Primitives and Graphics Context Attributes

This chapter discusses the XGL primitives and their associated Graphics Context attributes. It includes information on the following topics:

- XGL drawing primitive data structures
- Annotated primitives and text
- Raster-level primitives
- Accumulation buffers
- Examples demonstrating primitives

Introduction to XGL Drawing Primitives

Primitives are the basic graphics drawing routines that render geometry on a hardware device. XGL primitives render relatively simple geometric entities, such as lines, circles, arcs, and rectangles, and more complex entities, such as B-spline surfaces, quadrilateral mesh, and triangle strips. Geometry is rendered via the Context object; for this reason, the drawing primitives are considered to be Context operators. The dimension of the rendered geometry is determined by the Context object: 2D geometry requires 2D input data and is rendered through a 2D Context, and 3D geometry requires 3D input data and is rendered through a 3D Context. Most XGL primitives can be used with both 2D and 3D Context objects.

8

As in basic geometry, XGL deals with three kinds of geometric entities: points, lines, and surfaces. The primitives that XGL provides to draw these entities are summarized below and briefly described in Table 8-2 on page 179.

- Point primitive

  To render points and symbols, XGL provides the multimarker primitive. `xgl_multimarker()` takes a list of 2D or 3D points for the locations of the markers and then draws the specified marker symbol at those positions.

- Line primitives

  To render lines, XGL provides the multipolyline primitive. `xgl_multipolyline()` draws a list of unconnected 2D or 3D polylines, where each polyline is a series of line segments.

  To render curves, XGL provides a B-spline curve primitive. `xgl_nurbs_curve()` draws 2D or 3D non-uniform, rational or non-rational B-spline curves. The points of the B-spline point list are interpreted as the set of control points for the curve. Context attributes control the smoothness and other characteristics of the curve.

- Surface primitives

  A number of XGL primitives render surfaces (which are also known as filled areas, polygons, or closed regions). The surface perimeters defined by the geometry in the point list enclose an area (surface) that can be filled with color, patterns, or texture. Surface primitives can be rendered with shading or lighting, and the front and back faces of surfaces can be determined and rendered with different color, patterns, illumination characteristics, or texture. The following XGL primitives render surfaces:

Table 8-1 XGL Surface Primitives

Primitive	Description
`xgl_multirectangle()`	Renders a list of 2D or 3D rectangles.
`xgl_multicircle()`	Renders a list of 2D or 3D circles.
`xgl_multiarc()`	Renders a list of 2D or 3D arcs. 3D arcs are approximated as polygons.
`xgl_polygon()`	Renders a single-surface (planar) 2D or 3D polygon that can be defined by more than one boundary; a polygon can also be defined with holes.

Table 8-1 XGL Surface Primitives *(Continued)*

Primitive	Description
`xgl_multi_simple_polygon()`	Renders a list of 2D or 3D single-boundary polygons.
`xgl_multi_elliptical_arc()`	Renders a list of 3D elliptical arcs.
`xgl_triangle_strip()`	Renders a 3D triangle strip. Triangle strips, triangle lists, and quadrilateral meshes reduce the amount of geometry transferred to XGL by using individual points in more than one surface definition, if the surfaces are connected along their boundaries.
`xgl_triangle_list()`	Renders a 3D triangle list. Triangle lists can consist of connected triangles arranged as a triangle strip, connected triangles arranged as a triangle star, or unconnected triangles.
`xgl_quadrilateral_mesh()`	Renders a 3D quadrilateral mesh.
`xgl_nurbs_surface()`	Renders a NURBS surface as a set of triangles. The NURBS surface can be trimmed.

- Text primitives

  XGL provides two primitives to render stroke text and an associated primitive to determine the rectangle that encompasses a text string. The stroke text primitive, `xgl_stroke_text()`, renders 2D or 3D text as a set of polylines. The annotation text primitive, `xgl_annotation_text()`, also renders 2D or 3D text but generates the characters in a plane parallel to the display surface.

Note – Bitmap fonts can be rendered using the raster operators `xgl_context_copy_buffer()` and `xgl_image()`. For more information on raster text, see “Raster Text” on page 346.

Twelve of the available primitives are illustrated in Figure 8-1 on page 178.

8

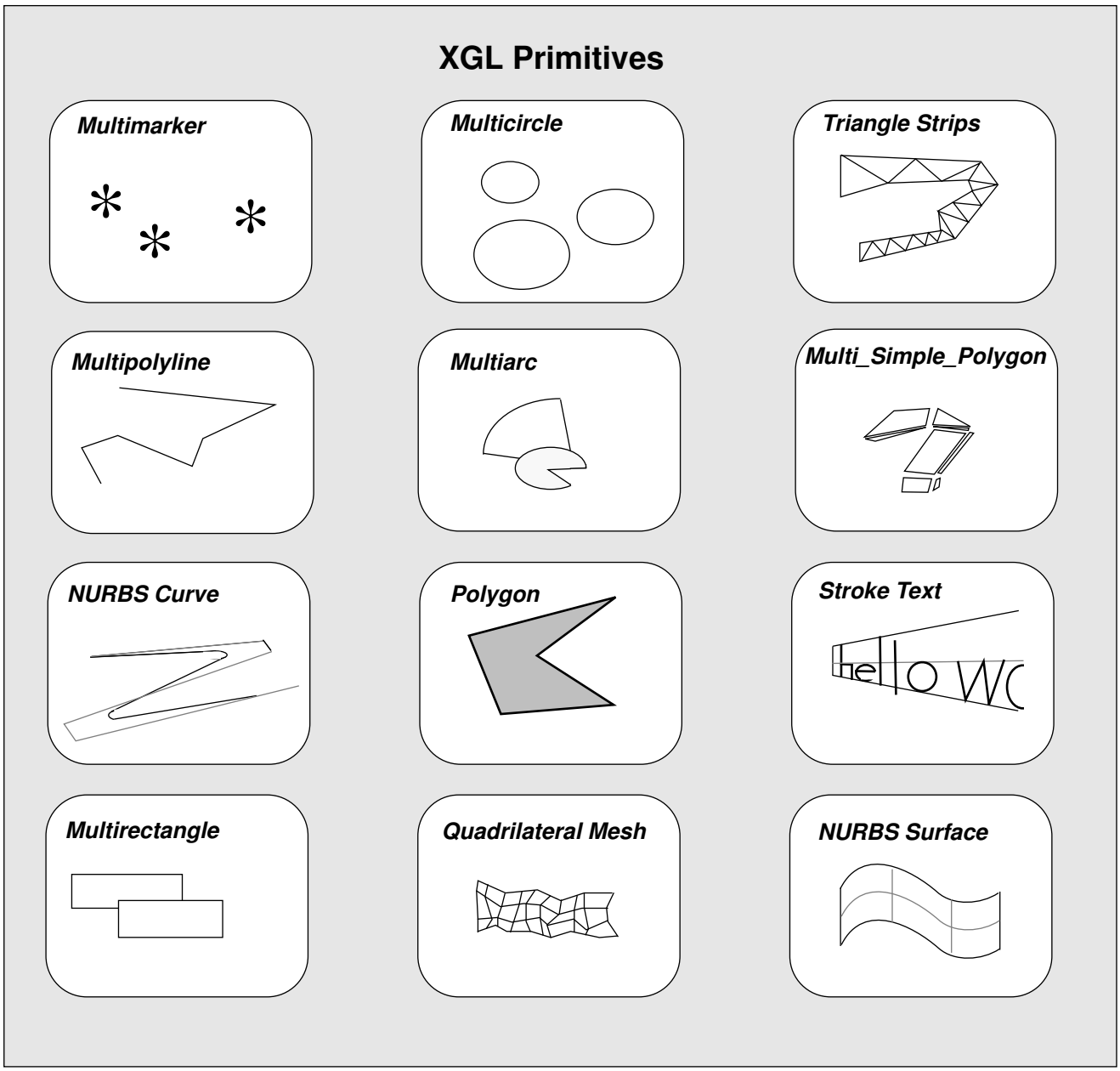

Figure 8-1 XGL Drawing Primitives Illustrated

Description of Drawing Primitives

Table 8-2 provides an overview of the input parameters required for each of the XGL geometric primitive operators. For more information on the input parameters for each operator, see the appropriate manual page in the *XGL Reference Manual*.

Table 8-2 Geometric Primitive Operators *(1 of 4)*

Operator	Description
void **xgl_annotation_text** (Xgl_ctx ctx, void *str, Xgl_pt_f*d *ref_pos, Xgl_pt_f*d *ann_pos);	Draws a string indexed by the characters in `str` using the Context `ctx`. Annotation text primitives differ from text primitives in that the plane upon which the characters are generated is always parallel to the display surface. Annotation text primitives are planar primitives. The Z-coordinate of the annotation point defines the depth of the annotation text.
void **xgl_multiarc** (Xgl_ctx ctx, Xgl_arc_list *arc_list);	Draws a list of arcs of a circle according to the graphics state described in the 2D or 3D Context. The arc traverses a counterclockwise (CCW) path from starting angle to stopping angle when the *x*-axis points right and the *y*- axis points up; this path is clockwise (CW) when the *x*-axis points right and the *y*-axis points down. The argument `arc_list` is a pointer to the list of arcs to draw. Each arc in the list is transformed by the current transformation matrix and drawn in the plane of the view surface.
void **xgl_multicircle** (Xgl_ctx ctx, Xgl_circle_list *circle_list);	Draws a list of circles according to the graphics state described in the Context `ctx`. The argument `circle_list` is a pointer to a list of circles to draw.
void **xgl_multi_elliptical_arc** (Xgl_ctx ctx, Xgl_ell_list *ell_list);	Draws a list of 3D elliptical arcs according to the graphics state described in the 3D Context. The `ell_list` argument is a pointer to the list of elliptical arcs to draw. Each elliptical arc in the list is transformed by the current transformation matrix and drawn in the plane described by the two direction vectors provided with each arc.

Table 8-2 Geometric Primitive Operators *(2 of 4)*

Operator		Description
void **xgl_multimarker** (Xgl_ctx Xgl_pt_list	 ctx, *pl);	Draws markers at a given list of 2D or 3D points. Characteristics of the markers are determined by the graphics state of the Context `ctx`. The argument `pl` is a pointer to the list of points where the markers are drawn.
void **xgl_multipolyline** (Xgl_ctx Xgl_bbox Xgl_usgn32 Xgl_pt_list	 ctx, *bbox, num_pt_lists, pl[]);	Draws a list of 2D or 3D unconnected polylines. Each point list describes one polyline. The argument `pl` is the first element in the array of point list structures that specify the vertices of the polylines.
void **xgl_multirectangle** (Xgl_ctx Xgl_rect_list	 ctx, *rect_list);	Draws a list of rectangles using the current surface attributes. The `rect_list` argument is a pointer to the list of rectangles to draw.
void **xgl_multi_simple_polygon**(Xgl_ctx Xgl_facet_flags Xgl_facet_list Xgl_bbox Xgl_usgn32 Xgl_pt_list	 ctx, flags, *facets, *bbox, num_pt_lists, pl[]);	Draws a set of simple polygons according to the parameter list and the graphics state described in a 2D or 3D Context. The `pl` argument is the first element of an array of point structures, each of which specifies the vertices of a simple polygon. The `facet` argument is a pointer to facet information about individual polygons.
void **xgl_nurbs_curve** (Xgl_ctx Xgl_nurbs_curve Xgl_bounds_f1d Xgl_curve_color_spline	 ctx, *curve, *range, *color_spline);	Draws a non-uniform, B-Spline (NURBS) curve. `curve` is a pointer to a structure defining the order of the curve, the number of knots, the knot vector, and a list of control points for the curve. The `range` argument is a pointer to a structure specifying which portion of the curve is actually displayed.

Table 8-2 Geometric Primitive Operators *(3 of 4)*

Operator	Description
void **xgl_nurbs_surface** (Xgl_ctx ctx, Xgl_nurbs_surf *surface, Xgl_trim_loop_list *trimming, Xgl_nurbs_surf_simple_geom *hints, Xgl_surf_color_spline *color_spline, Xgl_surf_data_spline_list *data_splines);	Draws a NURBS surface according to the parameter list and the graphics state described in a 3D Context. The surface is defined by a two-dimensional grid of control points, knot sequences in u,v parameter space, and trimming information specifying which portion of parameter space is actually rendered. Optional information can define the original geometry.
void **xgl_polygon** (Xgl_ctx ctx, Xgl_facet_type facet_type, Xgl_facet *facet, Xgl_bbox *bbox, Xgl_usgn32 num_pt_lists, Xgl_pt_list pl[]);	Draws a single polygon defined according to the parameter list and the graphics state defined in the Context `ctx`, which may be 2D or 3D. The `num_pt_lists` argument describes the number of boundaries in the multibounded polygon. The `pl` argument is the first element of an array of point lists. Each point list specifies the vertices of a single boundary of the polygon. `facet` is a pointer to facet information for individual polygons.
void **xgl_quadrilateral_mesh** (Xgl_ctx ctx, Xgl_usgn32 row_dim, col_dim, Xgl_facet_list *facets, Xgl_pt_list *pl);	Draws a quadrilateral (quad) mesh according to the parameter list and the graphics state described in a 3D Context. A quad mesh is defined by a single point list containing at least four vertices. The first four vertices define a quad; additional vertices define additional quads.
void **xgl_stroke_text** (Xgl_2d_ctx ctx, void *str, Xgl_pt_f*d *pos, Xgl_pt_f3d dir[]);	Draws a string indexed by the characters in `str` using the Context `ctx`. The `pos` argument is the 2D or 3D reference point for the rendered string. The `str` argument is a NULL-terminated, C-style string of characters. The `dir` argument is an array of two vectors that define the plane on which 2D text is located in 3D environments; this argument is ignored in 2D Contexts. Each point within the stroke character is transformed through the XGL transformation pipeline.

Table 8-2 Geometric Primitive Operators *(4 of 4)*

Operator		Description
void **xgl_stroke_text_extent** (Xgl_ctx ctx, void *string, Xgl_bounds_f2d *rect, Xgl_pt_f2d *cat_pt);		Computes a rectangle in text local coordinates bounding a stroke text string. The Context `ctx` contains the Stroke Font object and stroke text attributes necessary to compute the rectangular box returned in the argument `rect`. The `cat_pt` argument specifies where the string ends and can be used to append a new string to a previously rendered one.
void **xgl_triangle_list** (Xgl_ctx ctx, Xgl_facet_list *facets, Xgl_pt_list *pl, Xgl_tlist_flags flags);		Draws a triangle list according to the parameter list and the graphics state described in a 3D Context. A triangle list is defined by a single point list containing at least three vertices. The list may consist of a series of connected triangles arranged as a triangle strip, a series of connected triangles arranged as a triangle star, or a series of unconnected independent triangles. The first three vertices from the point list define a triangle. Flags passed to the primitive via the `flags` parameter or supplied in the flag word in the vertex data determine whether additional vertices produce a triangle strip, triangle star, independent triangles, or a combination of the three.
void **xgl_triangle_strip** (Xgl_ctx ctx, Xgl_facet_list *facets, Xgl_pt_list *pl);		Draws a triangle strip according to the parameter list and the graphics state described in a 3D Context. A triangle strip is defined by a single point list containing at least three vertices. The first three vertices define a triangle, and each additional vertex defines an adjacent triangle.

Drawing Primitive Data Structures

The geometry in XGL primitives is defined by a set of coordinates specified in the Local Modeling Coordinate system. XGL transforms the coordinates to the Device Coordinate system and draws the primitive at the given coordinates. Applications can supply coordinate point data in 2D integer or 2D or 3D floating point data types. Depending on the geometry to be drawn and on the Context, one point data type may be more appropriate than another.

XGL drawing primitives use a variety of data structures as operands in the function calls. The primary data structures are discussed in the following sections.

Point Types

Point data structures are used to supply drawing primitives with coordinate data. The point structures may also specify other information, such as color, flags that control the drawing of edges around surfaces, normals, or (*u*,*v*) data. The type of a point is defined by its *dimension* (2D, 3D, 2D homogeneous, or 3D homogeneous), its *data type* (int or float), and any additional information (a color, a normal, or a flag). See the *Xgl_pt_list* man page or the *xgl_struct* man page for information on the available point types in XGL.

Point Lists

Most XGL primitives take a list of points as an operand. Point lists store the coordinate values that define a primitive. When the coordinates in a point list are input to a primitive operator, they produce an image on the screen. The list of points is represented to XGL by the data structure *Xgl_pt_list*, which is defined as:

```
typedef struct {
    Xgl_pt_type        pt_type;
    Xgl_bbox_status    bbox_status;
    Xgl_usgn32         num_pts;
    Xgl_usgn32         num_data_values;
    Xgl_pt_pcnf        pts;
} Xgl_pt_list
```

The point list data structure has five fields: the type of points (all points in a list must have the same type), a bounding box (described in the next section), the number of points in the list, the number of data values at each point, and a pointer to the array of points.

Note – The primitives `xgl_multiarc()`, `xgl_multicircle()`, `xgl_multirectangle()`, and `xgl_multi_elliptical_arc()` take special data structures for drawing lists of arcs, circles, rectangles, ellipses, and elliptical arcs. These data structures are *Xgl_arc_list*, *Xgl_circle_list*, *Xgl_rect_list*, and *Xgl_ell_list*. They are similar to *Xgl_pt_list*. The primitive `xgl_multi_elliptical_arc()` is 3D only.

Facet Structures

Some surface primitives take an additional parameter: a *facet* structure, or a list of facet structures. Generally, a facet refers to a single planar section of a surface, which might be a simple polygon or one triangle within an XGL triangle strip. The facet data provide information such as a color or normal, or both, for that section of the surface. The 3D surface primitives that take facet structures are multi-simple polygons, general polygons, quadrilateral mesh, triangle strips, and triangle lists.

Facets are passed to XGL using the data structure *Xgl_facet_list*. The *Xgl_facet_list* data structure is:

```
typedef struct {
    Xgl_facet_type     facet_type;
    Xgl_usgn32         num_facets
    union {
        Xgl_color_facet           *color_facets;
        Xgl_normal_facet          *normal_facets;
        Xgl_color_normal_facet    *color_normal_facets;
    } facets;
} Xgl_facet_list
```

The facet list structure has three fields: the *type of facets* (all facets in a list must have the same type), the *number of facets* in the list, and a *pointer* to an array of facet data containing information about the facet color or normal vector. Facets are optional arguments; a facet structure can be set to `NULL`.

The color data in a facet structure specify how a facet is to be colored. When an application supplies more than one source of color, it can select the source of color (Context, facet, or vertex) to be used in rendering with the attribute `XGL_CTX_SURF_FRONT_COLOR_SELECTOR`. It can also select the illumination mode using the Context color, facet color, or vertex color. Color facet data can be supplied for a 2D or a 3D surface.

The normal vectors provided with facet data allow XGL to determine whether a facet on a 3D surface is front-facing or back-facing. Information on the orientation of a facet is needed for lighting, face culling, or face distinguishing. A facet normal points in the direction of the front face of the surface. Facet normals provided to XGL should be unit vectors.

An application can supply normal vectors per facet in the facet data structures, or it can supply normals per vertex in the point data structures. When facet data is not explicitly provided for surface primitives as part of an application's data, the XGL system calculates the facet normal for a surface using the Context attribute `XGL_3D_CTX_SURF_GEOM_NORMAL`. This attribute computes the geometric facet normal by selecting three vertices on the surface and determining whether the vertices are ordered in a clockwise or counterclockwise manner. For more information on how XGL calculates facet normals, see the man page for `XGL_3D_CTX_SURF_GEOM_NORMAL`.

Surface primitives that do not take facet structures as input require information about vectors as part of the point structures input to the primitive. These primitives are 3D multicircles, multiarcs, multirectangles, and multielliptical arcs. Fields in the 3D point structures for these primitives require orthogonal direction vectors to define the plane of the geometry. A normal vector for the surface can also be specified, or XGL will compute a normal vector to the surface using the vectors that define the plane. For more information on how the geometric normal is computed for a particular primitive, see the man page for the primitive.

Bounding Boxes

A bounding box is the smallest box that completely encloses all the geometry in a point list. A bounding box can be used to optimize clipping operations when rendering a primitive. If the entire bounding box is within the clip bounds, the image needs no clipping.

The bounding box data structures contain information about the minimum and maximum limits of the bounding box. The application has several ways to pass bounding box information to the XGL primitives. The application can:

- Pass bounding box information using the *Xgl_bbox* structure. This structure was available at the previous release of the XGL library and is retained to provide backward compatibility.
- Pass bounding box information using any of six new bounding box structures. These new structures reduce allocated memory and maintain compatibility with the previous release.
- Determine the status of the bounding box geometry and pass the status to the XGL primitive.

Each of these approaches is discussed in more detail in the sections below.

A bounding box is optional for all drawing primitives taking a point list. In addition, primitives taking a list of point lists (to render multiple geometric entities such as polylines) use an additional bounding box parameter, which specifies the bounding box of all the point list bounding boxes. The bounding box and the geometry are defined in the same coordinate system. The application can set the bounding box field to `NULL` if it does not need this data.

Providing Bounding Box Information Using the Xgl_bbox Structure

In previous releases of the XGL library, bounding box information was provided via a pointer to a structure of type *Xgl_bbox*. The *Xgl_bbox* structure is defined as:

```
typedef struct {
    Xgl_bbox_type      bbox_type;
    union {
        Xgl_bounds_i2d i2d;
        Xgl_bounds_f2d f2d;
        Xgl_bounds_f3d f3d;
    } box;
} Xgl_bbox;
```

For compatibility, the XGL library supports the *Xgl_bbox* structure, although this is no longer the recommended way to pass bounding box information to XGL primitives.

Providing Bounding Box Information Using the New XGL Bounding Box Structures

The current release of the XGL library provides six new structures in which to store bounding box information. These structures are similar to *Xgl_bbox*, but they enable the application to tailor memory usage to its specific requirements. The six structures are:

Xgl_bbox_i2d	*Xgl_bbox_status*
Xgl_bbox_f2d	*Xgl_bbox_d2d*
Xgl_bbox_f3d	*Xgl_bbox_d3d*

Each structure allocates only the memory needed. For example, *Xgl_bbox_f2d* is defined as:

```
typedef struct {
    Xgl_bbox_type      bbox_type;
    union {
        Xgl_bounds_f2d f2d;
    } box;
} Xgl_bbox_f2d;
```

In place of an *Xgl_bbox* structure, the application should use one of the new structures. However, if the `bbox` field in a point list points to a structure of a type other than *Xgl_bbox*, the application must cast the variable to *Xgl_bbox*

before compiling. For example, if an application uses a variable called `bbox_f3d` of type *Xgl_bbox_f3d*, it must cast the pointer to *Xgl_bbox* before passing the information to XGL, as shown in the code fragment below.

```
{
    Xgl_pt_list         pl;
    Xgl_bbox_f3d        bbox_f3d;

    pl.bbox = (Xgl_bbox *) (&bbox_f3d);
}
```

Providing Bounding Box Geometry Status

The application can optimize clipping operations by determining whether the geometry will be clipped before calling the drawing primitive. If the application wants to pass the geometry status to an XGL primitive, it can first call `xgl_context_check_bbox()` to determine the geometry status of the bounding box and then pass the status to the primitive using the *Xgl_bbox_status* structure. The `xgl_context_check_bbox()` operator enables the application to calculate the geometry status of the bounding box. This operator determines whether the geometry that is to be input to a drawing primitive will be completely clipped, partially clipped, or not clipped (inside the drawing area) after it is transformed to device coordinates. The primitive is defined as:

```
void xgl_context_check_bbox (
    Xgl_ctx                         ctx,
    Xgl_primitive_type              prim_type,
    Xgl_bbox                        *bbox,
    Xgl_geom_status                 *geom_status);
```

For polygonal primitives, if the application knows that the primitives contained in the bounding box are all front facing (or back facing), it can provide this information to the XGL primitive by logically OR'ing the flag `XGL_GEOM_STATUS_FACING_FRONT` (or `XGL_GEOM_STATUS_FACING_BACK`) with the geometry status calculated by `xgl_context_check_bbox()`, before passing the geometry status to XGL drawing operators.

The following code fragment is an example of how to use `xgl_context_check_bbox()` to calculate the geometry status of a bounding box and pass the status to an XGL primitive.

```
{
  Xgl_ctx            ctx;
  Xgl_primitive_type prim_type;
  Xgl_bbox_f2d       bbox_f2d;
  Xgl_bbox_status    bbox_status;
  Xgl_geom_status    geom_status;
  Xgl_pt_list        pl;

  prim_type = XGL_PRIM_MULTIMARKER;

  /* set up the bounding box for the 2d markers */
  bbox_f2d.bbox_type = XGL_BBOX_F2D;
  bbox_f2d.box.f2d.xmin = bbox_f2d.box.f2d.ymin = 100.0;
  bbox_f2d.box.f2d.xmax = bbox_f2d.box.f2d.ymax = 200.0;

 xgl_context_check_bbox(ctx, prim_type, (Xgl_bbox *)(&bbox_f2d),
                            &geom_status);

  /* set up the Xgl_bbox_status structure */
  bbox_status.bbox_type = XGL_BBOX_STATUS;
  bbox_status.box.status = geom_status;

  pl.bbox = (Xgl_bbox *)(&bbox_status);
  xgl_multimarker(ctx, &pl);
  }
```

The application must ensure that the bounding box's geometry status is valid for the geometry. Thus, if any of the XGL Context attributes affecting the Model Coordinate to Device Coordinate transformation and clipping (or affecting the front/back facing status) is modified, the bounding box geometry status must be recalculated.

Note – The geometry status calculated by `xgl_context_check_bbox()` only applies to XGL non-Gcache primitives such as `xgl_multimarker()`. For XGL Gcache operators, such as `xgl_gcache_multimarker()`, applications should use one of the other bounding box structures.

8

Annotated Primitives

An application has the option of rendering certain primitives in a plane parallel to the projection plane. Primitives rendered in this way are called annotation primitives. Annotation primitives are 3D multicircles, arcs, elliptical arcs, multirectangles, and annotation text. The multicircle, arc, elliptical arc, and multirectangle operators generate annotated primitives when the point list data structures input to the operators specify annotation data.

`xgl_multicircle()`
To generate annotation multicircles, specify type `XGL_MULTICIRCLE_AF3D` for the data type of the circle's defining points and use a point type of *Xgl_circle_af3d* to define the circle point structures. The center of each circle represents the reference point and is specified in model coordinates. The radius is in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multiarc()`
To generate annotation arcs, specify type `XGL_MULTIARC_AF3D` for the data type of the arc's defining points and use a point type of *Xgl_arc_af3d* to define the arc point structures. The center of each arc represents the reference point and is specified in model coordinates. The radius is in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multi_elliptical_arc()`
To generate annotation elliptical arcs, specify type `XGL_MULTIELLIARC_AF3D` for the data type of the arc's defining points and use a point type of *Xgl_ell_af3d* to define the elliptical arc point structures. The center of each elliptical arc represents the reference point and is specified in model coordinates. The major and minor axes are specified in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multirectangle()`
To generate annotation rectangles, specify type `XGL_MULTIRECT_AF3D` for the data type of the rectangle's defining points and use a point type of *Xgl_rect_af3d* to define the rectangle point structures. The reference corner of each rectangle is specified in model coordinates, and the other corner represents the virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

The annotation text primitive always generates text parallel to the projection plane. See Chapter 12, “Text”, for information on annotation text.

Graphics Context Attributes

Rendering describes how graphics data is drawn — how polygons are filled and lines are styled, as well as how clipping and transformations are performed. Rendering is defined by the attributes set within the Context object. Some attributes (like the `XGL_CTX_ROP` [Raster OPeration] mode) apply to all primitives, while others apply to specific primitives.

Sometimes the value of a Context attribute can be overridden during a call to a primitive operator. For example, the points in a polyline can have color values that are used for each line segment. These point color values can override the polyline color value. In addition, object attributes may interact with Context attributes to produce the final rendered appearance of geometry. For example, the Line Pattern object determines the pattern of the rendered lines or edges, while the Context line attributes determine the appearance of the lines or edges. As another example, even though the color of surfaces is set with a Context surface attribute, the Light object can modify the surface color of a primitive.

The Context attributes that contribute to the display of primitives are listed and briefly described in the sections below. The relationship between rendering primitives and Context attributes during the execution of a primitive is illustrated in Figure 8-2 on page 208. The sections below are intended merely as an overview of the function that the Context object encompasses. For information on specific Context attributes, see the subsequent chapters of this manual or the *XGL Reference Manual*.

General Rendering Attributes

The Context attributes in Table 8-3 apply to most or all of the drawing primitives.

Table 8-3 General Context Rendering Attributes

Attribute	Description
XGL_CTX_DEFERRAL_MODE	Determines whether graphics data sent to a Device is buffered. Primitives can be rendered as soon as possible (`XGL_DEFER_ASAP`) or rendered at a later time (`XGL_DEFER_ASTI`). In the latter case, primitives are rendered either when the buffer is full or when the operator `xgl_context_flush()` is called.
XGL_CTX_THRESHOLD	Defines a threshold value for a primitive's size, below which the primitive is simplified before rendering.
XGL_CTX_PLANE_MASK	Defines the pixel-plane mask.
XGL_CTX_ROP	Defines how the set of pixels comprising the graphics primitive is logically combined with the destination Raster's pixels to product the output written to the Raster Device.
XGL_CTX_BACKGROUND_COLOR	Defines the color used for background pixels.
XGL_CTX_RASTER_FILL_STYLE	Defines the fill style used for the Raster-level primitive `xgl_context_copy_buffer`.
XGL_CTX_RASTER_FPAT	Defines the Raster fill pattern used when `xgl_context_copy_buffer` is invoked using a stipple fill.
XGL_CTX_RASTER_FPAT_POSITION	Defines the mapping of the fill pattern when `xgl_context_copy_buffer` is invoked using a stipple fill.
XGL_CTX_RASTER_STIPPLE_COLOR	Defines the background color used when `xgl_context_copy_buffer` is invoked using an opaque stipple fill.
XGL_CTX_NEW_FRAME_ACTION	Sets the actions performed by the `xgl_context_new_frame` operator.

Table 8-3 General Context Rendering Attributes *(Continued)*

Attribute	Description
XGL_CTX_RENDER_BUFFER	Controls where primitives are rendered on the Device associated with the Context. Primitive operations can be rendered onto the Raster draw buffer, the image buffer, or the Z buffer.
XGL_CTX_RENDERING_ORDER	Allows a device to set the rendering order of primitives.
XGL_CTX_GEOM_DATA_IS_VOLATILE	Specifies whether the application's data can be used after the XGL primitive returns.

Context Accumulation Buffer Attributes

Context attributes specify the destination buffer for an accumulation operation and specify the amount of jittering for the accumulation. When jittered images are averaged together, the resulting image is effectively antialiased. Table 8-4 lists the Context accumulation buffer attributes. See page 228 for information on accumulation buffers.

Table 8-4 Context Accumulation Buffer Attributes

Attributes	Description
XGL_3D_CTX_ACCUM_OP_DEST	Specifies the buffer to be used as the destination buffer during an accumulation operation.
XGL_3D_CTX_JITTER_OFFSET	Specifies the amount by which geometry should be offset in device coordinates before drawing. This is used when rendering an image before accumulating into an accumulation buffer.

Hidden Line/Hidden Surface Removal (HLHSR)

In a 3D Context, the application programmer can use XGL_3D_CTX_HLHSR_MODE to enable the XGL hidden line and surface removal features. Z-buffering is used for all hidden line and surface removal. If the output device doesn't support hardware Z-buffering, a Z-buffer is created in system memory. However, simulating a hardware Z-buffer in this way is slower than using direct hardware support.

Hidden line and surface removal is defined with the Context attributes listed in Table 8-5.

Table 8-5 Context Hidden Line and Surface Removal Attributes

Attribute	Description
XGL_3D_CTX_HLHSR_MODE	Defines the mode used for hidden line and hidden surface removal.
XGL_3D_CTX_HLHSR_DATA	Defines the data used to clear the Z-buffer for hidden line/hidden surface mode.
XGL_3D_CTX_Z_BUFFER_WRITE_MASK	Defines which bit planes of the Z-buffer are written.
XGL_3D_CTX_Z_BUFFER_COMP_METHOD	Specifies the comparison technique used when the Z-buffer is updated. The result of the comparison determines how the destination buffer is updated.

Technique for Fast Hidden Line Removal

If only hidden line removal is needed, and the output device doesn't support hardware Z-buffering, the application programmer can use the attribute XGL_CTX_SURF_EDGE_FLAG to produce a fast, hidden line-removal technique. To use this technique, follow these general steps:

1. Set the XGL_CTX_SURF_EDGE_FLAG attribute to ON.
2. Set the front and back surface colors to the background color. (See the man pages for XGL_3D_CTX_SURF_BACK_COLOR, XGL_3D_CTX_SURF_FRONT_COLOR, and XGL_3D_CTX_BACKGROUND_COLOR.)

3. Set shading and depth cueing to `XGL_ILLUM_NONE` and `XGL_DEPTH_CUE_OFF`, respectively. (See the man pages for `XGL_3D_CTX_SURF_BACK_ILLUMINATION`, `XGL_3D_CTX_SURF_FRONT_ILLUMINATION`, and `XGL_3D_CTX_DEPTH_CUE_MODE`.)

The rendering is equivalent to a fast, hidden line removal technique, preserving correct readability of the displayed data.

Context Line Attributes

Context line attributes, listed in Table 8-6, set line style and line color. They also define the shape of the endpoints of lines and curves, set the line width scale factor, and define the shape of joins between line segments. Line primitives are polylines and B-spline curves. For information on line patterns, see Chapter 13, "Line Patterns."

Table 8-6 Context Line Attributes

Attribute	Description
XGL_CTX_LINE_ALT_COLOR	Defines the alternate color used in patterned lines.
XGL_CTX_LINE_AA_BLEND_EQ XGL_CTX_LINE_AA_FILTER_SHAPE XGL_CTX_LINE_AA_FILTER_WIDTH	Control whether lines are antialiased and defines the method of antialiasing.
XGL_CTX_LINE_CAP	Defines the shape of the endpoints of lines and curves.
XGL_CTX_LINE_COLOR	Defines the color of lines and curves.
XGL_CTX_LINE_COLOR_SELECTOR	Determines whether the source of a line's color is the Context or the primitive vertex data.
XGL_CTX_LINE_JOIN	Defines the shape of joins between line segments.
XGL_CTX_LINE_MITER_LIMIT	Defines the limit for mitering of line segment corners.
XGL_CTX_LINE_PATTERN	Defines the pattern used for patterned lines.
XGL_CTX_LINE_STYLE	Defines the style of a line, that is, whether the line is solid or patterned.

Table 8-6 Context Line Attributes *(Continued)*

Attribute	Description
XGL_CTX_LINE_WIDTH_SCALE_FACTOR	Defines the line width scale factor used to determine the width of lines and curves.
XGL_3D_CTX_LINE_COLOR_INTERP	Controls color interpolation along 3D lines.

Note – Solid and dashed wide lines are not displayed when viewed on end, but a point is displayed.

Context Marker Attributes

Context attributes define the size, color, and type of marker that is rendered. Context Marker rendering attributes are listed in Table 8-7. For information on Marker objects, see Chapter 14, "Markers."

Table 8-7 Context Marker Attributes

Attribute	Description
XGL_CTX_MARKER	Defines the Marker description drawn by `xgl_multimarker`.
XGL_CTX_MARKER_AA_BLEND_EQ XGL_CTX_MARKER_AA_FILTER_SHAPE XGL_CTX_MARKER_AA_FILTER_WIDTH	Control whether markers are antialiased and defines the method of antialiasing.
XGL_CTX_MARKER_COLOR	Defines the color of markers.
XGL_CTX_MARKER_COLOR_SELECTOR	Determines whether the source of a marker's color is the Context or the geometry point data.
XGL_CTX_MARKER_SCALE_FACTOR	Defines the Marker scale factor used to determine the width of Markers.

Context Curve Attributes

The attributes in Table 8-8 define NURBS curve rendering. Context line attributes also apply to NURBS curves. For more information on NURBS curves, see Chapter 9, “Rendering NURBS Curves and Surfaces with XGL.”

Table 8-8 Context NURBS Curve and Surface Attributes

Attribute	Description
XGL_CTX_NURBS_CURVE_APPROX	Defines the NURBS curve approximation method.
XGL_CTX_NURBS_CURVE_APPROX_VAL	Defines the NURBS curve approximation values.
XGL_CTX_MIN_TESSELLATION XGL_CTX_MAX_TESSELLATION	Defines the minimum and maximum number of segments in which a NURBS or 3D conic curve can be tessellated when using the primitives `xgl_nurbs_curve`, `xgl_multicircle`, `xgl_multiarc`, and `xgl_multi_elliptical_arc`.

Context Surface Attributes

Context attributes define many of the aspects of surface rendering. For example, Context attributes set the color used to fill surfaces and define the way in which surfaces are filled. Context attributes specify how a surface is shaded and illuminated. They also define how surface normals are calculated when the normals are not provided in application data.

Surface primitives are rectangles, circles, circular and elliptical arcs, polygons, quadrilateral mesh, triangle strips, triangle lists, and NURBS surfaces. Some surface rendering attributes can be specified for both 2D and 3D Contexts, while other attributes are only used in 3D Contexts or in 2D Contexts. A brief overview of the surface rendering attributes is provided in the sections below. The 3D surface characteristics related to lighting and shading are discussed more fully in Chapter 16, “Lighting, Shading, and Depth Cueing.”

Surface Edge Rendering Attributes

Edges are characteristics of the surfaces of primitives; they enclose a surface and would not exist if there were not a surface. Edges are drawn if the surface edge attribute `XGL_CTX_SURF_EDGE_FLAG` is set to `TRUE`. Some primitives have additional control over edge rendering: if the edge flag attribute is set to enable edge rendering, primitives that accept point data that includes edge flag information can use the edge flag information to control the rendering of edge boundaries on a per-vertex basis.

If edge rendering is enabled, Context attributes set the edge style, color, and width characteristics of the edges of surfaces. Surface edge rendering attributes are listed in Table 8-9.

Table 8-9 Context Edge Attributes

Attribute	Description
XGL_CTX_SURF_EDGE_FLAG	Controls whether edges are drawn around surfaces.
XGL_CTX_EDGE_AA_BLEND_EQ XGL_CTX_EDGE_AA_FILTER_SHAPE XGL_CTX_EDGE_AA_FILTER_WIDTH	Control whether surface edges are antialiased and defines the method of antialiasing.
XGL_CTX_EDGE_ALT_COLOR	Defines the alternate color used in patterned edges.
XGL_CTX_EDGE_CAP	Defines the shape of teh endpoints of an edge.
XGL_CTX_EDGE_COLOR	Defines the color used for edges of surfaces.
XGL_CTX_EDGE_JOIN	Defines the shape of the joins between edge segments.
XGL_CTX_EDGE_MITER_LIMIT	Defines the limit for mitering of edge segment corners.
XGL_CTX_EDGE_PATTERN	Specifies the line pattern used for patterned edges.
XGL_CTX_EDGE_STYLE	Defines the edge style used for edges of surfaces.
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR	Defines the edge width scale factor used to determine the width of surface edges.

Surface Color, Light, and Fill Attributes

Context attributes control the global rendering properties of surfaces, such as material color, and the reflection coefficients for the various components of reflected light. The front and back faces of 3D geometry can be determined and shaded or lit appropriately. If 3D surface face distinguishing is enabled by setting the attribute `XGL_3D_CTX_SURF_FACE_DISTINGUISH` to `TRUE`, the front-facing surfaces are rendered using the `XGL_3D_CTX_SURF_FRONT_` attributes, and the back-facing surfaces are rendered using the `XGL_3D_CTX_SURF_BACK_` attributes. Otherwise, both front and back surfaces are rendered with the `XGL_3D_CTX_SURF_FRONT_` or `XGL_CTX_SURF_FRONT_` attributes. 2D surfaces are always rendered as front surfaces. Table 8-10 provides an overview of the Context surface rendering attributes.

Table 8-10 Context Surface Attributes

Attribute	Description
XGL_CTX_ARC_FILL_STYLE	Defines how arcs are filled.
XGL_CTX_SURF_AA_BLEND_EQ XGL_CTX_SURF_AA_FILTER_SHAPE XGL_CTX_SURF_AA_FILTER_WIDTH	Controls whether surfaces are antialiased and defines the method of antialiasing. These attributes only control the appearance of hollow antialiased surfaces. To antialiase filled surfaces, see “Accumulation Buffer” on page page 228.
XGL_3D_CTX_SURF_FACE_DISTINGUISH	Controls whether front- and back-facing surfaces are distinguished from each other.
XGL_CTX_FRONT_COLOR XGL_3D_CTX_SURF_BACK_COLOR	Defines the color used to fill surfaces.
XGL_CTX_SURF_FRONT_COLOR_SELECTOR XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR	Selects the source of a surface’s color from the Context or from facet or vertex data provided with the primitive.
XGL_CTX_SURF_FRONT_FILL_STYLE XGL_3D_CTX_SURF_BACK_FILL_STYLE	Defines how surfaces are filled.
XGL_CTX_SURF_FRONT_FPAT XGL_3D_CTX_SURF_BACK_FPAT	Defines the fill pattern raster used for stipple fill.
XGL_CTX_SURF_FRONT_FPAT_POSITION XGL_3D_CTX_SURF_BACK_FPAT_POSITION	Defines the origin of the fill pattern raster used for stipple fill.

Table 8-10 Context Surface Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_SURF_FRONT_AMBIENT XGL_3D_CTX_SURF_BACK_AMBIENT	Defines the ambient reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_DIFFUSE XGL_3D_CTX_SURF_BACK_DIFFUSE	Defines the diffuse reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_ILLUMINATION XGL_3D_CTX_SURF_BACK_ILLUMINATION	Specifies how illumination calculations are performed.
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT	Determines which components of the lighting equation are used to compute a surface color. Any combination of the components ambient, diffuse, or specular can be enabled.
XGL_3D_CTX_SURF_FRONT_SPECULAR XGL_3D_CTX_SURF_BACK_SPECULAR	Defines the specular coefficient used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR	Defines the specular color used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER XGL_3D_CTX_SURF_BACK_SPECULAR_POWER	Defines the specular power used in computing lighting.
XGL_3D_CTX_SURF_FACE_CULL	Controls face culling in a 3D Context.
XGL_3D_CTX_SURF_GEOM_NORMAL	Controls how surface normals are calculated for a facet if the surface normals are not provided as part of the application data.
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP	Defines how surface normals are treated for lighting.
XGL_3D_CTX_SURF_NORMAL_FLIP	Specifies whether vertex and facet normals are flipped 180 degrees.
XGL_3D_CTX_LIGHT_SWITCHES	Defines the on and off switches that correspond to the 3D Context's array of lights. The number of switches is given by `XGL_3D_CTX_LIGHT_NUM`.
XGL_CTX_SURF_INTERIOR_RULE	Defines the way in which pixels are determined to be inside or outside polygons with multiple bounds.

Table 8-10 Context Surface Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_SURF_DC_OFFSET	Determines whether a DC offset is added to the z component of each vertex of a polygon. The offset value moves the polygon back from its computed depth position.
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG	Controls whether silhouette edges are drawn around a surface.

Surface Transparency Attributes

Context attributes enable applications to render transparent surfaces. XGL provides screen-door and blended (two-pass) transparency. If the transparency method is screen-door, a device-dependent *screen door* (a mesh that allows rendering of only some of the surface's pixels) is applied to the primitive. If the transparency method is blended, blending equations determine how the surface pixel values are blended with the background color or existing pixel values. Transparent surfaces are subject to hidden line and surface removal as defined by `XGL_3D_CTX_HLHSR_MODE`. Transparency is supported for 3D surface primitives only. Table 8-11 lists the surface transparency attributes.

Table 8-11 Context Surface Transparency Attributes

Attribute	Description
XGL_3D_CTX_SURF_TRANSP_METHOD	Defines the method used to render transparent surfaces. Methods are `XGL_TRANSP_BLENDED` or `XGL_TRANSP_SCREEN_DOOR`.
XGL_3D_CTX_SURF_FRONT_TRANSP XGL_3D_CTX_SURF_BACK_TRANSP	Specifies the transparency value for surface primitives. Surface transparency can range from fully transparent through translucent to opaque.
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ	Controls how the pixel values in transparent surfaces are blended with existing pixel values when the surface transparency method is `XGL_TRANSP_BLENDED`.
XGL_3D_CTX_BLEND_DRAW_MODE	Controls the drawing of blended and non-blended primitives.
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER	Defines whether blended primitives update the Z-buffer.

Curved Surface Attributes

Attributes for NURBS surface rendering are listed in Table 8-12. Context surface attributes also apply to NURBS surfaces. For information on NURBS surfaces, see Chapter 9, "Rendering NURBS Curves and Surfaces with XGL."

Table 8-12 Context Curved Surface Attributes

Attribute	Description
XGL_CTX_NURBS_SURF_APPROX	Defines the NURBS surface approximation method.
XGL_CTX_NURBS_SURF_APPROX_VAL_U XGL_CTX_NURBS_SURF_APPROX_VAL_V	Defines the NURBS surface approximation values.
XGL_CTX_NURBS_SURF_PARAM_STYLE	Defines the appearance of the NURBS surface.
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT	Defines the placement of isoparametric curves on a NURBS surface.
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM	Defines the number of isoparametric curves in the u or v directions on a NURBS surface.
XGL_CTX_MIN_TESSELLATION XGL_CTX_MAX_TESSELLATION	Defines the minimum and maximum number of segments in which a NURBS surface can be tessellated.

Surface Texturing Attributes

Attributes for applying textures to 3D surfaces are listed in Table 8-13. For information on texture mapping and texture mapping attributes, see Chapter 18, "Texture Mapping.".

Table 8-13 Context Texture Mapping Attributes

Attribute	Description
XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CTX_SURF_BACK_TMAP	Specifies a list of Texture Map objects for a 3D Context.
XGL_3D_CTX_SURF_FRONT_TMAP_NUM XGL_3D_CTX_SURF_BACK_TMAP_NUM	Specifies the number of Texture Map objects in a 3D Context.
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES	Specifies the list of on-off switches for a 3D Context's Texture Map objects.

Table 8-13 Context Texture Mapping Attributes

Attribute	Description
XGL_3D_SURF_TMAP_PERSP_CORRECTION	Specifies the method used to compute texture coordinate values for surface interiors.
XGL_3D_CTX_SURF_FRONT_DMAP XGL_3D_CTX_SURF_BACK_DMAP	Specifies a list of Data Map Texture objects for a 3D Context.
XGL_3D_CTX_SURF_FRONT_DMAP_NUM XGL_3D_CTX_SURF_BACK_DMAP_NUM	Specifies the number of Data Map Texture objects in a 3D Context.
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES	Specifies the list of on-off switches for a 3D Context's Data Map Texture objects.

Surface Depth Cueing Attributes

Context attributes define the type of depth cueing rendered, specify the color that the primitive will be modulated to as the depth increases, and define how the depth cueing interpolation is calculated. See Chapter 16, "Lighting, Shading, and Depth Cueing" for information on depth cueing. Depth cueing attributes are listed in Table 8-14.

Table 8-14 Context Surface Depth Cueing Attributes

Attribute	Description
XGL_3D_CTX_DEPTH_CUE_COLOR	For RGB Rasters, sets the Context's depth cue color, which is blended with the original colors of primitives to give the modulated colors of primitives after depth cueing.
XGL_3D_CTX_DEPTH_CUE_INTERP	Specifies whether interpolation according to calculated values of depth-cued color is to be performed.
XGL_3D_CTX_DEPTH_CUE_MODE	Defines the depth cueing mode in the Context. This can be set to: `XGL_DEPTH_CUE_OFF`, `XGL_DEPTH_CUE_LINEAR`, or `XGL_DEPTH_CUE_SCALED`.
XGL_3D_CTX_DEPTH_CUE_REF_PLANES	Specifies an array of two *z* values representing the front and the back planes. Used only for scaled depth cueing.
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS	Scaling factors at front and back reference planes. Used only for scaled depth cueing.

Context Stroke Text Attributes

Context text attributes define stroke text character height, width, spacing, and alignment. They also define stroke text horizontal and vertical alignment, direction, and color. Table 8-15 lists stroke text attributes. For more information on the Stroke Text object, see Chapter 12, "Text."

Table 8-15 Context Stroke Text Attributes

Attribute	Description
XGL_CTX_STEXT_AA_BLEND_EQ XGL_CTX_STEXT_AA_FILTER_SHAPE XGL_CTX_STEXT_AA_FILTER_WIDTH	Control whether stroke text is antialiased and defines the method of antialiasing.
XGL_CTX_SFONT_0 , XGL_CTX_SFONT_1, XGL_CTX_SFONT_2, XGL_CTX_SFONT_3	Specifies the Stroke Font for the current Context.
XGL_CTX_STEXT_CHAR_ENCODING	Specifies the character encoding scheme to be used with stroke text.
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR	Defines the horizontal expansion applied to stroke font text.
XGL_CTX_STEXT_CHAR_HEIGHT	Defines the nominal size of stroke text characters.
XGL_CTX_STEXT_CHAR_SLANT_ANGLE	Defines the angle that stroke text makes with the vertical direction.
XGL_CTX_STEXT_CHAR_SPACING	Defines the size of the gap between characters.
XGL_CTX_STEXT_CHAR_UP_VECTOR	Defines the up vector for stroke text.
XGL_CTX_STEXT_ALIGN_HORIZ	Defines the horizontal alignment for stroke text.
XGL_CTX_STEXT_ALIGN_VERT	Defines the vertical alignment for stroke text.
XGL_CTX_STEXT_COLOR	Defines the color used for rendering stroke text.
XGL_CTX_STEXT_PATH	Defines the direction in which stroke text is rendered.
XGL_CTX_STEXT_PRECISION	Defines the precision used for rendering stroke text.

Context Annotation Text Attributes

Annotation text is parallel to the display surface. Context attributes define annotation text character height, character alignment, text alignment, and text direction. Table 8-16 lists annotation text attributes. For more information on annotation text, see Chapter 12, “Text.”

Table 8-16 Context Annotation Text Attributes

Attribute	Description
XGL_CTX_ATEXT_CHAR_HEIGHT	Defines the nominal size of annotation text.
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE	Defines the angle that annotation text with respect to the vertical direction.
XGL_CTX_ATEXT_CHAR_UP_VECTOR	Defines the up vector for annotation text.
XGL_CTX_ATEXT_STYLE	Defines the style in which annotation text is rendered.
XGL_CTX_ATEXT_ALIGN_HORIZ	Defines the horizontal alignment for annotation text.
XGL_CTX_ATEXT_ALIGN_VERT	Defines the vertical alignment for annotation text.
XGL_CTX_ATEXT_PATH	Defines the direction in which annotation text is rendered.

Context View and Transform Attributes

Transform objects are manipulated via Context object attributes. Context Transform attributes direct the conversion of geometric data from application coordinates to device coordinates. Table 8-17 lists Transform and viewing attributes. For more information on the Transform object, see Chapter 10, "Transforms." For information on the XGL view model, see Chapter 11, "View Model."

Table 8-17 Context View and Transform Attributes

Attribute	Description
XGL_CTX_VDC_ORIENTATION	Defines the orientation of virtual device coordinate (VDC) space.
XGL_CTX_MODEL_TRANS	Returns a read-only copy of the XGL Model Transform.
XGL_CTX_LOCAL_MODEL_TRANS	Defines the Transform object used to transform geometry data from local model coordinates to global modeling coordinates.
XGL_CTX_GLOBAL_MODEL_TRANS	Defines the Transform object used to transform geometry data from global model coordinates to world coordinates.
XGL_CTX_VIEW_TRANS	Defines the Transform object used to transform geometry data from world coordinates to virtual device (VDC) coordinates.
XGL_CTX_VIEW_CLIP_BOUNDS	Defines the VDC clipping area.
XGL_CTX_VDC_WINDOW	Defines the VDC window.
XGL_CTX_DC_VIEWPORT	Defines the DC viewport in which rendering occurs.
XGL_CTX_CLIP_PLANES	Defines which planes are used in view clipping.
XGL_CTX_VDC_MAP	Controls mapping of geometry data from virtual device coordinates to device coordinates.
XGL_CTX_MC_TO_DC_TRANS	Returns a read-only copy of the Transform object used to transform data from global model coordinates to device coordinates.
XGL_3D_CTX_NORMAL_TRANS	Returns a read-only copy of the Normal Transform, which transforms normal vectors from model coordinates to world coordinates.

Table 8-17 Context View and Transform Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM	Specifies the number of model clip planes in use.
XGL_3D_CTX_MODEL_CLIP_PLANES	Specifies the list of model clip planes to use.
XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY	Controls clipping in homogeneous space.
XGL_CTX_MODEL_TRANS_STACK_SIZE	Defines the depth of the Transform stack associated with the Context. Used with the `xgl_context_update_model_trans` operator.

Context Paint Attributes

Context paint attributes specify the paint characteristics of a transparent overlay window. Table 8-18 lists the paint attributes.

Table 8-18 Context Paint Type Attributes

Attribute	Description
XGL_CTX_PAINT_TYPE	Specifies the paint type for a transparent overlay window.
XGL_CTX_NEW_FRAME_PAINT_TYPE	Specifies the paint type to be applied to an overlay window when the overlay window is cleared by `xgl_context_new_frame`.

Figure 8-2 on page 208 shows the rendering pipeline for the Context attributes. The figure illustrates the relationship between operators and attributes, and shows the order in which attributes affect the rendering of the final image.

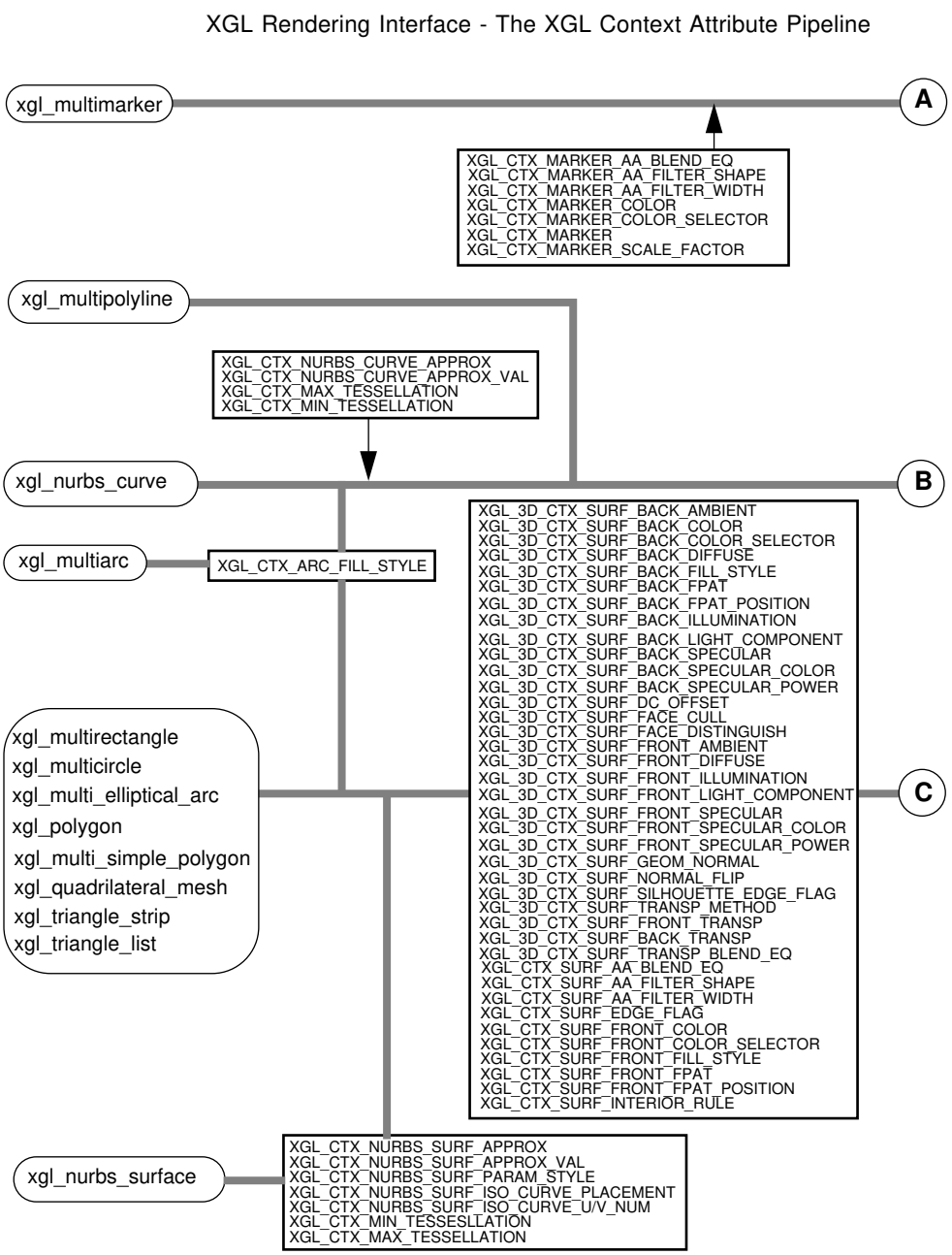

Figure 8-2 Context Attributes Pipeline

XGL Rendering Interface - The XGL Context Attribute Pipeline (cont.)

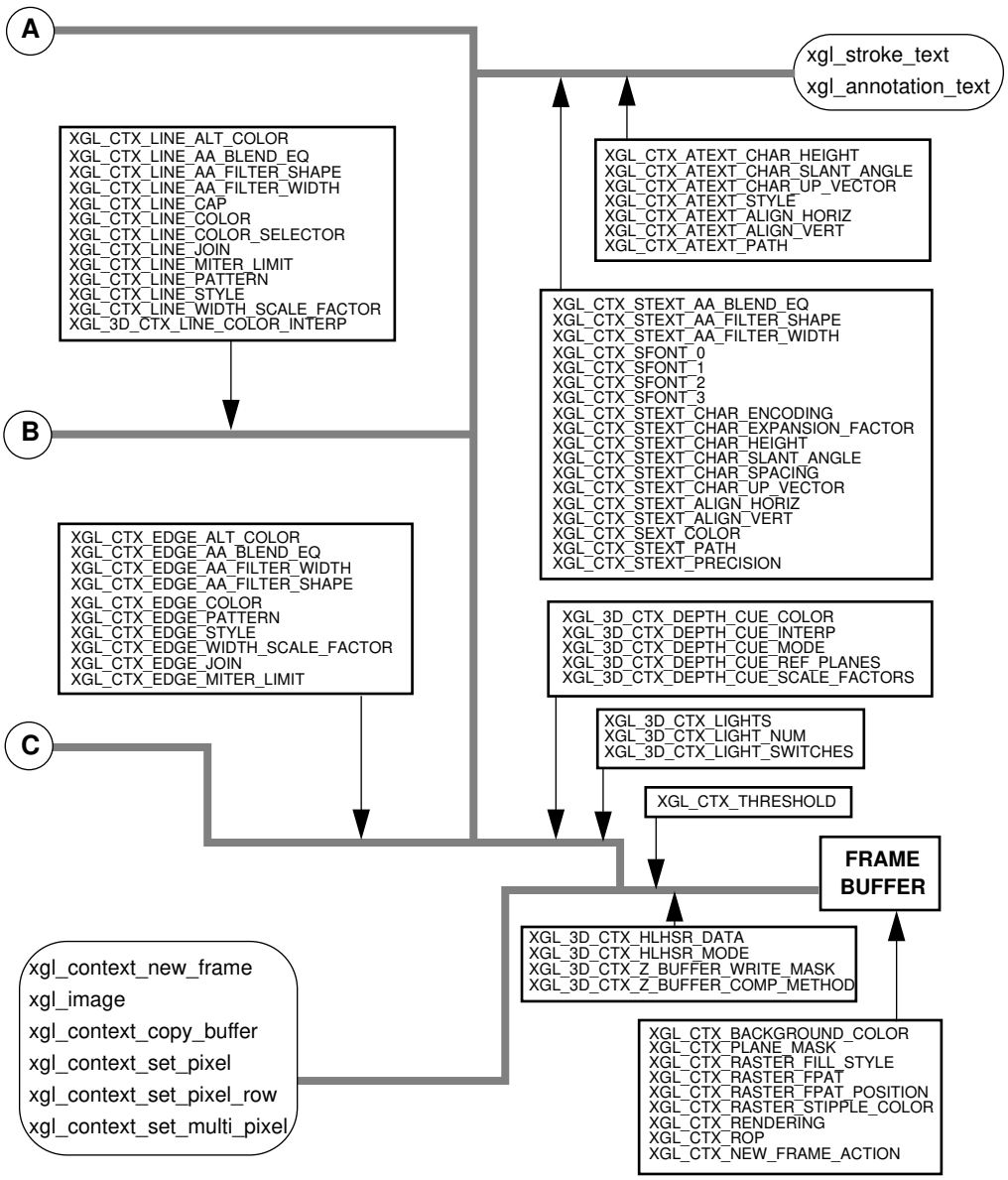

Figure 8-2 Context Attributes Pipeline (Continued)

8

Example Programs using XGL Primitives

The following example programs illustrate XGL primitives. Each example is a fragment of a larger program. The complete program includes `ex_utils.c` and `prims_2d_main.c`, both of which are listed in Appendix B, as well as all the examples listed in this section. To compile the complete program, type `make prims_2d` in the example program directory. The compiled program allows you to look at all the primitive example programs.

Polygon Example Program

This program, `prims_2d_pgon.c`, illustrates the XGL polygon primitive. Figure 8-3 shows the program output.

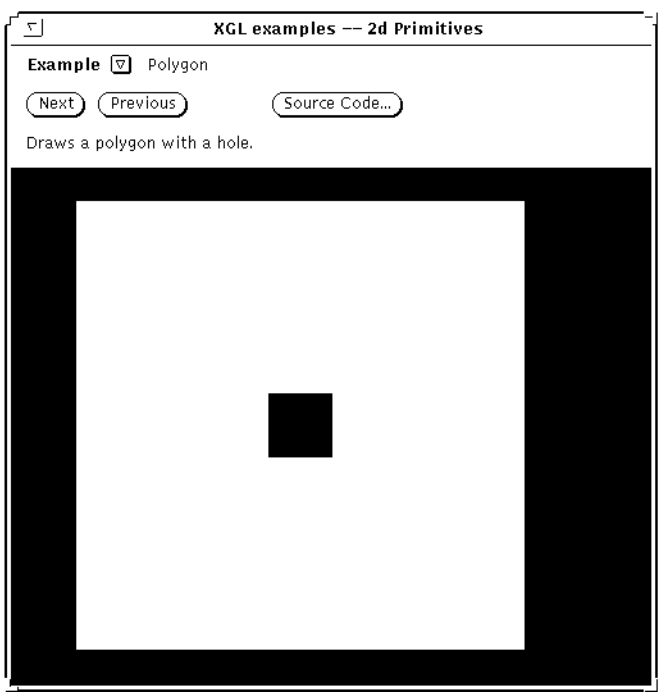

Figure 8-3 Output of `prims_2d_pgon.c`

Code Example 8-1 Polygon Example

```
/*
 * prims_2d_pgon.c
 */

#include "ex.h"

prims_2d_pgon (Xgl_object  ctx)
{
    Xgl_pt_i2d          pts1[20];
    Xgl_pt_i2d          pts2[20];
    Xgl_pt_list         pl[2];
    Xgl_color           color;

    /* points defining outer boundary of polygon */

    pts1[0].x =  50;
    pts1[0].y =  25;
    pts1[1].x = 400;
    pts1[1].y =  25;
    pts1[2].x = 400;
    pts1[2].y = 375;
    pts1[3].x =  50;
    pts1[3].y = 375;

    pl[0].pt_type = XGL_PT_I2D;
    pl[0].bbox = NULL;
    pl[0].num_pts = 4;
    pl[0].pts.i2d = pts1;

    /* points defining inner boundary of polygon */

    pts2[0].x = 200;
    pts2[0].y = 175;
    pts2[1].x = 250;
    pts2[1].y = 175;
    pts2[2].x = 250;
    pts2[2].y = 225;
    pts2[3].x = 200;
    pts2[3].y = 225;

    pl[1].pt_type = XGL_PT_I2D;
    pl[1].bbox = NULL;
    pl[1].num_pts = 4;
    pl[1].pts.i2d = pts2;
```

```
    /* draw cyan, two-boundary polygon */

    color = cyan_color;
    xgl_object_set (ctx,
          XGL_CTX_SURF_FRONT_COLOR, &color,
          NULL);

    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 2, pl);
}
```

Rectangle Example Program

This program illustrates the XGL rectangle primitive. The program results are shown in Figure 8-4.

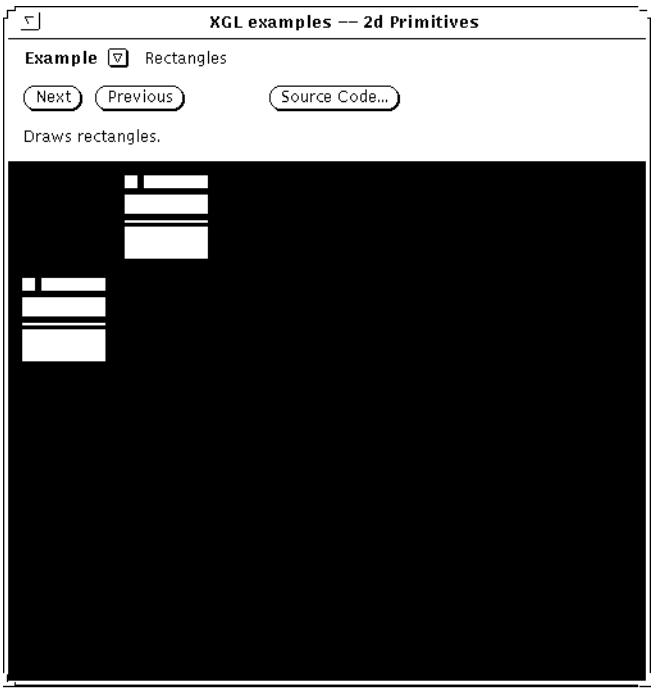

Figure 8-4 Output of `prims_2d_rect.c`

Code Example 8-2 Multirectangle Example

```
/*
 * prims_2d_rect.c
 */
#include "ex.h"

/* the 5 rectangles that will be drawn as a group several times */
static  Xgl_rect_i2d  rects[5] = {
              {{ 10, 10, 0 }, { 20, 20 }},
              {{ 25, 10, 0 }, { 75, 20 }},
              {{ 10, 25, 0 }, { 75, 40 }},
              {{ 10, 45, 0 }, { 75, 47 }},
              {{ 10, 50, 0 }, { 75, 75 }}
                        };

prims_2d_rect (Xgl_objectctx;
{
    Xgl_rect_list     rectlist;
    Xgl_color         color;
    Xgl_pt            pt;
    Xgl_pt_f2d        pt_f2d;
    Xgl_object        lmt;

    /* initialize rectlist struct */

    rectlist.rect_type = XGL_MULTIRECT_I2D;
    rectlist.num_rects = 5;
    rectlist.bbox = 0;
    rectlist.rects.i2d = rects;

    /* initialize pt struct used for translation */

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;

    /* draw 5 rectangles 1st time...all blue */

    color = blue_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multirectangle (ctx, &rectlist);

    /*
     * Get local model transform so that we can translate the
     * group of rectangles
     */
```

```
    xgl_object_get (ctx, XGL_CTX_LOCAL_MODEL_TRANS, &lmt);

   /* translate rectangle group 80 pixels to right */

    pt_f2d.x = 80.0;
    pt_f2d.y = 0.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 2nd time...all yellow */

    color = yellow_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);

    xgl_multirectangle (ctx, &rectlist);

    /* translate rectangle group 80 pixels down */

    pt_f2d.x = 0.0;
    pt_f2d.y = 80.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 3rd time...all cyan */

    color = cyan_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);
    xgl_multirectangle (ctx, &rectlist);

    /* translate rectangle group 250 pixels to right and down */

    pt_f2d.x = 250.0;
    pt_f2d.y = 250.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 4th time...all red */

    color = red_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);

    xgl_multirectangle (ctx, &rectlist);
    /* restore local modeling transform */
    xgl_transform_identity (lmt);
}
```

Circle Example Program

This program illustrates the circle primitive. Figure 8-5 shows the program output.

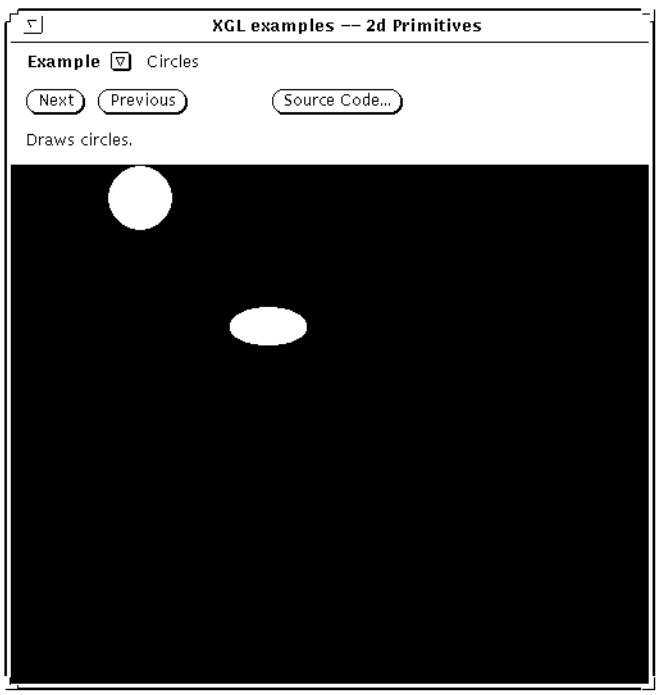

Figure 8-5 Output of `prims_2d_circle.c`

Code Example 8-3 Multicircle Example

```
/*
 * prims_2d_circle.c
 */

#include <math.h>
#include "ex.h"

Xgl_circle_f2d          circs[5] = {
                                {{ 25., 25., 0 },  25.},
                                {{ 25., 25., 0 },  15.},
                                {{ 25., 25., 0 },  5.},
                                {{ 100., 25., 0 }, 25.},
                                {{ 100., 25., 0 }, 15.}
                        };

extern render_ellipse(Xgl_object, Xgl_pt_f2d*, Xgl_pt_f2d*,
                      Xgl_pt_f2d*);


prims_2d_circle (Xgl_object  ctx)
{
    int                 i;
    Xgl_color           color, colors[5];
    Xgl_circle_list     circle_list;
    Xgl_pt_f2d          center, p1, p2;

    colors[0] = red_color;
    colors[1] = green_color;
    colors[2] = blue_color;
    colors[3] = magenta_color;
    colors[4] = yellow_color;

    /*
     * get a smooth circle
     */
    xgl_object_set(ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
                   XGL_CTX_CURVE_APPROX_VALUE, 1.0,
                   NULL);

    /* draw the circles */
    for (i = 0; i < 5; i++) {
        circle_list.type = XGL_MULTICIRCLE_F2D;
        circle_list.num_circles = 1;
        circle_list.bbox = 0;
```

```
            circle_list.circles.f2d = &circs[i];

            color = colors[i];
            xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color,
                        NULL);

            xgl_multicircle(ctx, &circle_list);
        }

        /* draw the ellipses */
        for (i = 0; i < 5; i++) {
            center.x = circs[i].center.x + 100.;
            center.y = circs[i].center.y + 100.;
            p1.x = center.x;
            p1.y = center.y + ((float)(5 - i) * 5. + 20.);
            p2.x = center.x;
            p2.y = center.y + ((float)(5 - i) * 5. + 5.);

            color = colors[i];
            xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color,
                        NULL);

            render_ellipse(ctx, &center, &p1, &p2);
        }
    }

    /*
     * modify the Xgl 2D transformations to render an ellipse by
     * using the circle primitive
     * center is the center point of the ellipse
     * p1 is the major axis and p2 is the minor axis
     */
    render_ellipse(
        Xgl_object  ctx,
        Xgl_pt_f2d  *center,
        Xgl_pt_f2d  *p1,
        Xgl_pt_f2d  *p2)
    {
        Xgl_circle_list     circle_list;
        Xgl_circle_f2d      f2d;
        Xgl_object          lmt;
        float               maj_axis, min_axis, frad;
        Xgl_pt              pt;
        Xgl_pt_f2d          ptf2d;

    #define SQR(x)   ((x) * (x))
```

```
    maj_axis = (float)sqrt(SQR(p1->x - center->x) +
                           SQR(p1->y - center->y));
    min_axis = (float)sqrt(SQR(p2->x - center->x) +
                           SQR(p2->y - center->y));
    frad = maj_axis;

    xgl_object_get (ctx, XGL_CTX_LOCAL_MODEL_TRANS, &lmt);

    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = -center->x;
    ptf2d.y = -center->y;
    pt.pt.f2d = &ptf2d;
    xgl_transform_translate(lmt, &pt, XGL_TRANS_REPLACE);
    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = maj_axis / frad;
    ptf2d.y = min_axis / frad;
    pt.pt.f2d = &ptf2d;
    xgl_transform_scale(lmt, &pt, XGL_TRANS_POSTCONCAT);

    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = center->x;
    ptf2d.y = center->y;
    pt.pt.f2d = &ptf2d;
    xgl_transform_translate(lmt, &pt, XGL_TRANS_POSTCONCAT);

    f2d.center.x = center->x;
    f2d.center.y = center->y;
    f2d.center.flag = 0;
    f2d.radius = frad;

    circle_list.num_circles = 1;
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.bbox = 0;
    circle_list.circles.f2d = &f2d;
    xgl_multicircle(ctx, &circle_list);

    xgl_transform_identity(lmt);
}
```

8

Raster Primitives

Primitives are available from within the XGL Context object for performing low-level operations on the contents of the associated raster. These primitives are used to manipulate raster contents on a per-pixel basis. They enable the application to interact directly with the pixel values stored in the raster. These primitives were introduced in Chapter 7, “Contexts.”

Getting and Setting Pixel Values

The primitives `xgl_context_set_pixel()` and `xgl_context_get_pixel()` are used to set and retrieve the color value of a single pixel stored in a specified buffer of an XGL raster. Arguments to these calls are the Context associated with the Raster, the location of the pixel, and the color value. The pixel color value is assumed to be the same color type as the Raster color type (`XGL_DEV_COLOR_TYPE`). The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER`.

The primitive `xgl_context_set_multi_pixel()` is comparable in functionality to `xgl_context_set_pixel()`, but it accepts an array of pixels instead of just one pixel. `xgl_context_set_pixel_row()` sets multiple pixels of the same row to a color.

Copying Buffer Contents

The operator `xgl_context_copy_buffer()` copies a rectangular block of pixels from a buffer in the source raster to one or more of the Raster buffers associated with the destination Context. In the simple case of copying from a Memory Raster with one buffer to a non-multibuffered Window Raster, the pixels are copied from the draw buffer of the source Raster to the draw buffer of the Raster associated with the destination Context.

For more complicated cases with multiple buffers, the application program must set attributes to specify the buffer to be used as the source buffer and the buffer to be used as the destination in the Raster. The buffer to be used as source in the source Raster is specified by the source raster’s attribute `XGL_RAS_SOURCE_BUFFER`. The buffer(s) to be used as destination in the Raster associated to the destination Context is determined by the destination context attribute `XGL_CTX_RENDER_BUFFER`. The entire source raster or a portion of it can be copied to the destination raster. The area that is copied is specified by a

rectangular region in the source buffer. It is copied into the destination buffer starting at the specified position. If the position is a `NULL` pointer, the top and left corner position is used. If the rectangle is a `NULL` pointer, the maximum area from the source buffer is copied.

Copying is only supported between buffers of the same type; in other words, pixels can only be copied between image buffers or between Z-buffers. When copying between image buffers, the destination Context plane mask, fill rule, and raster operation mode (`XGL_CTX_ROP`) attributes are applied during the copy operation. When copying between Z buffers, the destination Context Z comparison method (specified by the attribute `XGL_3D_CTX_Z_BUFFER_COMP_METHOD`) and Z write mask (specified by the attribute `XGL_3D_CTX_Z_BUFFER_WRITE_MASK`) are ignored during the copy operation.

Note – When copying to and from a Memory Raster, be sure that the Memory Raster has a Color Map that corresponds to its color type (`XGL_DEV_COLOR_TYPE`).

Note also that if the copy is from an 8-bit indexed Memory Raster to a 24-bit indexed Window Raster, separate Color Map objects are must be associated with the Rasters so that the XGL system can perform internal color map conversion correctly.

The operator `xgl_image()` copies a rectangular block of pixels from a source Memory Raster to the raster associated with the destination Context, which can be either a Memory Raster or a Window Raster. The operator performs the same operation as `xgl_context_copy_buffer()` does except that the location the pixels are copied to in the destination raster is expressed in the user's coordinate system, not in device coordinates. The `xgl_image()` operator displays pixel data starting at a given reference point in the user's Model Coordinates. In the case of 3D Context, `xgl_image()` can perform Z-buffering.

For more information about raster primitives and attributes, see the *XGL Reference Manual*.

Raster Primitive Example Programs

The following programs illustrate raster-level primitives. The program below shows how the `xgl_context_set_pixel()` and `xgl_context_get_pixel()` primitives are used. The program `copy_raster.c` on page 223 provides an example of the `xgl_context_copy_buffer()` primitive.

Example Program for Getting and Setting Pixels

Get and set pixel operations are performed in the example program, `gspixel.c`. This program sets a pixel at 10,10 from the upper left corner of the raster to yellow and then prints a message if the operation was successful. If the pixel is obscured, a message is printed. To compile this program, type `make gspixel` in the example program directory. The `make` command compiles `gspixel.c`, `gspixel_main.c`, and `ex_utils.c`. The latter two programs are listed in Appendix B.

Code Example 8-4 Example for Getting and Setting Pixels

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

#include "ex.h"


void
gspixel (Xgl_object      ctx)
{
    int                  i,
                         obscured;
    Xgl_color            set_pixel_color,
                         get_pixel_color;
    Xgl_pt_i2d           pos;

    /*
     * set pixel at 10, 10 to index value YELLOW
     */
    pos.x = pos.y = 10;
```

```
    set_pixel_color = yellow_color;
    xgl_context_set_pixel (ctx, &pos, &set_pixel_color);

    /*
     * get pixel from same location and return flag
     */
    obscured = xgl_context_get_pixel (ctx, &pos, &get_pixel_color);
    if (obscured) {
        /*
         * the pixel we are interested in covered by another
         * window or something else thus the color index
         * value in get_pixel_color is most likely incorrect
         */
        printf ("Pixel obscured\n");
    }
    else {
        /*
         * the pixel we read is NOT obscured therefore
         * we check to see if the value is YELLOW
         */
        if (ex_color_type == XGL_COLOR_INDEX &&
            get_pixel_color.index != YELLOW_INDEX) {
            /*
             * we shouldn't get into this part of the if
             * because the pixel value should be YELLOW
             */
            printf ("INDEX: Get pixel error\n");
        }
        else if (ex_color_type == XGL_COLOR_RGB &&
            get_pixel_color.rgb.r != 1. &&
            get_pixel_color.rgb.g != 1.) {
            /*
             * pixel value from get pixel function is equal
             * to pixel value used in set pixel function
             */
            printf ("RGB: Get pixel error\n");
        }
        else {
            printf ("Got YELLOW\n");
        }
    }
}
```

Example Program for Copying Pixels

This example program, `copy_raster.c`, copies pixels from a Memory Raster into a Context. To compile this program, type `make copy_raster` in the example program directory. The `make` command compiles `copy_raster.c`, `copy_raster_main.c`, and `ex_utils.c`.

Code Example 8-5 Example for Copying Pixels

```
/*
 * copy_raster.c
 */

#include <xview/xview.h>
#include <math.h>

#include "ex.h"

#define MR_WIDTH        32
#define MR_HEIGHT       32
#define ARC_RADIUS      16
#define DEG2RAD(x)      (((x) * 3.141592654) / 180.)

void
copy_raster (Xgl_object  ctx)
{
    double            sqrt (double);
    int               i;

    Xgl_object        win_ras;   /* window raster object */

    Xgl_arc_list      multiarc;  /* Xgl arc data structure */
    Xgl_arc_f2d       arc;       /* Xgl 2D-float-arc data
                                      * structure */
    Xgl_pt_i2d        cp_ras_pos;/* position of copy raster */

    /*
     * memory rasters objects for various positions of mouth
     */
    Xgl_object        open_mem_ras;    /* Mouth fully open */
    Xgl_object        half_mem_ras;    /* Mouth half open */
    Xgl_object        closed_mem_ras;  /* Closed Mouth */

    Xgl_color_type    color_type = XGL_COLOR_INDEX;
```

```
Xgl_color            surf_front_color;

xgl_object_get (ctx, XGL_CTX_DEVICE, &win_ras);

/*
 * set context surface color to YELLOW, set the new frame
 * action to be "wait for vertical retrace" then "clear
 * the area of the raster object associated with the context",
 * and increase number of subdivisions
 * for arcs so they look nice.
 */
surf_front_color = yellow_color;
xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &surf_front_color,
         XGL_CTX_NEW_FRAME_ACTION,
         XGL_CTX_NEW_FRAME_VRETRACE | XGL_CTX_NEW_FRAME_CLEAR,
         XGL_CTX_CURVE_APPROX_VALUE, 15.0,
         NULL);
/*
 * create memory raster used for source in copy raster
 * mouth open memory raster
 */
if (ex_win_depth == 24) ex_win_depth = 32;
if (ex_win_depth == 32) color_type = XGL_COLOR_RGB;

open_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                           XGL_DEV_COLOR_TYPE, color_type,
                           XGL_RAS_DEPTH, ex_win_depth,
                           XGL_RAS_WIDTH, MR_WIDTH,
                           XGL_RAS_HEIGHT, MR_HEIGHT,
                           NULL);
if (!open_mem_ras) {
    printf ("not enough memory (open memory raster)\n");
    return;
}

/*
 * mouth half open memory raster
 */
half_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                           XGL_DEV_COLOR_TYPE, color_type,
                           XGL_RAS_DEPTH, ex_win_depth,
                           XGL_RAS_WIDTH, MR_WIDTH,
                           XGL_RAS_HEIGHT, MR_HEIGHT,
                           NULL);
```

```
if (!half_mem_ras) {
    xgl_object_destroy (open_mem_ras);
    printf ("not enough memory (half memory raster)\n");
    return;
}

/*
 * mouth almost closed open memory raster
 */
closed_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                            XGL_DEV_COLOR_TYPE, color_type,
                            XGL_RAS_DEPTH, ex_win_depth,
                            XGL_RAS_WIDTH, MR_WIDTH,
                            XGL_RAS_HEIGHT, MR_HEIGHT,
                            NULL);
if (!closed_mem_ras) {
    xgl_object_destroy (open_mem_ras);
    xgl_object_destroy (half_mem_ras);
    printf ("not enough memory (closed memory raster)\n");
    return;
}

/*
 * set context device to open mouth memory object then
 * clear the memory of the memory raster and draw open mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, open_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * set multi-arc data structure to:
 *          1. one arc
 *          2. arc type is float 2D
 *          3. no bounding box
 *          4. point to arc data structure
 */
multiarc.num_arcs = 1;
multiarc.type = XGL_MULTIARC_F2D;
multiarc.bbox = NULL;
multiarc.arcs.f2d = &arc;

/*
 * set arc draw values to:
 *          1. center of arc is center of memory raster
 *          2. flag is always false (no edges)
 *          3. radius
```

```
 */
arc.center.x = (float) ARC_RADIUS;
arc.center.y = (float) ARC_RADIUS;
arc.center.flag = FALSE;
arc.radius = (float) ARC_RADIUS;

/*
 * by default, xgl draws arcs in a clockwise direction
 * because y is down by default and xgl always draws
 * arcs in a right-handed fashion open mouth arc
 */
arc.start_angle = DEG2RAD (45.); /* xgl draws in radians */
arc.stop_angle = DEG2RAD (315.); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set context device to half mouth memory object then
 * clear memory of memory raster and draw half mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, half_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * half open mouth arc
 */
arc.start_angle = DEG2RAD (22.5); /* xgl draws in radians */
arc.stop_angle = DEG2RAD (337.5); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set context device to closed mouth memory object then
 * clear memory of memory raster and draw closed mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, closed_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * (virtually) closed open mouth arc
 */
arc.start_angle = DEG2RAD (2.);   /* xgl draws in radians */
arc.stop_angle = DEG2RAD (358.5); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set device to saved raster and copy memory rasters to
 * context then clear the display
```

```
        */
        xgl_object_set (ctx, XGL_CTX_DEVICE, win_ras, NULL);
        for (i = 0; i < 248; i += 4) {
           /*
            * clear the context raster area
            */
           xgl_context_new_frame (ctx);

           /*
            * compute position of source raster in
            * destination raster using an interesting path
            */
           cp_ras_pos.x = i;
           cp_ras_pos.y = 50;

           /*
            * draw the three arc memory rasters and wait for
            * user to actually see the varying arc types
            */
           xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                    open_mem_ras);
           sleep(0.5);

           xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                    half_mem_ras);
           sleep(0.5);

          xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                    closed_mem_ras);
    }

    xgl_context_new_frame (ctx);
    xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                             closed_mem_ras);

    /*
     * restore context device object then destroy
     * memory raster object
     */

    xgl_object_set (ctx, XGL_CTX_DEVICE, win_ras, NULL);
    xgl_object_destroy (open_mem_ras);
    xgl_object_destroy (half_mem_ras);
    xgl_object_destroy (closed_mem_ras);
}
```

8

Accumulation Buffer

The XGL Raster object provides antialiasing of images with an accumulation buffer. The accumulation process takes a series of images and combines them using specified weights. Antialiasing by accumulating images in a buffer is not a real-time operation, but the quality of antialiasing can be seen to improve over time.

The accumulation buffer is a buffer containing an array of pixels where the contents of the accumulation process is stored. As an initial step in the accumulation process, the XGL application calls a series of XGL surface primitives and transfers the resulting image to the accumulation buffer. The application then changes the jitter offset and calls the same surface primitives again. Each rendering pass is jittered (moved) a subpixel distance from the previous pass and accumulated. The more passes that occur, the better the final image. The accumulation process involves weighted addition of the image buffer to the contents of the accumulation buffer. When slightly different images are averaged together, the resulting image is effectively antialiased. Typically, the application program accumulates into the accumulation buffer multiple times to produce an antialiased image and then copies the image to the display buffer. The process is complete when the image has an acceptable visual appearance.

The jitter offsets are determined by the application. The number of cycles for an acceptable image may be 60 or more for a 1× 1 convolution filter (in other words, no convolution filter is applied). For some 3 × 3 convolution filters, eight cycles may be enough. Some hardware may have support for fast rendering but no support for convolution filtering; other hardware may support convolution filtering. On most hardware devices, a 1 × 1 convolution filter (no filtering) is used in accumulation. Please check your frame buffer documentation to determine whether a filter size other than 1 × 1 is supported.

The accumulation buffer enables an application to antialiase filled surfaces. It can also be used to render motion blur, depth of field, or soft shadows. Only one accumulation buffer is associated with an XGL Raster and, similarly, only one Raster is associated with an accumulation buffer. An accumulation buffer has the same width and height as the associated raster's image buffer.

Note – Accumulation buffer functionality is currently only available on RGB rasters.

Accumulation Operator

The operator `xgl_context_accumulate()` accumulates pixel information from the draw buffer of a raster to the destination buffer of the raster as specified by the `XGL_3D_CTX_ACCUM_OP_DEST` attribute. When the accumulation operation is complete, the contents of the accumulation buffer can optionally be copied to an image buffer.

The accumulation calculation is:

*(destination buffer) = (source buffer * source weight + destination buffer * accum weight)*

The source buffer is always the Context's draw buffer, and the destination buffer is determined by the attribute `XGL_3D_CTX_ACCUM_OP_DEST`. The weights are specified as input parameters to the operator.

The accumulation buffer is created in the raster when `XGL_3D_CTX_ACCUM_OP_DEST` is set to `XGL_BUFFER_SEL_ACCUM` from one of the Context's to which the raster is attached and the first time either `xgl_context_accumulate()` or `xgl_context_clear_accumulation()` is called. The accumulation buffer is destroyed when the raster is destroyed.

This accumulation operator is defined as:

```
void xgl_context_accumulate (
    Xgl_ctx             ctx,
    Xgl_bounds_i2d      *rectangle,
    Xgl_pt_i2d          *pos,
    float               src_wt,
    float               accum_wt,
    Xgl_buffer_sel      buf);
```

The arguments are defined as follows:

- `ctx` specifies the Context to which the source raster is attached and whose draw buffer is used as the source in the accumulation process. The destination buffer (specified by the attribute `XGL_3D_CTX_ACCUM_OP_DEST`) is used as the destination buffer in the accumulation operation.
- `rectangle` specified by xmin, xmax, ymin, ymax gives the source area of the draw buffer of the raster attached to the Context. If the value is `NULL`, the maximum area of the raster is used.

- pos specifies the position in the destination buffer to be used as the starting position. The width and height up to a maximum of width and height of the rectangle is accumulated. If any region exceeds the bounds of the raster, it is correspondingly clipped. If the value of NULL, the to and left corners of the raster are used.
- src_wt is the weight used as the source weight in the accumulation calculation above.
- accum_wt is the weight used as the accumulation weight in the accumulation calculation above.
- buf specifies the buffer that the accumulated image is copied to. buf can be set to one of the following:

  XGL_BUFFER_SEL_NONE
  XGL_BUFFER_SEL_DISPLAY
  A value of 0 to N-1, where N is the number of buffers allocated.

To copy the accumulated image, set buf to XGL_BUFFER_SEL_DISPLAY. When copying is specified, it is done from the accumulated buffer to the buffer specified by buf. The rectangle starting from position pos and the width and height up to a maximum of the width and height of the rectangle is the source area for copy. The destination area starts at {xmin,ymin} of the rectangle and extends to the maximum width and height specified by rectangle.

XGL_BUFFER_SEL_NONE specifies that no pixel data is copied back to the image buffer.

The accumulation buffer can be cleared using the xgl_context_clear_accumulation() operator. This operator clears the buffer specified by the attribute XGL_3D_CTX_ACCUM_OP_DEST to the value value. The operator is defined as:

```
xgl_context_clear_accumulation (
    Xgl_ctx         ctx,
    Xgl_color*      value);
```

The destination buffer can also be cleared by setting both the src_wt and the accum_wt to zero in the xgl_context_accumulate() operator.

Accumulation Attributes

The attribute XGL_3D_CTX_ACCUM_OP_DEST specifies the buffer that is used as the destination buffer during the accumulation operation and the clear operation. If the value of this operator is set to XGL_BUFFER_SEL_ACCUM, the accumulation buffer is used as the destination buffer for xgl_context_accumulate() and xgl_context_clear_accumulation(). The accumulation buffer is created the first time xgl_context_accumulate() or xgl_context_clear_accumulation() is called and XGL_3D_CTX_ACCUM_OP_DEST is set to XGL_BUFFER_SEL_ACCUM. If the value of XGL_3D_CTX_ACCUM_OP_DEST is set to XGL_BUFFER_SEL_DISPLAY, the current display buffer is used as the destination buffer for xgl_context_accumulate() and xgl_context_clear_accumulation(). The accumulation destination can also be set to XGL_BUFFER_SEL_NONE. The default value is XGL_BUFFER_SEL_NONE.

The attribute XGL_RAS_ACCUM_BUFFER_DEPTH is a read-only attribute identifying the depth of the accumulation buffer. The depth of the accumulation buffer is a multiple of the raster depth and is device-dependent. XGL_RAS_ACCUM_BUFFER_DEPTH can be XGL_ACCUM_DEPTH_1X, which implies that the depth of the accumulation buffer is at least the depth of the raster, or XGL_ACCUM_DEPTH_2X, which implies that the depth of the accumulation buffer is at least twice the depth of the raster.

The attribute XGL_3D_CTX_JITTER_OFFSET is a 3D Context attribute that specifies the amount by which the geometry should be offset in Device Coordinates before rendering. An application can use jitter offset to render an image before accumulating.

Accumulation Example

The following code fragment accumulates data into an accumulation buffer in double buffer mode. After the last pass, the accumulated image is copied to the display buffer.

Code Example 8-6 Accumulation Buffer Example

```
Xgl_object        win_ras;
Xgl_object        context;
Xgl_pt_d2d        jitter;
Xgl_accum_depth   depth;
Xgl_color         value;

win_ras = xgl_object_create(sys_st, XGL_WIN_RAS,&obj_desc, NULL);
context = xgl_object_create(sys_st, XGL_3D_CTX,  &obj_desc,
                  XGL_CTX_DEVICE, win_ras,
                  NULL);

/*
 * Set the destination to be the accumulation buffer
 */
xgl_object_set(context, XGL_3D_CTX_ACCUM_OP_DEST,
               XGL_BUFFER_SEL_ACCUM, 0);

/*
 *   The Jitter is default {0.0,0.0} on the context during the
 *   first rendering operation
 */
xgl_context_new_frame(context);
render_image(context);

/*
 * Clear the entire accumulation buffer to a value to zero
 */
value.rgb.r = 0.0;
value.rgb.g = 0.0;
value.rgb.b = 0.0;
xgl_context_clear_accumulation_buffer(context, value);

/*
* Set the Jitter offset and draw in to the draw buffer (buffer 1)
*/
jitter.x = -1./3.;
jitter.y = 0.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 0.50, 0.5,
              XGL_BUFFER_SEL_NONE);
```

```
jitter.x = 0.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./3., 2./3.,
              XGL_BUFFER_SEL_NONE);

jitter.x = 1./3.;
jitter.y = 0.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./4., 3./4.,
              XGL_BUFFER_SEL_NONE);

jitter.x = 0.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./5., 4./5.,
              XGL_BUFFER_SEL_NONE);

jitter.x = -1./3.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./6., 5./6.,
              XGL_BUFFER_SEL_NONE);

jitter.x = -1./3.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./7., 6./7.,
              XGL_BUFFER_SEL_NONE);

jitter.x = 1./3.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0)
xgl_context_new_frame(context);
render_image(context);
```

```
xgl_context_accumulate(context, NULL, NULL, 1./8., 7./8.,
                XGL_BUFFER_SEL_NONE);

/*
 * Accumulate and copy to display buffer (buffer 0)
 */
jitter.x = 1./3.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./9., 8./9.,
                XGL_BUFFER_SEL_DISPLAY);
```