General Acceleration Information 1

This chapter provides general acceleration information that will enable an XGL program to run in the most efficient way regardless of the targeted platform. For specific information on the GX and GXplus devices, see Chapter 2, "GX and GXplus Accelerators". For information on accelerating XGL applications across a network, see Chapter 3, "Acceleration Across a Network With the Xpex Pipeline".

Note – Memory rasters are not accelerated on any device.

Batching

Due to the overhead involved in every XGL call, applications that use an XGL primitive to draw a single object per call do not achieve maximum performance. Applications run significantly faster through effective use of the *multi* primitives; drawing about 30 objects per call results in nearly peak performance.

Note – Avoid using the primitive `xgl_polygon()` unless absolutely necessary, since it draws only one polygon per call.

1

Color Type and Visual

For good performance, you should understand the color model of the targeted device and program accordingly. A *visual* is an Xlib concept that describes the way pixels are translated into colors. XGL supports these visual classes: PseudoColor, TrueColor, and DirectColor. Take care when selecting the visual because the visual class affects the XGL hardware color type, which in turn affects accelerator performance.

Although XGL allows the mixing of visuals and color types, for best results PseudoColor visuals should be used with a raster color type of XGL_COLOR_INDEX, and TrueColor and DirectColor visuals should be used with XGL_COLOR_RGB. Table 1-1 shows these combinations.

Table 1-1 X Visual Class and XGL Color Type

Visual Class	XGL Color Type
PseudoColor	XGL_COLOR_INDEX
DirectColor TrueColor	XGL_COLOR_RGB

The following code segment shows the proper way to select the visual that gives the best performance on any frame buffer (8- or 24-bit).

```
void
appli_get_accelerated_x_visual(dpy, scr, depth, vis)
Display        *dpy;      /* IN: Xlib display structure */
Screen         scr;       /* IN: X screen */
int            *depth;    /* OUT: depth of frame buffer */
Visual         **vis;     /* OUT: visual class for window */
{
  Visual          *appli_get_x_visual();

  *vis = NULL;
  if ((*vis = appli_get_x_visual(dpy, scr, 24, TrueColor)))
    *depth = 24;
  else
    /* we didn't get TrueColor, so try for DirectColor */
    if ((*vis = appli_get_x_visual(dpy, scr, 24,
                                  DirectColor)))
    *depth = 24;
  else
    /*
```

```
     * we didn't get TrueColor or DirectColor, try PseudoColor
     */
  if ((*vis = appli_get_x_visual(dpy, scr, 8, PseudoColor)))
    *depth = 8;
  else
          /* we didn't get a visual, return error */
    *depth = 0;
}

Visual          *
appli_get_x_visual(dpy, scr, depth, visual_class)
Display         *dpy;             /* IN: Xlib display structure */
Screen           scr;             /* IN: X screen */
int             *depth;           /* IN: depth of frame buffer */
int              visual_class;    /* IN: TrueColor, DirectColor
                                   * or PseudoColor
                                   */
{
  XVisualInfo     template;     /* X visual info structure */
  XVisualInfo    *visuals,
                 *v;            /* temporary structures */
  int             nvisuals;     /* number of visuals */
  int             i;

  template.screen = scr;
  template.depth = depth;

  visuals = XGetVisualInfo(dpy,
                           VisualScreenMask | VisualDepthMask,
                           &template, &nvisuals);

  for (v - visuals, i = 0; i < nvisuals; v++, i++)
    if (v -> class == visual_class)
      return (v -> visual);

  return ((Visual *) NULL);
}
```

Note – Some graphics devices have 24-bit X visuals that are linear. This means that they display colors as if the monitor had a linear intensity response. This usually requires color correction on the part of the device through a process called gamma correction. XGL works best when it renders to a window that was created with a linear visual. For details on how to find a linear visual, refer to the *Solaris X Window System Developer's Guide*.

Tessellating Data

Tessellating (breaking down a complex polygon into triangles, for example) results in faster rendering on the GX accelerator. This performance optimization should be used only if the data can be easily tessellated into triangles, quadrilaterals, or rectangles. If the data cannot be easily tessellated, use XGL Geometry Caches to break down complex polygons and store them for later rendering. This is essentially a device-dependent performance optimization. Future graphics accelerators (from Sun or other vendors) may handle untessellated data more efficiently.

XGL Data Types

You should avoid copying geometric data into an XGL data structure and then calling an XGL primitive, since this costs both memory and time. If the format of the application's data is compatible with an XGL data type, you can set up pointers to the data and call XGL primitives directly with that data.

In addition, providing normals with your 3D data whenever necessary enables XGL to bypass calculating normals. If an object is to be rendered several times (such as in an animation sequence), this can result in a significant increase in speed.

Not sending down enough information can degrade performance, as can sending down too much data. For instance, sending down facet color data (or, worse yet, vertex data) when all the facets (vertices) have the same color results in slower rendering. Instead, the context color should be updated and only the geometry (x, y, z) information given to XGL.

Using `xgl_inquire()`

The API function `xgl_inquire()` enables an application program to get specific information about the frame buffer being used (such as the frame buffer name, the color type supported, whether hardware double buffering is available, and so on). The application program can use this information to render as efficiently as possible on that frame buffer. For instance, if the frame buffer supports hardware double-buffering, there is no need to set up Color Map double-buffering.

Use ASTI Deferral Mode

The default value of XGL_CTX_DEFERRAL_MODE (XGL_DEFER_ASTI) should be used whenever possible. Deferral mode XGL_DEFER_ASAP is slower and should be avoided.

Disable Error Checking

XGL error checking (enabled by setting XGL_SYS_ST_ERROR_DETECTION to TRUE) should be turned on while you are developing your application and turned off when you compile your application for distribution. Setting XGL_SYS_ST_ERROR_DETECTION to TRUE interferes with certain internal XGL optimizations. Error checking is disabled by default.

Use of Backing Store

If XGL_WIN_RAS_BACKING_STORE is set to TRUE, rendering will be slower whenever the window is partially or fully obscured, and slower still if anything is drawn into the obscured portion of the window (since such drawing is into a memory raster, which is not accelerated).

Faster Rendering Into Memory Rasters and Unaccelerated Frame Buffers

XGL releases subsequent to XGL 3.0 are significantly faster than XGL 3.0 at rendering certain primitives into memory rasters, or into simple frame buffers such as the CG3 (indexed color) and CG8 (TrueColor). Faster primitives include 2D vectors, 3D unshaded vectors (Z-buffered or not Z-buffered), and 3D polygons (including shaded and/or Z-buffered polygons) using either true or indexed color. The increase in speed for polygons currently applies only to 8-bit deep rasters and frame buffers.

This enhancement also applies to the GX and GXplus accelerators when rendering is unaccelerated for Z-buffered 3D polylines and Z-buffered and/or shaded polygons.

Faster Bounding Box Checking

XGL 3.0.1 and subsequent releases are faster than previous releases at performing bounding box checks on individual primitives. Complex primitives that are likely to be fully clipped away (during pan and zoom operations, for example) can gain in performance through the use of bounding boxes.