Texture Mapping 18

This chapter describes the XGL texture mapping capabilities. The chapter includes information on the following topics:

- How a texture image is created and mapped to a surface
- Color interpolation and composition
- Attributes and data structures used in texture mapping

Overview of Texture Mapping

Texture mapping in computer graphics generally refers to the process of mapping a two-dimensional image onto geometric primitives. The primitives are annotated with an extra set of 2D coordinates that orient the image on the primitive. The coordinate system axes of the image space are typically denoted u and v for the horizontal and vertical axes, respectively. When the geometry is processed, the texture is applied to the geometry and appears draped over the geometry definition like paint or cloth.

The texture to be draped on the geometric primitive can be stored as an array of colors that will eventually be mapped onto the polygonal surface. The surface to be textured is specified with vertex coordinates and texture coordinates (u,v), the latter being used to map the color array on the polygon's surface. The u and v are interpolated across the span and then used as indices into the texture map to obtain the texture color. This color is combined with the

primitive color (obtained by interpolating vertex colors across spans) or the colors specified by the application to obtain a final color value at the pixel location.

Texture maps do not have to be color arrays; they can also be arrays of intensities used for color modulation. In this case, the application can specify two colors to modulate with the intensity, or it can take one of the colors from the primitive. The software takes the colors and uses the intensity in the texture map to determine how much of each color to blend to produce the color of the pixel. This is useful for defining mottled textures found in landscape or cloth.

Texture maps can provide reasonable images in only very constrained situations. For a more robust implementation, MIP maps are required.

MIP Maps

The ideal projection of pixel areas results in a computationally expensive process. This process may involve filtering the source image texture values (texels) over arbitrarily shaped large areas. A MIP map approach provides an approximation to an ideal filtering of an image. In MIP mapping, a series of filters is applied to the original texture image. Each successive filter results in successively smaller, filtered versions of the orginal texture. Each of these filtered images constitutes a layer of the MIP map pyramid.

Because the MIP map pyramid represents a hierarchy of filters applied to a single source image, the visual quality of a particular texture is strongly related to the characteristics of the filters used. The XGL library supports several methods for combining samples from a pyramid to produce an output sample, such as point sampling, bilinear interpolation on a single level, or trilinear interpolation between two levels.

Multiple Textures

Using XGL, the application can apply several textures to an individual primitive while rendering. This capability is important for certain images. For example, in rendering a helicopter, an application could texture the sides with camouflage painting and decal text over the camouflage. This is done by applying two textures: a first texture for the camouflage, and a second texture for the text. The second texture uses α values to make the text transparent

everywhere except where the letters appear. The application can also specify the stage in the rendering process where a texture should apply, for example, before or after lighting is applied, or after depth cueing is applied.

How Texture Mapping Works in XGL

XGL provides texture mapping functionality via the MipMap Texture object and the Texture Map object. The MipMap Texture object encapsulates texture images and provides the filtering capabilities to generate a MIP map. The Texture Map object references a texturing description structure that specifies how textures are applied to a primitive.

Note – At this release, a new object, the Texture Map object, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but it will be removed from the XGL library at a future release. It is recommended that application programs use the Texture Map object.

Overview Steps for Mapping a Texture to a Surface

You map a texture to a surface by completing these general procedures.

1. Define the texture data by creating a texture image and writing the image data to one or more Memory Raster objects. See “Defining the Texture Data” on page 460.
2. Specify the properties of the texture in the Texture Map object. Note that unexpected results may occur if you mix the use of the Texture Map object and the Data Map Texture object in the same Context. See “Specifying the Properties of the Texture” on page 466.
3. Provide texture coordinates in the vertex data of a primitive. The texture coordinates are used to orient a texture map on the primitive. See “Mapping the Texture to a Surface Primitive” on page 487

For an example of a texture mapping program, see page 490.

Note – Texturing is only supported when surfaces are rendered through a 3D Context object and when the Raster attached to the 3D Context has a color type of `XGL_COLOR_RGB`.

18

Defining the Texture Data

In XGL, texture image data resides in a MIP map pyramid. A MIP map pyramid stores an array of prefiltered versions of the texture image. Each image in the array has half the dimension of the image before it. The MIP map pyramid typically consists of a series of levels, with each level containing an image. The MIP map data is encapsulated in a MipMap Texture object, which is used by a texture object when rendering textured primitives.

Before the texture can be stored in a MipMap Texture object, it must be written into an XGL Memory Raster object. There are two ways for an application to provide texture data:

- Store the image in a Memory Raster object and then use an XGL utility operator to create the MIP map. The utility creates the MIP map pyramid and encapsulates it in a MipMap Texture object.
- Provide an application-defined MIP map and store the levels of the MIP map in an array of Memory Rasters, with one level of the MIP map in each Memory Raster.

The sections below provide information on defining the texture data in these ways.

Letting XGL Build the Mip Map

XGL automatically creates a MIP map from a texture pattern in a Memory Raster object. To generate a MIP map in this way, follow these procedures:

1. Create a Memory Raster object and write the texture pattern into it. See "Writing the Texture Data into a Memory Raster" on page 461.
2. Create a MipMap Texture object. See "Creating a MipMap Texture Object" on page 462.
3. Create a MIP map using the MipMap Texture operator `xgl_mipmap_texture_build()` to automatically generate the MIP map pyramid from the Memory Raster. "Converting the Texture Data to MIP Map Format" on page 463.

Writing the Texture Data into a Memory Raster

A texture image consists of an array of values for each texel. These values can provide red, green, and blue color values, an α value, or a scale factor. Each 8 bits of data provides one piece of information, such as the red component of the color at the pixel; this piece of information is referred to as a channel. A texture image can have from one to four channels.

The texture for a texturing operation is initially written into an XGL Memory Raster object. The Memory Raster for a MipMap object can be 8-bit or 32-bit, where a Memory Raster with a depth of 8 bits is used to create a one-channel texture image, and a Memory Raster with a depth of 32 bits is used to create a four-channel texture image. The application can create a 24-bit texture image by creating a 32-bit Memory Raster and setting the depth to 24; in this case, the extra 8 bits of the Memory Raster are ignored. A Memory Raster object with four channels is created as follows.

```
mem_ras = xgl_object_create(sys_state, XGL_MEM_RAS, NULL,
                            XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                            XGL_RAS_DEPTH, 32,
                            XGL_RAS_HEIGHT, 8,
                            XGL_RAS_WIDTH, 8, NULL);
```

The values written into a Memory Raster are unsigned, and each channel has a value between 0 and 255. During a texturing operation, the value in each channel is divided by 255 to obtain a floating point value, and this value is then used in the texturing computation. In a four-channel texture map, the color values are stored in RGBα format, that is, the uppermost byte is α, followed by 8 bits each of blue, green, and red intensity values.

Note – The color type of the Memory Raster used in the texture image creation is ignored during texturing operations. Thus, the values in the Memory Raster are used as is, and no color conversion is performed.

Once the Memory Raster object is created, its address is retrieved, and the texture pattern is written into memory. To get the address of a Memory Raster, use the attribute `XGL_MEM_RAS_IMAGE_BUFFER_ADDR`, as shown in the code fragment that follows.

```
/* Create a mem ras object; 8x8 32-bits deep */
 mem_ras_front = xgl_object_create(sys_st, XGL_MEM_RAS, NULL,
                                   XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                    XGL_RAS_DEPTH, depth,
                                    XGL_RAS_WIDTH, width,
                                    XGL_RAS_HEIGHT, height, NULL);

 /* get memory address of the raster */
 xgl_object_get(mem_ras_front, XGL_MEM_RAS_MEMORY_ADDRESS,
                &addr);

 /* write texture into memory */
 tmp_addr = addr;

 /* Width is a multiple of 2 for this example */
 /* Write a checked board pattern into memory */
 for (i = 0; i < height; i += 2) {
    for (j = 0; j < width; j+= 2) {
         *tmp_addr++ =  0xFFFFFFFF;
         *tmp_addr++ = 0x00000000;
     }
     for (j = 0; j < width; j+= 2) {
         *tmp_addr++ =  0x00000000;
         *tmp_addr++ = 0xFFFFFFFF;
     }
 }
```

For more information on the memory representation of a Memory Raster, see the reference manual page for the `XGL_MEM_RAS_IMAGE_BUFFER_ADDR` attribute. For information on Memory Raster objects and attributes, see also Chapter 5, “Devices”.

Creating a MipMap Texture Object

To create a MipMap Texture object, use `xgl_object_create()` as follows:

```
mipmap = xgl_object_create(sys_state, XGL_MIPMAP_TEXTURE, NULL, 0);
```

Converting the Texture Data to MIP Map Format

To request that the MipMap Texture object generate the MIP map, use the operator `xgl_mipmap_texture_build()`. This operator takes as input a single Memory Raster, which is used as the base texture to compute an entire MIP map pyramid of Memory Rasters on behalf of the application. XGL uses the Mitchell filter to create the MIP map.

The application first must define the number of levels for the MIP map. To compute the number of levels for a square Memory Raster, take the logarithm to the base of 2 of the width or height. If the Memory Raster is not square, take the logarithm to the base of 2 of the smaller of the width or height. Set the attribute `XGL_MIPMAP_TEXTURE_LEVELS` to the value.

Once the number of levels has been defined, the application can call the `xgl_mipmap_texture_build()` operator. This operator is defined as:

```
xgl_mipmap_texture_build(mipmap, in_mem_ras, u_bound, v_bound);
```

where:

`mipmap`	A MipMap Texture object
`in_mem_ras`	The Memory Raster object that the application wants to turn into a MIP map within the MipMap Texture object
`u_bound, v_bound`	Specify how the application wants to handle texture borders. These hints are used during the MIP map creation. The values for `u_bound`, `v_bound` are listed in Table 18-1.

Table 18-1 Texture Border Conditions

Value	Description
XGL_TEXTURE_BOUNDARY_WRAP	The texture is repeated across the face of a polygon, and the border between each copy should appear seamless.
XGL_TEXTURE_BOUNDARY_TRANSPARENT	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will be transparent. This option could also be used if the texture were going to be wrapped and the border between each copy does not need to appear seamless.
XGL_TEXTURE_BOUNDARY_CLAMP	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will clamp to the values given by the application in the `boundary_values` field of the *Xgl_texture_mipmap_desc* structure.
XGL_TEXTURE_BOUNDARY_MIRROR	Texel sampling is reversed across the texture map, creating a mirrored alternating pattern of the texture map across the primitive.
XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will clamp to a color specified by the closest boundary texel.

The Memory Raster data is moved to a device-independent MIP map format internal to the MipMap Texture object; thus, after the `xgl_mipmap_texture_build()` operator has been called, the original Memory Raster can be deleted.

Creating a MipMap Texture object and calling `xgl_mipmap_texture_build()` is shown in the example below.

```
/* Create a front MipMap Texture object */
    front_tm = xgl_object_create(sys_st, XGL_MIPMAP_TEXTURE,
                                 NULL, 0);

/* Do the filtering operation on the front mipmap */
/* Set the number of levels for the front MipMap Texture object */
    xgl_object_set(front_tm, XGL_MIPMAP_TEXTURE_LEVELS, 3, NULL);
```

```
/* Set the boundary conditions for generating the MIP map */
    u_bound = XGL_TEXTURE_BOUNDARY_WRAP;
    v_bound = XGL_TEXTURE_BOUNDARY_WRAP;

/* Build  a MIP map pyramid of level 3 from the mem ras */
xgl_mipmap_texture_build(front_tm, in_mem_ras, u_bound, v_bound);
```

Using an Application-Supplied MIP Map

The application can create its own MIP map data, generating an array of Memory Rasters with the filtered data needed for each MIP map level. In this case, the application uses its own methods of filtering and stores the MIP map pyramid in a list of Memory Rasters using standard ways of creating and drawing into Memory Rasters. It then passes this set of Memory Rasters to the MipMap Texture object. The dimension of each Memory Raster in the list should have half the dimension of the previous Memory Raster.

When the application has stored its MIP map pyramid in XGL's Memory Raster format, it must set the number of MIP map levels with the attribute `XGL_MIPMAP_TEXTURE_LEVELS`. The number of Memory Rasters in the list equals the number of levels.

When the number of levels has been specified, the application passes the data in the Memory Rasters to the MipMap Texture object using the `XGL_MIPMAP_TEXTURE_MEM_RAS_LIST` attribute and `xgl_object_set()`, as in the example below. The `XGL_MIPMAP_TEXTURE_MEM_RAS_LIST` attribute specifies an array of Memory Rasters that form a MIP map pyramid. Each level of the pyramid has dimensions that are half as large as the previous level.

```
xgl_object_set(mipmap,
           XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, memras_ptr_list,
           NULL);
```

This call copies the application MIP maps into a device-independent format internal to XGL. When the data has been copied, the original Memory Rasters can be deleted.

Determining the Size of the Base Texture Image

To determine the size of the base texture image, use the read-only attributes `XGL_MIPMAP_TEXTURE_WIDTH` and `XGL_MIPMAP_TEXTURE_HEIGHT`. These values are set indirectly when the application uses the operator `xgl_mipmap_texture_build()` or sets the attribute `XGL_MIPMAP_TEXTURE_MEM_RAS_LIST`. The `XGL_MIPMAP_TEXTURE_DEPTH` attribute defines the number of bits used to specify one texture element. The number of channels per texture element is obtained by dividing the depth of the texture by 8.

Specifying the Properties of the Texture

There are two objects that the application can use to associate texture mapping information with a 3D Context: the Texture Map object and the Data Map Texture object. Each object references the image data in a MipMap Texture object for the texturing data. The differences between the two texture objects are as follows:

- Texture Map object – Has a one-to-one correspondence with a MipMap object. This is the recommended interface for this release, and it will be the only supported interface for future releases.
- Data Map Texture object – Can reference multiple textures in an array of MipMap objects. This interface is supported at this release but may not be supported in future releases.

The general procedure for setting up these objects is as follows:

1. Create a texture object of either type.

Note – Do not use Texture Map objects and Data Map Texture objects with the same Context, or unexpected results may occur. Use Texture Map objects unless the application has already been written with Data Map Texture objects.

---

2. Each object has a texturing description structure that defines the MipMap Texture object(s) to be used, specifies how texel color is obtained, and describes boundary conditions for the texture. Fill in the texturing description structure with the appropriate control parameters and associate the structure with the object.

3. Associate the texture object with a 3D Context.

The following sections provide information on setting up these objects and their associated data structures.

Creating a Texture Map Object

A Texture Map object is created with `xgl_object_create()`:

```
texture_map = xgl_object_create(sys_state, XGL_TMAP,
                                NULL, NULL);
```

An application defines the texturing control parameters and the texture image (defined in the MipMap Texture object) in a texture descriptor structure. The descriptor structure is associated with the Texture Map object using the `XGL_TMAP_DESCRIPTOR` attribute, as shown below.

```
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);
```

Specifying Texture Properties With the Texture Map Object

The texture descriptor structure for the Texture Map object is *Xgl_texture_general_desc:*

```
typedef struct {
    Xgl_texture_type                    texture_type;
    Xgl_texture_comp_info               comp_info;
    union {
        Xgl_texture_general_mipmap_desc  mipmap;
        Xgl_usgn32                       unused[64];
    } texture_info;
} Xgl_texture_general_desc;
```

This structure specifies the type of texture map that the application will use, specifies composition information, and provides a description of the MIP map. These fields are discussed in more detail in the following sections.

Texture Type

The texture descriptor defines the type of the texture map. Currently, the only value for texture type is `XGL_TEXTURE_TYPE_MIPMAP`.

Composition Method

The texture descriptor points to a structure that specifies the composition method. The composition method specifies how the texture map is blended with the primitive's existing data. At this release, the only possible composition method is color composition.

```
typedef union {
    Xgl_texture_color_comp_info      color_info;
} Xgl_texture_comp_info;
```

Color Composition

The color composition method defines how the information present in the texture image is used to compose the final color value. The Texture Map object provides several color composition methods, such as blend, decal and modulate modes. The application specifies this using the *Xgl_texture_color_comp_info* structure. This structure is defined as:

```
typedef struct {
    Xgl_usgn32                  num_render_comp_desc;
    Xgl_render_component_desc   render_component_desc[2];
    Xgl_usgn32                  num_channels[2];
    Xgl_usgn32                  channel_number[2];
} Xgl_texture_color_comp_info;
```

The argument `num_render_comp_desc` specifies the number of rendering component descriptors used by the Texture Map object. The rendering component descriptor `render_component_desc` defines where in the pipeline a texture map is applied. The first element of the `render_component_desc` array is used in composing the color, and the second element is used in composing the alpha value. The `num_channels` array specifies how many channels in the texture are used to modify the destination color. Like the `render_component_desc` array, the first element

of the num_channels array is used in composing the color, and the second element is used in composing the alpha value. The value of the num_channels element can be either 3 or 1. If the value of num_channels is 1, the corresponding element in the channel_number field determines the channel number in the texture image that is used in composing the color. For example, if num_channels[0] is 1 and if channel_number[0] is 0, channel number 0 in the texture image is used in composing color. Note that since alpha uses only one channel, num_channels[1] is always 1; however, channel_number[1] specifies which channel in the texture image is used in composing alpha. Note also that channel_number is ignored if num_channels is set to 3.

Rendering Component Descriptor for Final Texel Color

The field render_component_desc describes the stage of the rendering pipeline that is affected by the texture descriptor and describes how the color intensity from the texture image (MipMap Texture object) defined in the texture descriptor is combined to obtain the final color value. The Xgl_render_component_desc structure is defined as:

```
typedef struct {
    Xgl_render_component                    comp;
    Xgl_texture_op                          texture_op;
    union Xgl_op_info {
        Xgl_texture_blend_op                blend;
        Xgl_texture_decal_op                decal;
        Xgl_texture_blend_intrinsic_op      blend_int;
    } op;
} Xgl_render_component_desc;
```

The comp parameter defines the stage in the rendering pipeline that is affected by the texture description. When multiple textures are applied to a single surface, they are applied to the rendering stage designated in the comp field at a time that is most convenient for the renderer. If more than one texture is to apply to the same component, they are applied in the order that their texture descriptors are listed in the Texture Map object. Note that since alpha values are unaffected by lighting or depth cueing, the comp parameter is only valid for color composition and is ignored for render_component_desc[1].

Table 18-2 shows the possible values for the `comp` parameter.

Table 18-2 Texture Mapping in the Rendering Pipeline

Value	Description
`XGL_RENDER_COMP_DIFFUSE_COLOR`	This value designates the diffuse or intrinsic color of the surface as the component to apply the texture to. When textures are applied to this component, the surface can also have lighting applied.
`XGL_RENDER_COMP_REFLECTED_COLOR`	This value designates the color obtained after lighting is modified using texturing.
`XGL_RENDER_COMP_FINAL_COLOR`	This value indicates to XGL that the final resulting color (after lighting and depth cueing) will be modified by a color from the texture.

The `texture_op` parameter defines how the texture map is combined with the current color or alpha value to obtain a final color. Table 18-3 lists the available texture mapping compositing methods. See the `XGL_TMAP_TEXTURE_DESCRIPTOR` man page for mathematical definitions of the methods.

Table 18-3 Texture Mapping Compositing Methods

Value	Description
`XGL_TEXTURE_OP_MODULATE`	Texture values modulate the object color or alpha value.
`XGL_TEXTURE_OP_DECAL`	Decal mode is only available when a four-channel texture map is used. In this mode, the alpha channel of the texture map specifies how to blend the texture color with a constant color, resulting in the texture appearing like a decal on the object.
`XGL_TEXTURE_OP_DECAL_INTRINSIC`	Decal mode is only available when a four-channel texture map is used. In this mode, the alpha channel of the texture map specifies how to blend the texture color with the object color.
`XGL_TEXTURE_OP_BLEND`	The object color is replaced by two constant colors modulated by the texture color. For example, you could define mountains with grassy areas and trees using a blend between brown and green. In the case of alpha, the incoming alpha value is modulated by the texture alpha to produce the final alpha value.

Table 18-3 Texture Mapping Compositing Methods *(Continued)*

Value	Description
XGL_TEXTURE_OP_BLEND_INTRINSIC	The object color and a constant color are blended by the texture map color to produce the destination color. In the case of alpha, the incoming alpha value is modulated by the texture alpha to produce the final alpha value.
XGL_TEXTURE_OP_REPLACE	The texture value replaces the object color, or the texture image alpha value replaces the the alpha value. This mode is only supported when num_channel[0] is 3.

The op parameter in the *Xgl_render_component_desc* provides the constant color for the XGL_TEXTURE_OP_DECAL, XGL_TEXTURE_OP_BLEND, or XGL_TEXTURE_OP_BLEND_INTRINSIC texture mapping compositing methods.

Channel Information

For the compositing methods XGL_TEXTURE_OP_MODULATE, XGL_TEXTURE_OP_BLEND, and XGL_TEXTURE_OP_BLEND_INTRINSIC, channel information can be used as a way to pack information. For color values, the value of num_channels[0] specifies how many channels in the texture are used to modify the destination color. To use a single channel of information to modify the red, green, and blue destination colors, set the num_channels[0] parameter to 1, and set channel_number to the channel in the texture to be used. To use three channels of information, with one channel for each of red, green, and blue destination colors, set num_channels[0] to 3; in this case, channel_number is ignored.

The value of num_channels[1] is used in composing the alpha value. Since alpha uses only one channel, num_channels[1] is always 1; however, the value of channel_number[1] specifies which channel in the texture image should be used in composing alpha. The channel number can be any of the four channels.

MIP Map Descriptor

The MIP map to be used as the texture in a texture descriptor structure is defined by an *Xgl_texture_general_mipmap_desc* structure.

```
typedef struct {
    Xgl_texture                 texture_map;
    float                       max_u_freq;
    float                       max_v_freq;
    Xgl_texture_boundary        u_boundary;
    Xgl_texture_boundary        v_boundary;
    Xgl_usgn8                   boundary_values[4];
    float                       depth_interp_factor;
    Xgl_texture_interp_info     interp_info;
    Xgl_matrix_f2d              orientation_matrix;
} Xgl_texture_general_mipmap_desc;
```

Pointer to MipMap Texture Object

The `texture_map` field specifies the MipMap Texture object that contains the texture image.

Note – At this release, the fields `max_u_freq` and `max_v_freq` are not implemented.

Boundary Conditions

The `u_boundary`, `v_boundary` fields of the *Xgl_texture_mipmap_desc* structure specify how to apply texturing to a primitive when the texture bounds are exceeded. Typically when texturing a polygon, the colors that are used come from the texturing process. However, it is possible to texture a polygon in such a way that the texture does not cover the entire face of the polygon (by specifying *u* and *v* values with the primitive's vertices that are less than 0.0 or greater than 1.0). When this happens, the user can specify whether to wrap (repeat) or mirror (reverse) the texture again and again until the polygon is completely covered, or he can specify a color or intensity to clamp to. Clamping results in a polygon with one copy of the texture and the remaining part of the polygon filled in with the color of a constant clamp value or the boundary texel. He may also specify that the texture becomes transparent once

the edge of the texture bounds are reached, which means that the remaining untextured part of the polygon takes on the appearance that it would have had if the current texture had not been applied.

The `u_boundary`, `v_boundary` fields are flags that indicate how the renderer should finish rendering the polygon when the edge of the texture has been reached. Table 18-4 lists possible values for these fields.

Table 18-4 MipMap Descriptor Boundary Conditions

Value	Description
`XGL_TEXTURE_BOUNDARY_CLAMP`	Finish texturing using a constant value.
`XGL_TEXTURE_BOUNDARY_WRAP`	Repeat the texture.
`XGL_TEXTURE_BOUNDARY_TRANSPARENT`	Finish rendering as though texturing were not being done.
`XGL_TEXTURE_BOUNDARY_MIRROR`	Reverse the texture.
`XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY`	Finish texturing using a color specified by the closest boundary texel.

Note – If the *u* and *v* boundary conditions are not identical (for example, if one specifies clamp and the other specifies transparent), and if both the *u* and *v* values exceed the texture boundary in both directions, then an ambiguous situation arises where the renderer can't determine which method to use. In this case, the renderer follows the following order of precedence in deciding which of the boundary methods to use:

1. `XGL_TEXTURE_BOUNDARY_TRANSPARENT`
2. `XGL_TEXTURE_BOUNDARY_CLAMP`
3. `XGL_TEXTURE_BOUNDARY_WRAP`
4. `XGL_TEXTURE_BOUNDARY_MIRROR`
5. `XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY`

Boundary Condition Clamp Color Source

The `boundary_values` field specifies the value (color) to clamp to when either the *u* or *v* value exceeds the texture boundaries. This is the value used by the `XGL_TEXTURE_BOUNDARY_CLAMP` value of the boundary condition fields. The array index 0 corresponds to red, the index 1 to green, the index 2 to blue, and the index 3 to α.

Depth Sampling

The `depth_interp_factor` field modifies the calculation of which MIP map level to sample. There are times when the application would prefer that the texture be more or less aliased as it is applied to a surface (more jaggy or more blurry). This can be accomplished by changing the texture map depth level at which texels are sampled. This attribute is a floating point number that allows the application to adjust the sample depth level in either direction. A value of 1.0 moves away from the MIP map base towards more blurry levels. A value of -1.0 moves the depth towards the MIP map pyramid's base, which has more detail in it, and may make the picture appear more jaggy.

Texture Sampling Method

During the texturing process, the data from the texture MIP map can be sampled or interpolated in a number of ways in order to come up with a value to apply to the pixel. The application specifies how to sample and interpolate the MIP map data using the *Xgl_texture_interp_info* structure, which is defined as:

```
typedef struct {
    Xgl_texture_interp_method     filter1;
    Xgl_texture_interp_method     filter2;
} Xgl_texture_interp_info;
```

`filter1` is applied when the pixel being textured maps to an area greater than one texel. `filter2` is applied when the pixel being textured maps to an area less than or equal to one texel. The values for the *Xgl_texture_interp_method* structure are the texture sampling methods listed in Table 18-5.

Table 18-5 Texture Sampling Methods

Value	Description
`XGL_TEXTURE_INTERP_POINT`	Sample one texel point from the base top or first MIP map level at the closest *u* and *v* to the pixel.
`XGL_TEXTURE_INTERP_BILINEAR`	Sample the four texels closest to the center of the pixel from the base MIP map level and average them.
`XGL_TEXTURE_INTERP_MIPMAP_POINT`	Sample one texel point from the nearest MIP map level at the closest *u* and *v* to the pixel.

Table 18-5 Texture Sampling Methods

Value	Description
XGL_TEXTURE_INTERP_MIPMAP_BILINEAR	Sample the four texels closest to the center of the pixel from the nearest MIP map level and average them.
XGL_TEXTURE_INTERP_MIPMAP_TRILINEAR	Sample the four texels closest to the center of the pixel from the nearest MIP map level above the depth *d* and the four texels closest to the MIP map level below the depth *d* and average all eight of them.
XGL_TEXTURE_INTERP_MIPMAP_PT_LINEAR	Choose the two MIP maps that most closely match the size of the pixel being textured and find the nearest texel to the center of the pixel to produce a texture value from each MIP map. The final texture value is an average of those two values.

Orientation Matrix

The orientation_matrix field in the MipMap descriptor structure defines a 3×2 matrix with which textures can be rotated or translated across the face of a polygon by transforming the *u* and *v* values of the texture. The matrix is type float. All texture *u* and *v* values are transformed by the matrix as the texturing process takes place. Note that the *u*, *v* values in the primitive point list are subject to the XGL_TMAP_T0_INDEX and XGL_TMAP_T1_INDEX attributes before the orientation matrix is applied.

Associating the Texture Map Object With a 3D Context

To set the Texture Map object on the Context, specify the Texture Map object or the list of Texture Map objects with the XGL_3D_CTX_SURF_FRONT_TMAP attribute or the XGL_3D_CTX_SURF_BACK_TMAP attribute, as in the following example:

```
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 1,
               XGL_3D_CTX_SURF_FRONT_TMAP, &texture_map,
               NULL);
```

An application can apply multiple Texture Map objects to the same primitive or associate a different set of Texture Map objects for the front and back faces of a surface.

Specify the number of Texture Map objects in the texture map array with the attributes `XGL_3D_CTX_SURF_FRONT_TMAP_NUM` or `XGL_3D_CTX_SURF_BACK_TMAP_NUM`. The attributes `XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES` and `XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES` specify a front and back array of on and off switches corresponding to the Context's array of Texture Map objects. After creating the array of switches, the application can turn on or off the use of Texture Map objects using these attributes.

Note – Texturing is only supported when the Raster associated with the 3D Context has a color type of RGB.

Additional Texture Map Functionality

The Texture Map object provides attributes that supply the functionality discussed in the following sections.

Perspective Correction of Textured Surfaces

The attribute `XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION` specifies the method used to compute texture coordinate values in surface interiors when a textured surface primitive is rendered. Possible values for this attribute are:

`XGL_TEXTURE_PERSP_NONE`
: Texture mapping coordinates are linearly interpolated without perspective correction.

`XGL_TEXTURE_PERSP_PIXEL`
: As the texture mapping coordinates are interpolated, their values are manipulated to account for a perspective projection.

`XGL_TEXTURE_PERSP_NO_INTERP`
: Texture mapping coordinates are not interpolated at all. Instead, texture mapping coordinates are used per-vertex only. This method is also known as vertex-level texture mapping.

Vertex-Level Texture Mapping

Vertex-level texture mapping can be considered a naive implementation of texture mapping. Traditional pixel-level texture mapping computes texture color at every pixel. Because of this, the software implementation is usually slow. If texture mapping is not supported in hardware, the software implementation is the only alternative.

Vertex-level texture mapping, on the other hand, computes texture color only per vertex of the primitive, and the interior is then simply Gouraud-shaded. Hardware accelerators that support fast triangle rendering can be used to perform the shading. However, if the original primitives cover large areas, the resulting image may be of poor quality.

To improve the quality of vertex-level texture mapping, an application may need to subdivide large primitives into a union of smaller subprimitives so that per-vertex sampling more accurately reflects the texture map. With NURBS surfaces, however, the application can tessellate the surface into a large number of polygons, thereby increasing texture quality.

Note the following limitations on vertex-level texture mapping:

- The mipmap level is always 0.
- Only front facing surfaces are supported.
- The value of *Xgl_render_component* in the *Xgl_render_component_desc* structure can only be `XGL_RENDER_COMP_DIFFUSE_COLOR`. This means that vertex-level texture mapping can only modify the color of a surface before lighting is applied.

Specifying the Source of Texture Coordinates

The Texture Map object attribute `XGL_TMAP_COORD_SOURCE` specifies the initial source within a primitive's vertex data of the texture coordinate. Depending on a texture map's parametrization method, this may or may not be the final texture coordinate (this initial source may be processed into the final texture coordinate). This attribute has the following values:

`XGL_TEXTURE_COORD_VERTEX`
: The source texture coordinate values are the vertex coordinate values, transformed to WC.

XGL_TEXTURE_COORD_NORMAL
The source texture coordinate values are the vertex normal values, transformed to WC. If a vertex normal is not explicitly defined with the primitive, the calculated vertex normal is used instead.

XGL_TEXTURE_COORD_DATA
The source texture coordinate's source is a member of the vertex's floating point data. (The index value indicates which member of the list.) If the primitive does not include the floating point data list for each vertex, this texture coordinate value will be set to 0. This is the default value.

If XGL_TMAP_COORD_SOURCE is set to XGL_TEXTURE_COORD_DATA, the attributes XGL_TMAP_T0_INDEX and XGL_TMAP_T1_INDEX define the indices into the vertex data array of the surface primitives. XGL_TMAP_T0_INDEX indicates that the *u* value of the data values is used as *u* texture coordinates, and XGL_TMAP_T1_INDEX indicates that the *v* value of the data values is used as *v* texture coordinates. The mapping of the texture onto the polygon face is done by listing *u* and *v* values in the data fields of vertices. Since it is possible to have several data values listed with each vertex, an application must use these attributes to specify which *u* and *v* values to use with the Texture Map object.

Specifying the Parameterization Method

The XGL_TMAP_PARAM_TYPE attribute defines the parameterization method for mapping a texture to a 3D surface. With the attribute XGL_TEXTURE_PARAM_EXPLICIT, the application specifies the mapping of the texture to the surface using the *u*,*v* values in the point list. All other parameterization methods provide environment texture mapping.

Environment texture mapping textures a surface as if it were reflected from a shiny object such as a sphere. The viewer position and the position of the surface determine how objects are reflected. For environment texture mapping, the attribute XGL_TMAP_PARAM_INFO provides information that is used in the generation of texture coordinates.

The XGL_TMAP_PARAM_TYPE attribute has the following values:

XGL_TEXTURE_PARAM_EXPLICIT
The mapping of texture to the surface is explicitly specified by the application by u,v values in the point list. This is the default value.

XGL_TEXTURE_PARAM_REFLECT_SPHERE_EC

The mapping is defined by reflection of the eye-to-vertex vector with respect to the vertex normal in Eye Coordinates. Specifically, the *u*,*v* values are derived as follows:

- Vertex and normal are transformed to World Coordinates. Eye Coordinates are obtained by transforming the World Coordinates by the view orientation, which is an orthogonal portion of the viewing transformation.
- The reflection vector is computed as follows:

  $s = i - 2 \times n \times (n \cdot i)$, where

  s is the reflection vector
  n is the unit vertex normal
  i is the unit eye-to-vertex vector
- The reflection vector s is then transformed by a reflection matrix, yielding vector t. The reflection matrix is specified using the *geom_orient_matrix* field of the *Xgl_texture_projection* structure. This structure is a part of the *Xgl_texture_param_info* structure, which is set using the XGL_TMAP_PARAM_INFO attribute.
- The u, v values are computed as follows:

$$u = 0.5 \times \left(\frac{t_x + 1}{length}\right)$$

$$v = 0.5 \times \left(\frac{t_y + 1}{length}\right),$$

$$length = \sqrt{t_x \times t_x + t_y \times t_y + (t_z + 1) \times (t_z + 1)}$$

XGL_TEXTURE_PARAM_REFLECT_SPHERE_WC

The mapping is defined by reflection of eye-to-vertex vector with respect to vertex normal in World Coordinates. Specifically, the u,v values are derived as follows:

- Vertex and normal are transformed to World Coordinates.
- The reflection vector is computed as follows:

  $s = 1 - 2 \times n \times (n \cdot i)$, where

s is the reflection vector
n is the unit vertex normal
i is the unit eye-to-vertex vector

- The reflection vector s is then transformed by a reflection matrix, yielding vector $\{tx, ty, tz\}$. The reflection matrix is specified using the *geom_orient_matrix* field of the *Xgl_texture_projection* structure. This structure is a part of the *Xgl_texture_param_info* structure, which is set using the `XGL_TMAP_PARAM_INFO` attribute.
- The u,v values are computed as follows:

$$u = 0.75 \text{ if } t_x = 0.0,\ t_z > 0.0$$
$$= 0.25 \text{ if } t_x = 0.0,\ t_z > 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 0.5 \quad \text{if} \quad t_x < 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 1.0 \quad \text{if} \quad t_x > 0.0,\ t_z > 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 0.0 \quad \text{if} \quad t_x > 0.0,\ t_z > 0.0$$

$$v = \frac{\arcsin\left(t_y\right)}{\pi} + 0.5$$

where: $\left(\frac{-\pi}{2} \le \arctan\left(r\right) \le \frac{\pi}{2}\right)$ and $\left(\frac{-\pi}{2} \le \arcsin\left(r\right) \le \frac{\pi}{2}\right)$

XGL_TEXTURE_PARAM_LINEAR_EC
The mapping is defined by a linear projection in Eye Coodinates. The *u,v* values are derived as follows:

- The vertex is transformed into Eye Coordinates (see above), resulting in point {*sx, sy, sz*}.
- The specified two homogeneous vectors *p0* and *p1* are transformed from Eye Coordinates into Model Coordinates. Projection values are specified using the *linear* field of the *Xgl_texture_projection* structure. This structure is part of the *Xgl_texture_param_info* structure, which is set using the XGL_TMAP_PARAM_INFO attribute.
- The u,v values are computed as follows:

$$u = s_x \times p0_x + s_y \times p0_y + s_z \times p0_z + p0_w$$

$$v = s_x \times p1_x + s_y \times p1_y + s_z \times p1_z + p1_w$$

In the case of NURBS surfaces, depending on texture coordinate source and texture parametrization method, either the (u,v) coordinates of the point data, or tessellation vertex and normal of the NURBS are used for the texture coordinates.

The mapping of the texture onto the surface can be modified using the orientation matrix. See “Orientation Matrix” on page 475.

Specifying the Texture Domain

At this release, the XGL_TMAP_DOMAIN attribute specifies that the texture is to be applied to color domain. The attribute’s only value is XGL_TEXTURE_COLOR.

Creating a Data Map Texture Object

A Data Map Texture object is created as follows:

```
dmap_texture = xgl_object_create(sys_state, XGL_DMAP_TEXTURE,
                                 NULL, NULL);
```

An application defines the texturing control parameters and the texture image (MipMap Texture object) using a texture descriptor structure. A Data Map Texture object can have many such texture descriptors. These descriptor structures are associated with the Data Map Texture object using the `XGL_DMAP_TEXTURE_DESCRIPTORS` attribute. The number of texture descriptors is specified by the `XGL_DMAP_TEXTURE_NUM_DESCRIPTORS` attribute. When multiple textures are used, they are applied sequentially and are cumulative.

Specifying Texture Properties With the Data Map Texture Object

The texture descriptor structure for the Data Map Texture object is *Xgl_texture_desc*:

```
typedef struct {
    Xgl_texture_type              texture_type;
    Xgl_texture_color_comp_info   comp_info;
    Xgl_texture_interp_info       interp_info;
    union {
        Xgl_texture_mipmap_desc   mipmap;
        Xgl_usgn32                unused[64];
    } info;
} Xgl_texture_desc;
```

This structure does the following:

- Specifies the type of texture map to be used
- Specifies how the channels in the texture map affect rendering
- Specifies the method of sampling to be used
- Provides a description of the MIP map

These fields are discussed briefly in the sections below. Since much of the information provided in the fields and substructures of this structure is the same as that in the *Xgl_texture_general_desc* structure for the Texture Map object, references to preceding sections are provided, and the information is not duplicated here.

Texture Type

Each texture descriptor defines the type of the texture. This value is used when accessing the fields of the *info* union. Currently, the only texture type value is `XGL_TEXTURE_TYPE_MIPMAP`.

Color Composition Method

The `comp_info` field of the texture descriptor defines how the information present in the texture image is used to compose the final color value. The application specifies this using the *Xgl_texture_color_comp_info* structure. This structure is defined as:

```
typedef struct {
    Xgl_usgn32                  num_render_comp_desc;
    Xgl_render_component_desc   render_component_desc[2];
    Xgl_usgn32                  num_channels[2];
    Xgl_usgn32                  channel number[2];
} Xgl_texture_color_comp_info;
```

This structure defines where in the rendering pipeline a texture map is applied. Color composition functionality is the same in the Texture Map object and the Data Map Texture object. For complete information on color composition, see page 468.

Texture Sampling Method

During the texturing process, the data from the texture MIP map can be sampled or interpolated in a number of ways in order to arrive at a value to apply to the pixel. Texture sampling functionality in the two texture objects is the same, although texture sampling has been moved from the texture descriptor to the MIP map descriptor in the Texture Map object. See “Texture Sampling Method” on page 474 for information on texture sampling.

MIP Map Descriptor

The MIP map to be used as the texture in a texture descriptor structure is defined by an *Xgl_texture_mipmap_desc* structure.

```
typedef struct {
    Xgl_texture           texture_map;
    float                 max_u_freq;
    float                 max_v_freq;
    Xgl_texture_boundary  u_boundary;
    Xgl_texture_boundary  v_boundary;
    Xgl_usgn8             boundary_values[4];
    float                 depth_interp_factor;
} Xgl_texture_mipmap_desc;
```

Texture Map

The `texture_map` field specifies the MipMap Texture object that contains the texture image.

Note – At this release, the fields `max_u_freq` and `max_v_freq` are not implemented.

Boundary Conditions

The `u_boundary`, `v_boundary` fields of the *Xgl_texture_mipmap_desc* structure specify how to apply texturing to a primitive when the texture bounds are exceeded. This functionality is the same as that provided in the boundary conditions functionality in the MIP map descriptor for the Texture Map object. See page 472 for information.

Boundary Condition Clamp Color Source

The `boundary_values` field specifies the value (color) to clamp to when either the *u* or *v* value exceeds the texture boundaries. See “Boundary Condition Clamp Color Source” on page 473.

Depth Sampling

The `depth_interp_factor` field modifies the calculation of which MIP map level to sample. See “Depth Sampling” on page 474.

Associating the Data Map Texture Object With a 3D Context

In XGL, an application can have multiple Data Map Texture objects applied to the same primitive. In addition, XGL supports the capability of associating a different set of Data Map Texture objects for the front and back faces of a surface.

To set the Data Map Texture objects on the Context, specify the number of Data Map Texture objects in the array with the attributes `XGL_3D_CTX_SURF_FRONT_DMAP_NUM` or `XGL_3D_CTX_SURF_BACK_DMAP_NUM`. Then specify the list of Data Map Texture objects with the `XGL_3D_CTX_SURF_FRONT_DMAP` attribute or the `XGL_3D_CTX_SURF_BACK_DMAP` attribute, as in the following example:

```
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_NUM, 1,
               XGL_3D_CTX_SURF_FRONT_DMAP, &dmap_texture,
               NULL);
```

The attributes `XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES` and `XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES` specify a front and back array of on and off switches corresponding to the Context's array of Data Map Texture objects. After creating the array of switches, the application can turn on or off the use of Data Map Texture objects using these attributes.

Note – Texturing is only supported when the Raster associated with the 3D Context has a color type of RGB.

Additional Data Map Texture Attributes

The Data Map Texture object provides attributes that supply the functionality discussed in the following sections.

Orientation Matrix

The `XGL_DMAP_TEXTURE_ORIENTATION_MATRIX` attribute defines a 3×2 matrix with which textures can be rotated or translated across the face of a polygon by transforming the u and v values of the texture. The matrix is type float. All texture u and v values are transformed by the matrix as the texturing

process takes place. Note that the *u*, *v* values in the primitive point list are subject to the XGL_DMAP_TEXTURE_U_INDEX and XGL_DMAP_TEXTURE_V_INDEX attributes before the orientation matrix is applied.

Specifying the Source of Texture Coordinates

The attributes XGL_DMAP_TEXTURE_U_INDEX and XGL_DMAP_TEXTURE_V_INDEX define the indices into the vertex data array of the surface primitives and indicate which of the data values to use as *u* and *v* texture coordinates. The mapping of the texture onto the polygon face is done by listing *u* and *v* values in the data fields of vertices. Since it is possible to have several data values listed with each vertex, an application must use these attributes to specify which *u* and *v* values to use with the Data Map Texture object.

Specifying the Parameterization Method

The XGL_DMAP_TEXTURE_PARAM_TYPE attribute defines the parameterization method for mapping a texture to a 3D surface. For information about the values for this attribute, see "Specifying the Parameterization Method" on page 478.

In the case of NURBS surfaces, the (u,v) coordinates of the NURBS parameter space scaled to the (0,1) range are used for texture coordinates. For other surface primitives, the (u,v) coordinates must be supplied by the application.

The mapping of the texture onto the surface can be modified using the orientation matrix.

Mapping the Texture to a Surface Primitive

Texture is mapped onto a surface primitive via the vertex information in the primitive's point data. The point data type used for a texture-mapped surface varies for the Texture Map object and the Data Map Texture object as follows:

- Texture Map object – The point type can be any 3D point type, and the vertex values, vertex normal values, or the vertex floating point data can be used for the texture coordinates. The Texture Map object attribute `XGL_TMAP_COORD_SOURCE` specifies the source for the texture coordinates within a primitive's vertex data. For example, if the vertex values are used as the texture coordinates, the point's x and y values in world coordinates are used for u and v values of the texture coordinates. If the floating point data is used, the attributes `XGL_TMAP_T0_INDEX` and `XGL_TMAP_T1_INDEX` define the indices into the vertex data array. For information on `XGL_TMAP_COORD_SOURCE`, see "Specifying the Source of Texture Coordinates" on page 477.
- Data Map Texture object – The point type must be one of the `DATA` point types, such as *Xgl_pt_data_f3d*. Data point types include u and v values that correspond to the coordinate space of the texture image. The attributes `XGL_DMAP_TEXTURE_U_INDEX` and `XGL_DMAP_TEXTURE_V_INDEX` define the indices into the vertex data array of the primitives and indicate which of the data values to use as u and v texture coordinates.

Using Data Point Types for Texture Mapping

The coordinate space of the texture MIP map that u values correspond to is a space that places 0.0 at the left edge of the texture and 1.0 at the right edge of the texture. Similarly for v, the coordinate space of the texture MIP map that v values use is a space that places 0.0 at the bottom edge of the texture and 1.0 at the top edge of the texture.

If an application were to map a texture completely onto a four-sided polygon in a simple manner with no wrapping but covering the entire polygon, it would specify at the lower left vertex of the polygon the u,v pair of (0.0, 0.0), at the upper left vertex (0.0, 1.0), at the upper right vertex (1.0, 1.0), and at the lower right vertex (1.0, 0.0). The following code fragment shows the point data for this mapping.

```
Xgl_pt_list      pl[1];
float            pts[20];

pl[0].pt_type = XGL_PT_DATA_F3D;
pl[0].bbox = NULL;
pl[0].num_pts = 4;
pl[0].num_data_values = 2;
pl[0].pts.data_f3d = (Xgl_pt_data_f3d*) &pts[0];

/*
 * For the first point:
 * {x,y,z,u,v} = {100.0, 100.0, 0.0, 0.0, 0.0}
 */
pts[0] = 100.0;
pts[1] = 100.0;
pts[2] = 0.0;
pts[3] = 0.0;
pts[4] = 0.0;

/*
 * For the second point:
 * {x,y,z,u,v} = {100.0, 300.0, 0.0, 0.0, 1.0}
 */
pts[5] = 100.0;
pts[6] = 300.0;
pts[7] = 0.0;
pts[8] = 0.0;
pts[9] = 1.0;

/*
 * For the third point:
 * {x,y,z,u,v} = {300.0, 300.0, 0.0, 1.0, 1.0}
 */
pts[10] = 300.0;
pts[11] = 300.0;
pts[12] = 0.0;
pts[13] = 1.0;
pts[14] = 1.0;

/*
 * For the fourth point:
 * {x,y,z,u,v} = {300.0, 100.0, 0.0, 1.0, 0.0}
 */
pts[15] = 300.0;
pts[16] = 100.0;
pts[17] = 0.0;
```

```
pts[18] = 1.0;
pts[19] = 0.0;
```

Figure 18-1 shows a polygon with texturing.

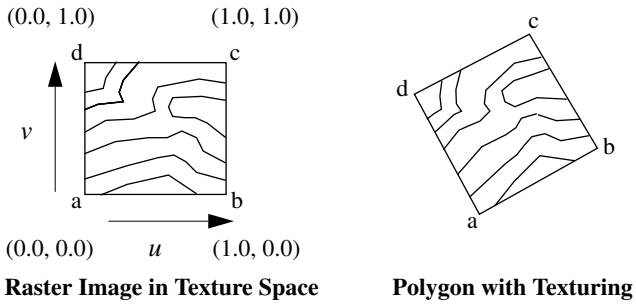

Figure 18-1 Polygon With Texturing

As a second and more involved example, if the application were to map a texture onto a four-sided polygon so that the texture appeared once in the center of the polygon with a border of the original polygon visible around the sides of the texture, then it might specify at the lower left vertex of the polygon the u,v pair of (-0.5, -0.5), at the upper left vertex (-0.5, 1.5), at the upper right vertex (1.5, 1.5), and at the lower right vertex (1.5, -0.5).

If *u* and *v* values are not assigned to polygon vertices in a linear way, that is, if there is no linear transformation that maps *u*,*v* coordinates to the polygon's vertices, then a textured polygon might look different when rendered on different graphics accelerators. This is because some graphics hardware devices break up more complex polygons into triangles, and if the mapping is non-linear, all the triangles will not be textured the same from device to device.

Note – For any frame buffer, some primitives are accelerated and some may go through software rendering. The inherent differences in the rendering may lead to pixel imperfection when the two types of rendering are intermixed in the same scene.

18

Texture Mapping Example Program

The following program illustrates how to add texture mapping to XGL polygons and how to use the Texture Map object and the MipMap Texture object. The program draws a single polygon without texture, then draws the polygon again with `texture_op` set to `XGL_TEXTURE_OP_BLEND`, and finally redraws the polygon after rotation with the `texture_op` set to `XGL_TEXTURE_OP_MODULATE`. To compile the program, type `make texture`.

Code Example 18-1 Texture Mapping Example

```
/*
 * texture.c
 */

#include <stdio.h>
#include <xgl/xgl.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>

static  Display   *display;  /* the connection to the server */
static  int       screen;    /* the screen within the display */
static  Visual    *vis;      /* the visual within the screen */
static  Colormap   cmap;     /* the colormap within the visual */
static  Window     win ;     /* the window */
static  GC         g;        /* the graphics context */
static  Window     create_window(int, int, int, int) ;
static  void       set_properties(Window, int, int, int, int,
                                  char*, int, char**);
static u_long      black, white;

main(
    int                             argc,
    char                            *argv[])
{
    Xgl_object                      sys_st;
    Xgl_object                      tmap;
    Xgl_object                      tmap;
    Xgl_object                      ctx;
    Xgl_pt_list                     pl[1];
    float                           pts[20];
    Xgl_object                      mem_ras_front;
    Xgl_object                      front_tm;
```

```
  Xgl_texture_boundary            u_bound, v_bound;
  Xgl_usgn32*                     addr;
  Xgl_texture_general_desc        tm_desc;
  Xgl_render_component_desc*      render_desc;
  Xgl_usgn32                      height = 8;
  Xgl_usgn32                      width = 8;
  Xgl_usgn32                      depth = 32;
  Xgl_usgn32                      i, j;
  Xgl_usgn32*                     tmp_addr;
  Xgl_object                      ras = NULL;
  Xgl_boolean                     tmap_switches[1];
  float                           val = 45.0/180.0 * 3.14;
  char                            hit_return[20];
  Xgl_color                       edge_color;
  Xgl_object                      view_trans;


  Xgl_obj_desc                obj_desc ;
  Xgl_X_window                win_desc ;
  Xgl_matrix_f2d              mat;


if ( ( display = XOpenDisplay(NULL) ) == NULL) {
        (void) fprintf(stderr, "display no good! %X\n", display);
        exit(-1);

      screen = DefaultScreen(display);
      vis = XDefaultVisual(display, screen) ;
      cmap = DefaultColormap(display, screen) ;
      win = create_window(200,200,500,500) ;
      set_properties(win,200,200,500,500,"Texture Example",
                     argc,argv) ;

      XMapWindow(display, win);
      XSync(display,False) ;

  win_desc.X_display = display ;
  win_desc.X_screen = screen ;
  win_desc.X_window = win ;
  obj_desc.win_ras.type = XGL_WIN_X ;
  obj_desc.win_ras.desc = (void *)&win_desc ;


  /* Calling xgl_open() */
  sys_st = xgl_open(XGL_SYS_ST_ERROR_DETECTION, 1, 0);
```

```
/* Create a window raster of color type RGB */
ras = xgl_object_create(sys_st, XGL_WIN_RAS, &obj_desc,
                        XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB, 0);

/* Create a 3d context */
ctx = xgl_object_create(sys_st, XGL_3D_CTX, NULL,NULL);

/* Set the Hidden line hidden surface removal to Z Buffer */
xgl_object_set(ctx, XGL_CTX_DEVICE, ras, XGL_3D_CTX_HLHSR_MODE,
               XGL_HLHSR_ZBUFFER, NULL);

xgl_object_set(ctx, XGL_CTX_NEW_FRAME_ACTION,
   XGL_CTX_NEW_FRAME_CLEAR| XGL_CTX_NEW_FRAME_HLHSR_ACTION, 0);
xgl_context_new_frame(ctx);

/* Create a mem ras object; 8x8 32-bits deep */
mem_ras_front = xgl_object_create(sys_st, XGL_MEM_RAS, NULL,
                                  XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                   XGL_RAS_DEPTH, depth,
                                   XGL_RAS_WIDTH, width,
                                   XGL_RAS_HEIGHT, height, NULL);

/* get memory address of the raster */
xgl_object_get(mem_ras_front, XGL_MEM_RAS_MEMORY_ADDRESS,
               &addr);

/* write texture into memory */
tmp_addr = addr;

/*
 * Assume that width is a multiple of
 * 2 for this simple example
 * Write a checked board pattern into memory
 */
for (i = 0; i < height; i += 2) {
   for (j = 0; j < width; j+= 2) {
        *tmp_addr++ =  0xFFFFFFFF;
        *tmp_addr++ = 0x00000000;
    }
    for (j = 0; j < width; j+= 2) {
        *tmp_addr++ =  0x00000000;
        *tmp_addr++ = 0xFFFFFFFF;
    }
}
```

```
/* Create a front MipMap Texture object */;
front_tm = xgl_object_create(sys_st, XGL_MIPMAP_TEXTURE,
                             NULL, 0);
/* Do the filtering operation on the front mipmap */

/* Set the number of levels in front MipMap Texture object */
xgl_object_set(front_tm, XGL_MIPMAP_TEXTURE_LEVELS, 3, NULL);


/* Set the boundary conditions for generating the MIP map */
u_bound = XGL_TEXTURE_BOUNDARY_WRAP;
v_bound = XGL_TEXTURE_BOUNDARY_WRAP;

/*
 * Build  a MIP map pyramid of level 3
 * from the memory raster
 */
xgl_mipmap_texture_build(front_tm, mem_ras_front, u_bound,
                         v_bound);

/*
 * Once the texture is created,
 * mem_ras has no impact on the texture map
 * They can be modified or destroyed
 */
xgl_object_destroy(mem_ras_front);

/* Create the front Texture Map Object
 * For this example, use the attribute defaults:
 * XGL_TMAP_T0_INDEX         : 0
 * XGL_TMAP_T1_INDEX         : 1
 * XGL_TMAP_PARAM_TYPE    : XGL_TEXTURE_PARAM_EXPLICIT
 */
tmap = xgl_object_create(sys_st, XGL_TMAP,NULL, NULL);

/* Fill in the texture descriptor structure */
tm_desc.texture_type = XGL_TEXTURE_TYPE_MIPMAP;

/* Assign the front MipMap Texture Object and  Texture object */
tm_desc.texture_info.mipmap.texture_map = front_tm;

/* Max u and Max v frequency */
tm_desc.texture_info.mipmap.max_u_freq = 1.0;
tm_desc.texture_info.mipmap.max_v_freq = 1.0;
```

```
/* Boundary conditions */
tm_desc.texture_info.mipmap.u_boundary = u_bound;
tm_desc.texture_info.mipmap.v_boundary = v_bound;

/* Assign Boundary color value as Green */
tm_desc.texture_info.mipmap.boundary_values[0] = 0;
tm_desc.texture_info.mipmap.boundary_values[1] = 255;
tm_desc.texture_info.mipmap.boundary_values[2] = 0;
tm_desc.texture_info.mipmap.boundary_values[3] = 0;

/* Set the depth adjustment factor to 0 */
tm_desc.texture_info.mipmap.depth_interp_factor = 0.0;

/* Set the orientation matrix to identity */
tm_desc.texture_info.mipmap.orientation_matrix[0][0] = 1.0;
tm_desc.texture_info.mipmap.orientation_matrix[0][1] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[1][0] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[1][1] = 1.0;
tm_desc.texture_info.mipmap.orientation_matrix[2][0] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[2][1] = 0.0;

/* Sampling methods */
tm_desc.texture_info.mipmap.interp_info.filter1 =
           XGL_TEXTURE_INTERP_MIPMAP_POINT;
tm_desc.txture_info.mipmap.interp_info.filter2 =
           XGL_TEXTURE_INTERP_POINT;

/* Set the color composition information */
tm_desc.comp_info.color_info.num_render_comp_desc = 1;

render_desc =
    &tm_desc.comp_info.color_info.render_component_desc[0];

/*
 * The texture acts on the diffuse color and
 * texture_op is blend
 */
render_desc->comp       = XGL_RENDER_COMP_DIFFUSE_COLOR;
render_desc->texture_op = XGL_TEXTURE_OP_BLEND;

/* Blend colors Red and Blue */
render_desc->op.blend.rgb.c1.r = 1.0;
render_desc->op.blend.rgb.c1.g = 0.0;
render_desc->op.blend.rgb.c1.b = 0.0;

render_desc->op.blend.rgb.c2.r = 0.0;
```

```
 render_desc->op.blend.rgb.c2.g = 0.0;
 render_desc->op.blend.rgb.c2.b = 1.0;

 /* Set this texture desc in the front texture map */
 xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);

/*
 * Setup a Multi Simple Polygon
 * Pt_type should have DATA
 * Intialize num_data_values
 */
 pl[0].pt_type = XGL_PT_DATA_F3D;
 pl[0].bbox = NULL;
 pl[0].num_pts = 4;
 pl[0].num_data_values = 2;
 pl[0].pts.data_f3d = (Xgl_pt_data_f3d*)&pts[0];

/*
 * For the first point:
 * {x,y,z,u,v} = {100.0, 100.0, 0.0, 0.0, 0.0}
 */
 pts[0] = 100.0;
 pts[1] = 100.0;
 pts[2] = 0.0;
 pts[3] = 0.0;
 pts[4] = 0.0;

/*
 * For the second point:
 * {x,y,z,u,v} = {100.0, 300.0, 0.0, 0.0, 1.0}
 */
 pts[5] = 100.0;
 pts[6] = 300.0;
 pts[7] = 0.0;
 pts[8] = 0.0;
 pts[9] = 1.0;

/*
 * For the third point:
 * {x,y,z,u,v} = {300.0, 300.0, 0.0, 1.0, 1.0}
 */
 pts[10] = 300.0;
 pts[11] = 300.0;
 pts[12] = 0.0;
 pts[13] = 1.0;
 pts[14] = 1.0;
```

```
/*
 * For the fourth point:
 * {x,y,z,u,v} = {300.0, 100.0, 0.0, 1.0, 0.0}
 */
 pts[15] = 300.0;
 pts[16] = 100.0;
 pts[17] = 0.0;
 pts[18] = 1.0;
 pts[19] = 0.0;

 /* Assign Texture Map Object to the 3D context*/
 xgl_object_set(ctx,
                XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 1,
                XGL_3D_CTX_SURF_FRONT_TMAP, &tmap,
                NULL);

 /*
  * Draw the polygon without texture on it
  * by setting the texture map switch to off
  */
 dmap_texture_switches[0] = FALSE;
 xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES,
                 tmap_switches, NULL);

 /* Turn the edge on and set the edge color */
 edge_color.rgb.r = 1.0;
 edge_color.rgb.g = 1.0;
 edge_color.rgb.b = 1.0;
 xgl_object_set(ctx, XGL_CTX_EDGE_COLOR, &edge_color,
                XGL_CTX_SURF_EDGE_FLAG, TRUE, NULL);

 /* Render an untextured multi simple polygon */
 xgl_multi_simple_polygon(ctx,
       XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
        NULL, NULL,1, pl);

 xgl_context_post(ctx,TRUE);

 fprintf(stderr, "Hit Return to Continue > ");
 fscanf(stdin, "%c", hit_return);

 xgl_context_new_frame(ctx);

/*
 * Draw the polygon with texture on
```

```
* by setting the texture map switch to on
*/
dmap_texture_switches[0] = TRUE;
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES,
               tmap_switches, NULL);

/* Render a textured multi simple polygon */
xgl_multi_simple_polygon(ctx,
      XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
      NULL, NULL,1, pl);

xgl_context_post(ctx,TRUE);

fprintf(stderr, "Hit Return to Continue > ");
fscanf(stdin, "%c", hit_return);

xgl_context_new_frame(ctx);

/* Rotate the object by setting the view transformation */
xgl_object_get(ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_rotate(view_trans, val, XGL_AXIS_Z,
                     XGL_TRANS_REPLACE);
xgl_transform_rotate(view_trans, val, XGL_AXIS_X,
                     XGL_TRANS_POSTCONCAT);
xgl_transform_rotate(view_trans, val, XGL_AXIS_Y,
                     XGL_TRANS_POSTCONCAT);

/* Change the texture_op to Modulate */
render_desc->texture_op = XGL_TEXTURE_OP_MODULATE;

/* Set this texture desc in the front texture map */
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);

/* Render a textured multi simple polygon */
xgl_multi_simple_polygon(ctx,
        XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
        NULL, NULL,1, pl);

xgl_context_post(ctx,TRUE);

/* Wait for the user to hit return */
fprintf(stderr, "Hit Return to Quit > ");
fscanf(stdin, "%c", hit_return);

/* Destroy all created objects */
xgl_object_destroy(tmap);
```

```
    xgl_object_destroy(front_tm);
    xgl_object_destroy(ctx);
    xgl_object_destroy(ras);

    /* call xgl_close */
  xgl_close(sys_st);
}

static Window
create_window(
        int     x,
        int     y,
        int     w,
        int     h)
{
        XSetWindowAttributes    attributes ;

        black = BlackPixel(display, screen) ;
        white = WhitePixel(display, screen) ;

        attributes.background_pixel = black ;
        attributes.border_pixel = black ;
        attributes.colormap = cmap ;

        return XCreateWindow(display, DefaultRootWindow(display),
                x, y, w, h, 5,
                CopyFromParent, CopyFromParent, vis,
               CWBackPixel|CWBorderPixel|CWColormap, &attributes) ;
}

static void
set_properties(
        Window  win,
        int     x,
        int     y,
        int     w,
        int     h,
        char    *label,
        int     argc,
        char    **argv)
{
        XSizeHints      size_hints;
        XWMHints        hints ;

        size_hints.x = x-5 ;
        size_hints.y = y-26 ;
```

```
        size_hints.width = w ;
        size_hints.height = h ;
        size_hints.flags = PPosition|PSize ;

        XSetStandardProperties(display, win, label, label,
                NULL, argv, argc, &size_hints );

        XSelectInput(display, win, ExposureMask | KeyPressMask |
                ButtonPressMask | StructureNotifyMask );

        hints.flags = InputHint ;
        hints.input = True ;
        XSetWMHints(display,win, &hints) ;
}
```

18