3-D, or 4-D. Specify the texture coordinates *s, t, q,* and *r* in that order for the array. The default for *r* is 0 and the default for *q* is 1.

Note: 3-D and 4-D texture coordinates are currently supported only on RealityEngine.

Table 18-5 lists the formats for the `t()` subroutine.

Array Type	2-D	3-D	4-D
Short integer	`t2s()`	`t3s()`	`t4s()`
Long integer	`t2i()`	`t3i()`	`t4i()`
Float	`t2f()`	`t3f()`	`t4f()`
Double	`t2d()`	`t3d()`	`t4d()`

Table 18-5 The `t()` Subroutine

Call the `t()` subroutines within a `bgnpolygon()/endpolygon()` sequence to texture individual vertices, as illustrated below.

```
bgnpolygon();
   t2f (coord1);
   v3f (vertex1);
   t2f (coord2);
   v3f (vertex2);
   t2f (coord3);
   v3f (vertex3);
   t2f (coord4);
   v3f (vertex4);
endpolygon();
```

18.5.2 Generating Texture Coordinates Automatically

The `texgen()` subroutine generates texture coordinates as a function of object geometry. Coordinates are generated on a per-vertex basis and *override* coordinates specified by the `t()` commands. You can independently control the generation of either or both texture coordinates. If you generate only one coordinate, the other is specified by the `t()` subroutines.

`texgen()` can compute the distance of a vertex from a reference plane and calculate texture coordinates proportional to this distance.

The following form of the plane equation is used to define the reference plane:

$$Ax + By + Cz + D = 0 \qquad \text{(EQ 18-2)}$$

Where the plane normal is the vector (EQ 18-3)

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix}$$

and the plane constant is D.

For example, the plane X=Y that passes through the origin is {1., -1., 0., 0.}.

The TG_LINEAR mode defines the reference plane in object coordinates so that the parameterization is fixed with respect to object geometry. For example, use TG_LINEAR to texture terrain for which sea level is the reference plane. In this case, the altitude of a terrain vertex is its distance from the reference plane. Use TG_LINEAR to make the vertex altitude index the texture so that white snow is mapped onto peaks and green grass is mapped onto foothills.

The following code fragment illustrates how to use the TG_LINEAR function to generate *s* coordinates proportional to vertex distance from the object coordinate plane. The first call to texgen() defines the generation algorithm for the *s* coordinate. The second call activates coordinate generation so that the system generates an *s* coordinate for each vertex.

```
float tgparams[] = {1., -1., 0., 0.};
texgen(TX_S, TG_LINEAR, tgparams);
texgen(TX_S, TG_ON, tgparams);
```

The TG_CONTOUR mode defines the specified plane in eye coordinates. The ModelView matrix in effect at the time of mode definition transforms the plane equation. Thus, the transformation matrix is not necessarily the same as that applied to vertices. This mode establishes a “field” of texture coordinates that can produce dynamic contour lines on moving objects.

The TG_SPHEREMAP mode defines parameters for reflection mapping, by generating texture coordinates based on the vertex and current normal. This causes a reflection of the nearby surrounding environment to map to the surface.

18.5.3 Texture Lookup Tables

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.6, "Texture Environments," if you do not have one of these systems.

RealityEngine supports the use of a texture lookup table (TLUT) for translating texture function outputs. Texture function outputs are used by the texture environment to modify the screen pixel color. The texture environment function is defined by `tevdef()` and selected by `tevbind()`.

For textures up to 8-bit per component, 1- or 2-component textures can reference an 8-bit lookup table of 8-bit per component R,G,B,A values to produce full-colored and translucent imagery from intensity textures. This saves memory space and increases the overall texture capacity of the system.

The texture lookup table is defined by `tlutdef()` and selected by `tlutbind()`.

The ANSI C specification for `tlutdef()` is:

```
void tlutdef(long index, long nc, long len,
             unsigned long *table, long np, float *props)
```

where:

index is the name of the texture look-up table being defined. Index 0 is reserved as a null definition, and cannot be redefined.

nc is the number of components per table entry.

len is the length of table in table entries.

table is a long-word aligned array of packed nc, 8-bit, component table entries.

np is the number of symbols and floating point values in props, including the termination symbol `TL_NULL`. If np is zero, it is ignored. Operation over network connections is more efficient when np is correctly specified, however.

props is an array of floating point symbols and values that define the texture look-up table. props must contain a sequence of symbols, each followed by the appropriate number of floating point values. The last symbol must be `TL_NULL`.

The ANSI C specification for `tlutbind()` is:

```
void tlutbind(long target, long index)
```

where:

target is the texture resource to which the texture function definition is to be bound. The only appropriate resource is `TL_TLUT_0`.

index is the name of the texture function that is being bound. Name is the index passed to `texdef2d()` when the texture function is defined.

By default, texture look-up table definition 0 is bound to `TL_TLUT_0`. Texture look-up table use is enabled when a texture function definition other than 0 is bound to `TX_TEXTURE_0`, a texture environment definition other than 0 is bound to `TV_ENV_0`, and a texture look-up table definition other than 0 is bound to `TL_TLUT_0`.

Table 18-6 shows the relationship between the number of components in the texture look-up table, the number of components in the texture, and the resultant action.

TLUT *nc*	Texture *nc*	Action
2	1	I looks up I,A
	2	I,A looks up I,A
	3	R,G,B passes through unchanged
	4	R,G,B,A passes through unchanged
3	1	I looks up R,G,B
	2	I,A passes through unchanged
	3	R,G,B looks up R,G,B
	4	R,G,B,A passes through unchanged
4	1	I looks up R,G,B,A
	2	I looks up R,G,B; A looks up A.
	3	R,G,B,B looks up R,G,B,A
	4	R,G,B,A looks up R,G,B,A

Table 18-6 Texture Look-up Table Actions

The following code fragment demonstrates how to use texture lookup tables:

```
maketable4()
{
 int i;
 unsigned long table[256];
 float tlutps[] = {TL_NULL};

/* inverts colors */
 for (i = 0; i < 256; i++){
 table[i] = ((255-i)<<24) | ((255-i)<<16) | ((255-i)<<8) | ((255-i));
 }
 tlutdef(4,4,256,table,0, tlutps);
 tlutbind(0,4);
}
```

18.5.4 Improving Interpolation Results on VGX Systems

This section describes a technique that applies only to IRIS-4D / VGX systems, so you might want to skip to Section 18.6, “Texture Environments,” if you do not have one of these systems.

On the VGX, texture coordinates are linearly interpolated in screen space by hardware the same way color is interpolated. Although this produces fast rendering, it is mathematically incorrect for perspective projections. For example, you can modify the sample program by replacing the `ortho()` subroutine with `perspective(600, 1., 1., 16.)`, which introduces perspective distortion. Because of incorrect interpolation, textures no longer appear fixed to a surface but shift as the surface moves. This effect is called *swimming*.

Swimming occurs because texture coordinates are interpolated after the perspective division (in screen coordinates) when they should be interpolated in eye coordinates. Because the VGX hardware does not support eye coordinate interpolation, you can use screen subdivision to improve texture coordinate interpolation. Screen subdivision can also improve the accuracy of fog by correctly interpolating *w* (see Chapter 13).

Note: On systems other than VGX, you *should not* use `scrsubdivide()` because the texture coordinates are already interpolated correctly in eye coordinates.

Use `scrsubdivide` to turn screen subdivision on or off. Use `SS_OFF` to turn off subdivision, which is the default.

Use the `SS_DEPTH` algorithm to subdivide polygons and lines into smaller pieces. Colors, texture coordinates, and the homogeneous coordinate *w* at newly generated vertices are correctly interpolated in eye coordinates rather than in screen coordinates. Because incorrect interpolation is limited to smaller pieces, error globally decreases and image quality increases. Consequently, you can "tune" image quality by modifying the amount of subdivision.

Note: `scrsubdivide()` is most effective for large, nontessellated polygons and lines. Highly tessellated surfaces (for example, curved surfaces) have, in essence, already been subdivided and thus benefit little from further subdivision.

`SS_DEPTH` subdivision slices screen coordinate polygons and lines by a fixed grid in *z*. Spacing between *z* planes is constant throughout the grid and is determined by the three `scrsubdivide()` parameters: maximum screen *z*, minimum screen size, maximum screen size. The first value in the parameter list specifies the desired distance between subdivision planes in units set by `lsetdepth()`.

If polygon slices generated using this metric span a distance in pixels less than minimum screen size, the distance between subdivision planes is increased until the slices are larger than the minimum screen size. This can occur when a polygon is oriented edge-on, so that it spans little screen distance.

If polygon slices generated using the maximum screen *z* metric span a distance in pixels greater than maximum screen size, the distance between subdivision planes is decreased until the slices are smaller than the maximum screen size. This parameter is often useful for polygons that face the viewer and suffer from too little subdivision.

In practice, the minimum and maximum screen size parameters are used to keep slices from becoming too small or too big, respectively. However, these parameters can introduce situations in which polygons that share an edge are sliced by differently spaced grids. This generates *T-vertices* that can cause pixel dropout along the shared edges. To avoid T-vertices, you can "turn off" the

screen size parameters by setting them to 0 so that only the maximum screen *Z* parameter is used. You can turn off any parameter by setting it to 0. For example, a parameter list of {0., 0., 10.} specifies subdivision every 10 pixels.

The following code fragment illustrates how to use screen subdivision:

```
float scrparams[] = {0., 0., 10.};
scrsubdivide(SS_OFF, scrparams);
```

To turn on screen depth subdivision, change the `SS_OFF` mode to `SS_DEPTH`. With the parameter list of {0., 0., 10.}, the quadrilateral is subdivided every 10 pixels and the image quality is improved. You can view the tessellation produced by `scrsubdivide()` by drawing only the polygon outlines, using the `polymode(PYM_LINE)` subroutine (see Chapter 2).

18.6 Texture Environments

A texture environment specifies how texture values modify the color and opacity of an incoming shaded pixel. Use the `tevdef()` subroutine to define a texture environment and the `tevbind()` subroutine to enable the texturing environment. As with `texbind()`, there can be only one texture environment bound (active) at a time.

There are three texture environment types:

`TV_MODULATE`	Modulates the polygon surface with the texture. This is the default environment and is valid for 1-, 2-, 3-, and 4-component textures.
`TV_BLEND`	Interpolates between the polygon color and a constant color based on the texel intensity. This environment is valid for 1- and 2-component textures only.
`TV_DECAL`	Applies the texture on top of the polygon color wherever texture alpha is nonzero.

The ANSI C specification for `tevdef()` is:

```
tevdef(long index, long np, float props[])
```

where:

index	is the unique index(name) for the texture environment.

np is the number of elements in the *props* array.

props is a array of floating point constants that defines how the texture is combined with incoming pixels to color screen pixels.

The texture environment function takes a shaded, incoming pixel color (R_{in},G_{in},B_{in},A_{in}) and computed texture values (R_{tex},G_{tex},B_{tex},A_{tex}) as input, and outputs a new color (R_{out},G_{out},B_{out},A_{out}). The equations for each environment are listed in the tables that follow the environment description.

`TV_MODULATE` multiplies the incoming color components by texture values, according to the equations listed in Table 18-7.

1-component	2-component	3-component	4-component
$R_{out} = R_{in} \cdot I_{tex}$	$R_{out} = R_{in} \cdot I_{tex}$	$R_{out} = R_{in} \cdot R_{tex}$	$R_{out} = R_{in} \cdot R_{tex}$
$G_{out} = G_{in} \cdot I_{tex}$	$G_{out} = G_{in} \cdot I_{tex}$	$G_{out} = G_{in} \cdot G_{tex}$	$G_{out} = G_{in} \cdot G_{tex}$
$B_{out} = B_{in} \cdot I_{tex}$	$B_{out} = B_{in} \cdot I_{tex}$	$B_{out} = B_{in} \cdot B_{tex}$	$B_{out} = B_{in} \cdot B_{tex}$
$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$	$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$

Table 18-7 `TV_MODULATE` Equations

`TV_BLEND` blends the incoming color and the active texture environment color, which is a single RGBA constant (R_{const},G_{const},B_{const},A_{const}) according to the equations used for `TV_BLEND` in Table 18-8. The texture environment color is specified with the `TV_COLOR` parameter.

Output Color	1-component	2-component
Red	$R_{out} = R_{in} \cdot (1 - I_{tex}) + R_{const} \cdot I_{tex}$	$R_{out} = R_{in} \cdot (1 - I_{tex}) + R_{const} \cdot I_{tex}$
Green	$G_{out} = G_{in} \cdot (1 - I_{tex}) + G_{const} \cdot I_{tex}$	$G_{out} = G_{in} \cdot (1 - I_{tex}) + G_{const} \cdot I_{tex}$
Blue	$B_{out} = B_{in} \cdot (1 - I_{tex}) + B_{const} \cdot I_{tex}$	$B_{out} = B_{in} \cdot (1 - I_{tex}) + B_{const} \cdot I_{tex}$
Alpha	$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$

Table 18-8 `TV_BLEND` Equations

`TV_COLOR` specifies the constant color used by the `TV_BLEND` environment. Four floating point values, in the range 0.0 through 1.0, must follow this symbol. These values specify R_{con}, G_{con}, B_{con}, and A_{con}. By default, all are set to 1.0.

`TV_DECAL` uses texture alpha (referred to as A_{tex} in the equations below) is used to blend the incoming color and the texture color, according to the blend equations in Table 18-9

Output Color	3-component	4-component
Red	$R_{out} = R_{tex}$	$R_{out} = R_{in} \cdot (1 - A_{tex}) + R_{tex} \cdot A_{tex}$
Green	$G_{out} = G_{tex}$	$G_{out} = G_{in} \cdot (1 - A_{tex}) + G_{tex} \cdot A_{tex}$
Blue	$B_{out} = B_{tex}$	$B_{out} = B_{in} \cdot (1 - A_{tex}) + B_{tex} \cdot A_{tex}$
Alpha	$A_{out} = A_{in}$	$A_{out} = A_{in}$

Table 18-9 `TV_DECAL` Equations

`TV_COMPONENT_SELECT`
allows the use of one or two components from a texture with more components. Some GL implementations may allow 4 component textures with a very small component size, such as 4 bits, which is smaller than the smallest addressable datum. Therefore, a 4 component texture with 4 bits per component may be used as four separate 1-component textures, or two 2-component textures, and so on.

The token is followed by one choice from the following:

`TV_I_GETS_R` uses the red component of a 4-component texture as a 1-component texture.

`TV_I_GETS_G` uses the green component of a 4-component texture as a 1-component texture.

`TV_I_GETS_B` uses the blue component of a 4-component texture as a 1-component texture.

`TV_I_GETS_A` uses the alpha component of a 4- or 2-component texture.

18.7 Texture Programming Hints

After you understand the basics of the IRIS GL texture routines, the following hints can be useful in getting the optimal performance from your system.

Most of these hints apply to RealityEngine, but because of the texturing capabilities of RealityEngine, some do not apply and are so noted. RealityEngine systems feature dedicated texture memory, rather than using framebuffer memory for textures. This provides the ability to display complex, texture-mapped scenes at fast frame rates, thereby improving image quality without sacrificing performance.

RealityEngine provides 4Mbytes of standard, on-line texture memory, stored as two banks of 2Mbyte memory areas. Different texture types and modes can be mixed together within the memory storage space. Built-in texture storage algorithms store textures sequentially in memory for maximum efficiency. The minimum texture size on RealityEngine is 2×2, and the maximum size is 1024×1024. The system can store two R,G,B,A full-color 1024×1024 textures with 16-bit texels.

Overall Hints

- Turn off texturing when you are not drawing textured geometry.

  Remember to turn off texturing when drawing nontextured geometry. Not supplying texture coordinates does *not* disable texturing. Texturing is disabled only with one of the following subroutine calls: `texbind(TX_TEXTURE_0,0)` or `tevbind(TV_ENV0,0)`.

- Texturing works only in RGB mode.

  The behavior of texturing is not defined in color index mode.

- Most texture calls are illegal between `bgn`/`end` sequences.

  With the exception of the `t()` commands, the texture subroutines described in this chapter cannot be called inside of `bgn`/`end` sequences such as `bgnpolygon()`/`endpolygon()`.

Hints for Using texdef2d

- Use images whose dimensions are powers of two whenever possible.

  Internally, the GL works only with images whose dimensions are powers of 2. `texdef2d` automatically resizes images as necessary. To avoid resizing, pass `texdef2d()` images that have widths and heights that are powers of 2.

- Use as few components as necessary.

  The more components a texture has, the longer it takes to map the texture onto a polygon. For optimal speed, use as few components as possible. This applies to RealityEngine unless you specify an internal format. If you are not taking advantage of a texture's alpha, define the texture as a 1- or 3-component texture. If you do not need a full-color texture, define the texture with one or two components.

- Use the simplest filter you need.

  The per-pixel speed of the texture filter functions is related to the number of interpolations the filter has to perform. The filters in order from fastest to slowest are `TX_POINT`, `TX_MIPMAP_POINT`, `TX_MIPMAP_LINEAR`, `TX_BILINEAR`, `TX_MIPMAP_BILINEAR`, `TX_MIPMAP_TRILINEAR` and `TX_TRILINEAR`, `TX_BICUBIC` and `TX_MIPMAP_QUADLINEAR`.

  There is some overhead per polygon for using MIPmap filters. If a scene has a large number of textured polygons, or if the polygons are subdivided finely, performance is improved if MIPmap filters are not used.

- Keep the texture size below the recommended maximum.

  Textures that exceed the maximum dimensions of the graphics hardware are resized to the maximum dimensions. The maximum dimensions for textures using MIPmapping are half of those that do not. The effect is that large textures using MIPmapping are more blurry than those that do not.

- Be aware that `texdef2d()` copies the texture image to memory.

  The image passed to `texdef2d()` is copied. All other data associated with the texture, such as its MIPmap, are saved with it in the user's memory space until the texture is redefined or the program exits.

Hints for Using texbind

- Bind textures as infrequently as possible.

  `texbind` can be a time-consuming operation, especially if the texture is not resident in the graphics hardware. To achieve maximum performance, draw all of the polygons that use the same texture together.

- Texture caching.

  Hardware texture memory is a finite resource managed by the IRIX kernel. The kernel guarantees that the currently bound texture of a program resides in this memory, whenever the program owns the graphics pipe. Beyond that, the kernel keeps as many additional textures as possible in the hardware texture memory. When a texture is bound, if it is not resident in the texture memory and there is not enough room remaining for this texture, one or more of the resident textures are swapped out. To minimize the frequency of this swapping, use smaller textures or try to switch them less often.

Hints for Using scrsubdivide

- Use `scrsubdivide()` only on VGX systems.
- Turn on `scrsubdivide()` only when you need it.

  Because `scrsubdivide()` generates many polygons from each incoming polygon, it is wise to turn off this feature when it is not needed, such as when you are drawing non–texture-mapped polygons or highly tessellated texture-mapped polygons.

- Use only as much subdivision as you need.

  Choose the `scrsubdivide()` parameters carefully. For maximum performance, use only as much subdivision as is necessary. Textures without high frequencies need less subdivision than those with high frequencies.

Hints for Using alpha

- Use `afunction()`, and/or `msalpha()` on RealityEngine for fast drawing of objects with texture alpha.

When the alpha component of a texture is used to approximate a geometry (such as when a texture is used to describe a tree), the polygons must be blended into the scene in sorted order to properly realize the coverage defined by the alpha component. This sorting and blending requirement can be removed by using `afunction()`. `afunction(0, AF_NOTEQUAL)` specifies that only pixels with nonzero alpha be drawn. See the `afunction()` man page for more details. On VGXT, `afunction(128, AF_GREATER)` works well.

- Alphaless systems.

  Systems without alpha memory also lack storage for a fourth texture component. On such systems, the alpha component of 4-component textures always appears to be 255. 1- and 3-component textures behave the same on systems with or without alpha.

18.8 Sample Texture Programs

This sample program, *brick.c*, creates a brick texture and lets you toggle `scrsubdivide()` with the mouse, to view texture "swimming."

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

float texprops[] = {TX_MINFILTER, TX_POINT,TX_MAGFILTER, TX_POINT,
                    TX_WRAP, TX_REPEAT, TX_NULL};

/* Texture color is brick-red */
float tevprops[] = {TV_COLOR, .75, .13, .06, 1., TV_BLEND, TV_NULL};

/* Subdivision parameters */
float scrparams[] = {0., 0., 10.};

unsigned long bricks[] =                    /*Define texture image */
    {0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00000000, 0x00000000,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0x00000000, 0x00000000};
```

```
/* Define texture and vertex coordinates */
float t0[2] = {0., 0.}, v0[3] = {-2., -4.,0.};
float t1[2] = {16., 0.}, v1[3] = {2., -4.,0.};
float t2[2] = {16., 32.}, v2[3] = {2., 4.,0.};
float t3[2] = {0., 32.}, v3[3] = {-2., 4.,0.};

main()
{

    short val;
    int dev, texflag;

    if (getgdesc(GD_TEXTURE) == 0) {
        fprintf(stderr, "texture mapping not availble on this machine\n");
        return 1;
    }
    keepaspect(1, 1);
    winopen("brick");
    subpixel(TRUE);
    RGBmode();
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qenter (LEFTMOUSE, 0);
    mmode(MVIEWING);
    perspective(600, 1., 1., 10.);
    texdef2d(1, 1, 8, 8, bricks, 0, texprops);
    tevdef(1, 0, tevprops);
    texbind(TX_TEXTURE_0, 1);
    tevbind(TV_ENV0, 1);
    texflag = getgdesc(GD_TEXTURE_PERSP);
    translate(0., 0., -6.);          /* Move poly away from viewer */

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    while (TRUE){
        while(qtest()){
        dev = qread(&val);
        switch(dev){
            case ESCKEY: exit(0);
                         break;
            case REDRAW: reshapeviewport();
                         break;
```

```
/* Screen subdivision - use it only if you have a VGX.
 Push the leftmouse button to see "swimming" on VGX's */
            case LEFTMOUSE:
                if (val){
                    switch(texflag){
                        case 0: scrsubdivide(SS_OFF, scrparams);
                                break;
                        case 1: printf("Your machine corrects in hardware\n");
                                break;
                    }
                }
                else
                    switch(texflag){
                        case 0: scrsubdivide(SS_DEPTH, scrparams);
                             break;
                        case 1: break;
                    }
                    break;
            } /* end main switch */
    } /* end qtest */
        cpack(0x0);
    clear();
    pushmatrix();
    rotate(getvaluator(MOUSEX)*5,'y');
    rotate(getvaluator(MOUSEY)*5,'z');
    cpack(0xffcccccc);
    bgnpolygon();
        t2f(t0); v3f(v0);
        t2f(t1); v3f(v1);
        t2f(t2); v3f(v2);
        t2f(t3); v3f(v3);
    endpolygon();
    popmatrix();
    swapbuffers();
}
texbind(TX_TEXTURE_0, 0); /* Turn off texturing */
}
```

This sample program, *heat.c*, illustrates texture mapping in color map mode.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

/* Texture environmnet */
float tevprops[] = {TV_MODULATE, TV_NULL};

/* RGBA texture map representing temperature as color and
 * opacity */
float texheat[] = {TX_WRAP, TX_CLAMP, TX_NULL};
/* Black->blue->cyan->green->yellow->red->white */
unsigned long heat[] = /* Translucent -> Opaque */
    {0x00000000, 0x55ff0000, 0x77ffff00, 0x9900ff00,
     0xbb00ffff, 0xdd0000ff, 0xffffffff};

/* Point sampled 1 component checkerboard texture */
float texbgd[] = {TX_MAGFILTER, TX_POINT, TX_NULL};

unsigned long check[] =
    {0xff800000, /* Notice row byte padding */
     0x80ff0000};

/* Subdivision parameters */
float scrparams[] = {0., 0., 10};

/* Define texture and vertex coordinates */
float t0[] = {0., 0.}, v0[] = {-2., -4., 0.};
float t1[] = {.4, 0.}, v1[] = { 2., -4., 0.};
float t2[] = {1., 0.}, v2[] = { 2., 4., 0.};
float t3[] = {.7, 0.}, v3[] = {-2., 4., 0.};
```

```
main()
{

    long    device;
    short   data, sub = 0;

    if (getgdesc(GD_TEXTURE) == 0){
        fprintf(stderr,
            "Texture mapping not available on this machine\n");
            return 1;
    }
    keepaspect(1,1);
    winopen("heat");
    RGBmode();
    doublebuffer();
    gconfig();
    subpixel(TRUE);
    lsetdepth(0x0, 0x7fffff);

    blendfunction(BF_SA, BF_MSA); /* Enable blending */

    mmode(MVIEWING);
    perspective(600, 1, 1., 16.);

    /* Define checkerboard */
    texdef2d(1, 1, 2, 2, check, 0, texbgd);
    /* Define heat */
    texdef2d(2, 4, 7, 1, heat, 0, texheat);
    tevdef(1, 0, tevprops);
    tevbind(TV_ENV0, 1);

    translate(0., 0., -6.);
    qdevice(ESCKEY);

    /* Determine if machine does perspective correction */
    if (getgdesc(GD_TEXTURE_PERSP) != 1) sub = 1;
```

```
    while(TRUE) {
        if(qtest()){
            device = qread(&data);
            switch(device){
                case ESCKEY: texbind(TX_TEXTURE_0, 0); /* Turn off texturing */
                             exit(0);
                             break;
            case REDRAW: reshapeviewport();
                             break;
            }
     }
     cpack(0x0);
     clear();
                                          /* Subdivision off */
     if (sub) scrsubdivide(SS_OFF, scrparams);
     texbind(TX_TEXTURE_0, 1);           /* Bind checkerboard */
     cpack(0xff102040);                  /* Background rectangle color */

     bgnpolygon();                       /* Draw textured rectangle */
        t2f(v0); v3f(v0);                /* Notice vertex */
        t2f(v1); v3f(v1);                /* coordinates are used */
        t2f(v2); v3f(v2);                /* as texture coordinates */
         t2f(v3); v3f(v3)
     enpolygon();

     pushmatrix();
        rotate(getvaluator(MOUSEX)*5, 'y');
        rotate(getvaluator(MOUSEY)*5, 'x');

  /* Screen subdivision - use it only if you have a VGX */
     if (sub) scrsubdivide(SS_DEPTH, scrparams);
     texbind(TX_TEXTURE_0, 2);                          /* Bind heat */
     cpack(0xffffffff);              /* Heated rectangle base color */
     bgnpolygon();                   /* Draw textured rectangle */
        t2f(t0); v3f(v0);
        t2f(t1); v3f(v1);
        t2f(t2); v3f(v2);
         t2f(t3); v3f(v3);
     endpolygon();

     popmatrix();
     swapbuffers();
    }
}
```