Lighting, Shading, and Depth Cueing

16

This chapter discusses the XGL lighting model. It includes information on the following topics:

- Available light types
- Reflection components
- Equations that govern the XGL light reflection model
- Information on using XGL for lighting and depth cueing
- Example program of lighting in indexed color space

Introduction to the 3D Rendering Pipeline

The XGL 3D rendering pipeline draws geometric primitives with varying degrees of realism. The 3D rendering pipeline consists of several stages: *lighting*, *shading*, *depth cueing*, and *color mapping*. The application can configure XGL for all or just some of the operations, depending on the final visual result desired and the time for producing an image.

Lighting

The lighting stage is early in the rendering pipeline. It applies only to surfaces defined in 3D. The current release of XGL limits lighting to surfaces composed of polygonal facets.

16

Lighting simulates the physical interaction of light reflecting from surfaces. The color and intensity of reflected light depends on the properties of lights and surfaces, and on the orientation of a surface relative to lights and the observer. The lighting calculations in XGL are executed on each polygonal facet independent of all other facets; effects such as shadows and reflections that are achievable with ray tracing are not provided.

XGL performs lighting calculations in RGB and indexed color spaces. Some conversion of colors may be necessary, for example, if the frame buffer operates in a color space that is different from that of the lighting calculation.

Lighting calculations span the major aspects of XGL including the geometric and color characteristics of a surface primitive, the rendering attributes of a 3D Context, and the objects associated with 3D Contexts, such as Lights, Transforms, and the Device's Color Map.

Shading

Shading computations are closely linked to lighting computations. The shading method determines how the rendering parameters are interpolated across a facet and where lighting calculations are performed. Lighting may precede or follow shading. Two common shading methods are:

- Color shading, which calculates the light reflected from the vertices of a facet, and then interpolates these reflected colors across the facet.
- Normal shading, which interpolates reflectance normal vectors (specified at vertices) across the facet, and then calculates the light reflected from intermediate locations on the facet that correspond to pixel positions. (The current release of XGL does not support normal shading.)

Depth Cueing

Depth cueing follows lighting and shading. It enhances depth perception by blending colors in the image. XGL blends colors by modulating the colors calculated in the preceding stage by another specified color. The amount of blending depends on the distance of a primitive from the observer. Depth cueing applies to all primitives when it is enabled, not just surfaces.

Surface Primitives

Polygonal surface primitives consist of ordered lists of vertices that define facets. Each vertex possesses a 3D position in Model Coordinates and possibly a reflectance unit normal vector, a color, and flag information (usually for specifying edge highlighting). A surface primitive may also contain a normal vector or color, or both, for each polygonal facet. Lighting calculations use this information in different ways, depending on the desired visual appearance.

The 3D Context in Lighting

A 3D Context has attributes that control the global rendering properties of materials, such as material color (used when a surface primitive does not have a color bound to it) and the reflection coefficients for the various components of reflected light.

A 3D Context also contains objects that affect lighting. The lights themselves can be positioned and aimed to illuminate graphical objects for a desired effect. The transformation pipeline implicitly gives the position of the observer, and the Model Transform maps the geometric data to World Coordinates where lighting takes place. Finally, the Context renders into a Device with a Color Map, which determines how to interpret, calculate, and map material colors into reflected colors.

Basic Lighting and Shading Concepts

This section describes basic concepts of the XGL lighting model for 3D polygonal surfaces. It discusses the factors involved in creating graphic realism in the XGL 3D rendering pipeline.

Colors

Three types of color make up the lighting portion of the 3D rendering pipeline:

- **Material Color**: Intrinsic to the material that geometric surfaces represent. For example, a red plastic pen is intrinsically red, even in a dark room.
- **Light Color**: The color of a light source. For example, a white light with a green filter emits a narrow range of wavelengths in the green part of the spectrum.

- **Reflected Color**: The characterization of the color of light reflected from a surface. Reflected color depends on the material color and the properties of the light sources including colors, directions, and positions.

Reflection Components

In the XGL lighting model, the light reflected from a surface consists of basic components with unique properties. These are called *reflection components*. This model is partly physical and partly empirical. It follows the lighting model outlined by PHIGS PLUS, which is a library of extensions to the PHIGS standard library. This section describes reflection components without equations. "Lighting Equations" on page 408 gives a quantitative description.

- **Ambient Reflection**: The ambient reflection component adds a uniform brightness to the light reflected from surfaces, simulating light reflected from multiple surfaces. Ambient light does not come from any single direction or position but comes fom all directions with uniform intensity. Ambient reflections do not depend on the orientation or position of a surface. As the name implies, ambient light provides low-level background illumination.
- **Diffuse Reflection**: The diffuse reflection component models the effect of directional light being scattered in all directions after reflection from a rough matte-like surface. The intensity of reflected light depends on the incident angle of the incoming light rays and the reflection coefficient of the surface. The diffuse reflection falls off with the cosine of the angle between the surface normal vector and the direction vector pointing to the light source, a property known as Lambert's cosine law.
- **Specular Reflection**: The specular reflection component produces highlights or glints on glossy or shiny surfaces. This type of reflection models the law of reflection in geometrical optics, where the reflectance angle equals the incident angle. The amount of specular reflection from a point on a surface depends on the direction of the viewer from that point. If that direction closely coincides with the reflectance direction, the viewer perceives a bright spot at that point on the surface.

Light Sources

XGL has four types of light sources, each having different properties for simple to complex illumination requirements. An application can combine any number and type of light sources to produce a desired effect. For a simple illustration of XGL light sources, see Figure 1-10 on page 12.

- **Ambient Light**: An *ambient* light has only an ambient reflection component. It does not have diffuse and specular reflection components.
- **Directional Light**: A *directional* light has parallel light rays. Direct sunlight is a physical example of this type of light. A directional light contributes to diffuse and specular reflections, which in turn depend on the orientation of a surface, but not its position. A directional light does not contribute to ambient reflections.
- **Positional Light**: A *positional* light radiates light rays uniformly from a single point in all directions. (A bare incandescent light bulb radiates light in this manner.) A positional light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface. A positional light does not contribute to ambient reflections. The intensity profile of a positional light depends on the distance of the light from the surface.
- **Spot Light**: A *spot* light radiates light rays from a single point. A spot light has both a position and a direction. The ray with the maximum intensity propagates in this direction. Rays lying off this principal axis decrease in intensity with the angle away from the axis. The light from a spot light affects surfaces lying in a cone. The intensity profile of a spot light depends on the distance of the light from the surface. A spot light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface.

Illumination and Shading

Common shading techniques are known as flat, Gouraud (color), and Phong (normal). Shading can change the appearance of polygonal facets through interpolation of color values across polygons during the scan conversion stage of rendering. Color and normal shading make polygonal representations look like smooth curved surfaces when the polygons are sufficiently small.

16

XGL enables applications to select an illumination type to specify where the lighting calculations take place. Depending on the illumination type and on the lights that are *on* at that time (as specified by the attribute `XGL_3D_CTX_LIGHT_SWITCHES`), XGL will automatically calculate surface color and shading. The current release of XGL supports four illumination types:

- **None**: No illumination with no interpolation renders polygons drawn in the material or intrinsic color.
- **None with color interpolation**: No illumination calculations are done; XGL directly interpolates vertex colors across the surface.
- **Facet**: Illumination per facet results in one lighting calculation for each polygon. XGL draws the entire polygon in this reflected color. This is known as *flat shading*. Note that a light must be turned on for facet illumination to occur, and if the Raster is indexed, a color map must be available. The vertex chosen for the lighting calculation is device-dependent.
- **Vertex**: Illumination per vertex leads to calculation of the reflected light at each vertex of each polygon. XGL draws the polygon by linearly interpolating the reflected colors at the vertices across the edges and subsequently across scan lines. This is known as *color shading*. Color shading can give the appearance of smooth surfaces. However, applications must supply unit-length reflectance normal vectors at each vertex. A vertex normal points approximately in the direction of the average of the facet normals of all facets incident on the vertex. Note that a light must be turned on for vertex illumination to occur, and if the Raster is indexed, a color map must be available.

Lighting Equations

XGL performs lighting calculations in the same color space as the destination Raster's color type. As discussed in Chapter 6, "Color," XGL supports two types of color: RGB and indexed. This section describes lighting calculations with variables and equations for these two color spaces.

The lighting equations involve variables that come from various parts of XGL. They may be stored in the 3D Context object, the Context's Light objects, the Context's Device's Color Map object, or the geometry of the primitive being rendered. Table 16-1 on page 409 describes the symbols used in the lighting calculations, where for the data types:

F = floating point
G = gray color type
I = indexed color type
RGB = RGB color type

and for the sources:
[1] = Light object
[2] = calculated
[3] = explicit or derived from object geometry
[4] = vertex color, facet color, or 3D Context object
[5] = 3D Context object
[6] = calculated from the equations that follow

Table 16-1 Lighting Variables

Symbol	Description	RGB	Indexed	Source
$\hat{L}_d$	Light source direction	3×F	3×F	[1]
L_c	Light source color	RGB	G	[1]
$\hat{L}_p$	Light source position	3×F	3×F	[1]
L_e	Light source concentration exponent	F	F	[1]
C_1, C_2	Attenuation coefficients	F	F	[1]
A_s	Spread angle	F	F	[1]
L_a	Light attenuation	F	F	[6]
$\hat{O}_p$	Object position	3×F	3×F	[3]
O_d	Object diffuse color	RGB	I	[4]
O_s	Object specular color	RGB	I	[5]
O_e	Object specular exponent	F	F	[5]
K_a	Ambient reflection coefficient	F	F	[5]
K_d	Diffuse reflection coefficient	F	F	[5]
K_s	Specular reflection coefficient	F	F	[5]
$\hat{V}_e$	Unit vector from object to eye point	3×F	3×F	[6]
$\hat{V}_r$	Unit reflection vector from object	3×F	3×F	[6]

Table 16-1 Lighting Variables (Continued)

Symbol	Description	RGB	Indexed	Source
$\hat{V}_l$	Unit vector from object to light source	3×F	3×F	[2]
$\hat{V}_n$	Unit normal vector to object	3×F	3×F	[6]
C_a	Ambient contribution from light source	RGB	I	[6]
C_d	Diffuse contribution from light source	RGB	I	[6]
C_s	Specular contribution from light source	RGB	I	[6]

XGL calculates the reflected light at a point on a primitive by adding the contributions from each reflection component created by active light sources (totaling N in number) as follows:

$$C = \sum_{i=1}^{N} (C_{a_i} + C_{d_i} + C_{s_i})$$

For ambient light sources, the reflectance components are as follows:

$$C_a = K_a O_d L_c$$
$$C_d = 0$$
$$C_s = 0$$

For directional light sources, the reflectance components are as follows:

$$C_a = 0$$
$$C_d = K_d O_d L_c (\hat{V}_n \cdot -\hat{L}_d)$$
$$C_s = K_s O_s L_c (\hat{V}_e \cdot \hat{V}_r)^{O_e}$$

For positional light sources, the reflectance components are as follows:

$$C_a = 0$$
$$C_d = K_d O_d L_c (\hat{V}_n \cdot \hat{V}_l) L_a$$

$$C_s = K_s O_s L_c (\hat{V}_e \cdot \hat{V}_r)^{O_e} L_a$$

For spot light sources, the reflectance components are as follows:

$$C_a = 0$$

$$C_d = \begin{cases} K_d O_d L_c (\hat{V}_n \cdot \hat{V}_r) (-\hat{L}_d \cdot \hat{V}_l)^{L_e} L_a, & (-\hat{L}_d \cdot \hat{V}_l) \geq \cos(\frac{A_s}{2}) \\ 0 & , (-\hat{L}_d \cdot \hat{V}_l) < \cos(\frac{A_s}{2}) \end{cases}$$

$$C_s = \begin{cases} K_s O_s L_c (\hat{V}_e \cdot \hat{V}_r)^{O_e} (-\hat{L}_d \cdot \hat{V}_l)^{L_e} L_a, & (-\hat{L}_d \cdot \hat{V}_l) \geq \cos(\frac{A_s}{2}) \\ 0 & , (-\hat{L}_d \cdot \hat{V}_l) < \cos(\frac{A_s}{2}) \end{cases}$$

The light attenuation for positional and spot lights is given by:

$$L_a = 1 / (C_1 + C_2 \cdot norm(\vec{O}_p - \hat{L}_p))$$

The light reflection vector is:

$$\hat{V}_r = 2 (\hat{V}_n \cdot \hat{V}_l) \hat{V}_n - \hat{V}_l$$

XGL replaces all negative dot product values with zero.

RGB Lighting

In RGB color space, each member of a triplet of color values stored in the frame buffer is converted to a voltage that drives the individual electron guns of a cathode ray tube. Each channel is independent of the other two.

In RGB lighting, the application specifies both material and light colors in the RGB format. XGL represents the value of each channel as a floating-point number in the range 0.0 to 1.0. If the result of a lighting calculation exceeds 1.0, XGL clamps the result to 1.0. RGB lighting applies the equations in turn to each of the three channels to produce RGB reflected colors.

RGB frame buffers require more planes of video memory than indexed frame buffers, and RGB lighting requires more computation than indexed lighting; however, RGB lighting provides more realistic images with a wider range of color variation.

XGL allows rendering in RGB even if the frame buffer is indexed. Applications do this by creating an RGB Raster on an indexed frame buffer. XGL performs the lighting calculation in RGB. Applications should have a color cube defined in the Color Map. A default color cube is created for all RGB Rasters. Then XGL dithers the colors in this color cube and writes indexes to the indexed frame buffer.

Indexed Lighting

Indexed frame buffers require significantly smaller amounts of video memory than RGB frame buffers. The indexed color method provides an indirect means of generating color on a graphics device. For each pixel, an indexed frame buffer stores an index into a lookup table containing RGB values that drive the electron guns. The palette of available colors is large, but the number of colors that can be displayed at any given moment is small.

Lighting in indexed color space has limitations because of the small choice of colors at any one time. Lighting in indexed color space is also more complicated than lighting in RGB. If an application is willing to accept these limitations, however, lighting in indexed space on indexed frame buffers is faster than lighting in RGB space and dithering the colors in a color cube.

In indexed lighting, lights are always white with varying levels of brightness. Applications specify the light intensity as a color of type *gray*. Material and reflected colors are indexes. The application partitions the color table into segments or *color ramps*, as demonstrated in the example program at the end of this chapter. (A material color index must reside in a defined color ramp. After lighting, the reflected color will reside within the same ramp.)

Color ramps typically start at black at the lowest index of the segment and increase linearly to some maximum value at the top of the ramp. For example, the color at the top could be bright saturated red. From the top, the intermediate values in the ramp decrease in brightness to black at the bottom of the ramp. Color ramps need not be linear. Both saturation and brightness may be nonlinear to give effects such as white specular highlights.

XGL calculates reflected light in indexed space in the following way. The calculation involves conversion of indexes to gray values, application of the lighting equations in gray color space, and finally conversion of the reflected gray color to an index. The process is as follows:

1. From the 3D Context, get the destination Raster's Color Map.
2. Given a color index for the diffuse material color, find the color segment in the Color Map to which the index belongs. Let the segment offset within the color table be I_s. Let the segment length be N_s.
3. Convert diffuse material color index to a floating-point gray value. The specified diffuse material color $\hat{O}_d$ is an index. Let the corresponding gray value be O_d, having a range from 0.0 to 1.0. Then,

$$O_d = \frac{\hat{O}_d - I_s}{N_s - 1}$$

4. If the specular color index is in the same segment as the diffuse color index, convert the specular color index to a floating point value. The specified specular material color $\hat{O}_s$ is an index. Let the corresponding gray value be O_s, having a range from 0.0. to 1.0. Then,

$$O_s = \frac{\hat{O}_s - I_s}{N_s - 1}$$

5. Otherwise, the specular color index does not reside in the same segment as the diffuse color index. To deal with this inconsistency, let

$$O_s = 0$$

6. Perform lighting in gray color space according to the equations given on page 408. Accumulate the reflected light contributions as a gray value C, ranging from 0.0 to 1.0
7. Clamp C so that it does not exceed 1.0.
8. Convert C to a color index residing in the original color ramp:

$$\hat{C} = truncate(C\,(N_s - 1) + I_s)$$

Note that in similar situations, lighting in index color space may give different results than lighting in RGB color space does. As long as application developers understand the limitations of indexed lighting, however, useful results are attainable.

XGL Lighting Attributes

The lighting calculations are affected by attributes associated with both individual Light objects and the associated 3D Context. Table 16-2 associates the parameters that affect lighting with XGL attributes.

Table 16-2 Lighting Parameters and Lighting Attributes

Symbol	XGL Lighting Attribute
$\hat{L}_d$	XGL_LIGHT_DIRECTION
L_c	XGL_LIGHT_COLOR
$\hat{L}_p$	XGL_LIGHT_POSITION
L_e	XGL_LIGHT_SPOT_EXPONENT
C_1	XGL_LIGHT_ATTENUATION_1
C_2	XGL_LIGHT_ATTENUATION_2
A_s	XGL_LIGHT_SPOT_ANGLE
O_d	XGL_CTX_SURF_FRONT_COLOR, XGL_3D_CTX_SURF_BACK_COLOR or color stored with facet or vertex
O_s	XGL_3D_CTX_SRF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR
O_e	XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, XGL_3D_CTX_SURF_BACK_SPECULAR_POWER
K_a	XGL_3D_CTX_SURF_FRONT_AMBIANT, XGL_3D_CTX_SURF_BACK_AMBIANT

Table 16-2 Lighting Parameters and Lighting Attributes (Continued)

Symbol	XGL Lighting Attribute
K_d	`XGL_3D_CTX_SURF_FRONT_DIFFUSE`, `XGL_3D_CTX_SURF_BACK_DIFFUSE`
K_s	`XGL_3D_CTX_SURF_FRONT_SPECULAR`, `XGL_3D_CTX_SURF_BACK_SPECULAR`

Creating a Light Object

An application can create a Light object and set its attributes to give particular lighting effects. At any time, lights can be set to any of the four light source types — ambient, directional, positional, or spot. By default, the light type is ambient. Note, however, that no Light objects are created at XGL initialization. In order to use lights, you must define them in one of two ways:

- You can creat a Light object and specify lighting attributes with `xgl_object_create()` used with the *type* parameter set to `XGL_LIGHT`. Light objects created in this way can be shared among Contexts.
- You can manipulate the array of lights defined in the 3D Context. When an application sets the number of Lights in a 3D Context, XGL creates or destroys Lights to adjust the Context's array of Lights to the specified number. XGL also adjusts the array of on/off light switches to match the number of Lights.

To define Light objects using the Context's light array, follow these steps:

1. Declare local variables for an array for the number of lights and an array for the light switches.
2. Set the 3D Context attribute `XGL_3D_CTX_LIGHT_NUM` to the number of lights.

   When this attribute is set, XGL creates new Light objects to match this attribute and changes the array of Light switches to a corresponding number. The light switches are set to `TRUE` by default. If the new number of lights exceeds the current number, XGL creates new Lights and switches and appends them to the current arrays; in this case, the Lights and switches that already existed remain unchanged. If the new number of lights is less that the current number, XGL destroys Lights and switches so that the number of lights and switches matches the attribute number.

3. Retrieve the list of lights and light switches from the Context.

   Alternatively, you can create an array of Lights using `xgl_object_create()` and attach the array to the Context.

4. Set attributes as needed. Note that the first light you define must be an ambient light.

The following code fragment illustrates the array of light objects. For additional examples, see the sample programs later in this chapter.

```
Xgl_light    light[2];
Xgl_boolean  light_switch[2];
Xgl_color    light_color;

/* Set the number of lights */
xgl_object_set (ctx, XGL_3D_CTX_LIGHT_NUM, 2,
                NULL);

/* Get lights and switches */
xgl_object_get(ctx, XGL_3D_CTX_LIGHTS, light);
xgl_object_get(ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

/* Set  the values for Light 0; must be type ambient */
xgl_object_set(light[0],
                XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
                XGL_LIGHT_COLOR, &light_color,
                NULL);

/* Set the values for Light 1 */
pt_f3d.x = -1.5;
pt_f3d.y = -1.0;
pt_f3d.z = 0.4;
xgl_object_set(light[1],
                XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                XGL_LIGHT_COLOR, &light_color,
                XGL_LIGHT_DIRECTION, &pt_f3d,
                NULL);

/* Turn Light 1 off */
light_switch[1] = FALSE;
xgl_object_set (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch,
                 NULL);
```

16

Light Operators and Attributes

Light Operator

An application can copy a Light object to another Light object. The `xgl_light_copy()` operator makes a copy of a Light object by transferring all state information from a Source Light to a Destination Light. This operator is defined as:

```
void xgl_light_copy (Xgl_light dest_light,Xgl_light src_light)
```

Light Attributes

Most Light attributes in the following list were introduced in the previous section as mathematical symbols used in the lighting equations. More information is available about each of these attributes in the *XGL Reference Manual*.

Table 16-3 Light Object Attributes

Attribute	Description
XGL_LIGHT_TYPE	Defines the type of a Light object. The type can be XGL_LIGHT_AMBIENT, XGL_LIGHT_SPOT, XGL_LIGHT_DIRECTIONAL, or XGL_LIGHT_POSITIONAL.
XGL_LIGHT_POSITION	Defines the position of a light.
XGL_LIGHT_DIRECTION	Defines the direction of a light.
XGL_LIGHT_COLOR	Defines the color of a light.
XGL_LIGHT_SPOT_ANGLE	Defines the illumination angle of influence for a spot light.
XGL_LIGHT_SPOT_EXPONENT	Defines the light source concentration exponent for a spot light.
XGL_LIGHT_ATTENUATION_1 XGL_LIGHT_ATTENUATION_2	Define the light attenuation coefficients of a Light object. These attributes apply to positional lights or spot lights.

16

3D Context Attributes for Lighting

Several 3D Context surface attributes supply the values for the different variables of the lighting equations. These attributes define the surface appearance of 3D objects. Some were introduced previously as mathematical symbols used in the lighting equations. Information is available on the use of each of these attributes in the *XGL Reference Manual*. Table 16-4 provides an overview of the Context surface rendering attributes.

Table 16-4 Context Light Attributes

Attribute	Description
XGL_CTX_FRONT_COLOR XGL_3D_CTX_SURF_BACK_COLOR	Define the color used to fill surfaces.
XGL_CTX_SURF_FRONT_COLOR_SELECTOR XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR	Select the source of a surface's color from the Context or from facet or vertex data provided with the primitive.
XGL_3D_CTX_SURF_FRONT_AMBIENT XGL_3D_CTX_SURF_BACK_AMBIENT	Define the ambient reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_DIFFUSE XGL_3D_CTX_SURF_BACK_DIFFUSE	Define the diffuse reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_ILLUMINATION XGL_3D_CTX_SURF_BACK_ILLUMINATION	Specify how illumination calculations are performed.
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT	Determine which components of the lighting equation are used to compute a surface color.
XGL_3D_CTX_SURF_FRONT_SPECULAR XGL_3D_CTX_SURF_BACK_SPECULAR	Define the specular coefficient used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR	Define the specular color used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER XGL_3D_CTX_SURF_BACK_SPECULAR_POWER	Defines the specular power used in computing lighting.
XGL_3D_CTX_SURF_FACE_CULL	Controls face culling in a 3D Context.
XGL_3D_CTX_SURF_FACE_DISTINGUISH	Controls whether front- and back-facing surfaces are distinguished from each other.
XGL_3D_CTX_SURF_GEOM_NORMAL	Controls how surface normals are calculated in a Context if the surface normals are not provided as part of the application data.

Table 16-4 Context Light Attributes (Continued)

Attribute	Description
XGL_3D_CTX_SURF_NORMAL_FLIP	Specifies whether vertex and facet normals are flipped.
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP	Defines how surface normals are treated for lighting.
XGL_3D_CTX_LIGHT_SWITCHES	Defines the on and off switches that correspond to the 3D Context's array of lights.
XGL_3D_CTX_LIGHTS	Defines the array of lights in a 3D Context.
XGL_3D_CTX_LIGHT_NUM	Sets the number of lights in a 3D Context.

Depth Cueing

As discussed earlier in this chapter depth cueing is the stage of the rendering pipeline that conceptually follows lighting and shading. Depth cueing assists observers with perception of depth by fading the usual colors by an amount that depends on the distance of a primitive from the observer. The amount of fading increases as the depth increases.

Unlike lighting and shading, depth cueing applies to all primitives when it is enabled, not just surface primitives.

Applications

Applications that use depth cueing often combine this effect with motion through animation. Two common applications of depth cueing are molecular modeling and flight simulation. In the molecular example, the background color is often black. Depth perception can be enhanced by fading the more distant parts of the molecule to black, making them appear dimmer. Oscillation of the molecule about a vertical axis enhances depth perception.

Linear Depth Cueing

The simplest method of depth cueing is to blend the usual color of a point on a primitive with a specified depth cue color. This color is usually the same as the color of the background.

Like lighting, depth cueing is simpler in RGB color space than in indexed color space. The colors of primitives and the depth cue color are in the RGB format. Let C be the usual color of a point on a primitive. Let C_{dc} be the depth cue color. Let z be the normalized depth, which equals 0 at the front of the VDC viewport and 1 at the back. The depth cued color $\tilde{C}$ is a weighted average between the usual color and the depth cue color:

$$\tilde{C} = (1 - z)\, C + z C_{dc}$$

XGL applies this equation in turn to each of the three channels (red, green, and blue).

On indexed Rasters, depth cueing has limitations, just as lighting does. Indexed depth cueing takes place within color ramps. XGL finds the color ramp in which the undepth-cued color index resides and blends this with the color index at the bottom of the color ramp. Let I be the usual color index on a point of a primitive. Let I_0 be the color index at the bottom of the color ramp in which I resides. Let z be the normalized depth, as in the RGB case above. The depth-cued color is the weighted-average:

$$\tilde{I} = (1 - z)\, I + z I_0$$

In linear depth cueing, the colors of primitives are the same as the "undepth-cued" colors at the front of the VDC viewport. At the back of the VDC viewport, the colors completely change to the specified depth cue color (RGB) or the color at the bottom of the color ramp (indexed). Between these boundaries, all colors modulate to a linear combination of the original color and the depth cue color.

Scaled Depth Cueing

Scaled depth cueing is a superset of linear depth cueing. Scaled depth cueing is linear between two reference planes, which can be placed at arbitrary values of depth in VDC. The scale factors at the front and back planes can be specified as values in the range [0.0, 1.0]. The scale factors outside of the linear region are clamped at the values specified at the reference planes.

The equations for scaled depth cueing are as follows:

$$(C' = S_f C + (1 - S_f)\, C_d) \quad Z \le Z_f$$

$$(C' = S_b C + (1 - S_b)\, C_d) \quad Z \ge Z_b$$

$$(C' = \lambda C + (1 - \lambda)\, C_d) \quad Z_f \le Z \le Z_b$$

$$\lambda = S_b + (S_f - S_b)\left(\frac{Z - Z_b}{Z_f - Z_b}\right)$$

where C' is the depth-cued color, C is the original color, and C_d is the depth-cue color. These equations for illustrative purposes assume Z increases from front to back. Figure 16-1 graphs the scaling factor in scaled depth cueing.

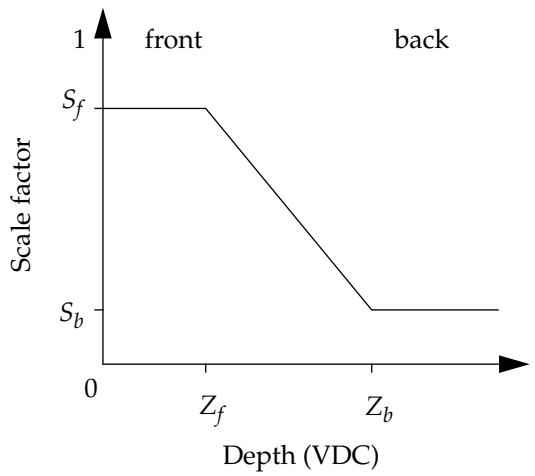

Figure 16-1 Scaling Factor Applied in Scaled Depth Cueing

Depth Cueing Attributes

`XGL_3D_CTX_DEPTH_CUE_MODE`
Selects one of three depth cue modes: `XGL_DEPTH_CUE_LINEAR`, which applies a linear depth cueing function; `XGL_DEPTH_CUE_SCALED`, which applies a linear depth cueing function within the limits of two reference planes; and `XGL_DEPTH_CUE_OFF`. The default value is `XGL_DEPTH_CUE_OFF`.

XGL_3D_CTX_DEPTH_CUE_INTERP
Specifies whether depth cue interpolation is calculated along the depth axis. If the value is set to TRUE, linear interpolation is performed along the primitive for each pixel rendered, using the color defined at each vertex of the primitive by the attribute XGL_3D_CTX_DEPTH_CUE_MODE. If the value is set to FALSE, depth cue color is calculated at each vertex, but no interpolation is done at rendering time. The attribute is assumed to be TRUE if either of the following conditions is true:

- Polylines are being rendered, and XGL_3D_CTX_LINE_COLOR_INTERP is TRUE.
- Surfaces are being rendered, and the attribute controlling illumination on the surface (either XGL_3D_CTX_SURF_BACK_ILLUMINATION or XGL_3D_CTX_SURF_FRONT_ILLUMINATION) is set to XGL_ILLUM_NONE_INTERP_COLOR or XGL_ILLUM_PER_VERTEX.

In other words, if per-pixel interpolated shading has already been requested by the setting of other Context attributes, the XGL_3D_CTX_DEPTH_CUE_INTERP attribute has no effect on rendering. The default value is TRUE.

XGL_3D_CTX_DEPTH_CUE_REF_PLANES
Defines two planes normal to the *z*-axis in VDC that are used when the depth cue mode is set to XGL_DEPTH_CUE_SCALED. In this case, the depth cueing interpolation is performed only on that part of the image that lies between the two depth cue reference planes. The default value is (0.0, 1.0).

XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS
Defines the portion of a primitive's color that is mixed with the depth cue color when the depth cue mode is set to XGL_DEPTH_CUE_SCALED. Two scale factors are used, one for the part of the primitive in front of the frontmost depth cue reference plane, and one for the part behind the rear-most plane. The default value is (1.0, 0.0).

XGL_3D_CTX_DEPTH_CUE_COLOR
Specifies the color that depth-cued primitives will tend toward for increasing values of Z. The color type of this attribute must match the color type of the Raster attached to the Context. The default value is index 0 or RGB (0, 0, 0).

Lighting Examples

The following examples illustrate lighting and shading for an indexed Raster. The main procedure is `light_main.c` in Appendix B. It initializes the data structures that define an icosahedron suitable for the `xgl_multi_simple_polygon()` primitive. It also initializes some XGL resources and the window system interface using the standard example utility file `ex_utils.c`. To compile the light example programs, type `make light`. The complete program includes `ex_utils.c`, `light_main.c`, `light_facet.c`, and `light_vertex.c`; it allows you to view a multi-sided polygon with facet lighting and vertex lighting.

Light Facet Example

This example illustrates illumination on a per-facet basis, also known as flat shading. It sets up two color ramps, initializes the View Transform, defines and turns on three lights, and draws the icosahedron. The output is shown in Plate 5a. Note that the first light that the program defines is an ambient light.

Code Example 16-1 Facet Lighting Example

```
/*
 * light_facet.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>
#include <math.h>

#include "ex.h"
#include "light.h"

extern Xgl_pt_list   pl[NUM_FACETS];
extern Xgl_color_typeex_color_type;

void
light_facet (Xgl_object    ctx)
{
    Xgl_usgn32          i;
    Xgl_object          ras;
    Xgl_cmap            cmap;
    Xgl_color_list      clist;
    Xgl_color           ctable[CTABLE_NUM];
    Xgl_segment         segment[2];
```

```
Xgl_pt              pt;
Xgl_bounds_d3d      vdc_window;
Xgl_pt_f3d          pt_f3d;
Xgl_pt_d3d          pt_d3d;
Xgl_object          trans;
Xgl_object          view_trans;
Xgl_color           cyan;
Xgl_light           light[4];
Xgl_boolean         light_switch[4];
Xgl_color           light_color;

if (ex_color_type == XGL_COLOR_INDEX) {
    /* Get the context's raster, and its color map */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);
    xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

    /* Destroy the old color map, as it will be replaced */
    xgl_object_destroy (cmap);

    /* Set up a color map with two color ramps */

    /* Gray ramp */
    for (i = 0; i < GRAY_LENGTH; i++) {
   ctable[i + GRAY_OFFSET].rgb.r =
   ctable[i + GRAY_OFFSET].rgb.g =
   ctable[i + GRAY_OFFSET].rgb.b =
       (float) i / (float) (GRAY_LENGTH - 1);
    }

    /* Cyan ramp */
    for (i = 0; i < CYAN_LENGTH; i++) {
   ctable[i + CYAN_OFFSET].rgb.r = 0.0;
   ctable[i + CYAN_OFFSET].rgb.g =
   ctable[i + CYAN_OFFSET].rgb.b =
  (float) i / (float) (CYAN_LENGTH - 1);
    }

    /* Color-list data */
    clist.start_index = 0;
    clist.length = CTABLE_NUM;
    clist.colors = ctable;

    /* Color-ramp data */
    segment[GRAY_SEG].offset = GRAY_OFFSET;
    segment[GRAY_SEG].length = GRAY_LENGTH;
    segment[CYAN_SEG].offset = CYAN_OFFSET;
```

```
    segment[CYAN_SEG].length = CYAN_LENGTH;

    /* Create the new color map with the gray and cyan ramps */
   cmap = xgl_object_create (sys_st, XGL_CMAP, NULL,
            XGL_CMAP_COLOR_TABLE_SIZE, CTABLE_NUM,
            XGL_CMAP_COLOR_TABLE, &clist,
            XGL_CMAP_RAMP_NUM, 2,
            XGL_CMAP_RAMP_LIST, segment,
            NULL);

    /* Associate the new color map with the raster */
    xgl_object_set (ras, XGL_RAS_COLOR_MAP, cmap, NULL);
}

/* Clear window since color map size has increased */
xgl_context_new_frame (ctx);

/* Create a transform */
trans = xgl_object_create (sys_st, XGL_TRANS, NULL, NULL);

/* Set up view transform */
pt.pt_type = XGL_PT_F3D;
pt.pt.f3d = &pt_f3d;
pt_f3d.x = 0.0;
pt_f3d.y = -11.0;
pt_f3d.z = -3.0;
xgl_transform_translate (trans, &pt, XGL_TRANS_REPLACE);
xgl_transform_rotate (trans, -PI / 2, XGL_AXIS_X,
        XGL_TRANS_POSTCONCAT);
xgl_transform_rotate (trans, atan (pt_f3d.x / pt_f3d.y) + PI,
        XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);
xgl_transform_rotate (trans,
        atan (-pt_f3d.z /
        sqrt (pt_f3d.x * pt_f3d.x +
        pt_f3d.y * pt_f3d.y)),
        XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

/* Set context's view transform */
xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_copy (view_trans, trans);

/* Set up VDC window */
vdc_window.xmin = -1.25;
vdc_window.xmax = 1.25;
vdc_window.ymin = -1.25;
vdc_window.ymax = 1.25;
```

```
vdc_window.zmin = -13.0;
vdc_window.zmax = -9.0;

if (ex_color_type == XGL_COLOR_INDEX) {
    /* Set up material color of polyhedron */
    cyan.index = CYAN_OFFSET + CYAN_LENGTH - 1;
}
else {
    cyan = cyan_color;
}

/* Set context for viewing facet-illuminated polyhedron */
xgl_object_set (ctx,
       XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
       XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
       XGL_CTX_VDC_WINDOW, &vdc_window,
       XGL_3D_CTX_SURF_FACE_CULL, XGL_CULL_BACK,
       XGL_CTX_SURF_FRONT_COLOR, &cyan,
       XGL_3D_CTX_SURF_FRONT_ILLUMINATION,
       XGL_ILLUM_PER_FACET,
       XGL_3D_CTX_SURF_FRONT_AMBIENT, 0.3,
       XGL_3D_CTX_SURF_FRONT_DIFFUSE, 0.7,
       XGL_3D_CTX_SURF_FRONT_SPECULAR, 0.3,
       XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &cyan,
       XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, 5.0,
       XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
       XGL_LIGHT_ENABLE_COMP_AMBIENT |
       XGL_LIGHT_ENABLE_COMP_DIFFUSE |
       XGL_LIGHT_ENABLE_COMP_SPECULAR,
       XGL_3D_CTX_LIGHT_NUM, 4,
       NULL);

/* Get lights and switches */
xgl_object_get (ctx, XGL_3D_CTX_LIGHTS, light);
xgl_object_get (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

/* Light 0: ambient */
if (ex_color_type == XGL_COLOR_INDEX)
    light_color.gray = 0.95;
else
    light_color.rgb.r = light_color.rgb.g =
          light_color.rgb.b = 0.95;
xgl_object_set (light[0],
       XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
       XGL_LIGHT_COLOR, &light_color,
       NULL);
```

```
    /* Light 1: directional */
    if (ex_color_type == XGL_COLOR_INDEX)
        light_color.gray = 0.85;
    else
        light_color.rgb.r = light_color.rgb.g =
              light_color.rgb.b = 0.85;
    pt_f3d.x = -1.5;
    pt_f3d.y = -1.0;
    pt_f3d.z = 0.4;
    xgl_object_set (light[1],
           XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_DIRECTION, &pt_f3d,
           NULL);

    /* Light 2: positional */
    if (ex_color_type == XGL_COLOR_INDEX)
        light_color.gray = 0.7;
    else
        light_color.rgb.r = light_color.rgb.g =
              light_color.rgb.b = 0.7;
    pt_d3d.x = -5.0;
    pt_d3d.y = 3.0;
    pt_d3d.z = 1.0;
    xgl_object_set (light[2],
           XGL_LIGHT_TYPE, XGL_LIGHT_POSITIONAL,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_POSITION, &pt_d3d,
           NULL);

    /* Turn Light 3 off; leave others on */
    light_switch[3] = FALSE;
    xgl_object_set (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch,
           NULL);

    /* Draw facet-illuminated polyhedron */
    xgl_multi_simple_polygon (ctx,
                XGL_FACET_FLAG_SIDES_ARE_3 |
                XGL_FACET_FLAG_SHAPE_CONVEX,
                NULL, NULL, NUM_FACETS, pl);

    /* Clean up */
    xgl_object_destroy (trans);
}
```

16

Light Vertex Example

This example illustrates lighting on a per-vertex basis. It uses the color ramps, View Transform, and Lights defined in the previous example. In addition, it defines one additional Light, turns it on, and redraws the icosahedron with the other color ramp and vertex illumination. The output is shown in Plate 5b.

Code Example 16-2 Vertex Lighting Example

```
/*
 * light_vertex.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "light.h"

extern Xgl_pt_list  pl[NUM_FACETS];
extern Xgl_color_typeex_color_type;

void
light_vertex (Xgl_object   ctx)
{
    Xgl_color            gray;
    Xgl_object           light[4];
    Xgl_boolean          light_switch[4];
    Xgl_color            light_color;
    Xgl_pt_f3d           pt_f3d;
    Xgl_pt_d3d           pt_d3d;

    /* Get the list of lights and switches */
    xgl_object_get (ctx, XGL_3D_CTX_LIGHTS, light);
    xgl_object_get (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

    /* Light 3: spot */
    if (ex_color_type == XGL_COLOR_INDEX)
       light_color.gray = 0.2;
    else
       light_color.rgb.r =
       light_color.rgb.g =
       light_color.rgb.b = 0.2;
    pt_d3d.x = 1.0;
    pt_d3d.y = 15.0;
```

```
    pt_d3d.z = 2.0;
    xgl_object_set (light[3],
           XGL_LIGHT_TYPE, XGL_LIGHT_SPOT,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_POSITION, &pt_d3d,
           NULL);
    pt_f3d.x = -1.0;
    pt_f3d.y = -15.0;
    pt_f3d.z = -2.0;
    xgl_object_set (light[3],
           XGL_LIGHT_DIRECTION, &pt_f3d,
           XGL_LIGHT_SPOT_ANGLE, PI / 20,
           XGL_LIGHT_SPOT_EXPONENT, 4.0,
           NULL);

    /* Turn Light 3 on */
    light_switch[3] = TRUE;
    xgl_object_set (ctx,
   XGL_3D_CTX_LIGHT_SWITCHES, light_switch, NULL);

    /* Set up context for Gouraud-shaded polygons */
    if (ex_color_type == XGL_COLOR_INDEX)
        gray.index = GRAY_OFFSET + GRAY_LENGTH - 1;
    else
        gray.rgb.r = gray.rgb.g = gray.rgb.b = 1.0;
    xgl_object_set (ctx,
          XGL_3D_CTX_SURF_FRONT_ILLUMINATION, XGL_ILLUM_PER_VERTEX,
           XGL_CTX_SURF_FRONT_COLOR, &gray,
           XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &gray,
           NULL);

/* Draw polyhedron with vertex-illuminated,Gouraud-shaded facets */
    xgl_multi_simple_polygon (ctx,
                XGL_FACET_FLAG_SIDES_ARE_3 |
               XGL_FACET_FLAG_SHAPE_CONVEX,
                NULL, NULL, NUM_FACETS, pl);
}
```

16

Linear Depth Cueing Example

This example illustrates linear depth cueing. This type of depth cueing changes the colors of primitives from the front to the back of the VDC view space linearly. To compile this program, type `make dcue` in the example programs directory. The complete program includes the `ex_utils.c` utility file listed in Appendix B, `dcue_main.c` listed in Appendix B, and the linear and scaled depth cueing example programs listed in this chapter.

Code Example 16-3 Linear Depth Cueing Example

```
/*
 * dcue_linear.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "dcue.h"

void
dcue_linear (Xgl_object   ctx)
{
    Xgl_object          ras;   /* raster associated with ctx */
    Xgl_object          cmap;  /* color map associated ras */
    Xgl_color           ln_color;      /* vector color */
    Xgl_object          view_trans;

    /*
     * get raster object from context
     */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

    if (ex_color_type == XGL_COLOR_INDEX) {

   /*
   * get color map object from raster
   */
   xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

   /*
    * if color map associated with raster isn't rampcmap then
    * put it into the raster
    */
```

```
if (cmap != rampcmap) {

    if (!rampcmap) {
    Xgl_sgn32          i;
    Xgl_color_list     clist;      /* color table list */
    Xgl_color          colors[RAMP_SIZE];/* list of colors */
    Xgl_segment        segment;          /* color ramp segment */

    /*
     * create color list
     */
    clist.start_index = 0;
    clist.length      = RAMP_SIZE;
    clist.colors      = colors;

    /*
     * gray ramp for depth cued vectors
     */
    for (i = 0; i < RAMP_SIZE; i++) {
        colors[i].rgb.r =
        colors[i].rgb.g =
       colors[i].rgb.b = ((float) i) / ((float)(RAMP_SIZE - 1));
    }

    /*
     * set up segment data
     */
    segment.offset = 0;
    segment.length = RAMP_SIZE;

    /*
     * create and set attributes for ramp color map
     */
    rampcmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
                XGL_CMAP_COLOR_TABLE_SIZE, RAMP_SIZE,
                XGL_CMAP_COLOR_TABLE, &clist,
                XGL_CMAP_RAMP_NUM, 1,
                XGL_CMAP_RAMP_LIST, &segment,
                NULL);
    }
     /*
      * set color map in raster
      */
     xgl_object_set (ras, XGL_RAS_COLOR_MAP, rampcmap, NULL);
    }
 } else {/* RGB raster */
```

```
            printf("no ramps on RGB rasters\n");
        }

        /*
         * modify the viewing transform so we can see the cube
         */
        xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
        xgl_transform_rotate (view_trans, 0.3,
                              XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);
        xgl_transform_rotate (view_trans, 0.3,
                              XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);
        xgl_transform_rotate (view_trans, 0.3,
                              XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

        /*
         * render a depth cued multipolyline
         */
       xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
        xgl_context_new_frame (ctx);
       xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

        xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                        XGL_DEPTH_CUE_LINEAR, NULL);

        if (ex_color_type == XGL_COLOR_INDEX)
           /* set color to highest value in ramp */
           ln_color.index = RAMP_SIZE - 1;
        else
           ln_color = white_color;

        xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);

        xgl_multipolyline (ctx, NULL, 1, &pl_3d);

        /* display color table */
        color_show (ras, -1);

        /* restore viewing transformation */
        xgl_transform_identity (view_trans);

        /* turn off depth cueing */
        xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                        XGL_DEPTH_CUE_OFF, NULL);
}
```

Scaled Depth Cueing Example

This example illustrates scaled depth cueing. This type of depth cueing is controlled by the application and occurs only between the two reference planes that are specified. It attenuates the colors of the primitives only to the amount specified.

Code Example 16-4 Scaled Depth Cueing Example

```
/*
 * dcue_scaled.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "dcue.h"

void
dcue_scaled (Xgl_object  ctx)
{
    Xgl_object     ras;  /* raster associated with ctx */
    Xgl_object     cmap; /* color map associated with ras */
    Xgl_color      ln_color;/* vector color */
    Xgl_object     view_trans;

    Xgl_bounds_d3d vdc_window;     /* VDC window coords */
    double         saved_zmn, saved_zmx;   /* saved VDC coords */
    double         ref_planes[2];  /* scaled dcue reference planes */
    float          scale_factors[2]; /* scaled dcue scaling factors */

    /*
     * get raster object from context
     */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

    if (ex_color_type == XGL_COLOR_INDEX) {

    /*
     * get color map object from raster
     */
    xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

    /*
```

```
 * if color map associated with raster isn't rampcmap then
 * put it into the raster
 */
if (cmap != rampcmap) {

   if (!rampcmap) {
   Xgl_sgn32          i;
   Xgl_color_list     clist;      /* color table list */
   Xgl_color          colors[RAMP_SIZE];/* list of colors */
   Xgl_segment        segment;    /* color ramp segment */

   /*
    * create color list
    */
   clist.start_index = 0;
   clist.length      = RAMP_SIZE;
   clist.colors      = colors;

   /*
    * gray ramp for depth cued vectors
    */
   for (i = 0; i < RAMP_SIZE; i++) {
       colors[i].rgb.r =
       colors[i].rgb.g =
      colors[i].rgb.b = ((float) i) / ((float)(RAMP_SIZE - 1));
   }

   /*
    * setup segment data
    */
   segment.offset = 0;
   segment.length = RAMP_SIZE;

   /*
    * create and set attributes for ramp color map
    */
   rampcmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
               XGL_CMAP_COLOR_TABLE_SIZE, RAMP_SIZE,
               XGL_CMAP_COLOR_TABLE, &clist,
               XGL_CMAP_RAMP_NUM, 1,
               XGL_CMAP_RAMP_LIST, &segment,
               NULL);
   }

    /*
     * set color map in raster
```

```
        */
       xgl_object_set (ras, XGL_RAS_COLOR_MAP, rampcmap, NULL);
     }
   } else {/* RGB raster */
         printf("no ramps on RGB rasters\n");
   }

   /*
    * modify the viewing transform so we can see the cube
    */
   xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

   xgl_transform_rotate (view_trans, 0.3,
           XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);

   xgl_transform_rotate (view_trans, 0.3,
           XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

   xgl_transform_rotate (view_trans, 0.3,
           XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

   /*
    * render a depth cued multipolyline
    */
  xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
   xgl_context_new_frame (ctx);
  xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

   /* get VDC window and set VDC z values */
   xgl_object_get (ctx, XGL_CTX_VDC_WINDOW, &vdc_window);
   GET_VDC_Z(saved_zmn, saved_zmx)
   SET_VDC_Z(-1.0, 1.0)

 /* set reference planes and scale factors for scaled depth cueing
*/
   ref_planes[0] = 0.1;
   ref_planes[1] = 0.3;
   scale_factors[0] = 0.3;
   scale_factors[1] = 1.0;
   xgl_object_set (ctx,
               XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, scale_factors,
                  XGL_3D_CTX_DEPTH_CUE_REF_PLANES, ref_planes,
                  XGL_CTX_VDC_WINDOW, &vdc_window,
                  NULL);
```

```
    /* turn on scaled depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                    XGL_DEPTH_CUE_SCALED, NULL);

    if (ex_color_type == XGL_COLOR_INDEX)
       /* set color to highest value in ramp */
       ln_color.index = RAMP_SIZE - 1;
    else
       ln_color = white_color;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);

    /* draw cube */
    xgl_multipolyline (ctx, NULL, 1, &pl_3d);

    /* display color table */
    color_show (ras, -1);

    /* restore VDC window */
    SET_VDC_Z(saved_zmn, saved_zmx)
    xgl_object_set (ctx, XGL_CTX_VDC_WINDOW, &vdc_window, NULL);

    /* restore viewing transformation */
    xgl_transform_identity (view_trans);

    /* turn off depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                    XGL_DEPTH_CUE_OFF, NULL);
}
```