NAME

xgl_triangle_strip – draws a triangle strip

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_triangle_strip (
            Xgl_ctx          ctx,
            Xgl_facet_list   *facets,
            Xgl_pt_list      *pl );
```

DESCRIPTION

Purpose

This operator draws a triangle strip according to the parameter list and the graphics state described in the Context *ctx*. Only 3D Contexts can be used with this operator.

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.

More precisely, a triangle strip consists of an ordered list of (n+2) vertices $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, v_{n+1}\}$, $n > 0$. The first edge lies between the vertices $[\mathbf{v}_0, \mathbf{v}_1]$. For $i = 2, 3, \ldots, n+1$, each vertex $\mathbf{v}_i$ results in two new edges: $[\mathbf{v}_{i-2}, \mathbf{v}_i]$, and $[\mathbf{v}_{i-1}, \mathbf{v}_i]$. Thus, there are n triangles, $\mathbf{t}_0, \mathbf{t}_1, \ldots, t_{n-1}$, containing the vertices $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$, $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$, $\{\mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$, ... , $\{\mathbf{v}_{n-1}, \mathbf{v}_n, \mathbf{v}_{n+1}\}$. The triangles are drawn in the order $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$.

Parameters

ctx The XGL 3D Context containing the graphics state used when processing **xgl_triangle_strip**(3)**.**

facets A pointer to an *Xgl_facet_list* structure, which is defined as:

```
typedef struct {
        Xgl_facet_type          facet_type;         /* type of facet */
        Xgl_usgn32              num_facets;         /* number of facets *
        union {                                     /* pointers to array o
                Xgl_color_facet         *color_facets;
                Xgl_normal_facet        *normal_facets;
                Xgl_color_normal_facet  *color_normal_facets;
        } facets;
} Xgl_facet_list;
```

facets→facet_type is the data type used to define each facet in the list. *Xgl_facet_type* defines four enumerated values as possible facet data types:

XGL_FACET_NONE,
XGL_FACET_COLOR,
XGL_FACET_NORMAL,
XGL_FACET_COLOR_NORMAL,

The value of this field determines the interpretation of the union *facets→facets*, which points to the first element of an array of facet information. For example, if *facets→facet_type* is XGL_FACET_COLOR, *facets→facets.color[i]* should be the data for the *i+1'th* Xgl_color_facet.

This array of facet data is assumed to be *facets→num_facets* elements long.

The *facets* information is not used to draw the triangles if either *facets* is NULL, *facets→facet_type* is XGL_FACET_NONE, or *facets→num_facets* is less than *pl→num_pts* – 2. Otherwise, the data in the *i'th* facet is used for triangle $\mathbf{t}_i$.

pl A pointer to an *Xgl_pt_list* structure, which is defined as

```
typedef struct {
        Xgl_pt_type     pt_type;        /* type of point */
        Xgl_bbox        *bbox;          /* optional box containing
                                           all points */
        Xgl_usgn32      num_pts;        /* number of points */
        Xgl_usgn32      num_data_values; /* number of data values at
                                           each point */
        Xgl_pt_ptr_union pts;           /* union of pointers to array
                                           of points */
} Xgl_pt_list;
```

pl→pt_type specifies the type of the points in *pl→pts*.

pl→bbox specifies a bounding box for the triangle strip. This parameter is either NULL, or it is the bounding box around all of the points in *pl→pts*. The bounding box information can be used to optimize clipping and also to simplify rendering when its projection on the view surface is small. (See **XGL_CTX_THRESHOLD**(3)**.**)

If the application does not want to provide bounding box information, *bbox* must be NULL. If the bounding box is given, it is the application's responsibility to ensure that it is correct.

In releases of XGL before 3.0, if an application provided a bounding box, then *bbox* was a pointer to a structure of type *Xgl_bbox*. In XGL 3.0 and subsequent releases, this approach is still supported for backward compatibility, but applications now can store bounding box information in any of six new structures, which are similar to *Xgl_bbox*:

Xgl_bbox_i2d
Xgl_bbox_f2d
Xgl_bbox_f3d
Xgl_bbox_status
Xgl_bbox_d2d
Xgl_bbox_d3d

In place of a structure of type *Xgl_bbox*, an application can use one of the above structures. For example, if an application uses a variable called *bbox_f2d* of type *Xgl_bbox_f2d*, it needs to cast the pointer to this structure using (Xgl_bbox *) (&bbox_f2d) before passing the information to XGL.

pl→num_pts is the number of points in the array *pts*. Specifically, it is the number of vertices in the triangle strip. Note that it is *not* the number of triangles.

pl→num_data_values specifies the number of data values. Specifically, it is the number of data values per vertex in the triangle strip.

Texture mapping can be done by using vertices with data values. The data

values, also known as *u* and *v* values for texturing, specify how the texture map should be mapped onto triangles of a triangle strip. For more information on texturing, see **xgl_polygon**(3)**.**

pl→pts is a union of pointers, one of which points to the array of points that make up the vertices of the *(i-2)th* triangle in the triangle strip. The value of *pl→pt_type* determines the interpretation of this union.

See **xgl_polygon**(3) for further details on this union of pointers, and the effect of *pl→pt_type* and *facets→facet_type* on how facets are colored. These details are the same for all primitives that draw polygonal surfaces.

Execution

If points with per-vertex flag information are used, several bits within the per-vertex flag determine rendering characteristics on a per-triangle basis.

XGL_DRAW_EDGE (0x1)
XGL_DRAW_PREV_EDGE (0x2)

These bits control edge drawing as shown below. They will be used to turn on specifically marked edges when the Context edge flag, **XGL_CTX_SURF_EDGE_-FLAG**(3)**,** is also set.

For vertex $\mathbf{v}_0$, the edge flag bits are ignored.

For vertex $\mathbf{v}_1$, the edge flag bit XGL_DRAW_EDGE (0x1) controls the edge [$\mathbf{v}_0$, $\mathbf{v}_1$], while the bit XGL_DRAW_PREV_EDGE (0x2) is ignored.

For vertices v_i , $i = 2$ to n+1, both edge flag bits are used. The flag bit XGL_DRAW_-EDGE controls the edge [$\mathbf{v}_{i-1}$, $\mathbf{v}_i$] while the bit XGL_DRAW_PREV_EDGE controls the edge [$\mathbf{v}_{i-2}$, $\mathbf{v}_i$].

XGL_EDGE_IS_INTERNAL (0x20)
XGL_PREV_EDGE_IS_INTERNAL (0x40)

These bits are similar to the bits XGL_DRAW_EDGE and XGL_DRAW_PREV_EDGE because they control edge drawing. They are used to mark specific edges as *internal* edges. An application developer who is performing some kind of triangular tessellation may want to mark some edges as being internal because they were not part of the non-tessellated surface. Internal edges will never be rendered no matter what the state of the other flags. The semantics defining which edges that these bits correspond to is identical to that shown for XGL_DRAW_EDGE and XGL_DRAW_PREV_EDGE (see above).

The per-vertex bits can be manipulated using macros defined in the header file *xgl.h*. For instance, the macros XGL_SET_EDGE_FLAG(PT) and XGL_SET_PREV_EDGE_FLAG(PT) can be used to set the per-vertex edge flag bits when given a pointer to a point containing flag information. Likewise, macros are available to clear the flag bits. For more information, see *xgl.h*.

FACET COLORING

Generally, a *facet* refers to a single triangle within the triangle strip. The Context attributes specify how this facet is to be colored. The vertex point type *pl[i].pt_type* and the facet type *facet_type* also interact to affect the color of the triangle. The model is simple:

The point and facet types determine how the normal and base colors of each vertex are obtained. Then the Context attributes are applied to color the facet. For more information about facet coloring, see **xgl_polygon**(3).

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_multi_simple_polygon(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)

XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_3D_CTX_DEPTH_CUE_INTERP(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_REF_PLANES(3)
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS(3)
XGL_3D_CTX_LIGHTS(3)
XGL_3D_CTX_LIGHT_NUM(3)
XGL_3D_CTX_LIGHT_SWITCHES(3)
XGL_3D_CTX_SURF_BACK_AMBIENT(3)
XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR(3)
XGL_3D_CTX_SURF_BACK_DIFFUSE(3)
XGL_3D_CTX_SURF_BACK_DMAP(3)
XGL_3D_CTX_SURF_BACK_DMAP_NUM(3)
XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES(3)
XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_3D_CTX_SURF_BACK_FPAT(3)
XGL_3D_CTX_SURF_BACK_FPAT_POSITION(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT(3)
XGL_3D_CTX_SURF_BACK_SPECULAR(3)
XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR(3)
XGL_3D_CTX_SURF_BACK_SPECULAR_POWER(3)
XGL_3D_CTX_SURF_BACK_TRANSP(3)
XGL_3D_CTX_SURF_FACE_CULL(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
XGL_3D_CTX_SURF_FRONT_AMBIENT(3)
XGL_3D_CTX_SURF_FRONT_DIFFUSE(3)
XGL_3D_CTX_SURF_FRONT_DMAP(3)
XGL_3D_CTX_SURF_FRONT_DMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT(3)
XGL_3D_CTX_SURF_FRONT_SPECULAR(3)
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR(3)

XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER(3)
XGL_3D_CTX_SURF_FRONT_TRANSP(3)
XGL_3D_CTX_SURF_GEOM_NORMAL(3)
XGL_3D_CTX_SURF_NORMAL_FLIP(3)
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3)
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ(3)
XGL_3D_CTX_SURF_TRANSP_METHOD(3)
XGL_CTX_EDGE_AA_BLEND_EQ(3)
XGL_CTX_EDGE_AA_FILTER_SHAPE(3)
XGL_CTX_EDGE_AA_FILTER_WIDTH(3)
XGL_CTX_EDGE_ALT_COLOR(3)
XGL_CTX_EDGE_CAP(3)
XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_JOIN(3)
XGL_CTX_EDGE_MITER_LIMIT(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_CTX_SURF_AA_BLEND_EQ(3)
XGL_CTX_SURF_AA_FILTER_SHAPE(3)
XGL_CTX_SURF_AA_FILTER_WIDTH(3)
XGL_CTX_SURF_EDGE_FLAG(3)
XGL_CTX_SURF_FRONT_COLOR(3)
XGL_CTX_SURF_FRONT_COLOR_SELECTOR(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_INTERIOR_RULE(3)
XGL_CTX_THRESHOLD(3)

NOTES

All points in *pl[]* must have the same *Xgl_pt_type*.

2D point types will *not* work with a 3D Context.

The XGL drawing primitives **xgl_quadrilateral_mesh**(3)**, xgl_triangle_strip**(3)**,** and **xgl_-triangle_list**(3) may not draw edges of adjoining facets when *z*-buffering is not enabled. This problem occurs on certain devices or when using XGL to render across a network. To avoid this problem applications using these operators should either enable *z*-buffering or convert the quadrilateral mesh, triangle strip, or triangle list data into individual polygon data and call a polygon primitive (**xgl_multi_simple_polygon**(3) or **xgl_-polygon**(3)) with *z*-buffering disabled. Both approaches will render each facet of the mesh, strip, or list individually and with correct edges.

The XGL drawing primitives **xgl_quadrilateral_mesh**(3)**, xgl_triangle_strip**(3)**,** and **xgl_-triangle_list**(3) may produce inconsistent results when used with an RGB Raster (**XGL_-DEV_COLOR_TYPE**(3) is XGL_COLOR_RGB) on the GX graphics accelerator. This will occur when there is no clipping of the object, depth cueing is off, and *z*-buffering is disabled. In this case, XGL fully accelerates its rendering and does not dither the RGB colors before rendering them on the device. To avoid this problem, either use an Indexed Raster or enable *z*-buffering.