NAME

xgl_quadrilateral_mesh – draws a quadrilateral mesh

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_quadrilateral_mesh (
        Xgl_ctx          ctx,
        Xgl_usgn32       row_dim, col_dim,
        Xgl_facet_list   *facets,
        Xgl_pt_list      *pl );
```

DESCRIPTION

Purpose

This operator draws a quadrilateral (quad) mesh according to the parameter list and the graphics state described in the Context *ctx*, which can only be a 3D Context.

A quad mesh is defined by a single point list containing at least four vertices. Four vertices define a quad, and additional vertices define additional quads. Each quad should be planar.

More precisely, a quad mesh consists of a matrix of m rows and n columns ($m, n > 2$) of vertices

$$\begin{matrix} \mathbf{v}_{0,0} & \mathbf{v}_{0,1} & \cdots & \mathbf{v}_{0,n-1} \\ \mathbf{v}_{1,0} & \mathbf{v}_{1,1} & \cdots & \mathbf{v}_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{v}_{m-1,0} & \mathbf{v}_{m-1,1} & \cdots & \mathbf{v}_{m-1,n-1} \end{matrix}$$

stored as an ordered list of $m \times n$ vertices

$$\{\mathbf{v}_{0,0}, \mathbf{v}_{0,1}, \cdots, \mathbf{v}_{0,n-1}, \mathbf{v}_{1,0}, \mathbf{v}_{1,1}, \cdots, \mathbf{v}_{1,n-1}, \cdots, \mathbf{v}_{m-1,0}, \mathbf{v}_{m-1,1}, \ldots, \mathbf{v}_{m-1,n-1}\}.$$

The quad mesh above defines $(m-1)\times(n-1)$ quads $\mathbf{q}_k$ (k=0, 1,..., $(m-1)(n-1)-1$) formed from each square of adjacent vertices

$$\begin{matrix} \mathbf{v}_{i,j} & \mathbf{v}_{i,j+1} \\ \mathbf{v}_{i+1,j} & \mathbf{v}_{i+1,j+1} \end{matrix}$$

where $i = \lfloor k/(m-1) \rfloor$ and $j = k \bmod (n-1)$. The quads are drawn in the order $\mathbf{q}_k$ (k=0, 1,..., $(m-1)(n-1)-1$).

Parameters

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

row_dim
The number of *vertex* rows in the quad mesh; equal to *m* in the above discussion.

col_dim The number of *vertex* columns in the mesh; equal to *n* in the above discussion.

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 quads if *facets* is NULL, if *facets→facet_type* is XGL_FACET_NONE, or if *facets→num_facets* is less than (*row_dim*–1)×(*col_dim*–1). Otherwise, the data in the *i'th* facet is used for quad $\mathbf{q}_i$.

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 quad mesh. This parameter is either NULL, or 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 ignored.

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

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 quads of a quadrilateral mesh. 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 quads in the mesh. The value of *pl→pt_type* determines the interpretation of this union.

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

If vertices with *flag* information are used, two bits within the flag determine if particular quadrilateral edges are drawn (1 means draw it). The bits can be manipulated using macros defined in the header file *xgl.h*. XGL_DRAW_EDGE and XGL_DRAW_PREV_EDGE specify the two edge flag bits (corresponding to 0x1 and 0x2, respectively). The macros XGL_SET_EDGE_FLAG(PT) and XGL_SET_PREV_EDGE_-FLAG(PT) can be used to set the flags when given a pointer to a point containing flag information. Likewise, macros are available to clear the flag bits. For more information, see the include file *xgl.h*.

The edge flag bits are used only if the attribute **XGL_CTX_SURF_EDGE_FLAG**(3) is TRUE. They are interpreted as follows:

The flag bits for vertex $\mathbf{v}_{0,0}$ are ignored.

For vertices $v_{i,0}$, $i > 0$, the flag bit XGL_DRAW_EDGE (0x1) controls the edge

$[\mathbf{v}_{i-1,0}, \mathbf{v}_{i,0}]$; bit XGL_DRAW_PREV_EDGE (0x2) is ignored.

For vertices $v_{0,j}$, $j > 0$, the flag bit XGL_DRAW_PREV_EDGE (0x2) controls the edge $[\mathbf{v}_{0,j-1}, \mathbf{v}_{0,j}]$; bit XGL_DRAW_EDGE (0x1) is ignored.

For vertices $v_{i,j}$, $i, j > 0$, the flag bit XGL_DRAW_EDGE (0x1) controls the edge $[\mathbf{v}_{i-1,j}, \mathbf{v}_{i,j}]$, and the flag bit XGL_DRAW_PREV_EDGE (0x2) controls the edge $[\mathbf{v}_{i,j-1}, \mathbf{v}_{i,j}]$.

FACET COLORING

Generally, a *facet* refers to a single polygon, which might be either a general or simple polygon. 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 polygon. 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_triangle_list(3)
xgl_triangle_strip(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 do 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, **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.

Non-planar quadrilaterals within a quadrilateral mesh may not be rendered properly on some devices.