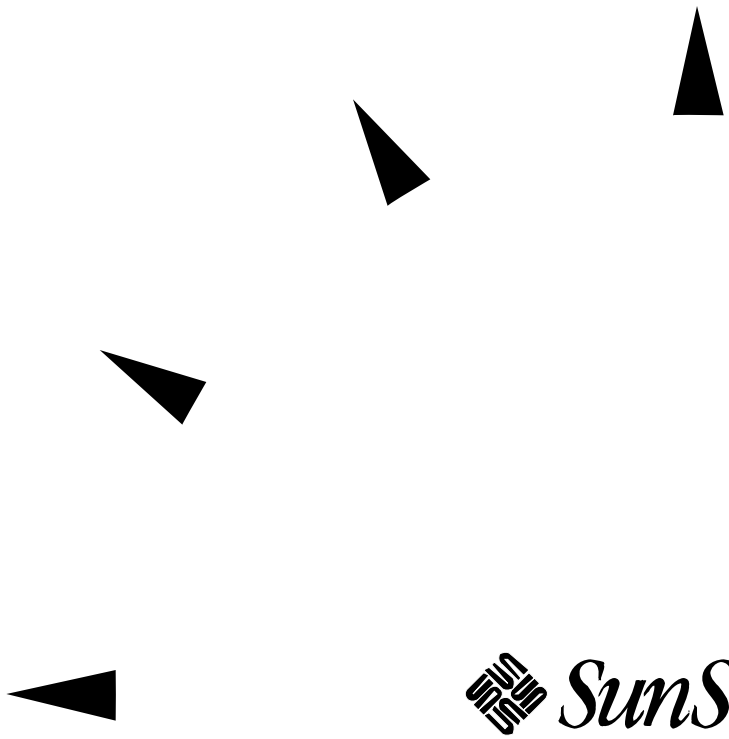
XGL Reference Manual



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

NAME

xgl_enum_types – XGL Enumerated Types

DESCRIPTION

This section lists the different C enumerated types XGL uses. The types appear with brief descriptions explaining their use, and they are listed by name in alphabetical order.

Accumulation Buffer Depth

```
typedef enum {
    XGL_ACCUM_DEPTH_1X,
    XGL_ACCUM_DEPTH_2X
} Xgl_accum_depth;
```

Arc Fill Styles

Choices available for **XGL_CTX_ARC_FILL_STYLE**(3)**.** Used when rendering arcs with **xgl_multiarc**(3)**.**

```
typedef enum {
    XGL_ARC_OPEN,
    XGL_ARC_SECTOR,
    XGL_ARC_CHORD
} Xgl_arc_fill_style;
```

Axes Used for Rotation Transforms

```
typedef enum {
    XGL_AXIS_X,
    XGL_AXIS_Y,
    XGL_AXIS_Z
} Xgl_axis;
```

Bounding Box Data Types

```
typedef enum {
    XGL_BBOX_I2D,
    XGL_BBOX_F2D,
    XGL_BBOX_F3D,
    XGL_BBOX_STATUS,
    XGL_BBOX_D2D,
    XGL_BBOX_D3D,
} Xgl_bbox_type;
```

Blend Equation

```
typedef enum {
    XGL_BLEND_NONE,
    XGL_BLEND_ARBITRARY_BG,
    XGL_BLEND_CONST_BG,
    XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

Blend Draw Modes

```
typedef enum {
    XGL_BLEND_DRAW_ALL,
    XGL_BLEND_DRAW_NOT_BLENDED,
    XGL_BLEND_DRAW_BLENDED
} Xgl_blend_draw_mode;
```

Status Returned by xgl_context_display_gcache

```
typedef enum {
    XGL_CACHE_NOT_CHECKED,              /* displayed but not check */
    XGL_CACHE_DISPLAY_OK,               /* display OK on this ctx */
    XGL_CACHE_ATTR_STATE_DIFFERENT,     /* something in this cache's
                                         * saved attribute state is
                                         * different from the state in ctx */
} Xgl_cache_display;
```

Homogeneous Color Type

```
typedef enum {
    XGL_COLOR_RGBW,
} Xgl_color_homogeneous_type;
```

Color Types

```
typedef enum {
    XGL_COLOR_INDEX,
    XGL_COLOR_GRAY,
    XGL_COLOR_RGB,
    XGL_COLOR_Z
} Xgl_color_type;
```

Curve Approximation Methods

Choices available for **XGL_CTX_NURBS_CURVE_APPROX**(3)**.** Used when rendering curves with **xgl_nurbs_curve**(3)**,** and when rendering 3D circles using **xgl_multicircle**(3)**.**

```
typedef enum {
    XGL_CURVE_UNUSED,   /* was XGL_CURVE_CONST_PARAM_SUBDIV */
    XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS,
    XGL_CURVE_METRIC_WC,
    XGL_CURVE_METRIC_VDC,
    XGL_CURVE_METRIC_DC,
    XGL_CURVE_CHORDAL_DEVIATION_WC,
    XGL_CURVE_CHORDAL_DEVIATION_VDC,
    XGL_CURVE_CHORDAL_DEVIATION_DC,
    XGL_CURVE_RELATIVE_WC,
    XGL_CURVE_RELATIVE_VDC,
    XGL_CURVE_RELATIVE_DC
} Xgl_curve_approx;
```

Data Types

The data types supported within XGL for geometric representation of data, transformations, and internal calculations.

```
typedef enum {
    XGL_DATA_INT,      /* 32-bit integer */
    XGL_DATA_FLT       /* single precision floating point */
    XGL_DATA_DBL       /* double precision floating point */
} Xgl_data_type;
```

Deferral Modes

Choices available for **XGL_CTX_DEFERRAL_MODE**(3)**.** Used to specify whether to buffer rendering operations.

```
typedef enum {
    XGL_DEFER_ASTI,
    XGL_DEFER_ASAP
} Xgl_deferral_mode;
```

Depth Cue Modes

Choices available for **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.** Used to specify the kind of depth-cueing desired.

```
typedef enum {
    XGL_DEPTH_CUE_OFF,
    XGL_DEPTH_CUE_LINEAR,
    XGL_DEPTH_CUE_SCALED
} Xgl_depth_cue_mode;
```

Graphics Library Error Categories

```
typedef enum {
    XGL_ERROR_SYSTEM,
    XGL_ERROR_CONFIGURATION,
    XGL_ERROR_RESOURCE,
    XGL_ERROR_ARITHMETIC,
    XGL_ERROR_USER
} Xgl_error_category;
```

Error Types

```
typedef enum {
    XGL_ERROR_RECOVERABLE,
    XGL_ERROR_NONRECOVERABLE
} Xgl_error_type;
```

Facet Types

Describes type of information available with polygonal facet data.

```
typedef enum {
    XGL_FACET_NONE,
    XGL_FACET_COLOR,
```

```
    XGL_FACET_NORMAL,
    XGL_FACET_COLOR_NORMAL,
} Xgl_facet_type;
```

Filter Shape

```
typedef enum {
    XGL_FILTER_GAUSSIAN
} Xgl_filter_shape;
```

Modes for Storing NURBS Geometry in Gcaches

```
typedef enum {
    XGL_GCACHE_NURBS_DYNAMIC,
    XGL_GCACHE_NURBS_COMBINED,
    XGL_GCACHE_NURBS_STATIC
} Xgl_gcache_nurbs_mode;
```

Method for Calculating Facet Normals

Choices available for **XGL_3D_CTX_SURF_GEOM_NORMAL**(3)**.** Defines how facet normals are calculated from vertex data if such normals are not provided by the application.

```
typedef enum {
    XGL_GEOM_NORMAL_FIRST_POINTS,
    XGL_GEOM_NORMAL_LAST_POINTS
} Xgl_geom_normal;
```

Hidden Line and Hidden Surface Mode

Choices available for **XGL_3D_CTX_HLHSR_MODE**(3)**.**

```
typedef enum {
    XGL_HLHSR_NONE,
    XGL_HLHSR_Z_BUFFER
} Xgl_hlhsr_mode;
```

Inquire Support Level

```
typedef enum {
    XGL_INQ_NOT_SUPPORTED,
    XGL_INQ_SOFTWARE,
    XGL_INQ_HARDWARE
} Xgl_inq_support_level;
```

Rule Used for Determining Interior of Filled Polygonal Surfaces

Choice available for **XGL_CTX_SURF_INTERIOR_RULE**(3)**.**

```
typedef enum {
    XGL_EVEN_ODD
} Xgl_interior_rule;
```

Isoparametric Curve Distribution

```
typedef enum {
    XGL_ISO_CURVE_BETWEEN_KNOTS,        /* u & v iso-curves spaced between knots */
    XGL_ISO_CURVE_BETWEEN_LIMITS        /* u & v iso-curves spaced between limits */
} Xgl_iso_curve_placement;
```

Light Object Types

```
typedef enum {
    XGL_LIGHT_AMBIENT,
    XGL_LIGHT_DIRECTIONAL,
    XGL_LIGHT_POSITIONAL,
    XGL_LIGHT_SPOT
} Xgl_light_type;
```

Line Cap Types

Choices available for **XGL_CTX_LINE_CAP**(3)**.** Used to specify the shape of the endpoints of a line.

```
typedef enum {
    XGL_CAP_BUTT,
    XGL_CAP_ROUND,
    XGL_CAP_SQUARE
} Xgl_line_cap;
```

Line Color Selector

Choices available for **XGL_CTX_LINE_COLOR_SELECTOR**(3)**.** Used to specify the source of line colors when rendering.

```
typedef enum {
    XGL_LINE_COLOR_CONTEXT,
    XGL_LINE_COLOR_VERTEX
} Xgl_line_color_sel;
```

Line Join Types

Choices available for **XGL_CTX_LINE_JOIN**(3)**.** Used to specify how line segments are joined together.

```
typedef enum {
    XGL_JOIN_MITER,
    XGL_JOIN_ROUND,
    XGL_JOIN_BEVEL,
    XGL_JOIN_DEVICE
} Xgl_line_join;
```

Line Patterns

Choices available for **XGL_CTX_EDGE_STYLE**(3) and **XGL_CTX_LINE_STYLE**(3)**.** Used to request rendering of patterned lines.

```
typedef enum {
    XGL_LINE_SOLID,
    XGL_LINE_PATTERNED,
    XGL_LINE_ALT_PATTERNED
} Xgl_line_style;
```

Line Pattern Coordinate System

Choice available for **XGL_LPAT_COORD_SYS**(3)**.**

```
typedef enum {
    XGL_LPAT_DC
} Xgl_lpat_coord_sys;
```

Line Pattern Styles

Choices available for **XGL_LPAT_STYLE**(3)**.**

```
typedef enum {
    XGL_LPAT_FIXED_OFFSET,
    XGL_LPAT_BALANCED_SEGMENT
} Xgl_lpat_style;
```

Line Pattern Balanced Dash

Choices available for **XGL_LPAT_BALANCED_DASH**(3)**.**

```
typedef enum {
    XGL_LPAT_BAL_DASH_0,
    XGL_LPAT_BAL_DASH_1
} Xgl_lpat_bal_dash;
```

Marker Color Selector

Choices available for **XGL_CTX_MARKER_COLOR_SELECTOR**(3)**.** Used to specify the source of marker colors when rendering.

```
typedef enum {
    XGL_MARKER_COLOR_CONTEXT,
    XGL_MARKER_COLOR_POINT
} Xgl_marker_color_sel;
```

Multiarc Types

```
typedef enum {
    XGL_MULTIARC_I2D,
    XGL_MULTIARC_F2D,
    XGL_MULTIARC_D2D,
    XGL_MULTIARC_F3D,
    XGL_MULTIARC_AF3D,
    XGL_MULTIARC_D3D,
    XGL_MULTIARC_AD3D
} Xgl_multiarc_type;
```

Multicircle Types

```
typedef enum {
    XGL_MULTICIRCLE_I2D,
    XGL_MULTICIRCLE_F2D,
    XGL_MULTICIRCLE_D2D,
    XGL_MULTICIRCLE_F3D,
    XGL_MULTICIRCLE_AF3D,
    XGL_MULTICIRCLE_D3D,
    XGL_MULTICIRCLE_AD3D
} Xgl_multicircle_type;
```

Multielliptical Arc Types

```
typedef enum {
    XGL_MULTIELLARC_F3D,
    XGL_MULTIELLARC_AF3D,
    XGL_MULTIELLARC_D3D,
    XGL_MULTIELLARC_AD3D
} Xgl_multiell_type;
```

Multirectangle Types

```
typedef enum {
    XGL_MULTIRECT_I2D,
    XGL_MULTIRECT_F2D,
    XGL_MULTIRECT_D2D,
    XGL_MULTIRECT_F3D,
    XGL_MULTIRECT_AF3D,
    XGL_MULTIRECT_D3D,
    XGL_MULTIRECT_AD3D
} Xgl_multirect_type;
```

NURBS Surface Types

```
typedef enum {
    XGL_SURF_NURBS,
    XGL_SURF_PLANAR,
    XGL_SURF_CYLINDRICAL,
    XGL_SURF_CONICAL,
    XGL_SURF_SPHERICAL
} Xgl_nurbs_surf_type;
```

XGL Object Types

```
typedef enum {
    XGL_OBJ,           /* Generic object (not valid in xgl_object_create()) */
    XGL_2D_CTX,        /* 2D Xgl context */
    XGL_3D_CTX,        /* 3D Xgl context */
    XGL_MEM_RAS,       /* Memory raster device */
    XGL_WIN_RAS,       /* Window raster device */
    XGL_STREAM,        /* Stream device */
    XGL_CMAP,          /* Color map */
    XGL_TRANS,         /* Transform */
```

```
    XGL_LIGHT,          /* Light */
    XGL_MARKER,         /* Marker */
    XGL_SFONT,          /* Stroke font */
    XGL_LPAT,           /* Line pattern */
    XGL_SYS_STATE,      /* System state */
    XGL_GCACHE,         /* Geometry Cache */
    XGL_DMAP_TEXTURE,   /* Data Map Texture */
    XGL_MIPMAP_TEXTURE /* MipMap Texture */
    XGL_TMAP            /* Texture Map */
    XGL_PCACHE          /* Primitive Cache */
} Xgl_obj_type;
```

Paint Types

Choices available for **XGL_CTX_PAINT_TYPE**(3)**.**

```
typedef enum {
    XGL_PAINT_TRANSPARENT,
    XGL_PAINT_OPAQUE
} Xgl_paint_type;
```

Picking Methods

Choices available for **XGL_CTX_PICK_STYLE**(3)**.** Used to specify how pick events are stored in the pick buffer.

```
typedef enum {
    XGL_PICK_LAST_N,
    XGL_PICK_FIRST_N
} Xgl_pick_style;
```

Picking Surface Styles

```
typedef enum {
    XGL_PICK_SURF_AS_SOLID,
    XGL_PICK_SURF_AS_FILL_STYLE
} Xgl_pick_surf_style;
```

Polygon Types

Choices available for **XGL_GCACHE_POLYGON_TYPE**(3)**.**

```
typedef enum {
    XGL_POLYGON_COMPLEX,        /* General, complex polygon */
    XGL_POLYGON_NSI             /* Not Self Intersecting complex polygon */
} Xgl_polygon_type;
```

Primitive Types

Choices available for **XGL_GCACHE_ORIG_PRIM_TYPE**(3) and **XGL_GCACHE_DISPLAY_PRIM_-TYPE**(3)**.**

```
typedef enum {
    XGL_PRIM_NONE,
    XGL_PRIM_MULTIMARKER,
    XGL_PRIM_MULTIPOLYLINE,
    XGL_PRIM_NURBS_CURVE,
    XGL_PRIM_MULTIRECTANGLE,
    XGL_PRIM_MULTIARC,
    XGL_PRIM_MULTICIRCLE,
    XGL_PRIM_ELLIPTICAL_ARC
    XGL_PRIM_MULTI_SIMPLE_POLYGON,
    XGL_PRIM_POLYGON,
    XGL_PRIM_TRIANGLE_LIST,
    XGL_PRIM_TRIANGLE_STRIP,
    XGL_PRIM_QUADRILATERAL_MESH,
    XGL_PRIM_NURBS_SURFACE,
    XGL_PRIM_STROKE_TEXT,
    XGL_PRIM_ANNOTATION_TEXT,
    XGL_PRIM_IMAGE,
} Xgl_primitive_type;
```

Point Types

```
typedef enum {
    XGL_PT_I2D,
    XGL_PT_COLOR_I2D,
    XGL_PT_FLAG_I2D,
    XGL_PT_I2H,
    XGL_PT_F2D,
    XGL_PT_COLOR_F2D,
    XGL_PT_FLAG_F2D,
    XGL_PT_F2H,
    XGL_PT_F3D,
    XGL_PT_COLOR_F3D,
    XGL_PT_NORMAL_F3D,
    XGL_PT_COLOR_NORMAL_F3D,
    XGL_PT_FLAG_F3D,
    XGL_PT_COLOR_FLAG_F3D,
    XGL_PT_NORMAL_FLAG_F3D,
    XGL_PT_COLOR_NORMAL_FLAG_F3D,
```

```
    XGL_PT_F3H,
    XGL_PT_DATA_F3D
    XGL_PT_COLOR_DATA_F3D
    XGL_PT_NORMAL_DATA_F3D
    XGL_PT_COLOR_NORMAL_DATA_F3D
    XGL_PT_FLAG_DATA_F3D
    XGL_PT_COLOR_FLAG_DATA_F3D
    XGL_PT_NORMAL_FLAG_DATA_F3D
    XGL_PT_COLOR_NORMAL_FLAG_DATA_F3D
    XGL_PT_D2D,
    XGL_PT_COLOR_D2D,
    XGL_PT_FLAG_D2D,
    XGL_PT_D2H,
    XGL_PT_D3D,
    XGL_PT_COLOR_D3D,
    XGL_PT_NORMAL_D3D,
    XGL_PT_COLOR_NORMAL_D3D,
    XGL_PT_FLAG_D3D,
    XGL_PT_COLOR_FLAG_D3D,
    XGL_PT_NORMAL_FLAG_D3D,
    XGL_PT_COLOR_NORMAL_FLAG_D3D,
    XGL_PT_D3H
} Xgl_pt_type;
```

Raster Fill Styles

Choices available for **XGL_CTX_RASTER_FILL_STYLE**(3)**.**

```
typedef enum {
    XGL_RAS_FILL_COPY,
    XGL_RAS_FILL_STIPPLE,
    XGL_RAS_FILL_OPAQUE_STIPPLE
} Xgl_raster_fill_style;
```

Color to Modify During Texture Mapping

```
typedef enum {
    XGL_RENDER_COMP_DIFFUSE_COLOR,
    XGL_RENDER_COMP_REFLECTED_COLOR,
    XGL_RENDER_COMP_FINAL_COLOR
} Xgl_render_component;
```

Rendering Order of Graphics Primitives

Choices available for **XGL_CTX_RENDERING_ORDER**(3)**.**

```
typedef enum {
    XGL_RENDERING_ORDER_GIVEN,
    XGL_RENDERING_ORDER_HLHSR,
    XGL_RENDERING_ORDER_ANY
} Xgl_rendering_order;
```

Stereo Setting

```
typedef enum {
    XGL_STEREO_NONE,
    XGL_STEREO_LEFT,
```

```
    XGL_STEREO_RIGHT
} Xgl_stereo_mode;
```

Surface Approximation Methods

```
typedef enum {
    XGL_SURF_UNUSED, /* was XGL_SURF_CONST_PARAM_SUBDIV */
    XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS,
    XGL_SURF_METRIC_WC,
    XGL_SURF_METRIC_VDC,
    XGL_SURF_METRIC_DC,
    XGL_SURF_PLANAR_DEVIATION_WC,
    XGL_SURF_PLANAR_DEVIATION_VDC,
    XGL_SURF_PLANAR_DEVIATION_DC,
    XGL_SURF_RELATIVE_WC,
    XGL_SURF_RELATIVE_VDC,
    XGL_SURF_RELATIVE_DC
} Xgl_surf_approx;
```

Surface Color Selector

Choices available for **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_3D_CTX_SURF_-BACK_COLOR_SELECTOR**(3)**.** Used to specify the source of surface colors when rendering.

```
typedef enum {
    XGL_SURF_COLOR_CONTEXT,
    XGL_SURF_COLOR_FACET,
    XGL_SURF_COLOR_VERTEX_ILLUM_DEP,
    XGL_SURF_COLOR_VERTEX_ILLUM_INDEP
} Xgl_surf_color_sel;
```

Surface Cull Modes

Choices available for **XGL_3D_CTX_SURF_FACE_CULL**(3)**.**

```
typedef enum {
    XGL_CULL_OFF,
    XGL_CULL_FRONT,
    XGL_CULL_BACK
} Xgl_surf_cull_mode;
```

Fill Styles for Polygonal Facets

Choices available for **XGL_CTX_SURF_FRONT_FILL_STYLE**(3) and other fill style attributes.

```
typedef enum {
    XGL_SURF_FILL_SOLID,
    XGL_SURF_FILL_STIPPLE,
    XGL_SURF_FILL_OPAQUE_STIPPLE,
    XGL_SURF_FILL_PATTERN,
    XGL_SURF_FILL_HOLLOW,
    XGL_SURF_FILL_EMPTY
} Xgl_surf_fill_style;
```

Illumination Type for Surfaces

Choices available for **XGL_3D_CTX_SURF_FRONT_ILLUMINATION**(3)**.**

```
typedef enum {
    XGL_ILLUM_NONE,
    XGL_ILLUM_NONE_INTERP_COLOR,
    XGL_ILLUM_PER_FACET,
    XGL_ILLUM_PER_VERTEX,
} Xgl_surf_illumination;
```

Stroke Text Types

Annotation text styles.

```
typedef enum {
    XGL_ATEXT_STYLE_NORMAL,
    XGL_ATEXT_STYLE_LINE
}Xgl_atext_style;
```

Horizontal text alignment for stroke fonts.

```
typedef enum {
    XGL_STEXT_ALIGN_HORIZ_LEFT,
    XGL_STEXT_ALIGN_HORIZ_CENTER,
    XGL_STEXT_ALIGN_HORIZ_RIGHT,
    XGL_STEXT_ALIGN_HORIZ_NORMAL
} Xgl_stext_align_horiz;
```

Vertical text alignment for stroke fonts.

```
typedef enum {
    XGL_STEXT_ALIGN_VERT_TOP,
    XGL_STEXT_ALIGN_VERT_CAP,
    XGL_STEXT_ALIGN_VERT_HALF,
    XGL_STEXT_ALIGN_VERT_BASE,
    XGL_STEXT_ALIGN_VERT_BOTTOM,
    XGL_STEXT_ALIGN_VERT_NORMAL
} Xgl_stext_align_vert;
```

Text path for stroke fonts.

```
typedef enum {
    XGL_STEXT_PATH_RIGHT,
    XGL_STEXT_PATH_LEFT,
    XGL_STEXT_PATH_UP,
```

```
    XGL_STEXT_PATH_DOWN
} Xgl_stext_path;
```

Text precision for stroke fonts.

```
typedef enum {
    XGL_STEXT_PRECISION_STROKE,
} Xgl_stext_precision;
```

Boundary Operation for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_BOUNDARY_CLAMP,
    XGL_TEXTURE_BOUNDARY_WRAP,
    XGL_TEXTURE_BOUNDARY_TRANSPARENT,
    XGL_TEXTURE_BOUNDARY_MIRROR,
    XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY,
} Xgl_texture_boundary;
```

Coordinate Source for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_COORD_VERTEX,
    XGL_TEXTURE_COORD_NORMAL,
    XGL_TEXTURE_COORD_DATA
} Xgl_texture_coord_source;
```

Domain for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_COLOR,
} Xgl_texture_domain;
```

Interpolation Method for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_INTERP_POINT,
    XGL_TEXTURE_INTERP_BILINEAR,
    XGL_TEXTURE_INTERP_MIPMAP_POINT,
    XGL_TEXTURE_INTERP_MIPMAP_BILINEAR,
    XGL_TEXTURE_INTERP_MIPMAP_TRILINEAR,
    XGL_TEXTURE_INTERP_MIPMAP_PT_LINEAR
} Xgl_texture_interp_method;
```

Operation for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_OP_NONE,
    XGL_TEXTURE_OP_MODULATE,
    XGL_TEXTURE_OP_REPLACE,
    XGL_TEXTURE_OP_DECAL,
    XGL_TEXTURE_OP_BLEND,
    XGL_TEXTURE_OP_DECAL_INTRINSIC,
```

```
    XGL_TEXTURE_OP_BLEND_INTRINSIC
} Xgl_texture_op;
```

Parameterization Type for Texture Mapping

```
typedef enum {
    XGL_TEXTURE_PARAM_EXPLICIT,
    XGL_TEXTURE_PARAM_NURBS_UNIFORM
} Xgl_texture_param_type;
```

Texture Perspective Correction

```
typedef enum {
    XGL_TEXTURE_PERSP_NONE,
    XGL_TEXTURE_PARAM_PIXEL
} Xgl_texture_persp_correction;
```

Type of Texture Mapping

```
typedef enum {
    XGL_TEXTURE_TYPE_MIPMAP
} Xgl_texture_type;
```

Transform Types

Transform spatial dimensions.

```
typedef enum {
    XGL_TRANS_2D,
    XGL_TRANS_3D
} Xgl_trans_dimension;
```

Methods for updating transforms using operators scale, rotate, translate.

```
typedef enum {
    XGL_TRANS_REPLACE,
    XGL_TRANS_PRECONCAT,
    XGL_TRANS_POSTCONCAT
} Xgl_trans_update;
```

Transparent Primitives Methods

```
typedef enum {
    XGL_TRANSP_NONE,
    XGL_TRANSP_SCREEN_DOOR,
    XGL_TRANSP_BLENDED
} Xgl_transp_method;
```

Trim Curve Approximation Methods

```
typedef enum {
   XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS,  /* static tessellation */
   XGL_TRIM_CURVE_VIEW_DEPENDENT                     /* dynamic tessellation */
```

```
} Xgl_trim_curve_approx;
```

Window Resizing Policy for Mapping VDC to DC

Choices available for **XGL_CTX_VDC_MAP**(3)**.**

```
typedef enum {
    XGL_VDC_MAP_OFF,
    XGL_VDC_MAP_ALL,
    XGL_VDC_MAP_ASPECT,
    XGL_VDC_MAP_DEVICE
} Xgl_vdc_map;
```

Orientations for Virtual Device Coordinates

Choices available for **XGL_CTX_VDC_ORIENTATION**(3)**.**

```
typedef enum {
    XGL_Y_DOWN_Z_AWAY,
    XGL_Y_UP_Z_TOWARD
} Xgl_vdc_orientation;
```

Z-buffer Comparison Methods

```
typedef enum {
    XGL_Z_COMP_LESS_THAN,
    XGL_Z_COMP_LESS_THAN_OR_EQUAL,
    XGL_Z_COMP_EQUAL,
    XGL_Z_COMP_GREATER_THAN,
    XGL_Z_COMP_GREATER_THAN_OR_EQUAL,
    XGL_Z_COMP_NOT_EQUAL,
    XGL_Z_COMP_NEVER,
    XGL_Z_COMP_ALWAYS
}  Xgl_z_comp_method;
```

SEE ALSO

xgl_macro_values(3)
xgl_pt_list(3)
xgl_struct(3)

NAME

xgl_macro_values – XGL macro values

DESCRIPTION

This section lists the different macro values XGL uses. The values appear with brief descriptions explaining their use, and they are listed by name in alphabetical order.

Buffer Selector

```
typedef         Xgl_sgn32       Xgl_buffer_sel;
    XGL_BUFFER_SEL_Z
    XGL_BUFFER_SEL_ACCUM
    XGL_BUFFER_SEL_DRAW
    XGL_BUFFER_SEL_DISPLAY
    XGL_BUFFER_SEL_NONE
```

Stroke Text Types

Character encoding schemes.

```
typedef         Xgl_usgn32      Xgl_char_encoding;
    XGL_CHAR_MBY
    XGL_CHAR_ISO
    XGL_SINGLE_STR
    XGL_MULTI_STR
```

Flags for Enabling View Clip Planes

```
typedef         Xgl_usgn32      Xgl_clip_planes;
    XGL_CLIP_XMIN
    XGL_CLIP_XMAX
    XGL_CLIP_YMIN
    XGL_CLIP_YMAX
    XGL_CLIP_ZMIN
    XGL_CLIP_ZMAX
```

Context New Frame Action

```
typedef         Xgl_usgn32       Xgl_ctx_new_frame_action;
    XGL_CTX_NEW_FRAME_NONE
    XGL_CTX_NEW_FRAME_VRETRACE
    XGL_CTX_NEW_FRAME_CLEAR
    XGL_CTX_NEW_FRAME_HLHSR_ACTION
    XGL_CTX_NEW_FRAME_SWITCH_BUFFER
```

Facet Flags

Describes the different facet flags to speed up polygon display.

```
typedef         Xgl_usgn32      Xgl_facet_flags;

/* mask and flags for facet sides */
    XGL_FACET_FLAG_SIDES_MASK
    XGL_FACET_FLAG_SIDES_UNSPECIFIED
    XGL_FACET_FLAG_SIDES_ARE_3
```

```
    XGL_FACET_FLAG_SIDES_ARE_4

/* mask and flags for convexity information */
    XGL_FACET_FLAG_SHAPE_MASK
    XGL_FACET_FLAG_SHAPE_UNKNOWN
    XGL_FACET_FLAG_SHAPE_NONCONVEX
    XGL_FACET_FLAG_SHAPE_CONVEX

/* mask and flags to indicate that Facet Normals are consistent */
    XGL_FACET_FLAG_FN_CONSISTENT_MASK
    XGL_FACET_FLAG_FN_NOT_CONSISTENT
    XGL_FACET_FLAG_FN_CONSISTENT

/* mask and flags for geometry data */
    XGL_FACET_FLAG_DATA_CONTIG_MASK
    XGL_FACET_FLAG_DATA_NOT_CONTIG
    XGL_FACET_FLAG_DATA_CONTIG

/* OR all facet flag mask */
    XGL_FACET_FLAG_ALL_MASKS
```

Flush Action

```
typedef         Xgl_usgn32      Xgl_flush_action;
    XGL_FLUSH_GEOM_DATA
    XGL_FLUSH_BUFFERS
    XGL_FLUSH_SYNCHRONIZE
```

Geometry Status

Geometry status for bounding box and facet orientation.

```
typedef         Xgl_usgn32      Xgl_geom_status;
    XGL_GEOM_STATUS_VIEW_ACCEPT
    XGL_GEOM_STATUS_VIEW_REJECT
    XGL_GEOM_STATUS_VIEW_SMALL
    XGL_GEOM_STATUS_MODEL_ACCEPT
    XGL_GEOM_STATUS_MODEL_REJECT
    XGL_GEOM_STATUS_FACING_FRONT
    XGL_GEOM_STATUS_FACING_BACK
```

Light Enable Component

```
typedef         Xgl_usgn32      Xgl_light_enable_component;
    XGL_LIGHT_ENABLE_COMP_AMBIENT
    XGL_LIGHT_ENABLE_COMP_DIFFUSE
    XGL_LIGHT_ENABLE_COMP_SPECULAR
```

Flags For Updating the Context Model Transforms

```
typedef         Xgl_usgn32      Xgl_model_trans_request;
    XGL_MTR_PUSH
    XGL_MTR_POP
    XGL_MTR_LOCAL_TRANS
    XGL_MTR_GLOBAL_TRANS
    XGL_MTR_NEW_LEVEL
```

NURBS Surface Styles

```
typedef         Xgl_usgn32      Xgl_nurbs_surf_param_style;
    XGL_SURF_PLAIN
    XGL_SURF_ISO_CURVES
    XGL_SURF_MC_LEVEL_CURVES
    XGL_SURF_WC_LEVEL_CURVES
    XGL_SURF_SHOW_TESSELLATION
    XGL_SURF_INCR_SILHOUETTE_TESS
    XGL_SURF_DEVICE_DEPENDENT
```

Render Mode

```
typedef         Xgl_usgn32      Xgl_render_mode;
    XGL_RENDER_DISPLAY_BUFFER
    XGL_RENDER_DRAW_BUFFER
    XGL_RENDER_Z_BUFFER
    XGL_RENDER_ACCUM_BUFFER
```

Raster Operations (Two Styles: Pixrect and X11)

```
typedef         Xgl_usgn32      Xgl_rop_mode;

/* Raster operations (Pixrect): XGL_ROP_SRC and XGL_ROP_DST can be combined
 * with the C the binary operators |, &, ^, and the macro XGL_ROP_NOT.
 */
    XGL_ROP_CLR
    XGL_ROP_SET
    XGL_ROP_NOT(op)
    XGL_ROP_SRC
    XGL_ROP_DST

/* Raster operations (X11) */
    XGL_ROP_CLEAR
    XGL_ROP_AND
    XGL_ROP_AND_REVERSE
    XGL_ROP_AND_INVERTED
    XGL_ROP_XOR
    XGL_ROP_OR
    XGL_ROP_NOR
    XGL_ROP_EQUIV
    XGL_ROP_INVERT
    XGL_ROP_OR_REVERSE
    XGL_ROP_COPY_INVERTED
    XGL_ROP_OR_INVERTED
    XGL_ROP_NAND
```

Triangle List Global Flag Word

```
typedef         Xgl_usgn32      Xgl_tlist_flags;

/* mutually exclusive tri_list flag bits */
    XGL_TLIST_FLAG_TRI_RESTART
    XGL_TLIST_FLAG_TRI_STRIP
    XGL_TLIST_FLAG_TRI_STAR
```

```
    XGL_TLIST_FLAG_USE_VTX_FLAGS
    XGL_TLIST_FLAG_TRI_MASK

/* other tri_list flag bits */
    XGL_TLIST_FLAG_EDGES_ON
    XGL_TLIST_FLAG_IS_PLANAR

/* mask and flag to indicate that Facet Normals are consistent */
    XGL_TLIST_FLAG_FN_CONSISTENT_MASK
    XGL_TLIST_FLAG_FN_NOT_CONSISTENT
    XGL_TLIST_FLAG_FN_CONSISTENT

    XGL_TLIST_FLAG_ALL_MASKS
```

Transform Memberships

```
typedef         Xgl_usgn32      Xgl_trans_member;
    XGL_TRANS_MEMBER_IDENTITY
    XGL_TRANS_MEMBER_TRANSLATION
    XGL_TRANS_MEMBER_SCALE
    XGL_TRANS_MEMBER_ROTATION
    XGL_TRANS_MEMBER_WINDOW
    XGL_TRANS_MEMBER_SHEAR_SCALE
    XGL_TRANS_MEMBER_LENGTH_PRESERV
    XGL_TRANS_MEMBER_ANGLE_PRESERV
    XGL_TRANS_MEMBER_AFFINE
    XGL_TRANS_MEMBER_LIM_PERSPECTIVE
```

Macros and Defines for Setting Vertex Flags

```
XGL_VERTEX_PICK_BITS          /* bits used to encode vertex pick info */

XGL_DRAW_EDGE                 /* edge from v(n) to v(n-1) */
XGL_DRAW_PREV_EDGE            /* other edge for strips/stars/quads */
XGL_DRAW_BOTH_EDGES

XGL_TRIANGLE_STRIP            /* per-vertex triangle star */
XGL_TRIANGLE_STAR             /* per-vertex triangle strip */
XGL_TRIANGLE_RESTART          /* per-vertex independent triangles */

/* For restartable triangles see XGL_RESTART_TRIANGLE(PT) below */
XGL_STAR_AND_BOTH_EDGES (XGL_TRIANGLE_STAR | XGL_DRAW_BOTH_EDGES)
XGL_STRIP_AND_BOTH_EDGES (XGL_TRIANGLE_STRIP | XGL_DRAW_BOTH_EDGES)

XGL_FACE_IS_CCW               /* face orientation */
XGL_EDGE_IS_INTERNAL          /* edge from v(n) to v(n-1) is internal */
XGL_PREV_EDGE_IS_INTERNAL     /* other edge is internal */

/* modify the following to set bits that copy convert should keep */
XGL_PRESERVED_FLAG_BITS (XGL_STAR_AND_BOTH_EDGES | XGL_VERTEX_PICK_BITS)

/* use the following macros to set and get pick vertex values from vertex flags */
XGL_SET_VERTEX_PICK(PICK_VAL) (PICK_VAL << 16)  /* shift PICK_VAL to proper bits for FLAG */

XGL_GET_VERTEX_PICK(FLAG)     ((FLAG >> 16) & 0xffff) /* get pick value bits from FLAG */

XGL_SET_EDGE_FLAG(PT)         (PT)->flag |= XGL_DRAW_EDGE;
```

```
XGL_SET_PREV_EDGE_FLAG(PT)    (PT)->flag |= XGL_DRAW_PREV_EDGE;

XGL_SET_BOTH_EDGE_FLAGS(PT)   (PT)->flag |= XGL_DRAW_BOTH_EDGES;

XGL_SET_TRI_STAR_FLAG(PT)     (PT)->flag |= XGL_TRIANGLE_STAR;

XGL_SET_TRI_STRIP_FLAG(PT)    (PT)->flag |= XGL_TRIANGLE_STRIP;

XGL_CLR_EDGE_FLAG(PT)         (PT)->flag &= ~XGL_DRAW_EDGE;

XGL_CLR_PREV_EDGE_FLAG(PT)    (PT)->flag &= ~XGL_DRAW_PREV_EDGE;

XGL_CLR_BOTH_EDGE_FLAGS(PT)   (PT)->flag &= ~XGL_DRAW_BOTH_EDGES;

XGL_CLR_TRI_STAR_FLAG(PT)     (PT)->flag &= ~XGL_TRIANGLE_STAR;

XGL_CLR_TRI_STRIP_FLAG(PT)    (PT)->flag &= ~XGL_TRIANGLE_STRIP;

XGL_RESTART_TRIANGLE(PT)      (PT)->flag &= ~(XGL_TRIANGLE_STRIP | XGL_TRIANGLE_STAR);
```

Window Types

Choice available for **XGL_WIN_RAS_TYPE**(3)**.**

```
typedef         Xgl_usgn32      Xgl_window_type;
    XGL_WIN_X


/* Protocols are in the upper 24 bits */
XGL_WIN_PROTO_MASK

/* WIN_X protocols */
XGL_WIN_X_PROTO_DEFAULT
XGL_WIN_X_PROTO_XLIB
XGL_WIN_X_PROTO_PEX
XGL_WIN_X_PROTO_DGA
```

SEE ALSO

xgl_enum_types(3)
xgl_pt_list(3)
xgl_struct(3)

NAME

xgl_pt_list – structure used to pass vertex information to several XGL primitives

SYNOPSIS

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

typedef struct {
    Xgl_pt_type         pt_type;
    Xgl_bbox            *bbox;
    Xgl_usgn32          num_pts;
    Xgl_usgn32          num_data_values;
    Xgl_pt_ptr_union    pts;
} Xgl_pt_list;
```

DESCRIPTION

Xgl_pt_list(3) is the structure used to pass vertex information to the following XGL primitives:

xgl_multi_simple_polygon(3)
xgl_multimarker(3)
xgl_multipolyline(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)
xgl_triangle_strip(3)

Other XGL primitives (for example, **xgl_multiarc**(3)) use more specialized data structures to define their associated vertex information. Still other XGL primitives (for example, **xgl_nurbs_curve**(3)) restrict the possible point types to a subset of the types listed here.

Double point types (*d2d and d3d*) are not supported for any primitives except for the Transform primitives.

The individual members of *Xgl_pt_list* are defined as follows:

pt_type Specifies the point type of the vertex information stored in the array *pts*. One of 38 different point types can be used, 12 for 2D Contexts, 26 for 3D Contexts. The enumerated type *Xgl_pt_type* specifies the 38 types available (see table below for a listing of all available types). The point type names are formed by combining words and abbreviations that specify the information available within the vertex information.

Colors, normals, data, and flags can be included within the vertex information (the flags specify whether edges will be drawn). The vertex position information can be specified in one of three data types: Integer (I), Floating Point (F), or Fixed-Point Binary (B). It can be 2D or 3D, and it can be homogeneous (H). As an example, vertex information that contains floating point *x*-, *y*-, and *z*-coordinate information (3D), with colors and normals included for each vertex, have a type name of: XGL_PT_COLOR_NORMAL_F3D.

bbox Specifies a bounding box that surrounds all of the vertices within this point list. This entry is optional; it saves some checking for XGL. If the user doesn't specify this field, it should 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.

num_pts
The number of vertices stored in the array *pts*.

num_data_values
The number of data values at each point. This is ignored if *pt_type* does not have data.

pts A pointer to an array of vertex information. The type *Xgl_pt_ptr_union* is a union of pointers to different point types. It is a structure defined in **xgl_struct**(3) as follows:

```
typedef union {
    Xgl_pt_i2d                      *i2d;
    Xgl_pt_color_i2d                *color_i2d;
    Xgl_pt_flag_i2d                 *flag_i2d;
    Xgl_pt_i2h                      *i2h;

    Xgl_pt_f2d                      *f2d;
    Xgl_pt_color_f2d                *color_f2d;
    Xgl_pt_flag_f2d                 *flag_f2d;
    Xgl_pt_f2h                      *f2h;
    Xgl_pt_f3d                      *f3d;
    Xgl_pt_color_f3d                *color_f3d;
    Xgl_pt_normal_f3d               *normal_f3d;
    Xgl_pt_color_normal_f3d         *color_normal_f3d;
    Xgl_pt_flag_f3d                 *flag_f3d;
    Xgl_pt_color_flag_f3d           *color_flag_f3d;
    Xgl_pt_normal_flag_f3d          *normal_flag_f3d;
    Xgl_pt_color_normal_flag_f3d    *color_normal_flag_f3d;
    Xgl_pt_f3h                      *f3h;
    Xgl_pt_data_f3d                 *data_f3d;
    Xgl_pt_color_data_f3d           *color_data_f3d;
    Xgl_pt_normal_data_f3d          *normal_data_f3d;
    Xgl_pt_color_normal_data_f3d    *color_normal_data_f3d;
    Xgl_pt_flag_data_f3d            *flag_data_f3d;
    Xgl_pt_color_flag_data_f3d      *color_flag_data_f3d;
    Xgl_pt_normal_flag_data_f3d     *normal_flag_data_f3d;
```

```
        Xgl_pt_color_normal_flag_data_f3d   *color_normal_flag_data_f3d;


        Xgl_pt_d2d                          *d2d;
        Xgl_pt_color_d2d                    *color_d2d;
        Xgl_pt_flag_d2d                     *flag_d2d;
        Xgl_pt_d2h                          *d2h;
        Xgl_pt_d3d                          *d3d;
        Xgl_pt_color_d3d                    *color_d3d;
        Xgl_pt_normal_d3d                   *normal_d3d;
        Xgl_pt_color_normal_d3d             *color_normal_d3d;
        Xgl_pt_flag_d3d                     *flag_d3d;
        Xgl_pt_color_flag_d3d               *color_flag_d3d;
        Xgl_pt_normal_flag_d3d              *normal_flag_d3d;
        Xgl_pt_color_normal_flag_d3d        *color_normal_flag_d3d;
        Xgl_pt_d3h                          *d3h;
    } Xgl_pt_ptr_union;
```

The tables below show how to use the *pts* member of the XGL point list structure:

For a 2D Context, the valid point types are:

Field Name	Type	Point Type
pts.i2d	Xgl_pt_i2d *	XGL_PT_I2D
pts.color_i2d	Xgl_pt_color_i2d *	XGL_PT_COLOR_I2D
pts.flag_i2d	Xgl_pt_flag_i2d *	XGL_PT_FLAG_I2D
pts.i2h	Xgl_pt_i2h *	XGL_PT_I2H
pts.f2d	Xgl_pt_f2d *	XGL_PT_F2D
pts.color_f2d	Xgl_pt_color_f2d *	XGL_PT_COLOR_F2D
pts.flag_f2d	Xgl_pt_flag_f2d *	XGL_PT_FLAG_F2D
pts.f2h	Xgl_pt_f2h *	XGL_PT_F2H
pts.d2d	Xgl_pt_d2d *	XGL_PT_D2D
pts.color_d2d	Xgl_pt_color_d2d *	XGL_PT_COLOR_D2D
pts.flag_d2d	Xgl_pt_flag_d2d *	XGL_PT_FLAG_D2D
pts.d2h	Xgl_pt_d2h *	XGL_PT_D2H

For a 3D Context, the valid point types are:

Field Name	Type	Point Type
pts.f3d	Xgl_pt_f3d *	XGL_PT_F3D
pts.color_f3d	Xgl_pt_color_f3d *	XGL_PT_COLOR_F3D
pts.normal_f3d	Xgl_pt_normal_f3d *	XGL_PT_NORMAL_F3D
pts.color_normal_f3d	Xgl_pt_color_normal_f3d *	XGL_PT_COLOR_NORMAL_F3D
pts.flag_f3d	Xgl_pt_flag_f3d *	XGL_PT_FLAG_F3D
pts.color_flag_f3d	Xgl_pt_color_flag_f3d *	XGL_PT_COLOR_FLAG_F3D
pts.normal_flag_f3d	Xgl_pt_normal_flag_f3d *	XGL_PT_NORMAL_FLAG_F3D
pts.color_normal_flag_f3d	Xgl_pt_color_normal_flag_f3d *	XGL_PT_COLOR_NORMAL_FLAG_F3D
pts.f3h	Xgl_pt_f3h *	XGL_PT_F3H
pts.data_f3d	Xgl_pt_data_f3d *	XGL_PT_DATA_F3D
pts.color_data_f3d	Xgl_pt_color_data_f3d *	XGL_PT_COLOR_DATA_F3D
pts.normal_data_f3d	Xgl_pt_normal_data_f3d *	XGL_PT_NORMAL_DATA_F3D
pts.color_normal_data_f3d	Xgl_pt_color_normal_data_f3d *	XGL_PT_COLOR_NORMAL_DATA_F3D
pts.flag_data_f3d	Xgl_pt_flag_data_f3d *	XGL_PT_FLAG_DATA_F3D
pts.color_flag_data_f3d	Xgl_pt_color_flag_data_f3d *	XGL_PT_COLOR_FLAG_DATA_F3D
pts.normal_flag_data_f3d	Xgl_pt_normal_flag_data_f3d *	XGL_PT_NORMAL_FLAG_DATA_F3D
pts.color_normal_flag_data_f3d	Xgl_pt_color_normal_flag_data_f3d *	XGL_PT_COLOR_NORMAL_FLAG_DATA_F3D
pts.d3d	Xgl_pt_d3d *	XGL_PT_D3D
pts.color_d3d	Xgl_pt_color_d3d *	XGL_PT_COLOR_D3D
pts.normal_d3d	Xgl_pt_normal_d3d *	XGL_PT_NORMAL_D3D
pts.color_normal_d3d	Xgl_pt_color_normal_d3d *	XGL_PT_COLOR_NORMAL_D3D
pts.flag_d3d	Xgl_pt_flag_d3d *	XGL_PT_FLAG_D3D
pts.color_flag_d3d	Xgl_pt_color_flag_d3d *	XGL_PT_COLOR_FLAG_D3D
pts.normal_flag_d3d	Xgl_pt_normal_flag_d3d *	XGL_PT_NORMAL_FLAG_D3D
pts.color_normal_flag_d3d	Xgl_pt_color_normal_flag_d3d *	XGL_PT_COLOR_NORMAL_FLAG_D3D
pts.d3h	Xgl_pt_d3h *	XGL_PT_D3H

SEE ALSO

xgl_multi_simple_polygon(3)
xgl_multimarker(3)
xgl_multipolyline(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)
xgl_triangle_strip(3)
xgl_nurbs_curve(3)
xgl_struct(3)

NAME

xgl_struct – XGL Data Structures

DESCRIPTION

This section lists the different C structures XGL uses. The structures appear with explanations about the fields used in a given structure.

Arc Structures

In the point given for the center of the arc, the flag field determines whether edges are drawn and also provides storage for an identification number to label individual arcs when picking multiarcs. The start and stop angles are in radians. All 2D angles increase counterclockwise and start (are equal to 0) on the positive *x* axis.

2D Arcs

```
typedef struct {
    Xgl_pt_flag_i2d   center;       /* center point */
    Xgl_usgn32        radius;       /* radius */
    float             start_angle,  /* start and stop angles */
                      stop_angle;
} Xgl_arc_i2d;


typedef struct {
    Xgl_pt_flag_f2d   center;       /* center point */
    float             radius;       /* radius */
    float             start_angle,  /* start and stop angles */
                      stop_angle;
} Xgl_arc_f2d;


typedef struct {
    Xgl_pt_flag_d2d     center;         /* Center point */
    double              radius;         /* Radius */
    double              start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_arc_d2d;
```

3D Arcs

```
typedef struct {
    Xgl_pt_flag_f3d     center;         /* center point */
    Xgl_pt_f3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
    float               radius;         /* radius */
    float               start_angle,    /* start and stop angles */
                        stop_angle;
} Xgl_arc_f3d;
```

```
typedef struct {
    Xgl_pt_flag_f3d     center;         /* center point = ref point in MC */
    float               radius;         /* radius in VDC */
    float               start_angle,    /* start and stop angles */
                        stop_angle;
} Xgl_arc_af3d;


typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point */
    Xgl_pt_d3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
    double              radius;         /* Radius */
    double              start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_arc_d3d;


typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point = ref point in MC */
    double              radius;         /* Radius in VDC*/
    double              start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_arc_ad3d;                         /* Annotation 3D arc */
```

Arc List Structure

The bounding box applies to all of the arc centers.

```
typedef struct {
    Xgl_usgn32          num_arcs;       /* Number of arcs */
    Xgl_multiarc_type   type;           /* Data type of multiarc */
    Xgl_bbox            *bbox;          /* Optional box containing all
                                           arcs; NULL means no box */
    union {                             /* Union of pointers to arcs */
        Xgl_arc_i2d     *i2d;
        Xgl_arc_f2d     *f2d;
        Xgl_arc_d2d     *d2d;
        Xgl_arc_f3d     *f3d;
        Xgl_arc_af3d    *af3d;
        Xgl_arc_d3d     *d3d;
        Xgl_arc_ad3d    *ad3d;
    } arcs;
} Xgl_arc_list;
```

Bounding Box Structures

XGL bounding boxes are used to optimize geometric operations within the XGL pipeline. The application should provide the bounding box information whenever possible. If you don't supply a bbox to function argument, use NULL instead.

```
typedef struct {
    Xgl_bbox_type        bbox_type;    /* specifies the type of the bbox */
    union {
        Xgl_bounds_i2d   i2d;          /* type can be integer 2d, */
        Xgl_bounds_f2d   f2d;          /*             float   2d, */
        Xgl_bounds_f3d   f3d;          /*          or float   3d  */
    } box;
} Xgl_bbox;
```

```
typedef struct {
    Xgl_bbox_type       bbox_type;
    union {
        Xgl_bounds_i2d  i2d;
    } box;
} Xgl_bbox_i2d;


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


typedef struct {
    Xgl_bbox_type       bbox_type;
    union {
        Xgl_bounds_f3d  f3d;
    } box;
} Xgl_bbox_f3d;


typedef struct {
    Xgl_bbox_type       bbox_type;
    union {
        Xgl_geom_status status;
    } box;
} Xgl_bbox_status;


typedef struct {
    Xgl_bbox_type       bbox_type;
    union {
        Xgl_bounds_d2d  d2d;
    } box;
} Xgl_bbox_d2d;


typedef struct {
    Xgl_bbox_type       bbox_type;
    union {
        Xgl_bounds_d3d  d3d;
    } box;
} Xgl_bbox_d3d;
```

Bounds Structures

Bounds structures are used to define bounding boxes, windows, and viewports or to delimit a range of values between minimal and maximal limits.

```
typedef struct {
    float        bmin;
    float        bmax;
} Xgl_bounds_f1d;


typedef struct {
    double      bmin;
    double      bmax;
```

```
} Xgl_bounds_d1d;


typedef struct {
    float        xmin;
    float        xmax;
    float        ymin;
    float        ymax;
} Xgl_bounds_f2d;


typedef struct {
    double      xmin;
    double      xmax;
    double      ymin;
    double      ymax;
} Xgl_bounds_d2d;


typedef struct {
    float        xmin;
    float        xmax;
    float        ymin;
    float        ymax;
    float        zmin;
    float        zmax;
} Xgl_bounds_f3d;


typedef struct {
    double      xmin;
    double      xmax;
    double      ymin;
    double      ymax;
    double      zmin;
    double      zmax;
} Xgl_bounds_d3d;


typedef struct {
    Xgl_sgn32    xmin;
    Xgl_sgn32    xmax;
    Xgl_sgn32    ymin;
    Xgl_sgn32    ymax;
} Xgl_bounds_i2d;
```

Circle Structures

In the point given for the center of a circle, the flag field determines whether edges are drawn and also provides storage for an identification number to label individual circles when picking multicircles.

```
typedef struct {
    Xgl_pt_flag_i2d  center;          /* center point */
    Xgl_usgn32       radius;          /* radius */
} Xgl_circle_i2d;
```

```
typedef struct {
    Xgl_pt_flag_f2d  center;           /* center point */
    float            radius;           /* radius */
} Xgl_circle_f2d;


typedef struct {
    Xgl_pt_flag_d2d     center;         /* Center point */
    double              radius;         /* Radius */
} Xgl_circle_d2d;


typedef struct {
    Xgl_pt_flag_f3d  center;           /* center point */
    Xgl_pt_f3d       dir[3];           /* direction and normal vectors */
    Xgl_boolean      dir_normalized;   /* TRUE when dir[] are unit vectors */
    Xgl_boolean      dir_normal;       /* TRUE when dir[3] is a valid vector */
    float            radius;           /* radius */
} Xgl_circle_f3d;


typedef struct {
    Xgl_pt_flag_f3d     center;         /* center point = ref point in MC */
    float               radius;         /* radius in VDC */
} Xgl_circle_af3d;


typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point */
    Xgl_pt_d3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
    double              radius;         /* Radius */
} Xgl_circle_d3d;


typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point = ref point in MC */
    double              radius;         /* Radius in VDC */
} Xgl_circle_ad3d;                      /* Annotation 3D circle */
```

Circle List Structure

The bounding box applies to all of the circle centers.

```
typedef struct {
    Xgl_usgn32           num_circles;   /* Number of circles */
    Xgl_multicircle_type type;          /* Data type of multicircle */
    Xgl_bbox             *bbox;         /* Optional box containing all
                                           circles; NULL means no box */
    union {                             /* Union of pointers to circles */
        Xgl_circle_i2d  *i2d;
        Xgl_circle_f2d  *f2d;
        Xgl_circle_d2d  *d2d;
        Xgl_circle_f3d  *f3d;
        Xgl_circle_af3d *af3d;
        Xgl_circle_d3d  *d3d;
        Xgl_circle_ad3d *ad3d;
    } circles;
} Xgl_circle_list;
```

Color Structure

The color can be defined as an index, a gray level, an RGB, or a z quadruplet.

```
typedef union {
    Xgl_color_index index;
    Xgl_color_gray  gray;
    Xgl_color_rgb   rgb;
    Xgl_color_z     z;
} Xgl_color;
```

Color Index Structure

```
typedef     Xgl_usgn32     Xgl_color_index;
```

Color Gray Structure

```
typedef     float          Xgl_color_gray;
```

Color RGB Structure

In RGB, the color is defined by a triplet of floating point values between 0.0 and 1.0.

```
typedef struct {
    float       r, g, b;
} Xgl_color_rgb;
```

Color Z Structure

```
typedef         Xgl_usgn32      Xgl_color_z;
```

Homogeneous Color Structure

```
typedef struct {
    float r, g, b;
    float      w;
} Xgl_color_rgbw;
```

Homogeneous Color Type

```
typedef union {
    Xgl_color_rgbw rgbw;
} Xgl_color_homogeneous;
```

Color List Structure

This structure is used to list color entries. It can be used to set an XGL Color Map.

```
typedef struct {
    Xgl_usgn32   start_index;
    Xgl_usgn32   length;
    Xgl_color    *colors;
} Xgl_color_list;
```

Color Models Returned by Inquiry

```
typedef struct {
    unsigned    index:1;
    unsigned    rgb:1;
} Xgl_color_type_supported;
```

Color Spline for Color-Mapped NURBS Curves

```
typedef struct {
    Xgl_usgn32                  order;
    Xgl_usgn32                  num_knots;
    float                       *knot_vector;
    Xgl_color_homogeneous       *colors;
} Xgl_curve_color_spline;
```

Elliptical Arc Structures

The elliptical arc structure flag determines whether edges are drawn and also provides storage for an identification number to label individual ellipse when picking multiellipse.

```
typedef struct {
    Xgl_pt_flag_f3d     center;         /* Center point */
    Xgl_pt_f3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
    float               major_axis,     /* Major & Minor axis */
                        minor_axis;
    float               rot_angle;      /* Ellipse rotation angle */
    float               start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_ell_f3d;


typedef struct {
    Xgl_pt_flag_f3d     center;         /* Center point = ref point in MC */
    float               major_axis,     /* Major & Minor axis in VDC*/
                        minor_axis;
    float               rot_angle;      /* Ellipse rotation angle */
    float               start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_ell_af3d;                         /* Annotation 3D elliptical arc */


typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point */
    Xgl_pt_d3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
    double              major_axis,     /* Major & Minor axis */
                        minor_axis;
    double              rot_angle;      /* Ellipse rotation angle */
    double              start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_ell_d3d;
```

```
typedef struct {
    Xgl_pt_flag_d3d     center;         /* Center point = ref point in MC */
    double              major_axis,     /* Major & Minor axis in VDC*/
                        minor_axis;
    double              rot_angle;      /* Ellipse rotation angle */
    double              start_angle,    /* Start and stop angles */
                        stop_angle;
} Xgl_ell_ad3d;                         /* Annotation 3D elliptical arc */
```

Elliptical Arc List Structure

```
typedef struct {
    Xgl_usgn32          num_ells;       /* Number of elliptical arcs */
    Xgl_multiell_type   type;           /* Data type of multiell */
    Xgl_bbox            *bbox;          /* Optional box containing all
                                           elliptical arcs; NULL means no box */
    union {                             /* Union of pointers to ells */
        Xgl_ell_f3d     *f3d;
        Xgl_ell_af3d    *af3d;
        Xgl_ell_d3d     *d3d;
        Xgl_ell_ad3d    *ad3d;
    } ells;
} Xgl_ell_list;
```

Error Structure Containing Error Attributes

```
typedef struct {
    Xgl_error_type      type;
    Xgl_error_category  category;
    char*               id;
    char*               msg;
    char*               cur_op;
    char*               cur_obj;
    char*               operand1;
    char*               operand2;
} Xgl_error_info;
```

Facet with Color Structure

A facet contains information for a simple polygon, such as a single boundary polygon, or one triangle in a triangle strip, but it can also represent a complex set of boundaries for a planar polygon. The information contained in a facet can be color, normal vector, or both.

```
typedef struct {
    Xgl_color  color;
} Xgl_color_facet;
```

Facet with Normal Vector Structure

For details, see Facet with Color Structure.

```
typedef struct {
    Xgl_pt_f3d  normal;
} Xgl_normal_facet;
```

Facet with Color and Normal Vector Structure

For details, see Facet with Color Structure.

```
typedef struct {
    Xgl_color    color;
    Xgl_pt_f3d   normal;
} Xgl_color_normal_facet;
```

Facet Structure

Used to specify the color or surface normal vector, or both, for one facet.

```
typedef union {
    Xgl_color_facet        color_facet;        /* color facet */
    Xgl_normal_facet       normal_facet;       /* normal facet */
    Xgl_color_normal_facet color_normal_facet; /* color & normal facet */
} Xgl_facet;
```

List of Polygonal Facets

When providing a list of facet information, all the facets must be of the same type.

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

List of Facet Lists

```
typedef struct {
    Xgl_usgn32          num_facet_lists;        /* number of facet lists */
    Xgl_facet_list      *facet_lists;           /* array of facet lists */
} Xgl_facet_list_list;
```

HLHSR Structure

Currently, the only supported HLHSR mode uses a *z*-buffer. This structure is used to pass initialization information to the *z*-buffer. Values given for *z*-value are positive numbers.

```
typedef union {
    struct {
        float   z_value;
    } z_buffer;
} Xgl_hlhsr_data;
```

Inquire Data Structure

```
typedef struct {
    char                        *name;
    Xgl_boolean                 dga_flag;
    Xgl_color_type_supported    color_type;
```

```
    Xgl_sgn32                   depth, width, height;
    Xgl_sgn32                   maximum_buffer;
    Xgl_boolean                 db_buffer_is_copy;
    Xgl_pt_type_supported       pt_type;
    Xgl_hlhsr_mode              hlhsr_mode;
    Xgl_inq_support_level       picking;
    Xgl_inq_support_level       double_buffer;
    Xgl_inq_support_level       indexed_color;
    Xgl_inq_support_level       true_color;
    Xgl_inq_support_level       depth_cueing;
    Xgl_inq_support_level       lighting;
    Xgl_inq_support_level       shading;
    Xgl_inq_support_level       hlhsr;
    Xgl_inq_support_level       antialiasing;
    Xgl_boolean                 stereo;
} Xgl_inquire;
```

Window Rectangles and Window Rectangles List

These structures define the storage type of clip lists used by the window system.

```
typedef struct {
    Xgl_usgn32  xmin;
    Xgl_usgn32  xmax;
    Xgl_usgn32  ymin;
    Xgl_usgn32  ymax;
} Xgl_irect;


typedef struct {
    Xgl_usgn32  num_irects;
    Xgl_irect   *rects;
} Xgl_irect_list;
```

Strings and Text Structures

XGL accepts two different ways of describing text. Text can be a *NULL-terminated* C-style string of characters, or it can be coded as a list of mono-encoded character strings.

```
typedef struct {
    Xgl_object          font_obj;       /* Each mono-encoded string has its */
    char                *text;          /* font and C type string of characters */
} Xgl_mono_text;


typedef struct {
    Xgl_usgn32          mono_num;       /* number of mono-encoded strings */
    Xgl_mono_text       *mono_list;     /* array of mono-encoded strings */
} Xgl_mono_text_list;
```

NURBS Curve and Surface Structures

These structures are used with the XGL curve and surface operators to draw NURBS curves and surfaces.

/* Nonuniform B-spline curve (XGL 1.0 and 2.0 definition of Xgl_nurbs_curve) */

```
typedef struct {
    Xgl_usgn32          order;          /* Order of curve */
    Xgl_usgn32          num_knots;      /* Number of knots */
    float               *knot_vector;   /* Knot vector */
    Xgl_bounds_f1d      range;          /* Range of parameter */
    Xgl_pt_list         ctrl_pts;       /* List of control points */
    Xgl_boolean         trim_curve_vis; /* If this is a trim curve, is
                                           its outline edge drawn? */
} Xgl_nu_bspline_curve;
```

/* Nonuniform B-spline curve (definition for XGL 3.0 and subsequent releases) */

```
typedef struct {
    Xgl_usgn32          order;          /* Order of curve */
    Xgl_usgn32          num_knots;      /* Number of knots */
    float               *knot_vector;   /* Knot vector */
    Xgl_pt_list         ctrl_pts;       /* List of control points */
} Xgl_nurbs_curve;


typedef struct {
    Xgl_usgn32          order_u;        /* Order of surface in u */
    Xgl_usgn32          order_v;        /* Order of surface in v */
    Xgl_usgn32          num_knots_u;    /* Number of knots in u */
    float               *knot_vector_u; /* Knot vector in u */
    Xgl_usgn32          num_knots_v;    /* Number of knots in v */
    float               *knot_vector_v; /* Knot vector in v */
    Xgl_pt_list         ctrl_pts;       /* List of control points
                                           (index changes faster in u
                                           direction than v). */
} Xgl_nurbs_surf;
```

NURBS Surface Geometry Description

```
typedef union {
    Xgl_plane           planar;
    struct {
        Xgl_pt_f3d      axial_pt;
        Xgl_pt_f3d      axis_dir;
        float           radius;
        Xgl_boolean     norm_flag;
    } cylindrical;
    struct {
        Xgl_pt_f3d      apex;
        Xgl_pt_f3d      axis_dir;
        float           cone_angle; /* 0 = zero width cone (line)*/
        Xgl_boolean     norm_flag;
    } conical;
    struct {
        Xgl_pt_f3d      center;
```

```
        float           radius;
        Xgl_boolean     norm_flag;
    } spherical;
} Xgl_nurbs_surf_geom_desc;
```

NURBS Surface Simple Geometry Structure

```
typedef struct {
        Xgl_nurbs_surf_type             surf_type;
        Xgl_nurbs_surf_geom_desc        geom_desc;
} Xgl_nurbs_surf_simple_geom;
```

Object Descriptor

```
typedef union {
    struct {
        Xgl_window_type type; /* window type */
        void            *desc;
    } win_ras;
    char *sfont_name;
    struct {
        Xgl_object raster;
    } accum_buf;
    struct {
        char *name;     /* library name (eg. xglSUNWcgm) */
        void *desc;     /* opaque handle for stream-specific info. */
    } stream;
} Xgl_obj_desc;
```

Pick Info Structure

When picking graphical entities, the pick IDs are stored into pick info structures.

```
typedef struct {
    Xgl_usgn32   id1;
    Xgl_usgn32   id2;
    Xgl_usgn32   vertex_flag;
} Xgl_pick_info;
```

Plane Structure

```
typedef struct {
    Xgl_pt_d3d  pt;             /* a point on the plane in WC */
    Xgl_pt_d3d  normal;         /* normal vector components */
} Xgl_plane;
```

Plane List Structure

```
typedef struct {
    Xgl_usgn32  num_planes;     /* number of planes */
    Xgl_plane   *planes;        /* pointer to array of planes */
} Xgl_plane_list;
```

Point Structures

XGL defines numerous data types an application can use to specify geometry. Most of these structures define data for a single vertex. A single vertex must at least have the point coordinates in model coordinates. It can also contain color, normal, data, or flag information. The coordinates can be float, double-precision float, or integer (2D). *Color* is defined with an *Xgl_color* structure; the *normal* vector is composed of three floating-point values representing a *normalized vector*; and, in the current implementation, the two lower bits of the *flag* field are used to specify whether an edge should be drawn. The 16 upper bits of the flag field can be used by the application to set an identifier on a per-vertex basis. XGL then returns the identifier vertex information to the application when picking is performed.

A single point structure with only position information.

```
typedef struct {
    Xgl_pt_type     pt_type;     /* type of point */
    Xgl_pt_position pt;          /* union of pointers to one point */
} Xgl_pt;
```

Structures for Floating Point Coordinates

```
typedef struct {
    float      x;
    float      y;
} Xgl_pt_f2d;


typedef struct {
    float      x;
    float      y;
    Xgl_color  color;
} Xgl_pt_color_f2d;


typedef struct {
    float      x;
    float      y;
    Xgl_usgn32 flag;
} Xgl_pt_flag_f2d;


typedef struct {
    float      x;
    float      y;
    float      w;
} Xgl_pt_f2h;


typedef struct {
    float      x;
    float      y;
    float      z;
} Xgl_pt_f3d;
```

```
typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_color  color;
} Xgl_pt_color_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_pt_f3d normal;
} Xgl_pt_normal_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_color  color;
    Xgl_pt_f3d normal;
} Xgl_pt_color_normal_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_usgn32 flag;
} Xgl_pt_flag_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_color  color;
    Xgl_usgn32 flag;
} Xgl_pt_color_flag_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_pt_f3d normal;
    Xgl_usgn32 flag;
} Xgl_pt_normal_flag_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    Xgl_color  color;
    Xgl_pt_f3d normal;
    Xgl_usgn32 flag;
} Xgl_pt_color_normal_flag_f3d;
```

```
typedef struct {
   float       x;
   float       y;
   float       z;
   float       data[1];
} Xgl_pt_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_color   color;
    float       data[1];
} Xgl_pt_color_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_pt_f3d  normal;
    float       data[1];
} Xgl_pt_normal_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_color   color;
    Xgl_pt_f3d  normal;
    float       data[1];
} Xgl_pt_color_normal_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_usgn32  flag;
    float       data[1];
} Xgl_pt_flag_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_color   color;
    Xgl_usgn32  flag;
    float       data[1];
} Xgl_pt_color_flag_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_pt_f3d  normal;
    Xgl_usgn32  flag;
    float       data[1];
```

```
} Xgl_pt_normal_flag_data_f3d;


typedef struct {
    float       x;
    float       y;
    float       z;
    Xgl_color   color;
    Xgl_pt_f3d  normal;
    Xgl_usgn32  flag;
    float       data[1];
} Xgl_pt_color_normal_flag_data_f3d;


typedef struct {
    float      x;
    float      y;
    float      z;
    float      w;
} Xgl_pt_f3h;
```

Structures for Double-Precision Floating Point Coordinates

```
typedef struct {
    double     x;
    double     y;
} Xgl_pt_d2d;


typedef struct {
    double      x;
    double      y;
    Xgl_color   color;
} Xgl_pt_color_d2d;


typedef struct {
    double      x;
    double      y;
    Xgl_usgn32  flag;
} Xgl_pt_flag_d2d;


typedef struct {
    double      x;
    double      y;
    double      w;
} Xgl_pt_d2h;


typedef struct {
    double     x;
    double     y;
    double     z;
} Xgl_pt_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
```

```
    Xgl_color  color;
} Xgl_pt_color_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_pt_f3d normal;
} Xgl_pt_normal_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_color  color;
    Xgl_pt_f3d normal;
} Xgl_pt_color_normal_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_usgn32 flag;
} Xgl_pt_flag_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_color  color;
    Xgl_usgn32 flag;
} Xgl_pt_color_flag_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_pt_f3d normal;
    Xgl_usgn32 flag;
} Xgl_pt_normal_flag_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    Xgl_color  color;
    Xgl_pt_f3d normal;
    Xgl_usgn32 flag;
} Xgl_pt_color_normal_flag_d3d;


typedef struct {
    double     x;
    double     y;
    double     z;
    double     w;
```

```
} Xgl_pt_d3h;
```

Structures for Integer Point Coordinates

```
typedef struct {
    Xgl_sgn32  x;
    Xgl_sgn32  y;
} Xgl_pt_i2d;


typedef struct {
    Xgl_sgn32  x;
    Xgl_sgn32  y;
    Xgl_color  color;
} Xgl_pt_color_i2d;


typedef struct {
    Xgl_sgn32  x;
    Xgl_sgn32  y;
    Xgl_usgn32 flag;
} Xgl_pt_flag_i2d;


typedef struct {
    Xgl_sgn32  x;
    Xgl_sgn32  y;
    Xgl_sgn32  w;
} Xgl_pt_i2h;
```

Point List and Point List List Structures

Each list of points describes a single set of connected lines, or a single boundary for polygons. For each list, the application can specify a bounding box in order to optimize geometrical computations. The point type should be identical for all the point lists passed to a single primitive at the same time.

```
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;


typedef struct {
    Xgl_bbox    *bbox;
    Xgl_usgn32  num_pt_lists;
    Xgl_pt_list *pt_lists;
} Xgl_pt_list_list;
```

Union of Pointers to Point Structures

These structures are used as pointers to any XGL point type.

```
typedef union {
    Xgl_pt_i2d                          *i2d;
    Xgl_pt_color_i2d                    *color_i2d;
    Xgl_pt_flag_i2d                     *flag_i2d;
    Xgl_pt_i2h                          *i2h;

    Xgl_pt_f2d                          *f2d;
    Xgl_pt_color_f2d                    *color_f2d;
    Xgl_pt_flag_f2d                     *flag_f2d;
    Xgl_pt_f2h                          *f2h;
    Xgl_pt_f3d                          *f3d;
    Xgl_pt_color_f3d                    *color_f3d;
    Xgl_pt_normal_f3d                   *normal_f3d;
    Xgl_pt_color_normal_f3d             *color_normal_f3d;
    Xgl_pt_flag_f3d                     *flag_f3d;
    Xgl_pt_color_flag_f3d               *color_flag_f3d;
    Xgl_pt_normal_flag_f3d              *normal_flag_f3d;
    Xgl_pt_color_normal_flag_f3d        *color_normal_flag_f3d;
    Xgl_pt_f3h                          *f3h;

    Xgl_pt_data_f3d                     *data_f3d;
    Xgl_pt_color_data_f3d               *color_data_f3d;
    Xgl_pt_normal_data_f3d              *normal_data_f3d;
    Xgl_pt_color_normal_data_f3d        *color_normal_data_f3d;
    Xgl_pt_flag_data_f3d                *flag_data_f3d;
    Xgl_pt_color_flag_data_f3d          *color_flag_data_f3d;
    Xgl_pt_normal_flag_data_f3d         *normal_flag_data_f3d;
    Xgl_pt_color_normal_flag_data_f3d *color_normal_flag_data_f3d;
    Xgl_pt_d2d                          *d2d;
    Xgl_pt_color_d2d                    *color_d2d;
    Xgl_pt_flag_d2d                     *flag_d2d;
    Xgl_pt_d2h                          *d2h;

    Xgl_pt_d3d                          *d3d;
    Xgl_pt_color_d3d                    *color_d3d;
    Xgl_pt_normal_d3d                   *normal_d3d;
    Xgl_pt_color_normal_d3d             *color_normal_d3d;
    Xgl_pt_flag_d3d                     *flag_d3d;
    Xgl_pt_color_flag_d3d               *color_flag_d3d;
    Xgl_pt_normal_flag_d3d              *normal_flag_d3d;
    Xgl_pt_color_normal_flag_d3d        *color_normal_flag_d3d;
    Xgl_pt_d3h                          *d3h;
} Xgl_pt_ptr_union;
```

Union of Pointers to Points with Only Position Information

```
typedef union {
    Xgl_pt_i2d  *i2d;
    Xgl_pt_f2d  *f2d;
    Xgl_pt_d2d  *d2d;
    Xgl_pt_f3d  *f3d;
    Xgl_pt_d3d  *d3d;
} Xgl_pt_position;
```

Point Types Returned by Inquiry

```
typedef struct {
    unsigned        pt_dim_2d:1;
    unsigned        pt_dim_3d:1;
    unsigned        pt_type_int:1;
    unsigned        pt_type_float:1;
    unsigned        pt_type_double:1;
} Xgl_pt_type_supported;
```

Rectangle Structures

In the point given for the first point of the rectangle, the flag field determines whether edges are drawn and also provides storage for an identification number to label individual rectangles when picking multirectangles.

```
typedef struct {
    Xgl_pt_flag_i2d corner_min;
    Xgl_pt_i2d      corner_max;
} Xgl_rect_i2d;


typedef struct {
    Xgl_pt_flag_f2d corner_min;
    Xgl_pt_f2d      corner_max;
} Xgl_rect_f2d;


typedef struct {
    Xgl_pt_flag_d2d     corner_min;
    Xgl_pt_d2d          corner_max;
} Xgl_rect_d2d;


typedef struct {
    Xgl_pt_flag_f2d corner_min;         /* 1st corner in the plane */
    Xgl_pt_f2d      corner_max;         /* 2nd corner in the plane */
    Xgl_pt_f3d      dir[3];             /* dir. vectors and normal vector */
    Xgl_boolean     dir_normalized;     /* TRUE when dir[] are unit vectors */
    Xgl_boolean     dir_normal;         /* TRUE if dir[2] is a valid vector */
} Xgl_rect_f3d;


typedef struct {
    Xgl_pt_flag_f3d     corner_min;     /* 1st corner = ref point in MC */
    Xgl_pt_f2d          corner_max;     /* deltas x&y in VDC, rel. to 1st point */
} Xgl_rect_af3d;


typedef struct {
    Xgl_pt_flag_d3d     corner_min;     /* reference corner of the rectangle */
    Xgl_pt_d2d          corner_max;     /* width & height of the rectangle, in MC */
    Xgl_pt_d3d          dir[3];         /* dir. vectors and normal vector */
    Xgl_boolean         dir_normalized; /* TRUE when dir[] are unit vectors */
    Xgl_boolean         dir_normal;     /* TRUE if dir[2] is a valid vector */
} Xgl_rect_d3d;


typedef struct {
    Xgl_pt_flag_d3d     corner_min;     /* 1st corner = ref point in MC */
    Xgl_pt_d2d          corner_max;     /* deltas x&y in VDC, rel. to 1st point */
} Xgl_rect_ad3d;                        /* Annotation 3D rectangle */
```

Rectangle List Structure

The bounding box applies to all the rectangles.

```
typedef struct {
    Xgl_usgn32          num_rects;      /* Number of rectangles */
    Xgl_multirect_type  rect_type;      /* Data type of multirectangle */
    Xgl_bbox            *bbox;          /* Optional box containing all
                                           rectangles; NULL means no box */
    union {                             /* Union of pointers to rectangles */
        Xgl_rect_i2d    *i2d;
        Xgl_rect_f2d    *f2d;
        Xgl_rect_d2d    *d2d;
        Xgl_rect_f3d    *f3d;
        Xgl_rect_af3d   *af3d;
        Xgl_rect_d3d    *d3d;
        Xgl_rect_ad3d   *ad3d;
    } rects;
} Xgl_rect_list;
```

Color Map Ramp Structure

An XGL color ramp is defined by an offset, where the ramp starts in the color_map, and by the length of the ramp.

```
typedef struct {
    Xgl_usgn32  offset;
    Xgl_usgn32  length;
} Xgl_segment;
```

Color Spline for Color-Mapped NURBS Surfaces

```
typedef struct {
    Xgl_usgn32                  order_u;
    Xgl_usgn32                  order_v;
    Xgl_usgn32                  num_knots_u;
    float                       *knot_vector_u;
    Xgl_usgn32                  num_knots_v;
    float                       *knot_vector_v;
    Xgl_color_homogeneous       *colors;
} Xgl_surf_color_spline;
```

Texture Operation Structures and Unions

```
typedef struct {
    Xgl_color_rgb       c1;
    Xgl_color_rgb       c2;
} Xgl_texture_blend_rgb;


typedef union {
    Xgl_texture_blend_rgb       rgb;
} Xgl_texture_blend_op;


typedef struct {
    Xgl_color_rgb   c;
} Xgl_texture_decal_rgb;
```

```
typedef union {
    Xgl_texture_decal_rgb rgb;
} Xgl_texture_decal_op;


typedef struct {
    Xgl_color_rgb       c;
} Xgl_texture_blend_intrinsic_rgb;


typedef union {
    Xgl_texture_blend_intrinsic_rgb       rgb;
} Xgl_texture_blend_intrinsic_op;


typedef struct {
    Xgl_render_component        comp;
    Xgl_texture_op              texture_op;
    union Xgl_op_info {
        Xgl_texture_blend_op    blend;
        Xgl_texture_decal_op    decal;
        Xgl_texture_blend_intrinsic_op       blend_int;
    } op;
} Xgl_render_component_desc;
```

Description for Texture Map Types

```
typedef struct {
    Xgl_texture_type            texture_type;   /* type of texture which is used
                                                 * to access the fields of the
                                                 * "info" union.
                                                 */
    Xgl_texture_color_comp_info comp_info;
    Xgl_texture_interp_info     interp_info;     /* filter to use */
    union {
       Xgl_texture_mipmap_desc  mipmap;
    } info;
} Xgl_texture_desc;


typedef struct {
    Xgl_texture_color_comp_info       color_info;
} Xgl_texture_comp_info;


typedef struct {
    Xgl_texture_type            texture_type;   /* type of texture which is used
                                                 * to access the fields of the
                                                 * "info" union.
                                                 */
    Xgl_texture_comp_info comp_info;
    union {
       Xgl_texture_general_mipmap_desc  mipmap;
    } texture_info;
} Xgl_texture_general_desc;
```

Texture Map Interpolation Information

```
typedef struct {
```

```
    Xgl_texture_interp_method       filter1;
    Xgl_texture_interp_method       filter2;
} Xgl_texture_interp_info;


typedef struct {
    Xgl_usgn32                  num_render_comp_desc; /* number of elements
                                                         in the following
                                                         array that are used
                                                       */
    Xgl_render_component_desc   render_component_desc[4];
    Xgl_usgn32                num_channels[2]; /* number of channels used to
                                                  /* do color composition */
    Xgl_usgn32                channel_number[2];/* channel number used if
                                                   /* num_channel is 1 */
} Xgl_texture_color_comp_info;
```

Texture MipMap Description

```
typedef struct {
    Xgl_texture                 texture_map;
    float                       max_u_freq;     /* range 1.0 to 0; use 1.0 */
    float                       max_v_freq;     /* as default */
    Xgl_texture_boundary        u_boundary;     /*clamp, wrap, or*/
    Xgl_texture_boundary        v_boundary;     /*  transparent  */
    Xgl_usgn8                   boundary_values[4];   /* values/colors to use
                                                         * if (U,V) exceeds
                                                         * the texture.
                                                         */
    float                       depth_interp_factor;  /* adjustment */
} Xgl_texture_mipmap_desc;


typedef struct {
    Xgl_texture                 texture_map;
    float                       max_u_freq;     /* range 1.0 to 0; use 1.0 */
    float                       max_v_freq;     /* as default */
    Xgl_texture_boundary        u_boundary;     /*clamp, wrap, or*/
    Xgl_texture_boundary        v_boundary;     /*  transparent  */
    Xgl_usgn8                   boundary_values[4];   /* values/colors to use
                                                         * if (U,V) exceeds
                                                         * the texture.
                                                         */
    float                       depth_interp_factor;  /* adjustment */
    Xgl_texture_interp_info   interp_info;         /* filter to use */
    Xgl_matrix_f2d            orientation_matrix;   /* orientation matrix */
} Xgl_texture_general_mipmap_desc;
```

Threshold Structure

Individual values can be given for primitives rendered with vectors (polylines, curves, stroke text), circles, and polygons.

```
typedef struct {
    Xgl_usgn32  vectors;
    Xgl_usgn32  circles;
    Xgl_usgn32  polygons;
} Xgl_threshold;
```

NURBS Trim Curve and Trim Loop List Structures

The trim curve and loop structures define the trimmed portion of the displayed NURBS surface.

```
typedef struct {
    Xgl_usgn32            order;                  /* Order of curve */
    Xgl_usgn32            num_knots;              /* Number of knots */
    float                 *knot_vector;           /* Knot vector */
    Xgl_bounds_f1d        range;                  /* Range of parameter */
    Xgl_pt_list           ctrl_pts;               /* List of control points */
    /*
     * Trim curve structure fields (applicable only to trim curves)
     */
    Xgl_boolean           trim_curve_vis;         /* Highlight trim curve */
    Xgl_trim_curve_approx trim_curve_approx;      /* Approx type */
    float                 trim_curve_approx_value; /* Approx value */
} Xgl_trim_curve;


typedef struct {
    Xgl_usgn32          num_curves;
    Xgl_trim_curve      *curves;
} Xgl_trim_loop;


typedef struct {
    Xgl_usgn32          num_loops;
    Xgl_trim_loop       *trim_loops;
} Xgl_trim_loop_list;
```

X11 Window Descriptor Structure

X11 data structure for create raster call with XGL_WIN_X. Please refer to example programs on how to fill the fields of this structure.

```
typedef struct {
    void        *X_display;       /* pointer to X display */
    int          X_screen;        /* X screen number */
    Xgl_usgn32   X_window;        /* X window or drawable from server */
} Xgl_X_window;
```

SEE ALSO

xgl_enum_types(3)
xgl_macro_values(3)
xgl_pt_list(3)

NAME

XGL_2D_CTX
XGL_3D_CTX – a 2D or 3D Context object

OVERVIEW

XGL_2D_CTX and XGL_3D_CTX are the XGL *Context* objects for 2D and 3D. The *type* of the 2D Context is XGL_2D_CTX. The *type* of the 3D Context is XGL_3D_CTX.

The XGL *Context* is an abstraction of a *graphics renderer* that draws onto an XGL Device object (which is an abstraction of a graphics device). Most of the operators of the Context class are the *primitives* that are rendered, such as lines, polygons, and text; the other operators control the Device. The attributes of the Context control the visual appearance of the primitives that are drawn.

At creation time, a Context is not associated with any Device. The application must explicitly do this by setting the **XGL_CTX_DEVICE**(3) attribute in the Context.

After creating the Context, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

When a Context is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

A System State object must have been created with the **xgl_open**(3) operator before any Context can be created.

Context uses the following attributes:

XGL_3D_CTX_ACCUM_OP_DEST(3)
XGL_3D_CTX_BLEND_DRAW_MODE(3)
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER(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_HLHSR_DATA(3)
XGL_3D_CTX_HLHSR_MODE(3)

XGL_3D_CTX_JITTER_OFFSET(3)

XGL_3D_CTX_LIGHTS(3)
XGL_3D_CTX_LIGHT_NUM(3)
XGL_3D_CTX_LIGHT_SWITCHES(3)

XGL_3D_CTX_LINE_COLOR_INTERP(3)

XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)
XGL_3D_CTX_MODEL_CLIP_PLANES(3)

XGL_3D_CTX_NORMAL_TRANS(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_TMAP(3)
XGL_3D_CTX_SURF_BACK_TMAP_NUM(3)
XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES(3)
XGL_3D_CTX_SURF_BACK_TRANSP(3)
XGL_3D_CTX_SURF_DC_OFFSET(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_TMAP(3)
XGL_3D_CTX_SURF_FRONT_TMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_TRANSP(3)
XGL_3D_CTX_SURF_GEOM_NORMAL(3)
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP(3)
XGL_3D_CTX_SURF_NORMAL_FLIP(3)
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3)
XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION(3)
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ(3)
XGL_3D_CTX_SURF_TRANSP_METHOD(3)

XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY(3)

XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)

XGL_CTX_ARC_FILL_STYLE(3)

XGL_CTX_ATEXT_CHAR_HEIGHT(3)
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE(3)
XGL_CTX_ATEXT_CHAR_UP_VECTOR(3)
XGL_CTX_ATEXT_STYLE(3)
XGL_CTX_ATEXT_ALIGN_HORIZ(3)
XGL_CTX_ATEXT_ALIGN_VERT(3)
XGL_CTX_ATEXT_PATH(3)

XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_CLIP_PLANES(3)
XGL_CTX_DC_VIEWPORT(3)
XGL_CTX_DEFERRAL_MODE(3)
XGL_CTX_DEVICE(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_GEOM_DATA_IS_VOLATILE(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)

XGL_CTX_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

XGL_CTX_LOCAL_MODEL_TRANS(3)

XGL_CTX_MARKER(3)
XGL_CTX_MARKER_AA_BLEND_EQ(3)
XGL_CTX_MARKER_AA_FILTER_SHAPE(3)
XGL_CTX_MARKER_AA_FILTER_WIDTH(3)

XGL_CTX_MARKER_COLOR(3)
XGL_CTX_MARKER_COLOR_SELECTOR(3)
XGL_CTX_MARKER_SCALE_FACTOR(3)

XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MC_TO_DC_TRANS(3)
XGL_CTX_MIN_TESSELLATION(3)

XGL_CTX_MODEL_TRANS(3)
XGL_CTX_MODEL_TRANS_STACK_SIZE(3)

XGL_CTX_NEW_FRAME_ACTION(3)

XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)

XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)
XGL_CTX_PICK_SURF_STYLE(3)

XGL_CTX_PLANE_MASK(3)

XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RASTER_FPAT(3)
XGL_CTX_RASTER_FPAT_POSITION(3)
XGL_CTX_RASTER_STIPPLE_COLOR(3)

XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_RENDERING_ORDER(3)
XGL_CTX_ROP(3)

XGL_CTX_SFONT_0(3)
XGL_CTX_SFONT_1(3)
XGL_CTX_SFONT_2(3)
XGL_CTX_SFONT_3(3)

XGL_CTX_STEXT_AA_BLEND_EQ(3)
XGL_CTX_STEXT_AA_FILTER_SHAPE(3)
XGL_CTX_STEXT_AA_FILTER_WIDTH(3)
XGL_CTX_STEXT_ALIGN_HORIZ(3)
XGL_CTX_STEXT_ALIGN_VERT(3)

XGL_CTX_STEXT_CHAR_ENCODING(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)
XGL_CTX_STEXT_CHAR_HEIGHT(3)
XGL_CTX_STEXT_CHAR_SLANT_ANGLE(3)
XGL_CTX_STEXT_CHAR_SPACING(3)
XGL_CTX_STEXT_CHAR_UP_VECTOR(3)
XGL_CTX_STEXT_COLOR(3)
XGL_CTX_STEXT_PATH(3)
XGL_CTX_STEXT_PRECISION(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)

XGL_CTX_VDC_MAP(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VDC_WINDOW(3)

XGL_CTX_VIEW_CLIP_BOUNDS(3)
XGL_CTX_VIEW_TRANS(3)

XGL_OBJ_APPLICATION_DATA(3)
XGL_OBJ_SYS_STATE(3)
XGL_OBJ_TYPE(3)

NAME

XGL_3D_CTX_ACCUM_OP_DEST – specifies the buffer to be used as destination during the accumulation operation

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_ACCUM_OP_DEST, Xgl_buffer_sel, 0);
xgl_object_get(ctx, XGL_3D_CTX_ACCUM_OP_DEST, Xgl_buffer_sel*);
```

DESCRIPTION

This attribute specifies the destination buffer during the accumulation operations **xgl_context_accumulate**(3) and **xgl_context_clear_accumulation**(3)**.** The data type *Xgl_buffer_sel* takes on values:

XGL_BUFFER_SEL_NONE
: No accumulation is done.

XGL_BUFFER_SEL_ACCUM
: The accumulation buffer is used as the destination buffer during **xgl_context_accumulate**(3) and **xgl_context_clear_accumulation**(3)**.** The accumulation buffer is created for the first time when this option is selected, and when **xgl_context_accumulate**(3) or **xgl_context_clear_accumulation**(3) is called.

XGL_BUFFER_SEL_DISPLAY
: The current display buffer is used as the destination buffer during **xgl_context_accumulate**(3) and **xgl_context_clear_accumulation**(3)**.**

The default value is XGL_BUFFER_SEL_NONE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_accumulate(3)
xgl_context_clear_accumulation(3)

NAME

XGL_3D_CTX_BLEND_DRAW_MODE – controls if blended primitives and non-blended are drawn

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_BLEND_DRAW_MODE, Xgl_blend_draw_mode, 0);
xgl_object_get(ctx, XGL_3D_CTX_BLEND_DRAW_MODE, Xgl_blend_draw_mode *);
```

DESCRIPTION

This attribute controls the drawing of blended and non-blended primitives. A blended primitive is one that uses a blending equation such as a transparent or antialiased primitive. Since the final color of blended primitives depends upon what is already on the Device, a typical approach is to draw the non-blended primitives first and then the blended ones. It is not always possible for the application to determine if a primitive will be blended when it is rendered (due to face distinguishing).

The attribute enables the application to send the same scene to XGL twice in a row. The first time XGL can only render non-blended primitives and the second time it renders blended primitives.

The data type *Xgl_blend_draw_mode* takes on values:

XGL_BLEND_DRAW_ALL
: All primitives are drawn.

XGL_BLEND_DRAW_NOT_BLENDED
: Draw only those primitives which are not blended but do not draw edges of surface primitives.

XGL_BLEND_DRAW_BLENDED
: Draw only blended surfaces and surface primitive edges.

The default value is XGL_BLEND_DRAW_ALL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(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_LINE_AA_BLEND_EQ(3)
XGL_CTX_MARKER_AA_BLEND_EQ(3)
XGL_CTX_STEXT_AA_BLEND_EQ(3)
XGL_CTX_SURF_AA_BLEND_EQ(3)

NAME

XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER – controls if blended primitives update the Z Buffer

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER, Xgl_boolean *);
```

DESCRIPTION

This attribute controls all blended primitives (antialiased strokes and transparent surfaces). If this attribute is TRUE, then blended objects don't update the Z buffer; the Z comparison is still made and the blended pixel is only written if the comparison is true. If this attribute is FALSE, then the Z buffer is updated.

The default value is TRUE.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(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_LINE_AA_BLEND_EQ(3)
XGL_CTX_MARKER_AA_BLEND_EQ(3)
XGL_CTX_STEXT_AA_BLEND_EQ(3)
XGL_CTX_SURF_AA_BLEND_EQ(3)

NAME

XGL_3D_CTX_DEPTH_CUE_COLOR – specifies the color that depth-cued primitives tend toward for increasing values of device coordinate *z*

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_DEPTH_CUE_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_3D_CTX_DEPTH_CUE_COLOR, Xgl_color *);
```

DESCRIPTION

Depth-cueing modulates the color of 3D primitives based upon their depth (or distance away from the viewer in VDC space). The attribute **XGL_3D_CTX_DEPTH_CUE_MODE**(3) controls whether depth-cueing is enabled.

XGL_3D_CTX_DEPTH_CUE_COLOR is applied only when the the application is using the RGB color model (that is, when **XGL_DEV_COLOR_TYPE**(3) is set to XGL_COLOR_RGB) and when **XGL_3D_CTX_DEPTH_CUE_MODE**(3) is set to XGL_DEPTH_CUE_LINEAR or XGL_DEPTH_CUE_-SCALED.

XGL_3D_CTX_DEPTH_CUE_COLOR specifies the color that vertices will be modulated to with increasing depth. In other words, vertices that have a greater *z*-value will have a color close to the XGL_3D_CTX_DEPTH_CUE_COLOR. For a more detailed description of depth-cueing in XGL, see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.**

The default value is: RGB (0,0,0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_3D_CTX_DEPTH_CUE_INTERP – defines the depth-cue interpolation mode in a Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_DEPTH_CUE_INTERP, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_DEPTH_CUE_INTERP, Xgl_boolean *);
```

DESCRIPTION

Depth-cueing attenuates the color of 3D primitives based upon their depth (distance away from the viewer), as defined by the attribute **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.** This attribute controls whether depth-cueing interpolation is calculated along the depth axis, for the current Context *ctx*. Possible values for the attribute are:

TRUE A linear interpolation is performed all along the primitive for each pixel rendered, using the color defined at each vertex of the primitive by the attribute **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.**

FALSE The attribute **XGL_3D_CTX_DEPTH_CUE_MODE**(3) is used to calculate the depth-cue color at each vertex; however, no interpolation is done at rendering time. Individual lines, in a polyline, are drawn using the color defined at the end vertex of the line, and surfaces are rendered using the color computed for the first vertex of the surface.

This attribute is assumed to be TRUE if either of the following conditions is true:

1) You are rendering polylines and **XGL_3D_CTX_LINE_COLOR_INTERP**(3) is TRUE.
2) You are rendering surfaces and the attribute controlling illumination on the surface, either **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3) or **XGL_3D_CTX_SURF_FRONT_-ILLUMINATION**(3)**,** is set to XGL_ILLUM_NONE_INTERP_COLOR or XGL_ILLUM_PER_VERTEX.

In other words, if per-pixel interpolated shading has already been requested by the setting of other Context attributes, then XGL_3D_CTX_DEPTH_CUE_INTERP has no effect on the rendering.

The default value is TRUE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_LINE_COLOR_INTERP(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)

NOTES

Some devices may not implement this functionality, because they are faster in doing interpolation at rendering time, instead of having to select a specific color. Interpolated depth-cue always gives a better visual effect.

NAME

XGL_3D_CTX_DEPTH_CUE_MODE – defines the depth-cue mode in a Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx                 ctx;
xgl_object_set(ctx, XGL_3D_CTX_DEPTH_CUE_MODE, Xgl_depth_cue_mode, 0);
xgl_object_get(ctx, XGL_3D_CTX_DEPTH_CUE_MODE, Xgl_depth_cue_mode *);
```

DESCRIPTION

Depth-cueing attenuates the color of 3D primitives based upon their depth (*z-value* in VDC space). This attribute controls how depth-cueing is calculated in the current Context *ctx*. The attribute is device-dependent (see below under the value XGL_DEPTH_CUE_LINEAR). Possible values for the attribute are shown below. The following notation is used in the equations below:

Notation:

C_i = input color value

$\mathbf{C}'_i$ = depth-cued output color value

$\mathbf{C}_d$ = depth-cue color [See comments following equations.]

$\mathbf{Z}$ = Z coordinate of vertex

$\mathbf{Z}_f$ = Z coordinate of front depth-cue reference plane [from XGL_3D_CTX_DEPTH_CUE_REF_PLANES]

$\mathbf{Z}_b$ = Z coordinate of back depth-cue reference plane [from XGL_3D_CTX_DEPTH_CUE_REF_PLANES]

$\mathbf{S}_f$ = front depth-cue scale factor [from XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS]

$\mathbf{S}_b$ = back depth-cue scale factor [from XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS]

The function (scalar * color) means different things, depending on the Device color type.

For indexed color, scalar * color → color´ means:

color´.index = scalar * (color.index — ramp.bottom) + ramp.bottom

(Where ramp.bottom is the smallest index in the ramp specified by color.index. See **XGL_CMAP_RAMP_LIST**(3))

For RGB color, scalar * color → color´ means:

color´.r = scalar * color.r

color´.g = scalar * color.g

color´.b = scalar * color.b

The possible values for XGL_3D_CTX_DEPTH_CUE_MODE are:

XGL_DEPTH_CUE_OFF

No depth-cueing is performed.

XGL_DEPTH_CUE_LINEAR

A linear depth-cueing function is applied. The color at each vertex of a primitive is calculated using the linear weighting technique shown below. Calculation of the weighting factor depends on the value of the attribute **XGL_CTX_VDC_-ORIENTATION**(3) using the following test when $\mathbf{Z}_f$ and $\mathbf{Z}_b$ are initialized:

if (XGL_CTX_VDC_ORIENTATION is XGL_Y_DOWN_Z_AWAY) {

$\mathbf{Z}_f$ = VDC *z* minimum

$\mathbf{Z}_b$ = VDC *z* maximum

} else { /* XGL_Y_UP_Z_TOWARD */

$\mathbf{Z}_f$ = VDC *z* maximum

$\mathbf{Z}_b$ = VDC *z* minimum

}

The resultant depth-cued color is calculated as follows:

Z in front of $\mathbf{Z}_f$:

$$\mathbf{C}_i' = \mathbf{C}_i$$

Z behind $\mathbf{Z}_b$:

$$\mathbf{C}_i' = \mathbf{C}_d$$

Z between $\mathbf{Z}_f$ and $\mathbf{Z}_b$:

$$\mathbf{r} = (\mathbf{Z} - \mathbf{Z}_b) / (\mathbf{Z}_f - \mathbf{Z}_b)$$

$$\mathbf{C}_i' = \mathbf{r} * \mathbf{C}_i + (1 - r) * \mathbf{C}_d$$

The weighting factor, **r**, is a normalized depth of each vertex. It determines the proportion of the primitive's color that will be mixed with the depth-cued color (described below).

Note that if an application has **XGL_CTX_VDC_MAP**(3) set to XGL_VDC_MAP_OFF, then it must adjust the value of **XGL_CTX_DC_VIEWPORT**(3) by the maximum Z value given by **XGL_DEV_MAXIMUM_COORDINATES**(3)**,** which varies across devices because of the actual depth of the Z-buffer. This adjustment is necessary for both correct Z-buffering and depth cueing. XGL automatically makes this adjustment for values of **XGL_CTX_VDC_MAP**(3) other than XGL_VDC_MAP_OFF.

XGL_DEPTH_CUE_SCALED

Similar to XGL_DEPTH_CUE_LINEAR in that a linear weighting function is applied. However, it is only applied between two depth-cueing reference planes that are specified by the application using the attribute **XGL_3D_CTX_DEPTH_CUE_REF_-PLANES**(3)**.** Also, the proportion of the original primitive's color that is mixed with the depth-cued color can be specified at the two reference planes using the attribute **XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS**(3)**.**

The resultant depth-cued color is calculated as follows:

Z in front of Z_f :

$$\mathbf{C}_i' = \mathbf{S}_f * \mathbf{C}_i + (1 - \mathbf{S}_f) * \mathbf{C}_d$$

Z behind $\mathbf{Z}_b$:

$$\mathbf{C}_i' = \mathbf{S}_b * \mathbf{C}_i + (1 - \mathbf{S}_b) * \mathbf{C}_d$$

Z between $\mathbf{Z}_f$ and $\mathbf{Z}_b$:

$$\mathbf{r} = \mathbf{S}_b + ((\mathbf{Z} - \mathbf{Z}_b) * (\mathbf{S}_f - \mathbf{S}_b) / (\mathbf{Z}_f - \mathbf{Z}_b))$$

$$\mathbf{C}_i' = \mathbf{r} * \mathbf{C}_i + (1 - r) * \mathbf{C}_d$$

The determination of the depth-cued color, $\mathbf{C}_d$, varies depending upon the Device color type, specified by the attribute **XGL_DEV_COLOR_TYPE**(3)**.** The two alternatives are as follows:

1. When the Device color type is XGL_COLOR_INDEX, the depth-cued color is the color index at the beginning of the *color ramp* associated with the vertex color. (For a description of color ramps, see **XGL_CMAP_RAMP_LIST**(3)**.**) Thus, for depth-cueing to be calculated properly with indexed colors, a color ramp must be associated with the vertex color.
2. When the Device color type is XGL_COLOR_RGB, the depth-cued color is the color specified by the attribute **XGL_3D_CTX_DEPTH_CUE_COLOR**(3)**.**

In all cases where depth-cueing is in effect, the attribute XGL_3D_CTX_DEPTH_CUE_INTERP controls how interpolation along rendered lines is performed.

The default value is XGL_DEPTH_CUE_OFF.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_3D_CTX_DEPTH_CUE_INTERP(3)
XGL_3D_CTX_DEPTH_CUE_REF_PLANES(3)
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS(3)
XGL_CMAP_RAMP_LIST(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_3D_CTX_DEPTH_CUE_REF_PLANES – defines the planes that specify where depth-cueing occurs under certain conditions

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx ctx;
xgl_object_set(ctx, XGL_3D_CTX_DEPTH_CUE_REF_PLANES, double *, 0);
xgl_object_get(ctx, XGL_3D_CTX_DEPTH_CUE_REF_PLANES, double *);
```

DESCRIPTION

Depth-cueing attenuates the color of 3D primitives based upon their depth (*z*-value in VDC space). This attribute is used only when the depth-cueing mode, as specified by **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**,** is set to XGL_DEPTH_CUE_SCALED. In this case, the depth-cueing function is only used between the two reference planes, *front* and *back*, specified by this attribute.

The two planes are normal to the *z*-axis in the VDC coordinate system and are specified as two *z*-values expressed as an array of two doubles. The first value in the array corresponds to the front reference plane, and the second value to the back plane.

Because the planes are specified in VDC, they are dependent on the value of the attribute **XGL_CTX_VDC_ORIENTATION**(3)**.** When the VDC orientation is XGL_Y_DOWN_Z_AWAY, the *z*-value of the front plane must be less than or equal to that of the back plane. In the opposite case, when VDC orientation is XGL_Y_UP_Z_TOWARD, the *z*-value of the front plane must be greater than or equal to that of the back plane.

For a complete description of scaled depth-cueing, see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.**

The default value is (0.0, 1.0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS(3)
XGL_CMAP_RAMP_LIST(3)
XGL_DEV_COLOR_TYPE(3)

NOTES

The default values for the reference planes are specified based on a **XGL_CTX_VDC_-ORIENTATION**(3) of XGL_Y_DOWN_Z_AWAY. Thus, if **XGL_CTX_VDC_ORIENTATION**(3) is changed to XGL_Y_UP_Z_TOWARD, and **XGL_3D_CTX_DEPTH_CUE_MODE**(3) is set to XGL_-DEPTH_CUE_SCALED before the reference planes are changed, an error message is generated and the values of the front and back reference planes are exchanged (so that the back reference plane is indeed behind the front reference plane). To avoid this, when the orientation is XGL_Y_UP_Z_TOWARD, set the reference planes to the values that you want to use before setting depth cueing to XGL_DEPTH_CUE_SCALED.

NAME

XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS – defines scale factors used in mixing primitive and depth-cued colors under certain conditions

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, float *, 0);
xgl_object_get(ctx, XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, float *);
```

DESCRIPTION

Depth-cueing attenuates the color of 3D primitives based upon their depth (*z* value in VDC space). This attribute modifies the appearance of depth-cued primitives and is only used when the depth-cueing mode, as specified by **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**,** is set to XGL_DEPTH_CUE_SCALED. In this case, depth-cueing occurs only between the two reference planes, *front* and *back*, specified by the attribute **XGL_3D_CTX_DEPTH_CUE_REF_-PLANES**(3)**.** Then, XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS defines scale factors used to determine the portion of the primitive's color that is mixed with the *depth-cued* color (described below).

The front and back scale factors are specified as two *z*-values expressed as an array of two floats. The first value in the array corresponds to the front scale factor, and the second value to the back scale factor.

The front scale factor defines the portion of the primitive color that is combined with (1 – front_scale_factor) times the depth-cued color at, or in front of, the front reference plane. Likewise, the back scale factor specifies the portion of the primitive color applied at, or behind, the back reference plane. In between the two planes, the portion of the depth-cued color combined with the primitive color is modulated by linearly interpolating between the two scaling factors. For a more complete description of scaled depth-cueing, see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.**

The determination of the depth-cued color varies depending on the Device color type, specified by the attribute **XGL_DEV_COLOR_TYPE**(3)**.** The two alternatives are as follows:

1. When the Device color type is XGL_COLOR_INDEX, the depth-cued color is the color index at the beginning of the *color ramp* associated with the vertex color. (For a description of color ramps, see **XGL_CMAP_RAMP_LIST**(3)**.**) Thus, for depth-cueing to be calculated properly with indexed colors, a color ramp must be associated with the vertex color.
2. When the Device color type is XGL_COLOR_RGB, the depth-cued color is the color specified by the attribute **XGL_3D_CTX_DEPTH_CUE_COLOR**(3)**.**

The default value is (1.0, 0.0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_REF_PLANES(3)
XGL_CMAP_RAMP_LIST(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_3D_CTX_HLHSR_DATA – defines data used to clear the *z*-buffer

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_HLHSR_DATA, Xgl_hlhsr_data *, 0);
xgl_object_get(ctx, XGL_3D_CTX_HLHSR_DATA, Xgl_hlhsr_data *);
```

DESCRIPTION

The operator **xgl_context_new_frame**(3) clears the device coordinate (DC) viewport in a way specified by the attribute **XGL_CTX_NEW_FRAME_ACTION**(3)**.** If one of the actions specified by XGL_CTX_NEW_FRAME_ACTION is XGL_CTX_NEW_FRAME_HLHSR_ACTION, *and* if the attribute **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER, then this attribute contains the value to write into the *z*-buffer for the new frame. The value must be between 0.0 and the maximum *z*-value allowed by the Device as specified by the attribute **XGL_DEV_MAXIMUM_COORDINATES**(3)**.** Values outside the maximum *z*-value will get clamped.

XGL sets the value of XGL_3D_CTX_HLHSR_DATA to the *z*-value of **XGL_DEV_MAXIMUM_-COORDINATES**(3) when an application associates a Device with a 3D Context.

The default value is device independent: infinity.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_new_frame(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_CTX_NEW_FRAME_ACTION(3)
XGL_DEV_MAXIMUM_COORDINATES(3)

NAME

XGL_3D_CTX_HLHSR_MODE – defines the mode used for hidden line / hidden surface removal

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_HLHSR_MODE, Xgl_hlhsr_mode, 0);
xgl_object_get(ctx, XGL_3D_CTX_HLHSR_MODE, Xgl_hlhsr_mode *);
```

DESCRIPTION

This attribute specifies whether hidden line and hidden surface removal (HLHSR) will be done on primitives rendered using the 3D Context *ctx*. A standard *z*-buffer technique is used to accomplish hidden line and hidden surface removal. The *z*-buffering is useful when rendering 3D images because primitives that are obscured by objects closer to the viewer are not shown. Two options are available for this attribute:

XGL_HLHSR_NONE
: Primitives are rendered into the frame buffer in the order they are received. Primitives that are near the observer can be overlapped by those farther away.

XGL_HLHSR_Z_BUFFER
: Each pixel in a primitive is rendered into the frame buffer only if its device coordinate *z*-value is less than or equal to the *z*-value of the pixel currently stored in the frame buffer.

The *z*-buffer is cleared by calling **xgl_context_new_frame**(3)**,** providing the attribute **XGL_CTX_NEW_FRAME_ACTION**(3) allows for XGL_CTX_NEW_FRAME_HLHSR_ACTION and that **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER. The *z*-buffer is cleared to the value specified by **XGL_3D_CTX_HLHSR_DATA**(3)**.**

The default value is XGL_HLHSR_NONE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_new_frame(3)
XGL_3D_CTX_HLHSR_DATA(3)
XGL_CTX_NEW_FRAME_ACTION(3)

NOTES

XGL does not clear or initialize the contents of the *z*-buffer upon an expose event. For the contents of the *z*-buffer to be initialized or reinitialized, the application must explicitly call **xgl_context_new_frame**(3) with the attribute **XGL_CTX_NEW_FRAME_ACTION**(3) allowing for XGL_CTX_NEW_FRAME_HLHSR_ACTION.

Such an exposure occurs upon, but may not be limited to, the following:

- The creation of the raster.
- A resize event.
- ''Popping'' the raster above others.

On devices that implement a software *z*-buffer, the first Context attached to a given Raster that has the setting XGL_HLHSR_Z_BUFFER causes the allocation of the software *z*-buffer. This *z*-buffer remains allocated until the Raster is destroyed. To determine whether your device uses a software *z*-buffer, check the *hlhsr_mode* field returned by the operator **xgl_inquire**(3)**.**

When an application requests a VDC mapping (see **XGL_CTX_VDC_MAP**(3)) of XGL_VDC_MAP_DEVICE and *z*-buffering is enabled, the range of the *z*-values in the application's data should be greater than the range of the *x*-values. Otherwise, incorrect values may be written into the *z*-buffer.

The results of picking are affected by this mode. If XGL_3D_CTX_HLHSR_MODE is XGL_HLHSR_NONE, all objects in a pick aperture can be picked. If it is on, only the visible (that is, not obscured by other primitives) can be picked.

NAME

XGL_3D_CTX_JITTER_OFFSET – specifies the amount by which the geometry should be offset in DC before drawing

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_JITTER_OFFSET, Xgl_pt_d2d*, 0);
xgl_object_get(ctx, XGL_3D_CTX_JITTER_OFFSET, Xgl_pt_d2d*);
```

DESCRIPTION

This attribute specifies the amount by which the geometry should be offset in DC before drawing. Alternatively, jitter offset is the value by which the DC viewport is offset before rendering the primitive.

An application can use jitter offset to render an image before accumulating.

The default value is (0.0, 0.0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_accumulate(3)

NAME

XGL_3D_CTX_LIGHTS – list of Lights in a 3D Context

SYNOPSIS

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

Xgl_3d_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_LIGHTS, Xgl_light [], 0);
xgl_object_get(ctx, XGL_3D_CTX_LIGHTS, Xgl_light []);
```

DESCRIPTION

This attribute is an array of Lights used by the 3D Context *ctx*. The number of Lights in the list is given by the 3D Context attribute **XGL_3D_CTX_LIGHT_NUM**(3)**.** The 3D Context has a corresponding list of switches given by **XGL_3D_CTX_LIGHT_SWITCHES**(3) for turning on and off the individual Lights.

An application can set XGL_3D_CTX_LIGHT_NUM to the required number of Lights, and then get the list of Lights. This gives the application access to the individual Lights used by the Context as given by XGL_3D_CTX_LIGHTS.

Alternatively, an application can explicitly create its own list of Lights with the operator **xgl_object_create**(3)**.** The number in this list should be the same as the value of XGL_3D_-CTX_LIGHT_NUM. Then the application can set XGL_3D_CTX_LIGHTS with its list. XGL destroys the Context's list of Lights (provided that no other 3D Context is using these Lights) before setting XGL_3D_CTX_LIGHTS to the list specified by the application.

The default value is NULL.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_LIGHT_NUM(3)
XGL_3D_CTX_LIGHT_SWITCHES(3)

NAME

XGL_3D_CTX_LIGHT_NUM – number of Lights in a 3D Context

SYNOPSIS

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

Xgl_3d_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_LIGHT_NUM, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_3D_CTX_LIGHT_NUM, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is the number of Lights in a 3D Context. When an application sets this attribute, XGL changes the number of Lights and their corresponding on and off switches to match this attribute. If the value of this attribute is *m*, the list of Lights as given by **XGL_3D_CTX_LIGHTS**(3) and its list of switches as given by **XGL_3D_CTX_LIGHT_SWITCHES**(3) are indexed from 0 to *m*-1. If the new number *n* exceeds the current number *m*, XGL creates *n*-*m* Lights and switches and appends these to the appropriate lists in the Context at index positions *m* to *n*-1. The light switches for the new lights are *on* by default. The Lights and switches that existed before increasing the number of Lights remain unchanged. If the new number is less than the current number, XGL destroys Lights and switches with indices *n* to *m*-1 (providing no other 3D Context is using these Lights).

The default value is 0.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
xgl_object_destroy(3)
XGL_3D_CTX_LIGHT_SWITCHES(3)
XGL_3D_CTX_LIGHTS(3)

NAME

XGL_3D_CTX_LIGHT_SWITCHES – list of switches for a 3D Context's Lights

SYNOPSIS

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

Xgl_3d_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_LIGHT_SWITCHES, Xgl_boolean [], 0);
xgl_object_get(ctx, XGL_3D_CTX_LIGHT_SWITCHES, Xgl_boolean []);
```

DESCRIPTION

This attribute is an array of on and off switches corresponding to the 3D Context's array of Lights, given by **XGL_3D_CTX_LIGHTS**(3)**.** The number of switches in the list is given by **XGL_3D_CTX_LIGHT_NUM**(3)**.**

The Context's array of Lights have indices 0 to *n*-1, where *n* is the value of XGL_3D_CTX_-LIGHT_NUM. An application can turn on Light *i* by setting flag *i* in its own array of switches to TRUE. Similarly, FALSE turns a Light off. After establishing the values of all switches, an application can set XGL_3D_CTX_LIGHT_SWITCHES with this array to turn Lights on or off.

The default value is NULL.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_object_create(3)
XGL_3D_CTX_LIGHT_NUM(3)
XGL_3D_CTX_LIGHTS(3)

NAME

XGL_3D_CTX_LINE_COLOR_INTERP – controls color interpolation along 3D lines

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_LINE_COLOR_INTERP, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_LINE_COLOR_INTERP, Xgl_boolean *);
```

DESCRIPTION

This attribute specifies whether a line, when drawn with an XGL operator that renders line segments (such as **xgl_multipolyline**(3)) in the 3D Context *ctx*, will have a color that is linearly interpolated between the colors of its vertices. Possible values are:

TRUE The interpolation will be done. Vertices must have a color associated with them. The distance between vertices along the line is used to control the interpolation. The interpolation is done in device coordinates. Thus, for instance, two lines that project to an equal length on the device show the same color-stepping interval, regardless of their orientation in world coordinates.

FALSE No line color interpolation will be done. The color of the line segment *n–1* to *n* is the color at vertex *n*.

Color interpolation applies only to primitives rendering 3D line segments that have colors specified at each vertex. The edges of a polygon cannot be rendered using the line color interpolation attribute.

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multipolyline(3)

NAME

XGL_3D_CTX_MODEL_CLIP_PLANES – specifies list of model clipping planes to use

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_MODEL_CLIP_PLANES, Xgl_plane [], 0);
xgl_object_get(ctx, XGL_3D_CTX_MODEL_CLIP_PLANES, Xgl_plane []);
```

DESCRIPTION

This attribute specifies the list of *model clipping* planes to use when rendering primitives within the Context *ctx*. Model clipping is provided to allow clipping of graphics primitives before viewing. The application uses this attribute in conjunction with the attribute XGL_3D_CTX_MODEL_CLIP_PLANE_NUM to specify which and how many model clipping planes should be considered when rendering XGL primitives.

Setting a new list of model clipping planes *replaces* the previous (if any) list of model clipping planes. The application is also responsible for making sure that enough memory is available when getting the list of clipping planes from XGL.

Each model clipping plane is defined as a point in world coordinates (WC), and as a unit vector normal to the plane, also in world coordinates, pointing in the direction of the half space.

All the graphics on the side of the *normal vector* are kept, and the graphics on the other side are clipped. Providing several planes defines a clipping volume, limited by a set of half spaces.

XGL_3D_CTX_MODEL_CLIP_PLANES is a pointer to a C structure of type *Xgl_plane*, which is defined as follows:

```
typedef struct {
        Xgl_pt_d3d    pt;       /* a point on the plane in WC */
        Xgl_pt_d3d    normal;   /* normal vector components */
} Xgl_plane;
```

When performing model clipping, XGL considers all of the enabled model clipping planes, in the order the application has provided them.

By default, XGL does not perform any model clipping.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)

NOTES

Model clipping of some primitives (annotation text, or annotation 3D circle, arc, rectangle, or ellipse, ...) is performed only on the reference point. If the reference point is clipped, the entire primitive is considered clipped.

Model clipping of primitives defined with a point type that lacks an edge flag field (XGL_PT_F3D, XGL_PT_COLOR_NORMAL_F3D,... for example) will generate a clipped primitive with all edges visible, even on the modeling clipping plane boundaries.

NAME

XGL_3D_CTX_MODEL_CLIP_PLANE_NUM – specifies the number of model clipping planes in use

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_MODEL_CLIP_PLANE_NUM, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_3D_CTX_MODEL_CLIP_PLANE_NUM, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the number of *model clipping* planes in use when rendering primitives within the Context *ctx*. The application uses this attribute in conjunction with the attribute XGL_3D_CTX_MODEL_CLIP_PLANES to specify which and how many model clipping planes should be considered when rendering XGL primitives.

Setting this attribute to 0 disables model clipping. Setting it back to a positive value enables model clipping.

When additional model clipping planes must be defined, the application must provide both the number of planes and the list of model clipping planes. To enable the first *n* planes of those already defined, the application may set only the new number of planes, without sending the list of model clipping planes to XGL.

When performing model clipping, XGL considers all the enabled model clipping planes, in the order the application provides them.

The default value is ‘‘no model clipping planes enabled’’ — that is, 0 (zero).

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_MODEL_CLIP_PLANES(3)

NAME

XGL_3D_CTX_NORMAL_TRANS – the Normal Transform

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_get(ctx, XGL_3D_CTX_NORMAL_TRANS, Xgl_trans *);
```

DESCRIPTION

This read-only attribute is the Transform object used for the Normal Transform, which transforms normal vectors from model coordinates (MC) to world coordinates (WC). The Transform's value is the inverse of the Model Transform (see **XGL_CTX_MODEL_TRANS**(3)).

To improve performance in XGL 3.0 and subsequent releases, evaluation of the Normal Transform is deferred until either XGL needs the Normal Transform internally or the application explicitly gets it. Before attempting to use the Normal Transform, call **xgl_object_get**(3) to get XGL_3D_CTX_NORMAL_TRANS to ensure XGL evaluates the Normal Transform if it requires updating.

Since this attribute is read-only, do not get a handle to this Transform and use it to set **XGL_CTX_LOCAL_MODEL_TRANS**(3), **XGL_CTX_GLOBAL_MODEL_TRANS**(3), or **XGL_CTX_VIEW_-TRANS**(3).

The Transform of this read-only Context attribute always has **XGL_TRANS_DATA_TYPE**(3) set to XGL_DATA_DBL.

The default value is the Identity Transform.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_FACE_CULL(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_MODEL_TRANS(3)
XGL_CTX_VIEW_TRANS(3)
XGL_TRANS_DATA_TYPE(3)

NAME

XGL_3D_CTX_SURF_DC_OFFSET – offsets in *z* the DC coordinates of 3D polygons

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_DC_OFFSET, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_DC_OFFSET, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether an offset should be added to polygons in device coordinates (DC) to backup polygons from their computed depth position. If TRUE, a DC offset is computed for each polygon rendered, using all the vertices of the polygon, and this offset is further added to the *z*-component of each vertex of the polygon before the polygon is rendered. Therefore, the polygon will be backed from its computed position. This attribute can be used to lay other primitives such as polylines ''on top'' of polygons, without unwanted *z*-buffer interactions.

Typically, when *z*-buffering is on, drawing a polygon followed by another graphic primitive to generate a *decal* effect will generate incomplete rendering of the last primitive. This is so because *z*-buffer values prevent the second primitive being rendered.

This attribute ensures that the first polygon will be behind its computed *z*-position in device coordinates so that the decal will be correctly rendered.

When FALSE, no offset is computed.

This attribute has no effect when the value of the Context attribute **XGL_3D_CTX_HLHSR_MODE**(3) is not XGL_HLHSR_Z_BUFFER.

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_HLHSR_MODE(3)

NAME

XGL_3D_CTX_SURF_FACE_CULL – controls face culling in a 3D Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_FACE_CULL, Xgl_surf_cull_mode, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FACE_CULL, Xgl_surf_cull_mode *);
```

DESCRIPTION

This attribute controls how face culling is done within the 3D Context *ctx*. When a face of a surface is ''culled,'' it is not rendered by XGL. For example, if an application programmer did not care about surfaces facing away from the viewer, the programmer could invoke back-face culling, which causes these surfaces to be ignored. The data type *Xgl_surf_cull_mode* takes on values:

XGL_CULL_OFF
: No culling is done.

XGL_CULL_FRONT
: Front-facing primitives are culled.

XGL_CULL_BACK
: Back-facing primitives are culled.

The default value is XGL_CULL_OFF.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NAME

XGL_3D_CTX_SURF_FACE_DISTINGUISH – controls whether front- and back-facing surfaces are distinguished from each other

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_FACE_DISTINGUISH, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FACE_DISTINGUISH, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether back-facing surfaces are distinguished from front-facing ones. If TRUE, back-facing primitives use the XGL_3D_CTX_SURF_BACK_... attributes. Otherwise, all surfaces use the XGL_3D_CTX_SURF_FRONT_... and XGL_CTX_SURF_FRONT_... attributes.

This attribute does not affect the processing of facets to determine if they are front- or back-facing. Thus, facets may be culled, for instance, by using the attribute **XGL_3D_CTX_SURF_FACE_CULL**(3)**,** even when XGL_3D_CTX_SURF_FACE_DISTINGUISH is FALSE.

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(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_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_SILHOUETTE_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)

NAME

XGL_3D_CTX_SURF_BACK_AMBIENT,
XGL_3D_CTX_SURF_FRONT_AMBIENT – define ambient lighting coefficients for a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_AMBIENT, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_AMBIENT, float, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_AMBIENT, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_AMBIENT, float *);
```

DESCRIPTION

These attributes define the ambient reflection coefficients used for lighting calculations within the 3D Context *ctx*.

XGL_3D_CTX_SURF_BACK_AMBIENT specifies the ambient reflection coefficient used for back-facing surfaces, and XGL_3D_CTX_SURF_FRONT_AMBIENT specifies the coefficient used for front-facing surfaces. The Context attribute **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) must be TRUE for XGL_3D_CTX_SURF_BACK_AMBIENT to be used (otherwise, XGL_3D_CTX_-SURF_FRONT_AMBIENT is used for both front-facing and back-facing surfaces).

The default value for each attribute is 0.0.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
The Lighting Equations section of the *XGL Programmer's Guide*.

NAME

XGL_3D_CTX_SURF_BACK_DIFFUSE,
XGL_3D_CTX_SURF_FRONT_DIFFUSE – define diffuse lighting coefficients for a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_DIFFUSE, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DIFFUSE, float, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_DIFFUSE, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_DIFFUSE, float *);
```

DESCRIPTION

These attributes define the diffuse reflection coefficients used for lighting calculations within the 3D Context *ctx*.

XGL_3D_CTX_SURF_BACK_DIFFUSE specifies the diffuse reflection coefficient used for back-facing surfaces, and XGL_3D_CTX_SURF_FRONT_DIFFUSE specifies the coefficient used for front-facing surfaces. The Context attribute **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) must be TRUE for XGL_3D_CTX_SURF_BACK_DIFFUSE to be used (otherwise, XGL_3D_CTX_SURF_-FRONT_DIFFUSE is used for both front-facing and back-facing surfaces).

The default value for each attribute is 1.0.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)

The Lighting Equations section of the *XGL Programmer's Guide*.

NAME

XGL_3D_CTX_SURF_BACK_DMAP
XGL_3D_CTX_SURF_FRONT_DMAP – specify a list of Data Map Texture objects in a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_DMAP, Xgl_data_map [], 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DMAP, Xgl_data_map [], 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_DMAP, Xgl_data_map []);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_DMAP, Xgl_data_map []);
```

DESCRIPTION

These attributes specify an array of Data Map Texture objects used by the 3D Context *ctx*. The number of data map *textures* in the lists is given by the 3D Context attributes **XGL_3D_CTX_SURF_BACK_DMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_DMAP_NUM**(3). The 3D Context has a corresponding list of switches given by **XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES**(3) and **XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES**(3) for turning on and off the individual data map *textures*.

An application can set **XGL_3D_CTX_SURF_BACK_DMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_DMAP_NUM**(3) to the required number of data map *textures*, and then get the list of data map *textures*. This gives the application access to the individual data map *textures* that the Context uses as given by XGL_3D_CTX_SURF_BACK_DMAP and XGL_3D_CTX_SURF_FRONT_DMAP.

Alternatively, an application can explicitly create its own list of data map *textures* with the operator **xgl_object_create**(3). The number in the list should be the same as the number specified in **XGL_3D_CTX_SURF_BACK_DMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_DMAP_NUM**(3). The application can now set XGL_3D_CTX_SURF_BACK_DMAP and XGL_3D_CTX_SURF_FRONT_DMAP with this list. XGL destroys the Context's list of Data Map Texture objects (provided that no other 3D Context is using them) before setting XGL_3D_CTX_SURF_BACK_DMAP and XGL_3D_CTX_SURF_FRONT_DMAP to the list specified by the application.

When multiple data maps are active, they are applied sequentially and are cumulative. That is, data map 0 is applied before data map 1 and so on.

The default value for each attribute is NULL.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_SURF_BACK_DMAP_NUM(3)
XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_DMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES(3)

NOTES

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_3D_CTX_SURF_BACK_DMAP_NUM
XGL_3D_CTX_SURF_FRONT_DMAP_NUM – specify the number of Data Map Texture objects in a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_DMAP_NUM, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_NUM, Xgl_usgn32, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_DMAP_NUM, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_NUM, Xgl_usgn32 *);
```

DESCRIPTION

These attributes set the number of Data Map Texture objects in a 3D Context. When an application sets these attributes, XGL changes the number of Data Map Texture objects and their corresponding on and off switches to match these attributes. If the value of these attributes is *m*, the lists of Data Map Texture objects given by **XGL_3D_CTX_SURF_-BACK_DMAP** and **XGL_3D_CTX_SURF_FRONT_DMAP** and their lists of switches as given by **XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES** and **XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES** are indexed from 0 to *m*-1. If the new number *n* exceeds the current number *m*, XGL creates *n-m* Data Map Texture objects and switches and appends these to the appropriate lists in the Context at index positions *m* to *n*-1.

The data map texture switches are on by default. The Data Map Texture objects and switches that existed before increasing the number of Data Map Texture objects remain unchanged. If the number is less than the current number, XGL destroys the Data Map Texture objects and switches with indices *n* to *m*-1 (provided that no other 3D Context is using them).

The default value for each attribute is 0.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_SURF_BACK_DMAP(3)
XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_DMAP(3)
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES(3)

NOTES

Texturing is enabled when these attributes are non-zero and the 3D surface primitive has data in its vertex information.

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES – specify the list of switches for a 3D Context's Data Map Texture objects

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES, Xgl_boolean [], 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES, Xgl_boolean [], 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES, Xgl_boolean []);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES, Xgl_boolean []);
```

DESCRIPTION

These attributes specify an array of on and off switches corresponding to the 3D Context's array of Data Map Texture objects, given by **XGL_3D_CTX_SURF_FRONT_DMAP**(3) and **XGL_3D_CTX_SURF_BACK_DMAP**(3). The number of switches in the list is given by **XGL_3D_CTX_SURF_FRONT_DMAP_NUM**(3) and **XGL_3D_CTX_SURF_BACK_DMAP_NUM**(3).

The Context's array of *textures* have indices 0 to *n*-1, where *n* is the value of **XGL_3D_CTX_SURF_FRONT_DMAP_NUM**(3) and **XGL_3D_CTX_SURF_BACK_DMAP_NUM**(3). An application can turn on texture *i* by setting flag *i* in its own array of switches to TRUE. Similarly, FALSE turns a *texture* off. After establishing the values of all switches, an application can set these attributes with this array to turn *textures* on or off.

The default value for each attribute is NULL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_BACK_DMAP(3)
XGL_3D_CTX_SURF_BACK_DMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_DMAP(3)
XGL_3D_CTX_SURF_FRONT_DMAP_NUM(3)

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_3D_CTX_SURF_BACK_ILLUMINATION, XGL_3D_CTX_SURF_FRONT_ILLUMINATION – specify the model used to calculate the illumination of surface primitives

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_ILLUMINATION, Xgl_surf_illumination, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_ILLUMINATION, Xgl_surf_illumination, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_ILLUMINATION, Xgl_surf_illumination *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_ILLUMINATION, Xgl_surf_illumination *);
```

DESCRIPTION

These attributes dictate where and how the illumination calculation is performed.

XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3) specifies the illumination model used for front-facing surfaces in the 3D Context *ctx*, and **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3) specifies the illumination model for back-facing surfaces. Four different options are available for this attribute:

XGL_ILLUM_NONE
: No illumination calculation is done. The surface is filled with a single color, as specified by the surface color. The surface will have this constant color, regardless of its position.

XGL_ILLUM_NONE_INTERP_COLOR
: No illumination calculation is done. XGL directly interpolates vertex colors across the surface.

XGL_ILLUM_PER_FACET
: The illumination calculation is done once for the surface. The surface is filled with a single color computed from the surface color, the surface normal, and the other illumination attributes. (This illumination model is also known as *flat* shading.)

  XGL selects the normal vector for the facet illumination calculation as follows: The normal in the primitive's facet data is used if the application specifies it; otherwise the normal is calculated from the vertices (see **XGL_3D_CTX_SURF_GEOM_NORMAL**(3)).

  However, if the facet is front-facing and the value of **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) is XGL_SURF_COLOR_VERTEX_ILLUM_INDEP, then the normal associated with the first vertex of the facet is used when the application specifies normals with vertices; in the absence of vertex normals, the normal is selected by the method in the previous paragraph. If the facet is back-facing, a similar procedure applies for **XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR**(3)**.**

XGL_ILLUM_PER_VERTEX
: The illumination calculation is done at each vertex; the resulting vertex colors are then interpolated across the surface. The vertex color and vertex normal are used

as input into the illumination calculation. If neither of the vertex color nor the vertex normal is supplied, the surface color or surface normal, respectively, are used. (This illumination model is also known as *Gouraud* shading.)

In the discussion above, the *surface color* used for the illumination is determined by the attributes **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_CTX_SURF_BACK_COLOR_-SELECTOR**(3)**.** For more information, see the man pages for these attributes.

The default value for each attribute is XGL_ILLUM_NONE.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR(3)
XGL_CTX_SURF_FRONT_COLOR_SELECTOR(3)

NOTES

Unexpected results can occur if the application uses XGL_ILLUM_PER_VERTEX illumination with point list data whose color indices lie in two or more different color ramps. In other words, XGL does not force all of the vertex color values of a ''per-vertex'' shaded polygon to lie within the same color ramp. It is the application's responsibility to ensure that the vertex colors for a single per-vertex shaded polygon are all within the same ramp.

NAME

XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT, XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT – define which components of the lighting equation are used

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT,
        Xgl_light_enable_component, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
        Xgl_light_enable_component, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT,
        Xgl_light_enable_component *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
        Xgl_light_enable_component *);
```

DESCRIPTION

These attributes specify which components of the illumination equation are used to compute a color within the 3D Context *ctx*. There are three components: *ambient*, *diffuse*, and *specular*. The application can enable any combination of the components by logically OR'ing flags representing each component. The flags are:

XGL_LIGHT_ENABLE_COMP_AMBIENT
: Enable the ambient component of the illumination equation.

XGL_LIGHT_ENABLE_COMP_DIFFUSE
: Enable the diffuse component of the illumination equation.

XGL_LIGHT_ENABLE_COMP_SPECULAR
: Enable the specular component of the illumination equation.

Front-facing surfaces use **XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT**(3). Back-facing surfaces use **XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT**(3). The Context attribute **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) must be TRUE for XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT to be used (otherwise, XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT is used for both front-facing and back-facing surfaces).

The default value for these attributes is XGL_LIGHT_ENABLE_COMP_AMBIENT | XGL_LIGHT_ENABLE_COMP_DIFFUSE

These attributes are pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)

NAME

XGL_3D_CTX_SURF_BACK_SPECULAR,
XGL_3D_CTX_SURF_FRONT_SPECULAR,
XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR,
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR,
XGL_3D_CTX_SURF_BACK_SPECULAR_POWER,
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER – define values used in specular component of lighting equation

SYNOPSIS

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

Xgl_ctx         ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR, Xgl_color *, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, Xgl_color *, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR_POWER, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, float, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR, Xgl_color *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, Xgl_color *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_SPECULAR_POWER, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, float *);
```

DESCRIPTION

These attributes define the specular coefficient, the specular color, and the specular power used in computing lighting within the 3D Context *ctx*.

Front-facing surfaces use **XGL_3D_CTX_SURF_FRONT_SPECULAR**(3), **XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR**(3), and **XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER**(3). Back-facing surfaces use **XGL_3D_CTX_SURF_BACK_SPECULAR**(3), **XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR**(3), and **XGL_3D_CTX_SURF_BACK_SPECULAR_POWER**(3). The Context attribute **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) must be TRUE for the back-facing surfaces to use the back-facing attributes. Otherwise, the front-facing attributes are used for both the front-facing and back-facing surfaces.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

The default value for the specular coefficient is 0.0.

The default value for the specular color is: INDEX 1 or RGB (green).

The default value for the specular power is 32.0.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)

The Lighting Equations section of the *XGL Programmer's Guide*.

NAME

XGL_3D_CTX_SURF_BACK_TMAP
XGL_3D_CTX_SURF_FRONT_TMAP – specify a list of Texture Map objects in a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_TMAP, Xgl_tmap [], 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP, Xgl_tmap [], 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_TMAP, Xgl_tmap []);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_TMAP, Xgl_tmap []);
```

DESCRIPTION

These attributes specify an array of Texture Map objects used by the 3D Context *ctx*. The number of Texture Map objects in the list is given by the 3D Context attributes **XGL_3D_CTX_SURF_BACK_TMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP_NUM**(3)**.** The 3D Context has a corresponding list of switches given by **XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES**(3) for turning on and off the individual Texture Map objects.

An application can set **XGL_3D_CTX_SURF_BACK_TMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP_NUM**(3) to the required number of Texture Map objects, and then get the list of *textures*. This gives the application access to the individual Texture Map objects that the Context uses as given by XGL_3D_CTX_SURF_BACK_TMAP and XGL_3D_CTX_SURF_FRONT_TMAP.

Alternatively, an application can explicitly create its own list of Texture Map objects with the operator **xgl_object_create**(3)**.** The number in the list should be the same as the number specified in **XGL_3D_CTX_SURF_BACK_TMAP_NUM**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP_NUM**(3)**.** The application can now set XGL_3D_CTX_SURF_BACK_TMAP and XGL_3D_CTX_SURF_FRONT_TMAP with this list. XGL destroys the Context's list of Texture Map objects (provided that no other 3D Context is using them) before setting XGL_3D_CTX_SURF_BACK_TMAP and XGL_3D_CTX_SURF_FRONT_TMAP to the list specified by the application.

When multiple texture map objects are active, they are applied sequentially and are cumulative. That is, texture map 0 is applied before texture map 1 and so on.

The default value for each attribute is NULL.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_3D_CTX_SURF_BACK_TMAP_NUM(3)
XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_TMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES(3)

NOTES

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

NAME

XGL_3D_CTX_SURF_BACK_TMAP_NUM
XGL_3D_CTX_SURF_FRONT_TMAP_NUM – specify the number of Texture Map objects in a 3D Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_TMAP_NUM, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_NUM, Xgl_usgn32, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_TMAP_NUM, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_NUM, Xgl_usgn32 *);
```

DESCRIPTION

These attributes set the number of Texture Map objects in a 3D Context. When an application sets these attributes, XGL changes the number of Texture Map objects and their corresponding on and off switches to match these attributes. If the value of these attributes is *m*, the lists of Texture Map objects given by **XGL_3D_CTX_SURF_BACK_TMAP**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP**(3) and their lists of switches as given by **XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES**(3) and **XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES**(3) are indexed from 0 to *m*-1. If the new number *n* exceeds the current number *m*, XGL creates *n*-*m* Texture Map objects and switches and appends these to the appropriate lists in the Context at index positions *m* to *n*-1.

The texture map switches are on by default. The Texture Map objects and switches that existed before increasing the number of Texture Map objects remain unchanged. If the number is less than the current number, XGL destroys the Texture Map objects and switches with indices *n* to *m*-1 (provided that no other 3D Context is using them).

The default value for each attribute is 0.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_SURF_BACK_TMAP(3)
XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES(3)
XGL_3D_CTX_SURF_FRONT_TMAP(3)
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES(3)

NOTES

Texturing is enabled when these attributes are non-zero and the 3D surface primitive has data in its vertex information.

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

NAME

XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES – specify the list of switches for a 3D Context's Texture Map objects

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES, Xgl_boolean [], 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES, Xgl_boolean [], 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES, Xgl_boolean []);
xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES, Xgl_boolean []);
```

DESCRIPTION

These attributes specify an array of on and off switches corresponding to the 3D Context's array of Texture Map objects, given by **XGL_3D_CTX_SURF_FRONT_TMAP**(3) and **XGL_3D_CTX_SURF_BACK_TMAP**(3). The number of switches in the list is given by **XGL_3D_CTX_SURF_FRONT_TMAP_NUM**(3) and **XGL_3D_CTX_SURF_BACK_TMAP_NUM**(3).

The Context's array of *textures* have indices 0 to *n*-1, where *n* is the value of **XGL_3D_CTX_SURF_FRONT_TMAP_NUM**(3) and **XGL_3D_CTX_SURF_BACK_TMAP_NUM**(3). An application can turn on texture *i* by setting flag *i* in its own array of switches to TRUE. Similarly, FALSE turns a *texture* off. After establishing the values of all switches, an application can set these attributes with this array to turn *textures* on or off.

The default value for each attribute is NULL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_BACK_TMAP(3)
XGL_3D_CTX_SURF_BACK_TMAP_NUM(3)
XGL_3D_CTX_SURF_FRONT_TMAP(3)
XGL_3D_CTX_SURF_FRONT_TMAP_NUM(3)

NAME

XGL_3D_CTX_SURF_FRONT_TRANSP,
XGL_3D_CTX_SURF_BACK_TRANSP – specify the transparency value for surface primitives

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TRANSP, float, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_TRANSP, float, 0);

xgl_object_get(ctx, XGL_3D_CTX_SURF_FRONT_TRANSP, float *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_TRANSP, float *);
```

DESCRIPTION

These attributes specify the transparency value of a surface primitive. The values can range from 0.0 to 1.0; a value of 0.0 means the surface is not transparent (i.e. it is opaque); a value of 1.0 means the surface is fully transparent and is not visible. A transparent surface is still subject to the **XGL_3D_CTX_HLHSR_MODE**(3) attribute.

Front-facing surfaces use XGL_3D_CTX_SURF_FRONT_TRANSP and back-facing surfaces use XGL_3D_CTX_SURF_BACK_TRANSP.

The attribute **XGL_3D_CTX_SURF_TRANSP_METHOD**(3) controls how transparent surfaces are rendered. The following attributes affect blended transparent surfaces:

XGL_3D_CTX_SURF_TRANSP_BLEND_EQ
: Controls what equation is used to blend the transparent pixel value with the current pixel value.

XGL_3D_CTX_BLEND_DRAW_MODE
: Controls if blended primitives are drawn.

XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER
: Controls if the Z buffer is updated.

The default value for both attributes is 0.0.

These attributes are pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_BLEND_DRAW_MODE(3)
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER(3)
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ(3)
XGL_3D_CTX_SURF_TRANSP_METHOD(3)

NAME

XGL_3D_CTX_SURF_GEOM_NORMAL – controls how surface normals are calculated, in a 3D Context, when they are not explicitly given in the application's data

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_GEOM_NORMAL, Xgl_geom_normal, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_GEOM_NORMAL, Xgl_geom_normal *);
```

DESCRIPTION

This attribute controls how the facet normal for a surface is calculated within the Context *ctx* if the facet normal is not provided as part of the application's data. The method for determining the facet normal depends on the XGL surface primitive type as shown below. The alternatives for this attribute are:

XGL_GEOM_NORMAL_FIRST_POINTS

For all surface primitives except **xgl_triangle_strip**(3)**, xgl_triangle_list**(3)**,** and **xgl_-quadrilateral_mesh**(3) –

The facet normal is computed by selecting three points: A, B, and C. Point A is the first point in the XGL point list defining the vertices for the surface. Point B is the next point from the list that is non-coincident with A. Point C is the next point after B that is non-colinear with A and B. The geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For xgl_triangle_strip –

The facet normal of each triangle within the strip is the normalized cross product of vectors along two of its sides. The vectors used are different for even numbered and odd numbered triangles (The triangles are implicitly numbered by their position within the strip. The first triangle within the strip is triangle number 1). If the three consecutive points defining a triangle are called A, B, and C, then the vectors are selected as follows:

For odd numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For even numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to C with the vector extending from A to B.

For xgl_triangle_list –

There are three cases to consider where a triangle list is arranged as a triangle strip, triangle star, and set of independent triangles.

1. In the triangle strip case:

   The facet normal of each triangle within the strip is the normalized cross product of vectors along two of its sides. The vectors used are different for even numbered and odd numbered triangles (The triangles are implicitly numbered by their position within the strip. The first triangle within the strip is triangle number 1). If the three consecutive points defining a triangle are called A, B, and C, then the vectors are selected as follows:

For odd numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For even numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to C with the vector extending from A to B.

2. In the triangle star and set of independent triangles cases:

   The facet normal of each triangle within the list is the normalized cross product of vectors along two of its sides. Call the three consecutive points defining a triangle A, B, and C. In the triangle star case, point A may not be contiguous to point B in the point list, instead it may be a holdover from farther back in the point list. In this case, point A is still considered to be the first point making up the triangle. The vectors used to compute the geometric normal are selected as follows:

   The geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For xgl_quadrilateral_mesh –

The facet normal of each quadrilateral within the mesh is the normalized cross product of its two diagonal vectors. If the two consecutive points in the first row making up the ''top'' of the quadrilateral are called A and B, and the two consecutive points in the next row that makes up its ''bottom'' are called C and D, then the normal is computed as follows:

The geometric normal is the normalized cross product of the vector extending from A to D with the vector extending from C to B.

XGL_GEOM_NORMAL_LAST_POINTS

For all surface primitives except **xgl_triangle_strip**(3)**, xgl_triangle_list**(3)**,** and **xgl_-quadrilateral_mesh**(3) –

The facet normal is computed by selecting three points: A, B, and C. Point A is the last point in the XGL point list defining the vertices for the surface. Point B is the next point from the list, working backwards from the last point, that is non-coincident with A. Point C is the next point before B, again working backwards, that is non-colinear with A and B. The geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For xgl_triangle_strip –

The facet normal of each triangle within the strip is the normalized cross product of vectors along two of its sides. The vectors used are different for even numbered and odd numbered triangles (The triangles are implicitly numbered by their position within the strip. The first triangle within the strip is triangle number 1). If the three consecutive points defining a triangle are called A, B, and C (beginning at the first point comprising the triangle), then the vectors are selected as follows:

For odd numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to C with the vector extending from A to B.

For even numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For xgl_triangle_list –

There are three cases to consider where a triangle list is arranged as a triangle strip, triangle star, and set of independent triangles.

1. In the triangle strip case:

   The facet normal of each triangle within the strip is the normalized cross product of vectors along two of its sides. The vectors used are different for even numbered and odd numbered triangles (The triangles are implicitly numbered by their position within the strip. The first triangle within the strip is triangle number 1). If the three consecutive points defining a triangle are called A, B, and C (beginning at the first point comprising the triangle), then the vectors are selected as follows:

   For odd numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to C with the vector extending from A to B.

   For even numbered triangles, the geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

2. In the triangle star and set of independent triangles cases:

   The facet normal of each triangle within the list is the normalized cross product of vectors along two of its sides. Call the three consecutive points defining a triangle A, B, and C. In the triangle star case, point A may not be contiguous to point B in the point list, instead it may be a holdover from farther back in the point list. In this case, point A is still considered to be the first point making up the triangle. The vectors used to compute the geometric normal are selected as follows:

   The geometric normal is the normalized cross product of the vector extending from A to B with the vector extending from A to C.

For xgl_quadrilateral_mesh –

The facet normal of each quadrilateral within the mesh is the normalized cross product of its two diagonal vectors. If the two consecutive points in the first row making up the ‘‘top’’ of the quadrilateral are called A and B, and the two consecutive points in the next row that makes up its ‘‘bottom’’ are called C and D, then the normal is computed as follows:

The geometric normal is the normalized cross product of the vector extending from C to B with the vector extending from A to D.

The default value is XGL_GEOM_NORMAL_FIRST_POINTS.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)
xgl_triangle_strip(3)

NAME

XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP – specifies how surface normals are to be treated for lighting

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP, Xgl_boolean*);
```

DESCRIPTION

This attribute defines how surface normals are to be treated for lighting. When set to TRUE, XGL will always use a front facing normal to light polygons. If a polygon is back facing, then the normal will be multiplied by -1 (flipped) so that the normal is front facing. If lighting is per facet, then the facet normal will be flipped. If vertex lighting is being performed than the vertex normals will be flipped. The flipping of the normal is for the lighting calculation only and does not affect face determination or face culling.

When this attribute is set to FALSE, the normal is used unchanged, regardless of the polygon's facing.

The default value is TRUE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NAME

XGL_3D_CTX_SURF_NORMAL_FLIP – specifies whether vertex and facet normals are flipped

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_NORMAL_FLIP, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_NORMAL_FLIP, Xgl_boolean *);
```

DESCRIPTION

This attribute can be set to either TRUE or FALSE. When TRUE, it will cause all the user supplied normals specified as facet or vertex data to be flipped 180 degrees. This is useful if the application's data has normals that point toward the back face instead of toward the front face, which is XGL's convention.

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTES

This attribute applies to the normal when it comes from the vertex or facet and not when XGL calculates it.

NAME

XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG – controls the drawing of silhouette edges around a surface

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_3d_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether silhouette edges are drawn around a surface. When solid objects are modeled with surfaces, silhouette edges show the horizons where the composite surface changes from front-facing to back-facing. Silhouette edges are not drawn when the side of a polygon corresponds to a clipping boundary.

The algorithm for detecting the presence of silhouette edges actually finds a close approximation to the actual silhouette edges. Vertex normal vectors must be supplied with the geometric surface data. The algorithm requires surfaces to be decomposed into triangles. For some primitives, XGL may have to execute this decomposition, render the surface and any silhouette edges, and finally dispose to the decomposed surface representation. The decomposition process may be computationally intensive, so it behooves the application to store the decomposed surface representation in a Geometry Cache (see **XGL_GCACHE**(3)) when the surface is needed for more than one frame.

XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3) ignores any local edge flag that may be stored in the vertices of the surface geometry. This attribute is also independent of XGL_-CTX_SURF_EDGE_FLAG. Although an application can display both silhouette edges and highlighting edges simultaneously, a user probably would prefer to see only one type at any given time.

All the attributes that apply to highlighted edges for color, width, pattern, and depth-cueing also apply to silhouette edges, as described in **XGL_CTX_SURF_EDGE_FLAG**(3)**.**

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)

XGL_CTX_SURF_FRONT_COLOR(3)
XGL_CTX_SURF_EDGE_FLAG(3)
XGL_GCACHE(3)

NAME

XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION – specifies the method used to compute texture coordinate values

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION,
        Xgl_texture_persp_correction, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION,
        Xgl_texture_persp_correction *);
```

DESCRIPTION

This attribute specifies the method used to compute texture coordinate values in surfact interiors when a textured surface primitive is rendered. Possible values for this attribute are:

XGL_TEXTURE_PERSP_NONE
: Texture mapping coordinates are linearly interpolated without perspective correction.

XGL_TEXTURE_PERSP_PIXEL
: As the texture mapping coordinates are interpolated, their values are manipulated to account for a perspective projection.

The default value is XGL_TEXTURE_PERSP_PIXEL.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_3D_CTX_SURF_TRANSP_BLEND_EQ – controls how transparent surfaces are rendered

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx         ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_TRANSP_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_TRANSP_BLEND_EQ, Xgl_blend_eq *);
```

DESCRIPTION

This attribute controls how the pixels values in transparent surfaces are blended with the existing pixel values in the Device when **XGL_3D_CTX_SURF_TRANSP_METHOD**(3) is XGL_TRANSP_BLENDED.

The type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that transparency use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_ADD_BG.

The default value is XGL_BLEND_NONE.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_TRANSP_METHOD(3)
XGL_CTX_SURF_AA_BLEND_EQ(3)

NOTES

If both **XGL_3D_CTX_SURF_TRANSP_BLEND_EQ**(3) and **XGL_CTX_SURF_AA_BLEND_EQ**(3) are other than XGL_BLEND_NONE, and a hollow antialiased transparent surface is being drawn, then the blending equation used will be the value of either the **XGL_3D_CTX_SURF_-TRANSP_BLEND_EQ**(3) or the **XGL_CTX_SURF_AA_BLEND_EQ**(3) attribute, and the choice is device dependent.

NAME

XGL_3D_CTX_SURF_TRANSP_METHOD – controls how transparent surfaces are rendered

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_SURF_TRANSP_METHOD, Xgl_transp_method, 0);
xgl_object_get(ctx, XGL_3D_CTX_SURF_TRANSP_METHOD, Xgl_transp_method *);
```

DESCRIPTION

This attribute controls how transparent surfaces are rendered. A transparent surface is one whose transparency value (as specified by **XGL_3D_CTX_SURF_FRONT_TRANSP**(3) or **XGL_3D_CTX_SURF_BACK_TRANSP**(3)) is greater than 0.0. The data type *Xgl_transp_method* takes on values:

XGL_TRANSP_NONE
: No transparency calculations are done, even if the surface's transparency value is greater than 0.0.

XGL_TRANSP_SCREEN_DOOR
: Transparent surfaces are rendered through a device-dependent stipple mask. This mask allows some pixels of the transparent surface to be written while it restricts others. Where the transparent surface pixels are restricted, the existing background pixels are left untouched and appear inside the surface.

  The human visual system then blends the pixels from the surface and the background together and the result is that the surface looks, somewhat, transparent. This method is so-named because it is similar to looking through a screen door or a window screen. This method is less computationally expensive than XGL_TRANSP_BLENDED and is a viable option in situations where performance is more important than image quality.

XGL_TRANSP_BLENDED
: Each value pixel of the transparent surface is blended with the XGL_CTX_BACKGROUND_COLOR or with the existing value pixel to determine the final pixel value. The equation used for blending is controlled by the **XGL_3D_CTX_SURF_TRANSP_BLEND_EQ**(3) attribute. The attribute **XGL_3D_CTX_BLEND_DRAW_MODE**(3) controls how blended surfaces are drawn. The attribute **XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER**(3) controls if the Z Buffer is updated.

The default value is XGL_TRANSP_NONE.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_FRONT_TRANSP(3)
XGL_3D_CTX_SURF_BACK_TRANSP(3)
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ(3)

XGL_3D_CTX_BLEND_DRAW_MODE(3)
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER(3)

NAME

XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY – controls clipping in homogeneous space

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY, Xgl_boolean *);
```

DESCRIPTION

It is possible for the application to specify a 3D Transform that maps geometry in homogeneous space to be in front of and behind the observer. In this case, the homogeneous coordinates can have *w* values that are both positive and negative. **XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY**(3) specifies how the clipping of geometry with positive and negative values of *w* is handled. It is a boolean attribute defined as follows:

FALSE The 3D clipping process checks for both positive *w* and negative *w*. This means that primitives are clipped twice: once for each region. For example, a line segment that has one endpoint in positive *w* and the other in negative *w* is split into two line segments.

TRUE Only clipping to positive *w* is done. This value produces incomplete pictures mathematically speaking, but corresponds to the physical reality.

Since clipping to both positive *w* and negative *w* is expensive and in most cases unnecessary, the default value is TRUE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTE

The value of this attribute may have significant impact on performance. To verify the best setting for the platform used, refer to your accelerator guide.

NAME

XGL_3D_CTX_Z_BUFFER_COMP_METHOD – specifies the Z comparison method to be used when updating current buffer

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_3D_CTX_Z_BUFFER_COMP_METHOD, Xgl_z_comp_method, 0);
xgl_object_get(ctx, XGL_3D_CTX_Z_BUFFER_COMP_METHOD, Xgl_z_comp_method *);
```

DESCRIPTION

This attribute defines the comparison technique to be used when operating on the Z buffer. It specifies the function used to compare the incoming Z value with the Z value present in the Z buffer. The result of the comparison determines how the destination buffer is updated.

When rendering into an image buffer, this attribute is only applicable when **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER. If the result of the Z comparison function is TRUE then the incoming pixel color is written to the image buffer.

Whether the Z buffer is updated or not is dependent on the value of **XGL_3D_CTX_Z_BUFFER_WRITE_MASK**(3)**.** The data type *Xgl_z_comp_method* takes on values:

Zin Incoming Z value

Zcur Z value currently in the Z buffer

Zcomp The result of the comparison. The destination buffer is updated if zcomp is TRUE.

XGL_Z_COMP_LESS_THAN
Zcomp = Zin < Zcur

XGL_Z_COMP_LESS_THAN_OR_EQUAL
Zcomp = Zin ≤ Zcur

XGL_Z_COMP_EQUAL
Zcomp = Zin == Zcur

XGL_Z_COMP_GREATER_THAN
Zcomp = Zin > Zcur

XGL_Z_COMP_GREATER_THAN_OR_EQUAL
Zcomp = Zin ≥ Zcur

XGL_Z_COMP_NOT_EQUAL
Zcomp = Zin != Zcur

XGL_Z_COMP_NEVER
Zcomp = FALSE

XGL_Z_COMP_ALWAYS
Zcomp = TRUE

The default value is XGL_ZCOMP_LESS_THAN_OR_EQUAL.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_new_frame(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)

NOTES

xgl_context_new_frame(3) is not affected by this attribute.

NAME

XGL_3D_CTX_Z_BUFFER_WRITE_MASK – defines which bit planes of the Z buffer are written

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_3D_CTX_Z_BUFFER_WRITE_MASK, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_3D_CTX_Z_BUFFER_WRITE_MASK, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the enable mask for writing to the Z buffer bit planes. Each bit in the mask corresponds to a bit plane of the Z buffer. In the current release, the only supported values are **0** and **0xffffffff**. A value of **0** implies writing into the Z buffer is disabled. A value of **0xffffffff** implies writing into the Z buffer is enabled.

When rendering into the image buffer, this attribute is used in conjunction with **XGL_3D_CTX_Z_BUFFER_COMP_METHOD**(3) and **XGL_3D_CTX_HLHSR_MODE**(3).

The default value is **0xffffffff**.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)

NAME

XGL_CGM – a CGM Metafile format

OVERVIEW

XGL_CGM is the type of a format representing a Computer Graphics Metafile (CGM) output device. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

XGL graphics are rendered onto a Device object. This Device object outputs a CGM format. The CGM is a Unix file, written in a standard format, for representing and exchanging 2D images. XGL currently outputs Version 1 of the CGM as defined by ISO 8632:1985.

The contents of the Metafile are written to a stream (that is, a FILE *). The application can indicate which file is associated with the output stream; the default is standard output. Since the CGM device is implemented as a stream pipeline, the name of the stream pipeline must be placed in the *desc* parameter. See the *XGL Programmer's Guide* for example code.

After creating the Metafile Device, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

The following is a brief description of the attributes associated with CGM Metafile format. These attributes are defined in the file "xgl_cgm-2.0.h" which must be included in any application using the CGM format. For more information on how to create a CGM device, see the man page for each attribute and the *XGL Programmer's Guide* for example code.

XGL_CGM_DESCRIPTION(3)

Specifies a structure that contains the output file pointer; a string containing the Metafile signature string; and the descriptor string, which describes the Metafile conformance level.

XGL_CGM_ENCODING(3)

Specifies the output encoding used to write the Metafile. The three alternatives are as follows:

XGL_CGM_CLEAR_TEXT
XGL_CGM_CHARACTER
XGL_CGM_BINARY

XGL_CGM_PICTURE_DESCRIPTION(3)

Specifies a string to describe the picture being output to the Metafile.

XGL_CGM_SCALE_FACTOR(3)

Specifies the mapping of one unit of VDC space to one millimeter of screen space. The value can be any floating point value. It is used only when the XGL_CGM_-SCALE_MODE is metric.

XGL_CGM_SCALE_MODE(3)

Specifies the type of scaling mode in the Metafile. The two alternatives are:

XGL_CGM_ABSTRACT
XGL_CGM_METRIC

XGL_CGM_TYPE(3)
Specifies which CGM version the Metafile conforms to. The only supported value for type is XGL_METAFILE_CGM_V1.

XGL_CGM_VDC_EXTENT(3)
Specifies the type and range of coordinate values in the Metafile. The three alternatives are as follows:

XGL_CGM_VDC_EXT_FLOAT
XGL_CGM_VDC_EXT_SHORT
XGL_CGM_VDC_EXT_LONG

The first value, XGL_CGM_VDC_EXT_FLOAT, is not currently implemented.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

XGL_STREAM(3)
This is the type of an object representing a Stream.

NAME

XGL_CGM_DESCRIPTION – describes the Metafile headers and output file

SYNOPSIS

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

Xgl_cgm_description    cgm;

xgl_object_set(cgm, XGL_CGM_DESCRIPTION, Xgl_cgm_description, 0);
xgl_object_get(cgm, XGL_CGM_DESCRIPTION, Xgl_cgm_description *);
```

DESCRIPTION

This attribute defines the Metafile header information and the file pointer to the Metafile output file. It can only be set prior to the writing to disk of any part of the Metafile. After that, this attribute becomes read-only.

The Metafile description information is defined by the data type *Xgl_cgm_description*:

```
typedef struct {
        FILE    *cgm_fp;
        char    *cgm_signature;
        char    *cgm_description;
} Xgl_cgm_description;
```

The three structure components are defined as follows:

cgm_fp Names the file where the Metafile should be written. If the file already exists, XGL appends the Metafile to it.

If XGL is given a structure with the file pointer equal to NULL, the default value is: stdout.

cgm_signature

A string containing the Metafile label. This label could include a description of the Metafile's contents.

If XGL is given a structure with the signature equal to NULL, the default value is: XGL <current release number> CGM 1.0 date:time:year.

cgm_description

A string containing descriptive information about the Metafile. This could include the author of the Metafile, the place of origin, and date and time of generation.

If XGL is given a structure with the description equal to NULL, the default value is: Sun Microsystems CGM TOP/FULL conformance.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NOTES

This attribute can only be set prior to the writing to disk of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CGM_DEV – a CGM Metafile Device object

OVERVIEW

XGL_CGM_DEV is the type of an object representing a Computer Graphics Metafile (CGM) output device. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

XGL graphics are rendered onto a Device object. This Device object outputs a CGM. The CGM is a Unix file, written in a standard format, for representing and exchanging 2D images. XGL currently outputs Version 1 of the CGM as defined by ISO 8632:1985.

The contents of the Metafile are written to a stream (that is, a FILE *). The application can indicate which file is associated with the output stream; the default is standard output. Since the CGM device is implemented as a stream pipeline, the name of the stream pipeline must be placed in the desc parameter. See the *XGL Programmer's Guide* for example code.

After creating the Metafile Device, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

The following is a brief description of the attributes associated with CGM Metafile format. For more information, see the man page for each attribute.

XGL_CGM_DESCRIPTION(3)
: Specifies a structure that contains the output file pointer; a string containing the Metafile signature string; and the descriptor string, which describes the Metafile conformance level.

XGL_CGM_ENCODING(3)
: Specifies the output encoding used to write the Metafile. The three alternatives are as follows:

  XGL_CGM_CLEAR_TEXT
  XGL_CGM_CHARACTER
  XGL_CGM_BINARY

XGL_CGM_PICTURE_DESCRIPTION(3)
: Specifies a string to describe the picture being output to the Metafile.

XGL_CGM_SCALE_FACTOR(3)
: Specifies the mapping of one unit of VDC space to one millimeter of screen space. The value can be any floating point value. It is used only when the XGL_CGM_-SCALE_MODE is metric.

XGL_CGM_SCALE_MODE(3)
: Specifies the type of scaling mode in the Metafile. The two alternatives are:

  XGL_CGM_ABSTRACT
  XGL_CGM_METRIC

XGL_CGM_TYPE(3)
: Specifies which CGM version the Metafile conforms to. The only supported value for type is XGL_METAFILE_CGM_V1.

XGL_CGM_VDC_EXTENT(3)
: Specifies the type and range of coordinate values in the Metafile. The three

alternatives are as follows:

XGL_CGM_VDC_EXT_FLOAT
XGL_CGM_VDC_EXT_SHORT
XGL_CGM_VDC_EXT_LONG

The first value, XGL_CGM_VDC_EXT_FLOAT, is not currently implemented.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

XGL_STREAM(3)
This is the type of an object representing a Stream.

NAME

XGL_CGM_ENCODING – the encoding used on the contents of the Metafile

SYNOPSIS

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

Xgl_cgm         cgm;

xgl_object_set(cgm, XGL_CGM_ENCODING, Xgl_cgm_encoding, 0);
xgl_object_get(cgm, XGL_CGM_ENCODING, Xgl_cgm_encoding *);
```

DESCRIPTION

This attribute defines the data representation used to store graphical objects in an XGL Metafile. It can only be set before any part of the Metafile is written to the file stream. After that, this attribute becomes read-only.

Three encoding formats are currently supported in XGL as defined by the enumerated data type *Xgl_cgm_encoding*, conforming to the CGM standard.

The three encoding styles are defined as follows:

XGL_CGM_CLEAR_TEXT
: This encoding uses only the 94 printing characters from the ISO 646 7-bit code table to represent data. This encoding is designed to be ‘‘human readable’’ and self-delimiting. Parsing and generating clear text encoding is slower and more complex than binary encoding.

XGL_CGM_CHARACTER
: This encoding uses ASCII bytes, including control characters, to represent element codes and parameters. It is self-delimiting and more compact than clear text encoding. It is designed for transmission over communication lines. It is slower to parse and generate than binary encoding.

XGL_CGM_BINARY
: This encoding outputs CGM elements in binary format, as described in ANSI X3.122-1986. This format is the most compact, and it is the fastest to read and write. It outputs elements in 16-bit blocks, and every bit is significant.

The default value is: XGL_CGM_CLEAR_TEXT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NOTES

This attribute can only be set prior to the writing to the file stream of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CGM_PICTURE_DESCRIPTION – picture label within a Metafile

SYNOPSIS

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

Xgl_cgm         cgm;

xgl_object_set(cgm, XGL_CGM_PICTURE_DESCRIPTION, char *, 0);
xgl_object_get(cgm, XGL_CGM_PICTURE_DESCRIPTION, char []);
```

DESCRIPTION

This attribute defines the descriptive label for a picture within an XGL Metafile. It can only be set prior to the writing to the file stream of any part of the Metafile. After that, the attribute becomes read-only. XGL currently supports only one picture per Metafile.

The string that defines the picture name is output to the Metafile with the BEGIN PICTURE delimiter in the Metafile. This delimiter is written (and a new picture is started) when the user calls **xgl_context_new_frame**(3)**.** Any changes to the picture name attribute after this call will not affect the current picture.

If no picture description is given, XGL uses the last picture name given.

If no picture name was ever given, the default picture name is *xgl_cgm_picture*.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CGM_DESCRIPTION(3)
XGL_CGM_TYPE(3)

NOTES

This attribute can only be set prior to the writing to the file stream of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CGM_SCALE_FACTOR – defines the scale factor if the Metafile is using Metric Scaling

SYNOPSIS

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

Xgl_cgm         cgm;

xgl_object_set(cgm, XGL_CGM_SCALE_FACTOR, float, 0);
xgl_object_get(cgm, XGL_CGM_SCALE_FACTOR, float *);
```

DESCRIPTION

This attribute defines the scale factor used when the SCALEMODE delimiter is set to METRIC. It can only be set prior to the writing to the file stream of any part of the Metafile. After that, the attribute becomes read-only.

The value is the address of a floating point number, defined as follows:

A scale factor used to convert the coordinates in the Metafile to the coordinates used to display the image on the screen. The scale factor sets the ratio between one unit in the Metafile coordinate values and one millimeter on the screen display.

The default value is 1.0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NOTES

This attribute can only be set prior to the writing to the file stream of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CGM_SCALE_MODE – defines the scaling mode used in the contents of the Metafile

SYNOPSIS

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

Xgl_cgm         cgm;

xgl_object_set(cgm, XGL_CGM_SCALE_MODE, Xgl_cgm_scale_mode, 0);
xgl_object_get(cgm, XGL_CGM_SCALE_MODE, Xgl_cgm_scale_mode *);
```

DESCRIPTION

This attribute defines the data representation used to store graphical objects in an XGL Metafile. It can only be set prior to the writing to the file stream of any part of the Metafile. After that, the attribute becomes read-only.

Two encoding formats are currently supported in XGL as defined by the enumerated data type *Xgl_cgm_scale_mode*, conforming to the CGM standard.

The two encoding styles are defined as follows:

XGL_CGM_ABSTRACT

This value means there is no correspondence between the range of coordinate values in the Metafile and the image that Metafile produces. In the Metafile, the SCALEMODE will be ABSTRACT, and the scale factor will always be 1.0.

XGL_CGM_METRIC

This value means there is a correspondence between the range of coordinate values in the Metafile and the image that Metafile produces. In the Metafile, the SCALEMODE will be METRIC, and the scale factor will indicate the mapping between one unit in the Metafile coordinate values and one millimeter in the displayed image. This mode allows images of predetermined sizes to be exchanged using CGM. The scale factor is set by **XGL_CGM_SCALE_FACTOR**(3)**.**

The default value is XGL_CGM_ABSTRACT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_CGM_SCALE_FACTOR(3)

NOTES

This attribute can only be set prior to the writing to the file stream of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CGM_TYPE – defines the type of an XGL Metafile object

SYNOPSIS

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

Xgl_cgm_type   type;

xgl_object_get(cgm, XGL_CGM_TYPE, Xgl_cgm_type *);
```

DESCRIPTION

This read-only attribute defines the type associated with an XGL object. It can be used to inquire about the object type when only a generic Xgl_cgm handle to that object is known. XGL currently defines the following values for an object type Xgl_obj_type:

XGL_METAFILE_CGM_V1
: A Metafile object that conforms to a version 1 CGM Metafile (ISO 8632:1985 standard).

The Metafile type is associated by XGL at the creation of the object. It cannot be modified later. Users cannot define their own objects.

SEE ALSO

xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_CGM_VDC_EXTENT – defines the range and type of allowed coordinate values in the Metafile

SYNOPSIS

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

Xgl_cgm         cgm;

xgl_object_set(cgm, XGL_CGM_VDC_EXTENT, Xgl_cgm_vdc_extent, 0);
xgl_object_get(cgm, XGL_CGM_VDC_EXTENT, Xgl_cgm_vdc_extent *);
```

DESCRIPTION

This attribute defines the range and type of coordinate values used to store graphical objects in an XGL Metafile. It can only be set prior to the writing to the file stream of any part of the Metafile. After that, the attribute becomes read-only.

Three encoding formats are currently supported in XGL as defined by the enumerated data type *Xgl_cgm_vdc_extent*.

The three extent types are defined as follows:

XGL_CGM_VDC_EXT_FLOAT

Note: This extent is currently not implemented. It may be supported in future releases of XGL.

All values in this extent are floating point. The values range from 0.0 to 1.0. If this extent is selected, the VDCTYPE output to the Metafile will be of type REAL.

XGL_CGM_VDC_EXT_SHORT

All values in this extent are 16 bit integers. The values range from 0 to 2^{16}–1. If this extent is selected, the VDCTYPE output to the Metafile will be of type INTEGER. With this extent, there is no danger of integer overflow.

XGL_CGM_VDC_EXT_LONG

All values in this extent are 32 bit integers. The values range from 0 to 2^{32}–1. If this extent is selected, the VDCTYPE output to the Metafile will be of type INTEGER. Although this extent has a larger range of values, there is a possibility of integer overflow.

The default value is XGL_CGM_VDC_EXT_SHORT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NOTES

This attribute can only be set prior to the writing to the file stream of any part of the Metafile. It is read-only thereafter.

NAME

XGL_CMAP – a Color Map object

OVERVIEW

XGL_CMAP is the type of an object representing a Color Map object. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

An XGL Color Map object provides the facility to convert colors from one color type to another. Once the Color Map has been created, to be useful it must be attached to a Raster. This is done explicitly by the application by setting the **XGL_DEV_COLOR_MAP**(3) attribute in the Raster.

After creating the Color Map, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

The following is a brief description of the attributes associated with Color Map objects. For more information, see the man page for each attribute.

XGL_CMAP_COLOR_CUBE_SIZE(3)
: The size of the color cube associated with a Color Map object when RGB-to-Index color conversions are required.

XGL_CMAP_COLOR_MAPPER(3)
: The function that maps color values from XGL_COLOR_INDEX to XGL_COLOR_RGB.

XGL_CMAP_COLOR_TABLE(3)
: The structure that includes an array of color values providing a mapping from XGL_COLOR_INDEX colors to XGL_COLOR_RGB colors.

XGL_CMAP_COLOR_TABLE_SIZE(3)
: The number of elements in the array associated with a color table array.

XGL_CMAP_DITHER_MASK_N(3)
XGL_CMAP_DITHER_MASK_X
XGL_CMAP_DITHER_MASK_Y
: The number of columns and rows, respectively, in the dither mask used for RGB-to-Index conversions.

XGL_CMAP_DITHER_MASK(3)
: The dithering array used for RGB-to-Index color conversions.

XGL_CMAP_INVERSE_COLOR_MAPPER(3)
: The function that maps color values from XGL_COLOR_RGB to XGL_COLOR_INDEX.

XGL_CMAP_MAX_COLOR_TABLE_SIZE(3)
: The maximum number of elements that can be stored in a color table array.

XGL_CMAP_NAME(3)
The name given to a window-system-generated Color Map. It allows for the sharing of window-system resources.

XGL_CMAP_RAMP_LIST(3)
An array of offset and length pairs describing color ramps embedded within a color table.

XGL_CMAP_RAMP_NUM(3)
Number of color ramps within a color table.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

When a Color Map is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

A System State object must have been created with the **xgl_open**(3) operator before a Color Map can be created.

NAME

XGL_CMAP_COLOR_CUBE_SIZE – defines the size of the color cube associated with a Color Map object when RGB-to-Index color conversions are required

SYNOPSIS

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

Xgl_cmap          cmap;

xgl_object_set(cmap, XGL_CMAP_COLOR_CUBE_SIZE, Xgl_usgn32 *, 0);
xgl_object_get(cmap, XGL_CMAP_COLOR_CUBE_SIZE, Xgl_usgn32 *);
```

DESCRIPTION

The color table associated with an XGL Color Map object may contain a color cube for converting RGB colors to indexed colors. An RGB-to-Index conversion is required if the XGL Device color type (see **XGL_DEV_COLOR_TYPE**(3)) is XGL_COLOR_RGB and the underlying hardware is indexed. It is used only in this RGB-to-Index conversion case.

This attribute specifies the size of the color cube. It is formatted as an array of three unsigned integers that specify the size of the red, green, and blue axes of the color cube, respectively. It can be used instead of the three separate size attributes (see **XGL_CMAP_COLOR_CUBE_RSIZE**(3)**, XGL_CMAP_COLOR_CUBE_GSIZE**(3)**,** and **XGL_CMAP_COLOR_CUBE_BSIZE**(3)**)** to set or to get simultaneously all three components of the color cube size.

The defaults are:

red = 6
green = 9
blue = 4

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CMAP_COLOR_CUBE_BSIZE(3)
XGL_CMAP_COLOR_CUBE_GSIZE(3)
XGL_CMAP_COLOR_CUBE_RSIZE(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_CMAP_COLOR_MAPPER,
XGL_CMAP_INVERSE_COLOR_MAPPER – define functions used to map colors from one color space to another

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_cmap            cmap;
xgl_object_set(cmap, XGL_CMAP_COLOR_MAPPER, void (*)(), 0);
xgl_object_set(cmap, XGL_CMAP_INVERSE_COLOR_MAPPER, void (*)(), 0);

xgl_object_get(cmap, XGL_CMAP_COLOR_MAPPER, void (*(*))());
xgl_object_get(cmap, XGL_CMAP_INVERSE_COLOR_MAPPER, void (*(*))());
```

DESCRIPTION

XGL_CMAP_COLOR_MAPPER specifies a function that maps the color values from the device color space set by the application (see **XGL_DEV_COLOR_TYPE**(3)) to the underlying real color space (see **XGL_DEV_REAL_COLOR_TYPE**(3)).

XGL_CMAP_INVERSE_COLOR_MAPPER specifies a function that maps the color values from the underlying real color space (see **XGL_DEV_REAL_COLOR_TYPE**(3)) to the device color space set by the application (see **XGL_DEV_COLOR_TYPE**(3)).

Conceptually, these functions are visible to the application programmer to provide flexibility when manipulating colors. However, in the current XGL release, most typical color operations can be handled without changing these two attributes.

If you wish to write your own mapper function, XGL will call your function with the parameters shown below. Notice that your function should take as parameters a Color Map object, a pointer to a color structure, a pointer to an unsigned long, and a boolean indicating if the function should use the pixel mapping array (if relevant). The unsigned long represents most often a pixel value. The only exception is when the pixel mapping array is not used (boolean set to FALSE) and **XGL_DEV_REAL_COLOR_TYPE**(3) is index; in this case the *value* represents the index in the color table of the Color Map. The functional interface to the color map function is as follows:

```
void
mapper(cmap, color, value, use_pixel_mapping)
Xgl_cmap            cmap;
Xgl_color           *color;
Xgl_usgn32          *value;
Xgl_boolean         use_pixel_mapping;
```

In the case of XGL_CMAP_COLOR_MAPPER, *color* is the input and *value* is the output. In the case of XGL_CMAP_INVERSE_COLOR_MAPPER, *value* is the input and *color* is the output.

The default values are functions defined in XGL that will provide the appropriate behavior.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_DEV_COLOR_TYPE(3)
XGL_DEV_REAL_COLOR_TYPE(3)

NAME

XGL_CMAP_COLOR_TABLE – the color table associated with a Color Map object

SYNOPSIS

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

Xgl_cmap          cmap;

xgl_object_set(cmap, XGL_CMAP_COLOR_TABLE, Xgl_color_list *, 0);
xgl_object_get(cmap, XGL_CMAP_COLOR_TABLE, Xgl_color_list *);
```

DESCRIPTION

A color table provides a way of mapping colors from one color space to another. This attribute specifies the color table that allows color mapping from XGL_COLOR_INDEX to XGL_COLOR_RGB.

XGL_CMAP_COLOR_TABLE is a pointer to a C structure of type *Xgl_color_list,* which is defined as follows:

```
typedef struct {
        Xgl_usgn32    start_index;    /* starting index for table */
        Xgl_usgn32    length;         /* number of entries in table */
        Xgl_color     *colors;        /* color table array */
} Xgl_color_list;
```

It contains a starting index of the color table, a length, and a pointer to an array of *Xgl_color* values of type XGL_COLOR_RGB. The color table array must be allocated by the application.

The maximum number of entries allowed in the *colors* array is constrained by the color map attribute **XGL_CMAP_MAX_COLOR_TABLE_SIZE**(3)**.** The number of usable entries in the array is specified by another color map attribute, **XGL_CMAP_COLOR_TABLE_SIZE**(3)**.**

XGL_CMAP_COLOR_TABLE_SIZE(3) should always be set prior to setting the color map with XGL_CMAP_COLOR_TABLE. Furthermore, the value of **XGL_CMAP_COLOR_TABLE_SIZE**(3) should be greater or equal to the sum of the *start_index* and *length* structure elements passed with XGL_CMAP_COLOR_TABLE.

The default value is a two-element, monochrome (black, white) color table.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_CMAP_COLOR_TABLE_SIZE(3)
XGL_CMAP_MAX_COLOR_TABLE_SIZE(3)

NAME

XGL_CMAP_COLOR_TABLE_SIZE,
XGL_CMAP_MAX_COLOR_TABLE_SIZE – define the maximum possible size of the color table associated with a Color Map object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_cmap          cmap;
xgl_object_set(cmap, XGL_CMAP_COLOR_TABLE_SIZE, Xgl_usgn32, 0);
xgl_object_get(cmap, XGL_CMAP_COLOR_TABLE_SIZE, Xgl_usgn32 *);
xgl_object_get(cmap, XGL_CMAP_MAX_COLOR_TABLE_SIZE, Xgl_usgn32 *);
```

DESCRIPTION

The **XGL_CMAP_COLOR_TABLE_SIZE**(3) attribute specifies the number of entries in the color table associated with a Color Map. The **XGL_CMAP_MAX_COLOR_TABLE_SIZE**(3) is a read-only attribute that defines the maximum number of entries the hardware allows in a color table. It is initialized when a Color Map object is attached to a Raster.

A color table associated with an XGL Color Map object can be any size up to the maximum size the hardware allows. For more information about color tables associated with XGL Color Maps, see **XGL_CMAP_COLOR_TABLE**(3)**.**

XGL_CMAP_COLOR_TABLE_SIZE should always be set prior to setting the color map with **XGL_CMAP_COLOR_TABLE**(3)**.**

The default value of XGL_CMAP_COLOR_TABLE_SIZE is 2.

The default value of XGL_CMAP_MAX_COLOR_TABLE_SIZE is 256. When a Color Map is associated with a device, the value of XGL_CMAP_MAX_COLOR_TABLE_SIZE will be reset to be consistent with this device.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CMAP_COLOR_TABLE(3)

NAME

XGL_CMAP_DITHER_MASK – defines the dither mask associated with a Color Map object

SYNOPSIS

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

Xgl_cmap        cmap;

xgl_object_set(cmap, XGL_CMAP_DITHER_MASK, Xgl_usgn32 **, 0);
xgl_object_get(cmap, XGL_CMAP_DITHER_MASK, Xgl_usgn32 **);
```

DESCRIPTION

If an application's Device color type (see **XGL_DEV_COLOR_TYPE**(3)) is XGL_COLOR_RGB and the underlying hardware is indexed, XGL will use a color cube (see **XGL_CMAP_COLOR_-CUBE_SIZE**(3)) and *dithering* to convert the application's RGB colors to the display's indexed colors. This attribute permits the application to specify the dither mask (or dither matrix) that is required for the RGB-to-Index conversion.

The XGL Color Map dither mask size is fixed at 8×8. To change the mask to 2×2 or 4×4, replicate the smaller mask into an 8×8 mask, and set this attribute to point to the new 8×8 mask.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CMAP_COLOR_CUBE_SIZE(3)
XGL_CMAP_COLOR_CUBE_GSIZE(3)
XGL_CMAP_COLOR_CUBE_BSIZE(3)
XGL_CMAP_DITHER_MASK_N(3)
XGL_DEV_COLOR_TYPE(3)

NOTES

The dither mask values should be in the range of 0 to 255.

NAME

XGL_CMAP_DITHER_MASK_N – defines the dither mask size for a Color Map object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_cmap        cmap;
xgl_object_get(cmap, XGL_CMAP_DITHER_MASK_N, Xgl_usgn32 *);
```

DESCRIPTION

If an application's Device color type (see **XGL_DEV_COLOR_TYPE**(3)) is XGL_COLOR_RGB and the underlying hardware is indexed, XGL uses a color cube (see **XGL_CMAP_COLOR_CUBE_SIZE**(3)) and *dithering* to convert the application's RGB colors to the display's indexed colors.

This read-only attribute can be used to get the size of the dither mask. The dither matrix is forced to be square, and its size is fixed at 8×8. To change the mask to 2×2 or 4×4, the smaller mask should be replicated into an 8×8 mask and then set into the XGL dither mask using the attribute **XGL_CMAP_DITHER_MASK**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CMAP_COLOR_CUBE_SIZE(3)
XGL_CMAP_COLOR_CUBE_RSIZE(3)
XGL_CMAP_COLOR_CUBE_GSIZE(3)
XGL_CMAP_COLOR_CUBE_BSIZE(3)
XGL_CMAP_DITHER_MASK(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_CMAP_NAME – permits the application to use color maps that have been created using window system function calls

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_cmap          cmap;
xgl_object_set(cmap, XGL_CMAP_NAME, Colormap, 0);
xgl_object_get(cmap, XGL_CMAP_NAME, Colormap *);
```

DESCRIPTION

The application can use the OpenWindows and X window systems to create a color map that will be used to map colors from the XGL application's color space to the hardware's color space. Note that here the term *color map* refers to the window-system-produced color map and should not be confused with the XGL Color Map object. This attribute, **XGL_CMAP_NAME**(3)**,** is the name given to the color map by the window system. The name can then be passed to XGL so that XGL can share the use of the color map with the window system. To do this, an application should do the following:

1. Create a color map solely through calls to window system functions. (In OpenWindows and X, the application uses the *XCreateColormap* function call.)
2. Get the *color map name* from the window system. (In OpenWindows and X, this is the *XID* returned by the *XCreateColormap* function call.)
3. Get the *pixel mapping array* returned by the window system after it creates the color map (see **XGL_WIN_RAS_PIXEL_MAPPING**(3)).
4. Set the XGL attribute XGL_CMAP_NAME to the window system color map name.
5. If the **XGL_DEV_COLOR_TYPE**(3) attribute is XGL_COLOR_INDEX, set the size of the color table with **XGL_CMAP_COLOR_TABLE_SIZE**(3)**.** If the **XGL_DEV_-COLOR_TYPE**(3) attribute is XGL_COLOR_RGB, set the size of the color cube with **XGL_CMAP_COLOR_CUBE_SIZE**(3)**.**
6. Set XGL_WIN_RAS_PIXEL_MAPPING to the window system pixel mapping array.

It is the responsibility of the application to make sure that the RGB values in the window system color map are correct. When using a window-system-generated color map, the application cannot inquire about the contents of it (by using **xgl_object_get**(3) on the Color Map object's color table). Instead, the application should manipulate the color map through window system calls.

If XGL creates the color map (such as the default Color Map object that is created when a new Window Raster is created), XGL_CMAP_NAME reflects an internal X color map used by XGL and the window system. In other words, in this case XGL implicitly calls the window system functions that create the color map and attaches this window-system-created color map to the Raster.

The default value is NULL. However, if XGL_CMAP_NAME has not been set, it will be set to the X colormap of the first window raster to which it is associated.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_CTX_BACKGROUND_COLOR(3)
XGL_WIN_RAS_PIXEL_MAPPING(3)

NOTES

If the application sets the XGL_CMAP_NAME attribute, the XGL_WIN_RAS_PIXEL_MAPPING attribute must also be set. This guarantees that the application's indexed colors are properly mapped to the hardware indices.

Non-contiguous pixel mapping, when used with XGL primitives requiring interpolated colors, produces undefined results on various graphics hardware and may degrade rendering performance. To ensure correct results for interpolated color rendering, the application should use a *contiguous* pixel mapping. (For a definition of *contiguous pixel mapping*, see **XGL_WIN_RAS_PIXEL_MAPPING**(3)**.**)

NAME

XGL_CMAP_RAMP_LIST,
XGL_CMAP_RAMP_NUM – list and number of ramps in a Color Map object

SYNOPSIS

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

Xgl_cmap          cmap;

xgl_object_set(cmap, XGL_CMAP_RAMP_LIST, Xgl_segment *, 0);
xgl_object_set(cmap, XGL_CMAP_RAMP_NUM, Xgl_usgn32, 0);

xgl_object_get(cmap, XGL_CMAP_RAMP_LIST, Xgl_segment *);
xgl_object_get(cmap, XGL_CMAP_RAMP_NUM, Xgl_usgn32 *);
```

DESCRIPTION

The color table associated with an XGL Color Map object can contain one or more *color ramps*. A color ramp is a section of the color table, of specific length, that is frequently used to store color values that change only by intensity. For instance, a gray ramp could have 32 color values from dark gray to light gray, in a monotonically increasing fashion.

Color ramps are important when using vertex shading (see **XGL_3D_CTX_SURF_FRONT_-ILLUMINATION**(3)) on data whose color type is XGL_COLOR_INDEX. The colors that result from XGL's lighting and shading calculations are interpolated along color ramps within the color table to produce smooth shading effects. Therefore, if the application programmer sets the illumination model to XGL_ILLUM_PER_VERTEX, and the Device color type (see **XGL_DEV_COLOR_TYPE**(3)) is XGL_COLOR_INDEX, then at least one color ramp should be defined within the color table associated with the Raster Color Map object (see **XGL_DEV_-COLOR_MAP**(3)). In the typical case, where each ramp contains one color hue with varying intensity values, XGL expects that the intensities will increase with increasing indices in the color table. If the ramp is not initialized in this way, unexpected shading effects can occur.

Similarly, color ramps are necessary for depth-cueing (see **XGL_3D_CTX_DEPTH_CUE_-MODE**(3)) when the data's color type is XGL_COLOR_INDEX.

XGL color ramps are specified by an array of *Xgl_segment* structures, with one member of the array defining each ramp. *Xgl_segment* is defined as:

```
typedef struct {
        Xgl_usgn32    offset;    /* starting index of ramp in color table */
        Xgl_usgn32    length;    /* length of ramp */
} Xgl_segment;
```

It contains an offset into the color table where the ramp begins, and a length for the ramp.

To make use of ramps, the application should first create a color table (see **XGL_CMAP_-COLOR_TABLE**(3)) containing one or more ramps. Then, the application should declare and initialize an array of *Xgl_segment*s. Each member of the array defines the offset and length of a ramp in the color table. The attribute **XGL_CMAP_RAMP_LIST**(3) defines a pointer to the beginning of this *Xgl_segment* array.

Next, the application should attach both the color table and the *Xgl_segment* array to a Color Map object, via either the **xgl_object_set**(3) or **xgl_object_create**(3) operator, using the attributes XGL_CMAP_COLOR_TABLE and XGL_CMAP_RAMP_LIST, respectively. Finally,

the number of color ramps within the color table must be specified by setting the **XGL_-CMAP_RAMP_NUM**(3) attribute.

The default value of XGL_CMAP_RAMP_LIST is NULL.

The default value of XGL_CMAP_RAMP_NUM is 0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)
XGL_CMAP_COLOR_TABLE(3)

NAME

XGL_CTX_ARC_FILL_STYLE – defines how arcs appear when drawn using the operator xgl_multiarc

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ARC_FILL_STYLE, Xgl_arc_fill_style, 0);
xgl_object_get(ctx, XGL_CTX_ARC_FILL_STYLE, Xgl_arc_fill_style *);
```

DESCRIPTION

The operator **xgl_multiarc**(3) has the capability to draw arcs three different ways, as specified by this attribute. The fill style determines the portion of the arc that is displayed on the Raster. Fill style can be one of the following values:

XGL_ARC_SECTOR
: Two line segments are drawn — one from the start point of the arc to the center of the circle on which the arc lies, and the other from the end point of the arc to the center of the circle. The enclosed area is filled according to the current Context surface attributes.

XGL_ARC_CHORD
: A line segment is drawn from the start point of the arc to the endpoint of the arc. The enclosed area is filled according to the current Context surface attributes.

XGL_ARC_OPEN
: The arc is not closed and not filled. Its properties are specified by the current context line attributes.

The default value is XGL_ARC_SECTOR.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multiarc(3)

NAME

XGL_CTX_ATEXT_ALIGN_HORIZ – defines the horizontal alignment used for annotation text characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_ALIGN_HORIZ, Xgl_stext_align_horiz, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_ALIGN_HORIZ, Xgl_stext_align_horiz *);
```

DESCRIPTION

This attribute specifies the horizontal alignment used when rendering annotation text characters in the Context *ctx*. In XGL, annotation text can be rendered, using four different horizontal alignments, with respect to the text position. The following values are allowed:

XGL_STEXT_ALIGN_HORIZ_RIGHT

XGL_STEXT_ALIGN_HORIZ_NORMAL

These values are used to specify a text rendered at the right side of the annotation position of the string.

XGL_STEXT_ALIGN_HORIZ_LEFT

This value should be used each time an application wants the text to be on the left side of the annotation position given for the annotation text operator.

XGL_STEXT_ALIGN_HORIZ_CENTER

This value centers the text on the annotation position given for the annotation text operator.

The application can use this attribute to specify how the text string should be aligned with respect to the position of the text. When rendering annotation text using **xgl_annotation_text**(3)**,** the characters are interpreted from stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's virtual device coordinate (VDC) system.

The default value is XGL_STEXT_ALIGN_HORIZ_NORMAL.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)

NAME

XGL_CTX_ATEXT_ALIGN_VERT – defines the vertical alignment used for annotation text

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_ALIGN_VERT, Xgl_stext_align_vert, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_ALIGN_VERT, Xgl_stext_align_vert *);
```

DESCRIPTION

This attribute specifies the vertical alignment used when rendering annotation text characters in the Context *ctx*. In XGL, annotation text can be rendered, using six different vertical alignments, with respect to the text position. The following values are allowed:

XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_ALIGN_VERT_NORMAL
: Indicates that the base line of the character cells must map on the annotation position specified for the string.

XGL_STEXT_ALIGN_VERT_BOTTOM
XGL_STEXT_ALIGN_VERT_TOP
: Used each time an application wants the annotation position given for annotation text operators to match the bottom line or the top line, respectively, of the character cells used in the current font.

XGL_STEXT_ALIGN_VERT_HALF
: Ensures that the text will be vertically centered on the annotation position given to the annotation text operators.

XGL_STEXT_ALIGN_VERT_CAP
: Maps the cap line of character cells to the annotation position given for the annotation text operators.

The application can use this attribute to specify how the text string should be aligned with respect to the position of the text. When rendering annotation text using **xgl_annotation_text**(3)**,** the characters are interpreted from stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system. The characters are transformed to match the given size value in the application's virtual device coordinate (VDC) system.

The default value is XGL_STEXT_ALIGN_VERT_NORMAL.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)

NAME

XGL_CTX_ATEXT_CHAR_HEIGHT – defines the nominal size of annotation text characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_CHAR_HEIGHT, float, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_CHAR_HEIGHT, float *);
```

DESCRIPTION

This attribute specifies the nominal height of text characters used with annotation text (**xgl_annotation_text**(3)), expressed in virtual device coordinates (VDC).

When rendering annotation text using xgl_annotation_text, the characters are interpreted from the stroke text data file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate systems, and the characters are transformed to match the given size value in the application's virtual device coordinate system.

This attribute affects globally the size of XGL annotation characters. Not only does it affect the height of annotation characters, but also the width and spacing between annotation characters (see **XGL_CTX_STEXT_CHAR_SPACING**(3) and **XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR**(3)).

The default value is 100.0.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)
XGL_CTX_STEXT_CHAR_SPACING(3)

NAME

XGL_CTX_ATEXT_CHAR_SLANT_ANGLE – defines the angle that annotation text makes with the upright direction

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_CHAR_SLANT_ANGLE, float, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_CHAR_SLANT_ANGLE, float *);
```

DESCRIPTION

This attribute specifies the angle of inclination of annotation text characters, expressed as an angle (in radians), with respect to the upright direction.

When rendering annotation text using **xgl_annotation_text**(3)**,** the characters are interpreted from the stroke text file in text local coordinates (TLC). This coordinate system is independent of the user's coordinates, and the characters are transformed to match the given size value in the user's virtual device coordinate (VDC) system.

This attribute is used to modify the appearance of the characters without changing their spacing, height, and width. It can take values between $-\pi/2$ and $+\pi/2$. A slant angle between $-\pi/2$ and 0 will slant the characters in the backward direction. A slant angle between 0 and $+\pi/2$ will slant the characters in the forward direction.

This attribute does not affect the annotation text alignment, as specified by the attributes **XGL_CTX_ATEXT_ALIGN_HORIZ**(3) and **XGL_CTX_ATEXT_ALIGN_VERT**(3)**.**

The default value is 0.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)
XGL_CTX_ATEXT_ALIGN_HORIZ(3)
XGL_CTX_ATEXT_ALIGN_VERT(3)

NAME

XGL_CTX_ATEXT_CHAR_UP_VECTOR – defines the up vector for annotation characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_CHAR_UP_VECTOR, Xgl_pt_f2d *, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_CHAR_UP_VECTOR, Xgl_pt_f2d *);
```

DESCRIPTION

This attribute specifies the *up vector* for annotation text characters within the Context *ctx*. The up vector gives the upright direction of characters in text local coordinates (TLC).

Changing the up vector is equivalent to defining a rotation for the characters when text is rendered using the **xgl_annotation_text**(3) operator. This vector is independent of the orientation of the virtual device coordinates (VDC) that is set by the application with the **XGL_CTX_VDC_ORIENTATION**(3) attribute.

With the VDC orientation set to either XGL_Y_UP_Z_TOWARD or XGL_Y_DOWN_Z_AWAY, an up vector of (0.0, 1.0) renders the text normally. XGL automatically sets the correct internal values to the transformation of the TLC so that the VDC orientation does not have any effect on the text to display.

The default value is (0.0, 1.0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)
XGL_CTX_VDC_ORIENTATION(3)

NAME

XGL_CTX_ATEXT_PATH – defines the path used when rendering annotation text

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ATEXT_PATH, Xgl_stext_path, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_PATH, Xgl_stext_path *);
```

DESCRIPTION

This attribute specifies the path used when rendering annotation text characters within the Context *ctx*. In XGL, annotation text can be rendered in four different directions from the text position. The following values are allowed to define the text path:

XGL_STEXT_PATH_RIGHT
XGL_STEXT_PATH_LEFT
XGL_STEXT_PATH_UP
XGL_STEXT_PATH_DOWN

The application can use this attribute to specify in which direction the text should be rendered. When rendering stroke text using the **xgl_annotation_text**(3) operator, the characters are interpreted from the text font file, using the path specified by the application to generate the correct data in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system. The annotation characters are transformed to match the given size value in the application's virtual device coordinate (VDC) system.

The default value is XGL_STEXT_PATH_RIGHT.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)

NAME

XGL_CTX_ATEXT_STYLE – defines the style used for rendering annotation text

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_ATEXT_STYLE, Xgl_atext_style, 0);
xgl_object_get(ctx, XGL_CTX_ATEXT_STYLE, Xgl_atext_style *);
```

DESCRIPTION

This attribute specifies the style used when rendering annotation text characters in the Context *ctx*. In XGL, annotation text can be rendered using two different styles. The following values are allowed:

XGL_ATEXT_STYLE_NORMAL
: Indicates that the characters are rendered at the annotation position specified for the string, without any particular effect.

XGL_ATEXT_STYLE_LINE
: Identical to XGL_ATEXT_STYLE_NORMAL, but the *reference point* is linked to the *annotation point* by a single line with attributes from the current settings for **xgl_multipolyline**(3). For a definition of *reference point* and *annotation point*, see **xgl_annotation_text**(3).

The default value is XGL_ATEXT_STYLE_NORMAL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)
xgl_multipolyline(3)

NAME

XGL_CTX_BACKGROUND_COLOR – defines the color used for painting background pixels

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_BACKGROUND_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_BACKGROUND_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute defines the color used to paint background pixels. When the XGL Raster, associated with the current Context, is cleared via a call to **xgl_context_new_frame**(3), XGL_CTX_BACKGROUND_COLOR is used to fill the Raster.

When XGL_3D_CTX_SURF_BACK_FILL_STYLE or XGL_CTX_SURF_FRONT_FILL_STYLE is XGL_SURF_FILL_OPAQUE_STIPPLE and the fill Raster pixel is *not* set, the corresponding pixel in the Destination Raster is set to the color XGL_CTX_BACKGROUND_COLOR.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

The default value is: INDEX 0 or RGB (0,0,0).

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_new_frame(3)
xgl_context_copy_raster(3)
XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_CLIP_PLANES – specifies which of the six clipping planes are used in view clipping

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_CLIP_PLANES, Xgl_clip_planes, 0);
xgl_object_get(ctx, XGL_CTX_CLIP_PLANES, Xgl_clip_planes *);
```

DESCRIPTION

This attribute specifies which planes are used in view clipping within the Context *ctx*. The application can enable any combination of the clipping planes by logically OR'ing flags representing each plane. The flags are:

XGL_CLIP_XMIN
XGL_CLIP_XMAX
XGL_CLIP_YMIN
XGL_CLIP_YMAX
XGL_CLIP_ZMIN
XGL_CLIP_ZMAX

Each flag handles a specific clipping plane in the virtual device coordinates (VDC). The flags XGL_CLIP_ZMIN and XGL_CLIP_ZMAX are valid only when applied to 3D Contexts.

The attribute **XGL_CTX_VIEW_CLIP_BOUNDS**(3) defines the specific boundaries in VDC space of the clipping area or volume. Note that the values in VDC space that are minimum and maximum depend on the value of the XGL Context attribute **XGL_CTX_VDC_ORIENTATION**(3)**.**

The default value is ''no clipping planes enabled''—that is, 0 (zero).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VDC_WINDOW(3)
XGL_CTX_VIEW_CLIP_BOUNDS(3)

NOTE

In 3D, XGL does not view clip geometry at the *z* boundary of the VDC window (see **XGL_CTX_VDC_WINDOW**(3)). The application needs to specify and enable view clip planes in *z* to obtain view clipping in *z*. The planes are specified with **XGL_CTX_VIEW_CLIP_BOUNDS**(3) and enabled with **XGL_CTX_CLIP_PLANES**(3)**.**

NAME

XGL_CTX_DEFERRAL_MODE – specifies whether primitives will be buffered before being sent to the Device

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_DEFERRAL_MODE, Xgl_deferral_mode, 0);
xgl_object_get(ctx, XGL_CTX_DEFERRAL_MODE, Xgl_deferral_mode *);
```

DESCRIPTION

This attribute controls whether graphics sent to the Device associated with the Context *ctx* will be buffered. There are two deferral modes:

XGL_DEFER_ASTI

In ASTI (*at some time*) mode, XGL keeps an internal buffer of primitives. It sends these primitives to the Device for display either when the buffer is full or when the XGL operator **xgl_context_flush**(3) is called by the application (hence, the name *at some time*). If the application wants to guarantee that primitives will be rendered on the Device, it must explicitly call **xgl_context_flush**(3)**.** The method of buffering is device-dependent.

XGL_DEFER_ASAP

In ASAP (*as soon as possible*) mode, XGL sends a primitive to the Device for display as soon as it can. Setting XGL_CTX_DEFERRAL_MODE to XGL_DEFER_ASAP automatically calls **xgl_context_flush**(3) (ctx, FALSE).

The default value is XGL_DEFER_ASTI.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_flush(3)

NAME

XGL_CTX_DEVICE – defines the Device associated with a Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_DEVICE, Xgl_dev, 0);
xgl_object_get(ctx, XGL_CTX_DEVICE, Xgl_dev *);
```

DESCRIPTION

This attribute defines the XGL Device object associated with the Context *ctx*. The Device object is created with the XGL operator **xgl_object_create**(3)**.** The device specifies the drawing surface for all subsequent graphic operations for the Context.

The current XGL implementation does not render into 1-bit-deep Memory Rasters, but the **xgl_context_copy_raster**(3) operator can copy from a 1-bit-deep Memory Raster into another 1-bit-deep Memory Raster.

The default value is NULL.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_CTX_EDGE_AA_BLEND_EQ,
XGL_CTX_EDGE_AA_FILTER_WIDTH,
XGL_CTX_EDGE_AA_FILTER_SHAPE – controls if and how edges are antialiased

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_AA_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_set(ctx, XGL_CTX_EDGE_AA_FILTER_WIDTH, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_EDGE_AA_FILTER_SHAPE, Xgl_filter_shape, 0);

xgl_object_get(ctx, XGL_CTX_EDGE_AA_BLEND_EQ, Xgl_blend_eq *);
xgl_object_get(ctx, XGL_CTX_EDGE_AA_FILTER_WIDTH, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_EDGE_AA_FILTER_SHAPE, Xgl_filter_shape *);
```

DESCRIPTION

These attributes control if and how edges are antialiased. Antialiasing smooths out single-pixel-wide edges by spreading out the spectral energy of the stroke over 2 or more adjacent pixels. The result is that the sides of the stroke are slightly blurred and, as seen from a normal viewing distance, smoother than non-antialiased edges.

The blurring of the edge is done by filtering the stroke onto neighboring pixels. The shape and size of this filter controls how the antialiased edge will look. Currently, XGL_FILTER_GAUSSIAN is the only possible value for the **XGL_CTX_EDGE_AA_FILTER_SHAPE**(3) attribute.

The width of the filter is controlled by the attribute **XGL_CTX_EDGE_AA_FILTER_WIDTH**(3)**.** This is the number of pixels touched by the filter. A Device will use the largest filter width it supports which is less than or equal to the specified filter width. A width of 0 is not an error and is treated as if the width was 1.

The attribute **XGL_CTX_EDGE_AA_BLEND_EQ**(3) controls how the pixel values in the antialiased edge are blended with the existing pixel values in the Device. The data type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that antialiasing use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_-CONST_BG.

All three attributes must have consistent settings in order to achieve visually pleasing results. Some notes on using these attributes:

- Both the XGL_CTX_EDGE_AA_BLEND_EQ and the XGL_CTX_EDGE_AA_FILTER_WIDTH must be set to something other than defaults to achieve antialiasing.
- Setting the XGL_CTX_EDGE_AA_BLEND_EQ to XGL_BLEND_NONE and the XGL_-CTX_EDGE_AA_FILTER_WIDTH to greater than 1 is a way to *undraw* antialiased edges. The color used for undrawing is the color that the edge would have been drawn in if antialiasing was off.
- Setting the XGL_CTX_EDGE_AA_BLEND_EQ to other than NONE and the XGL_CTX_-EDGE_AA_FILTER_WIDTH to 1 is undefined and should not be used. The results will be device dependent.

The default value for XGL_CTX_EDGE_AA_BLEND_EQ is XGL_BLEND_NONE.

The default value for XGL_CTX_EDGE_AA_FILTER_WIDTH is 1.

The default value for XGL_CTX_EDGE_AA_FILTER_SHAPE is XGL_FILTER_GAUSSIAN.

All these attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTES

Antialiasing support on 2D Contexts is device dependent.

NAME

XGL_CTX_EDGE_ALT_COLOR – defines the alternate color used in patterned edges

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_ALT_COLOR, Xgl_color * , 0);
xgl_object_get(ctx, XGL_CTX_EDGE_ALT_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute sets the color used to fill even (or alternate) line segments when the edge style (**XGL_CTX_EDGE_STYLE**(3)) in the Context *ctx* is set to **XGL_LINE_ALT_PATTERNED**(3)**.** The odd line segments are filled using the color specified by **XGL_CTX_EDGE_COLOR**(3)**.**

This attribute applies only to primitives capable of edge rendering.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 0 or RGB (0,0,0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_STYLE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_EDGE_CAP – defines the shape of the endpoints of an edge

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_CAP, Xgl_line_cap, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_CAP, Xgl_line_cap *);
```

DESCRIPTION

This attribute defines the shape of the endpoints of an edge. Edges with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

The *cap* is a projection beyond the endpoint of the edge. The amount of the projection depends on the edge width (see **XGL_CTX_EDGE_WIDTH_SCALE_FACTOR**(3)).

This attribute is used in conjunction with the **XGL_CTX_EDGE_JOIN**(3) and **XGL_CTX_EDGE_MITER_LIMIT**(3) attributes to specify the shape of wide edges.

The following information enumerates the possible values for XGL_CTX_EDGE_CAP.

XGL_CAP_BUTT
: Butt cap: The edge is squared off at the end with no projection.

XGL_CAP_SQUARE
: Square ends: The end of the edge is extended a distance half of the edge width.

XGL_CAP_ROUND
: Rounded ends: A semicircle with the diameter equal to the edge width is appended to the end of the edge.

The default value is XGL_CAP_BUTT.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_EDGE_JOIN(3)
XGL_CTX_EDGE_MITER_LIMIT(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_EDGE_COLOR – defines the color used for edges of surfaces

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_EDGE_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute sets the color in which surface edges are drawn. This color applies only to primitives capable of edge rendering.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NOTES

The color of edges is not affected by shading or lighting computations.

NAME

XGL_CTX_EDGE_JOIN – defines the shape of joints between edge segments

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_JOIN, Xgl_line_join, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_JOIN, Xgl_line_join *);
```

DESCRIPTION

This attribute defines the shape of *joins* (or joints) between edge segments. Edges with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

The *join* is a projection beyond the endpoints of two connected edge segments, that mathematically takes its point of emanation from the intersection of the two edges' center lines.

This attribute is used in conjunction with the **XGL_CTX_EDGE_CAP**(3) and **XGL_CTX_EDGE_-MITER_LIMIT**(3) attributes to specify the shape of wide edges.

The following information enumerates the possible values for XGL_CTX_EDGE_JOIN.

XGL_JOIN_DEVICE
: Ensures that the associated hardware uses the most efficient join type. This could mean that joins are not performed.

XGL_JOIN_BEVEL
: Each edge segment is fitted with a butt end cap. The notch between the two end caps is filled with a triangle.

XGL_JOIN_MITER
: Mitered corners are created by extending the edge segments until the outside edges of each of the wide edge intersect. If the angle between the two wide edges is very acute, however, the edge join projection is truncated to a beveled join. The attribute **XGL_CTX_EDGE_MITER_LIMIT**(3) controls when the truncation occurs.

XGL_JOIN_ROUND
: Each edge is fitted with a butt end cap. The notch between the two end caps is filled with a circular arc, the diameter of which equals the edge width. The center of the arc is at the point where the two edges intersect.

The default value is XGL_JOIN_MITER.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_CTX_EDGE_CAP(3)
XGL_CTX_EDGE_MITER_LIMIT(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_EDGE_MITER_LIMIT – defines the limit for mitering of edge segment corners

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_MITER_LIMIT, float, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_MITER_LIMIT, float *);
```

DESCRIPTION

This attribute sets a limit that determines the treatment of edge segments joined by mitering (for a description of mitered edge segments, see **XGL_CTX_EDGE_JOIN**(3)). XGL_CTX_EDGE_MITER_LIMIT must be greater than or equal to 1.0.

The *miter length* is the distance between the inside and outside intersections of the two edge segments being joined. If the ratio of the miter length to the edge width (see **XGL_CTX_EDGE_WIDTH_SCALE_FACTOR**(3)) exceeds the miter limit, the edges receive a beveled edge join.

Edges with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

To get a sense of why the miter limit is necessary, imagine two edge segments that are *almost* parallel and that should end with a miter join. The miter join would, if not constrained by the miter limit, shoot off past the intersection of the two edges and dwarf the length of the two edges it was supposed to be joining.

This attribute is used in conjunction with the XGL_CTX_EDGE_CAP and the XGL_CTX_EDGE_JOIN attributes to specify the shape of wide edges.

The default value is 10.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_EDGE_CAP(3)
XGL_CTX_EDGE_JOIN(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_EDGE_PATTERN – specifies the Line Pattern used for patterned edges

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_PATTERN, Xgl_lpat, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_PATTERN, Xgl_lpat *);
```

DESCRIPTION

This attribute is used to set or get the handle of the Line Pattern object XGL uses when rendering surfaces with patterned edges. It is applicable only when the **XGL_CTX_EDGE_-STYLE**(3) attribute is set to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED.

A Line Pattern object, which defines a patterned line, contains an array of line segment lengths specifying which portions of a line are *on* and *off*. (For more information, see **xgl_object_create**(3) and **XGL_LPAT_DATA**(3)**.**) The Line Pattern is specified in the coordinate system defined by the attribute **XGL_LPAT_COORD_SYS**(3)**.** The pattern wraps around over and over again along edges and around corners until the end of the edge is reached.

The application can either create its own Line Pattern object to be used for edges, or it can use one of the following XGL predefined line patterns.

xgl_lpat_dotted	.....
xgl_lpat_dashed	----------
xgl_lpat_dashed_dotted	...__...__
xgl_lpat_cgm_dotted	.....
xgl_lpat_cgm_dashed	_ _ _ _ _ _
xgl_lpat_dash_dot	._._._._._
xgl_lpat_dash_dot_dotted	_.._.._.._..
xgl_lpat_long_dashed	__ __ __ __

For example, to specify edges using dashed lines, the application sets the edge pattern as follows:

```
xgl_object_set(ctx, XGL_CTX_EDGE_STYLE, XGL_LINE_PATTERNED, 0);
xgl_object_set(ctx, XGL_CTX_EDGE_PATTERN, xgl_lpat_dashed, 0);
```

The color of a patterned edge is specified by one or both of the two attributes **XGL_CTX_-EDGE_COLOR**(3) and **XGL_CTX_EDGE_ALT_COLOR**(3)**.**

XGL_CTX_EDGE_PATTERN applies only to primitives capable of edge rendering.

The default value is xgl_lpat_dotted, a predefined Line Pattern of evenly spaced dots.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_object_create(3)
XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_ALT_COLOR(3)
XGL_CTX_EDGE_STYLE(3)
XGL_LPAT_COORD_SYS(3)
XGL_LPAT_DATA(3)

NAME

XGL_CTX_EDGE_STYLE – defines the edge style used for edges of surfaces

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_EDGE_STYLE, Xgl_line_style, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_STYLE, Xgl_line_style *);
```

DESCRIPTION

This attribute defines the style for edges on surfaces within the Context *ctx*. Edges are drawn on the perimeter of the surface, but they are not drawn on boundaries produced by clipping. This attribute applies only to primitives capable of edge rendering.

The following enumerates the values recognized by XGL. The values are the same as those recognized for line style, as defined by the attribute **XGL_CTX_LINE_STYLE**(3)**.**

XGL_LINE_PATTERNED
: Edges are patterned using the Line Pattern object given by the **XGL_CTX_EDGE_PATTERN**(3) attribute and the color specified by the **XGL_CTX_EDGE_COLOR**(3) attribute.

XGL_LINE_ALT_PATTERNED
: Edges are patterned using the Line Pattern object given by the XGL_CTX_EDGE_PATTERN attribute. The color is alternately specified by the XGL_CTX_EDGE_COLOR attribute for odd line segments, and by the **XGL_CTX_EDGE_ALT_COLOR**(3) attribute for even line segments.

XGL_LINE_SOLID
: Edges are *not* patterned. A solid edge is drawn using the color specified by the XGL_CTX_EDGE_COLOR attribute.

When a patterned edge is specified, the Context attribute XGL_CTX_EDGE_PATTERN is used to specify the edge pattern.

The default value is XGL_LINE_SOLID.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_object_create(3)

XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_ALT_COLOR(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)

NAME

XGL_CTX_EDGE_WIDTH_SCALE_FACTOR – defines the edge-width scale factor used to determine the width of surface edges

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_EDGE_WIDTH_SCALE_FACTOR, float, 0);
xgl_object_get(ctx, XGL_CTX_EDGE_WIDTH_SCALE_FACTOR, float *);
```

DESCRIPTION

This attribute specifies the scale factor by which a nominal edge width is multiplied to produce the edge width in device coordinates (DC). This attribute is similar to **XGL_CTX_LINE_WIDTH_SCALE_FACTOR**(3), which defines an analogous scale factor for XGL lines. Wide edges are drawn using a butt cap and device join.

The XGL_CTX_EDGE_WIDTH_SCALE_FACTOR applies only to primitives capable of edge rendering.

The default value of 0.0 corresponds to the notion of *thin lines* as defined by the X community. This means the method actually used to scan-convert the lines is device dependent and generally gives the best performance on that device, with a limited set of constraints when drawing 2D vectors (see *The X Window System*, R.W. Scheifler & AL., Digital Press). Any value greater than or equal to 1.0 will guarantee X-compliant lines in 2D. There is no compliance defined for 3D vectors.

In most cases, best results are obtained with an odd value for the scale factor, because the resulting width is correctly balanced on both sides of the edge. Setting XGL_CTX_EDGE_WIDTH_SCALE_FACTOR to a value that produces a width greater than or equal to 1.0 in device coordinates affect drawing performance.

The default value is 0.0.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_GEOM_DATA_IS_VOLATILE – specifies whether the application's data can be used after XGL primitives return

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_GEOM_DATA_IS_VOLATILE, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_CTX_GEOM_DATA_IS_VOLATILE, Xgl_boolean *);
```

DESCRIPTION

When this attribute is set to FALSE, the application program guarantees that the geometry data being sent to XGL will not be modified or destroyed until a subsequent **xgl_context_flush**(3) (ctx, XGL_FLUSH_GEOM_DATA) is called.

Some rendering devices can render by directly processing the application's data. While the rendering device is processing that data, it is possible for the rendering routine to return to the application's code to gain a degree of parallelism, thus improving the overall performance. This can only be done if it is guaranteed that the geometry data will not be changed or destroyed by the application.

When **XGL_CTX_GEOM_DATA_IS_VOLATILE**(3) is set to FALSE, the application is indicating to XGL that the geometry data will not change. The Context primitives can potentially return to the application program before the data in a parameter list has been completely used. If the application then wants to change or free the data, an appropriate **xgl_context_flush**(3) must be called.

When **XGL_CTX_GEOM_DATA_IS_VOLATILE**(3) is set to TRUE, the application cannot guarantee the state of the data. All access to the application's data must be finished before control is given back to the application's program.

Resetting **XGL_CTX_GEOM_DATA_IS_VOLATILE**(3) from FALSE to TRUE automatically calls **xgl_context_flush**(3) (ctx, XGL_FLUSH_GEOM_DATA).

The default value is TRUE.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_flush(3)

NAME

XGL_CTX_GLOBAL_MODEL_TRANS – defines the global model transformation in a Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_GLOBAL_MODEL_TRANS, Xgl_trans, 0);
xgl_object_get(ctx, XGL_CTX_GLOBAL_MODEL_TRANS, Xgl_trans *);
```

DESCRIPTION

This attribute defines the Transform object used for the Global Model Transform, which transforms geometry data from global model coordinates (GMC) to world coordinates (WC).

The Local Model Transform (as defined by the attribute **XGL_CTX_LOCAL_MODEL_TRANS**(3)**)** and the Global Model Transform (as defined by this attribute) are multiplied together, with the Local Model Transform on the left of the Global Model Transform, to form the Model Transform. The attribute **XGL_CTX_MODEL_TRANS**(3) returns a Transform object, which represents the concatenation of the XGL_CTX_LOCAL_MODEL_TRANS and the **XGL_CTX_GLOBAL_MODEL_TRANS**(3)**.**

If you set this Context attribute to a Transform whose handle is obtained from the same or a different Context, the original attribute and this attribute use the same Transform. Changing the Transform of one Context attribute has the same effect on the other Context attribute. Be careful of this consequence when you share a Transform in this manner. Do not attempt to set this Context attribute to a handle to a Transform obtained from **XGL_CTX_MODEL_TRANS**(3)**, XGL_CTX_MC_TO_DC_TRANS**(3)**,** or **XGL_3D_CTX_NORMAL_TRANS**(3) because these Transforms are read-only.

The default value is the Identity Transform.

This attribute is pushed by **xgl_context_push**(3)**,** and can also be manipulated with **xgl_context_update_model_trans**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_update_model_trans(3)
XGL_3D_CTX_NORMAL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_MC_TO_DC_TRANS(3)
XGL_CTX_MODEL_TRANS(3)

NAME

XGL_CTX_LINE_AA_BLEND_EQ,
XGL_CTX_LINE_AA_FILTER_WIDTH,
XGL_CTX_LINE_AA_FILTER_SHAPE – controls if and how lines are antialiased

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_LINE_AA_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_set(ctx, XGL_CTX_LINE_AA_FILTER_WIDTH, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_LINE_AA_FILTER_SHAPE, Xgl_filter_shape, 0);

xgl_object_get(ctx, XGL_CTX_LINE_AA_BLEND_EQ, Xgl_blend_eq *);
xgl_object_get(ctx, XGL_CTX_LINE_AA_FILTER_WIDTH, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_LINE_AA_FILTER_SHAPE, Xgl_filter_shape *);
```

DESCRIPTION

These attributes control if and how lines are antialiased. Antialiasing smooths out single-pixel-wide lines by spreading out the spectral energy of the stroke over 2 or more adjacent pixels. The result is that the sides of the stroke are slightly blurred and, as seen from a normal viewing distance, smoother than non-antialiased lines.

The blurring of the line is done by filtering the stroke onto neighboring pixels. The shape and size of this filter controls how the antialiased line will look. Currently, XGL_FILTER_GAUSSIAN is the only possible value for the **XGL_CTX_LINE_AA_FILTER_SHAPE**(3) attribute.

The width of the filter is controlled by the attribute **XGL_CTX_LINE_AA_FILTER_WIDTH**(3)**.** This is the number of pixels touched by the filter. A Device will use the largest filter width it supports which is less than or equal to the specified filter width. A width of 0 is not an error and is treated as if the width was 1.

The attribute **XGL_CTX_LINE_AA_BLEND_EQ**(3) controls how the pixel values in the antialiased line are blended with the existing pixel values in the Device. The data type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that antialiasing use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_-CONST_BG.

All three attributes must have consistent settings in order to achieve visually pleasing results. Some notes on using these attributes:

- Both the XGL_CTX_LINE_AA_BLEND_EQ and the XGL_CTX_LINE_AA_FILTER_WIDTH must be set to something other than defaults to achieve antialiasing.
- Setting the XGL_CTX_LINE_AA_BLEND_EQ to XGL_BLEND_NONE and the XGL_CTX_-LINE_AA_FILTER_WIDTH to greater than 1 is a way to *undraw* antialiased lines. The color used for undrawing is the color that the line would have been drawn in if antialiasing was off.
- Setting the XGL_CTX_LINE_AA_BLEND_EQ to other than NONE and the XGL_CTX_-LINE_AA_FILTER_WIDTH to 1 is undefined and should not be used. The results will be device dependent.

The default value for XGL_CTX_LINE_AA_BLEND_EQ is XGL_BLEND_NONE.

The default value for XGL_CTX_LINE_AA_FILTER_WIDTH is 1.

The default value for XGL_CTX_LINE_AA_FILTER_SHAPE is XGL_FILTER_GAUSSIAN.

All these attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTES

Antialiasing support on 2D Contexts is device dependent.

NAME

XGL_CTX_LINE_ALT_COLOR – defines the alternate color used in patterned lines

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_LINE_ALT_COLOR, Xgl_color * , 0);
xgl_object_get(ctx, XGL_CTX_LINE_ALT_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute sets the color to fill even (or alternate) line segments when the line style (**XGL_CTX_LINE_STYLE**(3)) in the Context *ctx* is set to **XGL_LINE_ALT_PATTERNED**(3)**.** The odd line segments are filled using the color specified by **XGL_CTX_LINE_COLOR**(3)**.**

This attribute applies to all primitives rendering *line* segments. Surface edges have their own set of attributes.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 0 or RGB (0,0,0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_STYLE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_LINE_CAP – defines the shape of the endpoints of lines and curves

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_LINE_CAP, Xgl_line_cap, 0);
xgl_object_get(ctx, XGL_CTX_LINE_CAP, Xgl_line_cap *);
```

DESCRIPTION

This attribute defines the shape of the endpoints of line segments or curves. Lines and curves with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

The *cap* is a projection beyond the endpoint of the line. The amount of the projection depends on the line width (see **XGL_CTX_LINE_WIDTH_SCALE_FACTOR**(3)).

This attribute is used in conjunction with the **XGL_CTX_LINE_JOIN**(3) and **XGL_CTX_LINE_-MITER_LIMIT**(3) attributes to specify the shape of wide lines.

The following information enumerates the possible values for XGL_CTX_LINE_CAP.

XGL_CAP_BUTT
: Butt cap: The line is squared off at the end with no projection.

XGL_CAP_SQUARE
: Square ends: The end of the line is extended a distance half of the line width.

XGL_CAP_ROUND
: Rounded ends: A semicircle with the diameter equal to the line width is appended to the end of the line.

The default value is XGL_CAP_BUTT.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_LINE_COLOR – defines the color of lines and curves in a Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_LINE_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_LINE_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute is used to set the color in which lines and curves are drawn. The color applies only to primitives rendering line segments. Surface edges have their own set of attributes.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multipolyline(3)
xgl_nurbs_curve(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_DEV_COLOR_MAP(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_CTX_LINE_COLOR_SELECTOR – selects the source of a line's color from the Context or from vertex data

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx                 ctx;
xgl_object_set(ctx, XGL_CTX_LINE_COLOR_SELECTOR, Xgl_line_color_sel, 0);
xgl_object_get(ctx, XGL_CTX_LINE_COLOR_SELECTOR, Xgl_line_color_sel *);
```

DESCRIPTION

This attribute selects the source of a line's color from the Context or the vertex data provided with a polyline.

XGL_CTX_LINE_COLOR_SELECTOR(3) specifies the color selection used for lines in the 2D or 3D Context *ctx*. Two options are available for this attribute:

XGL_LINE_COLOR_CONTEXT
: XGL takes the line color from the Context attribute XGL_CTX_LINE_COLOR. This is the color of a line at the beginning of the rendering pipeline. Depth-cueing affects the final appearance of the line in 3D.

XGL_LINE_COLOR_VERTEX
: If the data supplied with a polyline primitive has colors for the individual vertices, XGL applies these colors to the vertices upon entry to the rendering pipeline. Line color interpolation and depth-cueing affect the final appearance of the line in 3D. If the polyline primitive does not have colors for the individual vertices, XGL uses the Context line color.

The default value is XGL_LINE_COLOR_VERTEX.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_LINE_COLOR(3)

NAME

XGL_CTX_LINE_JOIN – defines the shape of joints between line segments

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_LINE_JOIN, Xgl_line_join, 0);
xgl_object_get(ctx, XGL_CTX_LINE_JOIN, Xgl_line_join *);
```

DESCRIPTION

This attribute defines the shape of *joins* (or joints) between line segments or curves. Lines and curves with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

The *join* is a projection, beyond the endpoints of two connected lines, that mathematically takes its point of emanation from the intersection of the two lines' center lines.

This attribute is used in conjunction with the **XGL_CTX_LINE_CAP**(3) and **XGL_CTX_LINE_-MITER_LIMIT**(3) attributes to specify the shape of wide lines.

The following enumerates the possible values for XGL_CTX_LINE_JOIN:

XGL_JOIN_DEVICE
: Ensures that the associated hardware uses the most efficient join type. This could mean that joins are not performed.

XGL_JOIN_BEVEL
: Each line is fitted with a butt end cap. The notch between the two end caps is filled with a triangle.

XGL_JOIN_MITER
: Mitered corners are created by extending the line segments until the outside edges of each of the wide lines intersect. If the angle between the two wide lines is very acute, however, the line join projection is truncated to a beveled join. The attribute XGL_CTX_LINE_MITER_LIMIT controls when the truncation occurs.

XGL_JOIN_ROUND
: Each line is fitted with a butt end cap. The notch between the two end caps is filled with a circular arc, the diameter of which equals the line width. The center of the arc is at the point where the two lines intersect.

The default value is XGL_JOIN_MITER.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_LINE_MITER_LIMIT – defines the limit for mitering of line segment corners

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_LINE_MITER_LIMIT, float, 0);
xgl_object_get(ctx, XGL_CTX_LINE_MITER_LIMIT, float *);
```

DESCRIPTION

This attribute sets a limit that determines the treatment of line segments joined by mitering (for a description of mitered line segments, see **XGL_CTX_LINE_JOIN**(3)). XGL_CTX_-LINE_MITER_LIMIT must be greater than or equal to 1.0.

The *miter length* is the distance between the inside and outside intersections of the two lines being joined. If the ratio of the miter length to the line width (see **XGL_CTX_LINE_-WIDTH_SCALE_FACTOR**(3)) exceeds the miter limit, the lines receive a beveled line join.

Lines and curves with a width, in device coordinates (DC), less than 2.0 ignore this attribute.

To get a sense of why the miter limit is necessary, imagine two lines that are *almost* parallel and that should end with a miter join. The miter join would, if not constrained by the miter limit, shoot off past the intersection of the two lines and dwarf the length of the two lines it was supposed to be joining.

This attribute is used in conjunction with the XGL_CTX_LINE_CAP and the XGL_CTX_LINE_-JOIN attributes to specify the shape of wide lines.

The default value is 10.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NAME

XGL_CTX_LINE_PATTERN – defines the pattern used for patterned lines

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_LINE_PATTERN, Xgl_lpat, 0);
xgl_object_get(ctx, XGL_CTX_LINE_PATTERN, Xgl_lpat *);

DESCRIPTION

This attribute is used to set or get the handle of the Line Pattern object XGL uses when rendering patterned lines or curves. It is applicable only when the **XGL_CTX_LINE_STYLE**(3) attribute is set to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED.

A Line Pattern object, which defines a patterned line, contains an array of line segment lengths specifying which portions of a line are *on* and *off* (for more information, see **xgl_object_create**(3) and **XGL_LPAT_DATA**(3)). The Line Pattern is specified in the coordinate system defined by the attribute **XGL_LPAT_COORD_SYS**(3)**.**

The pattern wraps around over and over again along lines, curves, and around corners until the end of the line is reached.

The application can either create its own Line Pattern object to be used for lines, or it can use one of the following XGL predefined line patterns.

xgl_lpat_dotted	.....
xgl_lpat_dashed	- - - - - -
xgl_lpat_dashed_dotted	...__...__
xgl_lpat_cgm_dotted	.....
xgl_lpat_cgm_dashed	_ _ _ _ _ _
xgl_lpat_dash_dot	._._._._._
xgl_lpat_dash_dot_dotted	_.._.._.._..
xgl_lpat_long_dashed	__ __ __ __

For example, to specify lines or curves that are dashed, the application sets the line pattern as follows:

```
xgl_object_set(ctx, XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED, 0);
xgl_object_set(ctx, XGL_CTX_LINE_PATTERN, xgl_lpat_dashed, 0);
```

The color of a patterned line is specified by one or both of the two attributes **XGL_CTX_LINE_COLOR**(3) and **XGL_CTX_LINE_ALT_COLOR**(3)**.**

XGL_CTX_LINE_PATTERN applies only to primitives rendering line segments. Surface edges have their own set of attributes.

The default value is xgl_lpat_dotted, a predefined Line Pattern of evenly spaced dots.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

xgl_object_create(3)

XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_STYLE(3)
XGL_LPAT_COORD_SYS(3)
XGL_LPAT_DATA(3)

NAME

XGL_CTX_LINE_STYLE – defines the line style used in a Context

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_LINE_STYLE, Xgl_line_style, 0);
xgl_object_get(ctx, XGL_CTX_LINE_STYLE, Xgl_line_style *);
```

DESCRIPTION

This attribute defines the way in which lines or curves are drawn using the Context *ctx*. XGL_CTX_LINE_STYLE applies only to primitives rendering line segments or curves. Surface edges have their own set of attributes.

XGL recognizes the following values. They are the same as the values recognized for edge style, as defined by the attribute **XGL_CTX_EDGE_STYLE**(3)**.**

XGL_LINE_PATTERNED
: Lines are patterned using the Line Pattern object given by the **XGL_CTX_LINE_-PATTERN**(3) attribute and the color specified by the **XGL_CTX_LINE_COLOR**(3) attribute.

XGL_LINE_ALT_PATTERNED
: Lines are patterned using the Line Pattern object given by the XGL_CTX_LINE_-PATTERN attribute. The color is alternately specified by the XGL_CTX_LINE_COLOR attribute for odd line segments, and by the **XGL_CTX_LINE_ALT_COLOR**(3) attribute for even line segments.

XGL_LINE_SOLID
: Lines are not patterned. A solid line is drawn using the color specified by the XGL_CTX_LINE_COLOR attribute.

When a patterned line is specified, the Context attribute XGL_CTX_LINE_PATTERN is used to specify the Line Pattern. By default, the Line Pattern set in the Context is xgl_lpat_dotted, a predefined Line Pattern of evenly spaced dots.

The default value is XGL_LINE_SOLID.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_object_create(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_PATTERN(3)

NAME

XGL_CTX_LINE_WIDTH_SCALE_FACTOR – defines the line width scale factor used to determine the width of lines and curves

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR, float, 0);
xgl_object_get(ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR, float *);
```

DESCRIPTION

This attribute specifies the scale factor by which a nominal line width is multiplied to produce the line width in device coordinates (DC). This attribute is similar to **XGL_CTX_EDGE_WIDTH_SCALE_FACTOR**(3), which defines an analogous scale factor for XGL surface edges.

When used in conjunction with the attributes **XGL_CTX_LINE_CAP**(3) and **XGL_CTX_LINE_JOIN**(3), XGL_CTX_LINE_WIDTH_SCALE_FACTOR specifies the shape of wide lines.

The line width applies only to primitives rendering line segments. Polygon edges have their own set of attributes.

The default value of 0.0 corresponds to the notion of *thin lines* as defined by the X community. This means the method actually used to scan-convert the lines is device dependent and generally gives the best performance on that device, with a limited set of constraints when drawing 2D vectors (see *The X Window System*, R.W. Scheifler & AL., Digital Press). Any value greater than or equal to 1.0 will guarantee X-compliant lines in 2D. There is no compliance defined for 3D vectors.

In most cases, best results are obtained with an odd value for the scale factor because the resulting width is correctly balanced on both sides of the line. Setting XGL_CTX_LINE_WIDTH_SCALE_FACTOR to a value that produces a width greater than or equal to 1.0 in device coordinates affect drawing performance.

The default value is 0.0.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_JOIN(3)

NAME

XGL_CTX_LOCAL_MODEL_TRANS – defines the local model Transform

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_LOCAL_MODEL_TRANS, Xgl_trans, 0);
xgl_object_get(ctx, XGL_CTX_LOCAL_MODEL_TRANS, Xgl_trans *);
```

DESCRIPTION

This attribute defines the Transform object used for the Local Model Transform, which transforms geometry data from model coordinates (MC) to global model coordinates (GMC).

The Local Model Transform (as defined by this attribute) and the Global Model Transform (as defined by the attribute **XGL_CTX_GLOBAL_MODEL_TRANS**(3)) are multiplied together, with the Local Model Transform preceding the Global Model Transform, to form the Model Transform. The attribute **XGL_CTX_MODEL_TRANS**(3) returns a Transform object that represents the concatenation of the XGL_CTX_LOCAL_MODEL_TRANS and the XGL_CTX_GLOBAL_MODEL_TRANS.

If you set this Context attribute to a Transform whose handle is obtained from the same or a different Context, the original attribute and this attribute use the same Transform. Changing the Transform of one Context attribute has the same effect on the other Context attribute. Be careful of this consequence when you share a Transform in this manner. Do not attempt to set this Context attribute to a handle to a Transform obtained from **XGL_CTX_MODEL_TRANS**(3)**, XGL_CTX_MC_TO_DC_TRANS**(3)**,** or **XGL_3D_CTX_NORMAL_TRANS**(3) because these Transforms are read-only.

The default value is the Identity Transform.

This attribute is pushed by **xgl_context_push**(3)**,** and can also be manipulated with **xgl_context_update_model_trans**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_update_model_trans(3)
XGL_3D_CTX_NORMAL_TRANS(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_MC_TO_DC_TRANS(3)
XGL_CTX_MODEL_TRANS(3)

NAME

XGL_CTX_MARKER – defines the Marker symbol used by the xgl_multimarker operator

SYNOPSIS

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

Xgl_ctx         ctx;

xgl_object_set(ctx, XGL_CTX_MARKER, Xgl_marker, 0);
xgl_object_get(ctx, XGL_CTX_MARKER, Xgl_marker *);
```

DESCRIPTION

This attribute defines the Marker symbol drawn when the operator **xgl_multimarker**(3) is used within the Context *ctx*. For a more complete description of Marker rendering, see **xgl_multimarker**(3)**.**

XGL_CTX_MARKER(3) can be set to point to one of the pre-defined Marker objects (listed below) or to an application-defined Marker object.

The following is a list of pre-defined Marker objects available in XGL. These are declared as external variables of type Xgl_marker in xgl.h.

xgl_marker_dot
: Draws a 1-pixel dot centered at the Marker control point. This Marker ignores the **XGL_CTX_MARKER_SCALE_FACTOR**(3)**;** it *always* draws a 1-pixel dot.

xgl_marker_plus
: Draws a plus sign (+) centered at the Marker control point.

xgl_marker_asterisk
: Draws an asterisk (*) centered at the Marker control point.

xgl_marker_circle
: Draws a circle (○) centered at the Marker control point.

xgl_marker_cross
: Draws a cross (×) centered at the Marker control point.

xgl_marker_square
: Draws a square (□) centered at the Marker control point.

xgl_marker_bowtie_ne
xgl_marker_bowtie_nw
: Draws a bowtie shaped Marker centered at the control point, which is either oriented from northeast to southwest or from northwest to southeast.

To define its own Marker objects, the application:

1. Creates a Marker object using **xgl_object_create**(3)**.**
2. Attaches an application-defined Marker description to that object using the Marker attribute **XGL_MARKER_DESCRIPTION**(3)**.**
3. Uses **XGL_CTX_MARKER**(3) to attach the application-defined Marker to the current Context.

Then, a call to **xgl_multimarker**(3) renders the application-defined Markers into an XGL Raster.

If the application data associated with this Marker object is a pointer to a floating point scale factor, when rendering through the CGM stream pipeline, the marker will be stroked out and scaled by a factor equal to the product of **XGL_CTX_MARKER_SCALE_-FACTOR**(3) and the float pointed to by **XGL_OBJ_APPLICATION_DATA**(3). If there is no object application data associated with the marker, then a marker other than one of the predefined markers will be rendered by the CGM stream pipeline as a dot marker. Note that the **XGL_OBJ_APPLICATION_DATA**(3) scale factor only applies to the CGM stream pipeline. See the *XGL Programmer's Guide* for an example.

The default value is xgl_marker_plus.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multimarker(3)

XGL_CTX_MARKER_COLOR(3)
XGL_CTX_MARKER_COLOR_SELECTOR(3)
XGL_CTX_MARKER_SCALE_FACTOR(3)
XGL_MARKER_DESCRIPTION(3)

NAME

XGL_CTX_MARKER_AA_BLEND_EQ,
XGL_CTX_MARKER_AA_FILTER_WIDTH,
XGL_CTX_MARKER_AA_FILTER_SHAPE – controls if and how markers are antialiased

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_MARKER_AA_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_set(ctx, XGL_CTX_MARKER_AA_FILTER_WIDTH, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_MARKER_AA_FILTER_SHAPE, Xgl_filter_shape, 0);

xgl_object_get(ctx, XGL_CTX_MARKER_AA_BLEND_EQ, Xgl_blend_eq *);
xgl_object_get(ctx, XGL_CTX_MARKER_AA_FILTER_WIDTH, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_MARKER_AA_FILTER_SHAPE, Xgl_filter_shape *);
```

DESCRIPTION

These attributes control if and how markers are antialiased. Antialiasing smooths out single-pixel-wide markers by spreading out the spectral energy of the stroke or dot over 2 or more adjacent pixels. The result is that the sides of the stroke or dot are slightly blurred and, as seen from a normal viewing distance, smoother than non-antialiased markers.

The blurring of the marker is done by filtering the stroke or dot onto neighboring pixels. The shape and size of this filter controls how the antialiased marker will look. Currently, XGL_FILTER_GAUSSIAN is the only possible value for the **XGL_CTX_MARKER_AA_FILTER_SHAPE**(3) attribute.

The width of the filter is controlled by the attribute **XGL_CTX_MARKER_AA_FILTER_WIDTH**(3)**.** This is the number of pixels touched by the filter. A Device will use the largest filter width it supports which is less than or equal to the specified filter width. A width of 0 is not an error and is treated as if the width was 1.

The attribute **XGL_CTX_MARKER_AA_BLEND_EQ**(3) controls how the pixel values in the antialiased marker are blended with the existing pixel values in the Device. The data type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that antialiasing use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_-CONST_BG.

All three attributes must have consistent settings in order to achieve visually pleasing results. Some notes on using these attributes:

- Both the XGL_CTX_MARKER_AA_BLEND_EQ and the XGL_CTX_MARKER_AA_-FILTER_WIDTH must be set to something other than defaults to achieve antialiasing.
- Setting the XGL_CTX_MARKER_AA_BLEND_EQ to XGL_BLEND_NONE and the XGL_-CTX_MARKER_AA_FILTER_WIDTH to greater than 1 is a way to *undraw* antialiased markers. The color used for undrawing is the color that the marker would have been drawn in if antialiasing was off.
- Setting the XGL_CTX_MARKER_AA_BLEND_EQ to other than NONE and the XGL_-CTX_MARKER_AA_FILTER_WIDTH to 1 is undefined and should not be used. The results will be device dependent.

The default value for XGL_CTX_MARKER_AA_BLEND_EQ is XGL_BLEND_NONE.

The default value for XGL_CTX_MARKER_AA_FILTER_WIDTH is 1.

The default value for XGL_CTX_MARKER_AA_FILTER_SHAPE is XGL_FILTER_GAUSSIAN.

All these attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTES

Antialiasing support on 2D Contexts is device dependent.

NAME

XGL_CTX_MARKER_COLOR – defines color of Markers in a Context

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_MARKER_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_MARKER_COLOR, Xgl_color *);
```

DESCRIPTION

Specifies the color in which Markers are drawn within the Context *ctx*.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multimarker(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_MARKER_COLOR_SELECTOR – selects the source of a Marker's color from the Context or from the point data

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx                 ctx;
xgl_object_set(ctx, XGL_CTX_MARKER_COLOR_SELECTOR, Xgl_marker_color_sel, 0);
xgl_object_get(ctx, XGL_CTX_MARKER_COLOR_SELECTOR, Xgl_marker_color_sel *);
```

DESCRIPTION

This attribute selects the source of a Marker's color from the Context or the point data provided with a multimarker primitive.

XGL_CTX_MARKER_COLOR_SELECTOR(3) specifies the color selection used for Markers in the 2D or 3D Context *ctx*. Two options are available for this attribute:

XGL_MARKER_COLOR_CONTEXT
: XGL takes the Marker color from the context attribute XGL_CTX_MARKER_COLOR. This is the color of a Marker at the beginning of the rendering pipeline. Depth-cueing affects the final appearance of the Marker in 3D.

XGL_MARKER_COLOR_POINT
: If the data supplied with a multimarker primitive has colors for the individual points, XGL applies these colors to the points upon entry to the rendering pipeline. Depth-cueing affects the final appearance of the Marker in 3D. If the multimarker primitive does not have colors for the individual points, XGL uses the Context Marker color.

The default value is XGL_MARKER_COLOR_POINT.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multimarker(3)
XGL_CTX_MARKER_COLOR(3)

NAME

XGL_CTX_MARKER_SCALE_FACTOR – defines the Marker scale factor used to determine the extent of Markers

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_MARKER_SCALE_FACTOR, float, 0);
xgl_object_get(ctx, XGL_CTX_MARKER_SCALE_FACTOR, float *);
```

DESCRIPTION

This attribute specifies a scale factor by which the nominal Marker extent of the Device (which is 1 pixel for Raster devices) is multiplied to produce the Marker extent in device coordinates (DC).

Note that the dot Marker (specified by setting **XGL_CTX_MARKER**(3) to xgl_marker_dot) is not affected by **XGL_CTX_MARKER_SCALE_FACTOR**(3)**.**

The default value is 1.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multimarker(3)
XGL_CTX_MARKER(3)

NAME

XGL_CTX_MAX_TESSELLATION – defines the maximum tessellation of a circle, arc, ellipse, curve, or surface

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_MAX_TESSELLATION, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_CTX_MAX_TESSELLATION, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is used to set the absolute maximum number of segments in which a 3D conic curve, a NURBS curve, or a NURBS surface can be tessellated when using the following XGL operators:

xgl_multicircle(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_nurbs_curve(3)
xgl_nurbs_surface(3)

The method specified by **XGL_CTX_NURBS_CURVE_APPROX**(3) or **XGL_CTX_NURBS_SURF_APPROX**(3)**,** as well as the value given by **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3)**, XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)**, or **XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)**, is used to compute the number of interpolated segments along the curve or along the isoparametric directions of the surface. In the case of NURBS curve or surface, the number of segments refers to the number of segments between pairs of adjacent knots. This number is then truncated to be less than or equal to XGL_CTX_MAX_TESSELLATION. Thus, XGL_CTX_MAX_TESSELLATION provides a trade-off between curve or surface quality and computation time.

XGL_CTX_MIN_TESSELLATION(3) provides a way to force the number of segments to an absolute minimum.

The default value is 64.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_multicircle(3)
xgl_nurbs_curve(3)
xgl_nurbs_surface(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)

XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)

NAME

XGL_CTX_MC_TO_DC_TRANS – returns the model-coordinate-to-device-coordinate Transform

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_get(ctx, XGL_CTX_MC_TO_DC_TRANS, Xgl_trans *);
```

DESCRIPTION

This read-only attribute returns the Transform object that maps geometry from model coordinates (MC) to device coordinates (DC).

The MC-to-DC Transform results from the concatenation of the the Model Transform **XGL_CTX_MODEL_TRANS**(3), the View Transform **XGL_CTX_VIEW_TRANS**(3), and the VDC-to-DC Transform, which XGL calculates internally from several attributes: **XGL_CTX_VDC_MAP**(3), **XGL_CTX_VDC_ORIENTATION**(3), **XGL_CTX_VDC_WINDOW**(3), **XGL_CTX_DC_VIEWPORT**(3), **XGL_DEV_MAXIMUM_COORDINATES**(3), and the Device's DC orientation. For 3D, the VDC-to-DC Transform also depends on **XGL_3D_CTX_JITTER_OFFSET**(3).

XGL defers evaluation of the MC-to-DC Transform until either XGL needs this Transform internally or the application explicitly gets it. Before attempting to use the MC-to-DC Transform, call **xgl_object_get**(3) to get XGL_CTX_MC_TO_DC_TRANS to ensure XGL evaluates the MC-to-DC Transform if it requires updating.

This attribute is read-only. To make changes to it, you must modify the following attributes:

XGL_CTX_LOCAL_MODEL_TRANS(3),
XGL_CTX_GLOBAL_MODEL_TRANS(3),
XGL_CTX_VIEW_TRANS(3),
XGL_CTX_VDC_MAP(3),
XGL_CTX_VDC_ORIENTATION(3),
XGL_CTX_VDC_WINDOW(3),
XGL_CTX_DC_VIEWPORT(3),
XGL_3D_CTX_JITTER_OFFSET(3).

XGL will not permit attempts to change this attribute directly with Transform operators.

Since this attribute is read-only, do not get a handle to this Transform and use it to set **XGL_CTX_LOCAL_MODEL_TRANS**(3), **XGL_CTX_GLOBAL_MODEL_TRANS**(3), or **XGL_CTX_VIEW_TRANS**(3)

The Transform of this read-only Context attribute always has **XGL_TRANS_DATA_TYPE**(3) set to XGL_DATA_DBL.

The default value is the Identity Transform.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_VIEW_TRANS(3)
XGL_CTX_VDC_MAP(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VDC_WINDOW(3)
XGL_CTX_DC_VIEWPORT(3)
XGL_DEV_MAXIMUM_COORDINATES(3)
XGL_TRANS_DATA_TYPE(3)
XGL_3D_CTX_JITTER_OFFSET(3)

NAME

XGL_CTX_MIN_TESSELLATION – defines the minimum tessellation of a circle, arc, ellipse, curve, or surface

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_MIN_TESSELLATION, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_CTX_MIN_TESSELLATION, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is used to set the absolute minimum number of segments in which a 3D conic curve, a NURBS curve, or a NURBS surface can be tessellated when using the following XGL operators:

xgl_multicircle(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_nurbs_curve(3)
xgl_nurbs_surface(3)

The method specified by **XGL_CTX_NURBS_CURVE_APPROX**(3) or **XGL_CTX_NURBS_SURF_APPROX**(3), as well as the value given by **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3), **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3), or **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3), is used to compute the number of interpolated segments along the curve or along the isoparametric directions of the surface. In the case of NURBS curve or surface, the number of segments refers to the number of segments between pairs of adjacent knots. This number is then adjusted to be greater than or equal to XGL_CTX_MIN_TESSELLATION. Thus, XGL_CTX_MIN_TESSELLATION provides a trade-off between curve or surface quality and computation time.

XGL_CTX_MAX_TESSELLATION(3) provides a way to limit the number of segments to an absolute maximum.

The default value is 1.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_multicircle(3)
xgl_nurbs_curve(3)
xgl_nurbs_surface(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)

XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)

NAME

XGL_CTX_MODEL_TRANS – returns the Model Transform

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_get(ctx, XGL_CTX_MODEL_TRANS, Xgl_trans *);
```

DESCRIPTION

This read-only attribute returns the Transform object that represents the Model Transform, which maps geometry from model coordinates (MC) to world coordinates (WC).

The Model Transform results from the concatenation of the Local Model Transform (as defined by the attribute **XGL_CTX_LOCAL_MODEL_TRANS**(3)) and the Global Model Transform (as defined by the attribute **XGL_CTX_GLOBAL_MODEL_TRANS**(3)), respectively. This attribute (the Model Transform) is affected by changes to either **XGL_CTX_LOCAL_-MODEL_TRANS**(3) or **XGL_CTX_GLOBAL_MODEL_TRANS**(3)**.**

To improve performance in XGL 3.0 and subsequent releases, evaluation of the Model Transform is deferred until either XGL needs the Model Transform internally or the application explicitly gets it. Before attempting to use the Model Transform, call **xgl_object_get**(3) to get XGL_CTX_MODEL_TRANS to ensure XGL evaluates the Model Transform if it requires updating.

This attribute is read-only. To make changes to it, you must modify XGL_CTX_LOCAL_-MODEL_TRANS or XGL_CTX_GLOBAL_MODEL_TRANS. Do not get this attribute and use a Transform operator on it if that operator changes the value of the Model Transform. Do not perform an **xgl_object_set**(3) on this attribute. Instead, use XGL_CTX_LOCAL_MODEL_-TRANS or XGL_CTX_GLOBAL_MODEL_TRANS. Attempts to explicitly change the Model Transform will result in an error.

Since this attribute is read-only, do not get a handle to this Transform and use it to set **XGL_CTX_LOCAL_MODEL_TRANS**(3)**,** **XGL_CTX_GLOBAL_MODEL_TRANS**(3)**,** or **XGL_CTX_VIEW_-TRANS**(3)**.**

The Transform of this read-only Context attribute always has **XGL_TRANS_DATA_TYPE**(3) set to XGL_DATA_DBL.

The default value is the Identity Transform.

This attribute is not pushed by **xgl_context_push**(3)**,** but the Model Transform can change because of some manipulations done with **xgl_context_update_model_trans**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_update_model_trans(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_VIEW_TRANS(3)
XGL_TRANS_DATA_TYPE(3)

NAME

XGL_CTX_MODEL_TRANS_STACK_SIZE – defines the size of the Context's Transform stack

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_MODEL_TRANS_STACK_SIZE, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_CTX_MODEL_TRANS_STACK_SIZE, Xgl_usgn32 *);
```

DESCRIPTION

This attribute defines the depth of the Transform stack associated with the Context *ctx*, expressed in number of Transforms, for use with the **xgl_context_update_model_trans**(3) operator.

Each XGL Context defines a special stack dedicated to pushing and popping the Local Model Transform or the Global Model Transform with the **xgl_context_update_model_trans**(3) operator in order to get the best possible performance.

Updating the value of XGL_CTX_MODEL_TRANS_STACK_SIZE will cause the contents of the current stack (if any) to be lost.

The default value is 0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_update_model_trans(3)

NAME

XGL_CTX_NEW_FRAME_ACTION – sets the actions performed by the operator xgl_context_new_frame

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_NEW_FRAME_ACTION, Xgl_ctx_new_frame_action, 0);
xgl_object_get(ctx, XGL_CTX_NEW_FRAME_ACTION, Xgl_ctx_new_frame_action *);

DESCRIPTION

This Context attribute is used to indicate the actions to be performed by the XGL operator **xgl_context_new_frame**(3) on the Device attached to the Context *ctx*. The purpose of xgl_context_new_frame is to clear the Device attached to the current Context.

The value of this attribute is the combination by logical OR of the new frame actions listed below. The available new frame actions are specified by the enumerated data type *Xgl_-ctx_new_frame_action*.

Possible actions are:

XGL_CTX_NEW_FRAME_NONE
: Nothing happens. In this mode, xgl_context_new_frame is equivalent to a no-operation.

XGL_CTX_NEW_FRAME_VRETRACE
: Wait for the frame buffer to enter vertical retrace before executing the xgl_-context_new_frame operation. This is useful for color-map double buffering, or when an application needs to synchronize on the device's vertical retrace.

XGL_CTX_NEW_FRAME_CLEAR
: The action taken by xgl_context_new_frame when this flag is set depends on the value of the attribute **XGL_CTX_VDC_MAP**(3)**.** When XGL_CTX_VDC_MAP is XGL_CTX_-VDC_MAP_ALL, XGL_CTX_VDC_MAP_ASPECT, or XGL_CTX_VDC_MAP_DEVICE, xgl_-context_new_frame clears the entire drawing area associated with the Device. When XGL_CTX_VDC_MAP is XGL_CTX_VDC_MAP_OFF, it clears only the device coordinate viewport (see **XGL_CTX_DC_VIEWPORT**(3)) portion of the Device. The cleared area is set to the color given by the Context attribute **XGL_CTX_-BACKGROUND_COLOR**(3)**.** When the Device is cleared, the **XGL_CTX_ROP**(3) attribute is ignored and the Raster operation (ROP) is forced to XGL_ROP_COPY.

XGL_CTX_NEW_FRAME_HLHSR_ACTION
: Perform the HLHSR (hidden line, hidden surface removal) action associated with the current value of the Context attribute **XGL_3D_CTX_HLHSR_MODE**(3)**,** using the current **XGL_3D_CTX_HLHSR_DATA**(3) value.

XGL_CTX_NEW_FRAME_SWITCH_BUFFER
: This flag is used to switch both the values of **XGL_WIN_RAS_BUF_DRAW**(3) and **XGL_WIN_RAS_BUF_DISPLAY**(3) and the actual hardware buffers. Before the switch, XGL will wait for the current display buffer to be displayed in its entirety. Setting XGL_CTX_NEW_FRAME_VRETRACE is not necessary to avoid visual artifacts and may reduce performance. If this flag is set, **xgl_context_new_frame**(3) sets XGL_WIN_-

RAS_BUF_DRAW to the prior value of XGL_WIN_RAS_BUF_DISPLAY, and it sets XGL_-WIN_RAS_BUF_DISPLAY to the prior value of XGL_WIN_RAS_BUF_DRAW.

To initialize XGL_WIN_RAS_BUF_DISPLAY and XGL_WIN_RAS_BUF_DRAW to different values, use **xgl_object_set**(3)**.** If their initial values do not differ, setting the flag XGL_CTX_NEW_FRAME_SWITCH_BUFFER has no effect.

To allocate multiple buffers, request the desired number of buffers by setting **XGL_WIN_RAS_BUFFERS_REQUESTED**(3)**.** Inquire the number of buffers actually allocated via **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3)**.**

If both XGL_CTX_NEW_FRAME_CLEAR and this attribute are set, the new display buffer is visible before the new draw buffer is cleared.

After a buffer switch where XGL_WIN_RAS_BUF_DRAW is not equal to XGL_WIN_-RAS_BUF_DISPLAY, the contents of XGL_WIN_RAS_BUF_DRAW are device-dependent unless the following flags are set:

- XGL_CTX_NEW_FRAME_CLEAR
- XGL_CTX_NEW_FRAME_HLHSR_ACTION (if *z*-buffering is being used)

This device dependency falls into two categories:

- After the switch, the pixels in the XGL_WIN_RAS_BUF_DISPLAY are equivalent to the pixels in the WIN_RAS_BUF_DRAW.
- Both buffer contents are unchanged. To determine the type of your device, check the **xgl_inquire**(3) operator and your accelerator guide.

The default value is XGL_CTX_NEW_FRAME_CLEAR.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_new_frame(3)
XGL_3D_CTX_HLHSR_DATA(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_DC_VIEWPORT(3)
XGL_CTX_ROP(3)
XGL_CTX_VDC_MAP(3)
XGL_WIN_RAS_BUFFERS_REQUESTED(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)

NAME

XGL_CTX_NEW_FRAME_PAINT_TYPE – specifies the new frame paint type for transparent overlays

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_CTX_NEW_FRAME_PAINT_TYPE, Xgl_paint_type, 0);
xgl_object_get(ctx, XGL_CTX_NEW_FRAME_PAINT_TYPE, Xgl_paint_type*);
```

DESCRIPTION

This attribute specifies the paint type for transparent overlays when an application calls **xgl_context_new_frame**(3) and **XGL_CTX_NEW_FRAME_ACTION**(3) with action XGL_CTX_NEW_FRAME_CLEAR. In this case, the color information is set to the background color and the paint type is set to the value of this attribute. The possible values are XGL_PAINT_TRANSPARENT or XGL_PAINT_OPAQUE .

The default value is XGL_PAINT_TRANSPARENT.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_new_frame(3)
XGL_CTX_NEW_FRAME_ACTION(3)
XGL_CTX_PAINT_TYPE(3)

NAME

XGL_CTX_NURBS_CURVE_APPROX – defines the curve approximation method

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_NURBS_CURVE_APPROX, Xgl_curve_approx, 0);
xgl_object_get(ctx, XGL_CTX_NURBS_CURVE_APPROX, Xgl_curve_approx *);
```

DESCRIPTION

This attribute defines the approximation method used to render 3D conic or NURBS curves when using the following operators:

xgl_multicircle(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_nurbs_curve(3)

The alternatives for this attribute are specified by the enumerated data type *Xgl_curve_approx* and are described below:

XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS

Ensures that the number of segments used to draw the curve between each pair of adjacent knots will be at least equal to the value of **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3) attribute.

XGL_CURVE_METRIC_WC
XGL_CURVE_METRIC_VDC
XGL_CURVE_METRIC_DC

Ensures that the distance between two consecutive points on the curve will be, at most, equal to the value of **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3) attribute. The distance is expressed in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC).

XGL_CURVE_CHORDAL_DEVIATION_WC
XGL_CURVE_CHORDAL_DEVIATION_VDC
XGL_CURVE_CHORDAL_DEVIATION_DC

Ensures that the distance between the curve and the chord between two consecutive evaluated points on the curve will be, at most, equal to the value of **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3) attribute. The distance is expressed in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC).

XGL_CURVE_RELATIVE_WC
XGL_CURVE_RELATIVE_VDC
XGL_CURVE_RELATIVE_DC

A value of **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3) attribute between 0 and 1 indicates the relative quality of the rendering to be maintained in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC). Higher values mean better relative quality.

The **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes ensure the absolute minimum and maximum number of segments in which a curve will be rendered.

The default value is XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_multicircle(3)
xgl_nurbs_curve(3)

XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)

NAME

XGL_CTX_NURBS_CURVE_APPROX_VAL – defines the curve approximation value

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_NURBS_CURVE_APPROX_VAL, float, 0);
xgl_object_get(ctx, XGL_CTX_NURBS_CURVE_APPROX_VAL, float *);
```

DESCRIPTION

This attribute is used to set the approximation quality of XGL 3D conic or NURBS curves when using the following operators:

xgl_multicircle(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_nurbs_curve(3)

Depending on the **XGL_CTX_NURBS_CURVE_APPROX**(3) attribute, XGL_CTX_NURBS_CURVE_-APPROX_VAL represents a number of subdivisions, maximum distance between two points, maximum deviation between the curve and the approximation, or a relative quality measure.

The **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes ensure the absolute minimum and maximum number of segments in which a curve will be rendered.

The default value is 10.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multiarc(3)
xgl_multi_elliptical_arc(3)
xgl_multicircle(3)
xgl_nurbs_curve(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX(3)

NAME

XGL_CTX_NURBS_SURF_APPROX – defines the surface approximation method

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_NURBS_SURF_APPROX, Xgl_surf_approx, 0);
xgl_object_get(ctx, XGL_CTX_NURBS_SURF_APPROX, Xgl_surf_approx *);
```

DESCRIPTION

This attribute defines the approximation method used to render NURBS surfaces when using the **xgl_nurbs_surface**(3) operator.

The alternatives for this attribute are specified by the enumerated data type *Xgl_surf_approx* and are described below:

XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS

Ensures that the number of segments used to display the surface between each pair of adjacent knots in U and V parametric direction will be at least equal to the respective values of **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3) and **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3) attributes.

XGL_SURF_METRIC_WC
XGL_SURF_METRIC_VDC
XGL_SURF_METRIC_DC

Ensures that the distance between two consecutive points on the surface in U and V parametric directions will be, at most, equal to the respective values of **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3) and **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3) attributes. The distance is expressed in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC).

XGL_SURF_PLANAR_DEVIATION_WC
XGL_SURF_PLANAR_DEVIATION_VDC
XGL_SURF_PLANAR_DEVIATION_DC

Ensures that the distance between the surface and the approximating facet will be, at most, equal to the value of **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3) attribute. The distance is expressed in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC). The **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3) attribute is ignored.

XGL_SURF_RELATIVE_WC
XGL_SURF_RELATIVE_VDC
XGL_SURF_RELATIVE_DC

A value of **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3) attribute between 0 and 1 indicates the relative quality of the rendering to be maintained in world coordinates (WC), virtual device coordinates (VDC), or in device coordinates (DC). Higher values mean better relative quality. The **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3) attribute is ignored.

The **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes ensure the absolute minimum and maximum number of segments in which a curve will be rendered.

The default value is XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_nurbs_surface(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)

NAME

XGL_CTX_NURBS_SURF_APPROX_VAL_U,
XGL_CTX_NURBS_SURF_APPROX_VAL_V – define the surface approximation values

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_NURBS_SURF_APPROX_VAL_U, float, 0);
xgl_object_set(ctx, XGL_CTX_NURBS_SURF_APPROX_VAL_V, float, 0);

xgl_object_get(ctx, XGL_CTX_NURBS_SURF_APPROX_VAL_U, float *);
xgl_object_get(ctx, XGL_CTX_NURBS_SURF_APPROX_VAL_V, float *);
```

DESCRIPTION

These attributes are used to set the approximation quality of XGL NURBS surfaces when using the **xgl_nurbs_surface**(3) operator.

Depending on the **XGL_CTX_NURBS_SURF_APPROX**(3) attribute, XGL_CTX_NURBS_SURF_-APPROX_VAL_U and XGL_CTX_NURBS_SURF_APPROX_VAL_V represent a number of subdivisions, maximum distance between two points in U or V parametric directions, maximum deviation between the surface and the approximation, or a relative quality measure.

The **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes ensure the absolute minimum and maximum number of segments in each parametric direction in which a surface will be rendered.

The default value is 10.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_nurbs_surface(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)

NAME

XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT – defines the placement of isoparametric lines on a NURBS surface

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT,
        Xgl_iso_curve_placement, 0);
xgl_object_get(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT,
        Xgl_iso_curve_placement *);
```

DESCRIPTION

This attribute defines the placement of isoparametric curves in U and V parametric directions to draw on the NURBS surface when using **xgl_nurbs_surface**(3) operator. The **XGL_CTX_NURBS_SURF_PARAM_STYLE**(3) attribute must be set to or be combined by logical OR with the XGL_SURF_ISO_CURVES value.

The alternatives for this attribute are specified by the enumerated data type *Xgl_iso_curve_placement* and are described below:

XGL_ISO_CURVE_BETWEEN_KNOTS
: The specified number of isoparametric curves will be drawn between each adjacent pair of knots.

XGL_ISO_CURVE_BETWEEN_LIMITS
: The specified number of isoparametric curves will be drawn between the parametric limits of the whole surface.

The number of isoparametric curves is specified by the **XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM**(3) and **XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM**(3) attributes.

The default value is XGL_ISO_CURVE_BETWEEN_KNOTS.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_nurbs_surface(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(3)

NAME

XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM – define the number of isoparametric curves on a NURBS surface

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM, Xgl_usgn32, 0);

xgl_object_get(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM, Xgl_usgn32 *);
```

DESCRIPTION

These attributes are used to set the number of isoparametric curves in U and V parametric directions to draw on the NURBS surface when using **xgl_nurbs_surface**(3) operator. The **XGL_CTX_NURBS_SURF_PARAM_STYLE**(3) attribute must be set to or be OR'd with the XGL_SURF_ISO_CURVES value.

Depending on the **XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT**(3) attribute, specified number of isoparametric lines is drawn between adjacent knots or between the parametric limits of the surface.

The default value is 0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_nurbs_surface(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)

NAME

XGL_CTX_NURBS_SURF_PARAM_STYLE – defines the appearance of the NURBS surface

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_NURBS_SURF_PARAM_STYLE , Xgl_nurbs_surf_param_style, 0);
xgl_object_get(ctx, XGL_CTX_NURBS_SURF_PARAM_STYLE , Xgl_nurbs_surf_param_style *);
```

DESCRIPTION

This attribute defines the appearance of the NURBS surface when using the **xgl_nurbs_surface**(3) operator.

The alternatives for this attribute are specified by the enumerated data type *Xgl_nurbs_surf_param_style* and are described below:

XGL_SURF_PLAIN
: No extra lines or edges will be displayed on the surface.

XGL_SURF_ISO_CURVES
: Isoparametric curves are drawn on the surface. The number of these curves in U and V directions and their placement is specified by the **XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM**(3), **XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM**(3), and **XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT**(3) attributes. Isoparametric curves are drawn using current line attributes.

XGL_SURF_SHOW_TESSELLATION
: Edges of all facets that approximate the surface are shown. Edges are shown using current edge attributes.

XGL_SURF_INCR_SILHOUETTE_TESS
: For better rendering quality, surface is tessellated more finely near areas of silhouette edges. This value is independent of **XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG**(3), which controls the drawing of the silhouette edges themselves. Also, this value is valid only for dynamic surface approximation types (i.e. for XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS type tessellation remains unchanged).

XGL_SURF_DEVICE_DEPENDENT
: This value will ensure best performance. No extra lines or edges will be displayed on the surface.

All values, except for XGL_SURF_DEVICE_DEPENDENT can be OR'd together for simultaneous display.

The default value is XGL_SURF_PLAIN.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_nurbs_surface(3)

XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(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_COLOR(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_3D_CTX_LINE_COLOR_INTERP(3)

NAME

XGL_CTX_PAINT_TYPE – specifies the paint type for transparent overlays

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_CTX_PAINT_TYPE, Xgl_paint_type, 0);
xgl_object_get(ctx, XGL_CTX_PAINT_TYPE, Xgl_paint_type*);
```

DESCRIPTION

This attribute specifies the paint type for transparent overlays. The possible values are XGL_PAINT_TRANSPARENT or XGL_PAINT_OPAQUE.

This attribute has effect only when the Device associated with a Context is an overlay window. Otherwise, it is ignored and the paint type is *opaque*.

Most XGL primitives will render pixels into overlay windows with the paint type given by the value of this attribute. The following is a list of exceptions:

- xgl_context_accumulate() -- Only the color information of the draw buffer is accumulated into an overlay window that is the destination buffer of the accumulation. The paint type is not changed.
- xgl_context_copy_buffer() -- If the device associated with the Context is an overlay window, the color information of the source raster is copied to the destination. If the source raster is an overlay window, the paint type is copied to the destination. Otherwise, the paint type of the destination is not changed.
- xgl_context_new_frame() -- If **XGL_CTX_NEW_FRAME_ACTION**(3) includes XGL_CTX_NEW_FRAME_CLEAR, then the color information is set to the background color and the paint type is set to the value of **XGL_CTX_NEW_FRAME_PAINT_TYPE**(3)**.**

The default value is XGL_PAINT_OPAQUE.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_CTX_NEW_FRAME_ACTION(3)
XGL_CTX_NEW_FRAME_PAINT_TYPE(3)

NAME

XGL_CTX_PICK_APERTURE – defines the area (2D) or volume (3D) used for picking tests

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

For a 2D Context:
xgl_object_set(ctx, XGL_CTX_PICK_APERTURE, Xgl_bounds_d2d *, 0);
xgl_object_get(ctx, XGL_CTX_PICK_APERTURE, Xgl_bounds_d2d *);

For a 3D Context:
xgl_object_set(ctx, XGL_CTX_PICK_APERTURE, Xgl_bounds_d3d *, 0);
xgl_object_get(ctx, XGL_CTX_PICK_APERTURE, Xgl_bounds_d3d *);

DESCRIPTION

This attribute defines the XGL *pick aperture*, which is the area (for a 2D Context) or the volume (for a 3D Context) where, to be picked, some part of the primitive must be located. Only one pick aperture can be active at any one time. The bounds of the aperture are specified in device coordinates (DC).

For a primitive to generate a pick event, the following conditions must be met:

- A pickable part of the primitive must be in the pick aperture (see discussion below).
- Any portion of the pickable part must pass the HLHSR test, if *z*-buffering is enabled. (Does not apply to 2D.) That is, only objects that are not completely obscured by closer objects can be picked.
- The current pick IDs must differ from the last entry in the pick buffer.

What Is a Pickable Part of a Primitive?

When point flag information is present for stroke rendering, only the enabled strokes are pickable (device-independent). The pickable part of a stroke, however, is device-dependent. (*Stroke* means any line-like primitive — for example, lines, edges, stroke text, annotation text, hollow polygons, markers, NURBS curve, and open arcs.)

When XGL_CTX_PICK_SURF_STYLE is XGL_PICK_SURF_AS_SOLID, all of the surface is pickable.

When XGL_CTX_PICK_SURF_STYLE is XGL_PICK_SURF_AS_FILL_STYLE, the pickable part of the surface is device-dependent for fill styles hollow and stipple. Please see your accelerator guide.

There are two kinds of device-dependent semantics: geometry and rasterization.

Geometry Semantic

In the geometry picking semantics, the pickable part is the geometric description. For lines and edges, for example, their mathematical center line is used. The line or edge style, cap, join, pattern, width, and anti-aliasing have no effect on picking. For hollow polygons, the bounding lines are picked. The surface stipple fill style picks no differently than the solid fill style does. The geometry semantic is preferred for performance reasons on devices that have no hardware picking support.

Rasterization Semantic

In the rasterization picking semantics, the pickable part is whatever would be rendered. The rasterization semantic is preferred for performance reasons on devices that provide hardware picking support.

Some examples comparing the two approaches appear below. Assume for this chart that the XGL_CTX_PICK_SURF_STYLE is XGL_PICK_SURF_AS_FILL_STYLE.

OBJECT	PICK APERTURE IS PLACED EXCLUSIVELY ON...	PICKED? GEOMETRY	RASTERIZATION
patterned stroke	Off part of line pattern	Y	N
wide stroke	Not over center line, over wide pixels	N	Y
anti-aliased stroke	Not over center line, just over outermost anti-aliased pixel	N	Y
stippled polygon	Part of polygon where stipple pattern is OFF	Y	N
any geometry where rop is XGL_ROP_NOOP or plane mask is 0	Over a pickable part of prim	Y	N
empty polygon	Over a pickable part of prim	N	N

When a *pick event* occurs, XGL stores a structure of type *Xgl_pick_info* into the Context *pick buffer*. The *Xgl_pick_info* structure contains the current values of the pick identifiers, **XGL_CTX_PICK_ID_1**(3) and **XGL_CTX_PICK_ID_2**(3), associated with the primitive that was picked. The value of the vertex flag copied into the *Xgl_pick_info* structure is undefined. The size of the pick buffer is specified by **XGL_CTX_PICK_BUFFER_SIZE**(3). *Xgl_pick_info* data are added to the pick buffer according to the method specified by **XGL_CTX_PICK_STYLE**(3).

The attribute **XGL_CTX_PICK_ENABLE**(3) must be TRUE for picking to occur.

The default value for a 2D Context is $[-1,1] \times [-1,1]$, and for a 3D Context it is $[-1,1] \times [-1,1] \times [-1,1]$.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)
XGL_CTX_PICK_SURF_STYLE(3)

NAME

XGL_CTX_PICK_BUFFER_SIZE – defines the size of the pick buffer used to store information about pick events

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_PICK_BUFFER_SIZE, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_CTX_PICK_BUFFER_SIZE, Xgl_usgn32 *);
```

DESCRIPTION

This attribute sets the size of the Context *pick buffer,* which stores information about pick events. **XGL_CTX_PICK_BUFFER_SIZE**(3) is the number of variables of type *Xgl_pick_info* that the Context pick buffer holds. Each time a primitive passes through the pick aperture specified by the attribute **XGL_CTX_PICK_APERTURE**(3)**,** a variable of type *Xgl_pick_info* is added to the pick buffer. *Xgl_pick_info* data are added to the pick buffer according to the method specified by **XGL_CTX_PICK_STYLE**(3)**.** If the buffer overflows, the oldest pick information may be dropped, or the new pick information is ignored, depending on the setting of XGL_CTX_PICK_STYLE.

When this attribute is modified, the pick buffer is cleared. It is an error to modify this attribute when XGL_CTX_PICK_ENABLE is TRUE. If this error occurs, the attribute is not modified.

The default value is 256.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)

NAME

XGL_CTX_PICK_ENABLE – controls whether picking is done

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_PICK_ENABLE, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_CTX_PICK_ENABLE, Xgl_boolean *);
```

DESCRIPTION

This attribute either turns picking on (when TRUE) or off (when FALSE). For a description of picking in XGL, see the attribute **XGL_CTX_PICK_ID_1**(3)**.**

When picking is enabled, rendering is automatically disabled.

Picking applies to all rendering primitives except the following:

xgl_context_clear_accumulation
xgl_context_accumulate
xgl_context_copy_buffer
xgl_context_set_pixel
xgl_context_new_frame
xgl_context_set_multi_pixel
xgl_context_set_pixel_row
xgl_image

When this attribute is set to TRUE, all of the above primitives are treated as non-operations (NO-OPS).

The default value is FALSE.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)

NAME

XGL_CTX_PICK_ID_1,
XGL_CTX_PICK_ID_2 – define the pick identifiers used to identify primitives that have been picked

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_PICK_ID_1, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_PICK_ID_2, Xgl_usgn32, 0);

xgl_object_get(ctx, XGL_CTX_PICK_ID_1, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_PICK_ID_2, Xgl_usgn32 *);
```

DESCRIPTION

These two attributes define XGL's pick identifiers, and they are used to identify primitives that are picked. When the picking part of the graphics pipeline is enabled (using the attribute **XGL_CTX_PICK_ENABLE**(3)), any primitive sent down the pipeline to be rendered will be checked to see if any part of it is inside the pick aperture. If it is, a pick event is generated. Note that you can only pick without changing the contents of the display. That is, by setting **XGL_CTX_PICK_ENABLE**(3) to TRUE, XGL will disable the rendering engine. XGL stores picking information into the Context *pick buffer*. This information includes the current values of **XGL_CTX_PICK_ID_1**(3) and **XGL_CTX_PICK_ID_2**(3)**.** Thus, the application can identify which primitives were picked depending on how the pick IDs have been set before each primitive rendering operation.

The operator **xgl_pick_get_identifiers**(3) is used to retrieve pick IDs of picked primitives. The Context attribute **XGL_CTX_PICK_STYLE**(3) controls how the pick IDs are stored in the pick buffer and the order in which the IDs are returned to the application by subsequent calls to xgl_pick_get_identifiers.

XGL_CTX_PICK_ID_1 and XGL_CTX_PICK_ID_2 are independent of each other; each can be set separately.

The default value for the pick ID attributes is 0.

The pick ID attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_STYLE(3)

NAME

XGL_CTX_PICK_STYLE – defines how pick IDs are added to the pick buffer when a primitive is picked

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_PICK_STYLE, Xgl_pick_style, 0);
xgl_object_get(ctx, XGL_CTX_PICK_STYLE, Xgl_pick_style *);
```

DESCRIPTION

This attribute controls how information associated with *pick events* is stored into the Context *pick buffer* when the buffer becomes full. It also controls how pick events are returned to the application when **xgl_pick_get_identifiers**(3) is invoked. Two options are available for this attribute:

XGL_PICK_FIRST_N

The first N pick events (where N is the pick buffer size specified by **XGL_CTX_PICK_BUFFER_SIZE**(3)) are stored in the buffer; subsequent pick events are not buffered. When using xgl_pick_get_identifiers, the events are returned in the order in which they were stored. The first event is returned as the first element of the return list.

XGL_PICK_LAST_N

The pick events are stored in a first-in, first-out manner. The first event is removed to make room for the N+1st event. When using xgl_pick_get_identifiers, the events are returned in the reverse order to that in which they were stored. The last event in the buffer is the first one returned.

The attribute **XGL_CTX_PICK_ENABLE**(3) must be TRUE for picking to occur. **XGL_CTX_PICK_STYLE**(3) can only be changed when XGL_CTX_PICK_ENABLE is FALSE.

The default value is XGL_PICK_LAST_N.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)

NAME

XGL_CTX_PICK_SURF_STYLE – controls how polygon picking occurs

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_PICK_SURF_STYLE, Xgl_pick_surf_style, 0);
xgl_object_get(ctx, XGL_CTX_PICK_SURF_STYLE, Xgl_pick_surf_style *);
```

DESCRIPTION

This attribute controls how polygon picking occurs. The alternatives for this attribute are specified by the enumerated data type *Xgl_pick_surf_style* and are described below:

XGL_PICK_SURF_AS_SOLID
: The polygon is picked as a solid polygon, regardless of the current setting of **XGL_CTX_SURF_FRONT_FILL_STYLE** or **XGL_3D_CTX_SURF_BACK_FILL_STYLE**.

XGL_PICK_SURF_AS_FILL_STYLE
: The polygon is picked as it is rendered, in accordance with the XGL picking semantics.

For a description of picking in XGL, see the attribute **XGL_CTX_PICK_APERTURE**(3). For a description of polygon fill styles, see the attribute **XGL_CTX_SURF_FRONT_FILL_STYLE**(3).

The default value is XGL_PICK_SURF_AS_SOLID.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_pick_clear(3)
xgl_pick_get_identifiers(3)
XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)

NAME

XGL_CTX_PLANE_MASK – defines the pixel plane mask

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_PLANE_MASK, Xgl_usgn32, 0);
xgl_object_get(ctx, XGL_CTX_PLANE_MASK, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the enable mask for writing to pixel bit planes. Each bit in the mask corresponds to a bit plane of the pixel. For example, an 8-bit-deep frame buffer would have 8 bit planes and 8 corresponding bits in XGL_CTX_PLANE_MASK. If a bit is **0** in the plane mask, writing to that plane is disabled. If a bit is **1**, writing to the plane is typically enabled, although contingent upon the restrictions described below.

Using a plane mask is well defined for indexed Rasters on indexed hardware. With other combinations, the plane mask may not work as expected (see NOTES below). In general, the hardware raster pixel values are masked with the source pixel in the hardware format.

For each bit in each pixel of the Destination Raster, the following expression defines how the pixel color is computed using the plane mask and the color map:

((rop(*src*,*dst*) AND (plane_mask AND plane_mask_mask)) OR (*dst* AND (NOT (plane_mask AND plane_mask_mask)))

where:
src is the source pixel to be rendered
dst is the pixel value in the frame buffer
plane_mask is the value of XGL_CTX_PLANE_MASK
rop is the ROP defined by XGL_CTX_ROP
plane_mask_mask is one of the following cases:

- If the device is NOT a window raster, or if the application manages itself the X colormap (i.e. it has set **XGL_CMAP_NAME**(3) of the Color Map object associated with the device that is associated with the context), then the value used for *plane_mask_mask* is -1.
- If the device is a window raster, and XGL manipulates the X colormap on behalf of the application (ie: the application has NOT set **XGL_CMAP_NAME**(3) of the Color Map object associated with the device that is associated with the context), then *plane_mask_mask* is the set of X pixel bits that XGL has allocated from X.

This is done in hardware for accelerated window rasters.

The default value is (Xgl_usgn32)-1 (all bits set to 1).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_CMAP_NAME(3)
XGL_CTX_ROP(3)

NOTE

XGL_CTX_PLANE_MASK(3) is not supported for the following configurations:

RGB on 8-bit rasters
RGB on memory rasters

For accelerated RGB devices, refer to your accelerator guide.

NAME

XGL_CTX_RASTER_FILL_STYLE – defines the fill style used when xgl_context_copy_-buffer is invoked

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_RASTER_FILL_STYLE, Xgl_raster_fill_style, 0);
xgl_object_get(ctx, XGL_CTX_RASTER_FILL_STYLE, Xgl_raster_fill_style *);
```

DESCRIPTION

This attribute sets the *fill style* used when **xgl_context_copy_buffer**(3) is invoked. The fill style specifies whether to copy the source Raster directly to the destination Raster, or whether to perform stippling during the copy. Three options are currently available for this attribute as defined in the enumerated type *Xgl_raster_fill_style*. The options are:

XGL_RAS_FILL_COPY

The source Raster is copied directly to the destination Raster. The arguments to the xgl_context_copy_buffer call define the rectangular block of pixels that will be copied.

XGL_RAS_FILL_STIPPLE

The copy of the source Raster to the destination Raster is modified by examining the bit values in the 1-bit-deep XGL Memory Raster specified by **XGL_CTX_-RASTER_FPAT**(3)**.** This Memory Raster is mapped to the destination Raster by using the position specified by the **XGL_CTX_RASTER_FPAT_POSITION**(3) attribute. If the XGL_CTX_RASTER_FPAT Memory Raster is smaller than the source Raster, it will be replicated by tiling, as needed. When a pixel is to be copied from the source to the destination Raster, the corresponding pixel in the XGL_CTX_RASTER_-FPAT Memory Raster is checked. If that pixel is 1, the pixel copy operation is done. Otherwise, there is no copy and the destination pixel remains unchanged.

XGL_RAS_FILL_OPAQUE_STIPPLE

This mode is similar to XGL_RAS_FILL_STIPPLE, except that when the pixel in the XGL_CTX_RASTER_FPAT Memory Raster is 0, the color defined by the attribute **XGL_CTX_RASTER_STIPPLE_COLOR**(3) is copied to the destination Raster.

The copy operation is also affected by the Context attributes **XGL_CTX_ROP**(3) and **XGL_-CTX_PLANE_MASK**(3)**.**

The default value is XGL_RAS_FILL_COPY.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_copy_buffer(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FPAT(3)
XGL_CTX_RASTER_FPAT_POSITION(3)
XGL_CTX_RASTER_STIPPLE_COLOR(3)
XGL_CTX_ROP(3)

NAME

XGL_CTX_RASTER_FPAT – defines the Memory Raster fill pattern used when xgl_context_copy_buffer is invoked using a stipple fill

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_RASTER_FPAT, Xgl_mem_ras, 0);
xgl_object_get(ctx, XGL_CTX_RASTER_FPAT, Xgl_mem_ras *);
```

DESCRIPTION

This attribute specifies the Memory Raster that contains the pattern used when performing a *stipple fill* Raster-to-Raster copy. The operator **xgl_context_copy_buffer**(3) is used to do the copy. The **XGL_CTX_RASTER_FPAT**(3) Memory Raster must be 1-bit deep.

If the Context attribute **XGL_CTX_RASTER_FILL_STYLE**(3) is set to XGL_RAS_FILL_COPY, then XGL_CTX_RASTER_FPAT is ignored, and may be NULL.

The following fill patterns are predefined in XGL. They are 1-bit-deep Memory Rasters that are created when an XGL session begins.

xgl_fpat_hatch_horiz
xgl_fpat_hatch_vert
xgl_fpat_hatch_diag_45
xgl_fpat_hatch_diag_135
xgl_fpat_hatch_grid_r
xgl_fpat_hatch_grid_d
xgl_fpat_hatch_horiz_dbl
xgl_fpat_hatch_vert_dbl
xgl_fpat_hatch_diag_45_dbl
xgl_fpat_hatch_diag_135_dbl
xgl_fpat_hatch_grid_r_dbl
xgl_fpat_hatch_grid_d_dbl

If the Context attribute **XGL_CTX_RASTER_FILL_STYLE**(3) is set to either XGL_RAS_FILL_STIPPLE or XGL_RAS_FILL_OPAQUE_STIPPLE, then xgl_context_copy_buffer examines the XGL_CTX_RASTER_FPAT Raster and uses it for stippling. The Raster is mapped to the destination Raster by the Context attribute **XGL_CTX_RASTER_FPAT_POSITION**(3)**.** When copying to the destination Raster, the xgl_context_copy_buffer routine checks the bit value of the pixel corresponding to the position of the destination pixel to determine stippling action. If the bit is 1, the source pixel is copied directly to the destination pixel. If the bit is 0, and the XGL_CTX_RASTER_FILL_STYLE is XGL_RAS_FILL_STIPPLE, the destination pixel is left untouched. Otherwise, if the bit is 0, and the XGL_CTX_RASTER_FILL_STYLE is XGL_RAS_FILL_OPAQUE_STIPPLE, the color defined by the Context attribute **XGL_CTX_RASTER_STIPPLE_COLOR**(3) is copied to the destination pixel.

The copy operation is also affected by the Context attributes **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3)**.**

The default value is NULL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_copy_buffer(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RASTER_FPAT_POSITION(3)
XGL_CTX_RASTER_STIPPLE_COLOR(3)
XGL_CTX_ROP(3)

NAME

XGL_CTX_RASTER_FPAT_POSITION – defines the mapping of the fill pattern Raster used when xgl_context_copy_buffer is invoked using a stipple fill

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_RASTER_FPAT_POSITION, Xgl_pt_i2d *, 0);
xgl_object_get(ctx, XGL_CTX_RASTER_FPAT_POSITION, Xgl_pt_i2d *);
```

DESCRIPTION

This attribute defines a point, in device coordinates (DC), of the destination Raster that specifies the origin of the stipple fill pattern Memory Raster used by **xgl_context_copy_buffer**(3).

This attribute; the Raster fill attributes **XGL_CTX_RASTER_FPAT**(3), **XGL_CTX_RASTER_FILL_STYLE**(3), and **XGL_CTX_RASTER_STIPPLE_COLOR**(3); and the general Context attributes **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3), are all used by xgl_context_copy_buffer.

The default value is (0, 0).

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_copy_buffer(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RASTER_FPAT(3)
XGL_CTX_RASTER_STIPPLE_COLOR(3)
XGL_CTX_ROP(3)

NAME

XGL_CTX_RASTER_STIPPLE_COLOR – defines the background color used when xgl_-context_copy_buffer is invoked using an opaque stipple fill

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_RASTER_STIPPLE_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_RASTER_STIPPLE_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute specifies the color to be used as the background color when the Context attribute **XGL_CTX_RASTER_FILL_STYLE**(3) is set to XGL_RAS_FILL_OPAQUE_STIPPLE during an **xgl_context_copy_buffer**(3) operation. For more information on opaque stipples, see **XGL_CTX_RASTER_FILL_STYLE**(3)**.**

The copy operation is also affected by the Context attributes **XGL_CTX_ROP**(3) and **XGL_-CTX_PLANE_MASK**(3)**.**

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_copy_buffer(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RASTER_FPAT(3)
XGL_CTX_RASTER_FPAT_POSITION(3)
XGL_CTX_ROP(3)

NAME

XGL_CTX_RENDER_BUFFER – controls where rendering is done

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_CTX_RENDER_BUFFER, Xgl_render_mode, 0);
xgl_object_get(ctx, XGL_CTX_RENDER_BUFFER, Xgl_render_mode *);
```

DESCRIPTION

This attribute controls where primitives are to be rendered on the Device associated with the Context *ctx*.

The value of this attribute is the logical OR'ing of the buffer options listed below. The available buffer options are specified by the data type *Xgl_render_mode*. The possible values are:

XGL_RENDER_DRAW_BUFFER
: The primitive operations are rendered onto the draw buffer of the raster.

XGL_RENDER_DISPLAY_BUFFER
: The primitive operations are rendered onto the display buffer of the raster. This option is ignored for memory rasters and single buffered window rasters. Thus, this option has an impact only on double buffered window rasters.

XGL_RENDER_Z_BUFFER
: The XGL primitive operators are treated as non-operations (NO-OPS). This attribute value is used only during raster operations, **xgl_context_copy_buffer**(3), **xgl_context_set_multi_pixel**(3), **xgl_context_set_pixel**(3), and **xgl_context_set_pixel_row**(3). In order for this attribute value to be effective, the attribute **XGL_3D_CTX_HLHSR_MODE**(3) should be XGL_HLHSR_Z_BUFFER during the raster operations.

The default value is XGL_RENDER_DRAW_BUFFER.

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_context_copy_buffer(3)
xgl_context_set_multi_pixel(3)
xgl_context_set_pixel(3)
xgl_context_set_pixel_row(3)
XGL_3D_CTX_HLHSR_MODE(3)

NAME

XGL_CTX_RENDERING_ORDER – allows a device to alter the rendering order of primitives

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx                 ctx;
xgl_object_set(ctx, XGL_CTX_RENDERING_ORDER, Xgl_rendering_order, 0);
xgl_object_get(ctx, XGL_CTX_RENDERING_ORDER, Xgl_rendering_order *);
```

DESCRIPTION

This attribute is used to allow hardware devices to batch primitives in a more efficient way to yield faster rendering performance. This attribute may have no effect if your hardware platform does not support rendering order.

Suppose an application draws a polyline, then a polygon and then another polyline. On some hardware accelerators, this could mean swapping in the line attributes, drawing a polyline, swapping in the polygon attributes, drawing a polygon and finally swapping in the polyline attributes again to draw the final polyline. Rendering performance could be increased on such a device if the device were allowed to render both polylines together and then render the polygon, resulting in less attribute swapping.

The possible values for this attribute are as follows:

XGL_RENDERING_ORDER_GIVEN
: Rendering is done in the same order as the XGL calls. No reordering of the primitives is allowed.

XGL_RENDERING_ORDER_HLHSR
: The hardware accelerator can alter the rendering order only if **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER. If **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_NONE then no reordering of the primitives is allowed. For 2D Contexts, this value performs the same as XGL_RENDERING_ORDER_GIVEN.

XGL_RENDERING_ORDER_ANY
: The hardware accelerator is free to alter the rendering order. If **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_NONE or the Context is 2D, and the accelerator reorders the primitives, the image may not appear as the user expects.

The default value is XGL_RENDERING_ORDER_GIVEN

This attribute is not pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_3D_CTX_HLHSR_MODE(3)

NAME

XGL_CTX_ROP – defines the Raster operation (ROP)

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_ROP, Xgl_rop_mode, 0);
xgl_object_get(ctx, XGL_CTX_ROP, Xgl_rop_mode *);
```

DESCRIPTION

This attribute defines the Raster operation (ROP) for the Context *ctx*. A ROP specifies how the *source* and *destination* pixels are logically combined to produce the result written to the Destination Raster. All XGL graphics primitives are rendered using this logical operation to combine the source and destination information. For primitive operations, the *source* is the set of pixels comprising the graphics primitive (for example, the pixels along a line produced by the **xgl_multipolyline**(3) call), and the *destination* is the Raster associated with the current Context.

The source, destination, and plane_mask are combined, using the algorithm shown below, to yield the new destination pixel values. For each bit in each pixel of the destination drawable, the following expression defines whether that bit is set.

((rop(*src*,*dst*) AND (plane_mask AND plane_mask_mask)) OR (*dst* AND (NOT (plane_mask AND plane_mask_mask)))

where:
src is the source pixel to be rendered
dst is the pixel value in the frame buffer
plane_mask is the value of XGL_CTX_PLANE_MASK
rop is the ROP defined by XGL_CTX_ROP
plane_mask_mask is one of the following cases:

- If the device is NOT a window raster, or if the application manages itself the X colormap (i.e. it has set **XGL_CMAP_NAME**(3) of the Color Map object associated with the device that is associated with the context), then the value used for *plane_mask_mask* is -1.
- If the device is a window raster, and XGL manipulates the X colormap on behalf of the application (ie: the application has NOT set **XGL_CMAP_NAME**(3) of the Color Map object associated with the device that is associated with the context), then *plane_mask_mask* is the set of X pixel bits that XGL has allocated from X.

This is done in hardware for accelerated window rasters.

The options available for this attribute are shown below along with the corresponding logical operation:

XGL_ROP_CLEAR
: Clear destination – *result = 0 (i.e. all bits are reset to 0)*

XGL_ROP_SET
: Set destination – *result = -1 (i.e. all bits are set to 1)*

XGL_ROP_COPY
: Copy source to destination – *result = source*

XGL_ROP_COPY_INVERTED
: Copy 1's complement of source to destination – *result = !source*

XGL_ROP_INVERT
: Invert destination – *result = !dest*

XGL_ROP_AND
: AND source and destination – *result = source & dest*

XGL_ROP_AND_REVERSE
: For each bit set in the destination, clear the source and write the source to the destination – *result = source & (!dest)*

XGL_ROP_AND_INVERTED
: For each bit set in the source, erase destination – *result = (!source) & dest*

XGL_ROP_NAND
: NAND source with destination – *result = !(source & dest)*

XGL_ROP_OR
: OR source with destination – *result = source | dest*

XGL_ROP_OR_REVERSE
: Replace the destination with the OR of the source and the 1's complement of the destination – *result = source | (!dest)*

XGL_ROP_OR_INVERTED
: OR of the destination with the 1's complement of the source – *result = !(source) | dest*

XGL_ROP_NOR
: NOR source with destination – *result = !(source | dest)*

XGL_ROP_XOR
: Exclusive OR source with destination – *result = source ^ dest*

XGL_ROP_EQUIV
: For each bit in source and destination that are equivalent, set the bit in the destination. For those that are not equivalent, clear the destination – *result = !(source ^ dest)*

XGL_ROP_NOOP
: Do nothing – *result = dest*

The default value is XGL_ROP_COPY.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_CMAP_NAME(3)
XGL_CTX_PLANE_MASK(3)

NOTE For accelerated RGB devices, refer to your accelerator guide.

NAME

XGL_CTX_SFONT_0,
XGL_CTX_SFONT_1,
XGL_CTX_SFONT_2,
XGL_CTX_SFONT_3 – specify which Stroke Font objects are used within the current Context

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_SFONT_*x*, Xgl_sfont, 0);
xgl_object_get(ctx, XGL_CTX_SFONT_*x*, Xgl_sfont *);
where *x* can be any of 0, 1, 2, or 3

DESCRIPTION

These attributes select the Stroke Font objects in use in the Context *ctx* for all subsequent stroke text rendering operations using either **xgl_stroke_text**(3)**, xgl_annotation_text**(3)**,** or **xgl_stroke_text_extent**(3)**.**

Character set 0 applies for both ISO or MBY / multi-byte encodings. Character sets 1, 2, and 3 only apply in the case of Extended UNIX Code / Multi-Byte (MBY) encoding. ISO or MBY encoding can be selected with the context attribute **XGL_CTX_STEXT_CHAR_-ENCODING**(3)**.**

Before a Stroke Font can be selected with one of these attributes, it must first be ''created'' using the **xgl_object_create**(3) operator. xgl_object_create returns a handle to the requested Stroke Font object, which can then be used to set the XGL_CTX_SFONT_x attributes.

Stroke text is drawn as a series of vectors (or strokes), and can be clipped with a maximum of precision when rendering onto the Device attached to the Context. As a bounding box is computed for the text before rendering, the **XGL_CTX_THRESHOLD**(3) attribute applies for stroke text.

XGL supports different Stroke Fonts. For a complete list of the supported fonts, see the **XGL_SFONT_NAME**(3) Stroke Font attribute.

For a description of the ISO or MBY schemes, see the Context attribute **XGL_CTX_STEXT_-CHAR_ENCODING**(3)**.**

Because MBY uses the most significant bit (MSB) of characters to determine the codeset, applications should not use this bit for purposes other than controlling the codeset of the character. Applications can use full 8 bit characters when ISO encoding is selected.

The default value is a Stroke Font of name *Roman_M* for character set 0. The three other character sets are not used by default.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_object_create(3)
xgl_annotation_text(3)
xgl_stroke_text(3)
xgl_stroke_text_extent(3)
XGL_CTX_STEXT_CHAR_ENCODING(3)
XGL_CTX_THRESHOLD(3)
XGL_SFONT_NAME(3)

NAME

XGL_CTX_STEXT_AA_BLEND_EQ,
XGL_CTX_STEXT_AA_FILTER_WIDTH,
XGL_CTX_STEXT_AA_FILTER_SHAPE – controls if and how text are antialiased

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_STEXT_AA_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_set(ctx, XGL_CTX_STEXT_AA_FILTER_WIDTH, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_STEXT_AA_FILTER_SHAPE, Xgl_filter_shape, 0);

xgl_object_get(ctx, XGL_CTX_STEXT_AA_BLEND_EQ, Xgl_blend_eq *);
xgl_object_get(ctx, XGL_CTX_STEXT_AA_FILTER_WIDTH, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_STEXT_AA_FILTER_SHAPE, Xgl_filter_shape *);

DESCRIPTION

These attributes control if and how text are antialiased. Antialiasing smooths out single-pixel-wide text by spreading out the spectral energy of the stroke over 2 or more adjacent pixels. The result is that the sides of the stroke are slightly blurred and, as seen from a normal viewing distance, smoother than non-antialiased text.

The blurring of the text is done by filtering the stroke onto neighboring pixels. The shape and size of this filter controls how the antialiased text will look. Currently, XGL_FILTER_-GAUSSIAN is the only possible value for the **XGL_CTX_STEXT_AA_FILTER_SHAPE**(3) attribute.

The width of the filter is controlled by the attribute **XGL_CTX_STEXT_AA_FILTER_WIDTH**(3)**.** This is the number of pixels touched by the filter. A Device will use the largest filter width it supports which is less than or equal to the specified filter width. A width of 0 is not an error and is treated as if the width was 1.

The attribute **XGL_CTX_STEXT_AA_BLEND_EQ**(3) controls how the pixel values in the antialiased text are blended with the existing pixel values in the Device. The data type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that antialiasing use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_-CONST_BG.

All three attributes must have consistent settings in order to achieve visually pleasing results. Some notes on using these attributes:

- Both the XGL_CTX_STEXT_AA_BLEND_EQ and the XGL_CTX_STEXT_AA_FILTER_-WIDTH must be set to something other than defaults to achieve antialiasing.
- Setting the XGL_CTX_STEXT_AA_BLEND_EQ to XGL_BLEND_NONE and the XGL_-CTX_STEXT_AA_FILTER_WIDTH to greater than 1 is a way to *undraw* antialiased stroke text. The color used for undrawing is the color that the text would have been drawn in if antialiasing was off.
- Setting the XGL_CTX_STEXT_AA_BLEND_EQ to other than NONE and the XGL_CTX_-STEXT_AA_FILTER_WIDTH to 1 is undefined and should not be used. The results will be device dependent.

The default value for XGL_CTX_STEXT_AA_BLEND_EQ is XGL_BLEND_NONE.

The default value for XGL_CTX_STEXT_AA_FILTER_WIDTH is 1.

The default value for XGL_CTX_STEXT_AA_FILTER_SHAPE is XGL_FILTER_GAUSSIAN.

All these attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NOTES

Antialiasing support on 2D Contexts is device dependent.

NAME

XGL_CTX_STEXT_ALIGN_HORIZ – defines the horizontal alignment used for stroke text characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_ALIGN_HORIZ, Xgl_stext_align_horiz, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_ALIGN_HORIZ, Xgl_stext_align_horiz *);
```

DESCRIPTION

This attribute specifies the horizontal alignment used when rendering stroke text characters in the Context *ctx*. In XGL, stroke text can be rendered using four different horizontal alignments with respect to the text position. The following values are allowed:

XGL_STEXT_ALIGN_HORIZ_RIGHT
XGL_STEXT_ALIGN_HORIZ_NORMAL

These values specify a text rendered at the right side of the position of the string.

XGL_STEXT_ALIGN_HORIZ_LEFT

This value should be used each time an application wants the text to be on the left side of the position given for the stroke text operator.

XGL_STEXT_ALIGN_HORIZ_CENTER

This value centers the text on the position given for the stroke text operator.

The application can use this attribute to specify how the text string should be aligned with respect to the position of the text. When rendering stroke text using the **xgl_stroke_text**(3) operator, the characters are interpreted from the stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's model coordinate (MC) system.

The default value is XGL_STEXT_ALIGN_HORIZ_NORMAL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)

NAME

XGL_CTX_STEXT_ALIGN_VERT – defines the vertical alignment used for stroke text

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_ALIGN_VERT, Xgl_stext_align_vert, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_ALIGN_VERT, Xgl_stext_align_vert *);
```

DESCRIPTION

This attribute specifies the vertical alignment used when rendering stroke text characters in the Context *ctx*. In XGL, stroke text can be rendered using six different vertical alignments with respect to the text position. The following values are allowed:

XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_ALIGN_VERT_NORMAL

Indicates that the base line of the character cells must map on the position specified for the string.

XGL_STEXT_ALIGN_VERT_BOTTOM
XGL_STEXT_ALIGN_VERT_TOP

Used each time an application wants the position given for stroke text operators to match the bottom line or the top line, respectively, of the character cells used in the current font.

XGL_STEXT_ALIGN_VERT_HALF

Ensures that the text will be vertically centered on the position given to the stroke text operators.

XGL_STEXT_ALIGN_VERT_CAP

Maps the cap line of character cells to the position given for the stroke text operators.

The application can use this attribute to specify how the text string should be aligned with respect to the position of the text. When rendering stroke text using the **xgl_stroke_text**(3) operator, the characters are interpreted from stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's model coordinate (MC) system.

The default value is XGL_STEXT_ALIGN_VERT_NORMAL.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)

NAME

XGL_CTX_STEXT_CHAR_ENCODING – specifies which character encoding scheme is used with stroke text

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_ENCODING, Xgl_char_encoding, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_ENCODING, Xgl_char_encoding *);
```

DESCRIPTION

Purpose

This attribute specifies which encoding scheme is used when passing text to the following XGL stroke text primitives:

xgl_stroke_text(3)
xgl_annotation_text(3)
xgl_stroke_text_extent(3)

Encoding Values

The current version of XGL supports several text encoding schemes:

Extended Unix Code, or multi-byte (MBY) encoding with single or multiple strings,
ISO encoding with single or multiple strings.

These schemes can be selected by changing the Context attribute XGL_CTX_STEXT_CHAR_-ENCODING by or-ing together two of the following values:

XGL_CHAR_MBY, XGL_CHAR_ISO

XGL_CHAR_MBY selects EUC / multi-byte encoding. XGL accepts up to four different fonts per Context object to support internationalization extensions. XGL provides the ability to switch between different languages or *character sets*, and can display characters from very large character sets. XGL_CHAR_ISO selects ISO encoding. In that case, the 256 entries of a single font can be used.

XGL_SINGLE_STR, XGL_MULTI_STR

XGL_SINGLE_STR selects the font(s) from the context, and the string of characters to render is a normal C string. XGL_MULTI_STR selects multiple strings encoding. XGL accepts texts composed of sets of (font, string) elements. Each string in a particular set is rendered using the Font object associated with it. There is no limit to the number of sets accepted in this coding scheme.

EUC encoding

EUC defines an encoding for character-encoding that allows multibyte characters as well as the mixing of different character sets within a string. EUC encoding uses bit patterns and control characters to select the different character sets in the string. XGL supports up to four character-encoding sequences, also called *codesets*. Codesets are numbered 0 to 3. EUC encoding uses the most significant bit (MSB) of the character byte and two special characters, SS2 and SS3, to map the characters in the input string into the codesets.

Codeset 0 is defined as characters with MSB set to 0. This is normal 7-bit ASCII character encoding.

Codeset 1 is defined as characters with MSB set to 1. This is equivalent to a logical OR of the character with octal value *'200*. ASCII A (*'101*) encoded in codeset 1 is octal *'301*.

Codeset 2 is defined as characters with MSB set to 1 and preceded by a single shift byte, called SS2. SS2 has the value 0x8E (octal *'216*). ASCII A (*'101*) encoded in codeset 2 is *'216'301*.

Codeset 3 is also defined as characters with MSB set to 1, but preceded by a single shift byte, called SS3. SS3 has the value 0x8F (octal *'217*). ASCII A (*'101*) encoded in codeset 3 is *'217'301*.

Because MBY uses the MSB of characters to determine the codeset, applications should not use this bit for purposes other than controlling the codeset of the character.

Some character sets, such as Japanese Kanji require two bytes to display one character. In these cases, the two bytes are encoded the same way. For example, if a Kanji character is *'101'102* in codeset 0, it will be *'217'301'302* in codeset 3.

Example

Many Japanese applications use EUC to encode English, Kanji, and Katakana (phonetic Japanese) characters within a single string. The codesets are often associated with character sets as follows:

Codeset 0 - ASCII
Codeset 1 - Kanji characters, each pair of input bytes specifies one Kanji character
Codeset 2 - Katakana characters
Codeset 3 - Implementation-dependent

Codeset 3 could be used for an additional character set, such as one for special symbols.

ISO encoding

ISO defines 256 entries character sets that contain accentuation and character symbols frequently used. When this attribute is selected, XGL complies with the ISO 8859 normalization. XGL-3.0 fonts comply to this standard.

Single string encoding

Single string encoding is the simplest way of handing text to XGL. A single C-like (NULL-terminated) text string is passed to the text functions.

Multiple string encoding

Multiple string encoding represents strings as an array of (font, string) elements, using a structure of type Xgl_mono_text_list as defined in **xgl_struct**(3)**.** When using this encoding scheme, the application provides text as an array of Xgl_mono_text structures. XGL scans each structure, loading the corresponding font and extracting the characters from the string to be rendered. Each sub-string is concatenated to the previous one, forming a single string of text when rendered.

The default value for this attribute is XGL_SINGLE_STR | XGL_CHAR_MBY. In the next major release of XGL, the default value for this attribute will change to XGL_SINGLE_STR | XGL_CHAR_ISO.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_annotation_text(3)
xgl_stroke_text(3)
xgl_stroke_text_extent(3)

NAME

XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR – defines the horizontal expansion applied to stroke text characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR, float, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR, float *);
```

DESCRIPTION

This attribute specifies the horizontal expansion factor of stroke text characters within the Context *ctx*. It is expressed as a *positive* factor of the nominal character height, as specified by the **XGL_CTX_STEXT_CHAR_HEIGHT**(3) attribute.

When rendering stroke text using **xgl_stroke_text**(3)**,** the characters are interpreted from stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's model coordinate (MC) system.

XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR is used to modify the width of the characters without changing the character height or the spacing between characters. The resultant character width becomes:

$H \times E$

where H = character height – **XGL_CTX_STEXT_CHAR_HEIGHT**(3)

E = character expansion factor – **XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR**(3)

The space between characters can be modified by using the **XGL_CTX_STEXT_CHAR_-SPACING**(3) attribute.

The default value is 1.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)
XGL_CTX_STEXT_CHAR_HEIGHT(3)
XGL_CTX_STEXT_CHAR_SPACING(3)

NAME

XGL_CTX_STEXT_CHAR_HEIGHT – defines the nominal size of stroke text characters

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_HEIGHT, float, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_HEIGHT, float *);
```

DESCRIPTION

This attribute specifies the nominal height of stroke text characters, expressed in the application's local model coordinates (LMC). (See **XGL_CTX_LOCAL_MODEL_TRANS**(3)**.**)

When rendering stroke text using **xgl_stroke_text**(3)**,** the characters are interpreted from the stroke text data file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's model coordinate (MC) system.

This attribute affects the size of XGL characters globally. Not only does it affect the character height, but it also affects the character width and the spacing between characters (see **XGL_CTX_STEXT_CHAR_SPACING**(3) and **XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR**(3)).

The default value is 100.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)
XGL_CTX_STEXT_CHAR_SPACING(3)

NAME

XGL_CTX_STEXT_CHAR_SLANT_ANGLE – defines the angle that stroke text makes with the upright direction

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_SLANT_ANGLE, float, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_SLANT_ANGLE, float *);
```

DESCRIPTION

This attribute specifies the angle of inclination of stroke text characters, expressed as an angle (in radians) with respect to the upright direction.

When rendering stroke text using **xgl_stroke_text**(3)**,** the characters are interpreted from the stroke text file in text local coordinates (TLC). This coordinate system is independent of the user's coordinates, and the stroke characters are transformed to match the given size value in the user's model coordinates (MC) system.

This attribute is used to modify the appearance of the characters without changing their spacing, height, and width. It can take values between $-\pi/2$ and $+\pi/2$. A slant angle between $-\pi/2$ and 0 slants the characters in the backward direction. A slant angle between 0 and $+\pi/2$ slants the characters in the forward direction.

This attribute does not affect the text alignment, as specified by the attributes **XGL_CTX_STEXT_ALIGN_HORIZ**(3) and **XGL_CTX_STEXT_ALIGN_VERT**(3)**.**

The default value is 0.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)
XGL_CTX_STEXT_ALIGN_HORIZ(3)
XGL_CTX_STEXT_ALIGN_VERT(3)

NAME

XGL_CTX_STEXT_CHAR_SPACING – defines the size of the gap between characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_SPACING, float, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_SPACING, float *);
```

DESCRIPTION

This attribute specifies the space provided between two consecutive stroke text characters drawn in the Context *ctx*. It is expressed as a fraction of the character height specified by the **XGL_CTX_STEXT_CHAR_HEIGHT**(3) attribute.

When rendering stroke text using **xgl_stroke_text**(3)**,** the characters are interpreted from the stroke text file in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's coordinate system.

XGL_CTX_STEXT_CHAR_SPACING is used to modify the spacing of stroke text characters without changing their height and width. The actual space between characters is defined as:

$$O + (S \times H)$$

where O = the *original* character spacing defined in the font file (usually 0)

S = the character spacing (the value of XGL_CTX_STEXT_CHAR_SPACING)

H = the character height (the value of XGL_CTX_STEXT_CHAR_HEIGHT)

For example, a value of 0.0 for XGL_CTX_STEXT_CHAR_SPACING adds no more additional spacing between characters than what is initially defined in the font file; a value of 1.0 adds space equal to one unit of nominal character height.

The default value is 0.0.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)
XGL_CTX_STEXT_CHAR_HEIGHT(3)

NAME

XGL_CTX_STEXT_CHAR_UP_VECTOR – defines the up vector for stroke characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_CHAR_UP_VECTOR, Xgl_pt_f2d *, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_CHAR_UP_VECTOR, Xgl_pt_f2d *);
```

DESCRIPTION

This attribute specifies the *up vector* for stroke characters within the Context *ctx*. The up vector gives the up direction of characters in text local coordinates (TLC).

Changing the up vector is equivalent to defining a rotation for the characters when text is rendered using the **xgl_stroke_text**(3) operator. This vector is independent of the orientation of the virtual device coordinates (VDC) set by the application with the **XGL_CTX_VDC_ORIENTATION**(3) attribute.

With the VDC orientation set to either XGL_Y_UP_Z_TOWARD or XGL_Y_DOWN_Z_AWAY, an up vector of (0.0, 1.0) renders the text normally. XGL automatically sets the correct internal values to the transformation of the TLC so that the VDC orientation does not have any effect on the text to display.

The character up vector is ignored when computing the text extent rectangle bounding the string (see **xgl_stroke_text_extent**(3)).

The default value is (0.0, 1.0).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)
xgl_stroke_text_extent(3)
XGL_CTX_VDC_ORIENTATION(3)

NAME

XGL_CTX_STEXT_COLOR – defines the color used when rendering stroke text characters

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_STEXT_COLOR, Xgl_color *, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute is used to set the color in which stroke text characters are drawn within the Context *ctx*, when using the **xgl_stroke_text**(3) operator.

This color applies only to stroke text rendering.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)

XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_CMAP_RAMP_LIST(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_STEXT_PATH – defines the path used when rendering stroke text characters

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_STEXT_PATH, Xgl_stext_path, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_PATH, Xgl_stext_path *);
```

DESCRIPTION

This attribute specifies the path used when rendering stroke text characters within the Context *ctx*. In XGL, stroke text can be rendered in four different directions from the text position. The following values are allowed to define the text path:

XGL_STEXT_PATH_RIGHT
XGL_STEXT_PATH_LEFT
XGL_STEXT_PATH_UP
XGL_STEXT_PATH_DOWN

The application can use this attribute to specify the direction in which the text should be rendered. When rendering stroke text using the **xgl_stroke_text**(3) operator, the characters are interpreted from the stroke text font file, using the path specified by the application, to generate the correct data in text local coordinates (TLC). This coordinate system is independent of the application's coordinate system, and the stroke characters are transformed to match the given size value in the application's model coordinate (MC) system.

The default value is XGL_STEXT_PATH_RIGHT.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)

NAME

XGL_CTX_STEXT_PRECISION – defines the precision used when rendering stroke text characters

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_STEXT_PRECISION, Xgl_stext_precision, 0);
xgl_object_get(ctx, XGL_CTX_STEXT_PRECISION, Xgl_stext_precision *);

DESCRIPTION

This attribute specifies the minimum precision used when rendering stroke text characters within the Context *ctx*. When using the XGL stroke text operator **xgl_stroke_text**(3), an application can ask for a given level of precision. This attribute ensures that a precision better than or equal to the precision requested by the application will be used to render stroke text.

Text precision is related to the way a string of text is clipped when falling outside the clipping boundaries. The following values are allowed to define the text precision:

XGL_STEXT_PRECISION_STROKE
: Gives the best results, making sure that the stroke text is clipped with pixel precision.

XGL_STEXT_PRECISION_CHAR (not currently supported)
: May render text faster, but any character partially outside the clipping boundaries is completely clipped.

XGL_STEXT_PRECISION_STRING (not currently supported)
: Means that if any character of a given string is even partially outside the clipping boundaries, the entire string is ignored.

The default value is XGL_STEXT_PRECISION_STROKE.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_stroke_text(3)

NOTES

XGL_STEXT_PRECISION_CHAR and XGL_STEXT_PRECISION_STRING text precision clipping modes are not currently supported by XGL.

NAME

XGL_CTX_SURF_AA_BLEND_EQ,
XGL_CTX_SURF_AA_FILTER_WIDTH,
XGL_CTX_SURF_AA_FILTER_SHAPE – controls if and how surfaces are antialiased

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_SURF_AA_BLEND_EQ, Xgl_blend_eq, 0);
xgl_object_set(ctx, XGL_CTX_SURF_AA_FILTER_WIDTH, Xgl_usgn32, 0);
xgl_object_set(ctx, XGL_CTX_SURF_AA_FILTER_SHAPE, Xgl_filter_shape, 0);

xgl_object_get(ctx, XGL_CTX_SURF_AA_BLEND_EQ, Xgl_blend_eq *);
xgl_object_get(ctx, XGL_CTX_SURF_AA_FILTER_WIDTH, Xgl_usgn32 *);
xgl_object_get(ctx, XGL_CTX_SURF_AA_FILTER_SHAPE, Xgl_filter_shape *);
```

DESCRIPTION

These attributes control if and how surfaces are antialiased. Antialiasing smooths out only hollow filled surfaces (other fill styles are not antialiased) by spreading out the spectral energy of the stroke over 2 or more adjacent pixels. The result is that the sides of the stroke are slightly blurred and, as seen from a normal viewing distance, smoother than non-antialiased surfaces.

The blurring of the surface is done by filtering the stroke onto neighboring pixels. The shape and size of this filter controls how the antialiased surface will look. Currently, XGL_FILTER_GAUSSIAN is the only possible value for the **XGL_CTX_SURF_AA_FILTER_SHAPE**(3) attribute.

The width of the filter is controlled by the attribute **XGL_CTX_SURF_AA_FILTER_WIDTH**(3)**.** This is the number of pixels touched by the filter. A Device will use the largest filter width it supports which is less than or equal to the specified filter width. A width of 0 is not an error and is treated as if the width was 1.

The attribute **XGL_CTX_SURF_AA_BLEND_EQ**(3) controls how the pixel values in the antialiased surface are blended with the existing pixel values in the Device. The data type *Xgl_blend_eq* has these values:

```
typedef enum {
        XGL_BLEND_NONE,
        XGL_BLEND_ARBITRARY_BG,
        XGL_BLEND_CONST_BG,
        XGL_BLEND_ADD_TO_BG
} Xgl_blend_eq;
```

The equations used are:

NONE	d = s
ARBITRARY_BG	d = (s * alpha) + (d * (1 - alpha))
CONST_BG	d = ((s - ctx_bg) * alpha) + d
ADD_TO_BG	d = (s * alpha) + d

where:

s = source pixel value
d = destination pixel value
alpha = weight of source pixel
ctx_bg = Context background color

We strongly recommend that antialiasing use XGL_BLEND_ARBITRARY_BG or XGL_BLEND_-CONST_BG.

All three attributes must have consistent settings in order to achieve visually pleasing results. Some notes on using these attributes:

- Both the XGL_CTX_SURF_AA_BLEND_EQ and the XGL_CTX_SURF_AA_FILTER_WIDTH must be set to something other than defaults to achieve antialiasing.
- Setting the XGL_CTX_SURF_AA_BLEND_EQ to XGL_BLEND_NONE and the XGL_CTX_-SURF_AA_FILTER_WIDTH to greater than 1 is a way to *undraw* antialiased surfaces. The color used for undrawing is the color that the surface would have been drawn in if antialiasing was off.
- Setting the XGL_CTX_SURF_AA_BLEND_EQ to other than NONE and the XGL_CTX_-SURF_AA_FILTER_WIDTH to 1 is undefined and should not be used. The results will be device dependent.

The default value for XGL_CTX_SURF_AA_BLEND_EQ is XGL_BLEND_NONE.

The default value for XGL_CTX_SURF_AA_FILTER_WIDTH is 1.

The default value for XGL_CTX_SURF_AA_FILTER_SHAPE is XGL_FILTER_GAUSSIAN.

All these attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ(3)

NOTES

Antialiasing support on 2D Contexts is device dependent.

If both **XGL_3D_CTX_SURF_TRANSP_BLEND_EQ**(3) and **XGL_CTX_SURF_AA_BLEND_EQ**(3) are other than XGL_BLEND_NONE, and a hollow antialiased transparent surface is being drawn, then the blending equation used will be the value of either the **XGL_3D_CTX_SURF_-TRANSP_BLEND_EQ**(3) or the **XGL_CTX_SURF_AA_BLEND_EQ**(3) attribute, and the choice is device dependent.

NAME

XGL_CTX_SURF_EDGE_FLAG – controls the drawing of edges around a surface

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_SURF_EDGE_FLAG, Xgl_boolean, 0);
xgl_object_get(ctx, XGL_CTX_SURF_EDGE_FLAG, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether edges are drawn around a surface. Drawing edges around surfaces enhances the perimeter of polygons. Edges are not drawn when the side of a polygon corresponds to a clipping boundary.

This attribute is a *global* edge-enabling flag in that it controls edges for all rendered surfaces. However, the input point-list data, which describes the surfaces to be rendered, may also contain an *edge flag* as part of the data at each vertex (call this a *vertex edge flag*). The vertex edge flag is a *local* flag because if it is set for the *jth* vertex, it only controls the edge from the *j-1th* to the *jth* vertex. Thus, if vertex edge flag information is provided as part of the input data, both the vertex edge flag and the **XGL_CTX_SURF_EDGE_FLAG**(3) attribute must be set for an edge to be drawn. If no vertex edge flags are provided, setting XGL_CTX_SURF_EDGE_FLAG to TRUE will cause edges to be drawn around all rendered surfaces.

This attribute is independent of **XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG**(3)**,** which controls the display of silhouette edges — or the horizons — where solids modeled with surfaces change from front-facing to back-facing. Although an application can display both highlighting edges and silhouette edges simultaneously, a user probably would prefer to see only one type at any given time.

Edges can be colored, patterned, and depth-cued using the **XGL_CTX_EDGE_COLOR**(3)**, XGL_CTX_EDGE_STYLE**(3)**,** and **XGL_3D_CTX_DEPTH_CUE_MODE**(3) attributes, respectively. The Pattern object chosen for patterned edges is selected with the **XGL_CTX_EDGE_-PATTERN**(3) attribute. Edges can be given a width using the **XGL_CTX_EDGE_WIDTH_-SCALE_FACTOR**(3) attribute. The application programmer should be aware that drawing wide edges can reduce drawing performance.

In a 3D Context, the application programmer can use XGL_CTX_SURF_EDGE_FLAG to produce a fast hidden–line-removal technique. To do this, first set this attribute to ON. Then, set the front and back surface colors to the background color (see **XGL_3D_CTX_-SURF_BACK_COLOR**(3)**, XGL_CTX_SURF_FRONT_COLOR**(3)**,** and **XGL_CTX_BACKGROUND_-COLOR**(3)). Finally, set shading and depth-cueing to XGL_ILLUM_NONE and XGL_DEPTH_-CUE_OFF, respectively (see **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3)**, XGL_3D_CTX_SURF_-FRONT_ILLUMINATION**(3)**,** and **XGL_3D_CTX_DEPTH_CUE_MODE**(3)). The rendering will be equivalent to a fast hidden–line-removal technique.

The default value is FALSE.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_EDGE_COLOR(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_SURF_FRONT_COLOR(3)

NAME

XGL_CTX_SURF_FRONT_COLOR,
XGL_3D_CTX_SURF_BACK_COLOR – define the color used to fill surfaces

SYNOPSIS

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

Xgl_ctx         ctx;

xgl_object_set(ctx, XGL_CTX_SURF_FRONT_COLOR, Xgl_color * , 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_COLOR, Xgl_color * , 0);

xgl_object_get(ctx, XGL_CTX_SURF_FRONT_COLOR, Xgl_color *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_COLOR, Xgl_color *);
```

DESCRIPTION

These attributes (called the *surface color attributes* below) specify the front-facing and back-facing colors used to fill the interior of surfaces under the following conditions:

1. The surface *fill style* (see attributes **XGL_CTX_SURF_FRONT_FILL_STYLE**(3) and **XGL_3D_CTX_SURF_BACK_FILL_STYLE**(3)) must be set to one of the following:
   XGL_SURF_FILL_SOLID
   XGL_SURF_FILL_STIPPLE
   XGL_SURF_FILL_HOLLOW
   XGL_SURF_FILL_OPAQUE_STIPPLE
2. The surface color selectors (see attributes **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR**(3)) and the absence of colors specified with the facets and/or vertices of the primitive establish that the surface color attributes have precedence. In this case, the surface color attributes specify the colors of the primitive at the beginning of the rendering pipeline, which may alter the final rendered surface colors via illumination, shading, and/or depth-cueing.

The differentiation between front- and back-facing surface attributes occurs only if the attribute **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) is TRUE. Otherwise, front-facing attributes are used for both front- and back-facing surfaces. Also, all surfaces in 2D Contexts use **XGL_CTX_SURF_FRONT_COLOR**(3) because there are no back-facing surfaces in 2D Contexts.

The color must be set according to the **XGL_DEV_COLOR_TYPE**(3) attribute of the Raster associated with the Context. If the Device color type is XGL_COLOR_INDEX, the type of the color should be XGL_COLOR_INDEX. Likewise, if the Device color type is XGL_COLOR_RGB, the color passed to the Context should also be XGL_COLOR_RGB.

When specifying a color of type XGL_COLOR_INDEX, the value given as the index should be consistent with the size of the color table within the Device Color Map object associated with the Context (see **XGL_DEV_COLOR_MAP**(3)).

When a Device of color type XGL_COLOR_INDEX is attached to a 3D Context, with depth-cueing enabled (see **XGL_3D_CTX_DEPTH_CUE_MODE**(3)), the color table of the Color Map object associated with the Raster attached to the Context should have consistent ramps to enable a correct rendering.

The default value is: INDEX 1 or RGB (green).

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_CTX_SURF_FRONT_COLOR_SELECTOR(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_COLOR_MAP(3)

NAME

XGL_CTX_SURF_FRONT_COLOR_SELECTOR, XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR – select the source of a surface's color from the vertex, facet, or Context

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_CTX_SURF_FRONT_COLOR_SELECTOR, Xgl_surf_color_sel, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR, Xgl_surf_color_sel, 0);

xgl_object_get(ctx, XGL_CTX_SURF_FRONT_COLOR_SELECTOR, Xgl_surf_color_sel *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR, Xgl_surf_color_sel *);
```

DESCRIPTION

These attributes select the source of a surface's color from the Context or from the facet or vertex data provided with a primitive.

When front and back faces are distinguished with **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3), **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) specifies the color selection used for front-facing surfaces in the 2D or 3D Context *ctx*, and **XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR**(3) specifies the color selection for back-facing surfaces in the 3D Context *ctx*. When front and back faces are not distinguished, **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) specifies the color selection for both front- and back-facing surfaces. Four options are available for this attribute:

XGL_SURF_COLOR_CONTEXT

XGL takes the surface color from the context attributes **XGL_CTX_SURF_FRONT_COLOR**(3) or **XGL_3D_CTX_SURF_BACK_COLOR**(3) when the surface is back-facing and face-distinguishing is enabled (**XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3) is TRUE). This is the color of the surface at the beginning of the rendering pipeline. Illumination, shading, and depth-cueing affect the final appearance of the surface.

XGL_SURF_COLOR_FACET

If the data supplied with a surface primitive has colors for the individual facets, XGL applies these colors to the facets upon entry to the rendering pipeline. Illumination, shading, and depth-cueing affect the final appearance of the facets. If the surface primitive does not have facet colors, XGL uses the context surface colors as described above.

XGL_SURF_COLOR_VERTEX_ILLUM_DEP

This mode applies only to 3D surfaces. In 2D, it gives the same result as XGL_SURF_COLOR_FACET.

In 3D, color selection for this mode depends on the illumination / shading model. If the data supplied with a surface primitive has colors for the individual vertices and the illumination / shading model requires interpolation of vertex colors (**XGL_3D_CTX_SURF_FRONT_ILLUMINATION**(3) is XGL_ILLUM_NONE_INTERP_COLOR or XGL_ILLUM_PER_VERTEX, or **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3) is either of these two values when the surface is back-facing and face distinguishing is

enabled), XGL applies these vertex colors to the vertices upon entry to the rendering pipeline. Illumination, shading, and depth-cueing affect the final appearance of the surface. If the surface primitive does not have colors for the individual vertices, or the illumination / shading model does not require interpolation of vertex colors, XGL applies the facet colors to the facets when facet colors are supplied with the primitive. If no facet colors are available, XGL uses the Context colors.

XGL_SURF_COLOR_VERTEX_ILLUM_INDEP

This mode applies only to 3D surfaces. In 2D, it gives the same result as XGL_-SURF_COLOR_FACET.

In 3D, color selection for this mode is independent of the illumination / shading model in the sense that XGL always uses the vertex colors when vertex colors are available.

If the data supplied with a surface primitive has colors for the individual vertices and the illumination / shading model requires interpolation of vertex colors (**XGL_3D_CTX_SURF_FRONT_ILLUMINATION**(3) is XGL_ILLUM_NONE_INTERP_COLOR or XGL_ILLUM_PER_VERTEX, or **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3) is either of these two values when the surface is back-facing and face distinguishing is enabled), XGL applies these vertex colors to the vertices upon entry to the rendering pipeline. Illumination, shading, and depth-cueing affect the final appearance of the surface.

If the data supplied with a surface primitive has colors for the individual vertices and the illumination / shading model does not require interpolation of vertex colors (**XGL_3D_CTX_SURF_FRONT_ILLUMINATION**(3) is XGL_ILLUM_NONE or XGL_-ILLUM_PER_FACET, or **XGL_3D_CTX_SURF_BACK_ILLUMINATION**(3) is either of these two values when the surface is back-facing and face distinguishing is enabled), XGL treats the color stored with the first vertex of a facet as the facet color at the beginning of the rendering pipeline. For example, in a quadrilateral mesh, a quadrilateral formed by the vertices with indices (2, 5), (2, 6), (3, 5), and (3, 6) would have facet color equivalent to the color stored with vertex (2, 5). In a polygon with multiple boundaries, the color stored with the first vertex of the first boundary would be the facet color.

If illumination is performed per facet, XGL obtains the normal in the following way. If each vertex holds both a color and a normal, XGL treats the color and normal stored with the first vertex of a facet as the facet color and normal at the beginning of the rendering pipeline. If each vertex holds only a color, XGL attempts to use the normal stored with the facet data; if one is not available, XGL calculates a facet normal (see **XGL_3D_CTX_SURF_GEOM_NORMAL**(3)). The attribute **XGL_3D_CTX_SURF_NORMAL_FLIP**(3) applies to the normal when it comes from the vertex or facet, but not when XGL calculates it.

If the first vertex of a facet is removed by clipping, XGL may select the color and normal stored at a newly generated vertex on the clipping boundary. The color and normal there are calculated by interpolating the values stored at the vertices

of the original facet's edge (before clipping).

If the surface primitive does not have colors for the individual vertices, XGL applies the facet colors to the facets when facet colors are supplied with the primitive. If no facet colors are available, XGL uses the Context colors.

The default value for each attribute is XGL_SURF_COLOR_VERTEX_ILLUM_DEP.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_CTX_SURF_FRONT_COLOR(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_GEOM_NORMAL(3)
XGL_3D_CTX_SURF_NORMAL_FLIP(3)

NAME

XGL_CTX_SURF_FRONT_FILL_STYLE,
XGL_3D_CTX_SURF_BACK_FILL_STYLE – define the method by which surfaces are filled

SYNOPSIS

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

Xgl_ctx                 ctx;

xgl_object_set(ctx, XGL_CTX_SURF_FRONT_FILL_STYLE, Xgl_surf_fill_style, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_FILL_STYLE, Xgl_surf_fill_style, 0);

xgl_object_get(ctx, XGL_CTX_SURF_FRONT_FILL_STYLE, Xgl_surf_fill_style *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_FILL_STYLE, Xgl_surf_fill_style *);
```

DESCRIPTION

These attributes define how a surface is filled. For front-facing surfaces in 3D Contexts and all surfaces in a 2D Context, **XGL_CTX_SURF_FRONT_FILL_STYLE**(3) determines the fill style. For back-facing surfaces in a 3D Context, **XGL_3D_CTX_SURF_BACK_FILL_STYLE**(3) determines the fill style when back faces are distinguished from front faces with **XGL_3D_CTX_SURF_FACE_DISTINGUISH**(3); otherwise, **XGL_CTX_SURF_FRONT_FILL_STYLE**(3) applies to back-facing surfaces.

Note that for front-facing 3D surfaces and all 2D surfaces, the default surface color is defined by **XGL_CTX_SURF_FRONT_COLOR**(3). For back-facing 3D surfaces, the default surface color is defined by **XGL_3D_CTX_SURF_BACK_COLOR**(3) when back faces are distinguished from front faces. The surface color selectors (see **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR**(3)) and the availability or absence of colors specified with the facets and/or vertices of a surface primitive determine the surface colors that enter the rendering pipeline. The final rendered colors or 3D surfaces may be altered by lighting, shading, and depth-cueing.

The fill style can have any one of the following values as specified by the data type *Xgl_surf_fill_style*:

XGL_SURF_FILL_SOLID
: The interior is filled entirely.

XGL_SURF_FILL_HOLLOW
: A line is drawn around the interior of the surface in the surface color; the rest of the interior is not filled. The width of the line is device-dependent; however, on Raster devices, the line is 1-pixel wide. Clipping a surface can produce new boundaries for a surface. In that case, no line is drawn for these new boundaries. An XGL_SURF_FILL_HOLLOW filled surface can also be drawn with edges (see **XGL_CTX_SURF_EDGE_FLAG**(3)); in this case, the edges overwrite the line that comprises the hollow surface.

XGL_SURF_FILL_EMPTY
: Nothing is drawn for the interior. If edges are requested, however, they will be drawn around the bounds of the ''invisible'' surface.

XGL_SURF_FILL_OPAQUE_STIPPLE
: Opaque stippling or hatch styles work only with 1-bit-deep fill Rasters (which are

Memory Rasters). Depending on the polygon orientation in 3D, each pixel in the fill Raster, specified by **XGL_CTX_SURF_FRONT_FPAT**(3) and aligned at the **XGL_CTX_SURF_FRONT_FPAT_POSITION**(3)**,** or **XGL_3D_CTX_SURF_BACK_FPAT**(3) and aligned at the **XGL_3D_CTX_SURF_BACK_FPAT_POSITION**(3)**,** is examined. If the fill Raster pixel is set, the corresponding pixel in the destination Raster is set to the surface color. If the fill Raster pixel is not set, the corresponding pixel in the destination Raster is set to the *background color* as specified by the attribute **XGL_CTX_BACKGROUND_COLOR**(3)**.** Only those planes enabled by **XGL_CTX_PLANE_MASK**(3) are modified in the destination Raster.

XGL_SURF_FILL_STIPPLE

Stippling or transparent hatch styles work only with 1-bit-deep fill Rasters (which are Memory Rasters). Depending on the polygon orientation in 3D, each pixel in the fill Raster, specified by XGL_CTX_SURF_FRONT_FPAT and aligned at the XGL_CTX_SURF_FRONT_FPAT_POSITION, or XGL_3D_CTX_SURF_BACK_FPAT and aligned at the XGL_3D_CTX_SURF_BACK_FPAT_POSITION, is examined. If the fill Raster pixel is set, the corresponding pixel in the destination Raster is set to the surface color. If the fill Raster pixel is not set, the corresponding pixel in the destination Raster is unchanged. Only those planes enabled by XGL_CTX_PLANE_MASK are modified in the destination Raster.

XGL_SURF_FILL_PATTERN

Pattern styles work only with fill Rasters (which are Memory Rasters) that have the same depth as the destination Raster attached to the Context. Depending on the polygon orientation in 3D, each pixel in the fill Raster, specified by XGL_CTX_SURF_FRONT_FPAT and aligned at the XGL_CTX_SURF_FRONT_FPAT_POSITION, or XGL_3D_CTX_SURF_BACK_FPAT and aligned at the XGL_3D_CTX_SURF_BACK_FPAT_POSITION, is copied. The fill Raster pixel is copied onto the corresponding pixel in the destination Raster with a replication of the fill Raster if necessary. Only those planes enabled by XGL_CTX_PLANE_MASK are modified in the destination Raster.

For details about how the surface color used for filling is determined, see **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR**(3)**.**

The default value for each attribute is XGL_SURF_FILL_SOLID for Raster Devices.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR(3)
XGL_3D_CTX_SURF_BACK_FPAT(3)
XGL_3D_CTX_SURF_BACK_FPAT_POSITION(3)
XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_FACE_DISTINGUISH(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_SURF_EDGE_FLAG(3)

XGL_CTX_SURF_FRONT_COLOR_SELECTOR(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_FRONT_COLOR(3)

NAME

XGL_CTX_SURF_FRONT_FPAT,
XGL_3D_CTX_SURF_BACK_FPAT – define fill pattern Raster used for stipple fill and pattern fill during primitive rendering

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_SURF_FRONT_FPAT, Xgl_mem_ras, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_FPAT, Xgl_mem_ras, 0);

xgl_object_get(ctx, XGL_CTX_SURF_FRONT_FPAT, Xgl_mem_ras *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_FPAT, Xgl_mem_ras *);

DESCRIPTION

These attributes specify the Memory Raster that contains the pattern used when performing a *stipple fill* or a *pattern fill* during any primitive-rendering operation within the Context *ctx*. The **XGL_CTX_SURF_FRONT_FPAT**(3) and **XGL_3D_CTX_SURF_BACK_FPAT**(3) fill Rasters (which are Memory Rasters) must be 1-bit deep for *stipple fill*. For *pattern fill*, the fill Rasters (which are Memory Rasters) must have the same depth as the destination Raster attached to the Context.

If the Context *fill style* attributes **XGL_CTX_SURF_FRONT_FILL_STYLE**(3) or **XGL_3D_CTX_-SURF_BACK_FILL_STYLE**(3) are set to XGL_SURF_FILL_STIPPLE, XGL_SURF_FILL_OPAQUE_STIPPLE, or XGL_SURF_FILL_PATTERN, the front or back Memory Rasters are examined and used for rendering. If the fill style is set to something else, however, the XGL_CTX_SURF_FRONT_FPAT and XGL_3D_CTX_SURF_BACK_FPAT Rasters are not read and may be NULL.

The following 1-bit-deep fill patterns are predefined in XGL and can be used for generating ''hatch'' styled fills:

xgl_fpat_hatch_horiz
xgl_fpat_hatch_vert
xgl_fpat_hatch_diag_45
xgl_fpat_hatch_diag_135
xgl_fpat_hatch_grid_r
xgl_fpat_hatch_grid_d
xgl_fpat_hatch_horiz_dbl
xgl_fpat_hatch_vert_dbl
xgl_fpat_hatch_diag_45_dbl
xgl_fpat_hatch_diag_135_dbl
xgl_fpat_hatch_grid_r_dbl
xgl_fpat_hatch_grid_d_dbl

When rendering into the destination Raster with a rendering style of type *stipple*, to determine stippling action, XGL checks the bit value of the pixel that corresponds to the position of the destination pixel. If the bit is 1, the corresponding pixel in the destination Raster is set to the surface color. If the bit is 0, and the fill style is XGL_SURF_FILL_STIPPLE, the destination pixel is left untouched. Otherwise, if the bit is 0, and the fill style is XGL_RAS_-FILL_OPAQUE_STIPPLE, the color in the Context attribute **XGL_CTX_BACKGROUND_COLOR**(3) is copied to the destination pixel. For *pattern fill*, the fill Raster pixel is copied onto the

corresponding pixel in the destination Raster with a replication of the fill Raster if necessary. Only those planes enabled by XGL_CTX_PLANE_MASK are modified in the destination Raster.

The XGL_CTX_SURF_FRONT_FPAT Raster is mapped to the destination Raster by the Context attribute **XGL_CTX_SURF_FRONT_FPAT_POSITION**(3)**.**

The XGL_3D_CTX_SURF_BACK_FPAT Raster is mapped to the destination Raster by the Context attribute **XGL_3D_CTX_SURF_BACK_FPAT_POSITION**(3)**.**

The default value is NULL.

These attributes are pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_3D_CTX_SURF_BACK_FPAT_POSITION(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)

NAME

XGL_CTX_SURF_FRONT_FPAT_POSITION, XGL_3D_CTX_SURF_BACK_FPAT_POSITION – define the origin of the fill pattern Raster used when rendering a primitive while using a stipple fill or pattern fill

SYNOPSIS

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

Xgl_ctx          ctx;

xgl_object_set(ctx, XGL_CTX_SURF_FRONT_FPAT_POSITION, Xgl_pt_i2d *, 0);
xgl_object_set(ctx, XGL_3D_CTX_SURF_BACK_FPAT_POSITION, Xgl_pt_i2d *, 0);

xgl_object_get(ctx, XGL_CTX_SURF_FRONT_FPAT_POSITION, Xgl_pt_i2d *);
xgl_object_get(ctx, XGL_3D_CTX_SURF_BACK_FPAT_POSITION, Xgl_pt_i2d *);
```

DESCRIPTION

These attributes define a point in device coordinates (DC) that specifies the origin of the stipple fill pattern Raster (front and back) used when a surface is filled with a pattern. The fill occurs during the rendering of a primitive. The origin is defined as an offset into the destination Raster.

To request stipple, opaque-stipple, pattern, or hatch fills, use the Context attributes **XGL_3D_CTX_SURF_BACK_FILL_STYLE**(3) and **XGL_CTX_SURF_FRONT_FILL_STYLE**(3). The Context attributes **XGL_CTX_SURF_FRONT_FPAT**(3) and **XGL_3D_CTX_SURF_BACK_FPAT**(3) define the fill pattern Memory Raster.

The default value is (0, 0).

These attributes are pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_3D_CTX_SURF_BACK_FPAT(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)

NAME

XGL_CTX_SURF_INTERIOR_RULE – defines how the interior of a polygon is determined

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_SURF_INTERIOR_RULE, Xgl_interior_rule, 0);
xgl_object_get(ctx, XGL_CTX_SURF_INTERIOR_RULE, Xgl_interior_rule *);
```

DESCRIPTION

This attribute determines the pixels that are *inside* and those that are *outside* of a polygon with multiple bounds. Single boundary polygons without overlapping edges are not affected by the setting of this attribute. This attribute affects only those polygons that have multiple boundaries or overlapping edges.

The value for this attribute is specified by the enumerated data type *Xgl_interior_rule* and is described below:

XGL_EVEN_ODD

A point is considered inside the polygon if, when an arbitrary line is drawn from the outside of the polygon to the point, the count of edge crossings is odd. In this case, the direction of the edges is not considered.

The default value is XGL_EVEN_ODD.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)

NAME

XGL_CTX_THRESHOLD – defines a threshold value for a primitive's size, below which the primitive is simplified before rendering

SYNOPSIS

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

Xgl_ctx         ctx;

xgl_object_set(ctx, XGL_CTX_THRESHOLD, Xgl_threshold *, 0);
xgl_object_get(ctx, XGL_CTX_THRESHOLD, Xgl_threshold *);
```

DESCRIPTION

This Context attribute is used to indicate when a primitive, or set of primitives, will be simplified before being rendered on the Device. It sets a size limit (in pixels). If the size of the current primitive is equal to or smaller than the size specified by the attribute, the primitive is *simplified* (using the rules described below) before the rendering. This is useful when the application has many small objects to render and it wants to render them as quickly as possible.

XGL_CTX_THRESHOLD is defined as a C structure of type *Xgl_threshold* that accepts three different Xgl_usgn32 threshold values for three different rendering types — *vectors*, *circles*, and *polygons*. The table below shows which drawing operator is associated with which rendering type:

vectors	*circles*	*polygons*
Markers	Circles	Rectangles
Stroke Text	Arcs	Polygons
Polylines	Ellipses	Multi_simple_polygons
Curves		Triangle strips
		Quadrilateral meshes

Each XGL primitive is provided with a bounding box. This bounding box information, after transformation to device coordinates, can be mapped to a portion of the device space and its area in pixels calculated.

If this area exceeds the corresponding threshold value, or if no bounding box is specified, the primitive is rendered normally.

If this area is less than or equal to the corresponding threshold value for the rendering type of the primitive, the primitive is simplified before being rendered. The simplification occurs as follows:

vectors are simplified by displaying only the edge between the first two vertices found by the primitive. Simplified vectors paint at least one pixel. The rules that apply for coloring lines created with **xgl_multipolyline**(3) also apply for coloring this simplified line.

circles are simplified by displaying a single pixel the same color as the conic, at a point as close to the conic center as possible.

polygons are simplified by converting them into triangles. The vertices of each triangle are the first three vertices in each polygon. If the simplified triangle is small enough, it will disappear because polygons do not overlap in XGL. The rules

that apply for coloring polygons also apply for coloring this simplified triangle.

The default value is {1, 1, 1}, so all primitives whose bounding boxes have an area of one pixel or less are simplified.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_multi_simple_polygon(3)
xgl_multiarc(3)
xgl_multicircle(3)
xgl_multi_elliptical_arc(3)
xgl_multimarker(3)
xgl_multipolyline(3)
xgl_multirectangle(3)
xgl_nurbs_curve(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_stroke_text(3)
xgl_triangle_strip(3)

NOTES

In a 3D environment, only the *x-y* projection of the bounding box is used to compare with the threshold level.

The threshold attribute can be ignored in certain circumstances. For example, the bounding box can be ignored by a particular accelerator, since it may be faster to render the primitive rather than to transform the bounding box and then test the covered area. Also, some accelerators may not implement the threshold attribute for any or all of the three rendering types.

The threshold is active only for the primitives that are completely inside the drawing area — that is, not clipped.

Each of the three threshold values is an integer.

NAME

XGL_CTX_VDC_MAP,
XGL_CTX_VDC_WINDOW,
XGL_CTX_DC_VIEWPORT – control mapping from Virtual Device Coordinates (VDC) to Device Coordinates (DC)

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

xgl_object_set(ctx, XGL_CTX_VDC_MAP, Xgl_vdc_map, 0);
xgl_object_get(ctx, XGL_CTX_VDC_MAP, Xgl_vdc_map *);

For a 2D Context:
xgl_object_set(ctx, XGL_CTX_VDC_WINDOW, Xgl_bounds_d2d *, 0);
xgl_object_set(ctx, XGL_CTX_DC_VIEWPORT, Xgl_bounds_d2d *, 0);

xgl_object_get(ctx, XGL_CTX_VDC_WINDOW, Xgl_bounds_d2d *);
xgl_object_get(ctx, XGL_CTX_DC_VIEWPORT, Xgl_bounds_d2d *);

For a 3D Context:
xgl_object_set(ctx, XGL_CTX_VDC_WINDOW, Xgl_bounds_d3d *, 0);
xgl_object_set(ctx, XGL_CTX_DC_VIEWPORT, Xgl_bounds_d3d *, 0);

xgl_object_get(ctx, XGL_CTX_VDC_WINDOW, Xgl_bounds_d3d *);
xgl_object_get(ctx, XGL_CTX_DC_VIEWPORT, Xgl_bounds_d3d *);

DESCRIPTION

These attributes control the mapping of the primitives' geometrical data from virtual device coordinates (VDC) to device coordinates (DC). The mapping transforms a rectangular region (an area for 2D Contexts and a volume for 3D Contexts), aligned with the axes, from VDC to DC. It cannot represent rotations or the perspective transform; these types of transformations are handled by the XGL Transform objects specified by the **XGL_CTX_LOCAL_MODEL_TRANS**(3), **XGL_CTX_GLOBAL_MODEL_TRANS**(3), and **XGL_CTX_VIEW_TRANS**(3) attributes.

The attribute **XGL_CTX_VDC_MAP**(3) specifies whether the application provides the mapping from VDC to DC, or whether XGL provides part of the mapping. Another Context attribute, **XGL_CTX_VDC_ORIENTATION**(3) controls the interpretation of coordinates in the VDC system.

XGL_CTX_VDC_MAP can take on one of four values as specified by the enumerated type *Xgl_vdc_map*:

XGL_VDC_MAP_OFF

The application specifies the VDC mapping by specifying both XGL_CTX_VDC_WINDOW and XGL_CTX_DC_VIEWPORT. In this case, to supply the correct regions for the mapping, the application must know the size limits of the Device and the value of XGL_CTX_VDC_ORIENTATION.

If the XGL_CTX_VDC_ORIENTATION is XGL_Y_DOWN_Z_AWAY, the VDC window points are mapped to the DC viewport points as follows:

VDC		DC
(xmin, ymin [, zmin])	→	*(xmin, ymin [, zmin])*

(xmax, ymax [, zmax]) → *(xmax, ymax [, zmax])*

If the XGL_CTX_VDC_ORIENTATION is XGL_Y_UP_Z_TOWARD, the VDC window points are mapped to the DC viewport points as follows:

VDC		DC
(xmin, ymin [, zmin])	→	*(xmin, ymax [, zmax])*
(xmax, ymax [, zmax])	→	*(xmax, ymin [, zmin])*

If you specify the VDC window or DC viewport bounds, XGL always requires that *xmin* < *xmax*, *ymin* < *ymax*, and, for 3D, *zmin* < *zmax*.

If the application tries to set the value of XGL_CTX_DC_VIEWPORT so that the DC viewport extends partly or entirely outside of the Device's boundaries, XGL will not accept this value. Instead, XGL will set XGL_CTX_DC_VIEWPORT to the entire extent of the Device. This is usually not the desired effect, so applications should detect this condition and act accordingly. If the DC viewport is partially outside the Device's boundaries, the application should reduce the size of the DC viewport so that it is entirely inside the Device, and it should reduce the VDC window by a proportional amount. If the DC viewport is entirely outside of the Device's boundaries, the application can set the value of XGL_CTX_RENDERING to FALSE so that primitives will not draw into the Device.

XGL_VDC_MAP_ALL

The application supplies the XGL_CTX_VDC_WINDOW and XGL sets the XGL_CTX_-DC_VIEWPORT to use the full range of the Device's coordinate space. This may result in the image being scaled differently along the *x*-axis and *y*-axis.

XGL_VDC_MAP_ASPECT

Similar to XGL_VDC_MAP_ALL, except that the aspect ratio (ratio of the width to the height) of the XGL_CTX_VDC_WINDOW is preserved in the transformation to DC. The VDC window is scaled so that it uses as much of the DC space as possible while maintaining its aspect ratio. Thus, not all of the DC space may be used. The top left corner of the XGL_CTX_VDC_WINDOW is mapped to the top left corner (0, 0) of the Device.

XGL_VDC_MAP_DEVICE

XGL_CTX_VDC_WINDOW is ignored. XGL sets the XGL_CTX_DC_VIEWPORT to use the full range of the Device's coordinate space, and sets the DC mapping so that VDC maps directly to DC; one unit in VDC is equal to one unit in DC.

In the case of a Raster Device, this allows the application to access pixels directly.

If the Device is a Window Raster Device, its size can be changed by the operator; this affects the VDC-to-DC mapping. When the window is resized, the application must adjust the VDC-to-DC map itself if the XGL_CTX_VDC_MAP is XGL_VDC_MAP_OFF. If the XGL_CTX_VDC_MAP is some other value, the application needs to call **xgl_window_raster_-resize**(3)**.**

The default value for XGL_CTX_VDC_MAP is XGL_VDC_MAP_DEVICE.

The default value for XGL_CTX_VDC_WINDOW is [-1, 1]×[-1, 1] for 2D Contexts and [-1, 1]×[-1, 1]×[0, 1] for 3D Contexts.

The default value for XGL_CTX_DC_VIEWPORT is [-1, 1]×[-1, 1] for 2D Contexts and [-1, 1]×[-1, 1]×[0, 1] for 3D Contexts. However, as long as XGL_CTX_VDC_MAP is different from XGL_VDC_MAP_OFF, each time the XGL_CTX_DEVICE is changed, XGL_CTX_DC_VIEWPORT is reset to [0, M_x]×[0, M_y] for 2D Contexts and [0, M_x]×[0, M_y]×[0, M_z] for 3D Contexts where (M_x, M_y, M_z) are a function of both XGL_CTX_VDC_MAP and the maximum dimensions of the device **XGL_DEV_MAXIMUM_COORDINATES**(3) (see the rules above).

These attributes are pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
xgl_window_raster_resize(3)
XGL_CTX_CLIP_PLANES(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VIEW_CLIP_BOUNDS(3)
XGL_CTX_VIEW_TRANS(3)
XGL_DEV_MAXIMUM_COORDINATES(3)

NOTE

In 3D, XGL does not view clip geometry at the *z* boundary of the VDC window (see **XGL_CTX_VDC_WINDOW**(3)). The application needs to specify and enable view clip planes in *z* to obtain view clipping in *z*. The planes are specified with **XGL_CTX_VIEW_CLIP_BOUNDS**(3) and enabled with **XGL_CTX_CLIP_PLANES**(3).

NAME

XGL_CTX_VDC_ORIENTATION – defines the orientation of Virtual Device Coordinate (VDC) space

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_VDC_ORIENTATION, Xgl_vdc_orientation, 0);
xgl_object_get(ctx, XGL_CTX_VDC_ORIENTATION, Xgl_vdc_orientation *);
```

DESCRIPTION

This attribute defines how virtual device coordinate (VDC) space is related to device coordinate (DC) space. DC space is the final coordinate space in the XGL pipeline. Essentially, the operator of the application program views DC space on the workstation screen. It is a right-handed coordinate space with the positive *x*-axis going from left to right, the positive *y*-axis going from top to bottom, and, for 3D, the positive *z*-axis going into the screen.

Since the VDC mapping (described in **XGL_CTX_VDC_MAP**(3)) only maps a rectangular region of VDC space to a rectangular region in DC, the orientation of VDC is the same as DC. Some applications, however, prefer to have a VDC space with the *y*-axis going from bottom to top. The attribute **XGL_CTX_VDC_ORIENTATION**(3) provides XGL the capability to change the orientation of VDC space.

VDC space is always right-handed; it can have the following orientations:

XGL_Y_DOWN_Z_AWAY
: The *x*-axis points to the right, the *y*-axis goes down, and the *z*-axis points away from the operator.

XGL_Y_UP_Z_TOWARD
: The *x*-axis points to the right, the *y*-axis goes up, and the *z*-axis points toward the operator.

The XGL_CTX_VDC_ORIENTATION attribute controls the interpretation of *up*, *down*, *near*, and *far* for other XGL attributes. The application must set the following attributes in accordance with the XGL_CTX_VDC_ORIENTATION to control what the operator sees:

XGL_CTX_GLOBAL_MODEL_TRANS
XGL_CTX_LOCAL_MODEL_TRANS
XGL_CTX_VIEW_TRANS
XGL_CTX_VIEW_CLIP_BOUNDS
XGL_CTX_VDC_MAP
XGL_CTX_VDC_WINDOW

The default value is XGL_Y_DOWN_Z_AWAY.

This attribute is not pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)
XGL_CTX_VDC_MAP(3)
XGL_CTX_VDC_WINDOW(3)
XGL_CTX_VIEW_TRANS(3)
XGL_CTX_VIEW_CLIP_BOUNDS(3)

NAME

XGL_CTX_VIEW_CLIP_BOUNDS – defines the Virtual Device Coordinate (VDC) clipping bounds

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ctx ctx;

For a 2D Context:
xgl_object_set(ctx, XGL_CTX_VIEW_CLIP_BOUNDS, Xgl_bounds_d2d *, 0);
xgl_object_get(ctx, XGL_CTX_VIEW_CLIP_BOUNDS, Xgl_bounds_d2d *);

For a 3D Context:
xgl_object_set(ctx, XGL_CTX_VIEW_CLIP_BOUNDS, Xgl_bounds_d3d *, 0);
xgl_object_get(ctx, XGL_CTX_VIEW_CLIP_BOUNDS, Xgl_bounds_d3d *);

DESCRIPTION

This attribute defines the clipping area (for 2D Contexts) or volume (for 3D Contexts) in ordinary virtual device coordinates (VDC). XGL converts the clipping area / volume to homogeneous VDC.

The attribute **XGL_CTX_VDC_ORIENTATION**(3) determines how the clipping bounds are oriented—that is, what is *up*, *down*, *near*, and *far*, with respect to the operator.

The values given for **XGL_CTX_VIEW_CLIP_BOUNDS**(3) are defined in VDC space and should be consistent with the values given for the XGL_CTX_VDC_WINDOW attribute. The clipping planes defined by XGL_CTX_VIEW_CLIP_BOUNDS can be individually controlled by the **XGL_CTX_CLIP_PLANES**(3) attribute.

The default value is [-1,1]x[-1,1] for 2D and [-1,1]x[-1,1]x[0,1] for 3D.

This attribute is pushed by **xgl_context_push**(3).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_CTX_CLIP_PLANES(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VDC_WINDOW(3)

NOTE

In 3D, XGL does not view clip geometry at the *z* boundary of the VDC window (see **XGL_CTX_VDC_WINDOW**(3)). The application needs to specify and enable view clip planes in *z* to obtain view clipping in *z*. The planes are specified with **XGL_CTX_VIEW_CLIP_BOUNDS**(3) and enabled with **XGL_CTX_CLIP_PLANES**(3).

NAME

XGL_CTX_VIEW_TRANS – defines the View Transform object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_ctx          ctx;
xgl_object_set(ctx, XGL_CTX_VIEW_TRANS, Xgl_trans, 0);
xgl_object_get(ctx, XGL_CTX_VIEW_TRANS, Xgl_trans *);
```

DESCRIPTION

This attribute defines the Transform object used for the View Transform, which transforms geometry data from world coordinates (WC) to virtual device coordinates (VDC).

For 3D, the View Transform may include the perspective transformation that would change the implicit *w* component of the geometric data from 1.0 to some other value. The geometry is projected into ordinary 3D space after the view clipping (specified by the attributes **XGL_CTX_VIEW_CLIP_BOUNDS**(3) and **XGL_CTX_CLIP_PLANES**(3)). This means that XGL treats VDC as both a homogeneous space (*x*, *y*, *z*, *w*) and an ordinary space (*x*, *y*, *z*).

The View Transform should be defined by the application in accordance with clipping bounds specified by XGL_CTX_VIEW_CLIP_BOUNDS and the orientation of VDC space defined by **XGL_CTX_VDC_ORIENTATION**(3)**.**

If you set this Context attribute to a Transform whose handle is obtained from the same or a different Context, the original attribute and this attribute use the same Transform. Changing the Transform of one Context attribute has the same effect on the other Context attribute. Be careful of this consequence when you share a Transform in this manner. Do not attempt to set this Context attribute to a handle to a Transform obtained from **XGL_CTX_MODEL_TRANS**(3)**, XGL_CTX_MC_TO_DC_TRANS**(3)**,** or **XGL_3D_CTX_NORMAL_TRANS**(3) because these Transforms are read-only.

The default value is the Identity Transform.

This attribute is pushed by **xgl_context_push**(3)**.**

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_push(3)
XGL_3D_CTX_NORMAL_TRANS(3)
XGL_CTX_CLIP_PLANES(3)
XGL_CTX_MC_TO_DC_TRANS(3)
XGL_CTX_MODEL_TRANS(3)
XGL_CTX_VDC_ORIENTATION(3)
XGL_CTX_VIEW_CLIP_BOUNDS(3)

NAME

XGL_DEV_COLOR_MAP – specifies the Color Map object associated with an XGL Device.

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dev                 dev;
xgl_object_set(dev, XGL_DEV_COLOR_MAP, Xgl_cmap, 0);
xgl_object_get(dev, XGL_DEV_COLOR_MAP, Xgl_cmap *);
```

DESCRIPTION

This attribute permits the application to set its own Color Map into a Device. The Color Map is used to specify the colors rendered into the Device. The attribute specifies the handle to the Color Map object associated with the Device *dev*. For more information about the utility of Color Map objects, see the operator **xgl_object_create**(3) and the Color Map attributes.

Once a Color Map object has been created, it is attached to the specified device using **xgl_object_set**(3) and this attribute.

The default value is a handle to a Color Map object with all default attributes.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_DEV_COLOR_TYPE – color type of data used at the Device level

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dev                 dev;
xgl_object_get(dev, XGL_DEV_COLOR_TYPE, Xgl_color_type *);
```

DESCRIPTION

This attribute defines the color space used to render and store pixels in an XGL Device. It can be set only once — when the Device object is created via a call to **xgl_object_-create**(3)**,** and it must be the first attribute in the attribute-list. After it is set, this attribute becomes read-only.

Two color spaces are currently supported in XGL as defined by the data type *Xgl_color_type*.

The two color spaces are defined as follows:

XGL_COLOR_RGB

Color is specified as a fractional component of each of the three primary colors, red, green, and blue. Each component is a floating-point number between 0 and 1, and represents the additive weight that the primary color contributes to the final color. If the underlying hardware is indexed (see XGL_COLOR_INDEX below), a *color cube* is used to transform the RGB Device colors to what the hardware requires. In this case, the color cube is stored in the color table that is part of the XGL Color Map object. (For more information on color cubes, see **XGL_CMAP_-COLOR_CUBE_SIZE**(3) and **XGL_CMAP_COLOR_CUBE_BSIZE**(3)).

XGL_COLOR_INDEX

Color is specified as an unsigned integer index into a color table. The color table (see **XGL_CMAP_COLOR_TABLE**(3)) is part of an XGL Color Map object, and it maps a color index into an RGB color value.

The default value varies depending on whether the Device is a Memory Raster or Window Raster.

For Memory Rasters:

XGL_DEV_COLOR_TYPE defaults to XGL_COLOR_INDEX.

For Window Rasters:

XGL_DEV_COLOR_TYPE defaults to the hardware (display device) color type.

SEE ALSO

xgl_object_set(3)
xgl_object_create(3)
XGL_CMAP_COLOR_CUBE_BSIZE(3)
XGL_CMAP_COLOR_CUBE_SIZE(3)

XGL_CMAP_COLOR_TABLE(3)
XGL_RAS_DEPTH(3)

NAME

XGL_DEV_CONTEXTS,
XGL_DEV_CONTEXTS_NUM – return the Context objects associated with a Device

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dev                 dev;
xgl_object_get(dev, XGL_DEV_CONTEXTS, Xgl_ctx []);
xgl_object_get(dev, XGL_DEV_CONTEXTS_NUM, Xgl_usgn32 *);
```

DESCRIPTION

Both of these attributes are read-only. They are used by the application to get a list of the Context object handles associated with the Device *dev*. The Context attribute **XGL_CTX_DEVICE**(3) is used to associate a Device with a Context.

To use these attributes, the application should first get XGL_DEV_CONTEXTS_NUM, which is the number of Context objects associated with the Device. Then, it should allocate enough memory to hold this number of Context object handles (where each object handle is *sizeof(Xgl_ctx)* in size). Finally, the application can get **XGL_DEV_CONTEXTS**(3), which returns the list of Context handles in the array provided by the application.

For XGL_DEV_CONTEXTS_NUM, the default is 0.
For XGL_DEV_CONTEXTS, the default is NULL.

SEE ALSO

xgl_object_get(3)
XGL_CTX_DEVICE(3)

NAME

XGL_DEV_MAXIMUM_COORDINATES – returns the maximum coordinate values for the specified Device

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dev                 dev;
xgl_object_get(dev, XGL_DEV_MAXIMUM_COORDINATES, Xgl_pt_d3d *);
```

DESCRIPTION

This read-only attribute is used to interrogate the Device *dev* to get its maximum *x*- and *y*-coordinate values. Also, it's maximum *z*-buffer value, if applicable, is written into the *z* position of the *Xgl_pt_d3d* point that is returned by the **xgl_object_get**(3) operation. For example, assume that the application is using an XGL Window Raster that is associated with a 500×600 window created by the window system. A call to xgl_object_get on XGL_DEV_MAXIMUM_COORDINATES would produce a point containing: (499, 599, max-Z). *max-Z* would contain the largest *z*-buffer value allowed by the Device.

The default value is device-dependent.

SEE ALSO

xgl_object_get(3)

XGL_CTX_DEVICE(3)

NAME

XGL_DEV_REAL_COLOR_TYPE – color type of the underlying device

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dev                 dev;
xgl_object_get(dev, XGL_DEV_REAL_COLOR_TYPE, Xgl_color_type *);
```

DESCRIPTION

This read-only attribute indicates the real color space used to store pixels in an XGL Device, in other words, the kind of information that is contained in an actual pixel. The exact way a pixel is defined is device dependent.

Two color spaces are currently supported in XGL as defined by the data type *Xgl_color_type*.

The two color spaces are defined as follows:

XGL_COLOR_RGB
: The color stored in the device includes a component for each of the three primary colors - red, green, and blue.

XGL_COLOR_INDEX
: The color stored in the device is an index into a color table.

For Memory Rasters:
: If XGL_RAS_DEPTH = 4 or 8, then XGL_DEV_REAL_COLOR_TYPE = XGL_COLOR_INDEX.
If XGL_RAS_DEPTH = 32, then XGL_DEV_REAL_COLOR_TYPE = XGL_COLOR_RGB (see **XGL_RAS_DEPTH**(3)**).**

For Window Rasters:
: XGL_DEV_REAL_COLOR_TYPE is device dependent. Most often it matches the underlying X Visual Class of the X window used to create the XGL Window Raster.

SEE ALSO

xgl_object_set(3)

XGL_RAS_DEPTH(3)

NAME

XGL_DMAP_TEXTURE – a Data Map Texture object

OVERVIEW

XGL_DMAP_TEXTURE is the *type* of an object representing an XGL Data Map Texture object. It is used in texturing of surfaces (polygons, triangle strips, etc.) rendered through a 3D Context.

When a Data Map Texture object is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

After creating the Data Map Texture object, using **xgl_object_create**(3)**,** an application can inquire about the value of its attributes by using the **xgl_object_get**(3) operator.

A System State object must have been created with the **xgl_open**(3) operator before any Data Map Texture object can be created.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_DMAP_TEXTURE_DESCRIPTORS(3)
Defines a list of texture descriptors.

XGL_DMAP_TEXTURE_NUM_DESCRIPTORS(3)
Defines the number of texture descriptors in the object.

XGL_DMAP_TEXTURE_ORIENTATION_MATRIX(3)
Defines a 3×2 matrix by which textures can be rotated and/or translated before applying it to a 3D surface.

XGL_DMAP_TEXTURE_PARAM_TYPE(3)
Defines different parameterization methods of mapping texture onto a 3D surface.

XGL_DMAP_TEXTURE_U_INDEX(3)
XGL_DMAP_TEXTURE_V_INDEX(3)
These attributes are indices into the vertex data array (specified in the point list), and they indicate which of the data values to use in texture mapping.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_DMAP_TEXTURE_DESCRIPTORS – specifies list of *textures* in a Data Map Texture object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dmap_texture        dmap_texture;
xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_DESCRIPTORS, Xgl_texture_desc, 0);
xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_DESCRIPTORS, Xgl_texture_desc *);
```

DESCRIPTION

This attribute specifies the list of *textures* associated with this Data Map Texture object. The *textures* are used when rendering XGL 3D surface primitives. When a data map texture has multiple texture descriptions, they are applied sequentially and are cumulative.

The application uses this attribute in conjunction with the attribute **XGL_DMAP_TEXTURE_NUM_DESCRIPTORS**(3) to specify how many *textures* should be considered when rendering XGL primitives.

Setting a new list of *textures* replaces the previous (if any) list of *textures*. The application is also responsible for making sure that enough memory is available when getting the list of *textures* from XGL.

XGL_DMAP_TEXTURE_DESCRIPTORS is a pointer to a C structure of type *Xgl_texture_desc*, which is defined as follows:

```
typedef struct {
        Xgl_texture_type                texture_type
        Xgl_texture_color_comp_info     comp_info;
        Xgl_texture_interp_info         interp_info;  /* filter to use */
        union {
                Xgl_texture_mipmap_desc   mipmap;
        } info;
} Xgl_texture_desc;
```

Parameters

texture_type

Each texture descriptor has a *texture_type*. Currently, XGL_TEXTURE_TYPE_MIPMAP is the only possible value for this attribute.

comp_info

This parameter explains how different channels in the texture map are used to affect the rendering. The *Xgl_texture_color_comp_info* structure is defined as:

```
typedef struct {
        Xgl_usgn32                  num_render_comp_desc;
        Xgl_render_component_desc   render_component_desc[2];
        Xgl_usgn32                  num_channels[2];
        Xgl_usgn32                  channel_number[2];
} Xgl_texture_color_comp_info;
```

This structure can have a maximum of 2 render components. The *num_render_comp_desc* parameter gives the number of render components used by this

particular Texture Map object. Note that in the current release of XGL, only *num_render_comp_desc* of 1 is supported. The *num_channels[2]* parameter gives the number of channel used to do color composition. The *channel_number[2]* indicates the channel number used when *num_channel* is 1. The *Xgl_render_-component_desc* structure is defined as:

```
typedef struct {
        Xgl_render_component    comp;
        Xgl_texture_op          texture_op;
        union                   Xgl_op_info {
                Xgl_texture_blend_op            blend;
                Xgl_texture_decal_op            decal;
                Xgl_texture_blend_intrinsic_op  blend_int;
        } op;
} Xgl_render_component_desc;
```

The *comp* parameter determines which lighting component in the rendering pipeline (diffuse, reflected, and final-after depth cueing) is affected by this particular texture description. The *texture_op* parameter specifies how the current color value is combined with the value obtained from the texture map. The table below lists the valid combinations:

Texture Operations

# of Channel	Replace	Modulate	Decal	Blend	Decal Intrinsic	Blend Intrinsic
1		$C_f = S_t C_i$		$C_f = C_{k1} * (1-S_t) + C_{k2} * S_t$		$C_f = C_i * (1-S_t) + C_{k1} * S_t$
2		$C_f = S_t C_i$		$C_f = C_{k1} * (1-S_t) + C_{k2} * S_t$		$C_f = C_i * (1-S_t) + C_{k1} * S_t$
3	$C_f = C_t$	$C_f = C_t C_i$		$C_f = C_{k1} * (1-C_t) + C_{k2} * C_t$		$C_f = C_i * (1-C_t) + C_{k1} * C_t$
4	$C_f = C_t$	$C_f = C_t C_i$	$C_f = (1-\alpha_t) * C_{k1} + \alpha_t * C_t$	$C_f = C_{k1} * (1-C_t) + C_{k2} * C_t$	$C_f = C_i * (1-C_t) + \alpha_t * C_t$	$C_f = C_i * (1-C_t) + C_{k1} * C_t$

Where: C_{k1} = Constant color 1
C_{k2} = Constant color 2
C_i = Incoming color
C_f = Final color
S_t = Single channel texture image
C_t = First three channels of a 4 channel texture image
α_t = Second channel of a 2 channel texture image or
Fourth channel of a 4 channel texture image

Note: Blank fields indicate undefined values. The values from the texture image are in the range [01]. Currently in XGL, a 3 channel texture image is specified as a 32-bit value. The upper byte is ignored for 3 channel texture operations.

In the *Xgl_op_info* union, the texture operations *blend*, *decal*, and *blend_int* use values defined in the structures *Xgl_texture_blend_op*, *Xgl_texture_decal_op*, and *Xgl_texture_blend_intrinsic_op*, respectively. The blending operation gets the constant colors C_{k1} and C_{k2} from the structure *Xgl_texture_blend_op*. The *decal* and *blend_int* operations get the constant color C_{k1} from the structures *Xgl_texture_-decal_op* and *Xgl_texture_blend_intrinsic_op*, respectively.

interp_info

This parameter gives information on the nature of sampling desired by the application. The *Xgl_texture_interp_info* structure is defined as:

```
typedef struct {
        Xgl_texture_interp_method    filter1;
        Xgl_texture_interp_method    filter2;
} Xgl_texture_interp_info;
```

During the texturing process, the data from the texture mipmap can be sampled or interpolated in a number of ways in order to come up with a value to apply to the pixel. The available methods of sampling and interpolating the mipmap are specified by the *Xgl_texture_interp_method* structure. The possible values are:

Note: The word *texel* used below refers to one element stored in a texture map.

XGL_TEXTURE_INTERP_POINT
: Sample one texel point from the base (top or first) mipmap level at the closest u and v to the pixel.

XGL_TEXTURE_INTERP_BILINEAR
: Sample the four texels closest to the center of the pixel from the base mipmap level and average them.

XGL_TEXTURE_INTERP_MIPMAP_POINT
: Sample one texel point from the nearest mipmap level at the closest u and v to the pixel.

XGL_TEXTURE_INTERP_MIPMAP_BILINEAR
: Sample the four texels closest to the center of the pixel from the nearest mipmap level and average them.

XGL_TEXTURE_INTERP_MIPMAP_TRILINEAR
: Sample the four texels closest to the center of the pixel from the nearest mipmap level above the depth *d* and the four texels closest from the mipmap level below the depth *d* and average all 8 of them.

XGL_TEXTURE_INTERP_MIPMAP_PT_LINEAR
: Chose the two mipmaps that most closely match the size fo the pixel being textured and find the nearest to the center of the pixel to produce a texture value from each mipmap. The final texture value is an average of those two values.

filter1 is used if the textured pixel maps to an area greater than one texel. *filter2* is used if the textured pixel maps to an area less than or equal to one texel. We

strongly recommend that *filter2* only use XGL_TEXTURE_INTERP_POINT or XGL_-TEXTURE_INTERP_BILINEAR.

mipmap This parameter defines the mipmap specific fields. The *Xgl_texture_mipmap_desc* structure is defined as:

```
typedef struct {
        Xgl_texture             texture_map;
        float                   max_u_freq;
        float                   max_v_freq;
        Xgl_texture_boundary    u_boundary;
        Xgl_texture_boundary    v_boundary;
        Xgl_usgn8               boundary_values[4];
        float                   depth_interp_factor;
} Xgl_texture_mipmap_desc;
```

The *texture_map* parameter gives a pointer to the texture map object. A texture map object is created using **xgl_object_create**(3) with XGL_MIPMAP_TEXTURE as the object type.

Note: At this release, the fields *max_u_freq* and *max_v_freq* are not implemented.

The parameters *u_boundary* and *v_boundary* specify the boundary condition to apply when texture bounds are exceeded. The available boundary options are specified by the *Xgl_texture_boundary* structure. The possible values are:

XGL_TEXTURE_BOUNDARY_CLAMP
: Finish texturing using a constant value.

XGL_TEXTURE_BOUNDARY_WRAP
: Repeat the texture.

XGL_TEXTURE_BOUNDARY_TRANSPARENT
: Finish rendering as though texturing were not being performed.

XGL_TEXTURE_BOUNDARY_MIRROR
: Reverse texel sampling creating a *mirrored alternating* pattern.

XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY
: Use a color specified by the closest boundary texel.

Note that if *u* and *v* boundary conditions are not identical, and if both values of *u* and *v* exceed the texture boundary in both directions, the boundary methods will be used in the following order:

(1) XGL_TEXTURE_BOUNDARY_TRANSPARENT

(2) XGL_TEXTURE_BOUNDARY_CLAMP

(3) XGL_TEXTURE_BOUNDARY_WRAP

(4) XGL_TEXTURE_BOUNDARY_MIRROR

(5) XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY

The *boundary_values* parameter specifies the value to clamp to when either *u* or *v* value exceeds bounds. The *depth_interp_factor* parameter is a floating point

number that allows the application to adjust the sample level in either direction to obtain a more or less aliased textured surface. A value of 1.0 moves the depth away from the mipmap base creating a blurry event. A value of -1.0 moves the depth towards the mipmap pyramid base for a more detailed picture.

The default value is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_DMAP_TEXTURE_NUM_DESCRIPTORS(3)

NOTES

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_DMAP_TEXTURE_NUM_DESCRIPTORS – specifies the number of *textures* in a Data Map Texture object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dmap_texture        dmap_texture;
xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_NUM_DESCRIPTORS, Xgl_usgn32, 0);
xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_NUM_DESCRIPTORS, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the number of *textures* to apply to the surface and the number of texture descriptor structures that follow within the Data Map Texture object.

The application uses this attribute in conjunction with the attribute **XGL_DMAP_TEXTURE_-DESCRIPTORS**(3) to specify which and how many *textures* should be considered when rendering XGL primitives.

When additional texture descriptors must be defined, the application must provide both the number of textures and the list of texture descriptors. To use only the first *n* textures of those already defined, the application need only set the new number of textures.

The default value is 0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_DMAP_TEXTURE_DESCRIPTORS(3)

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_DMAP_TEXTURE_ORIENTATION_MATRIX – defines a matrix by which *textures* can be translated and/or rotated

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dmap_texture        dmap_texture;
xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_ORIENTATION_MATRIX,
        Xgl_matrix_f2d *, 0);
xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_ORIENTATION_MATRIX,
        Xgl_matrix_f2d *);
```

DESCRIPTION

This attribute specifies a 3×2 matrix by which textures can be rotated or translated across the face of the polygon by transforming the u and v values through the matrix. The u and v values are specified as data values in the point list of a surface primitive (e.g. xgl_-multi_simple_polygon, xgl_triangle_strip, e.g.). The u, v values defined in the point list are subject to the **XGL_DMAP_TEXTURE_U_INDEX**(3) and **XGL_DMAP_TEXTURE_V_INDEX**(3) attributes before the orientation matrix is applied.

The default value is the Identity Matrix.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_multi_simple_polygon(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)
xgl_triangle_strip(3)
XGL_DMAP_TEXTURE_U_INDEX(3)
XGL_DMAP_TEXTURE_V_INDEX(3)

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_DMAP_TEXTURE_PARAM_TYPE – defines the parameterization method for texture mapping

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_dmap_texture        dmap_texture;
xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_PARAM_TYPE,
        Xgl_texture_param_type *, 0);
xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_PARAM_TYPE,
        Xgl_texture_param_type *);
```

DESCRIPTION

This attribute defines the parameterization method to use in mapping the texture to a 3D surface. XGL accepts the following value for *Xgl_texture_param_type*:

XGL_TEXTURE_PARAM_EXPLICIT
: This attribute value requires the application to provide the values of *u* and *v* as part of the vertex information in the point list of 3D surface primitives.

The default value is XGL_TEXTURE_PARAM_EXPLICIT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_DMAP_TEXTURE_U_INDEX,
XGL_DMAP_TEXTURE_V_INDEX – define the indices into the vertex data array

SYNOPSIS

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

Xgl_dmap_texture        dmap_texture;

xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_U_INDEX, Xgl_usgn32, 0);
xgl_object_set(dmap_texture, XGL_DMAP_TEXTURE_U_INDEX, Xgl_usgn32, 0);

xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_V_INDEX, Xgl_usgn32 *);
xgl_object_get(dmap_texture, XGL_DMAP_TEXTURE_V_INDEX, Xgl_usgn32 *);
```

DESCRIPTION

These attributes are indices into the vertex data array, and they indicate which of the data values to use as u and v texture coordinates.

The values of u and v are specified as real numbers in the vertex array. The coordinate space of the texture map that u values correspond to is a space that places 0.0 at the left edge of the texture and 1.0 at the right edge of the texture. Similarly for v, the coordinate space of the texture map that v values use is a space that places 0.0 at the bottom edge of the texture and 1.0 at the top edge of the texture.

If u and v values are not assigned to polygon vertices in a linear way, that is, if there is no linear transformation that maps u,v coordinates to the polygon's vertices, then a textured polygon may look different when rendered on different graphics accelerators. This is because some graphics hardware devices break up more complex polygons into triangles, and if the mapping is non-linear, the triangles won't all be textured the same from device to device.

The default value for XGL_DMAP_TEXTURE_U_INDEX is 0.

The default value for XGL_DMAP_TEXTURE_V_INDEX is 1.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NOTES

At this release, a new object, Texture Map, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but will be removed from the XGL library in a future release. Because of this, we recommend that application programs use the Texture Map object for texture mapping.

NAME

XGL_GCACHE – a Geometry Cache (Gcache) object

OVERVIEW

XGL_GCACHE is the *type* parameter value for a Geometry Cache object.

A Geometry Cache (Gcache) is an XGL object that stores a simplified version of an application primitive. In addition, the Gcache stores the values of the attributes on which the simplification is based. Further, for 3D primitives, if facet normals are not provided with the primitive, they will be calculated and stored in the Gcache facet list structure. For a complete description of XGL_GCACHE and its use, see the *XGL Programmer's Guide*.

When the Gcache is displayed, the application can validate the Gcache using the current values of the Context attributes. If the simplification is still valid (the attribute values are still the same), the simplified form of the primitive is rendered.

Geometry Cache (Gcache) uses the following attributes:

XGL_GCACHE_BYPASS_MODEL_CLIP(3)
: Controls whether model clipping is used when a Gcache is displayed with the *test* parameter to xgl_context_display_gcache indicated as FALSE.

XGL_GCACHE_DISPLAY_PRIM_TYPE(3)
: Returns the type of primitive used when the Gcache is displayed.

XGL_GCACHE_DO_POLYGON_DECOMP(3)
: Controls whether polygon decomposition is performed when **xgl_gcache_polygon**(3) is called.

XGL_GCACHE_FACET_LIST_LIST(3)
: Returns the list of facet lists used when the Gcache is displayed.

XGL_GCACHE_IS_EMPTY(3)
: Controls whether the Gcache is empty. When set to TRUE, the attribute causes the Gcache to be emptied.

XGL_GCACHE_NURBS_CURVE_MODE(3)
: Defines the mode of Gcache for NURBS curves.

XGL_GCACHE_NURBS_SURF_MODE(3)
: Defines the mode of Gcache for NURBS surfaces.

XGL_GCACHE_ORIG_PRIM_TYPE(3)
: Returns the type of the original primitive stored in the Gcache.

XGL_GCACHE_POLYGON_TYPE(3)
: Controls which decomposition algorithm is used if polygon decomposition is performed when **xgl_gcache_polygon**(3) is called.

XGL_GCACHE_PT_LIST_LIST(3)
: Returns the list of point lists used when the Gcache is displayed.

XGL_GCACHE_SHOW_DECOMP_EDGES(3)
: Controls whether decomposition edges are shown when **xgl_gcache_polygon**(3) or **xgl_gcache_multi_simple_polygon**(3) is displayed.

XGL_GCACHE_USE_APPL_GEOM(3)

Controls whether the Gcache references the application's geometry (the parameters to the Gcache operators, such as the list of points, bounding box, and facet information).

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

NAME

XGL_GCACHE_BYPASS_MODEL_CLIP – controls whether model clipping is used when a Gcache is displayed

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_gcache      gcache;
xgl_object_set(gcache, XGL_GCACHE_BYPASS_MODEL_CLIP, Xgl_boolean, 0);
xgl_object_get(gcache, XGL_GCACHE_BYPASS_MODEL_CLIP, Xgl_boolean *);
```

DESCRIPTION

A Gcache stores the geometry in model coordinates (MC). When the Gcache is displayed, the stored geometry is potentially subject to model clipping. This attribute controls whether model clipping is bypassed (not used) when a Gcache is displayed with the *test* parameter to xgl_context_display_gcache as FALSE. Since the *test* parameter is FALSE, in a sense the application is indicating that it does not care about the state of the Context for display versus the state of the Context when the Gcache was created. If this attribute is FALSE, model clipping is applied to the display of the cached geometry. If it is TRUE, model clipping is not applied.

The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_display_gcache(3)

NAME

XGL_GCACHE_DISPLAY_PRIM_TYPE – returns the type of primitive used when Gcache is displayed

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_gcache      gcache;
xgl_object_get(gcache, XGL_GCACHE_DISPLAY_PRIM_TYPE, Xgl_primitive_type *);
```

DESCRIPTION

This read-only attribute returns the type of primitive used when the Gcache is displayed. **xgl_gcache_stroke_text**(3) is called, The display primitive is used internally to display the Gcache data. If XGL_GCACHE_DISPLAY_PRIM_TYPE is not the original primitive type, some sort of optimization or simplification was performed on the data by XGL. This can happen, for instance, when a complex polygon is tessellated (broken down) into triangles, in which case xgl_multi_simple_polygon is used for rendering. For a complete description of how various primitives are handled by Gcaches, see the Man page for the Gcache version of the primitive.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_display_gcache(3)
xgl_enum_types(3)

NAME

XGL_GCACHE_DO_POLYGON_DECOMP – controls polygon decomposition when xgl_gcache_polygon or xgl_gcache_multi_simple_polygon is called

SYNOPSIS

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

Xgl_gcache      gcache;

xgl_object_set(gcache, XGL_GCACHE_DO_POLYGON_DECOMP, Xgl_boolean, 0);
xgl_object_get(gcache, XGL_GCACHE_DO_POLYGON_DECOMP, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether polygon decomposition is performed when **xgl_gcache_polygon**(3) or **xgl_gcache_multi_simple_polygon**(3) is called. If **XGL_GCACHE_DO_POLYGON_DECOMP**(3) is set to TRUE, any subsequent polygons that are cached will be stored as a decomposed list of triangles. If set to FALSE, decomposition will not be performed. For more information about polygon decomposition, see **xgl_gcache_polygon**(3) and **xgl_gcache_multi_simple_polygon**(3).

The default value is FALSE.

SEE ALSO

xgl_gcache_polygon(3)
xgl_gcache_multi_simple_polygon(3)

NAME

XGL_GCACHE_FACET_LIST_LIST – returns the list of facet lists used when the Gcache is displayed

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_gcache      gcache;
xgl_object_get(gcache, XGL_GCACHE_FACET_LIST_LIST, Xgl_facet_list_list *);
```

DESCRIPTION

This read-only attribute returns the list of facet lists used when the Gcache is displayed. The *Xgl_facet_list_list* will be in the format required for rendering, using the primitive returned by XGL_GCACHE_DISPLAY_PRIM_TYPE. For a detailed description of this list, see the page for the specific XGL primitive in this manual.

SEE ALSO

XGL_GCACHE_DISPLAY_PRIM_TYPE(3)

NAME

XGL_GCACHE_IS_EMPTY – controls whether the Gcache is empty

SYNOPSIS

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

Xgl_gcache          gcache;

xgl_object_set(gcache, XGL_GCACHE_IS_EMPTY, Xgl_boolean, 0);
xgl_object_get(gcache, XGL_GCACHE_IS_EMPTY, Xgl_boolean *);
```

DESCRIPTION

When set to TRUE, this attribute causes the Gcache to be emptied. When an empty Gcache is displayed, nothing is drawn. Emptying a Gcache does not free the resources of the Gcache. To do that, the Gcache must be destroyed. Setting this attribute to FALSE does not affect the Gcache. It is a NO-OP.

This attribute returns TRUE if the Gcache is empty (has no geometry stored in it). A value of FALSE means that the Gcache has some geometry in it.

The default value is TRUE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_destroy(3)
xgl_context_display_gcache(3)

NAME

XGL_GCACHE_NURBS_CURVE_MODE – defines the mode of Gcache for NURBS curves

SYNOPSIS

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

Xgl_gcache          gcache;

xgl_object_set(gcache, XGL_GCACHE_NURBS_CURVE_MODE, Xgl_gcache_nurbs_mode, 0);
xgl_object_get(gcache, XGL_GCACHE_NURBS_CURVE_MODE, Xgl_gcache_nurbs_mode *);
```

DESCRIPTION

This attribute controls the Gcache mode when **xgl_gcache_nurbs_curve**(3) or **xgl_gcache_multi_elliptical_arc**(3) operators are called.

The alternatives for this attribute are specified by the enumerated data type *Xgl_gcache_nurbs_mode* and are described below:

XGL_GCACHE_NURBS_DYNAMIC
: A tessellation-independent Gcache is created. At each frame, the curve is retessellated according to the current context. Memory consumption in this case is minimal.

XGL_GCACHE_NURBS_STATIC
: At Gcache creation time, the curve is tessellated, and the result is stored in the tessellation Gcache in the form of polylines. Dynamic modification of the curve appearance according to NURBS-specific context attributes is not possible. Memory consumption in this case could be considerable.

XGL_GCACHE_NURBS_COMBINED
: Both tessellation-independent and tessellation gcaches are created. Portions of the curve are retessellated only if it is necessary according to the current context, else the tessellation Gcache is used. Thus, dynamic modification of the curve appearance is possible. Memory consumption in this case could be considerable.

The default value is XGL_GCACHE_NURBS_DYNAMIC.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_gcache_multi_elliptical_arc(3)
xgl_gcache_nurbs_curve(3)

NAME

XGL_GCACHE_NURBS_SURF_MODE – defines the mode of Gcache for NURBS surfaces

SYNOPSIS

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

Xgl_gcache          gcache;

xgl_object_set(gcache, XGL_GCACHE_NURBS_SURF_MODE, Xgl_gcache_nurbs_mode, 0);
xgl_object_get(gcache, XGL_GCACHE_NURBS_SURF_MODE, Xgl_gcache_nurbs_mode *);
```

DESCRIPTION

This attribute controls the Gcache mode when **xgl_gcache_nurbs_surface**(3) operator is called.

The alternatives for this attribute are specified by the enumerated data type *Xgl_gcache_-nurbs_mode* and are described below:

XGL_GCACHE_NURBS_DYNAMIC
: A tessellation-independent Gcache is created. At each frame, the surface is retessellated according to the current context. Memory consumption in this case is minimal.

XGL_GCACHE_NURBS_STATIC
: At Gcache creation time, the surface is tessellated, and the result is stored in the tessellation Gcache in the form of polygons (quadrilateral meshes, triangle strips, and/or polylines). Dynamic modification of the surface appearance according to NURBS-specific context attributes is not possible. Memory consumption in this case could be considerable.

XGL_GCACHE_NURBS_COMBINED
: Both tessellation-independent and tessellation gcaches are created. Portions of the surface are retessellated only if it is necessary according to the current context, else the tessellation Gcache is used. Thus, dynamic modification of the surface appearance is possible. Memory consumption in this case could be considerable.

The default value is XGL_GCACHE_NURBS_DYNAMIC.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_gcache_nurbs_surface(3)

NAME

XGL_GCACHE_ORIG_PRIM_TYPE – returns the type of original primitive stored in the Gcache

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_gcache      gcache;
xgl_object_get(gcache, XGL_GCACHE_ORIG_PRIM_TYPE, Xgl_primitive_type *);
```

DESCRIPTION

This read-only attribute returns the type of the original primitive stored in the Gcache. For example, if **xgl_gcache_stroke_text**(3) is called, this attribute returns XGL_PRIM_-STROKE_TEXT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_display_gcache(3)
xgl_enum_types(3)

NAME

XGL_GCACHE_POLYGON_TYPE – controls which decomposition algorithm is used when xgl_gcache_polygon is called

SYNOPSIS

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

Xgl_gcache      gcache;

xgl_object_set(gcache, XGL_GCACHE_POLYGON_TYPE, Xgl_polygon_type, 0);
xgl_object_get(gcache, XGL_GCACHE_POLYGON_TYPE, Xgl_polygon_type *);
```

DESCRIPTION

This attribute controls which decomposition algorithm is used if polygon decomposition is performed when **xgl_gcache_polygon**(3) is called. The *Xgl_polygon_type* value of XGL_POLYGON_NSI means the polygon is non–self-intersecting. The value XGL_POLYGON_COMPLEX means the polygon can be arbitrarily complex in shape. For more information about polygon decomposition, see **xgl_gcache_polygon**(3).

The default value is XGL_POLYGON_COMPLEX.

SEE ALSO

xgl_enum_types(3)
xgl_gcache_polygon(3)
XGL_GCACHE_DO_POLYGON_DECOMP(3)

NAME

XGL_GCACHE_PT_LIST_LIST – returns the list of point lists used when the Gcache is displayed

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_gcache      gcache;
xgl_object_get(gcache, XGL_GCACHE_PT_LIST_LIST, Xgl_pt_list_list *);
```

DESCRIPTION

This read-only attribute returns the list of point lists used when the Gcache is displayed. The *Xgl_pt_list_list* will be in the format required for rendering, using the primitive returned by **XGL_GCACHE_DISPLAY_PRIM_TYPE**(3)**.** For a detailed description of this list, see the page for the specific XGL primitive in this manual.

SEE ALSO

xgl_struct(3)

XGL_GCACHE_DISPLAY_PRIM_TYPE(3)

NAME

XGL_GCACHE_SHOW_DECOMP_EDGES – controls whether decomposition edges are shown when **xgl_gcache_polygon**(3) or **xgl_gcache_multi_simple_polygon**(3) is displayed

SYNOPSIS

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

Xgl_gcache      gcache;

xgl_object_set(gcache, XGL_GCACHE_SHOW_DECOMP_EDGES, Xgl_boolean, 0);
xgl_object_get(gcache, XGL_GCACHE_SHOW_DECOMP_EDGES, Xgl_boolean *);
```

DESCRIPTION

If both this attribute and **XGL_CTX_SURF_EDGE_FLAG**(3) are TRUE, and the **xgl_gcache_polygon**(3) or **xgl_gcache_multi_simple_polygon**(3) being displayed was decomposed, the edges outlining the decomposition are shown. The color is determined by the attribute **XGL_CTX_EDGE_COLOR**(3). If XGL_GCACHE_SHOW_DECOMP_EDGES is FALSE, the decomposition edges are not shown.

The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_gcache_polygon(3)
xgl_gcache_multi_simple_polygon(3)
XGL_CTX_EDGE_COLOR(3)
XGL_GCACHE_DO_POLYGON_DECOMP(3)

NAME

XGL_GCACHE_USE_APPL_GEOM – controls whether the application's geometry is referenced

SYNOPSIS

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

Xgl_gcache          gcache;

xgl_object_set(gcache, XGL_GCACHE_USE_APPL_GEOM, Xgl_boolean, 0);
xgl_object_get(gcache, XGL_GCACHE_USE_APPL_GEOM, Xgl_boolean *);
```

DESCRIPTION

When set to TRUE, this attribute causes the Gcache to try to reference the application's geometry (the parameters to the Gcache operators, such as the list of points, bounding box, and facet information) in the Gcache. When it is FALSE, the Gcache copies the application's geometry into the Gcache.

If this attribute is TRUE, it is the application's responsibility to keep the geometry allocated. The application's geometry must not be freed until after the contents of the Gcache have been either emptied or replaced by another geometry, or the Gcache has been destroyed.

The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_destroy(3)

NAME

XGL_LIGHT – a Light object

OVERVIEW

XGL_LIGHT is the type of an object representing an XGL Light object. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

A Light object is an abstraction of a physical light source. It is used to illuminate surfaces (polygons, triangle strips, and so on) rendered by a 3D Context. A Light object can simulate different types of light sources; the more realistic the model of the light source, the more computation is needed.

When a Light is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

After creating the Light, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

A System State object must have been created with the **xgl_open**(3) operator before any Light can be created.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_LIGHT_ATTENUATION_1(3)
XGL_LIGHT_ATTENUATION_2(3)

Define the light attenuation coefficients for positional or spot lights.

XGL_LIGHT_COLOR(3)

Defines the color of a light.

XGL_LIGHT_DIRECTION(3)

Defines the direction of light propagation for a spot or directional light.

XGL_LIGHT_POSITION(3)

Defines the position in World Coordinates of a positional or spot light.

XGL_LIGHT_SPOT_ANGLE(3)
XGL_LIGHT_SPOT_EXPONENT(3)

Define the intensity profile for a spot light.

XGL_LIGHT_TYPE(3)

Defines the type of the Light object. A light can be an ambient, a directional, a positional, or a spot light.

XGL_OBJ_APPLICATION_DATA(3)

This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)

This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)

This read-only attribute defines the type of an XGL object.

NAME

XGL_LIGHT_ATTENUATION_1,
XGL_LIGHT_ATTENUATION_2 – define the light attenuation coefficients of a Light

SYNOPSIS

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

Xgl_light       light;

xgl_object_set(light, XGL_LIGHT_ATTENUATION_1, float, 0);
xgl_object_set(light, XGL_LIGHT_ATTENUATION_2, float, 0);

xgl_object_get(light, XGL_LIGHT_ATTENUATION_1, float *);
xgl_object_get(light, XGL_LIGHT_ATTENUATION_2, float *);
```

DESCRIPTION

These attributes define the Light attenuation coefficients. These attributes apply to Lights whose **XGL_LIGHT_TYPE**(3) is either XGL_LIGHT_POSITIONAL or XGL_LIGHT_SPOT.

The *light attenuation factor* for a specific Light can be computed from the light attenuation coefficients as follows:

Let *La* be the light attenuation factor.
Let *C1* be the value of XGL_LIGHT_ATTENUATION_1.
Let *C2* be the value of XGL_LIGHT_ATTENUATION_2.
Let *Op* be the object position.
Let *Lp* be the light position.

Then $La = 1 / (C1 + C2(\text{norm}(Op - Lp)))$.

The default value for XGL_LIGHT_ATTENUATION_1 is 1.0.

The default value for XGL_LIGHT_ATTENUATION_2 is 0.0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LIGHT_TYPE(3)

NAME

XGL_LIGHT_COLOR – defines the color of a Light

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_light       light;
xgl_object_set(light, XGL_LIGHT_COLOR, Xgl_color *, 0);
xgl_object_get(light, XGL_LIGHT_COLOR, Xgl_color *);
```

DESCRIPTION

This attribute defines the color of a Light. Lights of all types (see **XGL_LIGHT_TYPE**(3)) have a color associated with them.

When the Device color type (see **XGL_DEV_COLOR_TYPE**(3)) is indexed (XGL_COLOR_INDEX), the Light color type is gray (XGL_COLOR_GRAY), with a value ranging between 0.0 and 1.0. The lighting calculation determines the index for the reflected light color; the material color is given by an index. XGL determines the color ramp (see **XGL_CMAP_RAMP_LIST**(3)) where the material color resides within the color table of the Raster's Color Map and converts this to a floating-point value. The lighting calculation proceeds in floating point. The final reflected color index comes from scaling the floating-point reflected color by the size of the material color's ramp and adding the offset of the ramp's first index.

When the Device color type is RGB (XGL_COLOR_RGB), the lighting calculation applies to each of the red, green, and blue channels. The range on each of the red, green, and blue channels is 0.0 to 1.0.

The default value for the index color model is a gray value of 1.0; the default value for the rgb color model is white.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR(3)
XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR(3)
XGL_CMAP_COLOR_TABLE(3)
XGL_CMAP_RAMP_NUM(3)
XGL_CMAP_RAMP_LIST(3)
XGL_CTX_SURF_FRONT_COLOR(3)
XGL_LIGHT_TYPE(3)
XGL_DEV_COLOR_TYPE(3)

NAME

XGL_LIGHT_DIRECTION – defines the direction of a Light

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_light          light;
xgl_object_set(light, XGL_LIGHT_DIRECTION, Xgl_pt_f3d *, 0);
xgl_object_get(light, XGL_LIGHT_DIRECTION, Xgl_pt_f3d *);
```

DESCRIPTION

This attribute defines the direction of Light propagation in world coordinates (WC). The attribute applies to Lights whose **XGL_LIGHT_TYPE**(3) is either XGL_LIGHT_DIRECTIONAL or XGL_LIGHT_SPOT. XGL normalizes the length of the lighting vector given by the application.

The light rays for Directional Lights are parallel, propagating in the direction given by this attribute.

A Spot Light radiates light rays from a single point. The ray with the maximum intensity propagates in the direction given by this attribute. Rays lying off this principal axis fall off in intensity with the angle away from the axis when the Light spot exponent is greater than 1.0.

The default value is the vector (0.0, 0.0, 1.0).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LIGHT_SPOT_ANGLE(3)
XGL_LIGHT_SPOT_EXPONENT(3)
XGL_LIGHT_TYPE(3)

NAME

XGL_LIGHT_POSITION – defines the position of a Light

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_light       light;
xgl_object_set(light, XGL_LIGHT_POSITION, Xgl_pt_d3d *, 0);
xgl_object_get(light, XGL_LIGHT_POSITION, Xgl_pt_d3d *);
```

DESCRIPTION

This attribute defines the position of a Light in world coordinates (WC). The attribute applies to Lights whose **XGL_LIGHT_TYPE**(3) is either XGL_LIGHT_POSITIONAL or XGL_LIGHT_-SPOT.

The default value is the point (0.0, 0.0, 0.0).

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LIGHT_TYPE(3)

NAME

XGL_LIGHT_SPOT_ANGLE,
XGL_LIGHT_SPOT_EXPONENT – define the intensity profile of a Spot Light

SYNOPSIS

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

Xgl_light          light;

xgl_object_set(light, XGL_LIGHT_SPOT_ANGLE, float, 0);
xgl_object_set(light, XGL_LIGHT_SPOT_EXPONENT, float, 0);

xgl_object_get(light, XGL_LIGHT_SPOT_ANGLE, float *);
xgl_object_get(light, XGL_LIGHT_SPOT_EXPONENT, float *);
```

DESCRIPTION

These attributes define the intensity profile of a Spot Light (for a description of Spot Lights, see **XGL_LIGHT_TYPE**(3)). The intensity is at its maximum along the light ray pointing in the direction given by **XGL_LIGHT_DIRECTION**(3)**,** radiating from the Spot Light position given by **XGL_LIGHT_POSITION**(3)**.**

XGL_LIGHT_SPOT_ANGLE gives the illumination angle of influence for the Spot Light. The angle is specified in radians. For example, a value of *pi* for this attribute would result in a Spot Light that illuminates a hemisphere.

XGL_LIGHT_SPOT_EXPONENT gives the light source concentration exponent. A larger value causes the intensity to fall off faster as the angle increases between the Spot Light direction given by XGL_LIGHT_DIRECTION and the direction that points from the light position to the object position. This angle is raised to the power of this attribute's value, which should be greater than or equal to 1.0.

The default value for XGL_LIGHT_SPOT_ANGLE is π.

The default value for XGL_LIGHT_SPOT_EXPONENT is 1.0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LIGHT_DIRECTION(3)
XGL_LIGHT_POSITION(3)
XGL_LIGHT_TYPE(3)

NAME

XGL_LIGHT_TYPE – defines the type of a Light object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_light light;
xgl_object_set(light, XGL_LIGHT_TYPE, Xgl_light_type, 0);
xgl_object_get(light, XGL_LIGHT_TYPE, Xgl_light_type *);
```

DESCRIPTION

This attribute specifies the Light type, which determines the properties of a Light. Lights of all types have a color associated with them (see **XGL_LIGHT_COLOR**(3)). XGL accepts the following values for *Xgl_light_type*:

XGL_LIGHT_AMBIENT

An *Ambient* Light adds a uniform brightness to the light reflected from surfaces. It simulates light reflected from multiple surfaces. Ambient Light does not come from any single direction or position; instead, Ambient Light models light coming from all directions uniformly. Ambient reflections do not depend on the orientation or position of a surface. An Ambient Light does not contribute to diffuse and specular reflections. An Ambient Light has only a color, as specified with XGL_LIGHT_COLOR.

XGL_LIGHT_DIRECTIONAL

A *Directional* Light has parallel light rays. It contributes to diffuse and specular reflections, which in turn depend on the orientation of a surface, but not its position. A Directional Light does not contribute to ambient reflections. A Directional Light has a color specified with XGL_LIGHT_COLOR, and a direction specified with **XGL_LIGHT_DIRECTION**(3)**.**

XGL_LIGHT_POSITIONAL

A *Positional* Light radiates light rays uniformly from a single point in all directions. It contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface. A Positional Light does not contribute to ambient reflections. The intensity profile of a Positional Light depends on the distance of the Light from the surface. A Positional Light has a color specified with **XGL_LIGHT_COLOR**(3)**,** a position specified with **XGL_LIGHT_POSITION**(3)**,** and an attenuation profile specified with **XGL_LIGHT_ATTENUATION_1**(3) and **XGL_-LIGHT_ATTENUATION_2**(3)**.**

XGL_LIGHT_SPOT

A *Spot* Light radiates light rays from a single point. A Spot Light has both a position and a direction. The ray with the maximum intensity propagates in this direction. Rays lying off this principal axis fall off in intensity with the angle away from the axis. The light from a Spot Light affects surfaces lying within the Light's cone of illumination. The intensity profile of a Spot Light depends on the distance of the Light from the surface. A Spot Light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface. A Spot Light has a color specified with XGL_LIGHT_COLOR, a direction specified with XGL_LIGHT_DIRECTION, a position specified with XGL_LIGHT_-

POSITION, a spot profile specified with **XGL_LIGHT_SPOT_ANGLE**(3) and **XGL_LIGHT_SPOT_EXPONENT**(3), and an attenuation profile specified with XGL_LIGHT_ATTENUATION_1 and XGL_LIGHT_ATTENUATION_2.

The default value is XGL_LIGHT_AMBIENT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_3D_CTX_SURF_FRONT_LIGHT_TYPE(3)
XGL_3D_CTX_SURF_BACK_LIGHT_TYPE(3)
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT(3)
XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT(3)
XGL_LIGHT_ATTENUATION_1(3)
XGL_LIGHT_ATTENUATION_2(3)
XGL_LIGHT_COLOR(3)
XGL_LIGHT_DIRECTION(3)
XGL_LIGHT_POSITION(3)
XGL_LIGHT_SPOT_ANGLE(3)
XGL_LIGHT_SPOT_EXPONENT(3)

NAME

XGL_LPAT – a Line Pattern object

OVERVIEW

XGL_LPAT is the *type* parameter value for a Line Pattern.

A Line Pattern object can be used to draw patterned lines and edges. To use the Line Pattern, it must first be attached to a Context by setting the attribute **XGL_CTX_LINE_PATTERN**(3) to the handle of the pattern object returned by the operator **xgl_object_create**(3)**.** If the Line Pattern is to be used for edges, the attribute **XGL_CTX_EDGE_PATTERN**(3) should be set.

To draw patterned lines, set the Context attribute **XGL_CTX_LINE_STYLE**(3) to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED.

To draw patterned edges, set the Context attribute **XGL_CTX_EDGE_STYLE**(3) to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED.

The Context attribute **XGL_CTX_LINE_COLOR**(3) determines what the color of the line will be. If XGL_CTX_LINE_STYLE is set to XGL_LINE_ALT_PATTERNED, the line is patterned alternately with dashes of **XGL_CTX_LINE_ALT_COLOR**(3) and XGL_CTX_LINE_COLOR (edge colors are handled in an analogous fashion using the attributes **XGL_CTX_EDGE_COLOR**(3) and **XGL_CTX_EDGE_ALT_COLOR**(3)).

The application can either create its own Line Pattern object to be used for lines, or it can use one of the predefined Line Patterns (see **XGL_CTX_LINE_PATTERN**(3)).

After creating the Line Pattern, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

The following is a brief description of the attributes associated with Line Pattern objects. For more information, see the man page for each attribute.

XGL_LPAT_BALANCED_DASH(3)
: The attribute to set to define the Line Pattern balance.

XGL_LPAT_DATA(3)
: The array that defines the on / off segment lengths of the Line Pattern.

XGL_LPAT_DATA_SIZE(3)
: The length of the XGL_LPAT_DATA data array.

XGL_LPAT_DATA_TYPE(3)
: The data type of the elements within the XGL_LPAT_DATA data array.

XGL_LPAT_OFFSET(3)
: The location within the XGL_LPAT_DATA data array where XGL will begin using the Line Pattern data.

XGL_LPAT_STYLE(3)
: The attribute to set to define the Line Pattern style.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the

object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

NAME

XGL_LPAT_DATA – specifies the line segments, stored in an array, that comprise a Line Pattern

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_lpat                lpat;
xgl_object_set(lpat, XGL_LPAT_DATA, Xgl_sgn32 | float [], 0);
xgl_object_get(lpat, XGL_LPAT_DATA, Xgl_sgn32 | float*);
```

DESCRIPTION

This attribute defines the on and off segment lengths of a Line Pattern, which are stored as an array. The attribute is used with the operator **xgl_object_create**(3) to create a Line Pattern object. Once created, the application can attach it to the current Context by using one or both Context attributes, **XGL_CTX_LINE_PATTERN**(3) and **XGL_CTX_EDGE_PATTERN**(3). Then, the pattern will be used if **XGL_CTX_LINE_STYLE**(3) or **XGL_CTX_EDGE_STYLE**(3) is set to patterned lines.

The on and off line segment data is an array of positive, non-zero, 32-bit integers or floating point numbers. The data type used is specified by the Line Pattern attribute **XGL_LPAT_DATA_TYPE**(3). It should be set before passing the data.

The length of the line segment data array is set with the Line Pattern attribute **XGL_LPAT_DATA_SIZE**(3). It should also be set before passing the data.

The line segment data array stores line segment lengths of alternating *on* and *off* pattern segments. The first segment is assumed to be an *on* segment. The lengths are specified in the coordinate system defined by the Line Pattern attribute **XGL_LPAT_COORD_SYS**(3).

The segments are used cyclically; when the end of the pattern is reached (that is, when the end of the array is reached), the pattern is started over from the beginning. For patterns containing an even number of segments, the cyclical replication occurs in a straightforward manner. A pattern such as 1 (*on*), 3 (*off*), 10 (*on*), 5 (*off*) repeats as 1 (*on*), 3 (*off*), 10 (*on*), 5 (*off*), 1 (*on*), 3 (*off*), 10 (*on*), 5 (*off*), and so on. For an odd number of segments, however, the pattern is first concatenated with itself to produce a list of even length. This list is then used as the data array. For example, a pattern such as 1 (*on*), 3 (*off*), 10 (*on*) becomes 1 (*on*), 3 (*off*), 10 (*on*), 1 (*off*), 3 (*on*), 10 (*off*), which repeats as 1 (*on*), 3 (*off*), 10 (*on*), 1 (*off*), 3 (*on*), 10 (*off*), 1 (*on*), 3 (*off*), 10 (*on*), 1 (*off*), 3 (*on*), 10 (*off*), and so on.

The attribute **XGL_LPAT_OFFSET**(3) defines an offset into the Line Pattern where the pattern will begin. Its data type is always *float*.

When using xgl_object_get to get the Line Pattern data, the application must pass a pointer to an array large enough to hold the XGL_LPAT_DATA_SIZE number of data items.

No default value is defined.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_CTX_EDGE_PATTERN(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_EDGE_STYLE(3)
XGL_CTX_LINE_STYLE(3)
XGL_LPAT_COORD_SYS(3)
XGL_LPAT_DATA_SIZE(3)
XGL_LPAT_DATA_TYPE(3)
XGL_LPAT_OFFSET(3)
XGL_LPAT_STYLE(3)

NAME

XGL_LPAT_DATA_SIZE – defines the size of the Line Pattern data array

SYNOPSIS

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

Xgl_lpat                lpat;

xgl_object_set(lpat, XGL_LPAT_DATA_SIZE, Xgl_usgn32, 0);
xgl_object_get(lpat, XGL_LPAT_DATA_SIZE, Xgl_usgn32 *);
```

DESCRIPTION

This attribute defines the size (number of elements) of the data array that stores the lengths of on and off segments describing a Line Pattern. The data array is specified (that is, attached to the Context) by setting the attribute **XGL_LPAT_DATA**(3)**.**

XGL_LPAT_DATA_SIZE should be set before setting the data with XGL_LPAT_DATA. Line Pattern objects are created by using the operator **xgl_object_create**(3)**.**

The default value is 0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LPAT_DATA(3)

NAME

XGL_LPAT_DATA_TYPE – defines the data type of line segment lengths stored in the Line Pattern data array

SYNOPSIS

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

Xgl_lpat                lpat;

xgl_object_set(lpat, XGL_LPAT_DATA_TYPE, Xgl_data_type, 0);
xgl_object_get(lpat, XGL_LPAT_DATA_TYPE, Xgl_data_type *);
```

DESCRIPTION

This attribute specifies the data type of the line segment lengths stored in the Line Pattern data array. The Line Pattern data array is specified by the attribute **XGL_LPAT_DATA**(3), and Line Patterns are created with the operator **xgl_object_create**(3). The line segment data can be either an array of 32-bit-integer or floating-point values. Thus, this attribute can be specified as either XGL_DATA_INT or XGL_DATA_FLT.

The Line Pattern array length, which is specified by the attribute **XGL_LPAT_DATA_SIZE**(3), and the Line Pattern data type should be set before setting the Line Pattern data with the attribute XGL_LPAT_DATA.

The default value is XGL_DATA_FLT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LPAT_DATA(3)
XGL_LPAT_DATA_SIZE(3)

NAME

XGL_LPAT_OFFSET – defines an offset into the Line Pattern data array where the Line Pattern will begin

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_lpat                lpat;
xgl_object_set(lpat, XGL_LPAT_OFFSET, float, 0);
xgl_object_get(lpat, XGL_LPAT_OFFSET, float *);
```

DESCRIPTION

This attribute is used to specify the location within a Line Pattern object where XGL will begin using the pattern to render a patterned line. It is defined as an *offset* into a Line Pattern specified by a Line Pattern data array (see **XGL_LPAT_DATA**(3)).

The offset into the Line Pattern is used only at the beginning of the patterned line that is being rendered. It is interpreted as a distance into the Line Pattern (measured in Line Pattern coordinate system space as specified by the attribute **XGL_LPAT_COORD_SYS**(3)) at which the pattern should be started. Before beginning to draw a line, XGL searches through the elements of the XGL_LPAT_DATA array, adding up distances in the array without rendering the line. When XGL has traveled the **XGL_LPAT_OFFSET**(3) distance into the pattern, it starts rendering the patterned line as it normally would — that is, drawing the line when the corresponding segment defined in the array is *on* and not drawing it when the segment is *off*. (For more information, see XGL_LPAT_DATA.) If, while drawing the patterned line, the end of the pattern (the Line Pattern data array) is reached, the pattern is started over from the beginning of the array.

XGL_LPAT_OFFSET must be greater than or equal to 0.0.

The default value is 0.0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

XGL_LPAT_DATA(3)
XGL_LPAT_DATA_SIZE(3)
XGL_LPAT_STYLE(3)

NOTES

This attribute applies only when the value of XGL_LPAT_STYLE is XGL_LPAT_FIXED_OFFSET.

NAME

XGL_LPAT_STYLE,
XGL_LPAT_BALANCED_DASH – define Line Pattern style and Line Pattern balancing

SYNOPSIS

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

Xgl_lpat                lpat;

xgl_object_set(lpat, XGL_LPAT_STYLE, Xgl_lpat_style, 0);
xgl_object_set(lpat, XGL_LPAT_BALANCED_DASH, Xgl_lpat_bal_dash, 0);

xgl_object_get(lpat, XGL_LPAT_STYLE, Xgl_lpat_style *);
xgl_object_get(lpat, XGL_LPAT_BALANCED_DASH, Xgl_lpat_bal_dash *);
```

DESCRIPTION

XGL_LPAT_STYLE(3) specifies the Line Pattern style. The following enumerated values of *Xgl_lpat_style* define the options:

XGL_LPAT_FIXED_OFFSET
: XGL applies the fixed Line Pattern offset of *lpat* given by **XGL_LPAT_OFFSET**(3) when rendering polylines, open arcs, NURBS curves, or polygon edges. For each line segment, the Line Pattern carries over from the segment to the next adjacent segment.

XGL_LPAT_BALANCED_SEGMENT
: For each segment of a polyline, XGL balances the Line Pattern with respect to the midpoint of the segment before clipping. The balancing is independent of a segment's adjacent segments. This line pattern style is applicable only to polylines and not other primitives. If an application tries to use this line pattern style with open arcs, NURBS curves, or polygon edges, XGL will render them with the value XGL_LPAT_FIXED_OFFSET.

XGL_LPAT_BALANCED_DASH takes effect only when XGL_LPAT_STYLE has the value XGL_LPAT_BALANCED_SEGMENT. It allows an application to select the first or second dash in the Line Pattern around which to balance the Line Pattern. In the following descriptions, let the Line Pattern data consist of the dashes with lengths $D[0]$, $D[1]$, ..., $D[n-1]$.

The enumerated values of *Xgl_lpat_bal_dash* define the following options:

XGL_LPAT_BAL_DASH_0
: The center of the first dash with length $D[0]$ coincides with the midpoint of the segment. The Line Pattern will be symmetric with respect to this midpoint if n is even and one of the following two conditions holds: (1) $n = 2$, or (2) $n > 2$ and $D[1] = D[n-1]$, $D[2] = D[n-2]$, ..., and $D[(n/2)-1] = D[(n/2)+1]$.

XGL_LPAT_BAL_DASH_1
: The center of the second dash with length $D[1]$ coincides with the midpoint of the segment. The Line Pattern will be symmetric with respect to this midpoint if n is even and one of the following two conditions holds: (1) $n = 2$, or (2) $n > 2$ and $D[0] = D[2]$, $D[n-1] = D[3]$, ..., and $D[(n/2)+2] = D[n/2]$.

Note that if n is odd, XGL replicates the Line Pattern data once, producing an array of dash lengths twice the size of the specified array (see **XGL_LPAT_DATA**(3)). XGL ignores the value of **XGL_LPAT_OFFSET**(3) when XGL_LPAT_STYLE has the value XGL_LPAT_-

BALANCED_SEGMENT.

The default value of XGL_LPAT_STYLE is XGL_LPAT_FIXED_OFFSET.

The default value of XGL_LPAT_BALANCED_DASH is XGL_LPAT_BAL_DASH_0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_LPAT_OFFSET(3)
XGL_LPAT_DATA(3)

NAME

XGL_MARKER – a Marker object

OVERVIEW

XGL_MARKER is the type of an object representing an XGL Marker object. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

A Marker is a symbol that is rendered at each of the points in the **xgl_multimarker**(3) primitive. The application can define its own Marker symbols by creating a Marker object and defining its geometry.

Marker uses the following attribute:

XGL_MARKER_DESCRIPTION(3)
: Specifies the geometry of the Marker symbol.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

NAME

XGL_MARKER_DESCRIPTION – geometric points defining a Marker object

SYNOPSIS

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

Xgl_marker          marker;

xgl_object_set(marker, XGL_MARKER_DESCRIPTION, Xgl_pt_list_list *, 0);
xgl_object_get(marker, XGL_MARKER_DESCRIPTION, Xgl_pt_list_list *);
```

DESCRIPTION

This attribute is the description for a Marker object. All Marker objects consist of a list of 2D float coordinates that define one or more polylines that comprise the marker. The application can get and set the coordinates that define a Marker for application-created Markers. But the Xgl default Marker objects are read-only, and the application cannot set the geometry associated with them.

When setting a Marker description, the application should fill in an *Xgl_pt_list_list* data structure and set the Marker object description to this data structure.

When getting a Marker description, the application need only declare the Xgl_pt_list_list variable and pass XGL a pointer to it. XGL assumes that the application will not know the size of the point lists and thus will not be able to allocate memory to hold the point list information; therefore, XGL allocates the memory for the *Xgl_pt_list_list* data structure on behalf of the application. Once the memory is allocated, however, it is the responsibility of the application to free the memory with a free system call. The following lines of code show how to free the memory for an *Xgl_pt_list_list* data structure:

```
for ( i=0; i < point_list_list.num_pt_lists; i++)
        free (point_list_list.pt_lists[i].pts.f2d);
free (point_list_list.pt_lists);
```

Marker object descriptions are limited to float 2D coordinates only. Thus, the application must use either XGL_PT_F2D, XGL_PT_COLOR_F2D or XGL_PT_FLAG_F2D when specifying a Marker description. If any other point types are used, the results may by incorrect. Marker object descriptions are limited to 128 2D float coordinates.

For the application to define its own Marker objects, it: 1) creates a Marker object using **xgl_object_create**(3)**,** 2) attaches an application-defined Marker description to that object using XGL_MARKER_DESCRIPTION, and 3) uses the Context attribute **XGL_CTX_MARKER**(3) to attach the application-defined marker to the current Context. Then a call to **xgl_-multimarker**(3) renders the application-defined markers into an XGL Raster.

If the XGL_CTX_MARKER attribute is set to an application-defined Marker that doesn't have a description, a call to **xgl_multimarker**(3) results in nothing being rendered.

If the application data associated with this Marker object is a pointer to a floating point scale factor, when rendering through the CGM stream pipeline, the marker will be stroked out and scaled by a factor equal to the product of **XGL_CTX_MARKER_SCALE_-FACTOR**(3) and the float pointed to by **XGL_OBJ_APPLICATION_DATA**(3). If there is no object application data associated with the marker, then a marker other than one of the predefined markers will be rendered by the CGM stream pipeline as a dot marker. Note that the **XGL_OBJ_APPLICATION_DATA**(3) scale factor only applies to the CGM stream

pipeline. See the *XGL Programmer's Guide* for an example.

The default value depends on the Marker object. The default value for application-defined Markers is NULL — no geometry.

SEE ALSO

xgl_object_create(3)
xgl_object_get(3)
xgl_object_set(3)
xgl_multimarker(3)
XGL_CTX_MARKER(3)

NAME

XGL_MEM_RAS – a Memory Raster Device object

OVERVIEW

XGL_MEM_RAS is the type of an object representing a Memory Raster Device. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

XGL graphics are rendered *on* a Device object. A specific Device object is the Memory Raster Device that draws into *pixels*, which are stored in the application's memory space. Because these pixels are not on a frame buffer, they are not visible to the operator. In contrast, a Window Raster Device object (see **xgl_object_create**(3) and XGL_WIN_RAS), which represents a hardware display device with a frame buffer, can display visible images.

After creating the Memory Raster Device, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

A Memory Raster Device object shares many attributes with the Window Raster Device object. These attributes have the common prefix XGL_RAS_. Attributes specific only to the Memory Raster Device have the prefix XGL_MEM_RAS_.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_DEV_COLOR_MAP(3)
: Specifies the Color Map Object used by the Memory Raster Device object.

XGL_DEV_COLOR_TYPE(3)
: Specifies the color type (that is, the color model) of the Raster. It can be set only once, when the Memory Raster Device object is created. The two alternatives are: XGL_COLOR_INDEX and XGL_COLOR_RGB.

XGL_DEV_CONTEXTS(3)
XGL_DEV_CONTEXTS_NUM(3)
: These attributes return the Context object handles used by the Memory Raster Device object.

XGL_DEV_MAXIMUM_COORDINATES(3)
: Returns the maximum coordinate values for the Memory Raster Device object.

XGL_DEV_PICK_BUFFER_SIZE(3)
: Returns the maximum pick buffer size that a Memory Raster Device object can support efficiently.

XGL_DEV_REAL_COLOR_TYPE(3)
: Specifies the real color space used by the Memory Raster Device object.

XGL_MEM_RAS_IMAGE_BUFFER_ADDR(3)
: Returns the starting address of the block of memory set used by the Memory Raster Device.

XGL_MEM_RAS_Z_BUFFER_ADDR(3)
: Returns the starting address of the block of memory set used by the Z buffer of a memory raster.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

XGL_RAS_DEPTH(3)
XGL_RAS_HEIGHT(3)
XGL_RAS_WIDTH(3)

These attributes give the size of the Memory Raster Device. They can be set at any time, however. If any of them are changed after the Raster has been created, all pixel data currently stored in the Raster is lost. The depth of a Memory Raster Device can be 1, 4, 8, or 32, which is the number of *bit-planes* in the Raster (that is, the number of bits that are used to specify the color of one pixel).

XGL_RAS_RECT_LIST(3)
XGL_RAS_RECT_NUM(3)

These attributes define the number and list of clip rectangles in the application-specified clip list.

XGL_RAS_SOURCE_BUFFER(3)
This attribute specifies the buffer to be used as source during raster operations.

XGL does not render into 1-bit-deep Memory Rasters. This means that you cannot use a 1-bit Memory Raster as the device for a Context. The 1-bit Memory Rasters are used for stipple-filled polygons and for the xgl_context_copy_raster function, where the source Raster is a 1-bit Memory Raster, and the device for the destination context is either a 4-bit, 8-bit or 32-bit Raster.

NOTES

32-bit Memory Rasters with an XGL_DEV_COLOR_TYPE of XGL_COLOR_INDEX are not supported. When creating a 32-bit Memory Raster, the application should set XGL_DEV_COLOR_TYPE to XGL_COLOR_RGB.

4-bit and 8-bit Indexed (that is, XGL_DEV_COLOR_TYPE is XGL_COLOR_INDEX) Memory Rasters store their pixel values as application-defined color values (that is, the indices are exactly those specified, either explicitly or implicitly, by the application).

NAME

XGL_MEM_RAS_IMAGE_BUFFER_ADDR – the address of the array of pixels used in a Memory Raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mem_ras          ras;
xgl_object_set(ras, XGL_MEM_RAS_IMAGE_BUFFER_ADDR, Xgl_usgn32 *, 0);
xgl_object_get(ras, XGL_MEM_RAS_IMAGE_BUFFER_ADDR, Xgl_usgn32 **);
```

DESCRIPTION

This attribute allows the application to set or get the starting address of the block of memory set aside for an XGL Memory Raster. It also allows the application to read and write pixel data directly. Currently 1-, 4-, 8-, and 32-bit deep Memory Rasters are supported (see **XGL_RAS_DEPTH**(3)).

XGL always allocates memory when creating a Memory Raster (a Memory Raster is created using **xgl_object_create**(3)). Applications should try to avoid allocating their own memory for the Memory Raster and setting the memory address pointer in order to limit the number of memory allocation calls. It is important to note that the application programmer must know the depth (**XGL_RAS_DEPTH**(3)), width (**XGL_RAS_WIDTH**(3)), and height (**XGL_RAS_HEIGHT**(3)) of the Raster before accessing its pixel values.

XGL Memory Rasters are built as a collection of pixel rows. Each row begins at a short (16-bit) boundary; padding can be added to the least significant bits of the word at the end of a row. An application allocating its own memory to pass later to a Memory Raster must follow this rule to obtain correct rendering.

The allocation scheme for memory rasters is as follows:

```
Xgl_sgn32       linebytes;
Xgl_usgn32      *mem32;
Xgl_usgn16      *mem16;
Xgl_usgn8       *mem8;

linebytes = ((((width * depth) + 15) >> 3) & ~1);
mem32 = (Xgl_usgn32 *)malloc(height * linebytes);

if (depth == 1)
        mem16 = (Xgl_usgn16 *)mem32;
else if (depth == 4 || depth == 8)
        mem8 = (Xgl_usgn8 *)mem32;
```

Once the memory is allocated, you can point to it with a character or short pointer depending on how your application accesses the actual data in the memory raster. If the depth is 1, XGL accesses the memory as an unsigned short (Xgl_usgn16). If the depth is 4 or 8, XGL accesses the memory as an unsigned char (Xgl_usgn8). If the depth is 32, XGL accesses the memory as an unsigned long (Xgl_usgn32).

The color values are stored in xRGB format — that is, the uppermost byte is unused, followed by one byte each of blue, green, and red intensity values.

The size of the raster, in bytes, is:
4 * raster_height * raster_width

The x,y pixel is stored in the word offset from the memory address:
(Xgl_usgn32*)(mem_addr + y * raster_width + x)

The default value is the address of the memory allocated at the creation of the Memory Raster.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
xgl_object_create(3)
XGL_RAS_DEPTH(3)
XGL_RAS_WIDTH(3)
XGL_RAS_HEIGHT(3)

NOTE

When creating a Memory Raster, the **XGL_MEM_RAS_IMAGE_BUFFER_ADDR**(3) attribute must be the last attribute in the **xgl_object_create**(3) call.

In order to write machine independent code, it is necessary for the application to write the raster the same way as XGL reads (accesses) it. For example, 32-bit deep rasters should be written as words and not as 4 individual characters.

NAME

XGL_MEM_RAS_Z_BUFFER_ADDR – the address of the array of values used in the Z buffer of a memory raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mem_ras                ras;
xgl_object_set(ras, XGL_MEM_RAS_Z_BUFFER_ADDR, Xgl_usgn32 *, 0);
xgl_object_get(ras, XGL_MEM_RAS_Z_BUFFER_ADDR, Xgl_usgn32**);
```

DESCRIPTION

This attribute allows the application to set or get the starting address of the block of memory set aside for the Z buffer of a memory raster. It allows the application to read and write Z buffer values directly.

The Z buffer is allocated when the memory raster is attached to a 3D context and the attribute **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER. Applications should avoid allocating their own memory for the Z buffer and setting the memory address pointer in order to limit the number of memory allocation calls.

The Z buffer of a memory raster has 24 bit planes and hence has a depth of $(2^{24} - 1)$. The height of the Z buffer is based on the raster height **XGL_RAS_HEIGHT**(3) and the width of the Z buffer is based on the raster width **XGL_RAS_WIDTH**(3)**.** The Z buffer is a collection of Z values. Each Z value occupies 32 bits and XGL accesses the least significant three bytes of the 32 bit word during Z buffer operations. Thus, each row begins at a 32 bit word boundary.

The allocation scheme for the memory rasters is as follows:

```
Xgl_usgn32      line_bytes;
Xgl_usgn32      *mem32;
where line_bytes = width × 4
mem32 = (Xgl_usgn32 *) malloc(height × line_bytes)
```

The default value is NULL

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_RAS_HEIGHT(3)
XGL_RAS_WIDTH(3)
XGL_3D_CTX_HLHSR_MODE(3)

NAME

XGL_MIPMAP_TEXTURE – a MipMap Texture object

OVERVIEW

XGL_MIPMAP_TEXTURE is the *type* of an object representing an XGL MipMap Texture object. A texturing image to be applied to 3D surface primitives is encapsulated in this object. In addition, this object also provides filtering capabilities to generate a MipMap.

After creating the MipMap Texture object, using **xgl_object_create**(3)**,** an application can inquire about the value of its attributes by using the **xgl_object_get**(3) operator.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_MIPMAP_TEXTURE_DEPTH(3)
XGL_MIPMAP_TEXTURE_HEIGHT(3)
XGL_MIPMAP_TEXTURE_WIDTH(3)

These attributes give the size of the base texture image. They are read-only attributes. The values are set indirectly when an application sets the attribute **XGL_MIPMAP_TEXTURE_MEM_RAS_LIST**(3)**,** or uses the operator **xgl_mipmap_texture_build**(3) to build a MipMap from a memory raster.

XGL_MIPMAP_TEXTURE_LEVELS(3)
Defines the number of levels of mipmap in the MipMap Texture object.

XGL_MIPMAP_TEXTURE_MEM_RAS_LIST(3)
Specifies a list of memory rasters that form a MipMap.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

NAME

XGL_MIPMAP_TEXTURE_DEPTH – specifies the number of bits required to store one texture element

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mipmap_texture          mipmap_texture;
xgl_object_get(mipmap_texture, XGL_MIPMAP_TEXTURE_DEPTH, Xgl_usgn32 *);
```

DESCRIPTION

This read-only attribute defines the number of bits that are used to specify one texture element. The number of channels per texture element is obtained by dividing the depth by 8.

This attribute is set indirectly when an application sets the attribute **XGL_MIPMAP_-TEXTURE_MEM_RAS_LIST**(3) or uses the operator **xgl_mipmap_texture_build**(3)**.**

Supported depths are 8 and 32.

The default value is 0.

SEE ALSO

xgl_object_get(3)
xgl_mipmap_texture_build(3)
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST(3)

NAME

XGL_MIPMAP_TEXTURE_HEIGHT – specifies the height of a texture map

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mipmap_texture          mipmap_texture;
xgl_object_get(mipmap_texture, XGL_MIPMAP_TEXTURE_HEIGHT, Xgl_usgn32 *);
```

DESCRIPTION

This read-only attribute specifies the height of an XGL *texture* image. In case of a mipmap, the height corresponds to the base texture image. The height of the next level of mipmap is reduced by a factor of 2 from the previous level.

This attribute is set indirectly when an application sets the attribute **XGL_MIPMAP_TEXTURE_MEM_RAS_LIST**(3) or uses the operator **xgl_mipmap_texture_build**(3).

The default value is 0.

SEE ALSO

xgl_object_get(3)
xgl_mipmap_texture_build(3)
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST(3)

NAME

XGL_MIPMAP_TEXTURE_LEVELS – specifies the number of mipmap levels

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mipmap_texture          mipmap_texture;
xgl_object_set(mipmap_texture, XGL_MIPMAP_TEXTURE_LEVELS, Xgl_usgn32, 0);
xgl_object_get(mipmap_texture, XGL_MIPMAP_TEXTURE_LEVELS, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the number of mipmap levels when an application sets the attribute **XGL_MIPMAP_TEXTURE_MEM_RAS_LIST**(3)**.** The number of memory rasters in the list should be equal to the number of levels. This attribute is also used by the operator **xgl_mipmap_texture_build**(3) to generate a MipMap.

Note that once a mipmap is created, changing the number of levels destroys the current mipmap. Thus, after setting this attribute, the application should also set the attribute **XGL_MIPMAP_TEXTURE_MEM_RAS_LIST**(3) or the operator **xgl_mipmap_texture_build**(3)**.**

The default value is 0.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_mipmap_texture_build(3)
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST(3)

NAME

XGL_MIPMAP_TEXTURE_MEM_RAS_LIST – specifies a list of memory rasters that form a mipmap

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mipmap_texture          mipmap_texture;
xgl_object_set(mipmap_texture, XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, Xgl_mem_ras [], 0);
xgl_object_get(mipmap_texture, XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, Xgl_mem_ras []);
```

DESCRIPTION

This attribute specifies an array of memory rasters that form a mipmap pyramid. Each level of the pyramid has dimensions that are half as large as the previous level.

When setting this attribute, the number of elements in the array should be equal to the number of levels (specified by **XGL_MIPMAP_TEXTURE_LEVELS**(3)). XGL copies the mipmap to its internal format, thus an application can destroy the memory raster list after it finishes setting the attribute. During an **xgl_object_get**(3) of this attribute, the application is responsible for providing XGL an array of memory rasters (corresponding to **XGL_MIPMAP_TEXTURE_LEVELS**(3)**)** with correct dimensions. (The base dimension of the mipmap is specified by **XGL_MIPMAP_TEXTURE_WIDTH**(3)**, XGL_MIPMAP_TEXTURE_HEIGHT**(3) and **XGL_MIPMAP_TEXTURE_DEPTH**(3)**.**) XGL simply copies the mipmap to the address specified by memory raster objects. For details of memory raster creation and representation format, see **XGL_MEM_RAS**(3) and **XGL_MEM_RAS_IMAGE_BUFFER_ADDR**(3)**.**

The default value is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_MEM_RAS(3)
XGL_MEM_RAS_IMAGE_BUFFER_ADDR(3)
XGL_MIPMAP_TEXTURE_DEPTH(3)
XGL_MIPMAP_TEXTURE_HEIGHT(3)
XGL_MIPMAP_TEXTURE_LEVELS(3)
XGL_MIPMAP_TEXTURE_WIDTH(3)

NAME

XGL_MIPMAP_TEXTURE_WIDTH – specifies the width of a texture map

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_mipmap_texture          mipmap_texture;
xgl_object_get(mipmap_texture, XGL_MIPMAP_TEXTURE_WIDTH, Xgl_usgn32 *);
```

DESCRIPTION

This read-only attribute specifies the width of an XGL *texture* image. In case of a mipmap, the width corresponds to the base texture image. The width of the next level of mipmap is reduced by a factor of 2 from the previous level.

This attribute is set indirectly when an application sets the attribute **XGL_MIPMAP_TEXTURE_MEM_RAS_LIST**(3) or uses the operator **xgl_mipmap_texture_build**(3).

The default value is 0.

SEE ALSO

xgl_object_get(3)
xgl_mipmap_texture_build(3)
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST(3)

NAME

XGL_OBJ_APPLICATION_DATA – a pointer to application-dependent data associated with an XGL object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_object          obj;
xgl_object_set(obj, XGL_OBJ_APPLICATION_DATA, void *, 0);
xgl_object_get(obj, XGL_OBJ_APPLICATION_DATA, void **);
```

DESCRIPTION

This attribute allows the application program to store its own data with an object. For example, a pointer to an application data structure could be stored here.

XGL only stores the *void* * pointer. When the object *obj* is destroyed, XGL does not assume that XGL_OBJ_APPLICATION_DATA is a pointer and free it. If XGL_OBJ_APPLICATION_DATA is a pointer, the application must explicitly free what it points to.

The default value is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NOTES

See **XGL_CTX_MARKER_SCALE_FACTOR**(3) and the *XGL Programmer's Guide* for a description of using markers with CGM stream devices.

NAME

XGL_OBJ_SYS_STATE – returns the System State of an XGL object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_object          object;
xgl_object_get(object, XGL_OBJ_SYS_STATE, Xgl_sys_state *);
```

DESCRIPTION

This read-only attribute returns the System State that was used to create the object. The System State is a parameter to **xgl_object_create**(3)**.**

SEE ALSO

xgl_object_create(3)
xgl_object_get(3)
xgl_open(3)

NAME

XGL_OBJ_TYPE – defines the type of an XGL object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_object          object;
xgl_object_get(object, XGL_OBJ_TYPE, Xgl_obj_type *);
```

DESCRIPTION

This read-only attribute defines the type associated with an XGL object. It can be used to inquire about the object type when only a generic Xgl_object handle to that object is known. XGL currently defines the following values for an object type Xgl_obj_type:

XGL_OBJ
: A Generic object used to store all the attributes associated with an XGL object.

XGL_2D_CTX
: A Context object used to store all the attributes of a 2D Context.

XGL_3D_CTX
: A Context object used to store all the attributes of a 3D Context.

XGL_CGM_DEV
: A CGM Metafile object used to store all the attributes of a metafile output Device.

XGL_CMAP
: A Color Map object used to store all the attributes needed to define color information in XGL.

XGL_DMAP_TEXTURE
: A Data Map Texture object used to store all the attributes needed to define data map texture information in XGL.

XGL_GCACHE
: A Gcache (geometry cache) stores a simpler form of geometry (such as the tessellation of a NURB curve).

XGL_LIGHT
: A Light object used to store all the attributes of the Lights used in the lighting and shading computations.

XGL_LPAT
: A Line Pattern object used to store all the attributes used in defining a new pattern for a line. XGL has the following predefined Line Patterns:
  - xgl_lpat_dashed
  - xgl_lpat_dotted
  - xgl_lpat_dashed_dotted
  - xgl_lpat_dash_dot_dotted
  - xgl_lpat_long_dashed
  - xgl_lpat_cgm_dotted
  - xgl_lpat_cgm_dashed
  - xgl_lpat_dash_dot

XGL_MARKER

A Marker object used to store all the attributes associated with Markers. XGL has the following predefined markers:

xgl_marker_dot
xgl_marker_plus
xgl_marker_asterisk
xgl_marker_circle
xgl_marker_cross
xgl_marker_square
xgl_marker_bowtie_ne
xgl_marker_bowtie_nw

XGL_MEM_RAS

A Memory Raster Device object used to store all the attributes of an off-screen rendering Device. XGL has the following predefined fill patterns:

xgl_fpat_hatch_horiz
xgl_fpat_hatch_vert
xgl_fpat_hatch_diag_45
xgl_fpat_hatch_diag_135
xgl_fpat_hatch_grid_r
xgl_fpat_hatch_grid_d
xgl_fpat_hatch_horiz_dbl
xgl_fpat_hatch_vert_dbl
xgl_fpat_hatch_diag_45_dbl
xgl_fpat_hatch_diag_135_dbl
xgl_fpat_hatch_grid_r_dbl
xgl_fpat_hatch_grid_d_dbl

XGL_MIPMAP_TEXTURE

A MipMap Texture object used to store all the attributes needed to define mip-map texture information in XGL.

XGL_PCACHE

A Primitive Cache (Pcache) object used to store a sequence of zero or more XGL primitives and/or attribute settings.

XGL_SFONT

A Stroke Font object used to store all the attributes associated with a font of stroke characters.

XGL_STREAM

A Stream Device object used to store all the attributes of a stream Device.

XGL_SYS_STATE

The XGL System State object is used to store all information relevant to the general level of an application.

XGL_TMAP

A Texture Map object used to store all the attributes needed to define texture map information in XGL.

XGL_TRANS

A Transform object used to store all the attributes of 2D or 3D Transforms.

XGL_WIN_RAS
: A Window Raster Device object used to store all the attributes of an on-screen rendering Device.

The object type is associated by XGL at the creation of the object. It cannot be modified later. Users cannot define their own objects.

SEE ALSO

xgl_object_create(3)
xgl_object_get(3)
xgl_open(3)

NAME

XGL_PCACHE – a Primitive Cache (Pcache) object

OVERVIEW

XGL_PCACHE is a *type* parameter value for an XGL Primitive Cache object.

A Primitive Cache (Pcache) is an XGL object that stores a sequence of zero or more XGL primitives and / or attribute settings. A Pcache can be rendered with greater speed, in general, than its individual components can. Pcache contents are stored in a device dependent format in order to maximize display speed.

A Pcache must be bound to an XGL context and its device in order to be used. At creation time, a Pcache is not bound to any context. The application must do this by setting the **XGL_PCACHE_CONTEXT**(3) attribute in the Pcache.

Once a pcache is created and bound (to a context and its device), it may be used in place of the context object for the following XGL operators and attributes:

xgl_multimarker(3)
xgl_multipolyline(3)
xgl_stroke_text(3)
xgl_multi_simple_polygon(3)
xgl_triangle_list(3)
xgl_quadrilateral_mesh(3)
xgl_object_set(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_MARKER_COLOR_SELECTOR(3)
XGL_CTX_SURF_FACE_CULL(3)
XGL_CTX_SURF_FRONT_COLOR(3)
XGL_CTX_SURF_FRONT_COLOR_SELECTOR(3)
XGL_3D_CTX_SURF_FRONT_ILLUMINATION(3)
XGL_3D_CTX_SURF_BACK_COLOR(3)
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR(3)
XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)
XGL_3D_CTX_SURF_GEOM_NORMAL(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)

When these operators are applied to a Pcache, they (the operators) are recorded in a device specific format for display at a later time with **xgl_pcache_display**(3)**.** After **xgl_pcache_display**(3) is called, the pcache is *closed*, and the operators listed above may no longer be applied.

An application can empty (and re-open) a Pcache by applying the operator **xgl_context_new_frame**(3)**.** A Pcache is also emptied whenever it is associated with another context or its context is associated with another device. It is an error to use the Pcache (except to destroy or re-associate it) if either its context or device has been

destroyed.

When a Pcache is built, it may make assumptions about the state of the context to which it is attached. At display time, the application may test these assumptions to verify that they are still valid. If so, the Pcache is rendered. If not, the application is informed that the Pcache needs to be rebuilt.

The contents of the Pcache may be picked.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

XGL_PCACHE_CONTEXT(3)
: This attribute defines the XGL Context object associated with the Primitive Cache pcache.

NAME

XGL_PCACHE_CONTEXT – defines the context associated with a Pcache

SYNOPSIS

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

Xgl_pcache      pcache;

xgl_object_set(pcache, XGL_PCACHE_CONTEXT,Xgl_ctx, 0);
xgl_object_get(pcache, XGL_PCACHE_CONTEXT,Xgl_ctx *);
```

DESCRIPTION

This attribute defines the XGL Context object associated with the Primitive Cache *pcache*. The context object specifies the rendering state to be used when rendering *pcache*, and may be either 2D or 3D. After the context is associated with the *pcache*, the *pcache* may be used in place of the context to cache certain primitives and attributes. See **XGL_PCACHE**(3)**.**

When this attribute is set, the *pcache* is emptied, even if the setting is redundant. This attribute may be set to NULL.

The default value is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_PCACHE(3)

NAME

XGL_RAS_DEPTH – specifies the number of bits required to store the color of one pixel in a Raster

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ras ras;

For XGL Memory Rasters only:
xgl_object_set(ras, XGL_RAS_DEPTH, Xgl_usgn32, 0);

For XGL Window and Memory Rasters:
xgl_object_get(ras, XGL_RAS_DEPTH, Xgl_usgn32 *);

DESCRIPTION

This attribute defines the number of bits that are used to specify the color of one pixel in an XGL Raster.

For XGL Window Rasters, **XGL_RAS_DEPTH**(3) is read-only.

For Memory Rasters, XGL_RAS_DEPTH is read-write and can be changed at any time. However, only four depths are currently supported — 1, 4, 8, and 32. Of these four, the 1-bit depth can be used only for creating stipple patterns. The 1-bit Memory Rasters as stipple fills are used to fill surface primitives (see **XGL_3D_CTX_SURF_BACK_FILL_STYLE**(3) and **XGL_CTX_SURF_FRONT_FILL_STYLE**(3)) or raster primitives (see **XGL_CTX_RASTER_FPAT**(3)).

The 1-bit Memory Rasters can be used as the source Raster in the **xgl_context_copy_raster**(3) operator, where the Destination Context can have either a 1-bit, 4-bit, 8-bit, or 32-bit Memory Raster Device.

The 4-bit, 8-bit and 32-bit depths can be used to create Memory Rasters that the XGL application will render into.

Note that changing the depth of a Memory Raster results in the loss of all pixel data within the Raster.

Note also that 32-bit Memory Rasters do not use the Context ROP mode or plane mask when rendering. If a 32-bit Memory Raster is dithered into an 8-bit Memory Raster or an 8-bit frame buffer, the ROP mode and plane mask are not used.

For Memory Rasters, the default value is 8. For Window Rasters, the default value is device-dependent.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_copy_raster(3)
xgl_object_create(3)
XGL_3D_CTX_SURF_BACK_FILL_STYLE(3)
XGL_CTX_RASTER_FPAT(3)
XGL_CTX_SURF_FRONT_FILL_STYLE(3)

NOTES

32-bit deep Memory Rasters use just 24 bits to store color information. The remaining 8 bits are reserved for future use.

NAME

XGL_RAS_HEIGHT,
XGL_RAS_WIDTH – specify the height and width of a Raster Device

SYNOPSIS

#include <xgl/xgl.h>

Xgl_ras ras;

For XGL Memory Rasters only:
xgl_object_set(ras, XGL_RAS_HEIGHT, Xgl_usgn32, 0);
xgl_object_set(ras, XGL_RAS_WIDTH, Xgl_usgn32, 0);

For XGL Window and Memory Rasters:
xgl_object_get(ras, XGL_RAS_HEIGHT, Xgl_usgn32 *);
xgl_object_get(ras, XGL_RAS_WIDTH, Xgl_usgn32 *);

DESCRIPTION

These two attributes specify the height and width of an XGL Raster.

When using a Window Raster, the attributes are read-only and cannot be set. The width and height may change if the operator changes the size of the window using the window system.

When using a Memory Raster, these attributes may be set or reset at any time. The width and height cannot be larger than 4K × 4K.

Note that changing the height or width of a Memory Raster results in the loss of all pixel data within the Raster.

For Memory Rasters, the default value for both attributes is 256. For Window Rasters, the initial value is application and window system dependent.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_RAS_RECT_NUM,
XGL_RAS_RECT_LIST – define the application clip region for a Raster

SYNOPSIS

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

Xgl_ras         ras;

xgl_object_set(ras, XGL_RAS_RECT_NUM, Xgl_sgn32, 0);
xgl_object_set(ras, XGL_RAS_RECT_LIST, Xgl_irect *, 0);

xgl_object_get(ras, XGL_RAS_RECT_NUM, Xgl_sgn32 *);
xgl_object_get(ras, XGL_RAS_RECT_LIST, Xgl_irect *);
```

DESCRIPTION

These attributes define the number and list of clip rectangles in the application-specified clip list. The clip list is an array of rectangles that defines the device coordinate clipping imposed at rendering time on all primitives. The application specifies the list of clip rectangles by first specifying the number of rectangles in their list using XGL_RAS_RECT_NUM, and then specifying the actual list of rectangles that define the clip region using XGL_RAS_RECT_LIST.

One clip rectangle is defined by the following data structure:

```
typedef struct {
        Xgl_usgn32          xmin;
        Xgl_usgn32          xmax;
        Xgl_usgn32          ymin;
        Xgl_usgn32          ymax;
} Xgl_irect;
```

In the case of a window raster, the application clip list is merged with the window system clip list. The result is a list of rectangles XGL uses to determine the actual visible portion of the raster.

When getting the list of rectangles, the application must allocate all the memory necessary to store the rectangle clip list. Use XGL_RAS_RECT_NUM to determine exactly how many rectangles to allocate.

The default value for XGL_RAS_RECT_NUM is 0.

The default value for XGL_RAS_RECT_LIST is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NAME

XGL_RAS_SOURCE_BUFFER – specifies the buffer to be used as source during raster operations

SYNOPSIS

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

Xgl_ras          ras;

xgl_object_set(ras, XGL_RAS_SOURCE_BUFFER, Xgl_buffer_sel, 0);
xgl_object_get(ras, XGL_RAS_SOURCE_BUFFER, Xgl_buffer_sel*, 0);
```

DESCRIPTION

This attribute specifies the source buffer during raster operations **xgl_context_copy_buffer**(3) and **xgl_context_get_pixel**(3).

The possible values for this attribute are:

XGL_BUFFER_SEL_Z
: The buffer to be used as source is the Z buffer.

XGL_BUFFER_SEL_DRAW
: The buffer to be used as source is the draw buffer. Setting this option for a memory raster implies buffer 0.

A Value between 0 to N-1.
: For a window raster, N is the number of buffers allocated using attributes **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3) and **XGL_WIN_RAS_BUFFERS_REQUESTED**(3). For a memory raster, only buffer value of 0 is meaningful.

The default value is XGL_BUFFER_SEL_DRAW.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_copy_buffer(3)
xgl_context_get_pixel(3)

NAME

XGL_SFONT – a Stroke Font object

OVERVIEW

XGL_SFONT is the type of an object representing a Stroke Font Object. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

The Stroke Font object specifies the font used to render the text string in the **xgl_stroke_text(**3) operator. A single Stroke Font handle can be shared between different Contexts.

The following is a brief description of the attributes associated with Stroke Font objects. For more information, see the man page for each attribute.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

XGL_SFONT_COMMENT(3)
: Returns the comment string available with the Stroke Font Object, if one is available.

XGL_SFONT_DEFAULT_CHARACTER(3)
: Specifies a default character that will be used to replace undefined glyphs in the font.

XGL_SFONT_IS_MONO_SPACED(3)
: Returns if the font is *monospaced*. All the characters in a monospaced font are the same width.

XGL_SFONT_NAME(3)
: Returns the name of the font the Stroke Font object represents. This is the name that was given to **xgl_object_create**(3) when the Stroke Font Object was created.

When creating a font, using **xgl_object_create**(3)**,** XGL searches in the current Stroke Font directory. The Stroke Font directory can be set to an application-dependent value by using the **XGL_SYS_ST_SFONT_DIRECTORY**(3) attribute. The name of the font, needed at creation time, is the full name (including extension) of the file which stores the font.

NAME

XGL_SFONT_COMMENT – returns the comment associated with a font

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sfont       sfont;
xgl_object_get(sfont, XGL_SFONT_COMMENT, char **);
```

DESCRIPTION

This is a read-only Stroke Font object attribute, which returns the comment text associated with the font. In the current implementation of XGL, the name of the font is returned (which is the same result as if the Stroke Font attribute **XGL_SFONT_NAME**(3) was used).

SEE ALSO

xgl_object_get(3)
xgl_object_create(3)
XGL_SFONT_NAME(3)

NAME

XGL_SFONT_DEFAULT_CHARACTER – defines character used in place of undefined characters when rendering stroke text

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sfont       sfont;
xgl_object_set(sfont, XGL_SFONT_DEFAULT_CHARACTER, Xgl_usgn32, 0);
xgl_object_get(sfont, XGL_SFONT_DEFAULT_CHARACTER, Xgl_usgn32 *);
```

DESCRIPTION

This attribute defines the *default character* for a Stroke Font. It is the character that is rendered, when using stroke text, if no glyph is assigned within the requested font for the character value selected. Stroke text is rendered using the operator **xgl_stroke_text**(3)**.**

A Stroke Font object must be created before stroke text can be rendered using the stroke text operators. (For more information, see **xgl_object_create**(3)**.**) It is a good idea to specify a default character when creating the font with the xgl_object_create operator. Specify a character that is likely to exist, such as a blank space.

The default value is dependent on the font used.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
xgl_stroke_text(3)

NAME

XGL_SFONT_IS_MONO_SPACED – indicates whether the specified Stroke Font is monospaced

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sfont sfont;
xgl_object_get(sfont, XGL_SFONT_IS_MONO_SPACED, Xgl_boolean *);
```

DESCRIPTION

This attribute is a read-only attribute. When used in an xgl_object_get call for a given Stroke Font object, it will return TRUE if the font is a *monospaced* font, and FALSE otherwise.

A monospaced Stroke Font is one in which all the characters of the font have the same width. This makes it easier to compute the graphic area covered by a string of characters. With non-monospaced fonts, the application has to call the operator **xgl_stroke_text_extent**(3) each time it needs to know the width of a particular text string. With monospaced fonts, however, the application can easily determine the text string from the character attribute width because all characters in the font will have this same width.

SEE ALSO

xgl_object_get(3)
xgl_stroke_text_extent(3)

NAME

XGL_SFONT_NAME – specifies the name of a Stroke Font font family

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sfont sfont;
xgl_object_get(sfont, XGL_SFONT_NAME, char **);
```

DESCRIPTION

This is a read-only attribute that can be used to get the name of a Stroke Font associated with the Stroke Font object *sfont*.

Stroke Font objects are created with the operator **xgl_object_create**(3)**.** Here *create* means to get the handle of the Stroke Font information that is stored in the Stroke Font directory. (The directory is specified by the System State attribute **XGL_SYS_ST_SFONT_DIRECTORY**(3)**.**) In a given process, a Stroke Font object for a particular font name needs to be created only once. The object handle returned by the creation operator, xgl_object_create, can be used and shared simultaneously by different Contexts.

The list below shows the different Stroke Fonts XGL currently implements:

Cartographic
Cartographic_M
English_G
Greek
Greek_C
Greek_M
Headline
Italic_C
Italic_T
Miscellaneous
Miscellaneous_M
Roman
Roman_C
Roman_D
Roman_M
Roman_T
Script
Script_C

These fonts are stored in the XGL_SYS_ST_SFONT_DIRECTORY directory. The current fonts are stored with a *.font* extension. Obsolete font files (those which have the *.phont* extension) have been removed and replaced with symbolic links.

Note that not all of the fonts need to be present in the Stroke Font directory. Some of the fonts are very large, and, if they are not used by XGL applications on a particular system, they can be removed from the directory to save disk space.

The default value is *Roman_M*.

SEE ALSO

xgl_object_get(3)
xgl_object_create(3)
XGL_SYS_ST_SFONT_DIRECTORY(3)

NAME

XGL_STREAM – a Stream Device object

OVERVIEW

XGL_STREAM is the type of an object representing a Stream. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

The XGL Stream device provides protocol-independent stream pipelines, for creating formatted output such as PostScript or Computer Graphics Metafile (CGM). The XGL Stream Device maintains no protocol itself, but instead relies on an instantiated third-party device pipeline to implement the appropriate output protocol data format.

A Stream Device object is created using the **xgl_object_create**(3) operator. The application must specify XGL_STREAM as the value of the *type* parameter to **xgl_object_create**(3)**.** In addition, the application must also supply the *desc* parameter, which is a pointer to an *Xgl_obj_desc* union. The application needs to supply the information in the *stream* field of the *Xgl_obj_desc* union. The *stream* structure consists of two fields: *name*, which is the name identifying the specific device pipeline, and *desc*, which is a pointer to device specific information that may be required by this particular device pipeline.

After creating the Stream Device, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

A Stream Device object has all the Device attributes and possibly additional attributes provided by the specific device pipeline.

A brief overview of the generic Stream Device attributes follows. For more information, see the man page for each attribute.

XGL_DEV_COLOR_MAP(3)
: Specifies the Color Map Object used by the Stream Device object.

XGL_DEV_COLOR_TYPE(3)
: Specifies the color type (that is, the color model) of the Device. It can be set only once, when the Stream Device object is created. The two alternatives are: XGL_-COLOR_INDEX and XGL_COLOR_RGB.

XGL_DEV_CONTEXTS(3)
XGL_DEV_CONTEXTS_NUM(3)
: These attributes return the Context object handles used by the Stream Device object.

XGL_DEV_MAXIMUM_COORDINATES(3)
: Returns the maximum coordinate values for the Stream Device object.

XGL_DEV_PICK_BUFFER_SIZE(3)
: Returns the maximum pick buffer size that a Stream Device object can support efficiently.

XGL_DEV_REAL_COLOR_TYPE(3)
: Specifies the real color space used by the Stream Device object.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

SEE ALSO

XGL_CGM(3)

NAME

XGL_SYS_STATE – a System State object

OVERVIEW

XGL_SYS_STATE is the type of an object representing information relevant to the general level of an application. The operator **xgl_open**(3) creates and initializes a System State object and **xgl_close**(3) ends an XGL session.

The System State object acts as a repository for critical information that XGL requires for managing an XGL session. It keeps track of all other objects created by the application or by XGL itself.

Only one XGL System State per process is permitted at any given time. The application programmer can create a System State and set its attributes in the same call by providing the correct arguments in the list of attributes for **xgl_open**(3)**.**

The following is a brief description of the attributes associated with System State objects. For more information, see the man page for each attribute.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

XGL_SYS_ST_ERROR_DETECTION(3)
: Controls errors that are generated during an XGL session.

XGL_SYS_ST_ERROR_INFO(3)
: Returns a structure containing information about the most recent error that was detected within the XGL library during the execution of an XGL application.

XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION(3)
: Defines its own error-handling function to filter errors that are generated during an XGL session.

XGL_SYS_ST_SFONT_DIRECTORY(3)
: Specifies the directory path used by XGL when accessing Stroke Fonts used in the text primitives.

XGL_SYS_ST_VERSION(3)
: Returns the version number of the current XGL release.

NAME

XGL_SYS_ST_ERROR_DETECTION – controls optional error detection

SYNOPSIS

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

Xgl_sys_state          sys_st;

xgl_object_set(sys_st, XGL_SYS_ST_ERROR_DETECTION, Xgl_boolean, 0);
xgl_object_get(sys_st, XGL_SYS_ST_ERROR_DETECTION, Xgl_boolean *);
```

DESCRIPTION

This attribute controls whether optional error checking is executed as part of an XGL application.

The error-detection feature can be either TRUE or FALSE. When the flag is TRUE, additional error-checking tests are executed as part of any XGL call. For the most part, these additional tests check for USER, ARITHMETIC, and CONFIGURATION errors. When the flag is FALSE, XGL does not do these optional checks, but still checks for RESOURCE or SYSTEM errors and reports them.

The application can filter XGL errors by specifying its own error-reporting function with the **XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION**(3) attribute of the System State.

In XGL, errors are classified by the following five error categories:

SYSTEM:
: Internal errors, unsupported features, and errors that, generally, cannot be fixed by changing the application.

CONFIGURATION:
: Errors caused by improper installation or configuration of XGL (such as a .so not found).

RESOURCE:
: Unavailable resource errors including both hardware and software resources (such as memory, shared memory, window ID, frame buffer, etc.)

ARITHMETIC:
: Arithmetic exceptions (divide by 0, square root of -1, etc.)

USER:
: Errors caused by invalid function parameters, non-existent user files, or situations that are probably caused by application program logic errors.

XGL also includes the notion of RECOVERABLE vs. NONRECOVERABLE errors. These are known as error *types*.

A RECOVERABLE error is one where XGL can make reasonable assumptions about what the application intended to do and then continues processing. In most cases, the library uses the default value for an incorrect parameter and then continues the application.

A NONRECOVERABLE error is one where XGL, upon detection, immediately aborts further processing and returns control back to the caller. Errors of this type may cause a core dump if the application is allowed to continue execution because they leave XGL in an inconsistent state.

Error categories and types are related as follows:

Type	Category
NONRECOVERABLE	All SYSTEM and CONFIGURATION errors, most RESOURCE errors, some ARITHMETIC errors.
RECOVERABLE	Some RESOURCE errors, most ARITHMETIC errors, all USER errors.

Errors are identified by a code xx-##, where xx specifies an error file where the localized error message is stored (*di* is the device-independent error file, *xgl.mo*. Other values specify device-dependent files) and ## is an error number. NONRECOVERABLE errors are reported with error numbers between 0 and 99. RECOVERABLE errors have error numbers greater than 99.

For a list of XGL error messages, see the *XGL Programmer's Guide*. The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_SYS_ST_ERROR_INFO(3)
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION(3)

NAME

XGL_SYS_ST_ERROR_INFO – error information that describes the current error state

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sys_state          sys_st;
xgl_object_get(sys_st, XGL_SYS_ST_ERROR_INFO,Xgl_error_info *);
```

DESCRIPTION

This read-only System State attribute returns a structure containing information about the most recent error that was detected within the XGL library during the execution of an XGL application. This information can be used by a user-specified error notification function (see **XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION**(3)). The function would get the value of this attribute, examine the contents of the *Xgl_error_info* structure, and perform actions based on those contents.

The *Xgl_error_info* structure is defined as follows:

```
typedef struct {
        Xgl_error_type      type;
        Xgl_error_category  category;
        char*               id;
        char*               msg;
        char*               cur_op;
        char*               cur_obj;
        char*               operand1;
        char*               operand2;
} Xgl_error_info;
```

where the different member types are defined as:

type Two values: RECOVERABLE and NONRECOVERABLE. A RECOVERABLE error is one where the XGL library can make reasonable assumptions about what the application intended to do and then continues processing. A NONRECOVERABLE error is one where, upon detection, the XGL library immediately aborts further processing and returns control back to the caller.

category
Five values: SYSTEM, CONFIGURATION, RESOURCE, ARITHMETIC, and USER. See **XGL_SYS_ST_ERROR_DETECTION**(3) for more information.

id Identification code associated with the error. They are of the form xx-##, where xx specifies an error file where the localized error message is stored (*di* is the device-independent error file, *xgl.mo*. Other values specify device-dependent error files), and ## is an error number. NONRECOVERABLE errors are reported with error numbers between 0 and 99. RECOVERABLE errors have error numbers greater than 99.

msg The localized error message string associated with the error.

cur_op The current XGL operator that was being executed when the error occurred.

cur_obj The current XGL object being used when the error occurred.

operand1, *operand2*
Operands that give optional additional information about the error.

SEE ALSO

xgl_object_get(3)
XGL_SYS_ST_ERROR_DETECTION(3)
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION(3)

NAME

XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION – defines the function that is called in case of an error

SYNOPSIS

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

Xgl_sys_state          sys_st;

xgl_object_set(sys_st, XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION, void (*)(), 0);
xgl_object_get(sys_st, XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION, void (*(*))());
```

DESCRIPTION

This attribute gives the application the ability to provide its own error-handling function instead of using the XGL default function.

When an error is detected within an XGL application, the default function prints to *stderr* an XGL error code, a brief message describing the error, the name of the XGL operator that is currently being executed, the name of the current XGL object, and other (optional) information. By providing its own error-reporting function, the application can choose to filter some errors or choose its own method for alerting the user about an error.

The application can control the level of error detection by using the **XGL_SYS_ST_ERROR_DETECTION**(3) attribute of the System State.

The following is the interface (the call syntax and parameter list) for an application-provided error function.

(void)**my_error_function**(*Xgl_sys_state sys_state;*)

Within the error notification function, the application may get the structure containing current error state information (see **XGL_SYS_ST_ERROR_INFO**(3)) and handle the error based on the contents of this structure. For example, the following is a simple error notification function:

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

static Xgl_sgn32 newNotify(system_state)
    Xgl_sys_state system_state;
{
    Xgl_error_info    info;

    xgl_object_get (system_state, XGL_SYS_ST_ERROR_INFO, &info);

    printf ("id      = %s\n", info.id);
    printf ("msg     = %s\n", info.msg);
    printf ("cur_op  = %s\n", info.cur_op);
    printf ("cur_obj = %s\n", info.cur_obj);

    if (info.id < 100) {
        printf ("Non-Recoverable Error!0);
    } else {
        printf ("Recoverable Error!0);
    }
    return(1);
}
```

For further details, see **XGL_SYS_ST_ERROR_DETECTION**(3) and **XGL_SYS_ST_ERROR_INFO**(3)**.**
The default value is a pointer to the internal XGL error-reporting function.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_SYS_ST_ERROR_DETECTION(3)
XGL_SYS_ST_ERROR_INFO(3)

NAME

XGL_SYS_ST_SFONT_DIRECTORY – specifies the directory where XGL looks for Stroke Font files

SYNOPSIS

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

Xgl_sys_state          sys_st;

xgl_object_set(sys_st, XGL_SYS_ST_SFONT_DIRECTORY, char *, 0);
xgl_object_get(sys_st, XGL_SYS_ST_SFONT_DIRECTORY, char **);
```

DESCRIPTION

This attribute is used to specify the directory path used by XGL when accessing Stroke Fonts used in the text primitives (see **xgl_stroke_text**(3)). If XGL does not find the Stroke Font data before creating the Stroke Font object (see below), an error message is returned to the application, and the operation is cancelled.

The path name for the Stroke Font directory can also be modified without setting an XGL attribute, by using the $XGLHOME environment variable. If $XGLHOME has been set, the default value for XGL_SYS_ST_SFONT_DIRECTORY is *$XGLHOME/lib/xglfonts/stroke*. If it has not been set, the default is */opt/SUNWits/Graphics-sw/xgl/lib/xglfonts/stroke*.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
xgl_stroke_text(3)
XGL_CTX_SFONT_0(3)

NAME

XGL_SYS_ST_VERSION – returns the version number of the XGL release being used

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sys_state          sys_st;
xgl_object_get(sys_st, XGL_SYS_ST_VERSION, char **);
```

DESCRIPTION

This attribute is read-only and is provided for applications that need to inquire about the current version number (release number) of the XGL software.

The current default value is *XGL 3.1*.

SEE ALSO

xgl_object_get(3)

NAME

XGL_TMAP – a Texture Map object

OVERVIEW

XGL_TMAP is the *type* of an object representing an XGL Texture Map object. It is used in texturing of surfaces (polygons, triangle strips, etc.) rendered through a 3D Context.

When a Texture Map object is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

After creating the Texture Map object, using **xgl_object_create**(3)**,** an application can inquire about the value of its attributes by using the **xgl_object_get**(3) operator.

A System State object must have been created with the **xgl_open**(3) operator before any Texture Map object can be created.

A brief overview of the attributes follows. For more information, see the man page for each attribute.

XGL_TMAP_COORD_SOURCE(3)
: Defines the coordinate source for mapping texture onto a 3D surface.

XGL_TMAP_DESCRIPTOR(3)
: Defines a texture descriptor.

XGL_TMAP_DOMAIN(3)
: Defines the domain to use in texture mapping.

XGL_TMAP_PARAM_TYPE(3)
: Defines different parameterization methods for mapping texture onto a 3D surface.

XGL_TMAP_T0_INDEX(3)
XGL_TMAP_T1_INDEX(3)
: These attributes are indices into the vertex data array (specified in the point list), and they indicate which of the data values to use in texture mapping.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

NAME

XGL_TMAP_COORD_SOURCE – specifies method used to compute texture coordinate values

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_tmap        tmap;
xgl_object_set(tmap, XGL_TMAP_COORD_SOURCE, Xgl_texture_coord_source *, 0);
xgl_object_get(tmap, XGL_TMAP_COORD_SOURCE, Xgl_texture_coord_source *);
```

DESCRIPTION

This attribute specifies the initial source of the texture coordinate. Depending on a texture map's parameterization method, this may or may not be the final texture coordinate (this initial source may be processed into the final texture coordinate).

XGL accepts the following values for *Xgl_texture_coord_source*:

XGL_TEXTURE_COORD_VERTEX
: The source texture value is the vertex coordinate value, transformed to World Coordinates (WC).

XGL_TEXTURE_COORD_NORMAL
: The source texture coordinate value is the vertex normal value, transformed to WC. If a vertex normal is not explicitly defined with the primitive, the calculated vertex normal is used instead.

XGL_TEXTURE_COORD_DATA
: The source texture coordinate's source is a member of the vertex's floating point data. A separate index indicates the member of the list. If the primitive does not include the floating point data list for each vertex, this texture coordinate value will be set to 0.

The default value is XGL_TEXTURE_COORD_DATA .

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NAME

XGL_TMAP_DESCRIPTOR – specifies the properties of the texture map image

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_tmap        tmap;
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, Xgl_texture_general_desc, 0);
xgl_object_get(tmap, XGL_TMAP_DESCRIPTOR, Xgl_texture_general_desc *);
```

DESCRIPTION

This attribute specifies the list of *textures* associated with this Texture Map object. The *textures* are used when rendering XGL 3D surface primitives.

XGL_TMAP_DESCRIPTOR is a pointer to a C structure of type *Xgl_texture_general_desc*, which is defined as follows:

```
typedef struct {
        Xgl_texture_type                texture_type;
        Xgl_texture_comp_info           comp_info;
        union {
                Xgl_texture_general_mipmap_desc   mipmap;
        } texture_info;
} Xgl_texture_general_desc;
```

Parameters

texture_type

Each texture descriptor has a *texture_type*. Currently, Xgl_texture_type defines XGL_TEXTURE_TYPE_MIPMAP as the only possible value.

comp_info

This parameter explains how different channels in the texture map are used to affect the rendering. The *Xgl_texture_comp_info* structure is defined as:

```
typedef union {
        Xgl_texture_color_comp_info             color_info;
} Xgl_texture_comp_info;
```

The *Xgl_texture_color_comp_info* structure is defined as:

```
typedef struct {
        Xgl_usgn32                  num_render_comp_desc;
        Xgl_render_component_desc   render_component_desc[2];
        Xgl_usgn32                  num_channels[2];
        Xgl_usgn32                  channel_number[2];
} Xgl_texture_color_comp_info;
```

This structure can have a maximum of 2 render components. The *num_render_comp_desc* parameter gives the number of render components used by this particular Texture Map object. Note that in the current release of XGL, only *num_render_comp_desc* of 1 is supported. The *num_channels[2]* parameter gives the number of channel used to do color composition. The *channel_number[2]* indicates the channel number used when *num_channel* is 1. The *Xgl_render_component_desc* structure is defined as:

```
typedef struct {
        Xgl_render_component    comp;
        Xgl_texture_op          texture_op;
        union                   Xgl_op_info {
                Xgl_texture_blend_op            blend;
                Xgl_texture_decal_op            decal;
                Xgl_texture_blend_intrinsic_op  blend_int;
        } op;
} Xgl_render_component_desc;
```

The *comp* parameter determines which lighting component in the rendering pipeline (diffuse, reflected, and final-after depth cueing) is affected by this particular texture description. The *texture_op* parameter specifies how the current color value is combined with the value obtained from the texture map. The table below lists the valid combinations:

Texture Operations

# of Channel	Replace	Modulate	Decal	Blend	Decal Intrinsic	Blend Intrinsic
1		$C_f = S_t C_i$		$C_f = C_{k1} * (1-S_t) + C_{k2} * S_t$		$C_f = C_i * (1-S_t) + C_{k1} * S_t$
2		$C_f = S_t C_i$		$C_f = C_{k1} * (1-S_t) + C_{k2} * S_t$		$C_f = C_i * (1-S_t) + C_{k1} * S_t$
3	$C_f = C_t$	$C_f = C_t C_i$		$C_f = C_{k1} * (1-C_t) + C_{k2} * C_t$		$C_f = C_i * (1-C_t) + C_{k1} * C_t$
4	$C_f = C_t$	$C_f = C_t C_i$	$C_f = (1-\alpha_t) * C_{k1} + \alpha_t * C_t$	$C_f = C_{k1} * (1-C_t) + C_{k2} * C_t$	$C_f = C_i * (1-C_t) + \alpha_t * C_t$	$C_f = C_i * (1-C_t) + C_{k1} * C_t$

Where: C_{k1} = Constant color 1
C_{k2} = Constant color 2
C_i = Incoming color
C_f = Final color
S_t = Single channel texture image
C_t = First three channels of a 4 channel texture image
α_t = Second channel of a 2 channel texture image or
Fourth channel of a 4 channel texture image

Note: Blank fields indicate undefined values. The values from the texture image are in the range [01]. Currently in XGL, a 3 channel texture image is specified as a 32-bit value. The upper byte is ignored for 3 channel texture operations.

In the *Xgl_op_info* union, the texture operations *blend*, *decal*, and *blend_int* use values defined in the structures *Xgl_texture_blend_op*, *Xgl_texture_decal_op*, and *Xgl_texture_blend_intrinsic_op*, respectively. The blending operation gets the constant colors C_{k1} and C_{k2} from the structure *Xgl_texture_blend_op*. The *decal* and *blend_int* operations get the constant color C_{k1} from the structures *Xgl_texture_*-

decal_op and *Xgl_texture_blend_intrinsic_op*, respectively.

mipmap This parameter defines the mipmap specific fields. The *Xgl_texture_general_-mipmap_desc* structure is defined as:

```
typedef struct {
        Xgl_texture             texture_map;
        float                   max_u_freq;
        float                   max_v_freq;
        Xgl_texture_boundary    u_boundary;
        Xgl_texture_boundary    v_boundary;
        Xgl_usgn8               boundary_values[4];
        float                   depth_interp_factor;
        Xgl_texture_interp_info interp_info;
        Xgl_matrix_f2d          orientation matrix;
} Xgl_texture_general_mipmap_desc;
```

The *texture_map* parameter gives a pointer to the texture map object. A texture map object is created using **xgl_object_create**(3) with XGL_MIPMAP_TEXTURE as the object type.

Note: At this release, the fields *max_u_freq* and *max_v_freq* are not implemented.

The parameters *u_boundary* and *v_boundary* specify the boundary condition to apply when texture bounds are exceeded. The available boundary options are specified by the *Xgl_texture_boundary* structure. The possible values are:

XGL_TEXTURE_BOUNDARY_CLAMP
: Finish texturing using a constant value.

XGL_TEXTURE_BOUNDARY_WRAP
: Repeat the texture.

XGL_TEXTURE_BOUNDARY_TRANSPARENT
: Finish rendering as though texturing were not being performed.

XGL_TEXTURE_BOUNDARY_MIRROR
: Reverse texel sampling creating a *mirrored alternating* pattern.

XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY
: Use a color specified by the closest boundary texel.

Note that if *u* and *v* boundary conditions are not identical, and if both values of *u* and *v* exceed the texture boundary in both directions, the boundary methods will be used in the following order:

(1) XGL_TEXTURE_BOUNDARY_TRANSPARENT

(2) XGL_TEXTURE_BOUNDARY_CLAMP

(3) XGL_TEXTURE_BOUNDARY_WRAP

(4) XGL_TEXTURE_BOUNDARY_MIRROR

(5) XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY

The *boundary_values* parameter specifies the value to clamp to when either *u* or *v* value exceeds bounds. The *depth_interp_factor* parameter is a floating point number that allows the application to adjust the sample level in either direction to obtain a more or less aliased textured surface. A value of 1.0 moves the depth away from the mipmap base creating a blurry event. A value of -1.0 moves the depth towards the mipmap pyramid base for a more detailed picture.

The *interp_info* parameter gives information on the nature of sampling desired by the application. The *Xgl_texture_interp_info* structure is defined as:

```
typedef struct {
        Xgl_texture_interp_method    filter1;
        Xgl_texture_interp_method    filter2;
} Xgl_texture_interp_info;
```

During the texturing process, the data from the texture mipmap can be sampled or interpolated in a number of ways in order to come up with a value to apply to the pixel. The available methods of sampling and interpolating the mipmap are specified by the *Xgl_texture_interp_method* structure. The possible values are:

Note: The word *texel* used below refers to one element stored in a texture map.

XGL_TEXTURE_INTERP_POINT
: Sample one texel point from the base (top or first) mipmap level at the closest u and v to the pixel.

XGL_TEXTURE_INTERP_BILINEAR
: Sample the four texels closest to the center of the pixel from the base mipmap level and average them.

XGL_TEXTURE_INTERP_MIPMAP_POINT
: Sample one texel point from the nearest mipmap level at the closest u and v to the pixel.

XGL_TEXTURE_INTERP_MIPMAP_BILINEAR
: Sample the four texels closest to the center of the pixel from the nearest mipmap level and average them.

XGL_TEXTURE_INTERP_MIPMAP_TRILINEAR
: Sample the four texels closest to the center of the pixel from the nearest mipmap level above the depth *d* and the four texels closest from the mipmap level below the depth *d* and average all 8 of them.

XGL_TEXTURE_INTERP_MIPMAP_PT_LINEAR
: Chose the two mipmaps that most closely match the size fo the pixel being textured and find the nearest to the center of the pixel to produce a texture value from each mipmap. The final texture value is an average of those two values.

filter1 is used if the textured pixel maps to an area greater than one texel. *filter2* is used if the textured pixel maps to an area less than or equal to one texel. We strongly recommend that *filter2* only use XGL_TEXTURE_INTERP_POINT or XGL_-

TEXTURE_INTERP_BILINEAR.

The *orientation_matrix* parameter defines a matrix by which *textures* can be translated and / or rotated. This parameter specifies a 3×2 matrix by which textures can be rotated or translated across the face of the polygon by transforming the *u* and *v* values through the matrix. The *u* and *v* values are specified as data values in the point list of a surface primitive (for example, **xgl_multi_simple_polygon**(3)**, xgl_triangle_strip**(3)**,** etc.) if **XGL_TMAP_COORD_SOURCE**(3) is set to XGL_TEXTURE_COORD_DATA. The *u, v* values defined in the point list are subject to the **XGL_TMAP_T0_INDEX**(3) and **XGL_TMAP_T1_INDEX**(3) attributes before the orientation matrix is applied.

The default value is NULL.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
xgl_multi_simple_polygon(3)
xgl_polygon(3)
xgl_quadrilateral_mesh(3)
xgl_triangle_list(3)
xgl_triangle_strip(3)

XGL_TMAP_COORD_SOURCE(3)
XGL_TMAP_T0_INDEX(3)
XGL_TMAP_T1_INDEX(3)

NOTES

Texturing is only supported when the color type of the raster attached to a 3D Context is XGL_COLOR_RGB.

NAME

XGL_TMAP_DOMAIN – specifies that the texture is to be applied to color domain

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_tmap        tmap;
xgl_object_set(tmap, XGL_TMAP_DOMAIN, Xgl_texture_domain, 0);
xgl_object_get(tmap, XGL_TMAP_DOMAIN, Xgl_texture_domain);
```

DESCRIPTION

This attribute specifies that the texture is to be applied to color domain. XGL accepts the following value for *Xgl_texture_domain*:

XGL_TEXTURE_COLOR
: Texture value (texel) obtained from the texture map is used to modify the primitive's color according to the color composition method.

The default value is XGL_TEXTURE_COLOR.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NAME

XGL_TMAP_PARAM_TYPE – defines the parameterization method for texture mapping

SYNOPSIS

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

Xgl_tmap                tmap;

xgl_object_set(tmap, XGL_TMAP_PARAM_TYPE, Xgl_texture_param_type *, 0);
xgl_object_get(tmap, XGL_TMAP_PARAM_TYPE, Xgl_texture_param_type *);
```

DESCRIPTION

This attribute defines the parameterization method to use in mapping the texture to a 3D surface. XGL accepts the following value for *Xgl_texture_param_type*:

XGL_TEXTURE_PARAM_EXPLICIT
: This attribute value requires the application to provide the values of *u* and *v* as part of the vertex information in the point list of 3D surface primitives.

The default value is XGL_TEXTURE_PARAM_EXPLICIT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)

NAME

XGL_TMAP_T0_INDEX,
XGL_TMAP_T1_INDEX – define the indices into the vertex data array

SYNOPSIS

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

Xgl_tmap        tmap;

xgl_object_set(tmap, XGL_TMAP_T0_INDEX, Xgl_usgn32, 0);
xgl_object_set(tmap, XGL_TMAP_T1_INDEX, Xgl_usgn32, 1);

xgl_object_get(tmap, XGL_TMAP_T0_INDEX, Xgl_usgn32 *);
xgl_object_get(tmap, XGL_TMAP_T1_INDEX, Xgl_usgn32 *);
```

DESCRIPTION

If **XGL_TMAP_COORD_SOURCE**(3) is set to XGL_TEXTURE_COORD_DATA, these attributes are pointers to the list of data values to use as *u*, and *v* texture coordinates. *t0* is equal to the *u* value, and *t1* is equal to the *v* value.

The values of *u*, and *v* are specified as real numbers in the vertex array. The coordinate space of the texture map that *t0* values correspond to is a space that places 0.0 at the left edge of the texture and 1.0 at the right edge of the texture. For *t1*, the coordinate space of the texture map that *v* values use is a space that places 0.0 at the bottom edge of the texture and 1.0 at the top edge of the texture.

If *u* and *v* values are not assigned to polygon vertices in a linear way, that is, if there is no linear transformation that maps *u,v* coordinates to the polygon's vertices, then a textured polygon may look different when rendered on different graphics accelerators. This is because some graphics hardware devices break up more complex polygons into triangles, and if the mapping is non-linear, the triangles won't all be textured the same from device to device.

The default value for XGL_TMAP_T0_INDEX is 0.

The default value for XGL_TMAP_T1_INDEX is 1.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_TMAP_COORD_SOURCE(3)

NAME

XGL_TRANS – a Transform object

OVERVIEW

XGL_TRANS is the type of an object representing a Transform object. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

When a Transform is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

After creating the Transform, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_OBJ_APPLICATION_DATA(3)
: This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
: This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
: This read-only attribute defines the type of an XGL object.

XGL_TRANS_DATA_TYPE(3)
: Specifies the data type (integer, single-precision floating point, or fixed point) of the matrix in the Transform object.

XGL_TRANS_DIMENSION(3)
: Specifies the dimension (2D or 3D) of the Transform object.

NAME

XGL_TRANS_DATA_TYPE – defines the data type of a Transform

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_trans trans;
xgl_object_set(trans, XGL_TRANS_DATA_TYPE, Xgl_data_type, 0);
xgl_object_get(trans, XGL_TRANS_DATA_TYPE, Xgl_data_type *);
```

DESCRIPTION

This attribute defines the data type of a matrix associated with a Transform object. XGL accepts the following enumerated values for *Xgl_data_type*:

XGL_DATA_INT
: The Transform's matrix is stored as a 32-bit signed integer. This value is permitted only when the value of **XGL_TRANS_DIMENSION**(3) is XGL_TRANS_2D.

XGL_DATA_FLT
: The Transform's matrix is stored as single-precision floating point.

XGL_DATA_DBL
: The Transform's matrix is stored as double precision floating point.

The default value is XGL_DATA_FLT.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)
XGL_TRANS_DIMENSION(3)

NAME

XGL_TRANS_DIMENSION – defines the space (2D or 3D) in which a Transform operates

SYNOPSIS

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

Xgl_trans              trans;

xgl_object_set(trans, XGL_TRANS_DIMENSION, Xgl_trans_dimension, 0);
xgl_object_get(trans, XGL_TRANS_DIMENSION, Xgl_trans_dimension *);
```

DESCRIPTION

This attribute defines the dimension of the space in which the Transform operates. XGL accepts the following enumerated values for Xgl_trans_dimension:

XGL_TRANS_2D
: The Transform operates in two-dimensional space, making it suitable for 2D Contexts.

XGL_TRANS_3D
: The Transform operates in three-dimensional space, making it suitable for 3D Contexts.

The default value is XGL_TRANS_3D.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_WIN_RAS – a Window Raster Device object

OVERVIEW

XGL_WIN_RAS is the type of an object representing a Window Raster Device. It is used as the *type* parameter value for **xgl_object_create**(3)**.**

A specific Device object is the Window Raster Device, which draws into *pixels* that are stored in a *window* on a graphics frame buffer managed by a window system. These pixels are visible to the operator. In contrast, the Memory Raster Device object (see **xgl_object_create**(3) and **XGL_MEM_RAS**(3)) stores images in memory where they are not visible. The application must create the window before creating an XGL Window Raster Device on it.

A window can have only one Window Raster Device associated with it.

A Window Raster Device object is created using the **xgl_object_create**(3) operator. The application must specify XGL_WIN_RAS as the value of the *type* parameter to **xgl_object_create**(3)**.** In addition, the application must also supply the *desc* parameter, which is a pointer to an *Xgl_obj_desc* union. The application needs to supply the information in the *win_ras* field of the *Xgl_obj_desc* union. The *win_ras* structure consists of two fields: *type*, which is the window type, and *desc*, which is a pointer to descriptor information that identifies the window of the given type. The possible values for the window type are (currently, there is only one value):

XGL_WIN_X

The Window Raster Device is associated with an X11 window that may have been obtained from an XView Canvas or created by other means. The value of *desc* is a pointer to an *Xgl_X_window* structure. This example shows how to create an XGL_WIN_RAS on an X window:

```
#include <xgl/xgl.h>
Xgl_X_window        xgl_x;              /* X window descriptor */
Xgl_obj_desc        obj_desc;           /* XGL object descriptor */
Xgl_win_ras         ras;                /* newly created window raster */
Xgl_sys_state       sys_state;          /* XGL system state */

xgl_x.X_display = /* the X display identifier */
xgl_x.X_screen = /* the X screen identifier */
xgl_x.X_window = /* the X window identifier */

obj_desc.win_ras.type = XGL_WIN_X;
obj_desc.win_ras.desc = &xgl_x;

ras = xgl_object_create(sys_state, XGL_WIN_RAS, &obj_desc, 0);
```

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

```
/* WIN_X protocols */
#define XGL_WIN_X_PROTO_DEFAULT         (0x000)
#define XGL_WIN_X_PROTO_XLIB            (0x100)
#define XGL_WIN_X_PROTO_PEX             (0x200)
#define XGL_WIN_X_PROTO_DGA             (0x400)
```

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

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

After creating the Window Raster Device, using **xgl_object_create**(3)**,** the application can inquire about the values of its attributes by using the **xgl_object_get**(3) operator.

A Window Raster Device object shares many attributes with a Memory Raster Device object. These attributes have the common prefix XGL_RAS_. The attributes that are specific to the Window Raster Device have the prefix XGL_WIN_RAS_.

A brief overview of all of the attributes follows. For more information, see the man page for each attribute.

XGL_DEV_COLOR_MAP(3)
: Specifies the Color Map Object used by the Window Raster Device object.

XGL_DEV_COLOR_TYPE(3)
: Specifies the color type (that is, the color model) of the Raster. It can be set only once, when the Window Raster Device object is created. The two alternatives are: XGL_COLOR_INDEX and XGL_COLOR_RGB.

XGL_DEV_CONTEXTS(3)
XGL_DEV_CONTEXTS_NUM(3)
: These attributes return the Context object handles used by the Window Raster Device object.

XGL_DEV_MAXIMUM_COORDINATES(3)
: Returns the maximum coordinate values for the Window Raster Device object.

XGL_DEV_PICK_BUFFER_SIZE(3)
: Returns the maximum pick buffer size that a Window Raster Device object can support efficiently.

XGL_DEV_REAL_COLOR_TYPE(3)
: Specifies the real color space used by the Window Raster Device object.

XGL_OBJ_APPLICATION_DATA(3)
This attribute allows the application program to store its own data with an object.

XGL_OBJ_SYS_STATE(3)
This read-only attribute returns the System State that was used to create the object.

XGL_OBJ_TYPE(3)
This read-only attribute defines the type of an XGL object.

XGL_RAS_DEPTH(3)
XGL_RAS_HEIGHT(3)
XGL_RAS_WIDTH(3)

These attributes give the size of the Window Raster Device. They are read-only and cannot be set for a Window Raster Device. The width and height may change, however, if the operator changes the size of the window using the window system.

XGL_RAS_RECT_LIST(3)
XGL_RAS_RECT_NUM(3)

These attributes define the number and list of clip rectangles in the application-specified clip list.

XGL_RAS_SOURCE_BUFFER(3)
This attribute specifies the buffer to be used as source during raster operations.

XGL_WIN_RAS_BACKING_STORE(3)
This attribute provides backing store support for a Window Raster.

XGL_WIN_RAS_BUF_DISPLAY(3)
XGL_WIN_RAS_BUF_DRAW(3)
XGL_WIN_RAS_BUF_MIN_DELAY(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)
XGL_WIN_RAS_BUFFERS_REQUESTED(3)

These attributes control double buffering. XGL_WIN_RAS_BUF_DISPLAY specifies the current display buffer. XGL_WIN_RAS_BUF_DRAW specifies the current read / write buffer. XGL_WIN_RAS_BUF_MIN_DELAY specifies the minimum time delay between buffer switches. XGL_WIN_RAS_BUFFERS_ALLOCATED reflects the number of buffers that are available to the application for use. XGL_WIN_RAS_BUFFERS_REQUESTED specifies the number of hardware buffers requested by the application.

XGL_WIN_RAS_DESCRIPTOR(3)
Returns the window descriptor associated with the Window Raster Device. This is the value of the *win_ras_desc* field in the *Xgl_obj_desc* union.

XGL_WIN_RAS_PIXEL_MAPPING(3)
The *pixel mapping array* that maps the application's color indices to hardware color indices.

XGL_WIN_RAS_POSITION(3)
Specifies the position of the Window Raster object.

XGL_WIN_RAS_TYPE(3)
Returns the window type set via the *type* parameter to the create call.

XGL_WIN_RAS_STEREO_MODE(3)
Specifies whether the window being rendered into is a stereo window or a mono window.

NOTE The current implementation of XGL, using Direct Graphics Access (DGA), does not support using fork() or vfork() if there is an active XGL Window Raster.

NAME

XGL_WIN_RAS_BACKING_STORE – backing store support for a Window Raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras                ras;
void xgl_object_set(ras, XGL_WIN_RAS_BACKING_STORE, Xgl_boolean, 0);
void xgl_object_get(ras, XGL_WIN_RAS_BACKING_STORE, Xgl_boolean *);
```

DESCRIPTION

This attribute is set by the application to request backing store support from XGL for a Window Raster after backing store support is requested through Xlib call *XChangeWindowAttributes()*.

This request is only a hint and it is up to the server to decide whether to actually support backing store. No expose event will be issued when backing store is on. Even when backing store is supported, the server may decide to withdraw the support due to resource constraints and other reasons, so the application should always deal with an expose event (say by redrawing) by itself.

When XGL_WIN_RAS_BACKING_STORE is set to FALSE, XGL will drop backing store support from the XGL side. It is then the application's responsibility to request the server to drop the backing store support through Xlib call *XChangeWindowAttributes()*.

Since backing store and double-buffering cannot be supported at the same time, XGL will grant the request for backing store or double-buffering depending on which is requested first. If a request for backing store is made through setting XGL_WIN_RAS_BACKING_STORE to TRUE, XGL will check the value of XGL_WIN_RAS_BUFFERS_ALLOCATED. If the value of XGL_WIN_RAS_BUFFERS_ALLOCATED is greater than 1, XGL will not initiate the backing store request and in turn set the value of XGL_WIN_RAS_BACKING_STORE to FALSE, which can be retrieved through xgl_object_get.

If the backing store memory as allocated by the server is from the system, then only the inverse clipped-list will be used for all backing store operations.

Since the X server is not handling back copy of Z Buffer and Accumulation Buffer for backing store, support for Z Buffer and Accumulation Buffer in backing store is specified as follows:

When neither Z Buffer nor Accumulation Buffer is used:

No differences in backing store support when hardware or software, Z Buffer and/or Accumulation Buffer is used.

When Z Buffer and/or Accumulation Buffer is used, and:

The display device uses software Z Buffer and/or Accumulation Buffer.

> The backing store will share the same software buffers with the display window, so the contents will always be in-sync without copying.

The display device uses hardware Z Buffer and/or Accumulation Buffer.

> The buffers cannot be shared with the backing store. The device-pipeline for the backing store will use a separate Z Buffer for rendering and/or a separate Accumulation Buffer to accumulate. The resulting image after damage repair will

likely be inaccurate especially if the clipped-list has changed. Thus, backing store should not be used when accumulation is needed or HLHSR mode is on for 3D context.

The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)

NAME

XGL_WIN_RAS_BUF_DISPLAY – current display buffer

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_set(ras, XGL_WIN_RAS_BUF_DISPLAY, Xgl_usgn32, 0);
xgl_object_get(ras, XGL_WIN_RAS_BUF_DISPLAY, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the current display buffer, which is displayed in the application window.

The valid values for this attribute are those between 0 and XGL_WIN_RAS_BUFFERS_ALLOCATED – 1.

This attribute is changed implicitly by the primitive **xgl_context_new_frame**(3) if:

- The context flag XGL_CTX_NEW_FRAME_SWITCH_BUFFER is set in the attribute **XGL_CTX_NEW_FRAME_ACTION**(3).
- The value of XGL_WIN_RAS_BUF_DISPLAY differs from XGL_WIN_RAS_BUF_DRAW.

The default value is 0.

SEE ALSO

xgl_context_new_frame(3)
xgl_object_get(3)
xgl_object_set(3)
XGL_CTX_NEW_FRAME_ACTION(3)
XGL_WIN_RAS_BUF_DRAW(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)

NAME

XGL_WIN_RAS_BUF_DRAW – current read / write buffer

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_set(ras, XGL_WIN_RAS_BUF_DRAW, Xgl_usgn32, 0);
xgl_object_get(ras, XGL_WIN_RAS_BUF_DRAW, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies, for all XGL drawing primitives, the buffer to read and write pixels into.

The valid values for this attribute are those between 0 and XGL_WIN_RAS_BUFFERS_-ALLOCATED – 1.

This attribute is changed implicitly by the primitive **xgl_context_new_frame**(3) if:

- The context flag XGL_CTX_NEW_FRAME_SWITCH_BUFFER is set in the attribute **XGL_-CTX_NEW_FRAME_ACTION**(3)**.**
- The value of XGL_WIN_RAS_BUF_DISPLAY differs from XGL_WIN_RAS_BUF_DRAW.

If XGL_WIN_RAS_BUF_DRAW equals XGL_WIN_RAS_BUF_DISPLAY, and XGL_WIN_RAS_BUF_DRAW is set so that it is *not* equal to XGL_WIN_RAS_BUF_DISPLAY, the contents of XGL_WIN_RAS_-BUF_DRAW are undefined. On some devices, however, the buffer contents are unchanged. Please see your accelerator guide.

The default value is 0.

SEE ALSO

xgl_context_new_frame(3)
xgl_object_get(3)
xgl_object_set(3)

XGL_CTX_NEW_FRAME_ACTION(3)
XGL_WIN_RAS_BUF_DISPLAY(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)

NAME

XGL_WIN_RAS_BUF_MIN_DELAY – minimum time delay between buffer switches

SYNOPSIS

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

Xgl_win_ras          ras;

xgl_object_set(ras, XGL_WIN_RAS_BUF_MIN_DELAY, Xgl_usgn32, 0);
xgl_object_get(ras, XGL_WIN_RAS_BUF_MIN_DELAY, Xgl_usgn32 *);
```

DESCRIPTION

This attribute specifies the minimum milliseconds to wait between switches of the display and draw buffers. Minimum delay allows the application to tell XGL to slow down the animation loop so that it appears synchronized at a rate that the application rendering can support.

The default value is 0.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)

NAME

XGL_WIN_RAS_BUFFERS_ALLOCATED – number of buffers allocated in hardware underlying a Window Raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_get(ras, XGL_WIN_RAS_BUFFERS_ALLOCATED, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is the number of buffers allocated in the hardware underlying a Window Raster. It is a read-only attribute.

To change the number of buffers allocated, the application must use **XGL_WIN_RAS_-BUFFERS_REQUESTED**(3)**.** This attribute is the number of buffers granted after setting **XGL_-WIN_RAS_BUFFERS_REQUESTED**(3)**.**

A value of 1 indicates the hardware has only a *single buffer* and the application might do color map double buffering, if possible. A value of 2 indicates that the hardware is *double buffered*.

The value of this attribute determines the valid values for the attributes **XGL_WIN_RAS_-BUF_DISPLAY**(3) and **XGL_WIN_RAS_BUF_DRAW**(3)**.**

The default value is 1 (single buffered).

SEE ALSO

xgl_object_get(3)

XGL_WIN_RAS_BUF_DISPLAY(3)
XGL_WIN_RAS_BUF_DRAW(3)
XGL_WIN_RAS_BUFFERS_REQUESTED(3)

NAME

XGL_WIN_RAS_BUFFERS_REQUESTED – number of buffers requested by the application

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_set(ras, XGL_WIN_RAS_BUFFERS_REQUESTED, Xgl_usgn32, 0 );
xgl_object_get(ras, XGL_WIN_RAS_BUFFERS_REQUESTED, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is set by the application to request that XGL allocate a number of hardware buffers. If the application sets this to a value of 1, XGL will do single buffering. If the application sets this to a value of 2, XGL will do double buffering, if it is supported by the hardware underlying the Window Raster. The application should get the **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3) attribute value immediately after setting this attribute to see how many buffers were allocated in the hardware.

The allocation does not provide the requested number of buffers if any of the following are true:

- The hardware does not support the number of requested buffers. Determine via **xgl_inquire**(3)**.**
- Although the hardware does support the requested number of buffers, the resource is not available.
- Setting **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3) to the requested value would cause the current values of XGL_WIN_RAS_BUF_DRAW or XGL_WIN_RAS_BUF_DISPLAY to be invalid. The latter two must be greater than or equal to 0 and less than XGL_WIN_RAS_BUFFERS_ALLOCATED.
- The current value of XGL_RAS_ACCUM_SRC must be greater than or equal to 0 and less than XGL_WIN_RAS_BUFFERS_ALLOCATED.
- Backing store is on (**XGL_WIN_RAS_BACKING_STORE**(3) is TRUE) and XGL_WIN_RAS_BUFFERS_REQUESTED is greater than 1.

After setting this attribute, if the **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3) value is 1, the hardware has only a *single buffer*, and the application might do color map double buffering, if possible.

After setting this attribute, if the **XGL_WIN_RAS_BUFFERS_ALLOCATED**(3) value is 2, the hardware supports *double buffering* in which the initial content of the newly created buffer is undefined.

The value of this attribute determines the valid values for the attributes **XGL_WIN_RAS_BUF_DISPLAY**(3) and **XGL_WIN_RAS_BUF_DRAW**(3)**.**

The default value is 1 (single buffered).

SEE ALSO

xgl_inquire(3)
xgl_object_get(3)
xgl_object_set(3)
XGL_WIN_RAS_BACKING_STORE(3)
XGL_WIN_RAS_BUF_DISPLAY(3)
XGL_WIN_RAS_BUF_DRAW(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)

NAME

XGL_WIN_RAS_DESCRIPTOR – the window descriptor for the Window Raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_get(ras, XGL_WIN_RAS_DESCRIPTOR, void * );
```

DESCRIPTION

This read-only attribute is the *window descriptor* passed to XGL on the creation of the Window Raster. It is the window descriptor for the window that XGL draws into. The application receives it from the window system upon window creation.

The value of this attribute is the same as the value of the *win_ras→desc* parameter, a member of the Xgl_obj_desc union, that the application passes to XGL upon Window Raster creation (with the operator **xgl_object_create**(3)).

When **xgl_object_get**(3) is used to inquire about this variable, the data type of the returned window descriptor depends on the window system as shown below:

X11, OpenWindows
: Data type is Xgl_X_window *, a pointer to a structure containing X11 window or XView canvas information.

SEE ALSO

xgl_object_get(3)
xgl_object_create(3)
XGL_WIN_RAS_TYPE(3)

NAME

XGL_WIN_RAS_MBUF_DRAW – current MBX read / write buffer

SYNOPSIS

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

Xgl_win_ras          ras;

xgl_object_set(win_ras, XGL_WIN_RAS_MBUF_DRAW, Xgl_sgn32, 0 );
xgl_object_get(win_ras, XGL_WIN_RAS_MBUF_DRAW, Xgl_sgn32 *);
```

DESCRIPTION

This attribute specifies the buffer to read and write pixels into and is applicable for all primitives. The buffer index specified as the attribute value should have a one-to-one correspondence to those of the multibuffer array returned from the MBX API XmbufCreateBuffers(). The index can also be -1 to denote rendering to the window.

The application should explicitly set the buffer to draw to by using the **XGL_WIN_RAS_MBUF_DRAW**(3) attribute, and use the MBX API XmbufDisplayBuffers() to display a particular buffer. Note that unlike double buffering, **xgl_context_new_frame**(3) does not switch the buffers.

The default value is -1.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_context_new_frame(3)
XGL_WIN_RAS_BUF_DRAW(3)
XGL_WIN_RAS_MULTIBUFFER(3)

NOTE

While mixing XGL and X calls, the application should call xgl_context_flush(ctx, XGL_FLUSH_SYNCHRONIZE) after the XGL call and before making any X function call. Similarly, the application should call XSync() after the X call and before making any XGL function call.

For more information about MBX, please refer to the *OpenWindows Server Programmer's Guide*.

NAME

XGL_WIN_RAS_MULTIBUFFER – enables MBX for window raster

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
void xgl_object_set(win_ras, XGL_WIN_RAS_MULTIBUFFER, Xgl_boolean, 0 );
void xgl_object_get(win_ras, XGL_WIN_RAS_MULTIBUFFER, Xgl_boolean *);
```

DESCRIPTION

This attribute is set by the application to request Multi-Buffering extension to X(MBX) for a window raster. This attribute should be set to TRUE only after creating multiple buffers for the window through XmbufCreateBuffers(); XmbufCreateBuffers() is a MBX API function.

It is the application's responsibility to inform XGL of any subsequent multibuffer destruction and creation on the same window raster through the API setting of **XGL_WIN_RAS_MULTIBUFFER**(3)**.**

The application should set this attribute to FALSE before destroying the MBX multibuffers through the MBX extension.

The default value is FALSE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_WIN_RAS_MBUF_DRAW(3)

NOTE

While mixing XGL and X calls, the application should call xgl_context_flush(ctx, XGL_FLUSH_SYNCHRONIZE) after the XGL call and before making any X function call. Similarly, the application should call XSync() after the X call and before making any XGL function call.

For more information about MBX, please refer to the *OpenWindows Server Programmer's Guide*.

NAME

XGL_WIN_RAS_PIXEL_MAPPING – pixel mapping from user index colors to window system index colors

SYNOPSIS

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

Xgl_win_ras          ras;

xgl_object_set(ras, XGL_WIN_RAS_PIXEL_MAPPING, Xgl_usgn32 *, 0);
xgl_object_get(ras, XGL_WIN_RAS_PIXEL_MAPPING, Xgl_usgn32 *);
```

DESCRIPTION

This attribute is the *pixel mapping array*, which maps the application's color indices to hardware color indices. This additional mapping (level of indirection) is required to permit an XGL application to use color maps created by the OpenWindows and X window systems.

This attribute should only be used if the visual type of the raster device's window is PseudoColor. This attribute has no meaning on Xlib windows created from TrueColor or DirectColor visuals.

This pixel mapping is called *contiguous* if the mapping of indices from the application's color table to the hardware color table is consecutive and monotonically increasing. The mapping may begin at some offset into the hardware color table, but there can be no ''holes'' in the mapping from application to hardware. For example, if the application creates a color table array containing four RGB values in locations 0-3 of the array, the pixel mapping is contiguous if these four indices are mapped to four consecutive indices within the hardware color table (such as 5-8). This example produces the following contiguous mapping:

application index 0 → hardware index 5
application index 1 → hardware index 6
application index 2 → hardware index 7
application index 3 → hardware index 8

The offset in this case is 5.

A contiguous mapping is guaranteed by XGL if an application creates its own color table array (see **XGL_CMAP_COLOR_TABLE**(3)) and attaches it to a Color Map object. However, if an application creates an X color map (an X color map is analogous to an XGL color table) through Xlib function calls, the pixel mapping may or may not be contiguous. In this case, the application is responsible for ensuring contiguity of the pixel mapping if the application requires it.

If an application sets the **XGL_CMAP_NAME**(3) attribute in the Color Map object associated with the Window Raster, then XGL_WIN_RAS_PIXEL_MAPPING must be set immediately afterward. This guarantees that the application's indexed colors are properly mapped to the hardware indices. To do this, when creating an OpenWindows (X) color map, the application should save the pixel array returned by the window system and hand it to XGL through **XGL_WIN_RAS_PIXEL_MAPPING**(3)**.** (For more information about how to do this, see XGL_CMAP_NAME.)

When XGL is managing the X Colormap; in other words, when the application has not set **XGL_CMAP_NAME**(3)**,** some color map attributes can potentially change the pixel mapping array. These color map attributes are as follows:

XGL_CMAP_COLOR_CUBE_SIZE(3)
XGL_CMAP_COLOR_TABLE(3)

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
xgl_object_create(3)

XGL_CMAP_COLOR_CUBE_SIZE(3)
XGL_CMAP_COLOR_TABLE(3)
XGL_CMAP_NAME(3)

NOTES

Non-contiguous pixel mapping, when used with XGL primitives requiring interpolated colors, produces undefined results on various graphics hardware and may degrade rendering performance. To ensure correct results for interpolated color rendering, the application should use a contiguous pixel mapping. (See discussion above for a definition of contiguous pixel mapping.)

If the application sets the XGL_CMAP_NAME attribute, the XGL_WIN_RAS_PIXEL_MAPPING should be set immediately afterward. This guarantees that the application's indexed colors are properly mapped to the hardware indices.

NAME

XGL_WIN_RAS_POSITION – position of Window Raster object

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_get(ras, XGL_WIN_RAS_POSITION, Xgl_pt_i2d * );
```

DESCRIPTION

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

SEE ALSO

xgl_object_get(3)
xgl_object_create(3)

NAME

XGL_WIN_RAS_STEREO_MODE – specifies whether the window being rendered into is a stereo window or a mono window

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras          ras;
xgl_object_set(ras, XGL_WIN_RAS_STEREO_MODE, Xgl_stereo_mode, 0);
xgl_object_get(ras, XGL_WIN_RAS_STEREO_MODE, Xgl_stereo_mode *);
```

DESCRIPTION

This attribute specifies whether the window being rendered into is a stereo window or a mono window. XGL accepts the following enumerated values for *Xgl_stereo_mode*:

XGL_STEREO_NONE
: This value specifies that the window is a monocular (non-stereo) window. In this mode, if the frame buffer is in stereo mode, all graphics will be rendered into both the right and left eye buffers. This value must be specified when the frame buffer is not in stereo mode.

XGL_STEREO_LEFT
: This value specifies that the window is a stereo window, and that the image for the left eye buffer is being rendered. This is only legal if the frame buffer is in stereo mode.

XGL_STEREO_RIGHT
: This value specifies that the window is a stereo window, and that the image for the right eye buffer is being rendered. This is only legal if the frame buffer is in stereo mode.

Note that undefined results may occur if stereo is used when rendering through a 2D Context.

The default value is XGL_STEREO_NONE.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
xgl_inquire(3)

NAME

XGL_WIN_RAS_TYPE – the type of Window Raster Device

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_win_ras ras;
xgl_object_get(ras, XGL_WIN_RAS_TYPE, Xgl_window_type * );
```

DESCRIPTION

This attribute gives the *type* of the Window Raster Device. It is a read-only attribute describing the window system controlling the Window Raster. The value of this attribute is the same as the value that the application passed to XGL with the operator **xgl_object_create**(3)**.**

XGL will return the following value of *Xgl_window_type*:

XGL_WIN_X
: The Window Raster is an X11 window or XView canvas. Note that the window type can have a protocol encoded in it. (For a discussion of how to specify a protocol, see **XGL_WIN_RAS**(3)**.**) The masks XGL_WIN_MASK and XGL_WIN_PROTO_MASK can be used to get the window system type or protocol, respectively, from the returned value.

This attribute has no default value.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
xgl_object_create(3)
XGL_WIN_RAS_DESCRIPTOR(3)

NAME

intro – introduces the XGL graphics library

DESCRIPTION

These Reference Manual pages describe the syntax for using the operators (functions) and attributes in the XGL Graphics Library. Special sections immediately following this introduction describe the XGL enumerated types, XGL macro values, XGL point list structures, and XGL data structures. The next two sections describe, in alphabetical order, the XGL attributes and operators.

The XGL programming paradigm is built around *objects*, which are instances of abstract data types. The application can create an object and apply operators to it. The application cannot, however, extend the definition of an object or modify the behavior of operators. Therefore, XGL is not an object-oriented programming system, although its design does incorporate some object-oriented concepts.

The XGL operators are standard C language library calls and are documented as such. Note that some operators can take a variable number of parameters. For example, the operator xgl_object_set is described this way:

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

void  xgl_object_set (
            Xgl_object          obj,
            <attribute-list>    attributes );
```

The first parameter is of the type **Xgl_object**, which is a handle to the XGL object data. The next argument, whose type is shown as <*attribute-list*>, is a *variable length* list of attribute-value pairs. A zero (ASCII NULL character) is used to mark the end of the list.

Each XGL object has a set of attributes associated with it. An attribute is a named field inside the object. The application can set an attribute's value, or it can retrieve its value, or both. The documentation for the XGL attributes specifies the C data type needed for setting the attribute and the C data type needed for getting (retrieving) its value. The following example describes how to set and get the attribute **XGL_CTX_LINE_WIDTH_SCALE_FACTOR**(3)**,** which controls the width of lines in the Context object.

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

Xgl_ctx         ctx;

xgl_object_set(ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR, float, 0);
xgl_object_get(ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR, float *);
```

This notation indicates that to set the attribute value, the application passes a *float*, and to get the value, the application passes a *pointer* to a *float*, which is overwritten with a new value.

Naming Scheme

The XGL library defines symbols that start with *XGL*, *Xgl*, *xgl*, *XGLI*, *Xgli*, and *xgli*. The application program should not define any symbols (type names, enumerated values, cpp definitions, function names, and so on) beginning with these prefixes.

Printed Manual vs. On-line Manual

Most attributes and operators are discussed on their own separate manual page. In some cases, however, several related attributes are discussed together on a single manual page. In the printed reference manual, only the page where they are discussed is printed, but for the on-line manual, you can use **man** (1) for any of the topics discussed on that page.

For example, both **XGL_3D_CTX_SURF_BACK_AMBIENT**(3) and **XGL_3D_CTX_SURF_FRONT_-AMBIENT**(3) are discussed on the page for **XGL_3D_CTX_SURF_FRONT_AMBIENT**(3)**.** In the printed reference manual, you will find only the page for **XGL_3D_CTX_SURF_FRONT_-AMBIENT**(3) printed. You can look up **XGL_3D_CTX_SURF_BACK_AMBIENT**(3) in the index to see where it is discussed. You can issue either of the following commands for the on-line version, however:

man XGL_3D_CTX_SURF_BACK_AMBIENT

man XGL_3D_CTX_SURF_FRONT_AMBIENT

You can also access the following descriptions of other aspects of XGL:

Data Structures
described in **xgl_struct**(3)

Enumerated Types
described in **xgl_enum_types**(3)

Macro Values
described in **xgl_macro_values**(3)

Point Lists
described in **xgl_pt_list**(3)

Object Types
described on their individual man pages
XGL_2D_CTX(3)
XGL_3D_CTX(3)
XGL_CGM_DEV(3)
XGL_CMAP(3)
XGL_DMAP_TEXTURE(3)
XGL_GCACHE(3)
XGL_LIGHT(3)
XGL_LPAT(3)
XGL_MARKER(3)
XGL_MEM_RAS(3)
XGL_MIPMAP_TEXTURE(3)
XGL_PCACHE(3)
XGL_SFONT(3)
XGL_STREAM(3)
XGL_SYS_STATE(3)
XGL_TMAP(3)
XGL_TRANS(3)
XGL_WIN_RAS(3)

NAME

xgl_annotation_text – renders 2D and 3D annotation text

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_annotation_text (
            Xgl_ctx          ctx,
            void             *str,
            Xgl_pt_f{2,3}d   *ref_pos,
            Xgl_pt_f{2,3}d   *ann_pos );
            /* f2d for 2D Context, f3d for 3D Context */
```

DESCRIPTION

This Context operator renders the text indexed by the characters in *str* using the Context attributes within *ctx*.

Annotation text primitives differ from text primitives in that the plane upon which characters are generated is always parallel to the display surface. Annotation text primitives are *planar* primitives. The depth at which annotation text is located is dependent on the *z* coordinate of the *annotation point*, defined below.

Depending on the character-encoding scheme defined by the Context attribute **XGL_CTX_STEXT_CHAR_ENCODING**(3)**,** the *str* parameter represents a *NULL-terminated* C-style string of characters, or a pointer on a structure of type *Xgl_mono_text_list* defined in **xgl_struct**(3)**.** *ref_pos* is the reference point for the rendered string. This point is transformed through the XGL transformation pipeline. An *annotation point* is derived from the reference point by adding the values stored in *ann_pos* to the transformed reference point. *ann_pos* values are specified in virtual device coordinates (VDC).

If the string-encoding scheme used is of type XGL_SINGLE_STR, at least one font can be selected to render the characters in ISO or MBY character-encoding schemes:

XGL_CTX_SFONT_0(3)

When the character-encoding scheme used is of type XGL_CHAR_MBY, the fonts used to render the string are selected depending on the character code adding the following Context attributes:

XGL_CTX_SFONT_1(3)
XGL_CTX_SFONT_2(3)
XGL_CTX_SFONT_3(3)

If the string-encoding scheme used is of type XGL_MULTI_STR, the structure passed in parameter contains the XGL font Object used to render the characters. Both character-encodings ISO or MBY can be used.

The application can control the geometrical characteristics of the text using the following attributes:

XGL_CTX_ATEXT_CHAR_HEIGHT(3)
XGL_CTX_STEXT_CHAR_SPACING(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)

The text orientation is controlled by changing the components of the **XGL_CTX_ATEXT_CHAR_UP_VECTOR**(3) attribute. An inclination can be added to the orientation by using the **XGL_CTX_ATEXT_CHAR_SLANT_ANGLE**(3) attribute.

The text presentation, with respect to the position given by the parameter *ref_pos*, is managed by the following attributes:

XGL_CTX_ATEXT_ALIGN_HORIZ(3)
XGL_CTX_ATEXT_ALIGN_VERT(3)
XGL_CTX_ATEXT_PATH(3)

The color of the text can be modified with the **XGL_CTX_STEXT_COLOR**(3) attribute.

Annotation text is clipped differently than normal stroke text is. It will be visible if the reference point is within the clipping region or volume in 3D. Otherwise, the complete text is rejected. If the reference point implies a visible text, the text is further clipped to the limits of the clipping region or volume.

An additional attribute, **XGL_CTX_ATEXT_STYLE**(3)**,** controls whether the reference point is visually connected to the annotation point. A visible connection is represented by a single line, using the multipolyline current non-geometric attributes, between the reference point and the annotation point. For a list of non-geometric attributes, see **xgl_multipolyline**(3)**.**

XGL annotation text is rendered as a set of polylines. The width is fixed and there cannot be any pattern applied to the rendering of the text. The ROP and plane mask apply, as for any other rendering primitive, and can be modified with the **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3) attributes, respectively.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_multipolyline(3)

XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_CTX_STEXT_CHAR_ENCODING(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)
XGL_CTX_ATEXT_CHAR_HEIGHT(3)
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE(3)
XGL_CTX_ATEXT_CHAR_UP_VECTOR(3)
XGL_CTX_ATEXT_STYLE(3)
XGL_CTX_ATEXT_ALIGN_HORIZ(3)
XGL_CTX_ATEXT_ALIGN_VERT(3)
XGL_CTX_ATEXT_PATH(3)
XGL_CTX_SFONT_0(3)
XGL_CTX_SFONT_1(3)
XGL_CTX_SFONT_2(3)
XGL_CTX_SFONT_3(3)
XGL_CTX_STEXT_CHAR_SPACING(3)
XGL_CTX_STEXT_COLOR(3)

NOTES

For XGL annotation text, the character up vector, defined by the **XGL_CTX_ATEXT_CHAR_-UP_VECTOR**(3) attribute is independent of the VDC orientation (defined by **XGL_CTX_VDC_-ORIENTATION**(3)) and should always be interpreted as if the Y axis was ALWAYS pointing up.

NAME

xgl_close – ends an XGL session

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_close (
            Xgl_sys_state    system_state );
```

DESCRIPTION

This operator ends an XGL session. The application invokes the operator by passing to it the handle of the XGL System State object, created by a previous call to **xgl_open**(3)**,** which began the XGL session. When xgl_close is executed, the System State object and all associated resources that have been allocated during the current XGL session are destroyed, and then XGL is terminated.

An XGL session can have only one System State object present at a given time. It is possible to create and delete a System State object several times within a single application by using multiple, paired xgl_open and xgl_close calls. Other XGL operators must be invoked between a pair of xgl_open and xgl_close calls to ensure correct management of XGL library resources.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_destroy(3)
xgl_open(3)

NAME

xgl_context_accumulate – accumulates from the draw buffer of the raster attached to the context to the destination buffer

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_accumulate (
        Xgl_ctx             ctx,
        Xgl_bounds_i2d*     rectangle,
        Xgl_pt_f2d*         pos,
        float               src_wt,
        float               accum_wt,
        Xgl_buffer_sel      buf );
```

DESCRIPTION

This operator accumulates from the draw buffer of the raster attached to the Context *ctx* to the destination buffer specified by the attribute **XGL_3D_CTX_ACCUM_OP_DEST**(3)**.** The accumulated area is copied to the buffer specified by *buf*. The area in the draw buffer of the raster that is to be accumulated is specified by the rectangular region *rectangle*. The accumulation is done starting from position *pos* in the destination buffer. The rectangular region that is used as source is not clipped to DC viewport. It is clipped by the maximum dimension of the draw buffer as defined by **XGL_DEV_MAXIMUM_COORDINATES**(3)**.** If the rectangle is a NULL pointer, the maximum area from the draw buffer of the raster is used as source. If *position* is NULL, the top and left corner of the destination buffer is used.

After accumulation, the accumulated area is copied to the buffer *buf*. The rectangle starting from position *pos* and width and height up to a maximum of width and height of the *rectangle* is the source area for copy. The destination area starts at (xmin,ymin) of the rectangle up to a maximum width and height specified by *rectangle*.

The accumulation calculation occurs as follows:

DEST_BUFFER = SRC_BUFFER * SRC_WEIGHT + DEST_BUFFER * ACCUM_WEIGHT

Parameters

ctx Specifies the context to which 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**(3)**)** of the *ctx* is used as the destination buffer in the accumulation operation.

rectangle
Specifies the source area of the draw buffer of the raster attached to the *ctx* with *xmin, xmax, ymin, ymax*.

pos Specifies the position in the destination buffer (specified by the attribute **XGL_3D_CTX_ACCUM_OP_DEST**(3)**)** 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, all are correspondingly clipped.

src_wt The weight to be used as SRC_WEIGHT in the accumulation calculation given above.

accum_wt

The weight to be used as ACCUM_WEIGHT in the accumulation calculation given above.

buf Specifies the buffer to copy the accumulated image to. It takes on values:

XGL_BUFFER_SEL_NONE
: No copy back to the image buffers once it is done.

XGL_BUFFER_SEL_DISPLAY
: The contents of the destination buffer is always copied to the display buffer.

A value of 0 to N-1
: N is the number of buffers allocated in the raster attached to the Context *ctx*.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_set(3)
xgl_object_get(3)
XGL_3D_CTX_ACCUM_OP_DEST(3)
XGL_DEV_MAXIMUM_COORDINATES(3)

NAME

xgl_context_check_bbox – calculates the geometry status of a bounding box

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_check_bbox (
        Xgl_ctx             ctx,
        Xgl_primitive_type  prim_type,
        Xgl_bbox            *bbox,
        Xgl_geom_status     *geom_status );
```

DESCRIPTION

Purpose

This operator calculates the geometry status of the bounding box *bbox*, using the transformations in the Context *ctx*. A geometry status can be used by XGL to optimize the clipping operations when rendering primitives.

Parameters

ctx The XGL Context containing the graphics state used when processing xgl_context_check_bbox.

prim_type
The bounding box primitive type.

bbox A pointer to a bounding box which can be one of the following structures (see **xgl_struct**(3) for a complete description of the possible data structures).

Xgl_bbox_i2d
Xgl_bbox_f2d
Xgl_bbox_d2d
Xgl_bbox_f3d
Xgl_bbox_d3d

geom_status
A pointer to *Xgl_geom_status* in which the calculated geometry status is stored. The calculated geometry status is a combination of the following flags:

XGL_GEOM_STATUS_VIEW_ACCEPT
This flag is set if the bounding box is completely inside the view clip planes - that is, not clipped.

XGL_GEOM_STATUS_VIEW_REJECT
This flag is set if the bounding box is completely outside the view clip planes - that is, totally clipped.

XGL_GEOM_STATUS_VIEW_SMALL
This flag is set if the bounding box is completely inside the view clip planes, and the area occupied by the bounding box after transformation to device coordinates is less than or equal to the context threshold value for the primitive type prim_type. The XGL_GEOM_STATUS_VIEW_ACCEPT flag is set when this flag is set. See **XGL_CTX_THRESHOLD**(3) for more information about the usage of the context threshold values.

XGL_GEOM_STATUS_MODEL_ACCEPT
: This flag is set if model clip planes exist and the bounding box is completely inside the model clip planes - that is, not clipped. If no model clip planes exist, this flag is always set.

XGL_GEOM_STATUS_MODEL_REJECT
: This flag is set if model clip planes exist and the bounding box is completely outside the model clip planes - that is, totally clipped.

If none of the above flags are set (e.g. the geometry status is zero), the bounding box will be partially clipped by the view clip planes and the model clip planes.

Most XGL drawing primitives accept bounding box information. For these primitives, the application can either pass the bounding box or its geometry status to XGL. If the application wants to pass the geometry status to XGL, it can first call xgl_context_check_bbox to find out the geometry status of the bounding box and then pass the status to XGL using the *Xgl_bbox_status* structure. The following 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 geometry status calculated by xgl_context_check_bbox only applies to XGL non-gcache primitives such as **xgl_multimarker**(3)**.** For XGL gcache operators, such as **xgl_gcache_multimarker**(3)**,** applications should use one of the following bounding box structures, instead of *Xgl_bbox_status*, to pass bounding box information to XGL:

Xgl_bbox_i2d
Xgl_bbox_f2d
Xgl_bbox_d2d
Xgl_bbox_f3d
Xgl_bbox_d3d

For polygonal primitives, if the application knows that the primitives contained in the bounding box are all front facing (or back facing), it can give this hint to XGL 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 the application passes the geometry status to XGL non-gcache drawing operators using the Xgl_bbox_status structure.

It is the application's responsibility to ensure that all geometry statuses it passes to XGL operators are correct. Thus, if any of the XGL context attributes affecting the MC to DC transformation and clipping (or affecting the front/back facing status) is modified, the geometry status needs to be re-calculated.

If the application does not want to take advantage of the optimization provided by the geometry status feature, it can always set the geometry status to zero.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_struct(3)
XGL_CTX_THRESHOLD(3)

NAME

xgl_context_clear_accumulation – clears the buffer specified by the attribute XGL_3D_CTX_-ACCUM_OP_DEST to a certain value

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_clear_accumulation (
            Xgl_ctx       ctx,
            Xgl_color*    value );
```

DESCRIPTION

This operator clears the buffer specified by the attribute **XGL_3D_CTX_ACCUM_OP_DEST**(3) to a value *value*. The accumulation is done using **xgl_context_accumulate**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_accumulate(3)

XGL_3D_CTX_ACCUM_OP_DEST(3)

NAME

xgl_context_copy_buffer – copies a block of pixels from one buffer to another

SYNOPSIS

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

void  xgl_context_copy_buffer (
        Xgl_ctx          dest_ctx,
        Xgl_bounds_i2d   *rectangle,
        Xgl_pt_i2d       *position,
        Xgl_ras          src_ras );
```

DESCRIPTION

This operator 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.

For the simple case of copying from a memory raster with 1 buffer to a non-multi buffered window raster, the pixels are copied from the draw buffer of a source raster to a draw buffer of a raster associated with a destination Context.

For more complicated cases with multiple buffers, the application program must set attributes to specify the buffers to be used as the source buffer and the destination buffer. The buffer to be used as source in the source Raster *src_ras* is specified by the source raster's attribute **XGL_RAS_SOURCE_BUFFER**(3)**.** The buffer(s) to be used as destination in the Raster associated to the destination Context *dest_ctx* is determined by the destination context attribute **XGL_CTX_RENDER_BUFFER**(3)**.**

The area that is copied is specified by a rectangular region, *rectangle*, in the source buffer. It is copied into the destination buffer starting at the specified *position.* The rectangular region that is copied is not clipped to DC viewport. It is clipped by the maximum dimension of the draw buffer as defined by **XGL_DEV_MAXIMUM_COORDINATES**(3)**.** 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. That is, copying can be between image buffers or between Z buffers. The attribute **XGL_RAS_SOURCE_BUFFER**(3) of the source raster *src_ras* takes on values:

XGL_BUFFER_SEL_Z
: For this attribute value, if the destination context attribute value for **XGL_CTX_-RENDER_BUFFER**(3) has XGL_RENDER_Z_BUFFER as one of the render buffers, then the destination buffer is the Z buffer of the raster associated with the destination Context *dest_ctx*. Thus, copying takes place from the source raster's Z buffer to the destination raster's Z buffer.

XGL_BUFFER_SEL_DRAW
: For this attribute value, if the destination context attribute value for **XGL_CTX_-RENDER_BUFFER**(3) has XGL_RENDER_DRAW_BUFFER and / or XGL_RENDER_DISPLAY_-BUFFER, then copying takes place from the draw buffer to the image buffer(s) specified by attribute **XGL_CTX_RENDER_BUFFER**(3)**.**

A Value between 0 to N-1
: If the destination context attribute value for **XGL_CTX_RENDER_BUFFER**(3) has XGL_-

RENDER_DRAW_BUFFER and/or XGL_RENDER_DISPLAY_BUFFER, then copying takes place from the buffer specified by attribute **XGL_RAS_SOURCE_BUFFER**(3) to the image buffer(s) specified by attribute **XGL_CTX_RENDER_BUFFER**(3)**.**

Copy from a Z Buffer to another Z Buffer

When copying takes place between Z buffers, a plain copy is done. Specifically, the Z mask (specified by **XGL_3D_CTX_Z_BUFFER_WRITE_MASK**(3)) and the Z comparison function (specified by **XGL_3D_CTX_Z_BUFFER_COMP_METHOD**(3)) are ignored; the destination Z buffer is set to the value contained in the source Z buffer, given the precision limitations described below. In addition, the attribute **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER in the Context *dest_ctx*.

The Z buffers are assumed to have a depth up to a maximum of 24 bits or $(2^{24} - 1)$.

(a) Copying is possible between memory Raster's Z buffers. (All memory Rasters have a Z buffer depth of 24).

(b) Copying is possible from a window Raster's Z buffer to another window Raster's Z buffer only if they are from the same physical frame buffer (which implicitly means they are of the same depth).

(c) When copying from a window raster to a memory raster, if the window raster's Z depth is less than $2^{24} - 1$, then the window raster's Z buffers are copied into the least significant bits of the memory Raster's Z buffer.

(d) When copying from a memory raster to a window raster, if the window raster's Z depth is less than $2^{24} - 1$, then the least significant bits of memory Raster's Z buffer are copied into the window Raster's Z buffer.

Copy from an Image Buffer to another Image Buffer

When copying takes place between image buffers, then the ROP mode, plane mask, and the fill rule defined in the destination Context with **XGL_CTX_ROP**(3)**, XGL_CTX_PLANE_-MASK**(3) and **XGL_CTX_RASTER_FILL_STYLE**(3) attributes are enforced during this operation.

When copying to and from a memory raster, be sure that the memory raster has a Color Map which corresponds to its color type.

If the source raster of a copy operation is from a partially obscured window, the results of copying the obscured part is device dependent. This is true even if backing store is enabled. The application is responsible for repairing these areas if necessary.

Unobscured windows, on the other hand, will be properly clipped, even if the source rectangle extends beyond the window boundaries.

The following table specifies the supported configurations for copying between Raster Devices' image buffers.

From Source → To Destination

		Source						
Destination		M1	M4i	M8i	Mrgb	W4i	W8i	Wrgb
	M1	X	X	X	X	X	X	X
	M4i	Y	Y	X	X	Y	X	X
	M8i	Y	Y	Y	X	Y	Y	X
	Mrgb	Y	X	Y	Y	X	Y	Y
	W4i	Y	Y	X	X	Y*	X	X
	W8i	Y	Y	Y	X	X	Y*	X
	Wrgb	Y	X	Y	Y	X	X	Y*

Where: M1 = 1 bit deep memory raster
M4i = 4 bit deep memory raster whose color type is XGL_COLOR_INDEX
M8i = 8 bit deep memory raster whose color type is XGL_COLOR_INDEX
Mrgb = a memory raster whose color type is XGL_COLOR_RGB
W4i = 4 bit deep window raster whose color type is XGL_COLOR_INDEX
W8i = 8 bit deep window raster whose color type is XGL_COLOR_INDEX
Wrgb = a window raster whose color type is XGL_COLOR_RGB

Key: X : not supported
Y : supported
Y* : only supported if the src and dest window raster are the same.

When the source raster is a 1 bit memory raster, it is used as a stencil. When a pixel in the source raster is 1, the **XGL_CTX_SURF_FRONT_COLOR**(3) is written to the destination raster. When a source pixel is 0, the **XGL_CTX_BACKGROUND_COLOR**(3) is written.

The color type refers to **XGL_DEV_COLOR_TYPE**(3) of the corresponding devices.

SEE ALSO

XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_ROP(3)
XGL_CTX_SURF_FRONT_COLOR(3)
XGL_DEV_COLOR_TYPE(3)
XGL_DEV_MAXIMUM_COORDINATES(3)
XGL_RAS_SOURCE_BUFFER(3)

NOTES

Copying on a Z buffer is supported on a 3D Context only. During this operation, **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER. The Z buffer is created when the **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER for the first time.

XGL_CTX_RENDER_BUFFER(3) is a logical OR of XGL_RENDER_DRAW_BUFFER, XGL_RENDER_DISPLAY_BUFFER, and XGL_RENDER_Z_BUFFER. In order for the operation to be performed on a Z-buffer, the application should only use XGL_RENDER_Z_BUFFER. This results in operation on the selected image buffer and the ignoring of the Z-buffer.

NAME

xgl_context_display_gcache – displays a Gcache on a Context

SYNOPSIS

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

Xgl_cache_display  xgl_context_display_gcache (
                        Xgl_ctx         ctx,
                        Xgl_gcache      gcache,
                        Xgl_boolean     test,
                        Xgl_boolean     display );
```

DESCRIPTION

Purpose

This operator performs two tasks. It can display the Gcache *gcache* on the Device associated with the Context *ctx*. It can also compare the state (that is, the attribute values) of the Gcache with the current state in the Context and return the result of that comparison.

Parameters

test Controls whether the state in Gcache is validated, by comparing the saved attribute values in the Gcache with values in the Context. If *test* is TRUE, the comparison is done, and the return value is one of XGL_CACHE_DISPLAY_OK or XGL_CACHE_ATTR_STATE_DIFFERENT. If *test* is FALSE, the return value is XGL_CACHE_NOT_CHECKED.

display Controls whether the Gcache is rendered on the Device associated with the Context. If *display* is FALSE, the Gcache is not rendered. If it is TRUE and *test* is FALSE, the Gcache is rendered. If *display* is TRUE and *test* is TRUE, Gcache is rendered only if the validity test is TRUE.

The Gcache attribute XGL_GCACHE_BYPASS_MODEL_CLIP is used when the following conditions exist:

When *test* is FALSE, *display* is TRUE, and the geometry in the Gcache was Model Clipped when it was placed in the Gcache. In such a case, if the attribute is TRUE, cached geometry is not processed by the current Model Clipping settings. If it is FALSE, the cached geometry is processed by the current Model Clipping settings.

RETURN VALUE

Returns the status of the validity test:

XGL_CACHE_DISPLAY_OK
: The state in Gcache matches the state in the Context.

XGL_CACHE_ATTR_STATE_DIFFERENT
: The state in Gcache does not match the state in the Context.

XGL_CACHE_NOT_CHECKED
: The state in Gcache is not checked against the state in the Context.

SEE ALSO

xgl_object_create(3)

XGL_GCACHE_BYPASS_MODEL_CLIP(3)

NAME

xgl_context_flush – causes pending or asynchronous processing to complete

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_flush (
            Xgl_ctx             ctx,
            Xgl_flush_action    flush_action );
```

DESCRIPTION

Purpose

This operator allows some rendering devices to render by directly processing the application's data. While the devices are processing that data, it is possible for the rendering routine to return to the application's code to gain a degree of parallelism, thus improving the overall performance (see **XGL_CTX_GEOM_DATA_IS_VOLATILE**(3)**).**

Further, some devices can buffer XGL commands and require explicit flushing to post their buffers. A return from any XGL operator call does not guarantee that all actions have been completed (see **XGL_CTX_DEFERRAL_MODE**(3)**).**

The **xgl_context_flush**(3) operator is used to flush one or both of XGL's use of application's data and XGL internal buffers. In the case of XGL's buffers, the application might want XGL to block until all posted buffers have been processed by the rendering device.

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_context_flush**(3)**.**

flush_action

The types of flushing to be performed based on any OR'ed combination of the following values:

XGL_FLUSH_GEOM_DATA

Ensures that this context is not using any application data. If **XGL_CTX_-GEOM_DATA_IS_VOLATILE**(3) is set to TRUE, XGL_FLUSH_GEOM_DATA does nothing.

XGL_FLUSH_BUFFERS

Causes any pending graphics primitives for this context to be sent to the rendering device. If **XGL_CTX_DEFERRAL_MODE**(3) is set to XGL_DEFER_-ASAP, XGL_FLUSH_BUFFERS does nothing.

XGL_FLUSH_SYNCHRONIZE

Causes XGL to block until the rendering device has finished processing all of the graphics primitives for this context. If this flag is not set, XGL may return control to the application before the rendering device has finished processing the geometry data, allowing XGL and the application to run asynchronously. Specifying XGL_FLUSH_SYNCHRONIZE also implies the setting of XGL_FLUSH_GEOM_DATA and XGL_FLUSH_BUFFERS.

This operator is device-dependent in that a given action might be irrelevant on Devices that do not support the corresponding parallelism. In this case, the given action is a NO-OP. Consult your accelerator guide for details.

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_CTX_DEFERRAL_MODE(3)
XGL_CTX_GEOM_DATA_IS_VOLATILE(3)

NOTES

When mixing XGL with Xlib function calls on the same Window, and if the rendering order is important, **xgl_context_flush**(3) (ctx, XGL_FLUSH_SYNCHRONIZE) should be called at the end of a block of XGL calls that is followed by Xlib commands. Inversely, the Xlib calls should be flushed with XSync before further XGL commands be issued.

NAME

xgl_context_get_pixel – gets the color value of a specified pixel

SYNOPSIS

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

Xgl_boolean  xgl_context_get_pixel (
                    Xgl_ctx       ctx,
                    Xgl_pt_i2d    *position,
                    Xgl_color     *value );
```

DESCRIPTION

This operator gets the color value of a pixel located at *position* in the specified buffer of the Raster associated with the Context *ctx*. The buffer to be read from is specified by the attribute **XGL_RAS_SOURCE_BUFFER**(3) of the raster attached to the context. The retrieved color is stored in the location pointed to by the parameter *value*.

When reading from an image buffer, depending on the color type of the Device attached to the Context, as specified by **XGL_DEV_COLOR_TYPE**(3)**,** the color returned is either an index or a true RGB color (that is, of type XGL_COLOR_INDEX or XGL_COLOR_RGB).

When reading from a Z buffer, if the depth of the Z buffer is less than 32 bits, then the most significant bits are zeroed out in the returned values.

If the pixel position occurs at an obscured location (as may happen in the case of overlapping windows), the value of the hardware pixel cannot be guaranteed and the function returns TRUE. Otherwise it returns FALSE.

RETURN VALUE

Returns a boolean that indicates whether the pixel is obscured by an overlapping window.

SEE ALSO

XGL_3D_CTX_HLHSR_MODE(3)
XGL_DEV_COLOR_TYPE(3)
XGL_RAS_SOURCE_BUFFER(3)

NOTES

When reading from an image buffer, 4-bit and 8-bit Indexed (**XGL_DEV_COLOR_TYPE**(3) is XGL_COLOR_INDEX) Memory Rasters store their pixel values as application-defined color values (that is, the index stored in the Memory Raster is in the application-defined color space).

This operation on a Z buffer is supported on a 3D Context only. During this operation, **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER. The Z buffer is created when **XGL_3D_CTX_HLHSR_MODE**(3) is set to XGL_HLHSR_Z_BUFFER for the first time.

NAME

xgl_context_new_frame – clears the Device Coordinate (DC) viewport and possibly the *z*-buffer of the specified Context

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_new_frame (
            Xgl_ctx    ctx );
```

DESCRIPTION

This operator is used to clear the Device associated with the current Context *ctx*, to the background color specified by the attribute **XGL_CTX_BACKGROUND_COLOR**(3)**.**

The actions taken by the call to **xgl_context_new_frame**(3) are controlled by the attribute **XGL_CTX_NEW_FRAME_ACTION**(3)**.** This attribute is the logical OR of possible new frame actions defined by the enumerated data type *Xgl_ctx_new_frame_action*. The Device is cleared only if one of the actions specified by **XGL_CTX_NEW_FRAME_ACTION**(3) is XGL_-CTX_NEW_FRAME_CLEAR. In this case, the clearing operation depends on the value of the attribute **XGL_CTX_VDC_MAP**(3)**.** When XGL_CTX_VDC_MAP is XGL_VDC_MAP_ALL, XGL_VDC_-MAP_ASPECT, or XGL_VDC_MAP_DEVICE, xgl_context_new_frame clears the entire drawing area associated with the Device. When XGL_CTX_VDC_MAP is XGL_VDC_MAP_OFF, it clears only the DC viewport (see **XGL_CTX_DC_VIEWPORT**(3)) portion of the Device.

The plane mask enable bits defined by the attribute **XGL_CTX_PLANE_MASK**(3) are used to determine which pixel planes on the Device are cleared. The ROP mode, as specified by the attribute **XGL_CTX_ROP**(3)**,** is not used during the clear operation.

When using a 3D Context, this operator can also be used to clear the *z*-buffer, to switch buffers in a multiple buffer configuration, or to synchronize to the Device's vertical retrace. When clearing the *z*-buffer, all *z*-values within the buffer are reset to the value given by the attribute **XGL_3D_CTX_HLHSR_DATA**(3)**.** (For more information, see **XGL_3D_-CTX_HLHSR_DATA**(3) and **XGL_3D_CTX_HLHSR_MODE**(3)**.**)

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_3D_CTX_HLHSR_DATA(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_CTX_BACKGROUND_COLOR(3)
XGL_CTX_DC_VIEWPORT(3)
XGL_CTX_NEW_FRAME_ACTION(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_VDC_MAP(3)

NAME

xgl_context_pop – restores the most recently pushed Context state from XGL's internal stack to the current Context

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_pop (
            Xgl_ctx    ctx );
```

DESCRIPTION

This operator restores the most recent Context state, from Context *ctx*, pushed onto XGL's graphics context stack. All of the Context attributes pushed onto the stack by the most recent call to the xgl_context_push operator are restored in the Context.

For a list of pushable attributes, see **xgl_context_push**(3)**.**

Special cases

If the application only wants to pop (and later push) the model transforms (**XGL_CTX_-LOCAL_MODEL_TRANS**(3) or **XGL_CTX_GLOBAL_MODEL_TRANS**(3)), the Context operator **xgl_context_update_model_trans**(3) should be used instead of **xgl_context_pop**(3) and **xgl_context_push**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_push(3)
xgl_context_update_model_trans(3)

NOTES

For all the attributes of a context that reference other XGL objects (such as XGL_CTX_LINE_-PATTERN, for example), popping the attribute is equivalent to popping the object handle. There is one exception to this rule — when the attribute represents a Transform object. In such a case, the contents of the object are popped, not the object handle.

NAME

xgl_context_push – saves the current state of the Context on a stack

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_context_push (
            Xgl_ctx          ctx,
            Xgl_attribute    attr_list[] );
```

DESCRIPTION

Purpose

This operator saves the current state of the Context onto XGL's graphics context stack. Each Context has its own independent stack.

Parameters

attr_list A pointer to a list of attributes to be pushed on the stack. The end of the list is indicated by a NULL value. If the attribute list is NULL, all pushable attributes associated with the Context are saved.

The following is a list of all the pushable attributes. Some attributes listed here may not be present or implemented in the current version of XGL.

XGL_3D_CTX_BLEND_DRAW_MODE
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER
XGL_3D_CTX_DEPTH_CUE_COLOR
XGL_3D_CTX_DEPTH_CUE_INTERP
XGL_3D_CTX_DEPTH_CUE_MODE
XGL_3D_CTX_DEPTH_CUE_REF_PLANES
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS
XGL_3D_CTX_HLHSR_DATA
XGL_3D_CTX_HLHSR_MODE
XGL_3D_CTX_JITTER_OFFSET
XGL_3D_CTX_LIGHT_SWITCHES
XGL_3D_CTX_LINE_COLOR_INTERP
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM
XGL_3D_CTX_MODEL_CLIP_PLANES
XGL_3D_CTX_SURF_BACK_AMBIENT
XGL_3D_CTX_SURF_BACK_COLOR
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR
XGL_3D_CTX_SURF_BACK_DIFFUSE
XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES
XGL_3D_CTX_SURF_BACK_FILL_STYLE
XGL_3D_CTX_SURF_BACK_FPAT
XGL_3D_CTX_SURF_BACK_FPAT_POSITION
XGL_3D_CTX_SURF_BACK_ILLUMINATION
XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT
XGL_3D_CTX_SURF_BACK_SPECULAR
XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR
XGL_3D_CTX_SURF_BACK_SPECULAR_POWER
XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES

XGL_3D_CTX_SURF_BACK_TRANSP
XGL_3D_CTX_SURF_DC_OFFSET
XGL_3D_CTX_SURF_FACE_CULL
XGL_3D_CTX_SURF_FACE_DISTINGUISH
XGL_3D_CTX_SURF_FRONT_AMBIENT
XGL_3D_CTX_SURF_FRONT_DIFFUSE
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES
XGL_3D_CTX_SURF_FRONT_ILLUMINATION
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT
XGL_3D_CTX_SURF_FRONT_SPECULAR
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES
XGL_3D_CTX_SURF_FRONT_TRANSP
XGL_3D_CTX_SURF_GEOM_NORMAL
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP
XGL_3D_CTX_SURF_NORMAL_FLIP
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ
XGL_3D_CTX_SURF_TRANSP_METHOD
XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY
XGL_3D_CTX_Z_BUFFER_COMP_METHOD
XGL_3D_CTX_Z_BUFFER_WRITE_MASK
XGL_CTX_ARC_FILL_STYLE
XGL_CTX_ATEXT_CHAR_HEIGHT
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE
XGL_CTX_ATEXT_CHAR_UP_VECTOR
XGL_CTX_ATEXT_STYLE
XGL_CTX_ATEXT_ALIGN_HORIZ
XGL_CTX_ATEXT_ALIGN_VERT
XGL_CTX_ATEXT_PATH
XGL_CTX_BACKGROUND_COLOR
XGL_CTX_CLIP_PLANES
XGL_CTX_DC_VIEWPORT
XGL_CTX_EDGE_AA_BLEND_EQ
XGL_CTX_EDGE_AA_FILTER_SHAPE
XGL_CTX_EDGE_AA_FILTER_WIDTH
XGL_CTX_EDGE_ALT_COLOR
XGL_CTX_EDGE_CAP
XGL_CTX_EDGE_COLOR
XGL_CTX_EDGE_JOIN
XGL_CTX_EDGE_MITER_LIMIT
XGL_CTX_EDGE_PATTERN
XGL_CTX_EDGE_STYLE
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR

XGL_CTX_GLOBAL_MODEL_TRANS
XGL_CTX_LINE_AA_BLEND_EQ
XGL_CTX_LINE_AA_FILTER_SHAPE
XGL_CTX_LINE_AA_FILTER_WIDTH
XGL_CTX_LINE_ALT_COLOR
XGL_CTX_LINE_CAP
XGL_CTX_LINE_COLOR
XGL_CTX_LINE_COLOR_SELECTOR
XGL_CTX_LINE_JOIN
XGL_CTX_LINE_MITER_LIMIT
XGL_CTX_LINE_PATTERN
XGL_CTX_LINE_STYLE
XGL_CTX_LINE_WIDTH_SCALE_FACTOR
XGL_CTX_LOCAL_MODEL_TRANS
XGL_CTX_MARKER
XGL_CTX_MARKER_AA_BLEND_EQ
XGL_CTX_MARKER_AA_FILTER_SHAPE
XGL_CTX_MARKER_AA_FILTER_WIDTH
XGL_CTX_MARKER_COLOR
XGL_CTX_MARKER_COLOR_SELECTOR
XGL_CTX_MARKER_SCALE_FACTOR
XGL_CTX_MAX_TESSELLATION
XGL_CTX_MIN_TESSELLATION
XGL_CTX_NEW_FRAME_ACTION
XGL_CTX_NURBS_CURVE_APPROX
XGL_CTX_NURBS_CURVE_APPROX_VAL
XGL_CTX_NURBS_SURF_APPROX
XGL_CTX_NURBS_SURF_APPROX_VAL_U
XGL_CTX_NURBS_SURF_APPROX_VAL_V
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM
XGL_CTX_NURBS_SURF_PARAM_STYLE
XGL_CTX_PICK_ID_1
XGL_CTX_PICK_ID_2
XGL_CTX_PLANE_MASK
XGL_CTX_RASTER_FILL_STYLE
XGL_CTX_RASTER_FPAT
XGL_CTX_RASTER_FPAT_POSITION
XGL_CTX_RASTER_STIPPLE_COLOR
XGL_CTX_ROP
XGL_CTX_SFONT_0
XGL_CTX_SFONT_1
XGL_CTX_SFONT_2
XGL_CTX_SFONT_3

XGL_CTX_STEXT_AA_BLEND_EQ
XGL_CTX_STEXT_AA_FILTER_SHAPE
XGL_CTX_STEXT_AA_FILTER_WIDTH
XGL_CTX_STEXT_ALIGN_HORIZ
XGL_CTX_STEXT_ALIGN_VERT
XGL_CTX_STEXT_CHAR_ENCODING
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR
XGL_CTX_STEXT_CHAR_HEIGHT
XGL_CTX_STEXT_CHAR_SLANT_ANGLE
XGL_CTX_STEXT_CHAR_SPACING
XGL_CTX_STEXT_CHAR_UP_VECTOR
XGL_CTX_STEXT_COLOR
XGL_CTX_STEXT_PATH
XGL_CTX_STEXT_PRECISION
XGL_CTX_SURF_AA_BLEND_EQ
XGL_CTX_SURF_AA_FILTER_SHAPE
XGL_CTX_SURF_AA_FILTER_WIDTH
XGL_CTX_SURF_EDGE_FLAG
XGL_CTX_SURF_FRONT_COLOR
XGL_CTX_SURF_FRONT_COLOR_SELECTOR
XGL_CTX_SURF_FRONT_FILL_STYLE
XGL_CTX_SURF_FRONT_FPAT
XGL_CTX_SURF_FRONT_FPAT_POSITION
XGL_CTX_SURF_INTERIOR_RULE
XGL_CTX_THRESHOLD
XGL_CTX_VDC_MAP
XGL_CTX_VDC_WINDOW
XGL_CTX_VIEW_CLIP_BOUNDS
XGL_CTX_VIEW_TRANS

Special cases If the application only wants to push (and later pop) the model transforms (**XGL_CTX_LOCAL_MODEL_TRANS**(3) or **XGL_CTX_GLOBAL_MODEL_TRANS**(3)), the Context operator **xgl_context_update_model_trans**(3) should be used instead of **xgl_context_push**(3) and **xgl_context_pop**(3)**.**

RETURN VALUE Nothing is returned.

SEE ALSO
xgl_context_pop(3)
xgl_context_update_model_trans(3)
XGL_CTX_GLOBAL_MODEL_TRANS(3)
XGL_CTX_LOCAL_MODEL_TRANS(3)

NOTES For all the attributes of a context that reference other XGL objects (such as XGL_CTX_LINE_PATTERN, for example), pushing the attribute is equivalent to pushing the object handle. There is one exception to this rule — when the attribute represents a Transform object. In

such a case, the contents of the object are pushed, not the object handle.

NAME

xgl_context_set_multi_pixel – sets the *color* values for a list of pixel locations

SYNOPSIS

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

void  xgl_context_set_multi_pixel (
            Xgl_ctx       ctx,
            Xgl_usgn32    count,
            Xgl_pt_i2d    *position,
            Xgl_color     *value );
```

DESCRIPTION

This operator sets the color values for a list of pixels at the positions given by the *position* parameter in the specified buffer of the Raster associated with the Context *ctx*. The buffer is specified by the Context attribute **XGL_CTX_RENDER_BUFFER**(3). The *value* parameter specifies the color values that are set.

The parameter *position* is an array of x, y integers that contain pixel locations on screen. The parameter *value* is an array of colors that correspond to the position array. The length of these parameters is specified by the *count* parameter. Count pixels will be set in the Raster.

The Context attributes **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3) control the ROP operation realized when writing the pixel to the image buffer, and the *plane mask* limits the drawing operation to a given number of planes.

When setting a pixel to a Z Buffer, the value is set at *position*. Specifically, the Z mask (specified by **XGL_3D_CTX_Z_BUFFER_WRITE_MASK**(3)) and the Z comparison function (specified by **XGL_3D_CTX_Z_BUFFER_COMP_METHOD**(3)) are ignored.

When writing to an image buffer, if the color types of the Raster associated with the Context and underlying hardware match, no color conversion is done. If they don't match, a color conversion is done, using the Color Map object associated with the Device attached to the Context, so that the actual information written into the Device matches the hardware capabilities. The Color Map object associated with the Device can be controlled by the attribute **XGL_DEV_COLOR_MAP**(3)**.**

It is important to note that to have accurate comparisons, an application willing to compare pixel values written to an image buffer using **xgl_context_set_pixel**(3) with values obtained by reading the pixels from the image buffer using **xgl_context_get_pixel**(3)**,** should avoid any color conversion.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_get_pixel(3)
xgl_context_set_pixel(3)
xgl_context_set_pixel_row(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)
XGL_CTX_PLANE_MASK(3)

XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_ROP(3)
XGL_DEV_COLOR_MAP(3)
XGL_DEV_COLOR_TYPE(3)

NOTES

When the destination buffer is an image buffer, 4-bit and 8-bit Indexed (**XGL_DEV_-COLOR_TYPE**(3) is XGL_COLOR_INDEX) Memory Rasters store their pixel values as application-defined color values (that is, the index stored in the Memory Raster is exactly that specified by the *value* parameter of the function call).

This operation on a Z buffer is supported on a 3D Context only. During this operation, **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER. The Z buffer is created when the **XGL_3D_CTX_HLHSR_MODE** is set to XGL_HLHSR_Z_BUFFER for the first time.

XGL_CTX_RENDER_BUFFER(3) is a logical OR of XGL_RENDER_DRAW_BUFFER, XGL_RENDER_-DISPLAY_BUFFER, and XGL_RENDER_Z_BUFFER. In order for the operation to be performed on a Z-buffer, the application should only use XGL_RENDER_Z_BUFFER. This results in operation on the selected image buffer and the ignoring of the Z-buffer.

NAME

xgl_context_set_pixel – sets the color value of a specified pixel

SYNOPSIS

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

void  xgl_context_set_pixel (
            Xgl_ctx       ctx,
            Xgl_pt_i2d    *position,
            Xgl_color     *value );
```

DESCRIPTION

This operator sets the color value of a pixel located at the position given by *position* in a specified buffer of the Raster associated with the Context *ctx*. The buffer is specified by the Context attribute **XGL_CTX_RENDER_BUFFER**(3)**.** The color value that is set is specified by the parameter *value*.

The Context attributes **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3) are used to control the ROP operation realized when writing the pixel to the image buffer, and the *plane mask* is used to limit the drawing operation to a given number of planes.

When setting a pixel to a Z Buffer, the value is set at *position*. Specifically, the Z mask (specified by **XGL_3D_CTX_Z_BUFFER_WRITE_MASK**(3)) and the Z comparison function (specified by **XGL_3D_CTX_Z_BUFFER_COMP_METHOD**(3)) are ignored.

When writing to an image buffer, if the color types of the Raster associated with the Context and underlying hardware match, no color conversion is done. If they don't match, a color conversion is done, using the Color Map object associated with the Device attached to the Context, so that the actual information written into the Device matches the hardware capabilities. The Color Map object associated with the Device can be controlled by the Raster attribute **XGL_DEV_COLOR_MAP**(3)**.**

Note that to have accurate comparisons, an application willing to compare pixel values written to an image buffer using xgl_context_set_pixel with values obtained by reading the pixels from the image buffer using **xgl_context_get_pixel**(3)**,** should avoid any color conversion.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_set_multi_pixel(3)
xgl_context_get_pixel(3)
xgl_context_set_pixel_row(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_ROP(3)
XGL_DEV_COLOR_MAP(3)
XGL_DEV_COLOR_TYPE(3)

NOTES

When writing into an image buffer, 4-bit and 8-bit Indexed (**XGL_DEV_COLOR_TYPE**(3) is XGL_COLOR_INDEX) Memory Rasters store their pixel values as application-defined color values (that is, the index stored in the Memory Raster is exactly that specified by the *value* parameter of the function call).

This operation on a Z buffer is supported on a 3D Context only. During this operation, **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER. The Z buffer is created when the XGL_3D_CTX_HLHSR_MODE is set to XGL_HLHSR_Z_BUFFER for the first time.

XGL_CTX_RENDER_BUFFER(3) is a logical OR of XGL_RENDER_DRAW_BUFFER, XGL_RENDER_-DISPLAY_BUFFER, and XGL_RENDER_Z_BUFFER. In order for the operation to be performed on a Z-buffer, the application should only use XGL_RENDER_Z_BUFFER. This results in operation on the selected image buffer and the ignoring of the Z-buffer.

NAME

xgl_context_set_pixel_row – sets the color values for a row of pixel locations

SYNOPSIS

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

void  xgl_context_set_pixel_row (
            Xgl_ctx         ctx,
            Xgl_usgn32      stcol,
            Xgl_usgn32      row,
            Xgl_usgn32      count,
            Xgl_color       *value );
```

DESCRIPTION

This operator sets the color values for a row of pixel locations. The *stcol* parameter specifies the starting column in the specified buffer of the Raster to begin setting pixels. The row in the buffer where pixels will be set is specified in the *row* parameter. The buffer is specified by the Context attribute **XGL_CTX_RENDER_BUFFER**(3)**.**

The parameter *count* specifies the number of pixels to be set, and it is the length of the value array. The color of the pixels in the specified row is determined by the array of colors in the parameter *value*.

The Context attributes **XGL_CTX_ROP**(3) and **XGL_CTX_PLANE_MASK**(3) control the ROP operation realized when writing the pixel to the image buffer, and the *plane mask* limits the drawing operation to a given number of planes.

When setting a pixel to a Z Buffer, the value is set at *position*. Specifically, the Z mask (specified by **XGL_3D_CTX_Z_BUFFER_WRITE_MASK**(3)) and the Z comparison function (specified by **XGL_3D_CTX_Z_BUFFER_COMP_METHOD**(3)) are ignored.

When writing to an image buffer, if the color types of the Raster associated with the Context and underlying hardware match, no color conversion is done. If they don't match, a color conversion is done, using the Color Map object associated with the Device attached to the Context, so that the actual information written into the Device matches the hardware capabilities. The Color Map object associated with the Device can be controlled by the Raster attribute **XGL_DEV_COLOR_MAP**(3)**.**

It is important to note that, to have accurate comparisons, an application willing to compare pixel values written to an image buffer using xgl_context_set_pixel_row with values obtained by reading the pixels from the image buffer using **xgl_context_get_pixel**(3)**,** should avoid any color conversion.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_get_pixel(3)
xgl_context_set_pixel(3)
xgl_context_set_multi_pixel(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_3D_CTX_Z_BUFFER_COMP_METHOD(3)
XGL_3D_CTX_Z_BUFFER_WRITE_MASK(3)
XGL_CTX_PLANE_MASK(3)

XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_ROP(3)
XGL_DEV_COLOR_MAP(3)
XGL_DEV_COLOR_TYPE(3)

NOTES

When writing into an image buffer, 4-bit and 8-bit Indexed (**XGL_DEV_COLOR_TYPE**(3) is XGL_COLOR_INDEX) Memory Rasters store their pixel values as application-defined color values (that is, the index stored in the Memory Raster is exactly that specified by the *value* parameter of the function call).

This operation on a Z buffer is supported on a 3D Context only. During this operation, **XGL_3D_CTX_HLHSR_MODE**(3) should be set to XGL_HLHSR_Z_BUFFER. The Z buffer is created when the XGL_3D_CTX_HLHSR_MODE is set to XGL_HLHSR_Z_BUFFER for the first time.

XGL_CTX_RENDER_BUFFER(3) is a logical OR of XGL_RENDER_DRAW_BUFFER, XGL_RENDER_-DISPLAY_BUFFER, and XGL_RENDER_Z_BUFFER. In order for the operation to be performed on a Z-buffer, the application should only use XGL_RENDER_Z_BUFFER. This results in operation on the selected image buffer and the ignoring of the Z-buffer.

NAME

xgl_context_update_model_trans – performs specific operations with the Context's Model Transforms

SYNOPSIS

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

void  xgl_context_update_model_trans (
            Xgl_ctx                 ctx,
            Xgl_model_trans_request request );
```

DESCRIPTION

This operator is used to perform specific operations related to the Model Transform stack of the Context *ctx*, and the Context's Model Transforms. Some operations described here can be performed by using the **xgl_context_push**(3) and **xgl_context_pop**(3) operators, but using xgl_context_update_model_trans gives much better performance.

The actions taken by the call to xgl_context_update_model_trans are controlled by the parameter *request*. This parameter is the logical OR of possible Model Transforms actions defined by the enumerated data type *Xgl_model_trans_request*.

When building a request by OR-ing XGL_MTR_PUSH with XGL_MTR_LOCAL_TRANS or XGL_MTR_GLOBAL_TRANS, the xgl_context_update_model_trans operator pushes the contents of the Local Model Transform, or the contents of the Global Model Transform, respectively, onto the Model Transform stack associated with the Context. The maximum depth of the stack is controlled by the **XGL_CTX_MODEL_TRANS_STACK_SIZE**(3) attribute of the Context. It is possible to push both the Local and the Global Model Transforms in a single call to xgl_context_update_model_trans. In this case, the Local Model Transform is pushed first, followed by the Global Model Transform.

When building a request by OR-ing XGL_MTR_POP with XGL_MTR_LOCAL_TRANS or XGL_MTR_GLOBAL_TRANS, the xgl_context_update_model_trans operator will pop from the Model Transform stack associated with the Context to the contents of the Local Model Transform or the contents of the Global Model Transform, respectively. It is possible to pop both the Local and the Global Model Transforms in a single call to xgl_context_update_model_trans. In this case, the first Transform available on the stack is popped into the Global Model Transform, and the next available Transform is popped into the Local Model Transform.

A request with the XGL_MTR_NEW_LEVEL field set performs the following operations in the Context:

The Global Model Transform gets the contents of the Model Transform.

The Local Model Transform is reset to *Identity*.

This is a useful combination of operations when implementing a hierarchical display list in XGL. It ensures the best performance when traversing hierarchy levels.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_push(3)
xgl_context_pop(3)
XGL_CTX_MODEL_TRANS_STACK_SIZE(3)

NOTES

The *request* parameter is processed in the following order: xgl_context_update_model_-trans checks first for Transforms to pop, then for Transforms to push. Then, xgl_-context_update_model_trans checks for a *new level* flag.

NAME

xgl_gcache_multi_elliptical_arc – builds a Gcache of 3D elliptical arcs

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_multi_elliptical_arc (
            Xgl_gcache    gcache,
            Xgl_ctx       ctx,
            Xgl_ell_list  *ell_list );
```

DESCRIPTION

This operator caches the elliptical arcs in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the arcs are clipped against the model clipping planes, and the result is stored in the Gcache.

This Gcache can contain only 3D data, so the Context *ctx* must be a 3D Context. Also, a Gcache cannot be used with an annotation ellipse. The attribute XGL_GCACHE_USE_APPL_-GEOM is not used for arcs. The value of XGL_CTX_ARC_FILL_STYLE controls how the arcs are model clipped and rendered. If the fill style is XGL_ARC_OPEN, the arcs are model clipped and rendered as polylines; otherwise, they are treated as simple polygons.

The following attribute values are saved in the Gcache for later validation:

XGL_CTX_MIN_TESSELLATION,
XGL_CTX_MAX_TESSELLATION,
XGL_CTX_CURVE_APPROX,
XGL_CTX_CURVE_APPROX_VALUE,
XGL_CTX_ARC_FILL_STYLE,
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES, /* if MODEL_CLIP_PLANE_NUM > 0 */

The Gcache contains the arcs in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_multi_elliptical_arc(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_gcache_multi_simple_polygon – builds a Gcache of simple polygons

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_multi_simple_polygon (
        Xgl_gcache      gcache,
        Xgl_ctx         ctx,
        Xgl_facet_flags flags,
        Xgl_facet_list  *facets,
        Xgl_bbox        *bbox,
        Xgl_usgn32      num_pt_lists,
        Xgl_pt_list     pl[] );
```

DESCRIPTION

This operator caches the simple polygons in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the polygons are clipped against the model clipping planes, and the result is stored in the Gcache.

This Gcache can contain only 3D data, so the Context *ctx* must be a 3D Context. If the attribute XGL_GCACHE_USE_APPL_GEOM is TRUE and Model Clipping is not enabled, the Gcache does not copy the geometry. Instead, the Gcache keeps a pointer to it. It is the application's responsibility to ensure that the geometry passed as parameters to this function not be released to the system (freed) before **xgl_context_display_gcache**(3) is called.

A Gcache can also decompose a polygon into a set of triangles. The triangles are stored in model coordinates (MC), so the decomposed polygon can be treated by the same transformations as the original polygon. The attribute **XGL_GCACHE_DO_POLYGON_DECOMP**(3) controls whether decomposition is done. Decomposition is useful when using hardware graphics accelerators that specialize in triangles. Such an accelerator can draw the set of triangles in the decomposition faster than it can draw the polygon. However, the decomposition process can be time-consuming; it varies with the complexity of the polygon and the number of vertices that it contains.

The Gcache has two decomposition algorithms, one for complex polygons and one for non–self-intersecting (NSI) polygons. An NSI polygon can have any number of bounds, but the bounds cannot cross each other. A complex polygon does not have this restriction. An NSI polygon can be decomposed more quickly than a complex polygon. The Gcache uses the attribute XGL_GCACHE_POLYGON_TYPE to determine which decomposition algorithm to use.

Decomposition has a few limitations:

- The original polygon vertices are stored in the Gcache or referenced by it (if XGL_GCACHE_USE_APPL_GEOM is TRUE) even when the polygon is decomposed.
- The original polygon is used to draw the edges for the polygon or when the fill style is HOLLOW.
- Only polygons with floating point vertices can be decomposed.

The following attribute values are saved in the Gcache for later validation:

```
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES,      /* if MODEL_CLIP_PLANE_NUM > 0 */
```

The Gcache contains the polygons in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_multi_simple_polygon(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NOTES

All *pts* in the point list *pl[]* must have the same *Xgl_pt_type*.

NAME

xgl_gcache_multimarker – builds a Gcache of Markers

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_multimarker (
            Xgl_gcache   gcache,
            Xgl_ctx      ctx,
            Xgl_pt_list  *pl );
```

DESCRIPTION

This operator caches the Markers in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the Markers are clipped against the Model Clipping planes, and the result is stored in the Gcache.

A Gcache can only contain 3D data, so the Context *ctx* must be a 3D Context. If the attribute XGL_GCACHE_USE_APPL_GEOM is TRUE and Model Clipping is not enabled, the Gcache does not copy the geometry. Instead, the Gcache keeps a pointer to it. It is the application's responsibility to ensure that the geometry passed as parameters to this function not be released to the system (freed) before **xgl_context_display_gcache**(3) is called.

The following attribute values are saved in the Gcache for later validation:

```
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES,      /* if MODEL_CLIP_PLANE_NUM > 0 */
```

The Gcache contains the Markers in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_multimarker(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_gcache_multipolyline – builds a Gcache of multipolylines

SYNOPSIS

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

void  xgl_gcache_multipolyline (
            Xgl_gcache   gcache,
            Xgl_ctx      ctx,
            Xgl_bbox     *bbox,
            Xgl_usgn32   num_pt_lists,
            Xgl_pt_list  pl[] );
```

DESCRIPTION

This operator caches the polylines and their bounding boxes in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the polylines are clipped against the model clipping planes, and the result is stored in the Gcache.

A Gcache can only contain 3D data, so the Context *ctx* must be a 3D Context. If the attribute XGL_GCACHE_USE_APPL_GEOM is TRUE and Model Clipping is not enabled, the Gcache does not copy the geometry. Instead, the Gcache keeps a pointer to it. It is the application's responsibility to ensure that the geometry passed as parameters to this function not be released to the system (freed) before **xgl_context_display_gcache**(3) is called.

The following attribute values are saved in the Gcache for later validation:

XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES, /* if MODEL_CLIP_PLANE_NUM > 0 */

The Gcache contains the polylines in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_multipolyline(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NOTES

All *pts* in the point list *pl[]* must have the same *Xgl_pt_type*.

NAME

xgl_gcache_nurbs_curve – builds a Gcache of a NURBS curve

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_nurbs_curve (
        Xgl_gcache              gcache,
        Xgl_ctx                 ctx,
        Xgl_nurbs_curve         *curve,
        Xgl_bounds_f1d          *range,  /* optional */
        Xgl_curve_color_spline  *color_spline );  /* optional */
```

DESCRIPTION

This operator caches the NURBS curve defined by curve, range, and color spline parameters in the Gcache *gcache*. Depending on the value of **XGL_GCACHE_NURBS_CURVE_MODE**(3) attribute, tessellation-independent and/or polyline representations are created.

The parameters *curve*, *range*, and *color_spline* have the same meaning as they do in **xgl_nurbs_curve**(3)**.**

A Gcache of a curve can contain 2D or 3D data, so the Context *ctx* can be either a 2D or a 3D context. The attribute **XGL_GCACHE_USE_APPL_GEOM**(3) is ignored. The *Xgl_gcache_nurbs_mode* structure is defined as follows:

```
typedef enum {
        XGL_GCACHE_NURBS_DYNAMIC,
        XGL_GCACHE_NURBS_COMBINED,
        XGL_GCACHE_NURBS_STATIC
} Xgl_gcache_nurbs_mode;
```

If the value of **XGL_GCACHE_NURBS_CURVE_MODE**(3) attribute is XGL_GCACHE_NURBS_STATIC, the following attributes are used to compute the polyline representation of the curve and are also saved in the Gcache for later validation:

XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)
XGL_3D_CTX_MODEL_CLIP_PLANES(3)
/* if MODEL_CLIP_PLANE_NUM > 0 */

For Gcache mode XGL_GCACHE_NURBS_STATIC, the Gcache contains the polyline version of the curve in model coordinates (MC).

For Gcache modes XGL_GCACHE_NURBS_DYNAMIC and XGL_GCACHE_NURBS_COMBINED, no context attributes are saved for validation. For XGL_GCACHE_NURBS_DYNAMIC mode, only tessellation-independent form is stored in the Gcache. For XGL_GCACHE_NURBS_COMBINED mode, both tessellation-independent and polyline representations are stored. Also, in these cases, model clipping according to the Context *ctx* is not performed at Gcache creation time.

The display of Gcache is affected by the full transformation and clipping pipeline. The Gcache does not save the value of the parameters *curve, range*, and *color_spline*. The application is responsible for building a new Gcache with the new value of these parameters.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_nurbs_curve(3)
XGL_3D_CTX_MODEL_CLIP_PLANES(3)
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_GCACHE_NURBS_CURVE_MODE(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_gcache_nurbs_surface – builds a Gcache of a NURBS surface

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_nurbs_surface (
        Xgl_gcache                  gcache,
        Xgl_ctx                     ctx,
        Xgl_nurbs_surf              *surface,
        Xgl_trim_loop_list          *trimming,
        Xgl_nurbs_surf_simple_geom  *hints,  /* optional */
        Xgl_surf_color_spline       *color_spline,  /* optional */
        Xgl_surf_data_spline_list   *data_splines );  /* not supported */
```

DESCRIPTION

This operator caches the NURBS surface defined by surface, trimming, hints, color spline, and data splines parameters in the Gcache *gcache*. Depending on the value of **XGL_GCACHE_NURBS_SURF_MODE**(3) attribute, tessellation-independent and/or polygon representations (quadrilateral meshes, triangle strips, and/or polylines) are created.

The parameters *surface*, *trimming*, *hints*, *color_spline*, and *data_splines* have the same meaning as they do in **xgl_nurbs_surface**(3)**.**

A Gcache of a surface can contain 3D data only, so the Context *ctx* has to be a 3D context. The attribute **XGL_GCACHE_USE_APPL_GEOM**(3) is ignored. The *Xgl_gcache_nurbs_mode* structure is defined as follows:

```
typedef enum {
        XGL_GCACHE_NURBS_DYNAMIC,
        XGL_GCACHE_NURBS_COMBINED,
        XGL_GCACHE_NURBS_STATIC
} Xgl_gcache_nurbs_mode;
```

If the value of **XGL_GCACHE_NURBS_SURF_MODE**(3) attribute is XGL_GCACHE_NURBS_STATIC, the following attribute values are used to compute the tessellated representation of the surface and are also saved in the Gcache for later validation:

XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)
XGL_3D_CTX_MODEL_CLIP_PLANES(3)
/* if MODEL_CLIP_PLANE_NUM > 0 */

For Gcache mode XGL_GCACHE_NURBS_STATIC, the Gcache contains the tessellated version of the surface in model coordinates (MC).

For Gcache modes XGL_GCACHE_NURBS_DYNAMIC and XGL_GCACHE_NURBS_COMBINED, no context attributes are saved for validation. For XGL_GCACHE_NURBS_DYNAMIC mode, only tessellation-independent form is stored in the Gcache. For XGL_GCACHE_NURBS_COMBINED mode, both tessellation-independent and polygon representations are stored. Also, in these cases, model clipping according to the Context *ctx* is not performed at Gcache creation time.

The display of Gcache is affected by the full transformation and clipping pipeline. The Gcache does not save the value of the parameters *surface, trimming, hints, color_spline*, and *data_splines*. The application is responsible for building a new Gcache with the new value of these parameters.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_nurbs_surface(3)

XGL_3D_CTX_MODEL_CLIP_PLANES(3)
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)
XGL_GCACHE_NURBS_SURF_MODE(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_gcache_polygon – builds a Gcache of a general polygon

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_polygon (
        Xgl_gcache      gcache,
        Xgl_ctx         ctx,
        Xgl_facet_type  facet_type,
        Xgl_facet       *facet,
        Xgl_bbox        *bbox,
        Xgl_usgn32      num_pt_lists,
        Xgl_pt_list     pl[] );
```

DESCRIPTION

This operator caches a polygon in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the polygon is clipped against the model clipping planes, and the result is stored in the Gcache.

This Gcache can contain only 3D data, so the Context *ctx* must be a 3D Context. If the attribute XGL_GCACHE_USE_APPL_GEOM is TRUE and Model Clipping is not enabled, the Gcache does not copy the geometry. Instead, the Gcache keeps a pointer to it. It is the application's responsibility to ensure that the geometry passed as parameters to this function not be released to the system (freed) before **xgl_context_display_gcache**(3) is called.

A Gcache can also decompose a polygon into a set of triangles. The triangles are stored in model coordinates (MC), so the decomposed polygon can be treated by the same transformations as the original polygon. The attribute **XGL_GCACHE_DO_POLYGON_DECOMP**(3) controls whether decomposition is done. Decomposition is useful when using hardware graphics accelerators that specialize in triangles. Such an accelerator can draw the set of triangles in the decomposition faster than it can draw the polygon. However, the decomposition process can be time-consuming; it varies with the complexity of the polygon and the number of vertices that it contains.

The Gcache has two decomposition algorithms, one for complex polygons and one for non–self-intersecting (NSI) polygons. An NSI polygon can have any number of bounds, but the bounds cannot cross each other. A complex polygon does not have this restriction. An NSI polygon can be decomposed more quickly than a complex polygon. The Gcache uses the attribute XGL_GCACHE_POLYGON_TYPE to determine which decomposition algorithm to use.

Decomposition has a few limitations:

- The original polygon vertices are stored in the Gcache or referenced by it (if XGL_GCACHE_USE_APPL_GEOM is TRUE) even when the polygon is decomposed.
- The original polygon is used to draw the edges for the polygon or when the fill style is HOLLOW.
- Only polygons with floating point vertices can be decomposed.

The following attribute values are saved in the Gcache for later validation:

XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES, /* if MODEL_CLIP_PLANE_NUM > 0 */

The Gcache contains the polygon in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_polygon(3)
XGL_GCACHE_DO_POLYGON_DECOMP(3)
XGL_GCACHE_POLYGON_TYPE(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NOTES

All *pts* in the point list *pl[]* must have the same *Xgl_pt_type*.

NAME

xgl_gcache_stroke_text – builds a Gcache of a text string

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_gcache_stroke_text (
        Xgl_gcache      gcache,
        Xgl_ctx         ctx,
        void            *str,
        Xgl_pt_f{2,3}d  *pos,
        Xgl_pt_f3d      dir[] );
```

DESCRIPTION

This operator caches the polyline representation of the text string *str* in the Gcache *gcache*. The text attributes in the Context *ctx* are used when building the cached representation of the text string. The parameters *str*, *pos*, and *dir* have the same meaning as they do in **xgl_stroke_text**(3)**.**

A Gcache of stroke text can contain 2D or 3D data, so the Context *ctx* can be either a 2D or 3D Context. The attribute XGL_GCACHE_USE_APPL_GEOM is ignored for stroke text. The dimension of the context used to write text to the gcache must match the dimension of the context used to display it. For example, if you write 2d stroke text (using a 2d context), you must use a 2d context (although it doesn't have to be the same one).

The following attribute values are used to compute the polyline representation of the text string and are also saved in the Gcache for later validation:

XGL_CTX_SFONT_0
XGL_CTX_SFONT_1
XGL_CTX_SFONT_2
XGL_CTX_SFONT_3
XGL_CTX_STEXT_ALIGN_HORIZ
XGL_CTX_STEXT_ALIGN_VERT
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR
XGL_CTX_STEXT_CHAR_HEIGHT
XGL_CTX_STEXT_CHAR_SLANT_ANGLE
XGL_CTX_STEXT_CHAR_SPACING
XGL_CTX_STEXT_CHAR_UP_VECTOR
XGL_CTX_STEXT_PATH
XGL_CTX_STEXT_PRECISION
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM
XGL_3D_CTX_MODEL_CLIP_PLANES /* if MODEL_CLIP_PLANE_NUM > 0 */

The Gcache contains the polyline version of the text in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline. The Gcache does not save the value of the parameters (*str*, *pos*, and *dir*), however. The application is responsible for building a new Gcache with the new values of these parameters.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_stroke_text(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_gcache_triangle_strip – builds a Gcache of a triangle strip

SYNOPSIS

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

void  xgl_gcache_triangle_strip (
            Xgl_gcache      gcache,
            Xgl_ctx         ctx,
            Xgl_facet_list  *facets,
            Xgl_pt_list     *pl );
```

DESCRIPTION

This operator caches a triangle strip in the Gcache *gcache*. If Model Clipping is enabled (that is, if XGL_3D_CTX_MODEL_CLIP_PLANE_NUM is greater than zero), the triangle strip is clipped against the model clipping planes, and the result is stored in the Gcache.

This Gcache can contain only 3D data, so the Context *ctx* must be a 3D Context. If the attribute XGL_GCACHE_USE_APPL_GEOM is TRUE and Model Clipping is not enabled, the Gcache does not copy the geometry. Instead, the Gcache keeps a pointer to it. It is the application's responsibility to ensure that the geometry passed as parameters to this function not be released to system (freed) before **xgl_context_display_gcache**(3) is called.

The following attribute values are saved in the Gcache for later validation:

XGL_3D_CTX_MODEL_CLIP_PLANE_NUM,
XGL_3D_CTX_MODEL_CLIP_PLANES, /* if MODEL_CLIP_PLANE_NUM > 0 */

The Gcache contains the triangle strip in model coordinates (MC). Therefore, the display of the Gcache is affected by the full transformation and clipping pipeline.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_context_display_gcache(3)
xgl_triangle_strip(3)
XGL_GCACHE_USE_APPL_GEOM(3)

NAME

xgl_image – displays a block of pixels from an XGL Raster

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_image (
        Xgl_ctx             dest_ctx,
        Xgl_pt_f{2,3}d      *position,
        /* f2d for 2D context, f3d for 3D context */
        Xgl_bounds_i2d      *rectangle,
        Xgl_ras             src_ras );
```

DESCRIPTION

This operator copies a rectangular block of pixels from the source Memory Raster *src_ras* to the Raster associated with the destination Context *dest_ctx*, which can either be a Memory Raster or a Window Raster. The area to be copied is specified by a rectangular region, *rectangle*, in the source Raster. The copy starts at a position specified by *position* in the destination Context.

The position must be a valid point in the Context's *model space*, and it is transformed according to the Context state. Nothing is copied if the position is clipped by model or view clipping. Otherwise, the source rectangle is copied into the Raster, after clipping to the intersection of Context DC viewport limits and active VDC clipping bounds (transformed into DC coordinates).

This operator performs the copy, taking into account the HLHSR state of the Context. Depending on the state of the Context attribute **XGL_3D_CTX_HLHSR_MODE**(3)**,** *z* compare and *z* update may take place at the time of the copy, offering the application programmer the possibility of mixing raster images with 3D geometry. In this case, the *Z* value of the transformed position will be used for Z buffering the image.

If the rectangle is a NULL pointer, the maximum area from the source Raster is taken for the copy operation.

The ROP mode and plane mask defined in the destination Context with the **XGL_CTX_-ROP**(3) and **XGL_CTX_PLANE_MASK**(3) attributes are enforced during this operation. The fill rule (**XGL_CTX_RASTER_FILL_STYLE**(3)) is not used. This operator is unaffected by the **XGL_CTX_RENDER_BUFFER**(3) and **XGL_RAS_SOURCE_BUFFER**(3) attributes. The source and destination buffers are always draw buffer.

This operator enables copying from any source Memory Raster to the device attached to the Context. If the destination Raster attached to the Context and the source Raster have different depths, XGL performs a conversion.

This operator behaves similarly to the **xgl_context_copy_buffer**(3) operator with the following exceptions:

- it restricts *src_ras* to be an XGL Memory Raster,
- its position is in model coordinates,
- it does not use fill rule (**XGL_CTX_RASTER_FILL_STYLE**(3)),
- it accounts for the HLHSR state of the Context,

- the source and destination buffers are always draw buffer, and
- nothing is drawn when the position is clipped.

See **xgl_context_copy_buffer**(3) for more information.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_context_copy_buffer(3)
XGL_3D_CTX_HLHSR_MODE(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_RASTER_FILL_STYLE(3)
XGL_CTX_RENDER_BUFFER(3)
XGL_CTX_ROP(3)
XGL_RAS_SOURCE_BUFFER(3)

NAME

xgl_inquire – inquires the acceleration features underlying a window

SYNOPSIS

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

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

DESCRIPTION

Purpose

This operator inquires into the acceleration features underlying a particular window. XGL accelerates a primitive if some portion of the geometry that defines the primitive is processed by accelerating hardware. Acceleration includes geometric transformations and clipping as well as actual rendering. The **xgl_open**(3) operator must be called prior to calling this operator.

Parameters

sys_state

The handle to system state information for this XGL session. Use the value returned by the call to **xgl_open**(3)**.**

For more detailed information about this value, see **XGL_SYS_STATE**(3) and its corresponding man pages.

desc The application must fill in this parameter with the appropriate window information before calling this function.

Returned Information

The function returns a pointer to a structure of type Xgl_inquire, allocated by XGL. It is the responsibility of the application program to free the memory allocated by this function call when the returned information is no longer needed. The structure returned contains, in the following fields, information about the frame buffer underlying the window:

name The XGL name of the frame buffer underlying the window.

dga_flag (direct graphics access)

If this field is TRUE, XGL does direct graphics access to render pixels into the frame buffer underlying the window. If it is FALSE, XGL uses either the PEX or Xlib protocol to render pixels.

color_type

A structure with 2 bit fields that specify the color type accelerated by the frame buffer underlying the window. If *index* is TRUE, the frame buffer accelerates the XGL_COLOR_INDEX color model. If *rgb* is TRUE, the frame buffer accelerates the XGL_COLOR_RGB color model.

depth The depth of the frame buffer underlying the window.

width and height

The width and height of the frame buffer in pixel coordinates.

maximum_buffer

The maximum number of buffers available from the frame buffer underlying the window. A value of 2 means the application might be able to do

double buffering.

db_buffer_is_copy
If this field is TRUE, the underlying hardware device performs double buffering by copying pixels from the draw (hidden) buffer to the display buffer.

pt_type A structure of bit fields whose values are the pt_types supported by the frame buffer. The bit fields are:
pt_dim_2d, if TRUE means the frame buffer can accelerate 2D coordinates
pt_dim_3d, if TRUE means the frame buffer can accelerate 3D coordinates
pt_type_int, if TRUE means the frame buffer can accelerate integer coordinates
pt_type_float, if TRUE means the frame buffer can accelerate float coordinates
point types.

hlhsr_mode
Indicates whether the frame buffer underlying the window directly supports a *z*-buffer and *z*-buffer rendering. If this value is XGL_HLHSR_Z_-BUFFER, the frame buffer accelerates *z*-buffer rendering. If this value is XGL_HLHSR_NONE, the frame buffer does not support accelerated *z*-buffer rendering, and the *z*-buffer is supported in the host.

picking
double_buffer
indexed_color
true_color
depth_cueing
lighting
shading
hlhsr
antialiasing

Depending on the underlying hardware accelerator, if any, these fields will be set to XGL_INQ_NOT_SUPPORTED if the functionality is not supported, XGL_INQ_SOFTWARE if the support is done through software emulation, and XGL_INQ_HARDWARE if hardware acceleration is provided.

stereo This boolean indicates whether or not the device underlying the window has hardware support for stereo. Please refer to **XGL_WIN_RAS_STEREO_-MODE**(3) for more information on hardware stereo support.

extns This bitmask indicates the extensions supported by the frame buffer underlying the window. Currently, only the ability to query for multi-buffer extension to X(MBX) is provided. The logical AND of this field with bitmask XGL_INQ_MULTI_BUFFERING determines whether MBX is supported. If the result is non-zero, the frame buffer supports MBX.

SEE ALSO
xgl_object_create(3)
xgl_open(3)

XGL_3D_CTX_HLHSR_MODE(3)
XGL_SYS_STATE(3)
XGL_WIN_RAS_BUFFERS_ALLOCATED(3)
XGL_WIN_RAS_BUFFERS_REQUESTED(3)
XGL_WIN_RAS_STEREO_MODE(3)

NAME

xgl_light_copy – makes a copy of a Light

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_light_copy (
            Xgl_light    dest_light,
            Xgl_light    src_light );
```

DESCRIPTION

Purpose

This operator makes a copy of a Light. All the state information of a Source Light is transferred to a Destination Light. The state information consists of the values of all attributes.

Parameters

dest_light

The Destination Light to which the state information is stored. It must have been created previously with **xgl_object_create**(3)**.** Any state information previously stored in *dest_light* is lost.

src_light

The Source Light from which the state information is taken. It must have been created previously with xgl_object_create.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

NAME

xgl_mipmap_texture_build – generates a mipmap pyramid

SYNOPSIS

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

void  xgl_mipmap_texture_build (
          Xgl_mipmap_texture     *texture_obj,
          Xgl_mem_ras            *in_mem_ras,
          Xgl_texture_boundary   u_bound,
          Xgl_texture_boundary   v_bound );
```

DESCRIPTION

Purpose

This operator creates a mipmap pyramid using the *in_mem_ras* as the base of the pyramid. The number of levels generated in the mipmap is dependent on the attribute **XGL_MIPMAP_TEXTURE_LEVELS**(3)**.**

Parameters

texture_obj

A pointer to the already created texture object.

in_mem_ras

A pointer to the memory raster that the application wants to turn into a mipmap within the texture object.

u{v}_bound

These input fields are used as hints during the mipmap generation. Using these fields, an application provides information on how texture borders are treated for wrapping or clamping. XGL accepts the following values for *Xgl_texture_-boundary*:

XGL_TEXTURE_BOUNDARY_CLAMP

Finish texturing using a constant value.

XGL_TEXTURE_BOUNDARY_WRAP

Repeat the texture.

XGL_TEXTURE_BOUNDARY_TRANSPARENT

Finish rendering as though texturing were not being performed.

XGL_TEXTURE_BOUNDARY_MIRROR

Reverse texel sampling creating a *mirrored alternating* pattern.

XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY

Use a color specified by the closest boundary texel.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
XGL_MIPMAP_TEXTURE_LEVELS(3)

NAME

xgl_multi_elliptical_arc – draws a list of 3D elliptical arcs

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multi_elliptical_arc (
            Xgl_ctx         ctx,
            Xgl_ell_list    *ell_list );
```

DESCRIPTION

Purpose

This operator draws a list of elliptical arcs according to the graphics state described in the 3D Context *ctx*. Each elliptical arc in the list is transformed by the current transformation matrix (see **XGL_CTX_VIEW_TRANS**(3) and **XGL_CTX_MODEL_TRANS**(3)). The elliptical arcs are rendered in the plane described by the two direction vectors provided with each arc. The result is then projected using the current set of Transforms, so that all of the standard 3D pipeline applies. The elliptical arcs are approximated as polygons, using the following context attributes:

XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)

Parameters

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

ell_list A pointer to the list of arcs to draw. It is a structure defined as:

```
typedef struct {
        Xgl_usgn32          num_ells;   /* number of elliptical arcs */
        Xgl_multiell_type   type;       /* data type of multiell */
        Xgl_bbox            *bbox;      /* optional box containing all
                                           elliptical arcs, NULL means no box */
        union {                         /* union of pointers to ells */
                Xgl_ell_f3d     *f3d;
                Xgl_ell_af3d    *af3d;
                Xgl_ell_d3d     *d3d;
                Xgl_ell_ad3d    *ad3d;
        } ells;
} Xgl_ell_list;
```

ell_list→num_ells is the number of arcs in the list.

ell_list→type is the data type of the center point and radius used to define each arc in the list. *Xgl_multiell_type* currently defines the following enumerated values as possible multiell data types:

XGL_MULTIELLARC_F3D
XGL_MULTIELLARC_AF3D
XGL_MULTIELLARC_D3D
XGL_MULTIELLARC_AD3D

Ultimately, this determines which member of the union *ell_list→ells* (see below) points to the elliptical arc data.

ell_list→bbox points to a structure describing a bounding box for all of the arcs in the list. This may be NULL to indicate that no bounding box information is available.

ell_list→ells is a union of pointers, one of which points to the array of structures that defines each elliptical arc in the list. The *ell_list→ells* structure for elliptical arcs of type XGL_MULTIELLARC_F3D looks as follows:

```
typedef struct {
        Xgl_pt_flag_f3d   center;           /* center point */
        Xgl_pt_f3d        dir[3];           /* dir. vectors and normal vector */
        Xgl_boolean       dir_normalized;   /* TRUE when dir[] are unit vectors */
        Xgl_boolean       dir_normal;       /* TRUE if dir[2] is a valid vector */
        float             major_axis,       /* major and minor axis */
                          minor_axis;
        float             rot_angle;        /* ellipse rotation angle */
        float             start_angle,      /* start and stop angles */
                          stop_angle;
} Xgl_ell_f3d;
```

See **xgl_struct**(3) for structures defining other elliptical arcs in the list.

center and *major/minor* axis members of the structure are *Xgl_pt_flag_f3d* and *float*, respectively.

The first two elements of the field *dir[3]* contain the two orthogonal vectors in model coordinates (MC) used to define the plane on which the elliptical arc is going to be built. The third element is used to specify a normal vector for the arc.

dir_normalized should be set to TRUE when the vectors are normalized (that is, unit vectors). *dir_normal* should be set to TRUE when the third element of *dir* is set to a valid vector. Otherwise, XGL will compute a normal vector to the elliptical arc, using the two vectors specified for the arc plane.

When specifying type XGL_MULTIELLARC_AF3D for elliptical arcs, XGL generates *annotation* elliptical arcs. In other words, the elliptical arcs XGL generates are guaranteed to be in a plane parallel to the projection plane. In this particular case, the center of each arc represents the reference point and is specified in model coordinates. The major and minor axes are specified in virtual device coordinates (VDC). There is no need to specify any direction vectors, and the primitive is always front-facing.

As shown in the *Xgl_ell_f3d* structure, each elliptical arc is specified by an application-defined center point, major / minor axes, rotation angle, and start and stop angle. The *center* and *major/minor* axes are specified in the application's coordinate system, unless the application chooses to render annotation ellipses, as defined earlier. The *rotation* angle is the angle that specifies the rotation of the ellipse, measured between the axis defined by *dir[0]* in the plane of the ellipse and the major axis.

The *start_angle* is the angle from the axis defined by *dir[0]* in the plane of the ellipse to the start point of the arc. The *stop_angle* is the angle from the axis defined by *dir[0]* in the plane of the ellipse to the end point. All angles are specified in radians.

Elliptical arcs can be displayed in one of three ''styles'' depending on the current setting for the attribute **XGL_CTX_ARC_FILL_STYLE**(3). If the arc is closed (that is, if XGL_CTX_ARC_FILL_STYLE is set to either XGL_ARC_SECTOR or XGL_ARC_CHORD), the following surface fill attributes control the appearance of the enclosed area of each arc:

XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_FRONT_COLOR(3)

The following equivalent attributes for back-facing polygons also apply.

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_COLOR(3)

The lighting and shading attributes are also in effect.

Edges can be drawn around closed arcs. To do so, first set the attribute **XGL_CTX_SURF_EDGE_FLAG**(3) to TRUE. Edges will be drawn around each elliptical arc within the arc list that has the flag associated with its *center* point set. Note that the *center* point of each arc always has a flag associated with it. XGL assumes the application will initialize these flags (to either **0** or **1**) before calling xgl_multi_elliptical_arc. If the flags are not initialized, unpredictable results may occur.

For open arcs (that is, arcs drawn when XGL_CTX_ARC_FILL_STYLE is set to XGL_ARC_OPEN) the line attributes control how the arc is drawn on the Raster. The following attributes are used:

XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_COLOR(3)

XGL_CTX_LINE_ALT_COLOR(3)

For a more complete summary of these attributes, see **xgl_multipolyline**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_multipolyline(3)
xgl_object_get(3)
xgl_object_set(3)
XGL_CTX_ARC_FILL_STYLE(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(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_DC_OFFSET(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_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_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NOTES

xgl_multi_elliptical_arc is available only for 3D Contexts.

The *center* point of each arc must have the flag associated with it set to either **0** or **1** (to turn the arc's edge either Off or On). If the flag is not set by the application, unpredictable results may occur.

If the difference between the *start_angle* and *stop_angle* of an arc is greater than 2π radians, then an arc is drawn instead of an ellipse. For example, if *start_angle* is 0.0 radians and *stop_angle* is $(5*\pi/2)$ radians, then an arc is drawn as if the *stop_angle* were $\pi/2$ radians.

NAME

xgl_multi_simple_polygon – draws simple polygons from a list of data

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multi_simple_polygon (
        Xgl_ctx          ctx,
        Xgl_facet_flags  flags,
        Xgl_facet_list   *facets,
        Xgl_bbox         *bbox,
        Xgl_usgn32       num_pt_lists,
        Xgl_pt_list      pl[] );
```

DESCRIPTION

Purpose

This operator draws a set of simple polygons according to the parameter list and the graphics state described in the Context *ctx*, which may be either a 2D or 3D Context. All of the Context attributes that directly affect the display of polygons are listed in the **SEE ALSO** section below.

A *simple* polygon is a polygon that is described by a single list of *n* vertices *v(0), v(1), ..., v(n-1)*. These vertices form the edges *v(0)* to *v(1), v(1)* to *v(2), ..., v(n-2)* to *v(n-1)*, and *v(n-1)* to *v(0)*. The resulting polygon can be self-intersecting or non-planar. Each polygon described by *pl[]* is drawn separately, starting at *pl[0]* and ending at *pl[num_pt_lists-1]*. This order of drawing is guaranteed.

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing xgl_multi_simple_polygon.

flags Contains bits that specify the kind of polygons being drawn. Xgl_facet_flags is specified as an enumerated type, but it actually contains several values OR'd together. The bits in flags apply to *all* polygons in *pl[]*.

Here are the available *Xgl_facet_flags*:

Bits for number of sides

XGL_FACET_FLAG_SIDES_ARE_3
XGL_FACET_FLAG_SIDES_ARE_4
XGL_FACET_FLAG_SIDES_UNSPECIFIED

At most, one of these can be set. If either XGL_FACET_FLAG_SIDES_ARE_3 or XGL_FACET_FLAG_SIDES_ARE_4 is set, each contour in *pl[]* is assumed to represent a triangle or quadrilateral, respectively. Specifically, if XGL_-FACET_FLAG_SIDES_ARE_3 is set, the first three points of each *pl[]* element are used to draw the polygon. Similarly, if XGL_FACET_FLAG_SIDES_ARE_4 is set, the first four points are used. If XGL_FACET_FLAG_SIDES_-UNSPECIFIED is set, no assumption is made regarding the number of sides.

It is not an error for either the triangles or quadrilaterals to be degenerate. It is an error if there are fewer points in *pl[i]* than specified by this flag. The default value is XGL_FACET_FLAG_SIDES_UNSPECIFIED. To select these bits, use *flags* and XGL_FACET_FLAG_SIDES_MASK.

Bits for convexity

XGL_FACET_FLAG_SHAPE_CONVEX
XGL_FACET_FLAG_SHAPE_NONCONVEX
XGL_FACET_FLAG_SHAPE_UNKNOWN

At most, one of these can be set. In the 3D case, a non-planar polygon is generally not convex when projected to the viewing plane; therefore, it must have XGL_FACET_FLAG_SHAPE_NONCONVEX set. If no information is known about the polygon, XGL_FACET_FLAG_SHAPE_UNKNOWN must be set. The default value is XGL_FACET_FLAG_SHAPE_UNKNOWN. To test these bits, use *flags* and XGL_FACET_FLAG_SHAPE_MASK.

Bits for facet normal and vertex order

XGL_FACET_FLAG_FN_CONSISTENT
XGL_FACET_FLAG_FN_NOT_CONSISTENT

Setting the XGL_FACET_FLAT_FN_CONSISTENT flag indicates whether the direction of facet normal and the order in which the points are given are consistent for all multisimple polygons.

In a right-handed coordinate system, the order of the points implies a facet normal that is calculated as follows:

implied facet normal = $v1v2 \times v1v3$

A user-provided facet normal indicates the front face of a polygon. Call a group of polygons XGL_FACET_FLAG_FN_CONSISTENT if either of the following is true:

- The front of the polygon chosen by the user-provided facet normal always *agrees* with the front chosen by the implied facet normal.
- The front chosen by the user-provided facet normal always *disagrees* with that chosen by the implied facet normal.

Call a group of polygons XGL_FACET_FLAG_FN_NOT_CONSISTENT if neither of the above conditions holds. That is, for some polygons, there is agreement; for others, there is disagreement.

The default is XGL_FACET_FLAG_FN_NOT_CONSISTENT.

Setting this flag with XGL_FACET_FLAG_FN_CONSISTENT gives better performance on some devices. Consult your accelerator guide for details.

Bits for geometry data memory allocation

XGL_FACET_FLAG_DATA_CONTIG
XGL_FACET_FLAG_DATA_NOT_CONTIG

Setting the XGL_FACET_FLAG_DATA_CONTIG indicates that the geometry data being sent is contiguous in memory. With this information some rendering devices will be able to process the geometry data more efficiently.

When this attribute is set, the application guarantees that all geometry

data is stored in one contiguous memory location. Setting XGL_FACET_-FLAG_DATA_NOT_CONTIG means the application cannot guarantee that the geometry data is contiguous in memory, resulting in slower performance on some rendering devices. To determine if your rendering device can take advantage of XGL_FACET_FLAG_DATA_CONTIG, consult the accelerator guide for your device.

The default is XGL_FACET_FLAG_DATA_NOT_CONTIG. To test this bit, use *flags* and XGL_FACET_FLAG_DATA_NOT_CONTIG.

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 of facets */
                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, then *facets→facets.color[i]* should be the data for the *i'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 polygons if either *facets* is NULL, *facets→facet_type* is XGL_FACET_NONE, or *facets→num_facets* is less than *num_pt_lists*. Otherwise, the data in the *i'th* facet is used for the simple polygon whose boundary is described by *pl[i]*. Normal information supplied to 2D Contexts is ignored.

bbox Is either NULL or is the bounding box around all of the points in *pl[]*. The bounding box information is 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* can be NULL. If the bounding box is given, it is the application's responsibility to ensure 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.

num_pt_lists
The number of elements in *pl[]*.

pl A pointer to the first element of an array of Xgl_pt_list structures, which are 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;
```

Each *pl[i]* specifies the vertices of a simple polygon.

pl[i].pt_type specifies the type of the points in *pl[i].pts*. Note that all of the *types* in the list pointed to by *pts* must be of this type.

pl[i].bbox specifies a bounding box for the *i'th* polygon. If XGL_FACET_FLAG_SIDES_ARE_3 or XGL_FACET_FLAG_SIDES_ARE_4 are set, this bounding box is ignored, since it takes longer to process the bounding box data than to clip the triangle or quadrilateral directly.

pl[i].num_pts is the number of points in the array *pts*. Specifically, it is the number of vertices in the *i'th* simple polygon. If XGL_FACET_FLAG_SIDES_ARE_3 or XGL_FACET_FLAG_SIDES_ARE_4 are set, the number of vertices in each polygon is known. In that case, the value of *pl[i].num_pts* is ignored.

pl[i].num_data_values specifies the number of data values. Specifically, it is the number of data values per vertex in the *i'th* simple polygon. This value is looked into only if the *pl[i].pt_type* is specified as having data values. For more information on how the data values are used, see **xgl_polygon**(3)**.**

pl[i].pts is a union of pointers, one of which points to the array of points that make up the vertices of the *i'th* simple polygon. The value of *pl[i].pt_type* determines the interpretation of this union.

For further details about this union of pointers and the effect of *pl[i].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, the flag bits are interpreted as specified in **xgl_polygon**(3)**.**

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_polygon(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, and vice versa.

Only XGL_FACET_NONE or XGL_FACET_COLOR should be used in *facets→facet_type* for 2D Contexts.

The exact appearance of non-planar polygons when projected on the viewing surface is hardware-dependent.

The results of rendering degenerate polygons is hardware-dependent.

NAME

xgl_multiarc – draws a list of arcs

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multiarc (
            Xgl_ctx         ctx,
            Xgl_arc_list    *arc_list );
```

DESCRIPTION

Purpose

This operator draws a list of arcs of a circle according to the graphics state described in the Context *ctx*. Each arc in the list is transformed by the current transformation matrix (see **XGL_CTX_VIEW_TRANS**(3) and **XGL_CTX_MODEL_TRANS**(3)) and drawn in the plane of the view surface in the case of 2D Contexts. When calling xgl_multiarc with a 3D Context, the arcs are rendered in the plane described by the two direction vectors provided with each arc. The result is then projected using the current set of Transforms, so that all of the standard 3D pipeline applies. In the case of 3D arcs, the arcs are approximated as polygons, using the following context attributes:

XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_multiarc**(3)**.**

arc_list A pointer to the list of arcs to draw. It is a structure defined as:

```
typedef struct {
        Xgl_usgn32          num_arcs;   /* number of arcs */
        Xgl_multiarc_type   type;       /* data type of multiarc */
        Xgl_bbox            *bbox;      /* optional box containing all arcs */
        union {                         /* union of pointers to arcs */
                Xgl_arc_i2d     *i2d;
                Xgl_arc_f2d     *f2d;
                Xgl_arc_d2d     *d2d;
                Xgl_arc_f3d     *f3d;
                Xgl_arc_af3d    *af3d;
                Xgl_arc_d3d     *d3d;
                Xgl_arc_ad3d    *ad3d;
        } arcs;
} Xgl_arc_list;
```

arc_list→num_arcs is the number of arcs in the list.

arc_list→type is the data type of the center point and radius used to define each arc in the list. Xgl_multiarc_type defines the following enumerated values as possible multiarc data types:

XGL_MULTIARC_I2D

XGL_MULTIARC_F2D
XGL_MULTIARC_D2D
XGL_MULTIARC_F3D
XGL_MULTIARC_AF3D.
XGL_MULTIARC_D3D
XGL_MULTIARC_AD3D.

Ultimately, this determines which member of the union *arc_list→arcs* (see below) points to the arc data.

arc_list→bbox points to a structure describing a bounding box for all of the arcs in the list. This may be NULL to indicate that no bounding box information is available.

arc_list→arcs is a union of pointers, one of which points to the array of structures that defines each arc in the list. For example, the *arc_list→arcs* structure for multiarcs of type XGL_MULTIARC_I2D looks as follows:

```
typedef struct {
        Xgl_pt_flag_i2d   center;        /* center point */
        Xgl_usgn32        radius;        /* radius of curvature */
        float             start_angle,   /* start and stop angles */
                          stop_angle;
} Xgl_arc_i2d;
```

The structure for multiarcs of type XGL_MULTIARC_F2D is similar except that the data types of the *center* and *radius* members of the structure become *Xgl_pt_flag_f2d* and *float*, respectively.

The *arc_list→arcs* structure for multiarcs of type XGL_MULTIARC_F3D looks as follows:

```
typedef struct {
        Xgl_pt_flag_f3d   center;          /* center point */
        Xgl_pt_f3d        dir[3];          /* dir. vectors and normal vector */
        Xgl_boolean       dir_normalized;  /* TRUE when dir[] are unit vectors */
        Xgl_boolean       dir_normal;      /* TRUE if dir[2] is a valid vector */
        float             radius;          /* radius of curvature */
        float             start_angle,     /* start and stop angles */
                          stop_angle;
} Xgl_arc_f3d;
```

See **xgl_struct**(3) for structures defining other multiarcs in the list.

The first two elements of the field *dir[3]* contain the two orthogonal vectors in model coordinates (MC) used to define the plane on which the arc is going to be built. The third element is used to specify a normal vector for the arc.

dir_normalized should be set to TRUE when the vectors are normalized (that is, unit vectors). *dir_normal* should be set to TRUE when the third element of *dir* is set to a

valid vector. Otherwise, XGL computes a normal vector to the arc, using the two vectors specified for the arc plane.

When specifying type XGL_MULTIARC_AF3D for multiarcs, XGL generates *annotation arcs*. In other words, the arcs generated by XGL are guaranteed to be in a plane parallel to the projection plane. In this particular case, the center of the arc represents the reference point and is specified in model coordinates. The radius is specified in virtual device coordinates (VDC). There is no need to specify any direction vectors, and the primitive is always front-facing.

Execution

As shown above, each arc is specified by an application-defined center point, radius, and start and stop angle. The *center* and *radius* are specified in the application's coordinate system (except in the case of annotation arcs, as shown before). The *start_angle* is the angle from the positive *x*-axis in the plane of the circle to the start point of the arc. The *stop_angle* is the angle from the positive *x*-axis in the plane of the circle to the end point. The angles are specified in radians. The arc traverses a counterclockwise (CCW) path from start angle to stop 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 start and stop angles are defined with these senses, CCW or CW, respectively.

The arcs can be displayed in one of three ''styles,'' depending on the current setting for the attribute **XGL_CTX_ARC_FILL_STYLE**(3)**.** If the arc is closed (that is, if XGL_CTX_ARC_FILL_-STYLE is set to either XGL_ARC_SECTOR or XGL_ARC_CHORD), the following surface-fill attributes control the appearance of the enclosed area of each arc:

XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_FRONT_COLOR(3)

In 3D, the following equivalent attributes for back-facing polygons also apply:

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_COLOR(3)

The lighting and shading attributes are also in effect.

In addition, edges can be drawn around closed arcs. To enable edge drawing, first set the attribute **XGL_CTX_SURF_EDGE_FLAG**(3) to TRUE. Then, edges will be drawn around each arc within the arc list that has set the flag associated with its *center* point. Note that the *center* point of each arc always has a flag associated with it. XGL assumes the application will initialize these flags (to either **0** or **1**) before calling xgl_multiarc. If the flags are not initialized, unpredictable results may occur.

For open arcs (arcs drawn when XGL_CTX_ARC_FILL_STYLE is set to XGL_ARC_OPEN), the line attributes control how the arc is drawn on the Raster. The following attributes are used:

XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_CTX_LINE_CAP(3)

XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_ALT_COLOR(3)

For a more complete summary of these attributes, see the description for **xgl_multipolyline**(3).

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_multipolyline(3)
xgl_object_get(3)
xgl_object_set(3)

XGL_CTX_ARC_FILL_STYLE(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(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_DC_OFFSET(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_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_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NOTES

The *center* point of each arc must have the flag associated with it set to either **0** or **1** (to turn the arc's edge either Off or On). If the flag is not set by the application, unpredictable results may occur.

If the difference between the *start_angle* and *stop_angle* of an arc is greater than 2π radians, then an arc is drawn instead of a circle. For example, if *start_angle* is 0.0 radians and *stop_angle* is $(5*\pi/2)$ radians, then an arc is drawn as if the *stop_angle* were $\pi/2$ radians.

NAME

xgl_multicircle – draws a list of circles

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multicircle (
            Xgl_ctx          ctx,
            Xgl_circle_list  *circle_list );
```

DESCRIPTION

Purpose

This operator draws a list of circles according to the graphics state described in the Context *ctx*. The center and radius of each circle in the list are transformed by the current transformation matrix (see **XGL_CTX_VIEW_TRANS**(3) and **XGL_CTX_MODEL_TRANS**(3)) and drawn in the plane of the view surface.

If the composite view transformation has unequal scaling along the *x* and *y* axes, each circle is drawn as an ellipse, in the case of 2D Contexts. When calling xgl_multicircle with a 3D Context, the circles are ''drawn'' in the plane described by the two direction vectors provided with each circle. The result is then projected using the current set of Transforms, so that all the standard 3D pipeline applies. In the case of 3D circles, the circles are approximated as polygons, using the following context attributes:

XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_multicircle**(3)**.**

circle_list

A pointer to the list of circles to draw. It is a structure defined as:

```
typedef struct {
        Xgl_usgn32              num_circles;   /* number of circles */
        Xgl_multicircle_type    type;          /* data type of multicircle */
        Xgl_bbox                *bbox;         /* optional box contain. all circles */
        union {                                /* union of pointers to circles */
                Xgl_circle_i2d   *i2d;
                Xgl_circle_f2d   *f2d;
                Xgl_circle_d2d   *d2d;
                Xgl_circle_f3d   *f3d;
                Xgl_circle_af3d  *af3d;
                Xgl_circle_d3d   *d3d;
                Xgl_circle_ad3d  *ad3d;
        } circles;
} Xgl_circle_list;
```

circle_list→num_circles is the number of circles in the list.

circle_list→type is the data type of the points used to specify the center and radius

of each circle in the list. Xgl_multicircle_type defines the following enumerated values as possible multicircle data types:

XGL_MULTICIRCLE_I2D
XGL_MULTICIRCLE_F2D
XGL_MULTICIRCLE_D2D
XGL_MULTICIRCLE_F3D
XGL_MULTICIRCLE_AF3D
XGL_MULTICIRCLE_D3D
XGL_MULTICIRCLE_AD3D

Ultimately, this determines which member of the union *circle_list→circles* (see below) points to the circle data.

circle_list→bbox points to a structure describing a bounding box for all of the circles in the list. This may be NULL to indicate that no bounding box information is available.

circle_list→circles is a union of pointers, one of which points to the array of structures that define each circle in the list. For example, the *circle_list→circles* structure for multicircles of type XGL_MULTICIRCLE_I2D looks as follows:

```
typedef struct {
        Xgl_pt_flag_i2d   center;   /* center point of circle */
        Xgl_usgn32        radius;   /* radius of circle */
} Xgl_circle_i2d;
```

The structure for multicircles of type XGL_MULTICIRCLE_F2D is similar except that the data types of the *center* and *radius* members of the structure become *Xgl_pt_flag_f2d* and *float*, respectively.

The *circle_list→circles* structure for multicircles of type XGL_MULTICIRCLE_F3D looks as follows:

```
typedef struct {
        Xgl_pt_flag_f3d   center;           /* center point of circle */
        Xgl_pt_f3d        dir[3];           /* dir. vectors and normal vector */
        Xgl_boolean       dir_normalized;   /* TRUE when dir[] are unit vectors */
        Xgl_boolean       dir_normal;       /* TRUE if dir[2] is a valid vector */
        Xgl_usgn32        radius;           /* radius of circle */
} Xgl_circle_f3d;
```

See **xgl_struct**(3) for structures defining other multicircles in the list.

The first two elements of the field *dir[3]* contain the two vectors in model coordinates (MC), used to define the plane on which the circle is going to be built. The third element is used to specify a normal vector for the circle.

dir_normalized should be set to TRUE when the vectors are normalized (that is, unit vectors). *dir_normal* should be set to TRUE when the third element of *dir* is set to a valid vector. Otherwise, XGL computes a normal vector to the circle, using the two vectors specified for the circle plane.

When specifying type XGL_MULTICIRCLE_AF3D for multicircles, XGL generates *annotation circles*. In other words, the circles generated by XGL are guaranteed to be in a plane parallel to the projection plane. In this case, the center of each circle represents the reference point and is specified in model coordinates. The radius is specified in virtual device coordinates (VDC). There is no need to specify any direction vectors, and the primitive is always front-facing.

In all other cases, the *center* and *radius* are specified in the application's coordinate system.

Execution

Edges can be drawn around the circles. To do so, first set the attribute **XGL_CTX_SURF_EDGE_FLAG**(3) to TRUE. Edges are drawn around each circle within the circle list that has the flag associated with its *center* point set. Note that the *center* point of each circle always has a flag associated with it. XGL assumes the application will initialize these flags (to either **0** or **1**) before calling xgl_multicircle. If the flags are not initialized, unpredictable results may occur.

Circles act like 2D or 3D polygons (see **xgl_polygon**(3)) in that the following surface-fill attributes control the appearance of the enclosed area of each circle:

XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_FRONT_COLOR(3)

In 3D, the following equivalent attributes for back-facing polygons also apply:

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_COLOR(3)

The lighting and shading attributes are also in effect.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(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_DC_OFFSET(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_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)

NOTES

The *center* point of each circle must have the flag associated with it set to either **0** or **1** (to turn the circle's edge either Off or On). If the flag is not initialized by the application, unpredictable results can occur.

NAME

xgl_multimarker – draws a list of Markers

SYNOPSIS

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

DESCRIPTION

Purpose

This operator draws a Marker (a symbol specified by the **XGL_CTX_MARKER**(3) attribute) at each control point in *pl→pts*. The control points in *pl→pts* are transformed and clipped. If a control point is outside the clipping bounds, no Marker is drawn. If the control point is inside the clipping bounds, a Marker is drawn; however, the symbol is clipped at the clipping bounds.

The Marker is drawn in a plane parallel to the plane of the front of the Device. For 3D Contexts, this means its size is not affected by the perspective transformation — Markers at different *z* values have the same size. The size of the Marker is controlled by the attribute **XGL_CTX_MARKER_SCALE_FACTOR**(3)**.** The size (width or height) is the result of the XGL_CTX_MARKER_SCALE_FACTOR multiplied by the nominal Marker size, which is 1 pixel.

The attribute **XGL_CTX_MARKER_COLOR**(3) sets the color of all the Markers in the primitive. If a color is supplied at each point (see the discussion of valid point types below), however, the point color is used instead.

Other attributes that affect the appearance of a Marker are:

XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_COLOR(3)

Marker Definitions

The application can use Markers that it has created (see **XGL_MARKER_DESCRIPTION**(3) for a description of application-defined Markers) or it can use pre-defined Markers. (For a list of pre-defined Markers, see **XGL_CTX_MARKER**(3)**.**)

The Markers are drawn on the Device associated with the Context *ctx*. The Context may be 2D or 3D, and it determines the types of points that can be given to xgl_multimarker.

If the application data associated with this Marker object is a pointer to a floating point scale factor, when rendering through the CGM stream pipeline, the marker will be stroked out and scaled by a factor equal to the product of **XGL_CTX_MARKER_SCALE_FACTOR**(3) and the float pointed to by **XGL_OBJ_APPLICATION_DATA**(3). If there is no object application data associated with the marker, then a marker other than one of the predefined markers will be rendered by the CGM stream pipeline as a dot marker. Note that the **XGL_OBJ_APPLICATION_DATA**(3) scale factor only applies to the CGM stream pipeline. See the *XGL Programmer's Guide* for an example.

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_multimarker**(3)**.**

pl A pointer to an *Xgl_pt_list* structure that 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;
```

A Marker is drawn at each point specified in the array *pl→pts.*

For a complete description of the possible structures, see **xgl_pt_list**(3) and **xgl_struct**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_CTX_MARKER(3)
XGL_CTX_MARKER_AA_BLEND_EQ(3)
XGL_CTX_MARKER_AA_FILTER_SHAPE(3)
XGL_CTX_MARKER_AA_FILTER_WIDTH(3)
XGL_CTX_MARKER_COLOR(3)
XGL_CTX_MARKER_COLOR_SELECTOR(3)
XGL_CTX_MARKER_SCALE_FACTOR(3)

NAME

xgl_multipolyline – draws a list of unconnected polylines

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multipolyline (
        Xgl_ctx       ctx,
        Xgl_bbox      *bounding_box,
        Xgl_usgn32    num_pt_lists,
        Xgl_pt_list   pl[] );
```

DESCRIPTION

Purpose

This operator draws a list of unconnected polylines according to the graphics state described in the Context *ctx*. A polyline is defined by a list of *n* vertices *v(0)*, *v(1)*, ..., *v(n-1)*. The vertices form a series of connected line segments where a line segment is defined by vertex *v(i)* and *v(i+1)*, for *i = 0 to (n-1)*.

Attributes

The line segments are drawn in the style defined by the attribute **XGL_CTX_LINE_STYLE**(3)**.** **XGL_CTX_LINE_PATTERN**(3) is the attribute used to specify a pattern, if the line style needs one. The width of a line is controlled by **XGL_CTX_LINE_WIDTH_SCALE_FACTOR**(3)**.** The scale factor is multiplied by the nominal line width (which is 1 pixel) to arrive at the width of a line segment in device coordinates (DC).

The appearance of the end of a polyline is controlled by **XGL_CTX_LINE_CAP**(3)**.** The attributes **XGL_CTX_LINE_JOIN**(3) and **XGL_CTX_LINE_MITER_LIMIT**(3) control the appearance of the joint between line segments within a polyline.

XGL_CTX_LINE_COLOR(3) controls the color of the polyline, and **XGL_CTX_LINE_ALT_-COLOR**(3) specifies the color used in some patterned lines. If color information is supplied at each vertex of the polyline, however, the line segment from vertex *v(i)* to *v(i+1)* for *i = 0* to *(n-1)* is drawn in the color of vertex *v(i+1)*. For 3D polylines, the attribute **XGL_3D_CTX_-LINE_COLOR_INTERP**(3) controls whether the vertex colors are interpolated. If the polyline has the vertex colors and XGL_3D_CTX_LINE_COLOR_INTERP is FALSE, the line segments are drawn in constant colors as described above. If XGL_3D_CTX_LINE_COLOR_INTERP is TRUE, the color is interpolated from vertex *v(i)* to *v(i+1)*.

Other attributes that affect the appearance of this primitive are:

XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_COLOR(3)

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_multipolyline**(3)**.**

bounding_box
A pointer to a bounding box structure that defines a bounding box for *all* the points in each polyline. Put another way, this is the bounding box of all the individual bounding boxes for each polyline. A bounding box can be used by XGL to

optimize the clipping operations when rendering the primitive. For more information about bounding box usage in XGL, see **XGL_CTX_THRESHOLD**(3)**.** If the application does not want to provide bounding box information, *bounding_box* 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.

num_pt_lists

The number of point lists passed to **xgl_multipolyline**(3)**.** The *pl* element is the first element of an array of *Xgl_pt_list* structures. *num_pt_lists* is the size of the array.

pl[i] 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;
```

Note that all points in *pl[]* must have the same *Xgl_pt_type*.

For a complete description of the possible data structures, see **xgl_pt_list**(3) and **xgl_-struct**(3)**.**

If normals are specified as part of the point list, **xgl_multipolyline**(3) ignores this information.

If points with *flag* information are used, a bit within the flag word at each point determines if specific line segments within the polyline are drawn (1 means draw it). The bit can be manipulated using macros defined in the header file *xgl.h*. XGL_DRAW_EDGE corresponds with the flag bit (0x1 in the flag word). The macro XGL_SET_EDGE_FLAG(PT) can be used to set the flag when given a pointer to a point containing flag information. Likewise, a macro is available to clear the flag bit. For more information, see the include file *xgl.h*. The line segment draw flag is interpreted as follows:

The line segment from *v(i)* to *v(i+1)* is drawn if the XGL_DRAW_EDGE bit (0x1) of the flag *v(i+1)* is set to 1, for *i = 0 to n-1*. The other bits of the flag word are reserved for XGL use and future extensions.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)
xgl_object_set(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_3D_CTX_DEPTH_CUE_COLOR(3)
XGL_CTX_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_3D_CTX_LINE_COLOR_INTERP(3)
XGL_CTX_THRESHOLD(3)

NOTES

All the vertices, in all of the polylines, must be of the same point type.

NAME

xgl_multirectangle – draws a list of rectangles

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_multirectangle (
            Xgl_ctx         ctx,
            Xgl_rect_list   *rect_list );
```

DESCRIPTION

Purpose

This operator draws a list of rectangles according to the graphics state described in the Context *ctx*. Each rectangle in the list is transformed by the current transformation matrix (see **XGL_CTX_VIEW_TRANS**(3) and **XGL_CTX_MODEL_TRANS**(3)) and drawn in the plane of the view surface.

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_multirectangle**(3)**.**

rect_list A pointer to the list of rectangles to draw. It is a structure defined as:

```
typedef struct {
        Xgl_usgn32          num_rects;   /* number of rectangles */
        Xgl_multirect_type  rect_type;   /* data type of multirectangle */
        Xgl_bbox            *bbox;       /* optional box contain. all rectangles */
        union {                          /* union of pointers to rectangles */
                Xgl_rect_i2d    *i2d;
                Xgl_rect_f2d    *f2d;
                Xgl_rect_d2d    *d2d;
                Xgl_rect_f3d    *f3d;
                Xgl_rect_af3d   *af3d;
                Xgl_rect_d3d    *d3d;
                Xgl_rect_ad3d   *ad3d;
        } rects;
} Xgl_rect_list;
```

rect_list→num_rects is the number of rectangles in the list.

rect_list→type is the data type of the points used to specify two diagonal corners for each rectangle in the list. Xgl_multirect_type defines the following enumerated values as possible multirectangle data types:

XGL_MULTIRECT_I2D
XGL_MULTIRECT_F2D
XGL_MULTIRECT_D2D
XGL_MULTIRECT_F3D
XGL_MULTIRECT_AF3D
XGL_MULTIRECT_D3D
XGL_MULTIRECT_AD3D

Ultimately, this determines which member of the union *rect_list→rects* (see below) points to the rectangle data.

rect_list→bbox points to a structure describing a bounding box for all of the rectangles in the list. This may be NULL to indicate that no bounding box information is available.

rect_list→rects is a union of pointers, one of which points to the array of structures that defines each rectangle in the list. For example, the *rect_list→rects* structure for multirectangles of type XGL_MULTIRECT_I2D looks as follows:

```
typedef struct {
        Xgl_pt_flag_i2d   corner_min;   /* diagonal corners of rectangle */
        Xgl_pt_i2d        corner_max;
} Xgl_rect_i2d;
```

The structure for multirectangles of type XGL_MULTIRECT_F2D is similar, except that the data types of the two corner points become *Xgl_pt_flag_f2d* and *Xgl_pt_f2d*, respectively.

The *rect_list→rects* structure for multirectangles of type XGL_MULTIRECT_F3D looks as follows:

```
typedef struct {
        Xgl_pt_flag_f3d   corner_min;       /* reference corner of rectangle */
        Xgl_pt_f2d        corner_max;       /* width and height in MC */
        Xgl_pt_f3d        dir[3];           /* dir. vectors and normal vector */
        Xgl_boolean       dir_normalized;   /* TRUE when dir[] are unit vectors */
        Xgl_boolean       dir_normal;       /* TRUE if dir[2] is a valid vector */
} Xgl_rect_f3d;
```

The first two elements of the field *dir* contain the two vectors in model coordinates (MC), used to define the plane on which the rectangle is going to be built, using *corner_min* as a reference point. The third element is used to specify a normal vector for the rectangle.

Unlike the 2D case, *corner_max* represents the width and height of the rectangle in MC.

dir_normalized should be set to TRUE when the vectors are normalized (that is, unit vectors).

dir_normal should be set to TRUE when the third element of *dir* is set to a valid vector. Otherwise, XGL computes a normal vector to the rectangle, using the two vectors specified for the rectangle plane.

When specifying type XGL_MULTIRECT_AF3D for multirectangles, XGL generates *annotation rectangles*. In other words, the rectangles generated by XGL are guaranteed to be in a plane that is parallel to the projection plane. In this particular case, the reference corner of each rectangle is specified in model coordinates, and the other corner represents the virtual device coordinates (VDC) dimensions in *x* and *y* of the rectangle. There is no need to specify any direction vectors, and the primitive is always front-facing.

Except when rendering annotation rectangles, both corner points are specified in the application's coordinate system with the *corner_min* point defining the corner

at (*x*-min, *y*-min) and the *corner_max* point the one at (*x*-max, *y*-max). For example, in a standard Cartesian coordinate system (*x* increasing to the right and *y* increasing upward), *corner_min* would be the lower left corner of the rectangle, and *corner_max* would be the upper right corner.

Execution

Edges can be drawn around the rectangles. To enable edge drawing, first set the attribute **XGL_CTX_SURF_EDGE_FLAG**(3) to TRUE. Edges will be drawn around each rectangle within the rectangle list that has the flag associated with its *corner_min* point set. Note that the *corner_min* point of each rectangle always has a flag associated with it. XGL assumes the application will initialize these flags (to either **0** or **1**) before calling xgl_multirectangle. If the flags are not initialized, unpredictable results may occur.

Rectangles act like 2D polygons or 3D polygons (see **xgl_polygon**(3)) in that the following surface fill attributes control the appearance of the enclosed area of each rectangle:

XGL_CTX_SURF_FRONT_FILL_STYLE(3)
XGL_CTX_SURF_FRONT_FPAT(3)
XGL_CTX_SURF_FRONT_FPAT_POSITION(3)
XGL_CTX_SURF_FRONT_COLOR(3)

In 3D, the following equivalent attributes for back-facing polygons also apply:

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_COLOR(3)

The lighting and shading attributes are also in effect.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)
xgl_object_set(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_DC_OFFSET(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_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)

NOTES

The *corner_min* point of each rectangle must have the flag associated with it set to either **0** or **1** (to turn the rectangle's edge either Off or On). If the flag is not initialized by the application, unpredictable results can occur.

This operator fills rectangles in an X11-compliant fashion where for filled rectangles, the lower-right row of pixels within the rectangle will not be rendered.

NAME

xgl_nurbs_curve – draws a NURBS curve

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_nurbs_curve (
        Xgl_ctx                 ctx,
        Xgl_nurbs_curve         *curve,
        Xgl_bounds_f1d          *range,  /* optional */
        Xgl_curve_color_spline  *color_spline );  /* optional */
```

DESCRIPTION

This operator generates a NURBS curve of specified order based on the list of knots in the parameter space, list of control points, and the parametric range. Either rational or non-rational curve is selected, depending on the point type.

Parameters

ctx The XGL 2D and 3D Context containing the graphics state used when processing **xgl_nurbs_curve**(3)**.**

curve A pointer to an *Xgl_nurbs_curve* structure that is defined as follows:

```
typedef struct {
        Xgl_usgn32  order;          /* Order of curve */
        Xgl_usgn32  num_knots;      /* Number of knots */
        float       *knot_vector;   /* Knot vector */
        Xgl_pt_list ctrl_pts;       /* List of control points */
} Xgl_nurbs_curve;
```

curve→order must be a positive integer.

curve→num_knots is the number of knots and it must be equal to the number of control points plus the order of the spline.

curve→knot_vector is an array of floats which must form a non-decreasing sequence.

curve→ctrl_pts is an array of points. The weight of a rational control point cannot be less than or equal to zero.

range The range parameters define the parametric limits of the displayed curve.

The *Xgl_bounds_f1d* structure is defined as follows:

```
typedef struct {
        float   bmin;
        float   bmax;
} Xgl_bounds_f1d;
```

The minimum value should not be greater than the maximum value, and both values should lie between *curve→knot_vector[order-1]* and *curve→knot_vector[curve→num_knots-order]*. If the range parameter is NULL, the whole curve is displayed.

color_spline

A NURBS curve which describes the color distribution over the curve's geometry. Color spline is supported only in 3D and the parameter is ignored in 2D. It is a pointer to an *Xgl_curve_color_spline* structure that is defined as follows:

```
typedef struct {
        Xgl_usgn32              order;
        Xgl_usgn32              num_knots;
        float                   *knot_vector;
        Xgl_color_homogeneous   *colors;
} Xgl_curve_color_spline;
```

The order, num_knots, and knot_vector elements should meet the same requirement as structure *Xgl_nurbs_curve* above. Color splines need not have the same order, rationality or knot sequence as the geometry spline.

The control points of this color spline are color coordinates. The color coordinates can be in 3D (non-rational) or 3D+w (rational). The values of the color spline control point coordinates may be outside the range normally associated with the color value. The vertex color for any point on the mapped geometry of the curve can be determined by evaluating the color spline at the corresponding parameter value.

The *Xgl_color_homogeneous* union is defined as follows:

```
typedef union {
        Xgl_color_rgbw    rgbw;
} Xgl_color_homogeneous;
```

It is a union of all color types (except index) with w coordinate.

The *Xgl_color_rgbw* structure is defined as follows:

```
typedef struct {
        float    r, g, b;
        float    w;
} Xgl_color_rgbw;
```

If the non-rational color is used, w should be set to 1. The color type has to be consistent with the device color type and should not be indexed.

All attributes that affect line-rendering primitives also apply to this primitive.

The quality of smoothness of the curve can be controlled with the **XGL_CTX_NURBS_CURVE_APPROX**(3) and **XGL_CTX_NURBS_CURVE_APPROX_VAL**(3) attributes. The former selects the approximation method, and the latter selects the reference measure used in the approximation of the curve. The minimum and the maximum number of segments used to display the curve can be controlled with **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes.

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_CTX_NURBS_CURVE_APPROX(3)
XGL_CTX_NURBS_CURVE_APPROX_VAL(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(3)
XGL_CTX_LINE_AA_BLEND_EQ(3)
XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)

NOTES

The relationship between supported orders and curve approximation methods for non-rational curves is as follows:

Order	**Supported Approximation Method**
1-15	All methods supported
16-25	XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

The relationship between supported orders and curve approximation methods for rational curves is as follows:

Order	**Supported Approximation Method**
1-8	All methods supported
9-25	XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

Curves of order greater than 25 are not supported. Also, rendering of curves of high order (greater than 6) is likely to be very slow.

XGL currently supports the following point types in xgl_nurbs_curve operator:

Xgl_pt_i2d, Xgl_pt_f2d, Xgl_pt_i2h, Xgl_pt_f2h, Xgl_pt_f3d, Xgl_pt_f3h.

NAME

xgl_nurbs_surface – draws a NURBS surface

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_nurbs_surface (
        Xgl_ctx                     ctx,
        Xgl_nurbs_surf              *surface,
        Xgl_trim_loop_list          *trimming,
        Xgl_nurbs_surf_simple_geom  *hints,  /* optional */
        Xgl_surf_color_spline       *color_spline,  /* optional */
        Xgl_surf_data_spline_list   *data_splines );  /* not supported */
```

DESCRIPTION

This operator generates a NURBS surface of specified order based on the list of knots in the parameter space, list of control points, and the trimming information. Either rational or non-rational surface is selected, depending on the point type.

Parameters

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

surface A pointer to an *Xgl_nurbs_surf* structure that is defined as follows:

```
typedef struct {
        Xgl_usgn32  order_u;        /* Order of surface in u */
        Xgl_usgn32  order_v;        /* Order of surface in v */
        Xgl_usgn32  num_knots_u;    /* Number of knots in u */
        float       *knot_vector_u; /* Knot vector in u */
        Xgl_usgn32  num_knots_v;    /* Number of knots in v */
        float       *knot_vector_v; /* Knot vector in v */
        Xgl_pt_list ctrl_pts;       /* Row-ordered list of control
                                       points (index changes faster
                                       in u direction than v). */
} Xgl_nurbs_surf;
```

surface→order must be a positive integer.

surface→num_knots is the number of knots. In both U and V, the number of knots must be equal to the corresponding number of control points plus the order of the spline. For example, the number of knots in U is calculated as follows:

ctrl_pts_u + order_u

surface→knot_vector is an array of floats. In both U and V, the knot vector must form a non-decreasing sequence.

surface→ctrl_pts is an array of points. The weight of a rational control point cannot be less than or equal to zero. The number of control points is the product of the number of points in U and the number of points in V.

*(num_knots_u - order_u) * (num_knots_v - order_v)*

trimming

The trimming parameters define the trimmed portion of the displayed surface. If trimming is NULL, the whole surface between its U and V parametric ranges is displayed. Trimming information consists of a list of trim loops. Each trim loop consists of a list of trim curves, connected to each other in a head to tail fashion.

Xgl_trim_loop_list structure is defined as follows:

```
typedef struct {
        Xgl_usgn32      num_loops;
        Xgl_trim_loop   *trim_loops;
} Xgl_trim_loop_list;
```

list→num_loops is the number of trim loops.

list→trim_loops is a list of trim curves. It is a structure defined as:

```
typedef struct {
        Xgl_usgn32      num_curves;
        Xgl_trim_curve  *curves;
} Xgl_trim_loop;
```

loop→num_curves is the number of trim curves in a loop.

loop→curves is an array of 2D NURBS curves, defined in the parameter space of the surface.

Each trim loop must be closed, continuous and should not intersect itself or another trim loop. Moreover, valid trim loops must satisfy the odd winding rule and curve orientation rule. For more information about trimming, please refer to the *XGL Programmer's Guide*.

Xgl_trim_curve structure is defined as follows:

```
typedef struct {
        Xgl_usgn32              order;                  /* Order of curve */
        Xgl_usgn32              num_knots;              /* Number of knots */
        float                   *knot_vector;           /* Knot vector */
        Xgl_bounds_f1d          range;                  /* Range of parameter */
        Xgl_pt_list             ctrl_pts;               /* List of control points */
/*
 * Trim curve structure fields (applicable only to trim curves)
 */
        Xgl_boolean             trim_curve_vis;         /* Highlight trim curve */
        Xgl_trim_curve_approx   trim_curve_approx;      /* Approx type */
        float                   trim_curve_approx_value; /* Approx value */
} Xgl_trim_curve;
```

curve→order must be greater than or equal to 2.

curve→num_knots is the number of knots which must be equal to the corresponding number of control points plus the order of the trim curve.

curve→knot_vector is an array of floats. The knots must form a non-decreasing sequence.

curve→range is the range parameters defining the parametric limits of the trim curve.

Xgl_bounds_f1d structure is defined as follows:

```
typedef struct {
        float   bmin;
        float   bmax;
} Xgl_bounds_f1d;
```

The minimum value should not be greater than the maximum value, and both values should lie between *curve→knot_vector[order-1]* and *curve→knot_vector-[curve→num_knots-order]*.

curve→ctrl_pts is an array of points. Trim curve can be rational or non-rational, depending on the point type. The weight of a rational control point cannot be less than or equal to zero.

curve→trim_curve_vis controls the highlighting of the trim curve. If set, the trim curve will be highlighted using current edge attributes.

curve→trim_curve_approx and *trim_curve_approx_value* controls the quality of smoothness of a trim curve. The former selects the approximation method, and the latter selects the reference measure used in the approximation of the curve. If the Xgl_trim_curve_approx is XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_-KNOTS, trim curve is rendered with specified (by trim_curve_approx_value) number of segments between each adjacent pair of knots. If the Xgl_trim_-curve_approx is XGL_TRIM_CURVE_VIEW_DEPENDENT, trim curve tessellation is controlled by the tessellation of the surface. In this case, a value of *curve→trim_-curve_approx_value* between 0 and 1 indicates the relative quality of the trim curves compared to surface quality. Higher values mean better relative quality.

Xgl_trim_curve_approx structure is defined as:

```
typedef enum {
        XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS,    /* static tessellation */
        XGL_TRIM_CURVE_VIEW_DEPENDENT                       /* dynamic tessellation */
} Xgl_trim_curve_approx;
```

hints An application may optionally supply hints about the shape of the surface. If the surface is planar, spherical, cylindrical, or conical, XGL can optimize the rendering. *Xgl_nurbs_surf_simple_geom* structure is defined as:

```
typedef struct {
        Xgl_nurbs_surf_type         surf_type;
        Xgl_nurbs_surf_geom_desc    geom_desc;
} Xgl_nurbs_surf_simple_geom;
```

The *geom→surf_type* structure for different nurbs surfaces is as follows:

```
typedef enum {
        XGL_SURF_NURBS,             /* default, no optimization */
        XGL_SURF_PLANAR,            /* surface is planar */
        XGL_SURF_CYLINDRICAL,       /* surface is cylindrical */
        XGL_SURF_CONICAL,           /* surface is conical */
        XGL_SURF_SPHERICAL,         /* surface is spherical */
} Xgl_nurbs_surf_type;
```

The *geom→geom_desc* structure for nurbs surface geometry is as follows:

```
typedef union {
   Xgl_plane        planar;
   struct {
      Xgl_pt_f3d    axial_pt;
      Xgl_pt_f3d    axis_dir;
      float         radius;
      Xgl_boolean   norm_flag;
   } cylindrical;
   struct {
      Xgl_pt_f3d    apex;
      Xgl_pt_f3d    axis_dir;
      float         cone_angle; /* 0 = zero width cone (line)*/
      Xgl_boolean   norm_flag;
   } conical;
   struct {
      Xgl_pt_f3d    center;
      float         radius;
      Xgl_boolean   norm_flag;
   } spherical;
} Xgl_nurbs_surf_geom_desc;
```

A planar surface is defined by a point on it and the normal. A cylindrical surface is defined by an axis, a point on the axis, and a radius. A conical surface is defined by its apex, axis, and an angle between the axis and the side. Finally, a spherical surface is defined by its center and a radius. The norm_flag for cylindrical, conical, and spherical surfaces indicates whether normals to the surface point outwards (TRUE) or inwards (FALSE).

All the NURBS parameters to xgl_nurbs_surface operator must always be provided even though simple geometry hints is specified. It is the application's responsibility to ensure that provided hint does not conflict with the geometry defined by the NURBS parameters. If this is not the case, lighting may be incorrect.

color_spline

The color_spline is a nurbs surface which describes the color distribution over the primitive's geometry. It is a pointer to an *Xgl_surf_color_spline* structure that is defined as follows:

```
typedef struct {
        Xgl_usgn32              order_u;
        Xgl_usgn32              order_v;
        Xgl_usgn32              num_knots_u;
        float                   *knot_vector_u;
        Xgl_usgn32              num_knots_v;
        float                   *knot_vector_v;
        Xgl_color_homogeneous   *colors;
} Xgl_surf_color_spline;
```

The order, num_knots, and knot_vector elements should meet the same requirement as structure *Xgl_nurbs_surface* above. Color splines need not have the same order, rationality or knot sequence as the geometry spline.

spline→colors is an array of control points of the color spline. The number of control points is calculated in the same way as the number of control points (*surface→ctrl_pts*) discussed above.

The control points of the color spline are color coordinates. The color coordinates could be in 3D (non-rational) or 3D+w (rational). The values of the color spline control point coordinates may be outside the range normally associated with the color value. The vertex color for any point on the mapped geometry of the surface can be determined by evaluating the color spline at the corresponding parameter value.

Xgl_color_homogeneous union is defined as follows:

```
typedef union {
        Xgl_color_rgbw    rgbw;
} Xgl_color_homogeneous;
```

It is a union of all color types (except index) with w coordinate.

Xgl_color_rgbw structure is defined as follows:

```
typedef struct {
        float    r, g, b;
        float    w;
} Xgl_color_rgbw;
```

If the non-rational color is used, w should be set to 1. The color type has to be consistent with the device color type and should not be indexed.

Attributes

The quality of smoothness of the surface can be controlled with the **XGL_CTX_NURBS_SURF_APPROX**(3), **XGL_CTX_NURBS_SURF_APPROX_VAL_U**(3) and **XGL_CTX_NURBS_SURF_APPROX_VAL_V**(3) attributes. The first selects the approximation method, and the remaining two selects the reference measures used in the approximation of the surface. The minimum and the maximum number of segments used to display the surface can be controlled with **XGL_CTX_MIN_TESSELLATION**(3) and **XGL_CTX_MAX_TESSELLATION**(3) attributes.

The trim loops constitute the edges of the B-spline. If the B-spline is untrimmed, the parametric boundaries constitute the edges. Edges are rendered using current edge attributes. Individual trim curves may be turned on or off by using the *curve→trim_curve_vis* flag.

The **XGL_3D_CTX_SURF_FRONT_FILL_STYLE**(3) and the **XGL_3D_CTX_SURF_BACK_FILL_STYLE**(3) attributes control the appearance of the interior surface. If it is XGL_SURF_FILL_SOLID, the interior is filled entirely. If it is XGL_SURF_FILL_HOLLOW, the surface is rendered as lines along the edges. These lines have the surface color and may be shaded. If it is XGL_SURF_FILL_EMPTY, then the interior is not drawn.

Optionally, iso-parametric curves and level curves (currently not implemented) may be rendered on top of the surface. Curves are rendered using current line attributes. Also, edges of approximating facets may be displayed using current edge attributes. The **XGL_CTX_NURBS_SURF_PARAM_STYLE**(3) attribute controls the appearance of the surface.

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_CTX_NURBS_SURF_APPROX(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_U(3)
XGL_CTX_NURBS_SURF_APPROX_VAL_V(3)
XGL_CTX_NURBS_SURF_PARAM_STYLE(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM(3)
XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM(3)
XGL_CTX_MAX_TESSELLATION(3)
XGL_CTX_MIN_TESSELLATION(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_LINE_AA_BLEND_EQ(3)

XGL_CTX_LINE_AA_FILTER_SHAPE(3)
XGL_CTX_LINE_AA_FILTER_WIDTH(3)
XGL_CTX_LINE_ALT_COLOR(3)
XGL_CTX_LINE_CAP(3)
XGL_CTX_LINE_COLOR(3)
XGL_CTX_LINE_COLOR_SELECTOR(3)
XGL_CTX_LINE_JOIN(3)
XGL_CTX_LINE_MITER_LIMIT(3)
XGL_CTX_LINE_PATTERN(3)
XGL_CTX_LINE_STYLE(3)
XGL_CTX_LINE_WIDTH_SCALE_FACTOR(3)
XGL_3D_CTX_LINE_COLOR_INTERP(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_DC_OFFSET(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_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)

NOTES

The relationship between supported orders and surface approximation methods for non-rational surfaces is as follows:

Order	**Supported Approximation Method**
1-10	All methods supported
11-25	XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

Non-rational surfaces of order greater than 25 are not supported.

The relationship between supported orders and surface approximation methods for rational surfaces is as follows:

Order	**Supported Approximation Method**
1-6	All methods supported
7-10	XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

Rational surfaces of order greater than 10 are not supported.

The relationship between supported orders and trim curve approximation methods for non-rational trim curves is as follows:

Order	**Supported Approximation Method**
2-10	All methods supported
11-15	XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

The relationship between supported orders and trim curve approximation methods for rational trim curves is as follows:

Order	**Supported Approximation Method**
2-6	All methods supported
7-15	XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS only

Trim curves of order greater than 15 are not supported.

Rendering of surfaces of high U, V, or trim curve orders (greater than 6) is likely to be very slow.

XGL currently supports the following point types in xgl_nurbs_surface operator:

Xgl_pt_f3d, Xgl_pt_f3h.

NAME

xgl_object_create – creates an XGL Object

SYNOPSIS

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

Xgl_object    xgl_object_create (
                  Xgl_sys_state      sys_state,
                  Xgl_obj_type       type,
                  Xgl_obj_desc       *desc,
                  <attribute-list>   attributes );
```

DESCRIPTION

Purpose

This operator creates a new XGL Object of type defined by *type* and initializes it with the settings given by the specified attributes in the *attributes* list. **xgl_object_create**(3) is used to create any XGL Object, except the system state, which must be created using **xgl_open**(3)**.**

When an Object is no longer needed in an application, it can be explicitly destroyed by calling the **xgl_object_destroy**(3) operator to free the resources associated with it.

Parameters

sys_state

The handle to system state information for this XGL session. Use the value returned by the call to **xgl_open**(3)**.**

For more detailed information about this value, see **XGL_SYS_STATE**(3) and its corresponding man pages.

type,desc

type specifies the type of Object to create, and *desc* is a descriptor pointing to additional information required at Object creation time. *desc* is needed only in a few instances as specified below. When it is not needed, the application should pass NULL as the value for it. Xgl_obj_desc is a union defined as:

```
typedef union {
  struct {
    Xgl_window_type     type;        /* window type */
    void                *desc;
  } win_ras;
  char                  *sfont_name;
  struct {
    Xgl_object          raster;
  } accum_buf;
  struct {
    char *name;
    void *desc;
  } stream;
} Xgl_obj_desc;
```

type can take the following values. For more detailed information about these values, see their corresponding man pages.

XGL_2D_CTX(3)
Creates a 2D Context object. The parameter *desc* is ignored.

XGL_3D_CTX(3)
Creates a 3D Context object. The parameter *desc* is ignored.

XGL_CGM_DEV(3)
Creates a CGM Metafile Device object. The parameter *desc* is ignored.

XGL_CMAP(3)
Creates a Color Map object. The parameter *desc* is ignored.

XGL_DMAP_TEXTURE(3)
Creates a Data Map Texture object. The parameter *desc* is ignored.

XGL_GCACHE(3)
Creates a Geometry Cache (Gcache) object. The parameter *desc* is ignored.

XGL_LIGHT(3)
Creates a Light object. The parameter *desc* is ignored.

XGL_LPAT(3)
Creates a Line Pattern object. The parameter *desc* is ignored.

XGL_MARKER(3)
Creates a user defined Marker object. The parameter *desc* is ignored.

XGL_MEM_RAS(3)
Creates a Memory Raster Device object. The parameter *desc* is ignored.

XGL_MIPMAP_TEXTURE(3)
Creates a MipMap Texture object. The parameter *desc* is ignored.

XGL_PCACHE(3)
Creates a Primitive Cache (Pcache) object. The parameter *desc* is ignored.

XGL_SFONT(3)
Creates a Stroke Font object. The *sfont_name* field of the *desc* parameter is set to point to the name of the font.

XGL_STREAM(3)
Creates a Stream Device object. The *desc* parameter is a pointer to an *Xgl_obj_desc* union. The application needs to supply the information in the *stream* field of the *Xgl_obj_desc* union. The *stream* structure consists of two fields: *name*, which is the name identifying the specific device pipeline, and *desc*, which is a pointer to device specific information that may be required by this particular device pipeline.

XGL_TMAP(3)
Creates a Texture Map object. The parameter *desc* is ignored.

XGL_TRANS(3)

Creates a Transform object. The parameter *desc* is ignored.

XGL_WIN_RAS(3)
Creates a Window Raster Device object. The parameter *desc* contains a window type, which can be one of the Xgl_window_type enumerated types (see **xgl_enum_types**(3)), and a window descriptor whose contents depend on the window type.

attributes

A *NULL-terminated* list of attribute-value pairs that describe the initial state of the Object. For each pair, the attribute name is followed by a value. For example,

```
xgl_object_create(sys_state, type, desc, att1, v1, att2, v2, NULL);
```

sets att1 to v1 and att2 to v2 as part of the Object creation process. Those Object attributes not specified as parameters to the create operator are initialized to default values.

After creating the Object, the values of its attributes can be inquired by the application using the **xgl_object_get**(3) operator.

RETURN VALUE

Xgl_object – A handle to an XGL Object, if successful.
NULL – If unsuccessful.

SEE ALSO

xgl_object_destroy(3)
xgl_object_get(3)
xgl_object_set(3)
xgl_open(3)
xgl_enum_types(3)

XGL_2D_CTX(3)
XGL_3D_CTX(3)
XGL_CGM_DEV(3)
XGL_CMAP(3)
XGL_DMAP_TEXTURE(3)
XGL_GCACHE(3)
XGL_LIGHT(3)
XGL_LPAT(3)
XGL_MARKER(3)
XGL_MEM_RAS(3)
XGL_MIPMAP_TEXTURE(3)
XGL_PCACHE(3)
XGL_SFONT(3)
XGL_SYS_STATE(3)
XGL_STREAM(3)
XGL_TMAP(3)
XGL_TRANS(3)
XGL_WIN_RAS(3)

NOTES

A System State object must have been created with the **xgl_open**(3) operator before any other XGL Object can be created.

NAME

xgl_object_destroy – destroys an XGL Object

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_object_destroy (
            Xgl_object    handle );
```

DESCRIPTION

This operator destroys the specified XGL Object by freeing all resources associated with it. The Object is identified by the Object *handle*, which is passed to this operator as a parameter. The application can destroy only Objects that it has specifically created.

XGL may delay the actual destruction of an Object if it is still referenced by other Objects.

It is a good idea to destroy XGL Objects no longer used in a given application. This gives the system an opportunity to reutilize all freed resources (mainly memory) for new Objects. It also keeps the different internal lists used by XGL to a minimum size, thus improving performance in referencing Objects.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

NAME

xgl_object_get – returns the value of an attribute

SYNOPSIS

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

void  xgl_object_get (
            Xgl_object      obj,
            Xgl_attribute   attr,
            void            *val );
```

DESCRIPTION

This operator returns the value of the attribute *attr* associated with the XGL Object *obj*. The parameter *val* is a pointer to a memory location into which XGL stores the retrieved attribute's value. The application program is responsible for ensuring that *val* points to a memory area that is large enough to store the attribute's value. The application also must allocate the memory area. The man page for each attribute describes the data type that should be passed to xgl_object_get to get each attribute.

The operator **xgl_object_set**(3) is used to assign a new value to an attribute.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_set(3)

NAME

xgl_object_set – assigns a new value to an attribute

SYNOPSIS

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

void  xgl_object_set (
            Xgl_object          obj,
            <attribute-list>    attributes );
```

DESCRIPTION

This operator assigns a given value to an attribute (or list of attributes) in the XGL Object *obj*. Multiple attributes can be updated with a single call to xgl_object_set.

The parameter type *<attribute-list>* is used to indicate a *NULL-terminated* list of attribute-value pairs. For each pair, the attribute name is followed by the given value. For example,

xgl_object_set(obj, attribute1, value1, attribute2, value2, NULL);

sets *attribute1* to *value1* and *attribute2* to *value2* for the object *obj*.

The man page for each attribute describes the data type of the attribute value that is passed to the xgl_object_set operator.

If **XGL_SYS_ST_ERROR_DETECTION**(3) is TRUE, the attribute-value list is scanned for invalid values. If an error is found, it is reported and *none* of the attributes is set to the new value. The application must ensure that all attributes have valid values.

The operator **xgl_object_get**(3) stores the value of a single attribute into an application-specified block of memory.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_get(3)

XGL_SYS_ST_ERROR_DETECTION(3)

NAME

xgl_open – opens XGL for use

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_sys_state     xgl_open (
                      <attribute-list>    attributes );
```

DESCRIPTION

This operator is used to begin an XGL session. A single System State object is created and initialized when **xgl_open**(3) is invoked. If successful, a handle to the System State object is returned. This object can only be destroyed by a subsequent call to **xgl_close**(3) using the specified handle.

Only one XGL System State per process is permitted at any given time.

The application programmer can create a System State and set its attributes in the same call by providing the correct arguments in the list of attributes for xgl_open. The attributes of the System State are as follows:

The application program can set the *error detection flag* and the *error notification function* to control and filter errors that are generated during an XGL session. For more details, see the attributes **XGL_SYS_ST_ERROR_DETECTION**(3)**,** and **XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION**(3)**.**

The **XGL_SYS_ST_SFONT_DIRECTORY**(3) attribute gives the user the ability to specify a different path for the *stroke fonts directory* if the stroke fonts are not installed as specified by the default procedures.

Finally, the read-only attribute **XGL_SYS_ST_VERSION**(3) can be used to determine which *version* of XGL the application is currently using. This is useful for applications that use different versions of XGL.

The xgl_open operator must be called before creating any other XGL object with **xgl_object_create**(3)**.**

RETURN VALUE

Xgl_sys_state – A handle to the System State object, if successful.
NULL – If unsuccessful.

SEE ALSO

xgl_close(3)
xgl_object_create(3)
xgl_object_destroy(3)
xgl_object_get(3)
xgl_object_set(3)
XGL_SYS_ST_ERROR_DETECTION(3)
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION(3)
XGL_SYS_ST_SFONT_DIRECTORY(3)
XGL_SYS_ST_VERSION(3)

NAME

xgl_pcache_display – displays a Primitive Cache

SYNOPSIS

```
#include <xgl/xgl.h>
Xgl_cache_display xgl_pcache_display (
    Xgl_pcache    pcache,
    Xgl_boolean   test,
    Xgl_boolean   display,
    Xgl_boolean   restore);
```

DESCRIPTION

This operator performs the following tasks: 1) it closes the *pcache*, 2) it compares *pcache* and context states, 3) if the comparison is OK, it displays the *pcache*, and 4) it returns the result of the comparison.

Parameters

pcache Specifies the *pcache* to operate upon. The **XGL_PCACHE_CONTEXT**(3) attribute of the *pcache* specifies the context to be used for comparison and display.

test Controls the validation of the *pcache* and context states. If test is FALSE, the context state is not validated and the return value is XGL_CACHE_NOT_CHECKED. If test is TRUE, the context state is compared with that of the *pcache*; if the state is compatible, the return value is XGL_CACHE_DISPLAY_OK, otherwise it is XGL_CACHE_ATTR_STATE_DIFFERENT.

display Controls the display of the *pcache*. If display is FALSE, the *pcache* is not rendered. If display is TRUE, the *pcache* is rendered only if test is FALSE or the context state is compatible.

restore Controls whether or not context attributes can be altered by displaying *pcache*. If restore is TRUE, displaying a *pcache* will not change any of the context attributes. If restore is FALSE and the *pcache* contains attributes, the attributes of the context are changed to reflect those stored in the *pcache*.

RETURN VALUE

Returns the status of the validity test:

XGL_CACHE_DISPLAY_OK

The states in Pcache and Context are compatible.

XGL_CACHE_ATTR_STATE_DIFFERENT

The states in Pcache and Context are not compatible.

XGL_CACHE_NOT_CHECKED

The state in Pcache is not checked against the state in the Context.

SEE ALSO

xgl_object_create(3)

XGL_PCACHE(3)

XGL_PCACHE_CONTEXT(3)

NOTES

In the special case of an unclosed, empty Pcache, **xgl_pcache_display**(3) will return XGL_CACHE_ATTR_STATE_DIFFERENT if called with test = TRUE. This feature allows an application to omit the initial build of a Pcache object, since it can be (re)built the first time the application attempts to display it.

NAME

xgl_pick_clear – clears the pick buffer that stores pick events

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_pick_clear (
            Xgl_ctx     ctx );
```

DESCRIPTION

This operator clears the *pick buffer* associated with the Context *ctx* to the *nothing-picked* state. The Context *pick buffer* stores information about pick events. Its size is determined by the attribute **XGL_CTX_PICK_BUFFER_SIZE**(3)**.** Each time a primitive, or any part of it, is rendered into the pick aperture specified by the attribute **XGL_CTX_PICK_APERTURE**(3)**,** information about the *pick event* is stored in the buffer. The operator **xgl_pick_get_-identifiers**(3) is used to retrieve this information about the picked primitives. (For more information, see **XGL_CTX_PICK_ID_1**(3)**,** **XGL_CTX_PICK_ID_2**(3)**,** and **XGL_CTX_PICK_STYLE**(3)**.)**

The attribute **XGL_CTX_PICK_ENABLE**(3) must be TRUE for picking to occur.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_pick_get_identifiers(3)

XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)

NAME

xgl_pick_get_identifiers – returns the identifiers of picked primitives and clears the pick buffer

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_pick_get_identifiers (
            Xgl_ctx          ctx,
            Xgl_usgn32       *count,
            Xgl_pick_info    pick_id_list[] );
```

DESCRIPTION

This operator returns the identifiers (IDs) of picked primitives for the Context *ctx* and clears the pick buffer that stores pick events for this context.

The returned pick IDs are stored in the array *pick_id_list*. The number of pick IDs returned is stored at address count. The application must allocate *pick_id_list* to hold as many *Xgl_pick_info* structures as are in the Context pick buffer. Use the attribute **XGL_CTX_PICK_BUFFER_SIZE**(3) to both specify and inquire the size of the pick buffer.

The order in which XGL stores the pick IDs in the *pick_id_list* depends on **XGL_CTX_PICK_STYLE**(3)**.** If it is XGL_PICK_FIRST_N, XGL stores the first pick event at *pick_id_list[0]* and the last pick event recorded at *pick_id_list[*count –1]*. If it is XGL_PICK_LAST_N, XGL stores the last pick event at *pick_id_list[0]* and the earliest recorded pick event at *pick_id_list[*count –1]*. The number of pick events that occur may exceed the number of pick events recorded in the *pick_id_list*.

The value of the field *vertex_flag* returned in the *Xgl_pick_info* structure is undefined.

The attribute **XGL_CTX_PICK_ENABLE**(3) must be TRUE for picking to occur (that is, for anything to be stored in the pick buffer).

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_pick_clear(3)

XGL_CTX_PICK_APERTURE(3)
XGL_CTX_PICK_BUFFER_SIZE(3)
XGL_CTX_PICK_ENABLE(3)
XGL_CTX_PICK_ID_1(3)
XGL_CTX_PICK_ID_2(3)
XGL_CTX_PICK_STYLE(3)

NAME

xgl_polygon – draws a general-case polygon

SYNOPSIS

```
#include <xgl/xgl.h>
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[] );
```

DESCRIPTION

Purpose

This operator draws a single polygon according to the parameter list and the graphics state described in the Context *ctx*, which can be either a 2D or 3D Context.

A *general* polygon is defined by one or more point lists. Each point list defines a sequence of connected edges that together form an individual boundary. These boundaries, along with the filling rule specified by the **XGL_CTX_SURF_INTERIOR_RULE**(3) attribute, define the interior of the polygon.

For the boundary consisting of the *n* vertices *v(0)*, *v(1)*, ..., *v(n-1)*, the edges defined are those between vertices *v(0)* to *v(1)*, *v(1)* to *v(2)*, ..., *v(n-2)* to *v(n-1)*, and *v(n-1)* to *v(0)*. The resulting boundary can be self-intersecting or non-planar. The exact appearance of non-planar polygons when projected on the viewing surface is hardware-dependent.

Parameters

ctx The XGL 2D or 3D Context containing the graphics state used when processing **xgl_polygon**(3)**.**

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

facet A pointer to an *Xgl_facet* structure, which is defined as:

```
typedef union {
    Xgl_color_facet          color_facet;
    Xgl_normal_facet         normal_facet;
    Xgl_color_normal_facet   color_normal_facet;
} Xgl_facet;
```

The value of *facet_type* determines the interpretation of the union pointed to by *facet*. For example, if *facet_type* is XGL_FACET_COLOR_NORMAL, *facet→color_normal_facet* will contain the facet data for the polygon. If *facet_type* is XGL_FACET_NONE, *facet* can be NULL.

bbox Either NULL or is the bounding box around all of the points in *pl[]*. 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.

num_pt_lists
The number of elements in *pl[]*.

pl A pointer to the first element of an array of *Xgl_pt_list* structures, 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;
```

Each *pl[i]* specifies the vertices of a single boundary of the polygon.

pl[i].pt_type specifies the type of the points in *pl[i].pts*. Note that all of the *types* in the list pointed to by *pts* must be of this type.

pl[i].bbox specifies a bounding box for the *i'th* boundary. This bounding box is handled in the same way as *bbox* (above) is handled.

pl[i].num_pts is the number of points in the array *pts*. Specifically, it is the number of vertices in the *i'th* polygon boundary.

pl[i].num_data_values specifies the number of data values. Specifically, it is the number of data values per vertex in the *i'th* polygon boundary. This value is looked into only if the *pl[i].pt_type* is specified as having data values.

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 on a polygon. The coordinate space of the texture map that *u* values correspond to is a space that places 0.0 at left edge of the texture and 1.0 at the right edge of the texture. Similarly for *v*, the coordinate space of the texture map that *v* values use is a space that places 0.0 at the bottom edge of the texture and 1.0 at the top edge of the texture. An application can give several data values and can choose the specific data values to be used from the array by using the attributes **XGL_DMAP_TEXTURE_U_INDEX**(3) and **XGL_DMAP_TEXTURE_V_INDEX**(3)**.**

The data values are used only if the attributes **XGL_3D_CTX_SURF_FRONT_DMAP_-SWITCHES**(3) and **XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES**(3) have at least one switch set to TRUE.

pl[i].pts is a union of pointers, one of which points to the array of points that make up the vertices of the *i'th* boundary. The value of *pl[i].pt_type* determines the interpretation of this union. Refer to the **xgl_pt_list**(3) man page for the valid point types.

If vertices with *flag* information are used, a bit within the flag word at each point determines if specific polygon boundary edges are drawn (1 means draw it). The bit can be manipulated using macros defined in the header file *xgl.h*. XGL_DRAW_-EDGE corresponds with the flag bit (0x1 in the flag word). The macro XGL_SET_-EDGE_FLAG(PT) can be used to set the flag when given a pointer to a point containing flag information. Likewise, a macro is available to clear the flag bit. For more information, see the include file *xgl.h*.

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

The XGL_DRAW_EDGE bit (0x1) is used to determine whether particular boundary edges are drawn. For the boundary vertices *v(0), v(1), ..., v(n)*, the flag on vertex *v(i)* determines whether the edge between vertices *v(i-1)* and *v(i)* is drawn. The flag on vertex *v(0)* applies to the edge between vertices *v(n)* and *v(0)*.

FACET RENDERING

Generally, a *facet* refers to a single polygon, which might be either a general or simple polygon. Several Context attributes specify how this facet is to be rendered. The vertex point type *pl[i].pt_type* and the facet type *facet_type* can provide sources of color in addition to the surface color Context attributes **XGL_CTX_SURF_FRONT_COLOR**(3) and **XGL_3D_-CTX_SURF_BACK_COLOR**(3)**.**

When an application supplies more than one source of colors for a polygon, it can select the source — Context, facet, or vertex — to be chosen for rendering by setting the Context attributes **XGL_CTX_SURF_FRONT_COLOR_SELECTOR**(3) and **XGL_3D_CTX_SURF_BACK_-COLOR_SELECTOR**(3)**.** In 3D, the final appearance of the surface depends on the illumination model specified with the attributes **XGL_3D_CTX_SURF_FRONT_ILLUMINATION**(3) and

XGL_3D_CTX_SURF_BACK_ILLUMINATION(3)**,** as well as the depth-cue mode given by the attribute **XGL_3D_CTX_DEPTH_CUE_MODE**(3)**.**

As part of the processing of a facet, XGL may need to determine the orientation of a facet: either front- or back-facing. These situations include enabling illumination, face culling, or face distinguishing. For these cases, an application benefits by supplying a unit normal vector with the facet data, which points in the direction of the front face of the polygon. This saves XGL from calculating it from the vertices with the method given by **XGL_3D_CTX_SURF_GEOM_NORMAL**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

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, and vice versa.

When filling the interior of polygons, XGL uses a technique that ensures adjoining polygons do not overlap. XGL assumes that pixels along the top and left edges of a polygon are *inside* the polygon, and those along the right and bottom edges are *outside*. Using this scheme, pixels along the common edge of two adjoining polygons are not drawn twice. However, it also prevents pixels from being drawn along the right and bottom edges of the clipping window. (Note that this is true only for polygons, triangle strips, triangle lists, NURB surfaces, and quadrilateral meshes. It is not true for lines, edges, stroke text, curves, and markers.)

Use **xgl_polygon**(3) to draw self-intersecting or multibounded polygons. For polygons with a simpler definition, use **xgl_multi_simple_polygon**(3)**.**

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 of facets */
                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.

NAME

xgl_stroke_text – renders 2D and 3D stroke text

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_stroke_text (
        Xgl_ctx          ctx,
        void             *str,
        Xgl_pt_f{2,3}d   *pos,
        /* f2d for 2D text, f3d for 3D text */
        Xgl_pt_f3d       dir[] );
        /* Ignored for 2D text */
```

DESCRIPTION

Purpose

This Context operator renders the text indexed by *str*, using the Context attributes within *ctx*.

Parameters

pos is the 2D or 3D reference point for the rendered text. When using this operator in a 3D environment, *dir* defines the two vectors used for the orientation of the 2D plane on which the text sits. The first vector defines a logical horizontal direction that supports the text, and the second vector gives a logical vertical direction to define an orientation for the text. Each point within the stroke character is transformed through the XGL transformation pipeline. Depending on the character-encoding scheme defined by the Context attribute **XGL_CTX_STEXT_CHAR_ENCODING**(3)**,** the *str* parameter will represent a *NULL-terminated* C-style string of characters, or a pointer on a structure of type *Xgl_mono_text_list*.

If the string-encoding scheme used is of type XGL_SINGLE_STR, at least one font can be selected to render the characters in ISO or EUC / multi-byte character-encoding schemes:

XGL_CTX_SFONT_0(3)

When the character-encoding scheme used is of type XGL_CHAR_MBY, the fonts used to render the string are selected depending on the character code adding the following Context attributes:

XGL_CTX_SFONT_1(3)
XGL_CTX_SFONT_2(3)
XGL_CTX_SFONT_3(3)

If the string-encoding scheme used is of type XGL_MULTI_STR, the structure passed in parameter contains the XGL font Object used to render the characters. Both character-encodings ISO or MBY can be used.

The application can control the geometrical characteristics of the text using the following attributes:

XGL_CTX_STEXT_CHAR_HEIGHT(3)
XGL_CTX_STEXT_CHAR_SPACING(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)

The text orientation is controlled by changing the components of the **XGL_CTX_STEXT_-CHAR_UP_VECTOR**(3) attribute. An inclination can be added to the orientation by using the **XGL_CTX_STEXT_CHAR_SLANT_ANGLE**(3) attribute.

The text presentation, with respect to the position given by the parameters *dir* and *pos*, is managed by the following attributes:

XGL_CTX_STEXT_ALIGN_HORIZ(3)
XGL_CTX_STEXT_ALIGN_VERT(3)
XGL_CTX_STEXT_PATH(3)

The color of the text can be modified with the **XGL_CTX_STEXT_COLOR**(3) attribute. The way that stroke text is clipped on the Device is controlled by the **XGL_CTX_STEXT_PRECISION**(3) attribute.

XGL stroke text is rendered as a set of polylines. The width of the lines is fixed, and no pattern can be applied to the rendering of stroke text. The ROP, plane mask, and depth-cueing apply, as for any other rendering primitive, and can be modified with the following attributes, respectively:

XGL_CTX_ROP(3)
XGL_CTX_PLANE_MASK(3)
XGL_3D_CTX_DEPTH_CUE_MODE(3)

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_3D_CTX_DEPTH_CUE_MODE(3)
XGL_CTX_PLANE_MASK(3)
XGL_CTX_ROP(3)
XGL_CTX_SFONT_0(3)
XGL_CTX_SFONT_1(3)
XGL_CTX_SFONT_2(3)
XGL_CTX_SFONT_3(3)
XGL_CTX_STEXT_CHAR_ENCODING(3)
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR(3)
XGL_CTX_STEXT_CHAR_HEIGHT(3)
XGL_CTX_STEXT_CHAR_SLANT_ANGLE(3)
XGL_CTX_STEXT_CHAR_SPACING(3)
XGL_CTX_STEXT_CHAR_UP_VECTOR(3)
XGL_CTX_STEXT_ALIGN_HORIZ(3)
XGL_CTX_STEXT_ALIGN_VERT(3)
XGL_CTX_STEXT_COLOR(3)
XGL_CTX_STEXT_PATH(3)
XGL_CTX_STEXT_PRECISION(3)

NOTES

For XGL stroke text, the character up vector, defined by the **XGL_CTX_STEXT_CHAR_UP_VECTOR**(3) attribute is independent of the VDC orientation (defined by **XGL_CTX_VDC_ORIENTATION**(3)) and should always be interpreted as if the Y axis was ALWAYS pointing up.

NAME

xgl_stroke_text_extent – computes the text-extent rectangle bounding a stroke-text string

SYNOPSIS

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

void  xgl_stroke_text_extent (
            Xgl_ctx             ctx,
            void                *text,
            Xgl_bounds_f2d      *rect,
            Xgl_pt_f2d          *cat_pt );
```

DESCRIPTION

This operator computes the minimum-sized rectangle, in text local coordinates, that completely encompasses the text string specified by the parameter *text*. The parameter *ctx* is an XGL Context being used by the application. It contains the stroke text attributes necessary to compute the rectangular box returned in the parameter *rect*.

Depending on the character-encoding scheme defined by the Context attribute **XGL_CTX_STEXT_CHAR_ENCODING**(3)**,** the *text* parameter will represent a *NULL–terminated* C-style string of characters or a pointer on a structure of type *Xgl_mono_text_list*.

The parameter *cat_pt* is also computed. It contains the concatenation point, which is where the string ends, in text local coordinates. This information can be used to append a new string after a previously rendered one.

Because *rect* is returned as an *Xgl_bounds_f2d* structure for both 2D and 3D stroke text, *rect* is computed considering a character up vector (see **XGL_STEXT_CHAR_UP_VECTOR**(3)) of (0.0, 1.0), and an inclination angle (see **XGL_STEXT_CHAR_SLANT_ANGLE**(3)) of 0.0, so that its sides are parallel to the unrotated coordinate axes. In other words, the text-extent rectangle is computed independently of the character up vector.

This operator uses all of the geometric stroke text attributes to compute the elements returned to the caller. For a list of these attributes and their effects, see the **xgl_stroke_text**(3) and **xgl_annotation_text**(3) operators.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_annotation_text(3)
xgl_stroke_text(3)
XGL_CTX_STEXT_CHAR_ENCODING(3)
XGL_CTX_STEXT_CHAR_SLANT_ANGLE(3)
XGL_CTX_STEXT_CHAR_UP_VECTOR(3)

NAME

xgl_transform_copy – makes a copy of a Transform

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_copy (
            Xgl_trans    dest_trans,
            Xgl_trans    src_trans );
```

DESCRIPTION

Purpose

This operator copies the contents of the matrix associated with a Source Transform to the matrix associated with a Destination Transform. The Destination Transform retains its original data type and dimension, even if they differ from those of the Source Transform. Thus, this operator provides a means of converting a Transform of one type and dimension to another.

If the Source and Destination Transforms differ in type, a data conversion, possibly with round-off errors, is necessary.

If the two Transforms differ in dimension, the *z* coordinate is ignored. Three-dimensional homogeneous coordinates are treated as row vectors of the form [*x*, *y*, *z*, *w*]. If the Source Transform is 2D and the Destination Transform is 3D, **xgl_transform_copy**(3) fills the third row and third column of the Destination with the vector [0, 0, 1, 0], and the contents of the Source fill the remaining entries in the Destination Transform's matrix. If the Source Transform is 3D and the Destination is 2D, xgl_transform_copy strikes out the third row and third column of the Source and copies the remaining entries to the Destination Transform.

Parameters

dest_trans

The Destination Transform with the matrix to which the data is stored. It must have been created previously with xgl_object_create. Any information previously stored in the matrix of *dest_trans* is lost.

src_trans

The Source Transform with the matrix from which the data is taken. It must have been created previously with **xgl_object_create**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

NAME

xgl_transform_identity – sets a Transform to the identity

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_identity (
            Xgl_trans    trans );
```

DESCRIPTION

This operator sets a Transform's matrix to the identity. The data type and dimension of this matrix are given by the values of the Transform's attributes **XGL_TRANS_DATA_TYPE**(3) and **XGL_TRANS_DIMENSION**(3)**.**

The parameter *trans* is the Transform whose matrix is set to the identity. It must have been created previously with **xgl_object_create**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NAME

xgl_transform_invert – inverts a Transform

SYNOPSIS

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

Xgl_trans  xgl_transform_invert (
                Xgl_trans    dest_trans,
                Xgl_trans    src_trans );
```

DESCRIPTION

Purpose

This operator inverts a Transform's matrix. The numerical accuracy depends on the Transform's data type as given by the attribute **XGL_TRANS_DATA_TYPE**(3)**.**

Parameters

dest_trans

The Destination Transform to which the inverse is stored. It must have been created previously with **xgl_object_create**(3)**.** *dest_trans* can be the same as *src_trans* if the application wants the original matrix to be overwritten by the inverse. If *dest_trans* differs from *src_trans*, however, they must have the same dimension as given by **XGL_TRANS_DIMENSION**(3)**.**

src_trans

The Source Transform whose matrix is inverted. It must have been created previously with xgl_object_create.

Execution

A Transform may be singular; in other words, it has no unique inverse. **xgl_transform_-invert**(3) normally returns *dest_trans*, but it will return NULL if the Transform is singular.

RETURN VALUE

Returns *dest_trans* or *NULL* if *src_trans* has no inverse.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NOTES

Do not mix 2D and 3D Transforms with this call.

NAME

xgl_transform_multiply – multiplies two Transforms

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_multiply (
            Xgl_trans    dest_trans,
                         left_src_trans,
                         right_src_trans );
```

DESCRIPTION

Purpose

This operator multiplies two Transforms. All Transforms passed to this operator must have the same dimension as given by **XGL_TRANS_DIMENSION**(3)**.** However, the Transforms need not have the same data type as given by **XGL_TRANS_DATA_TYPE**(3)**.** If necessary, the Source Transforms are converted to floating point before being multiplied, and the resulting Transform is converted to the type of the Destination Transform.

Parameters

dest_trans
The Destination Transform.

left_src_trans
The Left Source Transform.

right_src_trans
The Right Source Transform.

All of the parameters must have been created previously with **xgl_object_create**(3)**.** *left_src_trans* can be the same as *right_src_trans*, and *dest_trans* can be the same as either *left_src_trans* and *right_src_trans*.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NOTES

Do not mix 2D and 3D Transforms with this call.

NAME

xgl_transform_point – transforms a single point in place

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_point (
            Xgl_trans   trans,
            Xgl_pt      *point );
```

DESCRIPTION

Purpose

This operator transforms a single point. The operator treats the point as a row vector and multiplies it by a Transform's matrix. The result is stored back in the memory allocated for the point, overwriting the untransformed point. If the transformation results in a point with a homogeneous *w* coordinate not equal to 1, the *x*, *y*, and *z* (for 3D) coordinates are normalized by *w* to produce an ordinary point with *w* = 1. The dimension of the Transform as given by **XGL_TRANS_DIMENSION**(3) must match the dimension of the point. A 2D point must be given with a 2D Transform. Similarly, a 3D point must be given with a 3D Transform. The data type of the Transform as given by **XGL_TRANS_DATA_TYPE**(3)**,** however, can be different from that of the point.

Parameters

trans The Transform whose matrix is applied in transforming the point. It must have been created previously with **xgl_object_create**(3)**.**

point A pointer to a structure of type *Xgl_pt* defined as follows:

```
typedef struct {
        Xgl_pt_type     pt_type;    /* type of point */
        Xgl_pt_position pt;         /* union of pointers to one point */
} Xgl_pt;
```

point→pt_type is the type of the point in *point→pt*. The table below gives the valid point types.

point→pt is a union of pointers. The appropriate pointer, as determined by *point→pt_type*, gives the address of a point structure containing coordinates. The valid members of this union are as follows:

Field Name	Type	Point Type
pt.i2d	Xgl_pt_i2d *	XGL_PT_I2D
pt.f2d	Xgl_pt_f2d *	XGL_PT_F2D
pt.d2d	Xgl_pt_d2d *	XGL_PT_D2D
pt.f3d	Xgl_pt_f3d *	XGL_PT_F3D
pt.d3d	Xgl_pt_d3d *	XGL_PT_D3D

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NOTES

The dimensions of the point and the Transform must be the same, either 2D or 3D.

NAME

xgl_transform_point_list – transforms a list of points

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_point_list (
        Xgl_trans     trans,
        Xgl_pt_list   *src_pt_list,
        void          *dest_pts );
```

DESCRIPTION

Purpose

This operator transforms a list of points. The operator treats each point as a row vector and multiplies it by a Transform's matrix. If the transformation results in a point with a homogeneous *w* coordinate not equal to 1, and the destination point list has no storage for the *w* coordinate, the *x*, *y*, and *z* (for 3D) coordinates are normalized by *w* to produce an ordinary point with *w* = 1. The dimension of the Transform as given by **XGL_TRANS_DIMENSION**(3) must match the dimension of the points in the list. A list of 2D points must be given with a 2D Transform. Similarly, a list of 3D points must be given with a 3D Transform. The data type of the Transform as given by **XGL_TRANS_DATA_TYPE**(3)**,** however, can be different from that of the points in the list.

Parameters

trans

The Transform whose matrix is applied in transforming the list of points. It must have been created previously with **xgl_object_create**(3)**.**

src_pt_list

A pointer to a structure of type *Xgl_pt_list* defined as follows:

```
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;
```

This parameter specifies the source list of points, which this operator transforms.

src_pt_list→pt_type is the type of the points in *src_pt_list→pts*. The tables below give the valid point types.

src_pt_list→bbox is ignored.

src_pt_list→num_pts is the number of points in the list.

src_pt_list→pts is a union of pointers. The appropriate pointer, as determined by *point→pt_type* gives the address of a block of memory where the list of source points is stored. The valid members of this union for a 2D Transform are as follows:

Field Name	Type	Point Type
pts.i2d	Xgl_pt_i2d *	XGL_PT_I2D
pts.color_i2d	Xgl_pt_color_i2d *	XGL_PT_COLOR_I2D
pts.flag_i2d	Xgl_pt_flag_i2d *	XGL_PT_FLAG_I2D
pts.i2h	Xgl_pt_i2h *	XGL_PT_I2H
pts.f2d	Xgl_pt_f2d *	XGL_PT_F2D
pts.color_f2d	Xgl_pt_color_f2d *	XGL_PT_COLOR_F2D
pts.flag_f2d	Xgl_pt_flag_f2d *	XGL_PT_FLAG_F2D
pts.f2h	Xgl_pt_f2h *	XGL_PT_F2H
pts.d2d	Xgl_pt_d2d *	XGL_PT_D2D
pts.color_d2d	Xgl_pt_color_d2d *	XGL_PT_COLOR_D2D
pts.flag_d2d	Xgl_pt_flag_d2d *	XGL_PT_FLAG_D2D
pts.d2h	Xgl_pt_d2h *	XGL_PT_D2H

The valid members of this union for a 3D Transform are as follows:

Field Name	Type	Point Type
pts.f3d	Xgl_pt_f3d *	XGL_PT_F3D
pts.color_f3d	Xgl_pt_color_f3d *	XGL_PT_COLOR_F3D
pts.normal_f3d	Xgl_pt_normal_f3d *	XGL_PT_NORMAL_F3D
pts.color_normal_f3d	Xgl_pt_color_normal_f3d *	XGL_PT_COLOR_NORMAL_F3D
pts.flag_f3d	Xgl_pt_flag_f3d *	XGL_PT_FLAG_F3D
pts.color_flag_f3d	Xgl_pt_color_flag_f3d *	XGL_PT_COLOR_FLAG_F3D
pts.normal_flag_f3d	Xgl_pt_normal_flag_f3d *	XGL_PT_NORMAL_FLAG_F3D
pts.color_normal_flag_f3d	Xgl_pt_color_normal_flag_f3d *	XGL_PT_COLOR_NORMAL_FLAG_F3D
pts.f3h	Xgl_pt_f3h *	XGL_PT_F3H
pts.data_f3d	Xgl_pt_data_f3d *	XGL_PT_DATA_F3D
pts.color_data_f3d	Xgl_pt_color_data_f3d *	XGL_PT_COLOR_DATA_F3D
pts.normal_data_f3d	Xgl_pt_normal_data_f3d *	XGL_PT_NORMAL_DATA_F3D
pts.color_normal_data_f3d	Xgl_pt_color_normal_data_f3d *	XGL_PT_COLOR_NORMAL_DATA_F3D
pts.flag_data_f3d	Xgl_pt_flag_data_f3d *	XGL_PT_FLAG_DATA_F3D
pts.color_flag_data_f3d	Xgl_pt_color_flag_data_f3d *	XGL_PT_COLOR_FLAG_DATA_F3D
pts.normal_flag_data_f3d	Xgl_pt_normal_flag_data_f3d *	XGL_PT_NORMAL_FLAG_DATA_F3D
pts.color_normal_flag_data_f3d	Xgl_pt_color_normal_flag_data_f3d *	XGL_PT_COLOR_NORMAL_FLAG_DATA_F3D
pts.d3d	Xgl_pt_d3d *	XGL_PT_D3D
pts.color_d3d	Xgl_pt_color_d3d *	XGL_PT_COLOR_D3D
pts.normal_d3d	Xgl_pt_normal_d3d *	XGL_PT_NORMAL_D3D
pts.color_normal_d3d	Xgl_pt_color_normal_d3d *	XGL_PT_COLOR_NORMAL_D3D
pts.flag_d3d	Xgl_pt_flag_d3d *	XGL_PT_FLAG_D3D
pts.color_flag_d3d	Xgl_pt_color_flag_d3d *	XGL_PT_COLOR_FLAG_D3D
pts.normal_flag_d3d	Xgl_pt_normal_flag_d3d *	XGL_PT_NORMAL_FLAG_D3D
pts.color_normal_flag_d3d	Xgl_pt_color_normal_flag_d3d *	XGL_PT_COLOR_NORMAL_FLAG_D3D
pts.d3h	Xgl_pt_d3h *	XGL_PT_D3H

dest_pts
A pointer to a block of storage for the transformed points, which have the same type as the untransformed points. This block must be at least as large as the block allocated for points that are not transformed. It can point to the same location as the source point list if the application wants to transform the points in-place, overwriting the original points.

If the source points contain information other than coordinates (such as normals, colors, or flags), this extra information is not transferred to the destination list of points.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NOTES

The dimensions of the points and the Transform must be the same, either 2D or 3D.

NAME

xgl_transform_read – reads a Transform's matrix into an array

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_read (
            Xgl_trans    src_trans,
            void         *matrix );
```

DESCRIPTION

Purpose

This operator reads a Transform's matrix into an array supplied by the application.

Parameters

src_trans

The Transform whose matrix is read. It must have been created previously with **xgl_object_create**(3)**.**

matrix One of the following types of array:

```
typedef  Xgl_sgn32  Xgl_matrix_i2d [3][2];   /* 2D integer matrix */
typedef  float      Xgl_matrix_f2d [3][2];   /* 2D 32-bit floating-point matrix */
typedef  double     Xgl_matrix_d2d [3][2];   /* 2D 64-bit floating-point matrix */
typedef  float      Xgl_matrix_f3d [4][4];   /* 3D 32-bit floating-point matrix */
typedef  double     Xgl_matrix_d3d [4][4];   /* 3D 64-bit floating-point matrix */
```

Execution

The Transform's data type given by **XGL_TRANS_DATA_TYPE**(3) and its dimension given by **XGL_TRANS_DIMENSION**(3) determine which array must be passed by the application. Only one type of array is valid at any given time. No conversion takes place, so the Transform's data type and dimension must match those of the array.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)

XGL_TRANS_DIMENSION(3)

NAME

xgl_transform_rotate – combines a Transform with a Rotation Transform

SYNOPSIS

```
#include <xgl/xgl.h>
void    xgl_transform_rotate (
                Xgl_trans           trans,
                double              angle,
                Xgl_axis            axis,
                Xgl_trans_update    update );
```

DESCRIPTION

Purpose

This operator combines a given Transform with a Rotation Transform constructed with the given parameters. The Rotation Transform can replace the given Transform, or it can be preconcatenated or postconcatenated with the given Transform.

Parameters

trans The Transform to be updated with a Rotation Transform. It must have been created previously with **xgl_object_create**(3)**.**

angle The angle of rotation in radians. For 2D Transforms, the angle is positive, in the counterclockwise sense, when the *x*-axis points right and the *y*-axis points up. The angle is positive, in the clockwise sense, when the *x*-axis points right and the *y*-axis points down. For 3D Transforms, the angle is positive in the right-handed sense.

axis Refers to the *x*, *y*, or *z*-axis, corresponding to the axis of rotation for 3D Transforms. XGL accepts the following values for *Xgl_axis*:

XGL_AXIS_X
: The axis of rotation is the *x*-axis

XGL_AXIS_Y
: The axis of rotation is the *y*-axis

XGL_AXIS_Z
: The axis of rotation is the *z*-axis

axis is ignored for 2D Transforms.

update Establishes the method of combining the Rotation Transform with the target Transform. XGL accepts the following values for *Xgl_trans_update*:

XGL_TRANS_REPLACE
: The Rotation Transform replaces the target Transform *trans.*

XGL_TRANS_PRECONCAT
: The Rotation Transform is preconcatenated to the target Transform *trans*, and the result is stored in *trans.* XGL treats coordinates as row vectors, which means that the matrix of the Rotation Transform on the left, multiplied by the matrix of *trans* on the right, replaces the matrix of *trans.*

XGL_TRANS_POSTCONCAT
: The Rotation Transform is postconcatenated to the target Transform

trans, and the result is stored in *trans*. XGL treats coordinates as row vectors, which means that the matrix of *trans* on the left, multiplied by the matrix of the Rotation Transform on the right, replaces the matrix of *trans*.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NOTES

Integer Transforms only allow rotations by angles of *pi* / 2 and its multiples.

NAME

xgl_transform_scale – combines a Transform with a Scale Transform

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_scale (
        Xgl_trans          trans,
        Xgl_pt             *scale_factors,
        Xgl_trans_update   update );
```

DESCRIPTION

Purpose

This operator combines a given Transform with a Scale Transform constructed with the given parameters. The Scale Transform can replace the given Transform, or it can be preconcatenated or postconcatenated with the given Transform.

Parameters

trans The Transform to be updated with a Scale Transform. It must have been created previously with **xgl_object_create**(3)**.**

scale_factors

Contains the scale factors used to construct a Scale Transform. *scale_factors* is a pointer to a structure of type *Xgl_pt* defined as follows:

```
typedef struct {
        Xgl_pt_type       pt_type;    /* type of point */
        Xgl_pt_position   pt;         /* union of pointers to one point */
} Xgl_pt;
```

scale_factors→pt_type is the type of the point in *scale_factors→pt*. The table below gives the valid point types.

scale_factors→pt is a union of pointers. The appropriate pointer, which is determined by *scale_factors→pt_type*, gives the address of a point structure containing coordinates. The valid members of this union are as follows:

Field Name	Type	Point Type
pt.i2d	Xgl_pt_i2d *	XGL_PT_I2D
pt.f2d	Xgl_pt_f2d *	XGL_PT_F2D
pt.d2d	Xgl_pt_d2d *	XGL_PT_D2D
pt.f3d	Xgl_pt_f3d *	XGL_PT_F3D
pt.d3d	Xgl_pt_d3d *	XGL_PT_D3D

The dimension of *trans*, as given by **XGL_TRANS_DIMENSION**(3)**,** and the data type, as given by **XGL_TRANS_DATA_TYPE**(3)**,** do not need to match those of *scale_factors*. If *trans* is 3D and *scale_factors* is 2D, the *z* scale factor of the Scale Transform is set to 0. If *trans* is 2D and *scale_factors* is 3D, the *z* scale factor is ignored.

update Establishes the method of combining the Scale Transform with the target Transform. XGL accepts the following values for Xgl_trans_update:

XGL_TRANS_REPLACE
: The Scale Transform replaces the target Transform *trans.*

XGL_TRANS_PRECONCAT
: The Scale Transform is preconcatenated to the target Transform *trans*, and the result is stored in *trans*. XGL treats coordinates as row vectors, which means that the matrix of the Scale Transform on the left, multiplied by the matrix of *trans* on the right, replaces the matrix of *trans.*

XGL_TRANS_POSTCONCAT
: The Scale Transform is postconcatenated to the target Transform *trans*, and the result is stored in *trans*. XGL treats coordinates as row vectors, which means that the matrix of *trans* on the left, multiplied by the matrix of the Scale Transform on the right, replaces the matrix of *trans.*

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NAME

xgl_transform_translate – combines a Transform with a Translation Transform

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_translate (
        Xgl_trans          trans,
        Xgl_pt             *offset,
        Xgl_trans_update   update );
```

DESCRIPTION

Purpose

This operator combines a given Transform with a Translation Transform that is constructed with the given parameters. The Translation Transform can replace the given Transform, or it can be preconcatenated or postconcatenated with the given Transform.

Parameters

trans The Transform to be updated with a Translation Transform. It must have been created previously with **xgl_object_create**(3)**.**

offset Contains the offset used to construct a Translation Transform. *offset* is a pointer to a structure of type *Xgl_pt* defined as follows:

```
typedef struct {
        Xgl_pt_type      pt_type;    /* type of point */
        Xgl_pt_position  pt;         /* union of pointers to one point */
} Xgl_pt;
```

offset→pt_type is the type of the point in *offset→pt*. The table below gives the valid point types.

offset→pt is a union of pointers. The appropriate pointer, as determined by *offset→pt_type*, gives the address of a point structure containing coordinates. The valid members of this union are as follows:

Field Name	Type	Point Type
pt.i2d	Xgl_pt_i2d *	XGL_PT_I2D
pt.f2d	Xgl_pt_f2d *	XGL_PT_F2D
pt.d2d	Xgl_pt_d2d *	XGL_PT_D2D
pt.f3d	Xgl_pt_f3d *	XGL_PT_F3D
pt.d3d	Xgl_pt_d3d *	XGL_PT_D3D

The dimension of *trans*, as given by **XGL_TRANS_DIMENSION**(3)**,** and the data type, as given by **XGL_TRANS_DATA_TYPE**(3)**,** do not need to match those of *offset*. If *trans* is 3D and *offset* is 2D, the *z* offset value of the Translation Transform is set to 0. If *trans* is 2D and *offset* is 3D, the *z* offset value is ignored.

update Establishes the method of combining the Translation Transform with the target Transform. XGL accepts the following values for Xgl_trans_update:

XGL_TRANS_REPLACE
: The Translation Transform replaces the target Transform *trans*.

XGL_TRANS_PRECONCAT
: The Translation Transform is preconcatenated to the target Transform *trans*, and the result is stored in *trans*. XGL treats coordinates as row vectors, which means that the matrix of the Translation Transform on the left, multiplied by the matrix of *trans* on the right, replaces the matrix of *trans*.

XGL_TRANS_POSTCONCAT
: The Translation Transform is postconcatenated to the target Transform *trans*, and the result is stored in *trans*. XGL treats coordinates as row vectors, which means that the matrix of *trans* on the left, multiplied by the matrix of the Translation Transform on the right, replaces the matrix of *trans*.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)

XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NAME

xgl_transform_transpose – transposes a Transform

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_transform_transpose (
            Xgl_trans    dest_trans,
            Xgl_trans    src_trans );
```

DESCRIPTION

Purpose

This operator transposes a Transform's matrix.

Parameters

dest_trans

The Destination Transform into which the transpose is stored. It must have been created previously with **xgl_object_create**(3)**.** *dest_trans* can be the same as *src_trans* if the application wants the original matrix to be overwritten by the transpose. The dimensions and data types of *dest_trans* and *src_trans* may differ. When they do, the conversions are similar to those performed by **xgl_-transform_copy**(3)**.**

src_trans

The Source Transform whose matrix is transposed. It must have been created previously with **xgl_object_create**(3)**.**

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_transform_copy(3)
xgl_object_create(3)
XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NAME

xgl_transform_write_specific – writes an array into a Transform's matrix along with a description of its membership to matrix groups

SYNOPSIS

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

void  xgl_transform_write_specific (
        Xgl_trans          dest_trans,
        void*              matrix,
        Xgl_trans_member   membership );
```

DESCRIPTION

Purpose

This operator writes an array supplied by the application into a Transform's matrix. An application can specify the matrix groups to which the matrix is a member. This membership information can significantly improve the performance of most operations involving Transforms.

Parameters

dest_trans

The Transform whose matrix is written. It must have been created previously with **xgl_object_create**(3)**.**

matrix One of the following types of array:

```
typedef   Xgl_sgn32   Xgl_matrix_i2d [3][2];   /* 2D integer matrix */
typedef   float       Xgl_matrix_f2d [3][2];   /* 2D 32-bit floating-point matrix */
typedef   double      Xgl_matrix_d2d [3][2];   /* 2D 64-bit floating-point matrix */
typedef   float       Xgl_matrix_f3d [4][4];   /* 3D 32-bit floating-point matrix */
typedef   double      Xgl_matrix_d3d [4][4];   /* 3D 64-bit floating-point matrix */
```

The Transform's data type, as given by **XGL_TRANS_DATA_TYPE**(3)**,** and its dimension, as given by **XGL_TRANS_DIMENSION**(3)**,** determine which array must be passed by the application. Only one type of array is valid at any given time. No data conversion takes place, so the Transform's data type and dimension must match those of the array.

membership

The membership of *matrix* to matrix groups. If *matrix* has the form described with one of the values below, then an application can specify that value for the membership parameter for increased performance in most operations involving Transforms. This parameter can be NULL, effectively the same as **xgl_transform_-write**(3)**,** but performance in most cases will be lower than specifying the applicable value. However, if an application specifies a value that does not apply to the matrix written into the Transform, then unexpected results can occur such as geometry in the wrong location or incorrect lighting.

Let **M** be a 4×4 matrix for 3D transformations. **M** has the following form:

$$\mathbf{M} = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}$$

The analogous 2D matrix can be obtained by striking out the third row and third column (corresponding to z). The array written into a 2D Transform is 3×2 because the last column is always $[\ 0\ 0\ 1\ ]^T$ so it is implicit.

XGL treats points as row vectors and normal vectors as column vectors.

$$\mathbf{p} = \begin{bmatrix} x & y & z & w \end{bmatrix}$$

$$\mathbf{n} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$$

An affine transformation is a geometric mapping that preserves the straightness of lines and the parallelism of parallel lines but possibly alters the distance between points and/or the angle between lines. Perspective transformations are not affine.

Since affine transformations frequently occur in graphics, we can benefit from identifying specific categories. In modern algebra, these are called groups.

If **M** is affine, it has the following form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & 1 \end{bmatrix}$$

where **A** is the upper-left 3×3 submatrix of **M**,
C is the lower-left 1×3 submatrix of **M**,
0 is a 3×1 column vectors of zeros,
1 is the lower-right 1×1 submatrix of **M**.

This notation is called block matrix because the *elements* are matrices.

The values for membership are as follows:

XGL_TRANS_MEMBER_IDENTITY

Matrices of this type have the following form:

For 2D

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For 3D

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

XGL_TRANS_MEMBER_TRANSLATION

Matrices of this type have the following form:

For 2D

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

For 3D

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

XGL_TRANS_MEMBER_SCALE

Matrices of this type have the following form:

For 2D

$$\mathbf{M} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For 3D

$$\mathbf{M} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If any scale factor is zero, the matrix is singular.

XGL_TRANS_MEMBER_ROTATION

Matrices of this type have the following form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}$$

where **A** is orthogonal meaning that $\mathbf{A}^{-1} = \mathbf{A}^{T}$. For 2D, **A** is 2×2; for 3D, **A** is 3×3.

XGL_TRANS_MEMBER_WINDOW

Matrices of this type have the following form:

For 2D

$$\mathbf{M} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

For 3D

$$\mathbf{M} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

A window matrix can scale and translate.

XGL_TRANS_MEMBER_SHEAR_SCALE

Matrices of this type have the following form:

For 3D

$$\mathbf{M} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ h_x & h_y & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This shear/scale matrix type does not apply to 2D.

XGL_TRANS_MEMBER_LENGTH_PRESERV

Matrices of this type have the following form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & 1 \end{bmatrix}$$

where **A** is orthogonal meaning that $\mathbf{A}^{-1} = \mathbf{A}^{T}$. For 2D, **A** is 2×2; for 3D, **A** is 3×3. **C** is any 1×2 submatrix for 2D or 1×3 submatrix for 3D.

A length-preserving matrix can translate, rotate, and reflect about an axis; these operations preserve the distance between any two points and the angle between any two intersecting lines.

XGL_TRANS_MEMBER_ANGLE_PRESERV

Matrices of this type have the following form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & 1 \end{bmatrix}$$

where **A** has the form $\mathbf{A} = s\mathbf{Q}$ and *s* is a positive scalar and **Q** is orthogonal or $\mathbf{Q}^{-1} = \mathbf{Q}^{T}$. For 2D, **A** and **Q** are 2×2; for 3D, **A** and **Q** are 3×3. **C** is any 1×2 submatrix for 2D or 1×3 submatrix for 3D.

An angle-preserving matrix can translate, rotate, reflect about an axis, and scale by an amount which is uniform in all directions; these operations preserve the angle between any two intersecting lines.

XGL_TRANS_MEMBER_AFFINE

Matrices of this type have the following form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & 1 \end{bmatrix}$$

where **A** is any nonsingular 2×2 matrix for 2D or 3×3 matrix for 3D. **C** is any 1×2 submatrix for 2D or 1×3 submatrix for 3D.

An affine matrix can translate, rotate, reflect, scale anisotropically, and shear; lines remain straight and parallel lines remain parallel but the angle between intersecting lines can change.

XGL_TRANS_MEMBER_LIM_PERSPECTIVE

Matrices of this type have the following form:

For 3D

$$\mathbf{M} = \begin{bmatrix} * & 0 & 0 & 0 \\ 0 & * & 0 & 0 \\ * & * & * & * \\ * & * & * & * \end{bmatrix}$$

This limited perspective matrix type does not apply to 2D.

RETURN VALUE

Nothing is returned.

SEE ALSO

xgl_object_create(3)
xgl_transform_write(3)
XGL_TRANS_DATA_TYPE(3)
XGL_TRANS_DIMENSION(3)

NAME

xgl_triangle_list – draws a triangle list

SYNOPSIS

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

DESCRIPTION

Purpose

This operator draws a triangle list according to the parameter list and the graphics state described in the 3D Context *ctx*. Only 3D Contexts can be used with this operator. All of the Context attributes that directly affect the display of triangle lists are listed in the **SEE ALSO** section below.

A triangle list is defined by a single point list containing at least three vertices. It may be composed of a series of connected triangles arranged as a triangle strip, or 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. To produce a *triangle strip*, each additional vertex replaces the *oldest vertex* from the previous triangle thus defining a new adjoining triangle. To produce a *triangle star*, the new vertex replaces the *second oldest vertex*, again defining an adjoining triangle. Finally, to produce *independent triangles*, each set of three vertices define a new triangle which (probably) will not be adjoining with the previous one. Flags, either passed to this primitive via the *flags* parameter or supplied in the flag word within the vertex data, determine whether the geometry produces a strip, star, independent triangles, or a combination of the three.

More precisely, a triangle list consists of an ordered list of (n+2) vertices, $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n+1}\}$, $\mathbf{n} > 1$. The edges for the first triangle lie between the vertices $[\mathbf{v}_0, \mathbf{v}_1]$, $[\mathbf{v}_1, \mathbf{v}_2]$, and $[\mathbf{v}_0, \mathbf{v}_2]$, respectively. For triangle stars and strips and for $i = 3, 4, \ldots, n+1$, each vertex $\mathbf{v}_i$ replaces one vertex from the previous triangle. This new vertex forms two new edges with the two vertices remaining from the previous triangle. They are used to form a new triangle that adjoins the previous one. In these two cases, XGL allows each new vertex $\mathbf{v}_i$ to replace one of two different vertices from the previous triangle — the oldest vertex from that triangle, $\mathbf{v}_{i-2}$, or the second oldest vertex, $\mathbf{v}_{i-1}$. The first case produces a *triangle strip* that is identical to that produced by the primitive **xgl_triangle_strip**(3)**.** The second case produces a *triangle star*. For independent triangles and for $i = 3, 6, 9, 12, \ldots, n-1$, each set of three vertices $\{\mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}\}$ produces a new triangle.

Assuming that there is no intermixing of strips, stars, and independent triangles within the same triangle_list, and assuming there are n+2 vertices, $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n+1}\}$, the following triangles will be produced in the three different cases:

Triangle Strip (*replace oldest*) –

n triangles, $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$ composed of the following vertices:
$\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$, $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$, $\{\mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$, ... , $\{\mathbf{v}_{n-1}, \mathbf{v}_n, \mathbf{v}_{n+1}\}$.

Triangle Star (*replace second oldest*) –

n triangles, $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$ composed of the following vertices:
$\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}, \{\mathbf{v}_0, \mathbf{v}_2, \mathbf{v}_3\}, \{\mathbf{v}_0, \mathbf{v}_3, \mathbf{v}_4\}, \ldots, \{\mathbf{v}_0, \mathbf{v}_n, \mathbf{v}_{n+1}\}$.

Independent Triangles (*restart every third vertex*) –

$(n+2)/3$ triangles, $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{(n+2)/3}$ composed of the following vertices:
$\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}, \{\mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5\}, \{\mathbf{v}_6, \mathbf{v}_7, \mathbf{v}_8\}, \ldots, \{\mathbf{v}_{n-1}, \mathbf{v}_n, \mathbf{v}_{n+1}\}$.

Bits within the flag associated with each vertex or within the *global* flag word, *flags*, passed as a parameter determine how the vertices are used to create triangles (see discussion of vertex flags in **Execution** section or discussion about *flags* parameter below). Triangle strips, stars and independent triangles may be intermixed within the same strip. To do this, the application must set the per-vertex flags appropriately (again, see **Execution** section below).

Parameters

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 of facets */
                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 any one of XGL_FACET_NONE, XGL_FACET_COLOR, XGL_FACET_-NORMAL, or 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'th* Xgl_color_facet.

The array of facet data is assumed to be *facets→num_facets* elements long with the data in the *i'th* facet corresponding to triangle $\mathbf{t}_i$. However, if the parameter *flags* has the XGL_TLIST_FLAG_IS_PLANAR flag set, then only the first facet in the facet list will be used to specify the facet characteristics for all of the triangles – the remainder of the facet list (if it exists) will be ignored.

The *facets* information is **not** used to draw the triangles if either *facets* is NULL or *facets→facet_type* is XGL_FACET_NONE. Also, if XGL_TLIST_FLAG_IS_PLANAR is not set, the facet information is ignored if *facets→num_facets* is less than *pl→num_pts* – 2, and if it is set, the facet information is ignored if *facets→num_facets* is less than 1.

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 list. This parameter is either NULL, or it is the bounding box around all of the points in *pl→pts*. The bounding box information is 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* can 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 list. 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 list.

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 list. For more information on texturing, see **xgl_polygon**(3)**.**

pl→pts is a union of pointers, one of which points to the array of vertex data that make up the triangle(s). The value of *pl→pt_type* determines the interpretation of this union.

For further details on 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 polygonal surfaces.

flags A flag word of type Xgl_tlist_flags containing bits specifying the *global* characteristics of the triangle list. *Global* characteristics are those that are common to the entire point list passed to the primitive.

Information in this flag word overrides any conflicting information provided by per-vertex flags (if available – see discussion about vertex flags in **Execution** section below).

Xgl_tlist_flags is specified as an enumerated type, but it actually contains some mutually exclusive values and some non-mutually exclusive values that may be OR'd together.

The mutually exclusive flags are:

XGL_TLIST_FLAG_USE_VTX_FLAGS

No global information is given about how the triangles are formed from the point list. Instead, per-vertex flags are used to define how each triangle is composed.

a. If this flag is set, then the vertex data **must** contain flag information (i.e. the ordering of the vertices in each triangle will be determined on a per-triangle basis). In order to use per-vertex flags with the xgl_triangle_list, this flag **must be set**.

b. If this flag is not set, then per-vertex flags are not required and either XGL_TLIST_FLAG_TRI_STRIP, XGL_TLIST_FLAG_TRI_STAR, or XGL_TLIST_FLAG_-TRI_RESTART will be set. In fact, if this flag is not set **and** per-vertex flags are present, the per-vertex bits controlling the triangle type will be ignored (the edge flag bits can still be used, however).

XGL_TLIST_FLAG_TRI_STRIP

All points in the point list are used to create a triangle strip (replace oldest vertex).

XGL_TLIST_FLAG_TRI_STAR

All points in the point list are used to create a triangle star (replace second-oldest vertex).

XGL_TLIST_FLAG_TRI_RESTART

Every third vertex in the point list is a restart vertex for a new triangle (this produces a batch of independent triangles).

The non-mutually exclusive flags are:

XGL_TLIST_FLAG_EDGES_ON

When set, and when per-vertex edge flags exist, causes all edges within the

triangle list to be drawn if the Context edge flag, **XGL_CTX_SURF_EDGE_-FLAG**(3) is also set. Both internal and non-internal edges will be drawn where "internal" edges are those marked with either the XGL_EDGE_IS_-INTERNAL or the XGL_PREV_EDGE_IS_INTERNAL flag bit.

If this bit is not set, then edges will be drawn if the per-vertex edge flag bits are set, the per-vertex internal edge flag bits are cleared, and the Context edge flag, **XGL_CTX_SURF_EDGE_FLAG**(3) is set.

XGL_TLIST_FLAG_IS_PLANAR

All triangles defined by the point list are assumed to be co-planar. This is a hint to, perhaps, allow for optimizations within XGL's lighting calculations.

If this flag is set, then only the first facet in the facet list will be used for facet information – the remainder of the facet list (if it exists) will be ignored.

XGL_TLIST_FLAG_FN_CONSISTENT
XGL_TLIST_FLAG_FN_NOT_CONSISTENT

Setting the XGL_TLIST_FLAT_FN_CONSISTENT flag indicates whether the direction of facet normal and the order in which the points are given are consistent for all triangles within the list.

In a right-handed coordinate system, the order of the points implies a facet normal that is calculated as follows:

$$\text{implied facet normal} = v1v2 \times v1v3$$

A user-provided facet normal indicates the front face of a triangle. Call a group of triangles XGL_TLIST_FLAG_FN_CONSISTENT if either of the following is true:

- The front of the triangle chosen by the user-provided facet normal always *agrees* with the front chosen by the implied facet normal.
- The front chosen by the user-provided facet normal always *disagrees* with that chosen by the implied facet normal.

Call a group of triangles XGL_TLIST_FLAG_FN_NOT_CONSISTENT if neither of the above conditions holds. That is, for some triangles, there is agreement; for others, there is disagreement.

The default is XGL_TLIST_FLAG_FN_NOT_CONSISTENT.

Setting this flag with XGL_TLIST_FLAG_FN_CONSISTENT gives better performance on some devices. Consult your accelerator guide for details.

If the parameter *flags* is NULL, then the following defaults apply:

1. The triangle list is composed of a chain of independent triangles (restart every third vertex) whose per-vertex flag bits controlling triangle type are **ignored**.
2. The per-vertex edge flag bits, if available, can still be used. An edge will be drawn if its corresponding edge flag bit is set, the corresponding internal edge flag bit is cleared, and the Context edge flag, **XGL_CTX_SURF_EDGE_-**

FLAG(3)**,** is set.

3. The triangles are not assumed to be co-planar.

Execution

If points with per-vertex flag information are used **and** the flag XGL_TLIST_FLAG_USE_VTX_-FLAGS within the *flags* parameter is set, 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 for the case of triangle strip and triangle star rendering. For a triangle strip, 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 [$\mathbf{v}_{i-2}$, $\mathbf{v}_i$]. For a triangle star, the flag bit XGL_DRAW_EDGE (0x1) controls the edge [$\mathbf{v}_{i-1}$, $\mathbf{v}_i$] as before. However, the bit XGL_DRAW_PREV_EDGE (0x2) controls the edge [$\mathbf{v}_{oldest}$, $\mathbf{v}_i$]. $\mathbf{v}_{oldest}$ is the oldest vertex retained from the previous triangle in creating the i^{th} triangle.

For vertices $\mathbf{v}_i$, $i = 2$ to n+1, with a set of independent triangles the edge bits are used as they are with triangle strips with the following exception:

- Every time a triangle is restarted, the first vertex is treated as $\mathbf{v}_0$ (i.e. its per-vertex flags are ignored, see above), the second vertex is treated as $\mathbf{v}_1$ (i.e. only the XGL_DRAW_EDGE bit is used), and the third vertex is treated as $\mathbf{v}_2$ (see above).

XGL_TRIANGLE_STRIP (0x4)
XGL_TRIANGLE_STAR (0x8)
XGL_TRIANGLE_RESTART (0x0)

These bits control the type of triangles created by the list. They allow for the intermixing of triangle strips, stars, and independent triangles within the same point list. They specify, on a per-vertex basis, which vertex from the previous triangle will be replaced by the current vertex:

XGL_TRIANGLE_STRIP – current vertex replaces oldest vertex from previous triangle.

XGL_TRIANGLE_STAR – current vertex replaces second-oldest vertex from previous triangle.

XGL_TRIANGLE_RESTART – current vertex is the first vertex of a new triangle. No information from the previous triangle is retained.

NOTE: The triangle type flags for the first three vertices in a point list are ignored (i.e. the first triangle is always a "RESTART" triangle). Thereafter, the triangle type flags will be checked at each vertex with the exception that when an XGL_-TRIANGLE_RESTART flag is encountered, the triangle type flags in the following two vertices will be ignored.

XGL_FACE_IS_CCW (0x10)

This bit specifies the ordering of the vertices for determining face orientation. CCW = counter-clockwise. This flag is only used at restart vertices (see XGL_-TRIANGLE_RESTART); elsewhere, it is ignored.

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. Also, five macros in xgl.h are to be used specifically for manipulating the per-vertex flag bits, XGL_TRIANGLE_STRIP, XGL_TRIANGLE_STAR, and XGL_TRIANGLE_RESTART, which xgl_triangle_list uses to define the type of triangles specified in the point list. These macros are: XGL_SET_TRI_STAR_FLAG(PT), XGL_SET_TRI_STRIP_FLAG(PT), XGL_CLR_TRI_-STAR_FLAG(PT), XGL_CLR_TRI_STRIP_FLAG(PT), and XGL_RESTART_TRIANGLE(PT). There is no separate set and clear macro for the RESTART flag because RESTART is implied when both the XGL_TRIANGLE_STAR and XGL_TRIANGLE_STRIP flags are cleared. The XGL_RESTART_-TRIANGLE(PT) clears these two bits. For more information, see *xgl.h*.

FACET COLORING

Generally, a *facet* refers to a single triangle within the triangle list. 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_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**(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.

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 of facets */
                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.

NAME

xgl_window_raster_resize – informs XGL of a window resize event

SYNOPSIS

```
#include <xgl/xgl.h>
void  xgl_window_raster_resize (
            Xgl_win_ras    ras );
```

DESCRIPTION

This operator is called by the application after every *window resize event* is reported by the window system. When **xgl_window_raster_resize**(3) is called, it causes XGL to update the size of the device (**XGL_DEV_MAXIMUM_COORDINATES**(3)) and the size of the DC viewport (**XGL_CTX_DC_VIEWPORT**(3)) unless the attribute **XGL_CTX_VDC_MAP**(3) is set to XGL_VDC_MAP_OFF, in which case the DC viewport must be explicitly set by the application and, for 3D Contexts, the software *z*-buffer updated.

For performance and toolkit-independence reasons, XGL does not trap window resize events. Therefore, this operator is required so that XGL can be aware of these events. The application must call xgl_window_raster_resize as part of its window resize handler. Otherwise XGL may produce unexpected results.

The parameter *ras* is the object handle of the Window Raster Device associated with the window that has been resized.

RETURN VALUE

Nothing is returned.

SEE ALSO

XGL_CTX_DC_VIEWPORT(3)
XGL_CTX_VDC_MAP(3)
XGL_DEV_MAXIMUM_COORDINATES(3)

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V

X

Z