Devices 5

This chapter discusses the Device objects. It includes information on the following topics:

- Functionality and characteristics of the Device object.
- Examples of XGL programs using Xlib, XView, and OLIT.
- Determining hardware acceleration features from within an application program.

Introduction to Device Objects

XGL defines a *device* as any surface on which it can render (create) an image, whether the image is visible or invisible. The Device class is an abstraction of many types of graphics display devices. It has two subclasses: *Raster*, which represents a two-dimensional rectangular array of discrete image samples (pixels), and *Stream*, which represents any picture representation, such as a Computer Graphics Metafile (CGM), that doesn't rely on discrete pixels to display the image.

The pixels in a raster can reside in a graphics frame buffer (that is, on the display device) or in memory. Thus, there are two types of Raster objects: *Memory Rasters*, which correspond to contiguous areas of memory that XGL will write into, and *Window Rasters*, which are areas on the display screen, managed by the window system, that XGL will write into.

5

A Window Raster requires a window system in which to run; it does not work with a raw screen. The actual display device containing the pixels is generally a window canvas created by the X11 server running as part of the window system. The application must execute the proper Xlib or X toolkit calls to create this canvas before attaching an XGL Window Raster Device. A window should have only one associated Window Raster Device object attached to it.

A Memory Raster is a block of memory allocated from main memory. XGL draws into it the same way it draws into Window Rasters. The pixels composing a Memory Raster Device are in the application's memory space.

XGL provides one Stream device at this time, a CGM device. A CGM device outputs CGM formatted graphics information to the output stream.

The graphics hardware devices running applications may differ considerably. Because XGL Device objects provide the framework for interacting with graphics devices in an abstract, device-independent manner, in most cases the application programmer does not need to know the hardware specifics of each system.

Creating Device Objects

Device objects are created with the `xgl_object_create()` operator using the input values in Table 5-1:

Table 5-1 Device Object Types

Device Type	***type* Parameter**	***desc* Parameter**
Window Raster	`XGL_WIN_RAS`	A pointer to an *Xgl_obj_desc* structure containing a window type and a window descriptor.
Memory Raster	`XGL_MEM_RAS`	`NULL`
Stream	`XGL_STREAM`	A pointer to an *Xgl_obj_desc* structure containing a pointer to the name of the library for the stream device and a pointer to any device-specific information for the stream device.
CGM	`XGL_CGM_DEV`	A pointer to an *Xgl_obj_desc* structure containing a pointer to the name of the library for the XGL-provided CGM Stream device.

Window Raster Device Object

To create a Window Raster Device object, XGL needs information on the X11 window that XGL will render to. This information is provided in the *desc* parameter, which points to a *Xgl_obj_desc* union containing a window type and a window descriptor structure. The only supported value for the window raster type is `XGL_WIN_X`. The window descriptor structure is:

```
typedef struct {
    void        *X_display;
    int          X_screen;
    Xgl_usgn32   X_window;
} Xgl_X_window;
```

The window system type can be modified to indicate which protocol XGL uses to communicate with the graphics device. XGL supports three protocols for the X window system: Xlib, PEX, and DGA (Direct Graphics Access). The protocol is selected by OR'ing in the name of the protocol with the constant `XGL_WIN_X`. The constants for the names of the protocol are:

```
#define XGL_WIN_X_PROTO_DEFAULT          (0x000)
#define XGL_WIN_X_PROTO_XLIB             (0x100)
#define XGL_WIN_X_PROTO_PEX              (0x200)
#define XGL_WIN_X_PROTO_DGA              (0x400)
```

Only one protocol can be specified. If a protocol is specified, XGL tries to use that protocol to communicate with the graphics device. If it is not available, an error is issued and the Window Raster is not created. If no protocol is specified (or `XGL_WIN_X_PROTO_DEFAULT` is used), XGL tries to use the best protocol for the particular device. If that is not available, XGL will try a backup protocol. In practical terms, XGL tries to use DGA, but if that is not available, it will attempt to use PEX, and if that fails, then it will use Xlib.

In most cases, an XGL application does not need to be concerned with selecting a protocol. However, this facility is available to aid applications that want to make the best use of the graphics resources, and this usually means part of the application that is device-dependent.

The code fragment below shows how a Window Raster object can be created in XView.

```
Xgl_win_ras       win_ras = NULL;/* Xgl window raster */
Xgl_X_window      xgl_x_win;     /* XGL-X data structure */
Xgl_obj_desc      win_desc;      /* XGL window raster structure */

pw = (Xv_Window) canvas_paint_window(canvas);
canvas_window = (Window) xv_get(pw, XV_XID);
frame_window = (Window) xv_get(frame, XV_XID);

/* put X stuff into XGL data structure */
xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW(pw);
xgl_x_win.X_window = (Xgl_usgn32) canvas_window;
xgl_x_win.X_screen = (int) DefaultScreen(display);

/* create Window Raster Device using XView canvas */
win_desc.win_ras.type = XGL_WIN_X;
win_desc.win_ras.desc = &xgl_x_win;
win_ras = xgl_object_create(sys_st, XGL_WIN_RAS, &win_desc,
                            NULL);
```

A Device object must be associated with a Context object for rendering to occur. The two objects are associated using the `XGL_CTX_DEVICE` attribute. This association can be made when the Context object is created, as in the following code fragment.

```
ctx = xgl_object_create(sys_st, XGL_2D_CTX, NULL,
                  XGL_CTX_DEVICE, win_ras,
                  NULL);
```

The association between the Device and the Context can also be set with the `xgl_object_set()` operator:

```
xgl_object_set(ctx,XGL_CTX_DEVICE, win_ras,NULL);
```

Note – The current implementation of XGL, using DGA, does not support `fork()` or `vfork()` if there is an active XGL Window Raster. The application can create only one XGL Raster for each OpenWindows (X) window.

Stream Device Object

The XGL Stream device provides protocol-independent stream pipelines for creating formatted output such as CGM output. The interface to the Stream device is independent of each specific Stream implementation. Each implemented Stream device provides an `include` file that contains the mapping of from the pipeline-specific attributes to the generic Stream attributes.

To create a Stream device object, the application needs to supply XGL with the name of the library that provides the Stream device functionality, and, if the Stream pipeline requires other information, a pointer to that information. This information is provided to XGL in the `xgl_object_create()` *desc* parameter, which points to an *Xgl_obj_desc* union containing a `stream` structure. The actual name of the library is defined in the header file provided with the device pipeline for the Stream device. See the documentation for the particular Stream device for specific information on the name of the library and for required device-specific information. For an example of a Stream device, see "CGM Device Object" on page 72.

A Stream device provides support for the following XGL attributes:

- `XGL_OBJ_APPLICATION_DATA`
- `XGL_OBJ_TYPE`
- `XGL_OBJ_SYS_STATE`
- `XGL_DEV_COLOR_MAP`
- `XGL_DEV_REAL_COLOR_TYPE`
- `XGL_DEV_COLOR_TYPE`
- `XGL_DEV_CONTEXTS`
- `XGL_DEV_CONTEXTS_NUM`
- `XGL_DEV_MAXIMUM_COORDINATES`

The Stream device may provide additional attributes that are passed through XGL from the application to the pipeline and from the pipeline to the application. After creating a Stream device, the application can inquire the values of the attributes using the `xgl_object_get()` operator. See the documentation for the Stream device for information on available attributes.

5

Raster Object Attributes

Raster attributes define the Window and Memory Raster Device objects created by the Device object operators. Some are shared by both Memory and Window Device objects, and others are specific to one or the other.

General Raster Attributes

Raster attributes that affect all Raster objects are:

`XGL_DEV_COLOR_MAP`
This attribute allows the application to establish its own color map in the Raster Device for specifying the colors rendered onto the Raster. Color maps apply to Rasters of type `XGL_COLOR_INDEX`. They are not applicable to Rasters of type `XGL_COLOR_RGB` since user colors are mapped directly to hardware lookup tables. Once a Color Map object has been created, it can be attached to the Raster using `xgl_object_set()` and this attribute. The default value is a handle to a Color Map object containing a two-element, black and white color table.

After an application sets this attribute and before attempting to draw into the Raster, the application must call `xgl_context_new_frame()` on the Context to which the Raster is attached. This ensures the correct colors.

`XGL_DEV_REAL_COLOR_TYPE`
This is a read-only attribute that indicates the actual color space used to store pixels in an XGL Device and the information contained in an actual pixel. This information is device-dependent.

`XGL_DEV_COLOR_TYPE`
This attribute defines the color space used for the calculation of color and lighting. It is set at the time of Raster creation and can only be set once. The color space of the device for a window raster object is directly related to the visual class of the X window underlying the window raster object. If the visual class of the X window is PseudoColor, then the color type of the window raster object should be `XGL_COLOR_INDEX`. If the visual class of the X window is TrueColor or DirectColor, the color type of the window raster object should be `XGL_COLOR_RGB`. The color spaces currently supported in XGL as defined by the data type *Xgl_color_type* are:

- XGL_COLOR_RGB, where color is specified as a fractional component of each of the three primary colors: red, green, and blue. Each component is a floating-point number between 0.0 and 1.0, which represents the additive weight the primary contributes to the final color. The visual type of the X window underlying a window raster of this color type should be TrueColor or DirectColor**.** Note that if the visual type is DirectColor, XGL will behave as if on a TrueColor visual.
- XGL_COLOR_INDEX, where color is specified as an unsigned integer index into a color table. The color table of an XGL Color Map object maps index colors into RGB color values. The default value is device-dependent. The visual type of the X window underlying a window raster of this color type should be PseudoColor**.**

Thus, for example, if the visual class of the X window underlying the window raster object is PseudoColor, the color type of the Window Raster object is set as shown in this code fragment:

```
if (vis->class == PseudoColor)
    ras = xgl_object_create(XGL_WIN_X, &xgl_x_win,
                    XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX, NULL);
```

If the visual class of the X window is TrueColor or DirectColor, the color type of the window raster object is set to XGL_COLOR_RGB as follows:

```
if (vis->class == TrueColor || vis->class == DirectColor)
    ras = xgl_object_create(XGL_WIN_X, &xgl_x_win,
                    XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB, NULL);
```

Note – For good performance, you should understand the color model of the target device and program accordingly. Take care when selecting the visual because the chosen visual class affects the XGL hardware color type, which in turn affects performance. For example code showing the proper way to select the visual that gives the best performance on any frame buffer, see the *XGL Accelerator Guide for Reference Frame Buffers*.

XGL_DEV_CONTEXTS_NUM
: This read-only attribute returns the number of Context objects associated with a Device. The default value is 0.

XGL_DEV_CONTEXTS
This read-only attribute returns a list of the Context object handles associated with a Device. To use this attribute, the application should first get the number of Contexts associated with the Device using the XGL_DEV_CONTEXTS_NUM attribute, allocate sufficient memory to hold the array of object handles, where each object handle is *Xgl_ctx* in size, and then get the list of object handles. The default value is NULL.

XGL_DEV_MAXIMUM_COORDINATES
This read-only attribute returns the Device's maximum *x* and *y* coordinates values. In addition, the maximum Z-buffer value allowed by the Device, is returned in the *z* value of the return *Xgl_pt_f3d* argument. The default values are device-dependent.

XGL_RAS_DEPTH
This attribute defines the number of bits used to specify the color of one pixel in an XGL Raster. It is read-only for Window Rasters and read-write for Memory Rasters. Four depths are supported: 1, 4, 8, and 32. The 1-bit depth is used only for creating stipple patterns by the `xgl_context_copy_buffer()` operator and other polygon-fill attributes. The 4-bit, 8-bit, and 32-bit depths are used for Memory Rasters into which the XGL application can render. The default value is 8 for Memory Rasters. If the depth of the Memory Raster is 8, the recommended device color type is XGL_COLOR_INDEX. If the depth of the Memory Raster is 32, the only device color type that is supported is XGL_COLOR_RGB. The default value is device-dependent for Window Rasters.

XGL_RAS_HEIGHT
This attribute defines the height of an XGL Raster. Window Raster height is obtained from the size of the window passed to the object create call. Memory Raster height can be set or reset at any time. The maximum Memory Raster height is 4K. Changing the height of a Memory Raster results in the loss of all pixel data it contains at that time. The default value is 256 for a Memory Raster.

XGL_RAS_WIDTH
This attribute defines the width of an XGL Raster. The Window Raster width is obtained from the size of the window passed to the object create call. The Memory Raster width can be set or reset at any time. The maximum Memory Raster width is 4K. Changing the width of a Memory Raster results in the loss of all pixel data it contains. The default value is 256 for a Memory Raster.

XGL_RAS_SOURCE_BUFFER
This attribute specifies the source buffer for the raster operations xgl_context_copy_buffer() and xgl_context_get_pixel(). The source buffer can be a raster's draw buffer or its Z-buffer. The default value is the draw buffer.

XGL_RAS_RECT_NUM
This attribute specifies the number of clip rectangles in the application clip list.

XGL_RAS_RECT_LIST
This attribute specifies the list of clip rectangles in the application clip list.

Window Raster Attributes

Raster attributes that affect all Window Raster objects are:

XGL_WIN_RAS_TYPE
This attribute supplies the type of Window Raster Device. It is a read-only attribute that describes the window system controlling the Window Raster. The value of this attribute is the value the application passes to XGL with the operator xgl_object_create(). XGL returns the following value:

XGL_WIN_X The Window Raster is an X11 window.

XGL_WIN_RAS_DESCRIPTOR
This attribute is the *descriptor* passed to XGL on the creation of the Window Raster. Its value is the same as the value passed to XGL by the operator xgl_object_create(). It returns a value of type *Xgl_X_window,* as X11 windows are the only supported window type. There is no default value.

XGL_WIN_RAS_POSITION
This read-only attribute specifies the position of the Window Raster object. It returns the position of the window in device coordinates relative to the upper left corner of the display screen.

XGL_WIN_RAS_PIXEL_MAPPING
This attribute specifies the pixel mapping array from the user's index colors to the window system color map index colors. This attribute must be set by the application immediately after setting XGL_CMAP_NAME for the Color

Map object associated with the Window Raster. The returned pixel array of the color map created for the Window System is sent to XGL via XGL_WIN_RAS_PIXEL_MAPPING. The default value is *identity*.

XGL_WIN_RAS_STEREO_MODE
This attribute specifies whether the window being rendered into is a stereo window or a monocular (non-stereo) window. If the attribute is set to XGL_STEREO_NONE and the frame buffer is in stereo mode, all graphics will be rendered into both the left and right eye buffers. If the value is set to XGL_STEREO_LEFT, the image for the left eye buffer is rendered; if the value is set to XGL_STEREO_RIGHT, the image for the right eye buffer is rendered. Note that undefined results may occur if stereo is used when rendering through a 2D Context.

XGL_WIN_RAS_BUF_DISPLAY
This attribute specifies the current display buffer in the application window. When the display and draw buffers are equivalent, the Window Raster is in single-buffer mode. The default value is 0, and the default buffering value is single buffer.

XGL_WIN_RAS_BUF_DRAW
This attribute specifies the buffer for reading and writing pixels. All XGL drawing primitives take place in this buffer. The default value is 0.

XGL_WIN_RAS_BUF_MIN_DELAY
This attribute specifies the minimum number of milliseconds to wait between switches of the display and draw buffers. Minimum delay allows the application to control the speed of the animation loop so it appears synchronized at a rate the application can support. The default value is 0.

XGL_WIN_RAS_BUFFERS_REQUESTED
This attribute allocates a specified number of hardware buffers if supported by the hardware. If the hardware does not support multi-buffering, then only 1 buffer will be allocated. If the application sets the number to 1, XGL will do single buffering. If the value is set to 2, XGL will do double buffering, if double buffering is supported by the hardware and the underlying Window Raster. The application should perform a *get* on the attribute XGL_WIN_RAS_BUFFERS_ALLOCATED immediately after setting XGL_WIN_RAS_BUFFERS_REQUESTED, to determine how many buffers were allocated in the hardware. If the hardware supports only a single buffer, color map double buffering is possible. (For information about Color Map double buffering, see Chapter 6, “Color”.)

The value of this attribute determines the valid values for the attributes XGL_WIN_RAS_BUF_DISPLAY and XGL_WIN_RAS_BUF_DRAW. The default value is 1 (single buffered).

XGL_WIN_RAS_BUFFERS_ALLOCATED
This read-only attribute reflects the number of buffers allocated in the hardware underlying a Window Raster. The application must first set the XGL_WIN_RAS_BUFFERS_REQUESTED attribute to either 1 (single buffered) or 2 (double buffered). The value of this attribute determines the valid values for the attributes XGL_WIN_RAS_BUF_DISPLAY and XGL_WIN_RAS_BUF_DRAW. The default value is 1 (single buffered).

Note – Since the concurrent use of backing store and double buffering is not supported, XGL will enable whichever is requested first. If a request for double buffering is made by setting XGL_WIN_RAS_BUFFERS_REQUESTED to 2 and backing store has already been enabled, XGL will not initiate the double buffering request and will set the value of XGL_WIN_RAS_BUFFERS_ALLOCATED to 1.

XGL_WIN_RAS_BACKING_STORE
This attribute enables backing store support for a Window Raster, after backing store support is requested through the Xlib call XChangeWindowAttributes(). Since concurrent use of backing store and double buffering is not supported, XGL will enable whichever is requested first. If a request for backing store is made by setting XGL_WIN_RAS_BACKING_STORE to TRUE and the value of XGL_WIN_RAS_BUFFERS_ALLOCATED is greater than 1, XGL will not initiate the backing store request and will set the value of XGL_WIN_RAS_BACKING_STORE to FALSE.

Note – The X server does not handle the copying of a Z-buffer or accumulation buffer for backing store. When a Z-buffer and/or an accumulation buffer is enabled and the display device uses a software Z-buffer and/or accumulation buffer, the backing store will share the software buffers with the display window, so the contents of the buffers will be the same without copying. If the display device uses a hardware Z-buffer and/or accumulation buffer, the buffers cannot be shared with the backing store. In this case, the device pipeline for the backing store will use a separate Z-buffer for rendering and/or a separate accumulation buffer to accumulate. The resulting image after damage repair is likely to be inaccurate, especially if the clip list has changed. Thus, backing store should not be used when accumulation is needed or when `XGL_3D_CTX_HLHSR_MODE` is set to `XGL_HLHSR_Z_BUFFER` for a 3D Context.

Memory Raster Attributes

`XGL_MEM_RAS_IMAGE_BUFFER_ADDR`

This attribute reflects the address of the array of pixels for an XGL Memory Raster, allowing the application programmer to read and write pixel data. The application programmer must know the depth, width, and height of a Raster to access its pixel values correctly. The memory address varies depending on the Raster depth:

- 1-bit Rasters: The memory address points to the word containing the upper left pixel. The pixels are stored along the row as bits, the left-most pixel being the most significant bit (MSB) in the word. To allow each row to begin at a 16-bit boundary, padding can be added to the least significant bits of the word at the end of each row.
- 4-bit Rasters: The memory address points to the word containing the upper left pixel. Each byte contains two pixels stored along the row. Each row begins at a 16-bit boundary, and padding may be added to the row.
- 8-bit Rasters: The memory address points to the word containing the upper left pixel. Each byte contains one pixel, stored as an unsigned char, along the row. Each row begins at a 16-bit boundary.
- 32-bit Rasters: The memory address points to the word containing the upper-left pixel. Each pixel occupies 32 bits, and the pixels are aligned to the 32-bit boundary.

Note – When creating a Memory Raster, the application program must set the `XGL_RAS_DEPTH`, `XGL_RAS_WIDTH`, and `XGL_RAS_HEIGHT` attributes in the `xgl_object_create()` call before setting the `XGL_MEM_RAS_IMAGE_BUFFER_ADDR` attribute.

`XGL_MEM_RAS_Z_BUFFER_ADDR`

This attribute reflects the starting address of the block of memory for the Z-buffer of a Memory Raster, allowing the application programmer to read and write Z-buffer values.

The Z-buffer is a collection of *z* values. Each *z* value occupies 32 bits. XGL accesses the least significant three bytes of the 32-bit word during Z-buffer operations. Thus, each row begins at a 32-bit word boundary. The Z-buffer of a Memory Raster has 24 bit planes and thus a depth of $(2^{24} - 1)$. The height of the Z-buffer is based on the raster height (`XGL_RAS_HEIGHT`), and the width is based on the raster width (`XGL_RAS_WIDTH`).

Note – When copying from an indexed 8-bit Memory Raster into an indexed 24-bit Window Raster, a separate Color Map object is required for each raster. Since XGL does internal color map conversion, which is different in these cases, the same color map cannot be used when the underlying hardware is different.

Note – If your application copies from a memory raster to a window raster and the color map being used does not start at zero, you may not get the colors you expect. The ROP mode for this copy buffer case sets the start index value for the color map to zero. To solve this problem, initialize the area using the `xgl_context_new_frame()` operator.

5

CGM Device Object

XGL can open and write to a Computer Graphics Metafile (CGM), Version 1. The XGL CGM file conforms to the ISO 8632:1985 standard. Metafiles are UNIX file streams that contain graphical information in a special format, and are used to store or interchange graphical information. A CGM is not a Raster Device. It is a data file consisting of a stream of graphical information. The CGM object uses higher-level primitives; the resulting output is compact and often easier to manipulate than a raster file. For example, modifying orientation is easier on CGM polygon data than on a polygon-shaped pixel area of a raster file.

The XGL CGM Device supports both 2D and 3D Contexts. However, the CGM standard deals strictly with 2D coordinates. The XGL implementation is further restricted to outputting only 2D integer coordinates. Although the CGM Device outputs only x and y coordinates, all stages of the XGL viewing pipeline are still supported. In the current implementation, stroke text, 3D circles, circular arcs, ellipses, elliptical arcs, and NURBs are output as polylines or polygons. Lines are interpolated, and surfaces are constant shaded. Picking is not supported. Although point colors and normals are ignored by the CGM device, the XGL pipeline processes them; therefore, face culling and face distinguishing are possible on a CGM device. Pixel operations are not supported in the current implementation. User-defined line and fill patterns are not supported.

The XGL implementation supports Binary, Character, and Clear Text CGM encoding formats. Both Long and Short integer coordinate representations are available in the current implementation. Floating point coordinates are not currently supported.

Creating a CGM Device

To create a CGM device, the application must supply the name of the XGL CGM library, `XGL_CGM_LIB_NAME`, in the Stream object descriptor. The name of the library is defined in the header file provided with the CGM device. At device creation time, the application can also pass in Stream device attributes. The code fragment below shows the creation of an XGL CGM device.

```
#include <stdlib.h>
#include <X11/Xlib.h>
#include <xgl/xgl.h>
#include <xgl/xgl_cgm-2.0.h>
```

```
static Xgl_3d_ctx     ctx = NULL;/* XGL context object */
static Xgl_sys_state  sys_st;    /* XGL system state object */

main (argc, argv)
   int     argc;
   char    *argv[];
{
    Display           *display;   /* pointer to X display */
    Xgl_obj_desc      obj_desc;   /* XGL object descriptor */
    char*                fname = "CGM_file";
    float                scale_factor = 1000.0;
    Xgl_cgm              cgm;
    Xgl_cgm_description  cgm_desc;

    if (argc == 2)
        fname = argv[1];

    sys_st = xgl_open(NULL);

    /* Use the XGL CGM pipeline provided with Solaris */
    obj_desc.stream.name = XGL_CGM_LIB_NAME;
    obj_desc.stream.desc = NULL;

    cgm_desc.cgm_fp = fopen(fname, "w");
    cgm_desc.cgm_signature = NULL;
    cgm_desc.cgm_description = NULL;
    cgm = xgl_object_create(sys_st,
                XGL_CGM_DEV, &obj_desc,
                XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                XGL_CGM_DESCRIPTION, &cgm_desc,
                XGL_CGM_ENCODING, XGL_CGM_CLEAR_TEXT,
                XGL_CGM_VDC_EXTENT, XGL_CGM_VDC_EXT_SHORT,
                XGL_CGM_SCALE_MODE, XGL_CGM_METRIC,
                XGL_CGM_SCALE_FACTOR, &scale_factor,
                NULL);

  /* create XGL Context object using the CGM device */
  ctx = xgl_object_create (sys_st,
                XGL_3D_CTX, NULL,
                XGL_CTX_DEVICE, cgm,
                XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                XGL_CTX_NEW_FRAME_ACTION, XGL_CTX_NEW_FRAME_CLEAR,
               NULL);
}
```

5

Creating a Picture

The `xgl_object_create()` operator results in the following lines being inserted into a metafile on a CGM Device outputting clear text:

```
(open file)
BEGIN METAFILE <string1> /*signature from description*/
METAFILE VERSION 1
METAFILE DESCRIPTION <string2>
VDC TYPE INTEGER /*all metafiles are this type*/
METAFILE ELEMENT LIST
DRAWING BEGMFDEFAULTS ENDMFDEFAULTS COLRINDEXPREC MAXCOLRINDEX
INTEGERPREC REALPREC INDEXPREC COLORPREC COLRVALUEEXT
SCALEMODE COLRMODE LINEWIDTHMODE MARKERSIZEMODE
```

The `xgl_object_destroy()` operator destroys the CGM Device. When `xgl_object_destroy()` is invoked, the CGM device closes the current picture, if it isn't already closed. The CGM Device then ensures that an "end metafile" delimiter has been written, and flushes the output stream. The user must close the stream. As with any XGL device, a call to `xgl_object_destroy()` will not have an effect until all XGL objects using the device are destroyed. Overlooking this can result in incomplete metafiles, since the file will not be flushed or closed.

Invoking the `xgl_context_new_frame()` operator (used to delimit CGM pictures) starts a new picture element in the Metafile. The current color map, which should be loaded by this time, is output to the Metafile. The `xgl_context_new_frame()` operator results in the output of the "begin picture" delimiter to the Metafile. Any picture already open is closed, and a new picture is started.

The following Context attributes apply to the CGM Device:

```
XGL_CTX_DEFERRAL_MODE
XGL_CTX_DEVICE
XGL_CTX_VDC_ORIENTATION
XGL_CTX_BACKGROUND_COLOR
```

CGM Metafile Device Attributes

Since a CGM Device is not a Raster Device, none of the Raster Device-specific attributes apply. Attributes that affect CGM Device objects are:

`XGL_CGM_TYPE`
This read-only attribute defines the type of CGM Metafile object associated with an XGL object; it cannot be modified later. It can be used to query the CGM Metafile object type when only a generic *Xgl_cgm* handle is known. The only type of CGM Metafile object that the XGL library currently supports is `XGL_CGM_V1`.

`XGL_CGM_ENCODING`
This attribute defines the data representation used to store graphical objects in an XGL Metafile. The attribute can only be set before any part of the Metafile is written to the output stream, after which the attribute becomes read-only.

XGL supports three standard CGM encoding formats.

- `XGL_CGM_CLEAR_TEXT` uses the 94 printing characters from the ISO 646 7-bit code table to represent data. Designed to be "human-readable" and self-delimiting, clear text encoding is comparatively slow and complex to parse and generate. This is the default value.
- `XGL_CGM_CHARACTER` uses ASCII bytes, including control characters, to represent element codes and parameters. It is self-delimiting, although more compact than clear text encoding. Designed for transmission over communication lines, it is comparatively slow to parse and generate.
- `XGL_CGM_BINARY` uses internally formatted binary numbers to represent data. It is the fastest to read and write, and provides the most compact representation.

`XGL_CGM_DESCRIPTION`
This attribute defines the Metafile header information and the file pointer to the Metafile output file. It can only be set before any part of the Metafile has been written to the output stream, after which the attribute becomes read-only. The Metafile description information is specified by the data type *Xgl_cgm_description*, which has three components:

- *cgm_fp* is the stream pointer where the Metafile should be written. If the file already exists, XGL appends the Metafile to it. If XGL is given a structure with the file pointer equal to NULL, the default value is `stdout`.

- *cgm_signature* is a string containing the Metafile label, which could include a description of the Metafile's contents. If XGL is given a structure with the string pointer equal to NULL, the default value is *XGL 3.0 CGM 1.0 current date and time.*

cgm_description is a string containing descriptive information about the Metafile, such as the author, the place of origin, and date and time of generation. If XGL is given a structure with the string pointer equal to NULL, the default value is: *Sun Microsystems CGM TOP/FULL conformance.*

`XGL_CGM_SCALE_MODE`
This attribute defines the data representation used to store graphical objects in an XGL Metafile. It can only be set before any part of the Metafile has been written to the output stream, after which time the attribute becomes read-only. Two encoding formats are currently supported in XGL as defined by the data type *Xgl_cgm_scale_mode*:

- `XGL_CGM_ABSTRACT` means there is no correspondence between the range of coordinate values in the Metafile and the image it produces. In this case, the SCALEMODE will always be ABSTRACT, and the scale factor will always be 1.0. `XGL_CGM_ABSTRACT` is the default value.
- `XGL_CGM_METRIC` means there is a correspondence between the range of coordinate values in the Metafile and the image it produces. In this case, the SCALEMODE will be METRIC, and the scale factor (set with the attribute `XGL_CGM_SCALE_FACT`), will indicate the mapping between one unit in Metafile coordinate values and one millimeter in the displayed image. This mode allows images of predetermined size to be exchanged using CGM.

`XGL_CGM_SCALE_FACTOR`
This attribute defines the scale factor used when the SCALEMODE delimiter is set to METRIC. It can only be set before any part of the Metafile has been written to the output stream, after which the attribute becomes read-only. The value is the address of a floating point number. The default scale value is 1.0.

`XGL_CGM_PICTURE_DESCRIPTION`
This attribute defines the descriptive label for a picture within an XGL Metafile. It can only be set before any part of the Metafile has been written, after which the attribute becomes read-only. The picture name attribute is described as a string, *Xgl_sgn8 **. This string value is output to the Metafile with the `BEGIN PICTURE` delimiter in the Metafile. This delimiter is written

(and a new picture started) when the user calls `xgl_context_new_frame()`. Any changes to the picture name attribute after this call does not affect the current picture.

`XGL_DEV_COLOR_MAP`
This attribute permits the application to set its own Color Map into the Metafile, which will be used to specify the colors rendered into the Metafile. The attribute specifies the handle of the Color Map object to be associated with the Metafile Device. Once the Color Map is created, it is attached to the Metafile using the `xgl_object_set()` operator and this attribute. Color Map objects can only be attached to Rasters and Metafiles, not Contexts. The default value is a handle to a Color Map object containing a two-element, monochrome (black and white) color table if `XGL_DEV_COLOR_TYPE` is set to `XGL_COLOR_INDEX` and a 6-9-4 color cube is `XGL_DEV_COLOR_TYPE` is set to `XGL_COLOR_RGB`.

`XGL_CGM_VDC_EXTENT`
This attribute controls the type and range of the coordinate values. XGL currently supports only integer extent. The values are:

- `XGL_CGM_VDC_EXT_SHORT`, a 16-bit VDC extent.
- `XGL_CGM_VDC_EXT_LONG`, a 32-bit VDC extent. This extent is implemented but not officially supported. This is due to overflow problems with transformations of 32-bit integers and floating point numbers in the XGL pipeline.

XGL CGM Line Patterns

CGM Metafiles only support a fixed set of line patterns. The default pattern is solid fill. User-defined line patterns or line pattern types not on the list below will default to solid lines. The line patterns supported are as follows:

```
xgl_lpat_cgm_dotted
xgl_lpat_cgm_dashed
xgl_lpat_dash_dot
xgl_lpat_dash_dot_dotted
xgl_lpat_long_dashed
```

An application must use the XGL attribute `XGL_CTX_LINE_PATTERN` or `XGL_CTX_EDGE_PATTERN` to achieve line or edge patterns in the CGM Metafile.

5

XGL CGM Markers

CGM Metafiles only support a fixed set of marker types. The default marker type is a dot marker. If there is no application data (XGL_OBJ_APPLICATION_DATA) attached to the marker object currently in use, user-defined markers and marker types not on the list below default to dot markers. The marker types supported are as follows:

```
xgl_marker_dot
xgl_marker_plus
xgl_marker_asterisk
xgl_marker_circle
xgl_marker_cross
```

An application must use the XGL attribute XGL_CTX_MARKER to set the current marker type in the CGM Metafile.

An application can place user-defined markers in the CGM Metafile by setting the XGL_OBJ_APPLICATION_DATA attribute of the marker object to point to the marker data, as in the example below:

```
Xgl_marker            my_marker;
Xgl_sys_state         sys_state;
Xgl_pt_list_list      pll;
float                 scale_factor;

sys_state - xgl_open(<attribute-list>);
<Initialize pll for user-defined marker>;
<Set scale factor >;
my_marker = xgl_object_create(sys_state,
                      XGL_MARKER,NULL,
                      XGL_MARKER_DESCRIPTION, &pll,
                      XGL_OBJ_APPLICATION_DATA, &scale_factor,
                      0);
```

When application data is added to a Marker object, XGL will stroke out the marker into lines and scale these lines by a factor equal to the scale factor pointed to by the application data multiplied by XGL_CTX_MARKER_SCALE_FACTOR. Whereas normal markers have a fixed size, stroked out markers will scale as the size of the image producted from the CGM Metafile is changed.

Important Notes on Integrating XGL with a Windowing System

XGL is an output-only library. The application uses the window system's input and event mechanisms to provide interaction with the user. X toolkit and Xlib function calls interact with the OpenWindows environment to create and manipulate windows on the display device, and XGL draws into the X11 windows created by the application. If the canvas has been created using an X toolkit, the application must explicitly get the X window ID from the canvas to associate the canvas with a Window Raster. The Xlib `XCreateWindow` call returns the handle that XGL requires.

Mixing XGL and Xlib Drawing Routines

The DGA facility allows XGL to communicate directly with the graphics hardware, without going through the Xlib communications path to the server. Because of this, application programmers must be careful when mixing XGL and Xlib calls, as latencies involved with XGL communication with graphics hardware and Xlib communication with the server can result in geometry being drawn in an unexpected order. For example, a line drawn by an Xlib function may appear on the screen after a polygon drawn by XGL even though the Xlib function was called first. If the order in which the primitives are drawn is important to an application, the following precautions should be taken when mixing XGL and Xlib functions:

- An application that has finished drawing with XGL primitives should call `xgl_context_flush(ctx,XGL_FLUSH_SYNCHRONIZE)` to ensure that XGL primitives are displayed before calling Xlib functions. This operator should be called even if the `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASAP`.
- Xlib calls should be flushed with `XSync` before further XGL commands are issued. However, the use of `XSync` may cause some performance degradation.

Note also that when rendering and clearing the window with Xlib calls, the application must clear the window with `xgl_context_new_frame()` before rendering with XGL primitives, even if the window has previously been cleared with Xlib calls.

Window Resize Events

In OpenWindows, window resize events are *not* trapped internally by XGL. The application must explicitly handle these events with the XGL operator `xgl_window_raster_resize()`. This operator causes XGL to resize its Device Coordinate boundaries and, for 3D applications, update the size of the software Z-buffer and accumulation buffer, if applicable.

If the application will run as a DGA application, it should call `XSync()` before calling `xgl_window_raster_resize()` in order to synchronize XGL and the server. XView, and possibly other toolkits, may call the application resize procedure before sending the resize request to the server. This causes DGA to return a window size that is behind by one resize event. Calling `XSync()` before calling `xgl_window_raster_resize()` will minimize this problem.

Note – Be sure to check that the XGL Window Raster has been created before calling the `xgl_window_raster_resize()` operator. An error may occur if a the resize operator is called before the Window Raster is created.

Device Example Programs

The following programs illustrate creating Device objects using Xlib, the XView toolkit, and the OLIT toolkit. In an XGL program, the Xlib or X toolkit calls are interleaved with the XGL calls, and the results of the X and XGL calls share the same drawable area on the screen. The output of all three example programs is the same and is shown in Figure 5-1 on page 81. The source code for the examples follows.

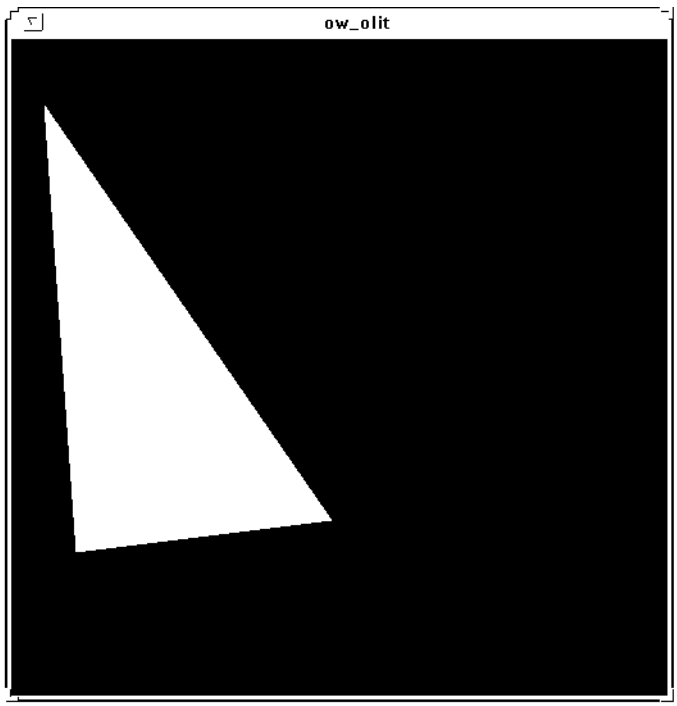

Figure 5-1 Output of `ow_olit.c`

XGL and Xlib

The example program `ow_xlib.c` uses Xlib calls to create the window used by XGL. To compile this program, type `make ow_xlib` in the example program directory.

Code Example 5-1 Using XGL with Xlib

```
/* ow_xlib.c - attach XGL Window Raster to OpenWindows Xlib window */
/*
 * This program demonstrates how to create an XGL Window Raster given
 * an Xlib window.
 */

#include <stdio.h>
#include <X11/Xlib.h>
```

```
#include <X11/Xutil.h>
#include <xgl/xgl.h>

/* need this global so that quit_proc can get it */
static Xgl_objectsys_st;/* XGL System State object */

/* need this global so that repaint procedure can get it */
static Xgl_objectctx = NULL;/* XGL graphics Context object */

main (
    intargc,
    char*argv[])
{
    Xgl_object    win_ras = NULL;/* XGL Window Raster object */
    Xgl_X_window  xgl_x_win;     /* XGL-X data structure */
    Xgl_obj_desc  win_desc;      /* XGL win ras structure */

    Xgl_pt_list   pl;            /* polygon point structure */
    Xgl_pt_i2d    pts_i2d[3];    /* integer 2d points */

    Display       *display;      /* X display data structure */
    int           screen;        /* X screen */
    Visual        *vis;          /* X color visual */
    Window        win;           /* X window */
    XSetWindowAttributes attrs;  /* X create window attributes */
    XEvent        event;         /* X event */

    intdone = 0;/* quit flag */

    /* open X display */
    if ((display = XOpenDisplay (NULL)) == NULL) {
       (void) fprintf (stderr, "cannot open display\n");
       exit (1);
    }

    /* get screen number for this display */
    screen = DefaultScreen (display);

    /* express interest in these events */
    attrs.event_mask = ButtonPressMask | ButtonReleaseMask |
                       ExposureMask;

    /* create window of type visual */
    if (!(win = XCreateWindow (display,
              RootWindow (display, screen),
              10, 10, 500, 500, 1,
```

```
            XDisplayPlanes (display, 0),
            CopyFromParent, CopyFromParent,
            CWEventMask, &attrs))) {
   printf ("can't create visual window\n");
   exit (1);
}
XSetStandardProperties (display, win,
                        "Press any mouse button to quit",
                        "X-Xgl", None, argv, argc, NULL);

/* map and wait for exposure */
XMapWindow (display, win);
do {
   XNextEvent (display, &event);
} while (event.type != Expose);

/* create XGL System State object */
sys_st = xgl_open (NULL);

/* copy X information into XGL data structure */
xgl_x_win.X_display = (void *) display;
xgl_x_win.X_window = (Xgl_usgn32) win;
xgl_x_win.X_screen = (int) screen;

/* create Window Raster Device using XView canvas */
win_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
win_desc.win_ras.desc = &xgl_x_win;

win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                             XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                             NULL);

/* create XGL graphics Context object using */
/* the Window Raster object */
ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
        XGL_CTX_DEVICE, win_ras,
        XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
        NULL);

/* clear canvas area to default background color of black */
xgl_context_new_frame (ctx);
/* setup data points for a triangle */
/* XGL will close the triangle */
pts_i2d[0].x = 25;
pts_i2d[0].y = 50;
pts_i2d[1].x = 50;
```

```
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);

    /* wait for user input and quit on key press event */
    done = 0;
    while (!done) {
       XNextEvent (display, &event);
       if (event.type == Expose) {
           if (!event.xexpose.count) {
               xgl_window_raster_resize(win_ras);
               /* clear rendering area */
               xgl_context_new_frame (ctx);
               /* draw a polygon in default color of white */
               xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
           }
       }
       else if (event.type == ButtonPress) {
           done = 1;
       }
    }

    xgl_context_post(ctx, TRUE);

    /* close XGL and free all memory used by XGL */
    xgl_close (sys_st);

    /* destroy X11 resources used */
    XDestroyWindow (display, win);
    XCloseDisplay (display);

    exit (0);
}
```

XGL and the XView Toolkit

The example program `ow_xview.c` uses the XView toolkit to create the window used by XGL. To compile this program, type `make ow_xview` in the example program directory.

Code Example 5-2 Using XGL with XView

```
/* ow_xview.c - attach XGL Window Raster to OpenWindows XView canvas
*/

/*
 * This program demonstrates how to create an XGL Window Raster given
 * an XView canvas.
 */

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

#include <xgl/xgl.h>

/*
 * need this global so that quit_proc can get it
 */
static Xgl_object    sys_st;    /* XGL System State object */

/*
 * need this global so that repaint procedure can get to it
 */
static Xgl_object    ctx = NULL; /* Graphics Context object */
static Xgl_object    win_ras = NULL; /* Xgl window raster */

void    event_proc (Xv_Window, Event*, Notify_arg);
void    repaint_proc (Canvas, Xv_Window, Display*, Window,
                      Xv_xrectlist*);
Notify_value    quit_proc (Frame, Destroy_status);

main (
    int             argc,
    char            *argv[])
```

```
{
    Xgl_X_window      xgl_x_win;  /* XGL-X data structure */
    Xgl_obj_desc      win_desc;   /* Window raster structure */

    Frame             frame;    /* XView frame around window */
    Canvas            canvas;   /* XView canvas in frame */

    Xv_Window         pw;              /* XView paint window */
    Window            frame_window;    /* XID of frame */
    Window            canvas_window;   /* XID of canvas */

    Display           *display;        /* pointer to X display */
    int               screen;          /* X screen number */

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    /*
     * create a XView simple window frame - WIN_DYNAMIC_TRUE must be
     * specified for OW 1.0 where the default is static color maps
     */
    frame = xv_create (NULL, FRAME,
                       FRAME_LABEL,      "View Window",
                       WIN_DYNAMIC_VISUAL,     TRUE,
                       OPENWIN_AUTO_CLEAR,     FALSE,
                       CANVAS_RETAINED,        FALSE,
                       CANVAS_FIXED_IMAGE,     FALSE,
                       WIN_DEPTH,              8,
                       XV_VISUAL_CLASS,        PseudoColor,
                       NULL);

    /*
     * create XView canvas - WIN_DYNAMIC_TRUE must be specified OW
     * where the default is static color maps
     */
    /* set event and repaint procedures */
    canvas = xv_create (frame, CANVAS,
                        WIN_DYNAMIC_VISUAL,      TRUE,
                        OPENWIN_AUTO_CLEAR,      FALSE,
                        CANVAS_RETAINED,         FALSE,
                        WIN_EVENT_PROC,          event_proc,
                        CANVAS_REPAINT_PROC,     repaint_proc,
                        WIN_DEPTH,               8,
                        XV_VISUAL_CLASS,         PseudoColor,
                        NULL);

    /* get X stuff */
```

```
    display = (Display *) xv_get (frame, XV_DISPLAY);

    pw = (Xv_Window) canvas_paint_window (canvas);
    canvas_window = (Window) xv_get (pw, XV_XID);
    frame_window  = (Window) xv_get (frame, XV_XID);

    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW (pw);
    xgl_x_win.X_window  = (Xgl_usgn32) canvas_window;
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

    /* wait for the window */
    sleep (2);

    /* create XGL System State object */
    sys_st = xgl_open (NULL);

    /* create Window Raster Device using XView canvas */
    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                                 NULL);

    /* create XGL Context object using the Window Raster object */
    ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
                             XGL_CTX_DEVICE, win_ras,
                             XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                             NULL);

    /* set quit procedure where xgl_close is called */
    notify_interpose_destroy_func (frame, quit_proc);

    /* go into XView loop */

    exit (0);
}


static
void
event_proc (
    Xv_Window            window,
    Event               *event,
    Notify_arg           arg)
{
    switch (event_action (event)) {
```

```
            /*
             * not anything to do here, but mouse and keyboard
             * events should be tracked in this procedure
             */
        default:
            break;
        }
}


/*
 * Xview repaint procedure.  Draw the polygon in this function.
 */
static
void
repaint_proc (
    Canvas              local_canvas,
    Xv_Window           local_pw,
    Display             *dpy,
    Window              xwin,
    Xv_xrectlist        *xrects)
{
    Xgl_pt_list         pl;  /* polygon point structure */
    Xgl_pt_i2d          pts_i2d[3]; /* integer 2d points */

    xgl_window_raster_resize(win_ras);

    /* clear canvas area to default background color of black */
    xgl_context_new_frame (ctx);

    /* setup data points for a triangle - XGL will close triangle */
    pts_i2d[0].x = 25;
    pts_i2d[0].y = 50;
    pts_i2d[1].x = 50;
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
```

```
}
static
Notify_value
quit_proc (
    Frame               fr,
    Destroy_status      status)
{
    if (status == DESTROY_CHECKING) {
        xgl_close (sys_st);
    }
    return (notify_next_destroy_func (fr, status));
}
```

XGL and the OLIT Toolkit

The example program `ow_olit.c` uses the OLIT toolkit calls to create the window used by XGL. To compile this program, type `make ow_olit` in the example program directory.

Code Example 5-3 Using XGL with OLIT

```
/* ow_olit.c - attach XGL Window Raster to OLIT stub widget */

/*
 * This program demonstrates how to create an XGL Window
 * Raster given an OLIT stub widget.
 */

#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/DrawArea.h>

#include <xgl/xgl.h>

/*
 * need these global so that expose procedure can get to them
 */
static Xgl_object     ctx = NULL;/* XGL Context object */
static Xgl_object     sys_st;    /* XGL System State object */
static Xgl_object     win_ras = NULL; /* XGL Window Raster */
```

```
main (
    int             argc;
    char            *argv[])
{
    Xgl_X_window    xgl_x_win; /* XGL-X data structure */
    Xgl_obj_desc    win_desc;  /* Window raster struct */

    Widget          toplevel;  /* Widget top-level */
    Widget          drawable;  /* Drawable widget */
    XtAppContext    app;

    Window          window;    /* X window for canvas */
    Display         *display;  /* pointer to X display */

    void            repaint(); /* OLIT repaint procedure */
    /*
     * Initialize toolkits
     */
    OlToolkitInitialize((XtPointer)NULL);
    toplevel = XtVaAppInitialize (&app, "ow_olit",
              (XrmOptionDescList)NULL, 0,
              &argc, argv, (String *)NULL,
              XtNtitle, "Simple OLIT Drawable",
              XtVaTypedArg, XtNvisual, XtRString,
              "PseudoColor", sizeof("PseudoColor"),
              NULL);

    /*
     * Create a draw area widget for rendering graphics
     * and add expose callback to draw area widget
     */
    drawable = XtVaCreateManagedWidget("drawable",
              drawAreaWidgetClass, toplevel,
              XtNlayout, OL_IGNORE,
              XtNheight, 512,
              XtNwidth,  512,
              NULL);
    XtAddCallback(drawable, XtNexposeCallback, repaint,
                  (XtPointer) NULL);

    XtRealizeWidget(toplevel);

    /* get X stuff */
    display = (Display *) XtDisplay (drawable);
    window = (Window) XtWindow (drawable);
```

```
    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) display;
    xgl_x_win.X_window  = (Xgl_usgn32) window;
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

    /* wait for the window */
    sleep (2);

    /* create XGL System State object */
    sys_st = xgl_open (NULL);

    /* create Window Raster Device using OLIT widget */
    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                                 NULL);

    /* create XGL Context object using the Window Raster object */
    ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
                             XGL_CTX_DEVICE, win_ras,
                             XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                             NULL);

    /* go into Xt main loop and wait for expose events */
    XtAppMainLoop(app);

    exit (0);
}

/*
 * Xgl repaint procedure called on expose events
 */
static
void
repaint ()
{
    Xgl_pt_list       pl;        /* polygon point structure */
    Xgl_pt_i2d        pts_i2d[3];/* integer 2d points */

    /* clear draw area to default background color of black */
    xgl_context_new_frame (ctx);

    /* set up data points for a triangle - XGL will close triangle */
    pts_i2d[0].x = 25;
    pts_i2d[0].y = 50;
    pts_i2d[1].x = 50;
```

```
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
}
```

Determining Device Acceleration

XGL accelerates a primitive if some portion of the geometry is processed directly by accelerating hardware. Acceleration can speed up geometric transformations and clipping, as well as actual rendering. To determine what acceleration features underlie a window, an application can use the `xgl_inquire()` operator. `xgl_inquire()` is defined as:

```
Xgl_inquire* xgl_inquire(
    Xgl_sys_state      sys_state
    Xgl_obj_desc       *desc);
```

The parameter `desc` contains window information. The operator returns a pointer to a structure of type *Xgl_inquire*. This structure contains the following fields of information about the frame buffer underlying the window:

Table 5-2 Fields Returned by `xgl_inquire()`

Field	Information
Name	The XGL name of the frame buffer underlying the window.
DGA flag	Specifies whether XGL does direct graphics access to render pixels into the frame buffer underlying the window. If DGA is not available, XGL uses Xlib or PEXlib to render pixels.
Color type	Indicates whether the frame buffer is color index or RGB.
Depth	Specifies the depth of the frame buffer underlying the window.

Table 5-2 Fields Returned by `xgl_inquire()` *(Continued)*

Field	Information
Width	Specifies the width of the frame buffer in pixel coordinates.
Height	Specifies the height of the frame buffer in pixel coordinates.
Maximum buffer	Specifies the maximum number of buffers available from the frame buffer underlying the window. A value of two means the application might be able to perform double buffering.
Double buffer is copy	Specifies whether the underlying hardware device performs double buffering by copying pixels from the draw (hidden) buffer to the display buffer.
Point type	Specifies the point types supported by the frame buffer.
HLHSR mode	Indicates whether the frame buffer underlying the window directly supports a Z-buffer and Z-buffer rendering.
Picking Double buffer Indexed color True color Depth cueing Lighting Shading Antialiasing HLHSR	These fields reflect the capabilities of the underlying hardware accelerator, if any. If the accelerator does not support this functionality, these fields are set to `XGL_INQ_NOT_SUPPORTED`. If support for the functionality is provided through software emulation, these fields are set to `XGL_INQ_SOFTWARE`. If hardware acceleration is provided, these fields are set to `XGL_INQ_HARDWARE`.
Stereo	Specifies whether the device underlying the window raster has hardware support for stereo.
Extensions	Bitmask indicating the extensions supported by the frame buffer underlying the window. Currently, only queries for the multibuffer extension to X (MBX). To determine whether MBX is supported, `AND` the value of this field with the bitmask `XGL_INQ_MULTI_BUFFERING`. If the result is not zero, the frame buffer supports MBX.

Note – The application must open XGL with a call to `xgl_open()` before calling `xgl_inquire()`. Device characteristics cannot be queried before XGL is open.

Note – The `xgl_inquire()` operator does not currently return information on accumulation or DGA transparent overlay capabilities.

Inquire Example Program

The program `inq.c` shows the use of `xgl_inquire()`. To compile this program, type `make inq` in the example program directory. An example of program output for a particular hardware device is shown in Figure 5-2. The source code for the example follows.

```
# inq
Inquire Information:
  Sun:GX
  Direct Graphics Access
    Color Model: INDEXED
    Depth : 8
    Width : 1152
    Height: 900
    Single Buffering
      Using bitblt transfer
    Point Types:
        2D
        3D
        INT
        FLOAT
    Software Zbuffer in host
    Picking emulated in software
    Double buffering emulated in software
    Index colors accelerated through hardware
    True colors emulated in software
    Depth cueing emulated in software
    Lighting emulated in software
    Shading emulated in software
    Hidden line removal emulated in software
    Antialiasing emulated in software
    No stereo support
    No Multi Buffering support
#
```

Figure 5-2 Output of `inq.c`

Code Example 5-4 Inquire Example

```
/*
 * inq.c
 */

#include <stdio.h>
#include <math.h>
#include <xgl/xgl.h>
#include <X11/Xlib.h>

/* Xgl inquiry function */

/*
 * main routine
 *      description:
 *          This example program opens an Xlib display and window
 *          and uses the Xgl inquiry function to determine the
 *          hardware features underlying that window.
 *      arguments:
 *              -display [X display number]
 */

static  char    *colon_zero = ":0";

main(
    int             argc,
    char            *argv[])
{
    Xgl_object      sys_st;
    Xgl_obj_desc    obj_desc; /* XGL window system information */
    Xgl_inquire     *inq_info;     /* Xgl inquiry structure */
    Xgl_X_window    xgl_x_win;     /* Xgl to X structure */
    char            *display_name; /* name of X display */
    Display         *display;      /* Xlib display structure */
    int             screen;        /* screen for X display */
    Window          window;        /* X window we create */

    if (argc < 2)
        display_name = colon_zero;
    else
        display_name = argv[2];

    display = XOpenDisplay(display_name);
```

```
    if (display == (Display *)NULL) {
        printf("could not open display %s\n", display_name);
        exit(1);
    }
    screen = DefaultScreen(display);

window = XCreateSimpleWindow(
        display,                        /* Xlib display structure */
        RootWindow(display, screen),    /* use X root window */
        0, 0,                           /* x, y position */
        1, 1,                           /* width, height */
        1,                              /* border width in pixels */
        BlackPixel(display, screen),    /* border color */
        BlackPixel(display, screen)); /* background color */

    if (window == 0) {
        printf("could not open window %s\n", display_name);
        exit(1);
    }

    sys_st = xgl_open (XGL_SYS_ST_ERROR_DETECTION, TRUE,
                       NULL);

    xgl_x_win.X_display = (void *)display;
    xgl_x_win.X_screen = screen;
    xgl_x_win.X_window = window;

    obj_desc.win_ras.type = XGL_WIN_X;
    obj_desc.win_ras.desc = &xgl_x_win;
    if (!(inq_info = xgl_inquire(sys_st, &obj_desc))) {
      printf("error getting inquiry\n");
      exit(1);
    }

    printf("Inquire Information:\n");

    printf("  %s\n", inq_info->name);
    if (inq_info->dga_flag)
        printf("  Direct Graphics Access\n");
    else
        printf("  Xlib port\n");

    printf("    Accelerated Color Type(s): ");
    if (!inq_info->color_type.index &&
        !inq_info->color_type.rgb)
```

```
    printf("None\n");
else {
    if (inq_info->color_type.index)
        printf("INDEXED ");
    if (inq_info->color_type.rgb)
      printf("RGB ");
    printf("\n");
}
printf("    Depth : %d\n", inq_info->depth);
printf("    Width : %d\n", inq_info->width);
printf("    Height: %d\n", inq_info->height);
if (inq_info->maximum_buffer == 1)
    printf("    Single Buffering\n");
if (inq_info->maximum_buffer == 2)
    printf("    Double Buffering\n");
if (inq_info->maximum_buffer > 2)
    printf("    Multiple Buffering (%d)\n",
                inq_info->maximum_buffer);
if (inq_info->db_buffer_is_copy)
    printf("      Using bitblt transfer\n");
else
    printf("      Using hardware swap\n");
printf("    Point Types:\n");
if (!inq_info->pt_type.pt_dim_2d &&
    !inq_info->pt_type.pt_dim_3d &&
    !inq_info->pt_type.pt_type_int &&
    !inq_info->pt_type.pt_type_float) {
  printf("        None (?)\n");
}
else {
  if (inq_info->pt_type.pt_dim_2d)
      printf("        2D\n");
  if (inq_info->pt_type.pt_dim_3d)
      printf("        3D\n");
  if (inq_info->pt_type.pt_type_int)
      printf("        INT\n");
  if (inq_info->pt_type.pt_type_float)
      printf("        FLOAT\n");
}
if (inq_info->hlhsr_mode == XGL_HLHSR_NONE)
    printf("    Software Zbuffer in host\n");
if (inq_info->hlhsr_mode == XGL_HLHSR_ZBUFFER)
    printf("    Hardware Zbuffer\n");

if (inq_info->picking == XGL_INQ_NOT_SUPPORTED)
    printf("    Picking not supported\n");
```

```
    else if (inq_info->picking == XGL_INQ_SOFTWARE)
        printf("    Picking emulated in software\n");
    else if (inq_info->picking == XGL_INQ_HARDWARE)
        printf("    Picking accelerated through hardware\n");

    if (inq_info->double_buffer == XGL_INQ_NOT_SUPPORTED)
        printf("    Double buffering not supported\n");
    else if (inq_info->double_buffer == XGL_INQ_SOFTWARE)
        printf("    Double buffering emulated in software\n");
    else if (inq_info->double_buffer == XGL_INQ_HARDWARE)
       printf("   Double buffering accelerated through hardware\n");

    if (inq_info->indexed_color == XGL_INQ_NOT_SUPPORTED)
        printf("    Index colors not supported\n");
    else if (inq_info->indexed_color == XGL_INQ_SOFTWARE)
        printf("    Index colors emulated in software\n");
    else if (inq_info->indexed_color == XGL_INQ_HARDWARE)
        printf("    Index colors accelerated through hardware\n");

    if (inq_info->true_color == XGL_INQ_NOT_SUPPORTED)
        printf("    True colors not supported\n");
    else if (inq_info->true_color == XGL_INQ_SOFTWARE)
        printf("    True colors emulated in software\n");
    else if (inq_info->true_color == XGL_INQ_HARDWARE)
        printf("    True colors accelerated through hardware\n");

    if (inq_info->depth_cueing == XGL_INQ_NOT_SUPPORTED)
        printf("    Depth cueing not supported\n");
    else if (inq_info->depth_cueing == XGL_INQ_SOFTWARE)
        printf("    Depth cueing emulated in software\n");
    else if (inq_info->depth_cueing == XGL_INQ_HARDWARE)
        printf("    Depth cueing accelerated through hardware\n");

    if (inq_info->lighting == XGL_INQ_NOT_SUPPORTED)
        printf("    Lighting not supported\n");
    else if (inq_info->lighting == XGL_INQ_SOFTWARE)
        printf("    Lighting emulated in software\n");
    else if (inq_info->lighting == XGL_INQ_HARDWARE)
        printf("    Lighting accelerated through hardware\n");

if (inq_info->shading == XGL_INQ_NOT_SUPPORTED)
        printf("    Shading not supported\n");
    else if (inq_info->shading == XGL_INQ_SOFTWARE)
        printf("    Shading emulated in software\n");
    else if (inq_info->shading == XGL_INQ_HARDWARE)
        printf("    Shading accelerated through hardware\n");
```

```
    if (inq_info->hlhsr == XGL_INQ_NOT_SUPPORTED)
        printf("    Hidden line removal not supported\n");
    else if (inq_info->hlhsr == XGL_INQ_SOFTWARE)
        printf("    Hidden line removal emulated in software\n");
    else if (inq_info->hlhsr == XGL_INQ_HARDWARE)
      printf(" Hidden line removal accelerated through hardware\n");

    if (inq_info->antialiasing == XGL_INQ_NOT_SUPPORTED)
        printf("    Antialiasing not supported\n");
    else if (inq_info->antialiasing == XGL_INQ_SOFTWARE)
        printf("    Antialiasing emulated in software\n");
    else if (inq_info->antialiasing == XGL_INQ_HARDWARE)
        printf("    Antialiasing accelerated through hardware\n");

    if (inq_info->stereo)
        printf("    Hardware stereo support\n");
    else
        printf("    No stereo support\n");

   if (inq_info->extns & XGL_INQ_MULTI_BUFFERING)
       printf("   Multi Buffering supported\n");
   else
       printf("   No Multi Buffering support\n");

    free (inq_info);
}
```

5

Transparent Overlay Windows

The XGL library supports the use of transparent overlay windows. These windows are X windows that allow the user to see through to the underlying window on a per-pixel basis. Transparent overlay windows allow applications to render temporary imagery on complex rendering in the underlying window. Users of an application that provides transparent overlay windows can annotate an image with text or graphical figures, temporarily highlight certain portions of the imagery, or animate figures that appear to move against the background of the imagery. When geometry in the overlay is cleared, any underlying graphics do not need to be regenerated.

Applications can draw into transparent overlay windows with *transparent* paint or *opaque* paint. Pixels rendered transparently have no intrinsic color; they derive their displayed color from the pixels that lie beneath. Pixels rendered opaquely obscure pixels in underlying windows. This allows the application to selectively erase the contents of the overlay window in such a way that the graphics in the underlay window show through. Overlay window background is transparent by default but may be set to other types of background.

An XGL application can render into overlay windows using XGL primitives or Xlib primitives.The XGL library provides Context attributes that enable an XGL application to render XGL geometry into an overlay window using opaque or transparent paint.

The transparent overlay support provided in the XGL library requires the transparent overlay API in the Solaris Overlay extension to the Solaris X server. For details on how to verify that the server includes the overlay extension, see the *Solaris X Window System Developer's Guide*, which is part of the Software Developer Kit AnswerBook.

Performance Cases for Transparent Overlays

If the server has overlay support, application geometry can be rendered to the overlay window using hardware overlay planes or software emulation of overlay functionality. Performance may vary considerably between hardware and software overlay rendering. Although the application cannot choose the rendering path, the following information may help the application understand performance differences.

- Hardware overlay planes and XGL/DGA support – If the hardware device provides hardware overlay planes and and the application chooses the correct visuals, the Solaris server can make use of the hardware overlay functionality for transparent overlay windows. If the device graphics handler implements overlay support, XGL coordinates with the server to provide accelerated rendering into transparent overlay windows using DGA. The ZX and SX accelerators are examples of devices that have hardware overlay planes and DGA rendering.

  Rendering through XGL/DGA to hardware overlay planes provides the best performance for rendering to overlay windows. In addition, DGA rendering to both the overlay and underlay windows ensures that images will touch the same pixels. However, there are restrictions that application programs must follow to enable this case. These restrictions are listed on page 103.

- Hardware overlay planes and Xlib support – If the hardware device provides hardware overlay planes, but the device graphics handler does not implement DGA overlay support, XGL renders to the overlay window using Xlib. Even if DGA support of overlays is available, there may be some cases in which the server refuses to grant the request for the overlay window. For example, the server will always deny access to an overlay Drawable on a window with backing store, and in these cases XGL will use Xlib for overlay rendering. In addition, if the application chooses not to follow the restrictions listed on page 103, Xlib is used to render to the overlay window.

  Rendering through Xlib to hardware overlay planes provides moderate performance. However, when geometry is rendered to the underlay window with DGA and to the overlay window with Xlib, there may be cases in which different pixels are touched.

- Software emulation – Even if the hardware does not have overlay planes, the Solaris server can create and use transparent overlay windows if the server includes overlay visuals. In this case, the server emulates overlay functionality in software, and XGL renders using Xlib.

  The GX and p9000 devices are examples of devices that provide overlay functionality through software emulation. Software emulation of overlay windows provides slower performance for overlay rendering.

Information on whether XGL is using DGA to render to the overlay window is not available from within the application, since `xgl_inquire()` does not currently return information on device support for transparent overlays. If the overlay rendering is slow, you may be able to infer that software emulation is

being used. For more accurate information, see your frame buffer documentation for the capabilities of the specific hardware devices, and try to make your application's use of overlay windows as portable as possible. Information on portability issues is provided in the following sections.

Creating an Overlay Window

An application creates an overlay window using the Solaris overlay API `XSolarisOvlCreateWindow`. This routine behaves exactly as `XCreateWindow` except that the resulting window is an overlay window. The newly created window can be rendered into with both opaque and transparent paint. An overlay window must be the child of another window, called an underlay window.

As part of creating an overlay window, the application must search for appropriate visuals for the underlay and overlay windows. Each X screen supporting the overlay extension defines a set of overlay visuals whose windows are best suited for use as children of underlay windows. For each underlay visual, there is a set of optimal overlay visuals. Together, all combinations of underlay visuals and their optimal overlay visuals form the set of optimal overlay/underlay pairs for that screen. The overlay and underlay visuals of an optimal pair are partners of each other.

Choosing Overlay Visuals

The Solaris overlay routine `XSolarisOvlSelectPartner` allows the application to select for a particular underlay visual an optimal overlay visual that meets certain criteria. This routine searches through the optimal partners of the given visual, applying the criteria specified. It returns a success or failure status depending on whether it finds a visual that meets the criteria.

The `XSolarisOvlSelectPartner` routine is designed to help applications construct portable overlay applications. Because of the wide variety of hardware devices, constructing a portable overlay application may be difficult unless care is taken with the choice of visuals. For example, for a given type of underlay window, some devices can provide overlay windows with high-performance rendering, while other devices provide the same type of overlay window but with slower rendering. The `XSolarisOvlSelectPartner` routine enables the application to negotiate with the system for a suitable

overlay/underlay visual pair. For more information on the `XSolarisOvlSelectPartner` routine, see the *Solaris X Window System Developer's Guide*.

Tips on Choosing Overlay Visuals for DGA Transparent Overlays

Some SMCC 3D accelerators, for example, the ZX and the SX, provide accelerated transparent overlay functionality using hardware overlay planes and DGA. However, the application must follow several restrictions to use DGA transparent overlays for XGL rendering:

1. The primary restriction is that the overlay window must always spatially occupy the same pixels as the underlay window. It is the responsibility of the XGL application to maintain this relationship.
2. The overlay window must always be a descendent of the original underlay window and must not be reparented to a different underlay window.
3. To get the appropriate overlay visuals for a frame buffer, use the Solaris overlay API `XSolarisOvlSelectPartner` with a selection type of `XSolarisOvlSelectBestOverlay`. See the program on page 106 for an example of the use of this routine.
4. No windows may be inserted between the overlay and underlay windows.

Note that these restrictions only apply when overlay windows are handled by the hardware device. However, as noted above, the application has no way of determining whether the device that the application is currently rendering on has DGA transparent overlay support because the XGL operator `xgl_inquire()` does not currently return this information. Therefore, to ensure portability, the application may want to abide by these hardware-specific restrictions.

Creating XGL Window Rasters for Overlay Windows

When an application has determined that the server provides overlay support and that the appropriate visuals are available, it can create XGL window rasters for the XGL overlay and underlay windows by copying the X information for these windows into XGL window descriptor structures and creating XGL Window Raster objects. The XGL Window Raster objects for overlay windows are created like any other XGL Window Raster.

Note – To improve the portability of your application and to minimize color flashing, use color maps with the same colors in both the overlay and underlay window color maps. If your application uses pixel-sharing overlay/underlay pairs, create a single color map for both windows. For more information on color maps and overlay windows, see the *Solaris X Window System Developer's Guide*.

Rendering to an Overlay Window Using the XGL API

In the transparent overlay model, the pixels, or *paint*, has a characteristic called the paint type. The paint type can be either opaque or transparent. In an XGL application, the paint type is set on the Context object using the `XGL_CTX_PAINT_TYPE` attribute. This attribute can be set to `XGL_PAINT_OPAQUE`, which is the default value, or `XGL_PAINT_TRANSPARENT`. The attribute `XGL_CTX_PAINT_TYPE` is in effect only when the Device associated with a Context is an overlay window; otherwise, `XGL_CTX_PAINT_TYPE` is ignored, and the paint type is opaque.

Most XGL primitives can render pixels into overlay windows. Note, however, the following exceptions:

- `xgl_context_accumulate()` – Only the color information of the draw buffer is accumulated into an overlay window that is the destination buffer of the accumulation. The paint type is not changed.
- `xgl_context_copy_buffer()` – If the Device associated with the Context is an overlay window, the color information of the source raster is copied to the destination raster as follows:
  - If the source raster is an overlay window and the destination raster is also an overlay window, the color and paint type are copied to the destination.
  - If the source raster is an overlay window but the destination raster is not an overlay window, the color is copied to the destination. If the source paint type is transparent, the color is undefined.

  In addition, if the source raster is not an overlay window but the destination raster is an overlay window, the pixel color is copied, and the paint type of the destination is not changed.

Clearing the Overlay Window Using the XGL API

The XGL attribute `XGL_CTX_NEW_FRAME_PAINT_TYPE` specifies the paint type for new frame clear actions on overlay windows. This attribute is applied when an application calls `xgl_context_new_frame()` and `XGL_CTX_NEW_FRAME_ACTION` includes `XGL_CTX_NEW_FRAME_CLEAR`. For overlay window clears, the color information is set to the background color, and the paint type is set to the value of `XGL_CTX_NEW_FRAME_PAINT_TYPE`. By default, the new frame paint type is `XGL_PAINT_TRANSPARENT`, which clears the overlay window.

Note that when the overlay is handled by the hardware device, erasing the overlay graphics does not damage the underlying graphics or generate an Expose event. This provides a performance advantage when the underlying graphics is complex and requires much time to repaint. However, when the overlay is in software, erasing the overlay graphics may generate an Expose event.

Transparent Overlay Example Program

The example program `transparency_ovl.c` shows the use of XGL primitives to render to overlay and underlay windows. The program draws an XGL multisimple polygon in the underlay window. It provides a pop-up window whose buttons the user can press to draw XGL text, markers, polylines, or filled polygons in the overlay window. The example program works on hardware that has DGA overlay support, for example the ZX and the SX, as well as on the GX, which does not have hardware overlay planes.

The program uses `XSolarisOvlSelectPartner` to choose the best overlay and underlay visuals for the screen. If the appropriate visuals are available, the program creates the overlay window with `XSolarisOvlCreateWindow` and sets the background of the overlay window to transparent with the `XSolarisOvlSetWindowTransparent` routine. Then the program opens XGL and sets up the XGL window raster descriptor structures for the overlay and underlay windows. The program sets up XGL Contexts for the windows and defines the paint type in the overlay Context as opaque using `XGL_CTX_PAINT_TYPE` set to `XGL_PAINT_OPAQUE`. The pop-up window buttons illustrate clearing the overlay window to transparency by calling `xgl_context_new_frame()` for the overlay Context as well as undrawing the overlay geometry by setting the overlay window paint type to `XGL_PAINT_TRANSPARENT`.

The sample code below is an excerpt from the complete `transparency_ovl.c` program. See the example program in `$XGLHOME/demo/examples` for the code preceding main.

Note – In this program, color flashing will occur in windows with a depth of 8 bits. You can minimize the color map flashing by offsetting the color map entries. See the example program `color_cube.c` on page 141. Color map flashing does not occur in 24-bit windows.

Code Example 5-5 Transparent Overlay Example

```
main (argc, argv)

    int argc;
    char *argv[];
{
    int     i;
    int     depth;
    int     screen;
    int     buffers;
    int     rotate;
    int     data_count;
    float   x_diff;
    float   y_diff;
    float   x_deg;
    float   y_deg;
    char   *version;
    long    plane_masks[2];
    long    event_mask;
    unsigned long failures;
    Arg     args[8];
    Widget  toplevel;
    Widget  control1;
    Widget  control2;
    Widget  button1;
    Widget  button2;
    Widget  button3;
    Widget  button4;
    Widget  button5;
    Widget  button6;
    Widget  button7;
    Widget  button8;
    Widget  button9;
```

```
    Widget  popup_shell;
    Widget  popup_shell2;
    Widget  underlay;
    Window  ol_window;
    Display *display = NULL;
    Visual  *visual;
    Visual  *default_visual;
    Visual  *ol_visual;
    Screen  *screen_ptr;
    XtAppContext app_context;
    XColor  gray;
    XColor  light_gray;
    XEvent  event;
    XVisualInfo info;
    XSolarisOvlVisualCriteria criteria;
    XSolarisOvlSelectStatus status;
    XSetWindowAttributes attr;
    XClientMessageEvent *client_message =
                        XClientMessageEvent*) &event;
    XVisualInfo visual_info;
    Xgl_segment segment;

/*
 *  Initialize the Openlook toolkit.
 */
    OlToolkitInitialize (NULL);

/*
 *  Create the top level shell.
 */
    toplevel = XtVaAppInitialize (&app_context,
                                  "",
                                  NULL,
                                  0,
                                  (Cardinal*)&argc,
                                  argv,
                                  NULL,
                                  NULL);
    ul_display = (Display*) XtDisplay (toplevel);
    display = ul_display;

/*
 *  Get the screen number.
 */
    screen = DefaultScreen (display);
    root = RootWindow (display, screen);
```

```
/*
 *  Get the default colormap.
 */
    default_colormap = DefaultColormap (display, screen);

/*
 *  Create a popup shell to hold the controls.
 */
    XtSetArg (args[0], XtNx, 100);
    XtSetArg (args[1], XtNy, 100);
    popup_shell = XtCreatePopupShell ("  ",
                                      topLevelShellWidgetClass,
                                      toplevel,
                                      args,
                                      2);

    control1 = XtVaCreateManagedWidget ("control1",
                                        controlAreaWidgetClass,
                                        popup_shell,
                                       XtNlayoutType, OL_FIXEDROWS,
                                        XtNmeasure, 2,
                                        NULL);

    control2 = XtVaCreateManagedWidget ("control",
                                        controlAreaWidgetClass,
                                        control1,
                                       XtNlayoutType, OL_FIXEDCOLS,
                                        XtNmeasure, 1,
                                        NULL);

/*
 *  Create the buttons.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */

/*
 *  Attach the callbacks to the buttons.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */

/*
 * Pop the shell.
 */
```

```
    XtRealizeWidget (popup_shell);
    XtPopup (popup_shell, XtGrabNone);

/*
 *  Create the colormap.
 */
    if (XMatchVisualInfo (display, screen, 24, TrueColor,
                          &visual_info)) {

   /*
    *  Set the visual, color flag and depth.
    */
       visual = visual_info.visual;
       truecolor = TRUE;
       depth = 24;

   /*
    *  Create a colormap for the 24-bit visual.
    */
       colormap = XCreateColormap (display,
                                   RootWindow (display, screen),
                                   visual,
                                   AllocNone);
    }
    else {

   /*
    *  For index color, use the default visual.
    */
       visual = DefaultVisual (display, screen);

   /*
    *  Set the color flag, colormap and depth.
    */
       truecolor = FALSE;
       colormap = default_colormap;
       depth = 8;
    }
    XSync (display, FALSE);

/*
 *  Create a popup shell to hold the graphics widget.
 *  Specify the visual, depth and colormap.
 */
    XtSetArg (args[0], XtNvisual, visual);
    XtSetArg (args[1], XtNdepth, depth);
```

```
    XtSetArg (args[2], XtNcolormap, colormap);
    XtSetArg (args[3], XtNx, 250);
    XtSetArg (args[4], XtNy, 100);
    popup_shell2 = XtCreatePopupShell ("XGL with Overlay",
                                       topLevelShellWidgetClass,
                                       toplevel,
                                       args,
                                       5);

/*
 *  Create a "canvas" and attach it to the popup shell.
 */
    underlay = XtVaCreateManagedWidget ("underlay",
                                        drawAreaWidgetClass,
                                        popup_shell2,
                                        XtNwidth,  400,
                                        XtNheight, 400,
                                        NULL);
/*
 *  Pop the shell.
 */
    XtRealizeWidget (popup_shell2);
    XtPopup (popup_shell2, XtGrabNone);
    ul_window = (Xgl_usgn32) XtWindow (underlay);

/*
 *  Create the overlay window.
 *  Choose the best overlay visual for the underlay window.
 */
    criteria.hardCriteriaMask = 0;
    criteria.softCriteriaMask = XSolarisOvlPreferredPartner;
    status = XSolarisOvlSelectPartner (display,
                                       screen,
                                       XVisualIDFromVisual(visual),
                                      XSolarisOvlSelectBestOverlay,
                                       2,
                                       &criteria,
                                       &info,
                                       &failures);

    if (status != XSolarisOvlSuccess) {
       printf ("Cannot find preferred partner overlay visual...\n");
        if (status != XSolarisOvlQualifiedSuccess) {
            printf("No suitable visuals found:  Exiting.\n");
            exit (0);
        } else
```

```
            printf("Using closest matching visual for overlay.\n");
    }

/*
 * Create the colormap for the overlay window.
 */
    ol_visual = info.visual;

    ol_colormap = XCreateColormap(display, RootWindow(display,
                  screen), ol_visual, AllocNone);

    XAllocColorCells (display, ol_colormap, (Bool) TRUE,
                      plane_masks, 0,
                      ol_color_mapping, 5);

    xcolor[0].flags = DoRed | DoGreen | DoBlue;
    xcolor[0].pixel = ol_color_mapping[0];
    xcolor[0].red   = 65535;
    xcolor[0].green = 65535;
    xcolor[0].blue  = 65535;

    xcolor[1].flags = DoRed | DoGreen | DoBlue;
    xcolor[1].pixel = ol_color_mapping[1];
    xcolor[1].red   =     0;
    xcolor[1].green =     0;
    xcolor[1].blue  =     0;

    xcolor[2].flags = DoRed | DoGreen | DoBlue;
    xcolor[2].pixel = ol_color_mapping[2];
    xcolor[2].red   = 65535;
    xcolor[2].green =     0;
    xcolor[2].blue  =     0;

    xcolor[3].flags = DoRed | DoGreen | DoBlue;
    xcolor[3].pixel = ol_color_mapping[3];
    xcolor[3].red   =     0;
    xcolor[3].green = 65535;
    xcolor[3].blue  =     0;

    xcolor[4].flags = DoRed | DoGreen | DoBlue;
    xcolor[4].pixel = ol_color_mapping[4];
    xcolor[4].red   =     0;
    xcolor[4].green =     0;
    xcolor[4].blue  = 65535;

    XStoreColors (display, ol_colormap, xcolor, 5);
```

```
    XSync (display, FALSE);

    attr.border_pixel = xcolor[0].pixel;
    attr.colormap = ol_colormap;
    attr.event_mask = ExposureMask;

/*  Create the overlay window -
 *  1. The parent underlay window is ul_window.
 *  2. The postion is 0,0 relative to the underlay window.
 *  3. The overlay window is the same size (400,400) as the
 *   underlay window.
 */
    ol_window = XSolarisOvlCreateWindow(display,
                                  ul_window,
                                  0, 0,
                                  400, 400,
                                  0,
                                  8,
                                  InputOutput,
                                  ol_visual,
                                  CWEventMask |
                                CWBorderPixel |
                                  CWColormap,
                                  &attr);

/* Set the overlay window background to be transparent. */

    XSolarisOvlSetWindowTransparent (display, ol_window);

/*
 * Define an 8-bit colormap if the underlay window is a
 * Pseudocolor visual.
 */
    if (!truecolor) {
       XAllocColorCells (display, default_colormap,
                         (Bool) TRUE, plane_masks, 0,
                         color_mapping, 128);

       xcolor[0].flags = DoRed | DoGreen | DoBlue;
       xcolor[0].pixel = color_mapping[0];
       xcolor[0].red   = 65535;
       xcolor[0].green = 65535;
       xcolor[0].blue  = 65535;

       xcolor[1].flags = DoRed | DoGreen | DoBlue;
```

```
        xcolor[1].pixel = color_mapping[1];
        xcolor[1].red   =     0;
        xcolor[1].green =     0;
        xcolor[1].blue  =     0;

        xcolor[2].flags = DoRed | DoGreen | DoBlue;
        xcolor[2].pixel = color_mapping[2];
        xcolor[2].red   = 65535;
        xcolor[2].green =     0;
        xcolor[2].blue  =     0;

        xcolor[3].flags = DoRed | DoGreen | DoBlue;
        xcolor[3].pixel = color_mapping[3];
        xcolor[3].red   =     0;
        xcolor[3].green = 65535;
        xcolor[3].blue  =     0;

        xcolor[4].flags = DoRed | DoGreen | DoBlue;
        xcolor[4].pixel = color_mapping[4];
        xcolor[4].red   =     0;
        xcolor[4].green =     0;
        xcolor[4].blue  = 65535;

        for (i=5; i<128; i++) {
           xcolor[i].flags = DoRed | DoGreen | DoBlue;
           xcolor[i].pixel = color_mapping[i];
           xcolor[i].red   = 0;
           xcolor[i].green = (i - 5) * 537;
           xcolor[i].blue  = 0;
        }

        segment.offset = 5;
        segment.length = 123;

        XStoreColors (display, default_colormap, xcolor, 128);
        XSync (display, FALSE);
    }

/*
 *  Open xgl.
 */
    sys_state = xgl_open (XGL_SYS_ST_ERROR_DETECTION, FALSE, NULL);

/*
 *  Exit if unable to open xgl.
 */
```

```
    if (sys_state == NULL) {
       printf ("Error:  Failed to open XGL. Exiting\n");
       exit (0);
    }

/*
 *  Get the xgl version number.
 */
    xgl_object_get (sys_state, XGL_SYS_ST_VERSION, &version);

/*
 *  Print the xgl version number.
 */
    printf ("\n");
    printf ("XGL Version = %s", version);

/*
 *  Print the color mode.
 */
    if (truecolor)
       printf ("Color mode = TrueColor\n");
    else
       printf ("Color mode = Index\n");

/*
 *  Set up the xgl descriptor structures for the XGL window
 *  rasters for the underlay and overlay windows.
 */
    ul_x_win.X_display = ul_display;
    ul_x_win.X_window  = ul_window;
    ul_x_win.X_screen  = DefaultScreen (ul_display);
    ul_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
    ul_desc.win_ras.desc = &ul_x_win;

    ol_x_win.X_display = display;
    ol_x_win.X_window  = ol_window;
    ol_x_win.X_screen  = DefaultScreen (display);
    ol_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
    ol_desc.win_ras.desc = &ol_x_win;

/*
 *  Define command line options that allow the user to specify
 *  xlib rendering for the underlay or overlay window or both.
 *  Note that these options may not work for all frame buffers.
 */
```

```
    {
       int             i, n;
       char           *s;
       for (i = 1; i < argc; i++) {
              s = argv[i];
              n = strlen(s);
              if (!strncmp("-ovlxlib", s, n)) {
              ol_desc.win_ras.type =
                  XGL_WIN_X | XGL_WIN_X_PROTO_XLIB;
              } else if (!strncmp("-ulxlib", s, n)) {
                  ul_desc.win_ras.type =
                  XGL_WIN_X | XGL_WIN_X_PROTO_XLIB;
              }
       }
    }

/*
 *  Map the windows.
 */
    XSetWMColormapWindows(ul_display,
           XtWindowOfObject(popup_shell2), &ol_window, 1);
    XMapWindow (ul_display, ul_window);
    XMapWindow (display, ol_window);
    do {
        XNextEvent (ul_display, &event);
    } while (event.type != Expose);

/*
 *  Select input.
 */
    XSelectInput (ul_display, ul_window, ExposureMask |
                                         ButtonPressMask |
                                         ButtonReleaseMask |
                                         ButtonMotionMask |
                                         StructureNotifyMask);

/*
 *  Create the xgl colormap object used by the overlay raster.
 */
   ol_cmap = xgl_object_create (sys_state, XGL_CMAP, NULL,
                               XGL_CMAP_NAME, ol_colormap,
                               XGL_CMAP_COLOR_TABLE_SIZE, 5,
                               NULL);

/*
 *  Create the xgl colormap object to be used by the underlay
```

```
 *  raster when the raster is 8 bit.
 */
   if (!truecolor) {
       cmap = xgl_object_create (sys_state, XGL_CMAP, NULL,
                                 XGL_CMAP_NAME, default_colormap,
                                 XGL_CMAP_COLOR_TABLE_SIZE, 128,
                                 NULL);
   }

/*
 *  Create the underlay raster.
 */
   if (truecolor) {
      raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ul_desc,
                                 XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                 XGL_WIN_RAS_BUFFERS_REQUESTED, 2,
                                 XGL_WIN_RAS_BUF_DRAW,    1,
                                 XGL_WIN_RAS_BUF_DISPLAY, 0,
                                 NULL);

      dir_light_color.rgb.r = 1.0;
      dir_light_color.rgb.g = 1.0;
      dir_light_color.rgb.b = 1.0;

      amb_light_color.rgb.r = 1.0;
      amb_light_color.rgb.g = 1.0;
      amb_light_color.rgb.b = 1.0;

      specular_color.rgb.r = 1.0;
      specular_color.rgb.g = 1.0;
      specular_color.rgb.b = 1.0;
   }
   else {
      raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ul_desc,
                                 XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                                 XGL_DEV_COLOR_MAP, cmap,
                                XGL_WIN_RAS_PIXEL_MAPPING, color_mapping,
                                 XGL_WIN_RAS_BUFFERS_REQUESTED, 2,
                                 XGL_WIN_RAS_BUF_DRAW,    1,
                                 XGL_WIN_RAS_BUF_DISPLAY, 0,
                                 NULL);

      dir_light_color.gray = 1.0;
      amb_light_color.gray = 1.0;
      specular_color.index = 0;
```

```
      xgl_object_set (cmap,
                      XGL_CMAP_RAMP_NUM, 1,
                      XGL_CMAP_RAMP_LIST, &segment,
                      NULL);
   }

/*
 *  Create the overlay raster.
 */
   ol_raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ol_desc,
                       XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                       XGL_DEV_COLOR_MAP, ol_cmap,
                       XGL_WIN_RAS_PIXEL_MAPPING, ol_color_mapping,
                       NULL);

   if (ol_raster == NULL) {
      printf ("Cannot create overlay window raster\n");
      exit (0);
   }

/*
 *  Create the xgl contexts.
 */
    ul_ctx = xgl_object_create (sys_state,XGL_3D_CTX, NULL,
              XGL_CTX_DEVICE, raster,
              XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASTI,
              XGL_CTX_VDC_WINDOW, &vdc_window,
              XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
              XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL,
              XGL_CTX_ATEXT_CHAR_HEIGHT, 0.5,
              XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
              XGL_LIGHT_ENABLE_COMP_AMBIENT   |
              XGL_LIGHT_ENABLE_COMP_SPECULAR  |
              XGL_LIGHT_ENABLE_COMP_DIFFUSE,
              XGL_3D_CTX_SURF_FRONT_AMBIENT,  0.30,
              XGL_3D_CTX_SURF_FRONT_DIFFUSE,  1.00,
              XGL_3D_CTX_SURF_FRONT_SPECULAR, 1.00,
              XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, 32.0,
              XGL_3D_CTX_SURF_FRONT_ILLUMINATION,
                          XGL_ILLUM_PER_FACET,
              XGL_3D_CTX_HLHSR_MODE, XGL_HLHSR_NONE,
              XGL_CTX_NEW_FRAME_ACTION,
                       XGL_CTX_NEW_FRAME_CLEAR |
                       XGL_CTX_NEW_FRAME_PAINT_TYPE_ACTION |
                       XGL_CTX_NEW_FRAME_SWITCH_BUFFER,
              XGL_3D_CTX_LIGHT_SWITCHES, light_switches,
```

```
                XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &specular_color,
                XGL_3D_CTX_LIGHT_NUM, 3,
                XGL_3D_CTX_SURF_FACE_CULL, XGL_CULL_BACK,
                XGL_CTX_BACKGROUND_COLOR, &ul_black,
                NULL);

    if (!truecolor){
       surf_color.index = 127;
       xgl_object_set (ul_ctx, XGL_CTX_SURF_FRONT_COLOR,
                     &surf_color, NULL);
    }

/*  Note that overlay window clears to a transparent
 *  background by default.
 */
    ol_ctx = xgl_object_create (sys_state, XGL_3D_CTX, NULL,
                XGL_CTX_DEVICE, ol_raster,
                XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASTI,
                XGL_CTX_VDC_WINDOW, &vdc_window,
                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL,
                XGL_CTX_ATEXT_CHAR_HEIGHT, 0.05,
                XGL_3D_CTX_SURF_FRONT_ILLUMINATION, XGL_ILLUM_NONE,
                XGL_3D_CTX_HLHSR_MODE, XGL_HLHSR_NONE,
                XGL_CTX_PAINT_TYPE, XGL_PAINT_OPAQUE,
                XGL_CTX_STEXT_COLOR, &ol_white,
                XGL_CTX_MARKER_COLOR, &ol_white,
                XGL_CTX_LINE_COLOR, &ol_white,
                XGL_CTX_SURF_FRONT_COLOR, &ol_white,
                XGL_CTX_NEW_FRAME_ACTION, XGL_CTX_NEW_FRAME_CLEAR,
                XGL_CTX_SURF_FRONT_COLOR_SELECTOR,
                            XGL_SURF_COLOR_CONTEXT,
                NULL);

/*
 *  Get the light objects.
 */
   xgl_object_get (ul_ctx, XGL_3D_CTX_LIGHTS, lights);

/*
 *  Define the lights.
 */
   xgl_object_set (lights[0],
                   XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
                   XGL_LIGHT_COLOR, &amb_light_color,
                   NULL);
```

```
    xgl_object_set (lights[1],
                    XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                    XGL_LIGHT_DIRECTION, &light_dir,
                    XGL_LIGHT_COLOR, &dir_light_color,
                    NULL);

    xgl_object_set (lights[2],
                    XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                    XGL_LIGHT_DIRECTION, &light_dir2,
                    XGL_LIGHT_COLOR, &dir_light_color,
                    NULL);

/*
 *  Clear the screen.
 */
    xgl_context_new_frame (ul_ctx);

/*
 *  Inquire about XGL attributes.
 */
    inq = xgl_inquire (sys_state, &ul_desc);

/*
 *  Check for DGA.
 */
    if (inq->dga_flag) {
       printf ("DGA = YES\n");
    }
    else {
       printf ("DGA = NO\n");
    }

/*
 *  Print the name of the frame buffer.
 */
    printf ("Frame Buffer = %s\n", inq->name);

/*
 *  Print the number of buffers allocated.
 */
    xgl_object_get (raster, XGL_WIN_RAS_BUFFERS_ALLOCATED,
                    &buffers);
    printf ("Buffers = %d\n", buffers);
    printf ("\n");
```

```
/*
 *  Initialize the data.
 */
    data_init ();

/*
 *  Get the global model transform.
 */
    xgl_object_get (ul_ctx, XGL_CTX_GLOBAL_MODEL_TRANS,
                    &global_model_trans);

/*
 *  Render the object.
 */
    ul_traverse ();
    ul_traverse ();

/*
 *  Handle the events.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */


}
```

For the subroutines and button callbacks in the remainder of the program, see the program transparency_ovl.c in $XGLHOME/demo/examples.