OpenGL & Window System Integration

“Most portable 3D; fastest 3D.”

SIGGRAPH ’97 Course

August 4, 1997

Mark J. Kilgard
Silicon Graphics, Inc.

Brian Paul
Avid Technology

Nate Robins
SGI, University of Utah,
Parametric Technology

Abstract

This practical course explains the application development options for writing portable, high−performance OpenGL programs for both the X Window System and Microsoft’s Windows 95 and NT. Instead of focusing on rendering images with OpenGL, this course focus on how OpenGL integrates with your native window system. The course emphasizes Windows programming and Motif−based approaches to writing real OpenGL applications. Techniques for ensuring portability between different platforms will be highlighted. The class also introduces high−level toolkits and alternative OpenGL interfaces. Advanced topics like stereo, effective debugging, and exotic input devices are covered.

OpenGL is a registered trademark of Silicon Graphics, Inc. X Window System is a registered trademark of X Consortium, Inc. Motif is a trademark of Open Software Foundation, Inc. Spaceball is registered trademark of Spatial Systems, Inc.

Copyright © 1994, 1995, 1996, 1997
Mark J. Kilgard, Brian Paul, Nate Robins.
All rights reserved.

Table of Contents

Course notes

Topic Discussion

Articles

Specification

Win32 Tutorials

The Speakers

Mark J. Kilgard

- o Member of the Technical Staff, Silicon Graphics, Inc.
- o Author of *Programming OpenGL for the X Window System.*
- o Directly involved in the design and implementation of SGI’s window system support for OpenGL.
- o Implemented OpenGL Utility Toolkit (GLUT).
- o Karaoke rendition of Dolly Parton’s ‘‘9 to 5’’ can’t be beat.

Address: Silicon Graphics, Inc., Mail Stop 8U−590, 2011 N. Shoreline Blvd., Moutain View, CA 94043−1389.
Email: `mjk@sgi.com` Phone: 415−390−2028 Fax: 415−965−2658.

Brian Paul

- o Graphics software engineer at Avid Technology.
- o Author of Mesa − free implementation of the OpenGL API.
- o Formerly: developer of scientific visualization software at University of Wisconsin − Madison.

Address: Avid Technology, 6400 Enterprise Lane − Suite 201, Madison, WI 53719.
Email: `brianp@sgi.com` Phone: 608−228−2014 Fax: 608−273−9198

The Speakers (cont'd)

Nate Robins

- Worked for Evans & Sutherland in the Graphics Systems Group.
- Ported the OpenGL Utility Toolkit (GLUT) to Windows 95 & NT.
- Worked for Parametric Technology porting Pro/3DPAINT to Windows NT.
- Currently an Intern at SGI.

OpenGL & Window System Integration

"Most portable 3D, fastest 3D."

OpenGL

Mark J. Kilgard	*Silicon Graphics, Inc.*
Brian Paul	*Avid Technology*
Nate Robins	*SGI, University of Utah,* *Parametric Technology*

SIGGRAPH '97 Course
August 4, 1997

Brian Paul

My background:

- **Graphics software engineer at Avid Technology**
- **Author of Mesa − free implementation of the OpenGL API**
- **Formerly: developer of scientific visualization software at University of Wisconsin − Madison**

Topics:

- ***OpenGL Development Choices***
- **Portability and Interoperability**
- **Off−screen Rendering**

OpenGL Development Choices

- **Overview**

Programming Languages

Low−level OpenGL interfaces

High−level OpenGL toolkits

Mesa

Development Choices: Overview

Choices:

- Programming language
- OpenGL integration method
- User interface toolkit

Issues:

- Commercial vs. free software
- Importance of cross−platform portability
- Complexity of the application

Programming Languages

- OpenGL API is defined by the C bindings.
- C++ bindings identical to those for C.
- Fortran bindings are common but inconsistent (identifier prefixes, identifier length restrictions).
- Ada, Modula−3, Tcl, Java and other bindings or wrappers are available.

OpenGL Integration Method

Low−level interfaces (GLX, WGL, etc):

- **Advantages:**
  - Provides access to all features (stereo, multisampling)
  - Standardized (i.e. GLX is used on all X/OpenGL systems)
- **Disadvantages:**
  - Doesn't provide GUI elements
  - Too many details, easy to make mistakes
  - Requires considerable window system programming knowledge

OpenGL Integration Method (2)

High−level interfaces (Motif, Tcl/Tk, etc):

Built on top of the low−level interfaces

- **Advantages:**
  - Hides implementation details
  - Clean integration with other GUI elements
  - May be more portable
- **Disadvantages:**
  - May not offer access to low−level features
  - May not be available for your GUI of choice

Example high−level interfaces

- Xt/Motif
- GLUT
- Tcl/Tk
- XForms
- Open Inventor
- OpenGL++
- Others

Xt/Motif

Motif is a popular widget set built on Xt, the X toolkit library.

The *GLwMDrawingArea* widget provides a canvas into which OpenGL can render.

- Motif advantages:
  Standardized
  Full featured

- Motif disadvantages:
  Large, complicated
  Not free

GLUT

GLUT is a free, portable toolkit which provides functions for creating windows, pop−up menus, event handling, simple geometric primitives and more.

- GLUT advantages:
  Free
  Very simple (like the OpenGL API)
  Good for demos and small applications

- GLUT disadvantages:
  Doesn’t provide all the GUI elements needed for real applications (buttons, scrollbars etc).

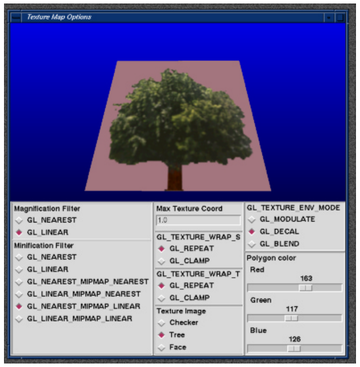
Tcl/Tk

Tcl is an interpreted scripting language. Tk is a GUI toolkit for Tcl. A number of Tk widgets are available for OpenGL rendering.

- Tcl/Tk advantages:
  Free
  Simple yet powerful
  Good for any size application
  Now available for X, Windows, Macintosh

- Tcl/Tk disadvantages:
  Interpreted; may not be fast enough in really demanding applications.
  OpenGL integration not standardized.

Tcl/Tk Example



Magnification Filter
○ GL_NEAREST
● GL_LINEAR

Minification Filter
○ GL_NEAREST
○ GL_LINEAR
○ GL_NEAREST_MIPMAP_NEAREST
○ GL_LINEAR_MIPMAP_NEAREST
● GL_NEAREST_MIPMAP_LINEAR
○ GL_LINEAR_MIPMAP_LINEAR

Max Texture Coord
1.0

GL_TEXTURE_WRAP_S
● GL_REPEAT
○ GL_CLAMP

GL_TEXTURE_WRAP_T
● GL_REPEAT
○ GL_CLAMP

Texture Image
○ Checker
● Tree
○ Face

GL_TEXTURE_ENV_MODE
○ GL_MODULATE
● GL_DECAL
○ GL_BLEND

Polygon color
Red 163
Green 117
Blue 126

Tcl/Tk Usage

Two approaches:

- Use Tcl wrappers for OpenGL to write an application entirely with Tcl/Tk (Tiger).
- Create and manage GUI and OpenGL canvas with Tcl/Tk but render into with with C code. (Togl)

Example: Togl (1)

- The Togl widget lets one create an OpenGL canvas in Tcl:

```
togl.my_widget -width 320 -height 200
  -rgba true -double true -depth true
  pack .my_widget
```

- Register C callback functions for widget creation, rendering, and resizing:

```
Togl_CreateFunc( create_cb );
Togl_DisplayFunc( display_cb );
Togl_ReshapeFunc( reshape_cb );
```

Example: Togl (2)

- C *create* callback function:

```
void create_cb( struct Togl *widget )
{
    glEnable( GL_DEPTH_TEST );
    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    /* load 3-D model */
    /* make display lists */
    /* etc. */
}
```

Example: Togl (3)

- C *rendering* callback function:

```
void display_cb( struct Togl *widget )
{
    glClear( GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT );

    /* draw something */

    Togl_SwapBuffers( widget );
}
```

Example: Togl (4)

- C *reshape* callback function:

```
void reshape_cb( struct Togl *widget )
{
    int width = Togl_Width( widget );
    int height = Togl_Height( widget );

    glViewport( 0, 0, width, height );
    // setup projection matrix with
    // glFrustum or glOrtho, etc.
}
```

Example: Togl (5)

- One may also define new commands implemented in C, callable from Tcl, to implement user−interface callbacks:

```
int reset_view_cb( struct Togl *widget,
                   int argc, char *argv[] )
{
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    Togl_PostRedisplay( widget );
    return TCL_OK;
}

Togl_CreateCommand("ResetView",reset_view_cb);
```

Example: Togl (6)

- Invoke the new command from Tcl with:

```
    .my_widget  ResetView
```

The main use of this feature is to send "messages" to the C program from Tcl in response to user input.

The command may simply modify a C variable or invoke an arbitrary computation.

Example: Togl (7)

- **Putting it all together:**

```
int main( int argc, char *argv[] )
{
    Tk_main( argc, argv, my_init );
    return 0;
}

int my_init( Tcl_Interp *interp )
{
    Tcl_Init( interp );
    Tk_Init( interp );
    Togl_Init( interp );
    Togl_CreateFunc( create_cb );
    Togl_DisplayFunc( display_cb );
    Togl_ReshapeFunc( reshape_cb );
    Togl_CreateCommand( "ResetView", reset_view_cb );
    return TCL_OK;
}
```

Tcl/Tk Summary

The approach of using Tcl/Tk for GUI construction and management while using C for computation and rendering is quite powerful:

- **The GUI may be changed without recompiling**
- **The C components allow efficient 3−D rendering.**

XForms

XForms is a free GUI toolkit built on top of X. Based on the original IRIS GL−based FORMS library. XForms includes a rudimentary OpenGL canvas widget.

- **XForms advantages:**
  - **Free**
  - **Simple**
- **XForms disadvantages:**
  - **Not as powerful as Motif or Tcl/Tk.**
  - **OpenGL support is minimal.**

Open Inventor

Open Inventor is a high−level 3−D graphics toolkit built on OpenGL. It includes functions for creating 3−D windows and methods for accessing the underlying window system.

- **Open Inventor advantages:**
  - **Higher−level 3−D environment**
  - **Powerful cene graph**
  - **Direct manipulation/interaction support**
  - **Available for many systems**
- **Open Inventor disadvantages:**
  - **Not free (but a free work−alike is coming)**
  - **Still need a GUI toolkit for for real apps**

OpenGL++

- Proposed toolkit for OpenGL which offers higher−level organizational and rendering support.

  Still in planning stages at this time.

  Should be widely adopted by OpenGL licensees.

Other high−level OpenGL toolkits

- Commercial toolkits: IRIS Performer for visual simulation. ImageVision for image processing.
- There are Python bindings for OpenGL, Tk, and GLUT.
- An unofficial set of OpenGL bindings for Java are available from the University of Waterloo.
- MET++ is a C++ multi−media application framework for Unix/X which includes OpenGL support.
- Tiger : Tcl wrappers for OpenGL API so a 3−D application may be written with just a Tcl script.

Mesa

- Mesa is a free 3−D graphics library which uses the OpenGL API and semantics.
- Expands the range of systems which support OpenGL development and execution: old workstations, X terminals, PCs, etc.
- Not 100% equivalent to OpenGL. A few features are not implemented yet.
- Not as fast as commercial OpenGL implementations, but still quite usable.
- Hardware support is under development.

Mesa (2)

- Drivers available for X Window System, Microsoft Windows 95/NT, Macintosh, Amiga, NextStep, BeBox, others...
- With X, supports rendering on almost any X server, even monochrome.
- Implements OpenGL 1.1 API and several extensions.
- Source code is free. Users have tuned it to improve their application’s performance.

Next topic:

- OpenGL Development Choices
- *Portability and Interoperability*
- Off−screen Rendering

Portability and Interoperability

- Overview
- Source code
- OpenGL details
- Using extensions correctly
- GLX/X11 interoperability

Source code

- Modular source code:
  - Window system, widget toolkit and OpenGL interface (GLX, WGL) code.
  - OpenGL graphics code.
  - OS−specific code.
  - Core data structures, number crunching, event callbacks.

Source code

- Clean code:
  - Follow standards (POSIX, use STL?)
  - Write clean module interfaces. Callbacks very helpful.
  - Develop and test on multiple platforms.
  - Use OpenGL extensions correctly.

OpenGL details

Despite OpenGL’s clean design, well−defined specification and lack of subsetting, developers must be aware of possible *gotchas*:

- **Optional features**
- **Implementation limits**
- **Versions and extensions**

OpenGL details (2)

- **Optional features**
  - **Frame buffer alpha planes**
  - **Overlay/underlay planes**
  - **Aux buffers**
  - **Singel/double buffering**

OpenGL details (3)

- **Implementation Limits:**

  OpenGL spec calls for minimum requirements in many areas. Can’t assume that an arbitrary OpenGL implementation will offer more.

OpenGL details (4)

- **Example Limits:**
  - **Stacks (Modelview: 32, Projection: 2, Texture: 2, Attribute: 16)**
  - **Textures may be limited to 64x64**
  - **Max viewport may equal screen size**
  - **Stencil buffer may be one bit deep**
  - **Max curve control points may be 8**
  - **Pixel map size may be only 32 entries**

Using OpenGL extensions

- **Naming conventions**
- **Compile−time testing**
- **Run−time testing**
- **OpenGL version 1.1**
- **Microsoft OpenGL extensions**

Extension naming conventions (1)

Core extensions have names of the form:
GL_*type_name*. (GLX: GLX_*type_name*)

***type* may be EXT, SGI, SGIX, SGIS, IBM, DEC, MESA, etc.**

***name* is a lowercase character string**

Examples:

```
GL_EXT_polygon_offset
GL_SGIS_detail_texture
```

Extension naming conventions (2)

Extensions may add new constants and/or functions.

Constants and functions are suffixed with the extension type.

Examples:

```
GL_FUNC_ADD_EXT
GL_MIN_EXT, GL_MAX_EXT
GL_DETAIL_TEXTURE_2D_SGIS
glBlendEquationEXT()
glPolygonOffsetEXT()
```

Compile−time extension testing

The header file will define a preprocessor symbol with name of the extension:

```
#define GL_EXT_polygon_offset   1
```

Surround code which uses the extension with preprocessor conditionals:

```
#ifdef GL_EXT_polygon_offset
     glPolygonOffsetEXT( a, b );
#endif
```

Run−time extension testing

- The glGetString(GL_EXTENSIONS) function returns a list of extensions supported by the renderer.
- Must be called <u>after</u> a rendering context has been made current.
- Be wary of using strstr() for searching the extension list string!

Extension fall−back scenerios

- **Disable:** If the GL_SGIS_multisample extension is not available, disable antialiasing.
- **Work−around:** If the GL_EXT_vertex_array extension isn't available use ordinary glVertex*() calls.
- **Abort:** If your volume visualization program depends on the GL_EXT_texture_3D extension you may have no choice but to abort. A last resort and discouraged!

Extension example: vertex arrays

- Determine if extension is available:

```
GLboolean HaveVertexArray = GL_FALSE;

/* MakeCurrent() must have already been called! */
#ifdef GL_EXT_vertex_array
   char *extensions = glGetString(GL_EXTENSIONS);
   if (strstr(extensions,"GL_EXT_vertex_array")) {
      HaveVertexArray = GL_TRUE;
   }
#endif
```

Note: See course notes for the CheckExtension() function to use instead of strstr().

Extension example: vertex arrays (2)

```
void DrawTriangleStrip(const GLfloat v[][3], GLuint n)
{
   if (HaveVertexArray) {
#ifdef GL_EXT_vertex_array
      glVertexPointerEXT( 3, GL_FLOAT, 0, n, v );
      glDrawArraysEXT( GL_TRIANGLE_STRIP, 0, n );
#endif
   }
   else {
      int i;
      glBegin( GL_TRIANGLE_STRIP );
      for (i=0;i<n;i++)
         glVertex3fv( v[i] );
      glEnd();
   }
}
```

Extensions and OpenGL 1.1

A number of extensions from OpenGL 1.0 are now standard features of OpenGL 1.1.

Problem: How to accommodate 1.0, extensions, and OpenGL 1.1

Example: the 1.0 extension function glBindTextureEXT() is called glBindTexture() in OpenGL 1.1.

Extensions and OpenGL 1.1 (2)

- **At compile time, also look for GL_VERSION_1_1 preprocessor symbol:**

```
if (HaveTextureObjects) {
#if defined(GL_VERSION_1_1)
    glBindTexture(GL_TEXTURE_2D, t);
#elif defined(GL_EXT_texture_object)
    glBindTextureEXT(GL_TEXTURE_2D, t);
#endif
}
else {
    // fall-back code
}
```

Extensions and OpenGL 1.1 (3)

- **At runtime, call glGetString(GL_VERSION) to determine if renderer supports OpenGL 1.1:**

```
GLboolean HaveTextureObjects = GL_FALSE;
GLubyte *version =glGetString(GL_VERSION);
if (strncmp((char*)version,"1.1",3)==0) {
    HaveTextureObjects = GL_TRUE;
}
```

Extensions and OpenGL 1.1 (4)

- **Dealing with extensions and OpenGL 1.1 can be messy.**
- **Best approach is probably to abstract the use of extensions or 1.1 features into functions which can hide the ugliness from your main code.**
- **In other cases, the C preprocessor can be useful for resolving naming differences.**
- **See course notes for details.**

Microsoft OpenGL extensions

- **Unfortunately, Microsoft OpenGL and SGI Cosmo OpenGL extensions are even more complicated.**

  An extension function can't be called directly as it may not exist in the OpenGL DLL.

  Instead, call function via pointer returned by *wglGetProcAddress()*.

Microsoft OpenGL extensions (2)

- **Example:**

```
#if defined(WIN32) && defined(GL_WIN_swap_hint)
   if (CheckExtension("GL_WIN_swap_hint")) {
      // The following type is found in the GL/gl.h file:
      PFNGLADDSWAPHINTRECTWINPROC glAddSwapHintRectWIN;
      // Get pointer to function.
      glAddSwapHintRectWIN = (PFNGLADDSWAPHINTRECTWINPROC)
               wglGetProcAddress("glAddSwapHintRectWIN");
      // Call the function
      if (glAddSwapHintRectWIN) {
         (*glAddSwapHintRectWIN)(x, y, width, height);
      }
   }
#endif
```

GLX and GLU extensions and versions

- **The GLX and GLU libraries can also have extensions. Several versions of these libraries exist.**

  Do compile and run−time extension and version testing similar to core OpenGL.

  See course notes for details.

GLX/X11 interoperability

GLX extends the X protocol to allow remote OpenGL rendering in a network.

- **In principle, nothing special must be done in a GLX application to support this.**
- **In practice, there are a number of issues to be aware of to be sure the application is robust and well−behaved.**

GLX/X11 interoperability (2)

- **Issues involved:**

  GLX Visuals (Mesa compatibility)

  Colormaps

  Double/single buffering

  Alpha planes

GLX Visuals (and Mesa)

- **Typically, glXChooseVisual() is used to select a GLX visual.**

GLX spec says:

Color index mode − return PseudoColor or StaticColor visual

RGB mode − return TrueColor or DirectColor visual

Mesa:

RGB mode − may return any visual type. Be prepared for that.

Colormaps

Different colormap strategies for RGB vs color index mode.

- **RGB mode − usually never alter the colormap entries**
- **CI mode − may or may not need to alter colormap entries**

In either case, want to avoid colormap *flashing* by sharing colormaps.

Colormaps (2)

Colormap flashing occurs when color demands exceed the hardware capabilities.

Common problem on low−end systems with only one hardware colormap.

Colormaps my be shared by windows using same visual type and depth.

Use default/root colormap when possible.

Colormaps (3)

- **For RGB mode:**

  If OpenGL visual matches root visual then Use root colormap. Mesa will manage to allocate all the colors it needs.

  Otherwise, look for a standard RGB colormap with XGetRGBColormaps().

  Last resort: Create new colormap with XCreateColormap(..., AllocNone)

Colormaps (4)

- **For Color Index mode:**

  Do you need to be able to store particular colors in particular colormap cells (lighting, fog, colormap animation)?

  If yes, you need a private, writable colormap.

  Otherwise, share an existing colormap and let X allocate colors or color cells for you.

Colormaps (5)

```
IF you need a private colormap THEN
    call XCreateColormap( ..., AllocAll)
    set colormap entries with XStoreColor().
ELSE
    IF GLX visual matches root/default visual THEN
        use root colormap
    ELSE
        XCreateColormap( .., AllocNone).
    ENDIF
    allocate read/write cells with XAllocColorCells()
    store colors into cells with XStoreColor()
    allocate read-only cells with XAllocColor()
    free colors or color cells with XFreeColors()
ENDIF
```

Colormaps (6)

Two more X colormap tips:

- **If XAllocColor() fails, get a copy of the colormap with XQueryColors() and search for closest match. See Mesa code for example.**
- **If your top−level window contains children with non−default colormaps inform the window manager with a call to XSetWMColormapWindows().**

GLX single / double buffering

- Be aware that GLX doesn't require the presence of both single and double buffered visuals.
- GLX may offer only single buffered visuals or only double buffered visuals.
- Write your glXChooseVisual() code with this in mind.
- Single buffering can be easily simulated with a double buffered visual by calling glDrawBuffer(GL_FRONT).

Mesa/X11 double buffering

- When using double buffering, Mesa can use either an X Pixmap or XImage as its back buffer.
- Use the MESA_BACK_BUFFER environment variable to determine which performs better with your application. This is especially important when remotely rendering.

GLX/X11 Alpha buffers

- Alpha (transparency) planes must be explicitly requested.
- If alpha planes are not supported in the hardware frame buffer they may be implemented in software -> slow.
- Alpha planes not needed for most transparency and blending effects.
- Mesa can simulate alpha planes.

Next topic:

- OpenGL Development Choices
- Portability and Interoperability
- *Off-screen Rendering*

Off−screen Rendering

Uses for off−screen rendering:

- **Intermediate image generation**
- **Hardcopy image generation**
- **Tiled rendering**

Off−screen Rendering (2)

Many ways to do off−screen rendering:

- **AUX buffers**
- **OpenGL for Microsoft Windows − device independent bitmaps**
- **GLX − GLX Pixmaps**
- **Mesa − Off−screen rendering API**
- **SGI Pbuffers**

AUX buffers

- **OpenGL spec defines auxillary (AUX) buffers.**

Request via glXChooseVisual() or ChoosePixelFormat()

Select with glDrawBuffer() and glReadBuffer()

Problem: available in few OpenGL implementations

DIBs and GLXPixmaps

Alternatives to AUX buffers:

- **Windows: device indepedent bitmaps (DIB)**
- **GLX: GLXPixmaps**

- **Create via window system−dependant functions.**
- **Bind OpenGL context to the buffer just like a window.**
- **Read back with glReadPixels.**

Problem: seldom hardware accelerated

Mesa: OSMesa interface

Mesa's off–screen rendering interface:

- **No operating system or window system dependencies. Very portable.**
- **Renders into a color buffer allocated by the client.**
- **Maximum size may be reconfigured.**

SGI Pbuffers

- **An SGI–only extension (GLX_SGIX_pbuffers)**
- **Auxilliary buffers allocated from frame buffer memory.**
- **Used in conjunction with the GLX_SGIX_fbconfig extension.**
- **Hardware accelerated!**
- **Difficult to use.**
- **Dependent on X.**

See course notes for example program.

Tiled Rendering

Often want to generate large, high–resolution images. For example: hard copy.

Problem: Maximum OpenGL image size limited by several factors:

- **Maximum window size.**
- **Maximum off–screen buffer size.**
- **Maximum viewport size (ex: 2k x 2k)**

Solution: tiled rendering– break large image into pieces then assemble pieces.

Tiled Rendering (2)

Difficulties in tiled rendering:

- **Must carefully setup projection matrix for each tile to avoid seam/edge artifacts in final image.**
- **Must manage memory carefully if generating very large images**
- **glRasterPos and glBitmap are troublesome.**

Tiled Rendering (3)

TR library makes it easy:

- **Takes care of projection and viewport arithmetic.**
- **Can automatically assemble final image from tiles.**
- **Allows access to intermediate tiles.**
- **trRasterPos() − solves glRasterPos problem.**
- **Generate arbitrarily large images without using lots of memory.**

Included on course notes CD−ROM with examples.

OpenGL & Window System Integration

"Most portable 3D, fastest 3D."

OpenGL

Mark J. Kilgard *Silicon Graphics, Inc.*
Brian Paul *Avid Technology*
Nate Robins *SGI, University of Utah, Parametric Technology*

SIGGRAPH '97 Course
August 4, 1997

Mark Kilgard

My background:

- **Author of *Programming OpenGL for the X Window System***
- **Member of Technical Staff at Silicon Graphics.**
- **Directly involved in implementation of SGI's X Window System support for OpenGL**
- **Implemented OpenGL Utility Toolkit (GLUT)**

It will help to know one or more of . . .

- **C programming**
- **Fundamentals of computer graphics**
- **Basics of OpenGL programming**
- **Xlib or Xt/Motif programming**
- **Windows programming**

Main Objective (entire course!):

Not learning how to use the OpenGL API and writing whizzy 3D programs...

But how to *properly integrate* whizzy 3D OpenGL programs with the your window system. Also, being portable & fast.

Often a neglected topics.

Basic X Topics

- OpenGL Integration for X
- OpenGL with Motif
- GLX Extensions

GLX Integrates X and OpenGL

- OpenGL = API for rendering
- Window management API left to the native window system
- With X Window System, Xlib and Xt/Motif = windowing API
- Still, X−specific OpenGL “binding” API between X calls and OpenGL needed
- Therefore, GLX.

Role of GLX

- OpenGL specification has no mention of the X Window System. GLX specifies how OpenGL and X interact.
- GLX is the “glue” between OpenGL and X.
- X server supports OpenGL if GLX is on its extension list.
- Wgl (pronounced “wiggle”) has a similar role for Windows NT. More on Wgl later.

GLX Functionality

- Extension queries.
- Visual selection.
- OpenGL context management.
- Pixmap handling.
- X font support.
- Buffer swapping.
- Synchronization.

GLX: API and Protocol

- **GLX is a programming interface (API).**
- **GLX routines begin with *glX* like *glXCreateContext***
- **GLX is *also* an X extension protocol.**
- **Protocol provides inter−vendor interoperability and network transparency.**
- **GLX API hides GLX protocol for OpenGL and GLX calls.**

When OpenGL routines are called...

- **Typical OpenGL routine call:**

  glEnable(GL_DEPTH_TEST);

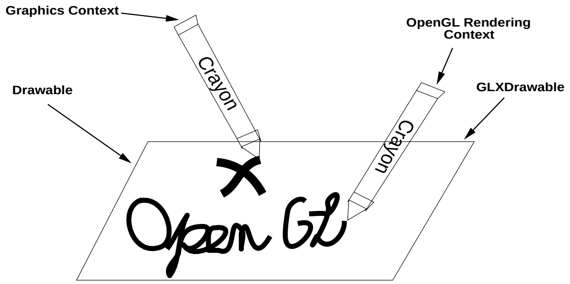
- **Notice no window destination specified.**
- **Also no X display connection specified.**
- **Also no context for OpenGL state.**
- **This information is implicit for each OpenGL call.**

GLX Contexts and Making Current

- **Programs use *glXCreateContext* to create OpenGL rendering context.**
- **Rendering context = instance of an OpenGL state machine.**
- **Programs use *glXMakeCurrent* to bind to context and OpenGL−capable drawable.**
- **Once bound, OpenGL calls render to current drawable using current context.**

Analogy of Rendering Models

- **OpenGL Rendering Context :: X Graphics Context :: Crayon**
- **GLXDrawable :: Drawable :: Paper**



Types of GLXDrawables

- **On−screen X windows.**

  Not every window has to be OpenGL capable though.

- **Off−screen GLXPixmaps.**

  A GLXPixmap is an “enhanced” version of a standard X pixmap.

- **Different GLXDrawables can have different frame buffer capabilities.**

Choosing frame buffer capabilities

- **Core X11 protocol uses “visuals” to abstract methods of mapping pixel values to color values at various depths.**

  Example: 24−bit TrueColor window.

- **OpenGL has frame buffer capabilities not known by core X.**

  Example: depth buffer, stencil buffer, double buffering, stereo

  OpenGL overloads X visuals with new info.

Some frame buffer capabilities

- **OpenGL−capable (all visuals don’t have to be!).**
- **Color index vs. RGBA color model.**
- **Bits of image resolution.**
- **Buffers: stencil, depth, accumulation.**
- **Double buffering.**
- **Frame buffer level (overlays, underlays).**

GLX Visual Attributes

Attribute	Type	Notes
GLX_USE_GL	boolean	true if OpenGL rendering is supported
GLX_BUFFER_SIZE	integer	depth of the color buffer
GLX_LEVEL	integer	frame buffer level: >0=overlay
GLX_RGBA	boolean	true if in RGB mode
GLX_RED_SIZE	integer	number of bits of red in RGB mode
GLX_GREEN_SIZE	integer	number of bits of green in RGB mode
GLX_BLUE_SIZE	integer	number of bits of blue in RGB mode
GLX_ALPHA_SIZE	integer	number of bits of alpha in RGB mode
GLX_DOUBLEBUFFER	boolean	true if front/back color buffers pairs
GLX_STEREO	boolean	true if left/right color buffers pairs
GLX_DEPTH_SIZE	integer	number of bits in the depth buffer
GLX_STENCIL_SIZE	integer	number of bits in the stencil buffer
GLX_AUX_BUFFERS	integer	number of auxiliary color buffers
GLX_ACCUM_RED_SIZE	integer	accumulation buffer red component
GLX_ACCUM_GREEN_SIZE	integer	accumulation buffer green component
GLX_ACCUM_BLUE_SIZE	integer	accumulation buffer blue component
GLX_ACCUM_ALPHA_SIZE	integer	accumulation buffer alpha component

(Further discussed when *glXChooseVisual* and *glXGetConfig* are introduced.)

X visuals advertise configurations

- Frame buffer configuration = supported set of OpenGL frame buffer capabilities.
- A given X server supporting OpenGL *enumerates* all its frame buffer configurations via its supported visuals.
- When an *XCreateWindow* is performed with a given visual, the new window supports the frame buffer configuration of the its visual.
- The configuration (like the visual) is fixed for the lifetime of the X window.

Important Distinction

- Number and types of *frame buffer configurations* (and therefore *capabilities*) can vary by OpenGL implementation.

  Depends on available hardware.
- But, all OpenGL *rendering capabilities* are <u>mandated</u> for all implementations.
- Still, GLX mandates high base−line of minimum guaranteed frame buffer configurations.

Frame buffer functionality baseline

- Every GLX−capable X server must provide at least one OpenGL−capable RGBA visual with at least the following:

  stencil buffer at least 1−bit deep
  depth buffer at least 12−bits deep
  an accumulation buffer
- If color index provided, one OpenGL color index visual must have:

  stencil buffer at least 1−bit deep
  depth buffer at least 12−bits deep

Example of OpenGL Visuals

Indigo Entry workstation (SGI's lowest end graphics) exports the following 5 visuals with these capabilities:

VisualID: 20
depth=8, class=**PsuedoColor**,
bufferSize=8, level=normal, rgba=no,
doubleBuffer=no, stereo=no, auxBuffers=0,
depthSize=32 bits, **stencilSize=8 bits**,
accumulationBuffer=no

VisualID: 24
depth=4, class=**PsuedoColor**,
bufferSize=4, level=normal, rgba=no,
doubleBuffer=yes, stereo=no, auxBuffers=0,
depthSize=32 bits, **stencilSize=8 bits**,
accumulationBuffer=no

VisualID: 25
depth=2, class=**PsuedoColor**,
bufferSize=2, level=**overlay**, rgba=no,
doubleBuffer=no, stereo=no, auxBuffers=0,
depthSize=0 bits, stencilSize=0 bits,
accumulationBuffer=no

VisualID: 22
depth=8, class=**TrueColor**,
bufferSize=8, level=normal, rgba=yes (redSize=1,
greenSize=2, blueSize=1, alphaSize=0),
doubleBuffer=no, stereo=no, auxBuffers=0,
depthSize=32 bits, **stencilSize=8 bits**,
accumulationBuffer=yes (redSize=16,
greenSize=16, blueSize=16, alphaSize=16)

VisualID: 23
depth=4, class=**TrueColor**,
bufferSize=4, level=normal, rgba=yes (redSize=1,
greenSize=2, blueSize=1, alphaSize=0),
doubleBuffer=yes, stereo=no, auxBuffers=0,
depthSize=32 bits, stencilSize=8 bits,
accumulationBuffer=yes (redSize=16,
greenSize=16, blueSize=16, alphaSize=16)

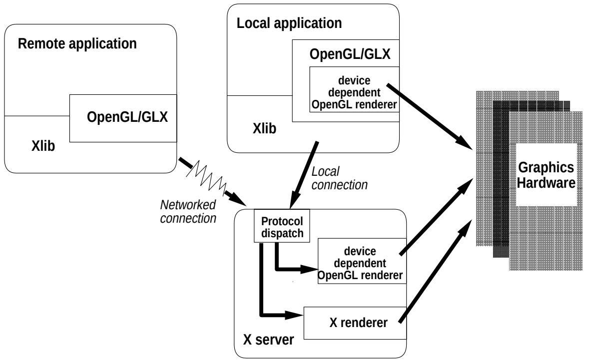
OpenGL Rending Contexts

- **OpenGL rendering context = full OpenGL state machine.**
- **Two options:**

  Indirect rendering **– uses GLX protocol, inter–operable, network–extensible, always supported.**

  Direct rendering **– higher local performance, direct access to hardware, not required.**

Direct & Indirect Rendering



GLX API Functionality (Part 1)

- **Extension queries:** *glXQueryExtension, glXQueryVersion, glXQueryExtensionsString, glXGetClientString, glXQueryServerString*
- **Visual selection:** *glXChooseVisual, glXGetConfig*
- **Context manipulation:** *glXCreateContext, glXCopyContext, glXDestroyContext*
- **Context/Drawable binding:** *glXMakeCurrent*
- **Context queries:** *glXGetCurrentContext, glXIsDirect*

GLX API Functionality (Part 2)

- **Drawable query:** *glXGetCurrentDrawable*
- **Buffer swapping:** *glXSwapBuffers*
- **Display listable X font support:** *glXUseXFont*
- **Synchronization:** *glXWaitGL, glXWaitX*

  GLX can also have API extensions, both standard and vendor supplied...

Header files for using OpenGL’s APIs

- **To get the OpenGL rendering API, use:**

```
#include <GL/gl.h>
```

- **To get the OpenGL GLX window system integration for X API, use:**

```
#include <GL/glx.h>
```

GLX Extension Queries

- **Does X server support OpenGL? Example:**

```
Display *dpy;
int error_base, event_base;
if(!glXQueryExtension(dpy, &error_base, &event_base))
     fatalError("no OpenGL GLX extension!");
```

- **Also, can query version of OpenGL/GLX. Example:**

```
Status status;
int major_vers, minor_vers;
status = glXQueryVersion(dpy, &major_vers, &minor_vers);
```

- **GLX 1.0, 1.1, and 1.2 are currently available.**

GLX 1.1

- **OpenGL 1.0 has mechanism to support API extensions to the basic OpenGL API.**
- **GLX 1.1 adds a similar mechanism to GLX.**

```
str=glXQueryExtension(dpy,screenNum);
```

- **GLX 1.1 adds no ‘‘real’’ functionality.**
- **Backward compatible.**

GLX 1.2: Most recent

- **One new call:**

```
dpy = glXGetCurrentDisplay();
```

  Fixes functionality oversight.

- **Mostly, provides the protocol specification and associated updates for OpenGL 1.1.**

Visual Selection

- *glXGetConfig* returns an OpenGL configuration for a specified visual. Example:

```
XVisualInfo *visual;
int value;

zeroIfSuccess = glXGetConfig(dpy, visual, GLX_USE_GL, &value)
if(value == True)
  printf("Visual 0x%x does GL\n",visual−>visualid);
```

- Examples of other configuration attributes:

GLX_USE_GL	True if OpenGL rendering supported
GLX_DEPTH_SIZE	Number of bits in the depth buffer

Quick and Dirty Visual Selection

- *glXChooseVisual* is ‘‘quick and dirty’’ visual selection routine.
- Example to find visual that supports double buffering, uses the RGBA color model, and has a depth buffer with at least 16 bits:

```
int configuration[] = { GLX_DOUBLEBUFFER, GLX_RGBA,
     GLX_DEPTH_SIZE, 16, None };
XVisualInfo *visual;

visual = glXChooseVisual(dpy, DefaultScreen(dpy),
     configuration);
```

Creating OpenGL Rendering Contexts

- Use glXCreateContext to create an OpenGL rendering context:

```
GLXContext context;
context = glXCreateContext(dpy,
     visual /* defines buffer resources of context */,
     NULL /* share context for display lists */,
     True /* try to create a direct context */);
```

- Note: contexts can share display lists.
- Note: a context’s visual must match the visual of drawables it can be bound to.

Destroying and Copying Contexts

- Use glXDestroyContext to destroy a created context:

```
glXDestroyContext(dpy, context);
```

- Contexts are expensive; recycle, don’t repeatedly create/destroy them.
- glXCopyContext allows a context’s OpenGL state to be copied to another context:

```
glXCopyContext(dpy, src_ctx, dest_ctx,
     /* copy everything */ GL_ALL_ATTRIB_BITS);
```

OpenGL rendering to pixmaps

- **To create a window for rendering OpenGL into, use Xlib's standard *XCreateWindow* routine; the window's visual determines the frame buffer configuration.**
- **But X pixmaps do not have visuals!**
- **OpenGL rendering is pretty limited without the benefit of ancillary buffers like a depth buffer.**
- **How do you render OpenGL into a pixmap then?**

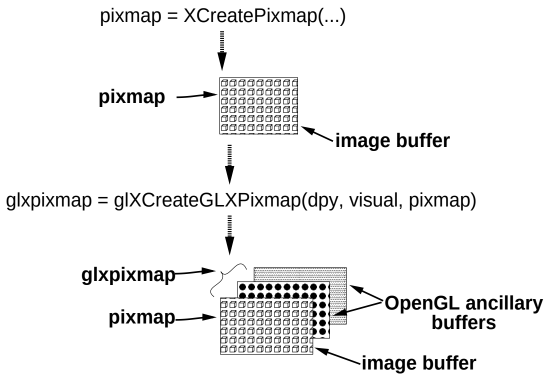
Handling GLXPixmaps

- **To render OpenGL into a pixmap, a *GLXPixmap* handle is created that "wraps" a pixmap created by *XCreatePixmap*. Example:**

```
XVisualInfo *visual;
Pixmap pixmap;
GLXPixmap glxpixmap;

pixmap = XCreatePixmap(dpy, DefaultRootWindow(dpy),
  width, height, depth);
glxpixmap = glXCreateGLXPixmap(dpy, visual, pixmap);
```

- **Draw core X rendering to *pixmap*, draw OpenGL rendering to *glxpixmap*.**

Wrapping a Pixmap for OpenGL



OpenGL's "make current" operation

- **OpenGL rendering commands do not take a *Display** or drawable or context per call.**
- **Instead, the current context and drawable are used.**
- ***glXMakeCurrent* establishes the current context and drawable for the calling thread:**

```
Display *dpy;
Window win;
GLXContext ctx;

glXMakeCurrent(dpy, win, ctx);
```

More about glXMakeCurrent

- Use *glXMakeCurrent* whenever you switch OpenGL rendering to a different context or drawable.
- Don’t call OpenGL API routines unless you are “made current.”
- You can unbind from a context and window by calling:

```
glXMakeCurrent(dpy, None, NULL);
```

Notes about GLX Contexts

- OpenGL rendering contexts are considered to “reside” in a given address space.

  Indirect contexts reside in the X server’s address space.

  Direct contexts reside in the application’s address space.

- Therefore, direct contexts can not be shared by distinct applications (though indirect contexts can).

GLX Queries

- `GLXDrawable glxdrawable;`

  `glxdrawable = glXGetCurrentDrawable();`

- `GLXContext context;`

  `context = glXGetCurrentContext();`

- `Display *display;`

  `display = glXGetCurrentDisplay();` ← GLX 1.2

-
  ```
  if(glXIsDirect(context))
        printf("this context is direct\n");
  ```

Buffer Swapping

- OpenGL supplies its own means to perform a buffer swap:

  `Window window;`

  `glXSwapBuffers(dpy, window);`

- Double buffering gets you seamless window updates.

Native X font usage by OpenGL

- **OpenGL has no native font support.**
- **The GLX API does supply a routine that turns X fonts into bitmap display lists so OpenGL and X can draw using the same bitmap fonts:**

```
Font font;
int first; /* first glyph to be used */
int count; /* number of glyphs */
int displayListBase; /* base display list ID */

font = XLoadFont(dpy, "fixed");
glXUseXFont(font, first, count,
    displayListBase);
```

Synchronizing X & OpenGL rendering

- **The command streams for X and OpenGL are considered separate.**
- **There is no guaranteed ordering for the execution of X request and OpenGL commands relative to each other.**
- **Two GLX routines allow efficient explicit synchronization:**

```
glXWaitX();
glXWaitGL();
```

Putting it all together

- **Now, we put the OpenGL, GLX, and Xlib APIs together.**
- **A short example that doesn't always do the smartest thing but demonstrates the basics...**
- **Start at the beginning, #includes:**

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <GL/gl.h>
#include <GL/glx.h>
```

- **Declare attribute lists to use with *glXChooseVisual*:**

```
static int configuration[] = {GLX_RGBA,
      GLX_DEPTH_SIZE, 16, None};
```

Continuing example

- **Declare variables:**

```
Display *dpy;
Window win;
GLXContext ctx;
XVisualInfo *visual;
Colormap cmap;
XSetWindowAttributes winattrs;
XEvent event;
```

- **Start main:**

```
main(int argc, char **argv)
{
```

- **Open X server connection:**

```
dpy = XOpenDisplay(NULL);
if(dpy == NULL) fatalError("bad DISPLAY");
```

Continuing example (2)

- **Find an appropriate visual:**

```
visual = glXChooseVisual(dpy,
      DefaultScreen(dpy), configuration);
if(visual == NULL)
      fatalError("no visual");
if(visual−>class != TrueColor)
      fatalError("expected TrueColor visual");
```

- **Create an OpenGL rendering context for visual:**

```
ctx = glXCreateContext(dpy, visual,
      NULL, /* go direct if possible */ True);
```

- **Impolite colormap strategy, just create one:**

```
cmap = XCreateColormap(dpy, RootWindow(dpy,
      visual−>screen), visual−>visual,
      AllocNone);
```

Continuing example (3)

- **Create the window with the correct visual; be careful, since it is likely not the default visual:**

```
winattrs.colormap = cmap;
winattrs.border_pixel = 0; /* avoid BadMatch */
winattrs.event_mask = StructureNotifyMask;
win = XCreateWindow(dpy, RootWindow(dpy,
      visual−>screen), 0, 0, 300, 300, 0,
      visual−>depth, InputOuput, visual−>visual,
      CWBorderPixel|CWColormap|CWEventMask,
      &winattrs);
```

- **Connect the context to the window:**

```
glXMakeCurrent(dpy, win, cx);
```

- **Map the window:**

```
XMapWindow(dpy, win);
```

Continuing example (4)

- **Wait for *MapNotify* event (assume *waitForNotify* was defined before *main*):**

```
static Bool
waitForNotify(Display *d, XEvent *e, char *arg)
{
      return (e−>type == MapNotify) &&
            (e−>xmap.window == (Window)arg);
}
```

- **Back in *main*...**

```
XIfEvent(dpy, &event, waitForNotify,
      (char*) win);
```

Continuing example (5)

- **Draw in the window using OpenGL; clear the window to red:**

```
glClearColor(1,0,0,1); /* red */
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
```

- **Sleep for a bit, then exit.**

```
sleep(10);
exit(0);
}
```

- **Greatly simplified, of course. Real application would have real X event loop and would do colormap selection better, etc., etc.**

Non−default Visuals

- **Note that it is likely that the visual you select is not the default visual.**
- **Be aware of the caveats about creating an X window with a non−default visual.**
- **When you create a top−level window not using the default visual, you can not inherit the colormap. You *must* specify a colormap created for your visual.**
- **Also the border pixel value *must* be specified; generally just supply 0.**

Event handling for OpenGL programs

- **The GLX extension adds no new events; still event handling for OpenGL programs has some caveats:**
- **An *Expose* event leaves the contents of all OpenGL ancillary buffers in the damaged region undefined.**

More Event handling for OpenGL

- **Usually OpenGL programs call *glViewport* to reshape the viewport of windows that receive a *ConfigureNotify* event indicating the window has been resized.**
- **Be aware the coordinate system origin for X is the upper−left corner; the origin for OpenGL's coordinate system is lower−left.**

  Translate button, keyboard, and motion event locations accordingly.

Basic X Topics

- **OpenGL Integration with X: GLX (mjk)**
- **OpenGL with Motif**
- **GLX Extensions**

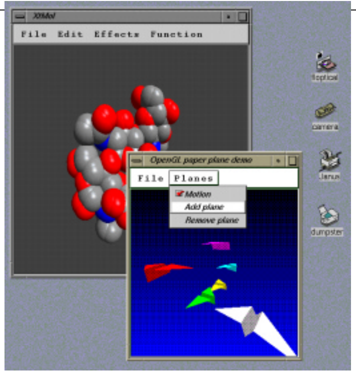
OpenGL with Motif

- Programmers typically combine OpenGL rendering with Motif user interface toolkit.
- Specialized OpenGL drawing area widgets make combining OpenGL and Motif relatively painless.
- Basic split:

User interface written using Motif.

3D OpenGL rendering done into special drawing area widgets.

Example of OpenGL and Motif



Motif Options

- OpenGL rendering into standard Motif drawing area widget. Involved.
- OpenGL rendering into specialized OpenGL drawing area widget. Fairly easy.

Using specialized OpenGL widget generally better option!

(Potential exists for more specialized OpenGL widgets. Open Inventor widgets are examples of this.)

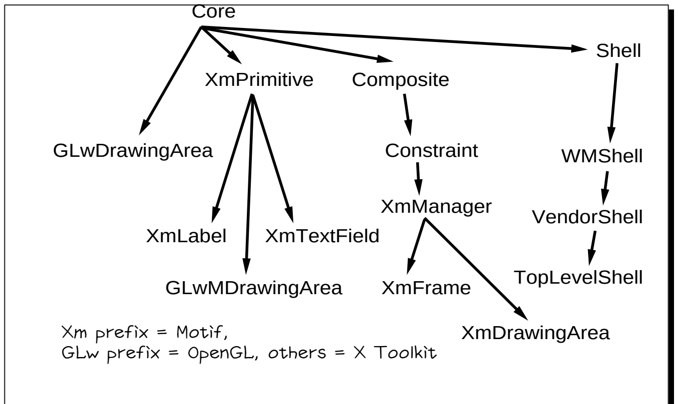
Why is a special widget needed?

- OpenGL relies on selecting appropriate visual for determining OpenGL frame buffer configuration.
- The X Toolkit (Xt) on which Motif relies, allows visual to be specified easily only for Shell and Shell−derived widgets.
- Non−shell widgets inherit visual from parent widget.
- Impossible (without resorting to widget internals) to set the visual of non−Shell Motif 1.2 widgets!

The OpenGL Widget(s)

- **Actually two OpenGL widgets!**
- ***GLwMDrawingArea* is Motif OpenGL widget.**
- ***GLwDrawingArea* is vanilla Xt OpenGL widget (notice lack of M) which can be used with non−Motif widget sets.**
- **Minor difference is the Motif OpenGL widget is derived from Motif's *XmPrimitive* widget.**

FYI: Partial Widget Class Hierarchy



Xm prefix = Motif,
GLw prefix = OpenGL, others = X Toolkit

Using an OpenGL widget

- **The Motif OpenGL widget header:**

  #include <X11/GLw/GLwMDrawA.h>

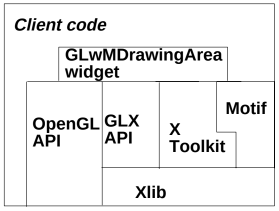
- **The vanilla OpenGL widget header (notice lack of M):**

  #include <X11/GLw/GLwDrawA.h>

- **Excepting obscure XmPrimitive resources, the two widgets are functionally equivalent.**

OpenGL widget software layering

- **Typical OpenGL widget program software layering:**



- **Typical library link options (in right order):**

  −lGLw −lGLU −lGL −lXm −lXt −lXext −lX11 −lm

Types of OpenGL widget resources

- ***Visual selection resources*: for selecting the appropriate OpenGL frame buffer configuration.**
- ***Callback resources*: for handling graphics initialization, resizes, exposes, and input.**
- ***Colormap management resources*: allocation of background and other colors, colormap installation.**

Visual selection resources

- ***GLwNvisualInfo*: allows particular *XVisualInfo** to specify visual. <u>Recommended</u>!**
- ***GLwNattribList*: contains list of GLX visual attributes to be passed to *glXChooseVisual*.**
- **Per−GLX visual attribute resources:**

  GLwNbufferSize
  GLwNrgba
  GLwNstereo
  GLwNredSize
  GLwNblueSize
  GLwNdepthSize
  GLwNaccumRedSize
  GLwNaccumBlueSize
  GLwNlevel
  GLwNdoublebuffer
  GLwNauxBuffers
  GLwNgreenSize
  GLwNalphaSize
  GLwNstencilSize
  GLwNaccumGreenSize
  GLwNaccumAlphaSize

Why GLwNvisualInfo recommended

- ***XtCreateWidget* has no ability to fail if described visual can not be found!**
- **The OpenGL widget terminates the program with a message if described visual not found.**

  (Limitation of X Toolkit!)
- **To guarantee described visual exists, call *glXChooseVisual* yourself (testing for failure), and then use *GLwNvisualInfo* to specify an explicit visual to use.**

Callback resources

- ***GLwNginitCallback*: called when widget is first realized. Good time to do OpenGL initialization.**
- ***GLwNresizeCallback*: called when the widget is resized. Good time to adjust OpenGL viewport, etc.**
- ***GLwNexposeCallback*: called when widget receives expose events. Redraw the scene.**
- ***GLwNinputCallback*: called in response to user input.**

OpenGL callback information

- The *call_data* structure passed to each OpenGL widget callback:

  ```
  typedef struct {
    int reason;
    XEvent *event;
    Dimension width, height;
  } GLwDrawingAreaCallbackStruct;
  ```

- *reason* is why callback called: *GLwCR_EXPOSE, GLwCR_RESIZE, GLwCR_INPUT*, & *GLwCR_GINIT*.
- *event* is X event that triggered callback; *NULL* for the *ginit* and *resize* callbacks.

Realizing Widgets & the ginit Callback

- X Toolkit does not create X window for widget until widget is realized.
- *XtWindow(widget)* will not return a valid window ID until window is realized.
- Therefore, you can not “make current” to a widget until realized.

  (Note: callbacks for some widgets can be called *before* a widget is realized like the resize callback!)
- The *GLwNginitCallback* helps you know when to start doing OpenGL state initialization, etc.

Creating GLXContexts for widgets

- It is your responsibility to call *glXCreateContext* to create OpenGL rendering contexts for use with widgets you create.
- Normally, this can be done before a widget is actually created since it does not require an X window, just the *XVisualInfo**.
- You can also wait to create OpenGL contexts until your ginit callback is called. Either works.
- How you share and use rendering contexts is up to you.

“Making current” for callbacks

- The OpenGL widget <u>does not</u> automatically perform a *glXMakeCurrent* before the callback.
- To make current, call:

  ```
  glXMakeCurrent(XtDisplay(widget),
                 XtWindow(widget), context);
  ```

- If there are multiple OpenGL drawing areas, you should <u>always</u> call *glXMakeCurrent* before calling any OpenGL routines within a widget callback.

The resize callback

- Typically used to change OpenGL viewport and possibly to update the projection matrix.
- Example:

```
void
resize(Widget w,
       XtPointer data, XtPointer callData)
{
  GLwDrawingAreaCallbackStruct *info =
    (GLwDrawingAreaCallbackStruct*) callData;

  glXMakeCurrent(XtDisplay(w),
    XtWindow(w), context);
  glViewPort(0, 0, info->width, info->height);
}
```

The expose callback

- Typically used to redraw the window's entire scene.
- Example:

```
void
redraw(Widget w,
       XtPointer data, XtPointer callData)
{
  glXMakeCurrent(XtDisplay(w),
    XtWindow(w), context);

  /* redraw the scene... */

  glXSwapBuffers(XtDisplay(w), XtWindow(w));
}
```

The input callback

- Typically used to handle user input for the window.
- As a programming convenience, by default, the OpenGL widget sets up the following translations:

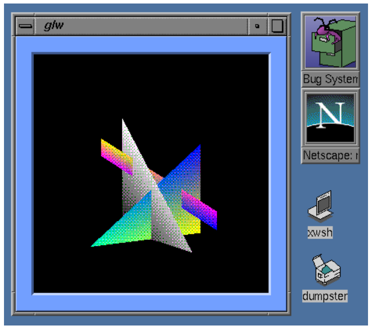
```
<KeyDown>:     glwInput()
<KeyUp>:       glwInput()
<BtnDown>:     glwInput()
<BtnUp>:       glwInput()
<BtnMotion>:   glwInput()
```

- *glwInput* calls the GLwNinputCallback.
- Alternate translations can be set up.

Widget colormap allocation

- If X colormap is not explicitly provided, OpenGL widget will attempt to allocate one.
- All widget instances share a colormap cache, so OpenGL widgets for the same visual will get assigned the same colormap.
- Good advice: allocate your own colormap and explicitly set it instead of letting widget do it for you. Better control!

An OpenGL widget example:



glw.c

glw.c demonstrates...

- Proper visual selection.
- Falling back to single buffering.
- Proper colormap allocation.
- Using *WorkProcs* for animation.
- Suspending *WorkProc* animation when iconified.
- OpenGL widget callback registration and handling.
- Handles indirect rendering.

glw.c (1): necessary headers

Necessary headers:

```
#include <stdlib.h>
#include <stdio.h>
#include <Xm/Form.h>           /* Motif Form widget */
#include <Xm/Frame.h>          /* Motif Frame widget */
#include <X11/GLw/GLwMDrawA.h> /* Motif OpenGL drawing area */
#include <X11/keysym.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>       /* for XA_RGB_DEFAULT_MAP */
#include <X11/Xmu/StdCmap.h> /* for XmuLookupStandardColormap */
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
```

glw.c (2): global variables

```
static int snglBuf[] = {GLX_RGBA,
                        GLX_DEPTH_SIZE, 16, None};
static int dblBuf[] =  {GLX_RGBA, GLX_DEPTH_SIZE, 16,
                        GLX_DOUBLEBUFFER, None};
static String fallbackResources[] = {
  "*glxarea*width: 300",      "*glxarea*height: 300",
  "*frame*x: 20",             "*frame*y: 20",
  "*frame*topOffset: 20",     "*frame*bottomOffset: 20",
  "*frame*rightOffset: 20",   "*frame*leftOffset: 20",
  "*frame*shadowType: SHADOW_IN", NULL
};
Display *dpy;
XtAppContext app;
XtWorkProcId workId = 0;
Widget toplevel, form, frame, glxarea;
XVisualInfo *visinfo;
GLXContext glxcontext;
Colormap cmap;
Bool doubleBuffer = True, spinning = False;
```

glw.c (3): initial main

Start of *main*, before OpenGL widget creation:

```
void main(int argc, char **argv)
{
  toplevel = XtAppInitialize(&app, "Glw", NULL, 0, &argc, argv,
    fallbackResources, NULL, 0);
  dpy = XtDisplay(toplevel);

  visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), dblBuf);
  if (visinfo == NULL) {
    visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), snglBuf);
    if (visinfo == NULL)
      XtAppError(app, "no good visual");
    doubleBuffer = GL_FALSE;
  }
  XtAddEventHandler(toplevel, StructureNotifyMask,
    False, map_state_changed, NULL);
  form = XmCreateForm(toplevel, "form", NULL, 0);
  XtManageChild(form);

  frame = XmCreateFrame(form, "frame", NULL, 0);
  XtVaSetValues(frame, XmNbottomAttachment, XmATTACH_FORM,
    XmNtopAttachment, XmATTACH_FORM, XmNleftAttachment, XmATTACH_FORM,
    XmNrightAttachment, XmATTACH_FORM, NULL);
  XtManageChild(frame);
```

glw.c (4): iconification & animation

Notice *main*'s *XtAddEventHandler* call.

An Xt *WorkProc* is used to control animation. Be sure to install and uninstall it on unmapping and unmapping of toplevel widget. Otherwise, useless rendering to unmapped window wastes CPU:

```
void map_state_changed(Widget w, XtPointer clientData,
                       XEvent * event, Boolean * cont)
{
  switch (event->type) {
  case MapNotify:
    if (spinning && workId != 0)
      workId = XtAppAddWorkProc(app, spin, NULL);
    break;
  case UnmapNotify:
    if (spinning)
      XtRemoveWorkProc(workId);
    break;
  }
}
```

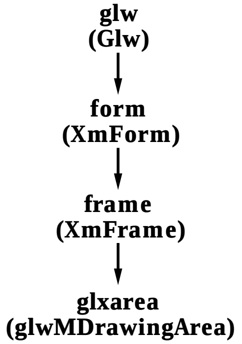
glw.c (5): OpenGL widget creation

Create your own colormap using the ICCCM colormap allocation conventions by calling Xt version of *getShareableColormap*.

Create *glwMDrawingArea* widget and add callbacks.

```
  cmap = getShareableColormap(visinfo);
  glxarea = XtVaCreateManagedWidget("glxarea",
    glwMDrawingAreaWidgetClass, frame,
    GLwNvisualInfo, visinfo, XtNcolormap, cmap, NULL);
  XtAddCallback(glxarea, GLwNginitCallback, init_callback, NULL);
  XtAddCallback(glxarea, GLwNexposeCallback, expose_callback, NULL);
  XtAddCallback(glxarea, GLwNresizeCallback, resize_callback, NULL);
  XtAddCallback(glxarea, GLwNinputCallback, input_callback, NULL);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(app);
}
```

glw's Widget Instance Hierarchy



glw
(Glw)

form
(XmForm)

frame
(XmFrame)

glxarea
(glwMDrawingArea)

glw.c (6): colormap allocation

Try to get a shared colormap:

```
Colormap getShareableColormap(XVisualInfo * vi) {
  XStandardColormap *standardCmaps;
  Colormap cmap; Status status; int i, numCmaps;

  /* be lazy; using DirectColor too involved for this example */
  if (vi−>class != TrueColor)
    XtAppError(app, "no support for non−TrueColor visual");
  /* if no standard colormap but TrueColor, just make an unshared one */
  status = XmuLookupStandardColormap(dpy, vi−>screen, vi−>visualid,
    vi−>depth, XA_RGB_DEFAULT_MAP, /* replace */ False,  /* retain */ True);
  if (status == 1) {
    status = XGetRGBColormaps(dpy, RootWindow(dpy, vi−>screen),
      &standardCmaps, &numCmaps, XA_RGB_DEFAULT_MAP);
    if (status == 1)
      for (i = 0; i < numCmaps; i++)
        if (standardCmaps[i].visualid == vi−>visualid) {
          cmap = standardCmaps[i].colormap;
          XFree(standardCmaps);
          return cmap;
        }
  }
  cmap = XCreateColormap(dpy, RootWindow(dpy, vi−>screen), vi−>visual, AllocNone);
  return cmap;
}
```

glw.c (7): ginit callback

Graphics initialization callback creates OpenGL context, binds context to widget’s window, and initializes OpenGL state:

```
void
init_callback(Widget w, XtPointer client_data, XtPointer call)
{
  XVisualInfo *visinfo;

  XtVaGetValues(w, GLwNvisualInfo, &visinfo, NULL);
  glxcontext = glXCreateContext(XtDisplay(w), visinfo,
    /* no sharing */ 0, /* direct if possible */ GL_TRUE);
  glXMakeCurrent(XtDisplay(w), XtWindow(w), glxcontext);
  /* setup OpenGL state */
  glEnable(GL_DEPTH_TEST);
  glClearDepth(1.0);
  glClearColor(0.0, 0.0, 0.0, 0.0); /* clear to black */
  glMatrixMode(GL_PROJECTION);
  gluPerspective(40.0, 1.0, 10.0, 200.0);
  glMatrixMode(GL_MODELVIEW);
  glTranslatef(0.0, 0.0, −50.0);
  glRotatef(−58.0, 0.0, 1.0, 0.0);
}
```

glw.c (8): resize callback

Resize callback updates OpenGL context’s viewport to reflect new window size:

```
void
resize_callback(Widget w,
                XtPointer client_data, XtPointer call)
{
  GLwDrawingAreaCallbackStruct *call_data;
  call_data = (GLwDrawingAreaCallbackStruct *) call;

  glViewport(0, 0, call_data−>width, call_data−>height);
}
```

glw.c (9): expose callback

Expose callback redraws the OpenGL widget’s window by calling the *draw* routine:

```
void
expose_callback(Widget w,
                XtPointer client_data, XtPointer call)
{
  draw();
}
```

Note: the common draw routine is also used when a redraw is generated by animation.

Two possible redraw reasons in glw:
1) expose events.
2) animation.

glw.c (10): draw routine

Draw routine renders OpenGL scene.

```
void
draw(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glColor3f(0.0, 0.0, 0.0);       glVertex3f(-10.0, -10.0, 0.0);
    glColor3f(0.7, 0.7, 0.7);       glVertex3f(10.0, -10.0, 0.0);
    glColor3f(1.0, 1.0, 1.0);       glVertex3f(-10.0, 10.0, 0.0);
  glEnd();
  glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 0.0);       glVertex3f(0.0, -10.0, -10.0);
    glColor3f(0.0, 1.0, 0.7);       glVertex3f(0.0, -10.0, 10.0);
    glColor3f(0.0, 0.0, 1.0);       glVertex3f(0.0, 5.0, -10.0);
  glEnd();
  glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 0.0);       glVertex3f(-10.0, 6.0, 4.0);
    glColor3f(1.0, 0.0, 1.0);       glVertex3f(-10.0, 3.0, 4.0);
    glColor3f(0.0, 0.0, 1.0);       glVertex3f(4.0, -9.0, -10.0);
    glColor3f(1.0, 0.0, 1.0);       glVertex3f(4.0, -6.0, -10.0);
  glEnd();
  if (doubleBuffer) glXSwapBuffers(dpy, XtWindow(glxarea));
  if (!glXIsDirect(dpy, glxcontext))
    glFinish();  /* avoid indirect rendering latency from queuing */
}
```

glw.c (11): input callback

Input callback starts and stops animation on key press:

```
void input_callback(Widget w, XtPointer clientData, XtPointer callData)
{
  XmDrawingAreaCallbackStruct *cd = (XmDrawingAreaCallbackStruct *) callData;
  char buffer[1];  KeySym keysym;

  switch (cd->event->type) {
  case KeyRelease:
    if(XLookupString((XKeyEvent *) cd->event, buffer, 1, &keysym, NULL) > 0) {
      switch (keysym) {
      case XK_S:  XK_s:       /* the S key */
        if (spinning) {
          XtRemoveWorkProc(workId);
          spinning = GL_FALSE;
        } else {
          workId = XtAppAddWorkProc(app, spin, NULL);
          spinning = GL_TRUE;
        }
        break;
      case XK_Escape:         /* the Escape key exists */
        exit(0);
    } }
    break;
} }
```

glw.c (12): spin WorkProc

Spin routine is registered as an Xt *WorkProc* to keep the scene spinning. Do rotate, redraw scene, keep *WorkProc* registered:

```
Boolean
spin(XtPointer clientData)
{
  glRotatef(2.5, 1.0, 0.0, 0.0);
  draw();
  return False;         /* leave work proc active */
}
```

Basic X Topics

- **OpenGL Integration with X: GLX (mjk)**
- **OpenGL with Motif**
- **GLX Extensions**

GLX Extensions

- Adds window system dependent functionality.
- Typically deal with new context handling or video capabilities or frame buffer capabilities.
- Capability for GLX extensions added with GLX 1.1.

Pbuffer extension (SGIX)

- Pbuffer = pixel buffer; new off−screen hardware accelerated drawable type.
- Brian talks about using these.
- A bit difficult to use.
- Often pbuffers are limited by hardware frame buffer memory limits.
- Available on RealityEngine, InfiniteReality, O2, Impact and Octane. Probably other vendors will support too.

FBconfig extension (SGIX)

- FBconfig = frame buffer configuration.
- FBconfigs are more general than X visuals.
- FBconfigs work for new non−window drawables like pbuffers.
- FBconfigs relax compatibility requires.
- FBconfigs permit off−screen drawable better than displayable window types.
- Likely for GLX 1.3.

Make Current Read extension (SGI)

- Normally, glCopyPixels copies from rectangle in current window to rectangle in the same window (source & destination drawable).
- glXMakeCurrentReadSGI allows a different source & destination drawable.
- Enables:
    - Window to window copies.
    - Pbuffer to window copies.
    - GLXPixmap to window copies, etc.

Import Context extension (EXT)

- Lets you share an indirect rendering context between multiple X connections.
- Limited usefulness.
- Easy to implement because of how GLX protocol works so easy for X vendors to support.
- See: glXImportContextEXT

Visual Info extension (EXT)

- Adds more frame buffer attributes.
- Permits matching on overlay transparency mode.
- Better control of X visual type selected.

Visual Rating extension (EXT)

- Adds one more frame buffer attributes.
- Indicates if an X visual or FBconfig has an caveats:

  None,

  Non−conformant, or

  Slow.
- Allows vendors to expose slow or non−compliant visuals and FBconfigs without confusing programs that probably don’t want caveated visuals.

Multisample extension (SGIS)

- Supports multisample antialiasing mode.
- Expensive; probably only available on high−end machines such as InfiniteReality and RealityEngine.
- No brainer way to eliminate (reduce) jaggies in your scenes. Greatly improves visual quality, particularly for animated scenes.
- New frame buffer attribute added.

More Advanced Topics

- **Advanced Topics: overlays, stereo, etc.**

Performance, Performance! (not just X)

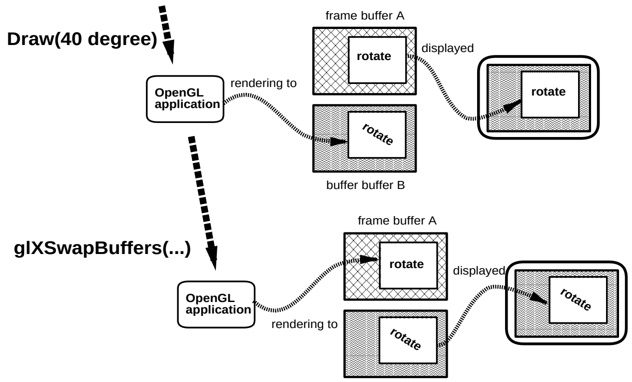
A Few Advanced Topics

- **Double Buffering**
- **Stereo**
- **Font Support**
- **Overlays**
- **OpenGL over a network**
- **Mixing 2D rendering with OpenGL**
- **Alternative Input Devices**

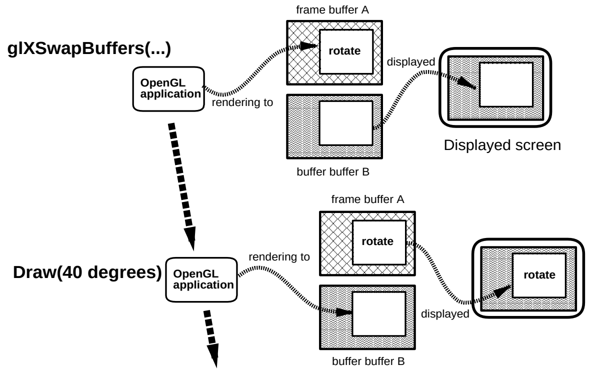
Double Buffering

- **OpenGL supports hardware double buffering. Call *glXSwapBuffers*.**
- **Visuals can be either exported as single or double buffered.**
- **The buffer naming scheme for OpenGL is "relative" scheme meaning buffers are referred to as *front* and *back*.**
- ***glDrawBuffer* determines what buffer gets drawn to.**

Double Buffering in Action (2)

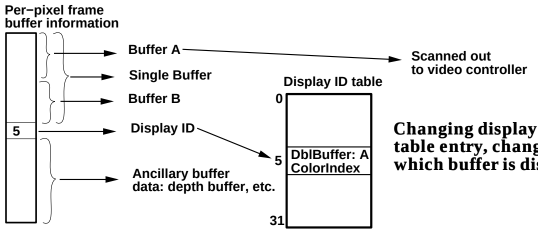


Double Buffering in Action (1)



Hardware Double Buffering

- Double buffer hardware usually splits the image data bits in a frame buffer into 2 buffers.
- Each pixel is usually assigned a display ID which indicates if the pixel is double buffered, and if so which buffer is currently displayed.



Changing display ID table entry, changes which buffer is displayed!

Hardware Double Buffering (2)

- For display ID style double buffer hardware, a buffer swap involves flipping the display buffer bit in the display ID table entry for the window.
- Normally this is synchronized with the vertical retrace to avoid any tearing artifacts.
- Sophisticated graphics systems also block the buffer swap initiator's further rendering to ensure no more rendering takes place until the swap has completed.

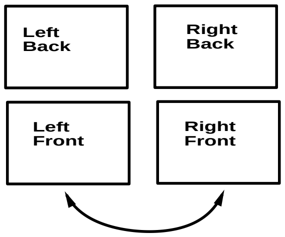
OpenGL Stereo

- Stereo support built into OpenGL standard.
- Model: “stereo in a window”
- Stereo window gets left & right color buffers.
- glDrawBuffer and glReadBuffer can choose left and/or right buffers.
- Almost always needs to be double buffered so left & right for front & back.

  4 buffers, so called “quad buffering”.
- Unfortunately, expensive style of stereo.

Quad buffered Stereo

Stereo OpenGL app renders scene twice from slightly different eye points.



Video display hardware switches between displaying left & right buffers, typically 120 Hz refresh rate. LCD shutter goggles show left or right based on stereo emitter signal.

Actually Using OpenGL Stereo

- **Render left eye, then right eye. Slight different eye points for stereo effect.**
- **One shared depth buffer. Left and right renders must be done serially.**
- **glXSwapBuffers swaps both left and right buffers.**
- **Nice, clean stereo model, but takes up twice the color buffer memory as a mono window.**

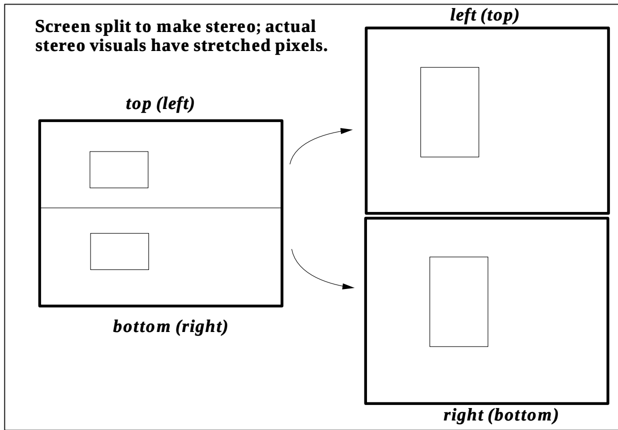
Cheaper OpenGL Stereo

- **Split from buffer into top & bottom half.**

  Top half is left; bottom half is right.
- **Special video format “streches” screen halves to fill entire screen (1x2 pixel aspect ratio).**
- **120 Hertz video refresh.**
- **Requires switching between two windows in top & bottom half of the screen.**
- **SGI has proprietary X server extension for this (does split screen stereo; X nicely draws into both top and bottom of frame buffer).**

Cheaper OpenGL Stereo Scheme

Screen split to make stereo; actual stereo visuals have stretched pixels.



Font Support

- *glXUseXFont* makes X fonts into an array of OpenGL bitmap display lists.
- These display lists can be called using *glCallLists* (and *glListBase*) to print out strings of text.
- Therefore, all available X fonts are available to OpenGL.
- Of course, X fonts are fairly limited since they are simply bitmaps in a single orientation, ie. limited utility within 3D scenes.

Using glXUseXFont generated fonts

- *glXUseXFont* makes X fonts into an array of OpenGL bitmap display lists.

```
Font xfont;
GLuint font_base;

xfont = XLoadFont(dpy, "fixed");
if(xfont == NULL) fatalError("font not found.");
base = glGenLists(128 /* 7-bit ASCII range */);
glXUseXFont(xfont, 0, 128, base);
```

- Then, render a string by calling:

```
output_string(int x, int y, char *string) {
  glRasterPos2i(x, y);
  glListBase(base);
  glCallLists(strlen(string), GL_UNSIGNED_BYTE,
    (GLubyte *)string);
}
```

More sophisticated fonts

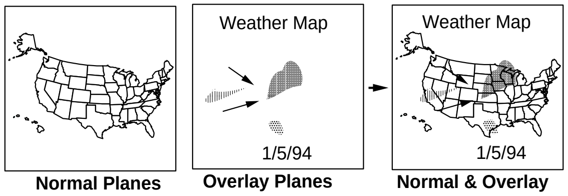
- OpenGL programmers are not limited to using X bitmap fonts via *glXUseXFont*.
- OpenGL's rendering facilities are well suited to other more sophisticated font rendering techniques:

  scalable outline fonts
  scalable stroke fonts
  anti−aliased fonts
  texture mapped fonts

- New GLC (OpenGL Character) API supports font rendering via OpenGL.

Overlays

- An overlay is an alternate set of frame buffer bitplanes that can be preferentially displayed instead of the standard bitplanes.
- Imagine a stack of frame buffer layers with transparent pixel values. Example:



X/OpenGL's overlay support

- OpenGL considers frame buffer layers to exist in a *single* window hierarchy.
- One of the OpenGL visual attributes is GLX_LEVEL that indicates what frame buffer layers the visual belongs to (0=normal, >0=overlay).
- OpenGL is compatible with the *SERVER_OVERLAY_VISUALS* convention. Used by SGI, HP, and others; Sun now supporting it!
- OpenGL doesn't advertise a transparent pixel value; either get it from the *SERVER_OVERLAY_VISUALS* property.

Creating an Overlay Sandwich

- To support the weather map example of using the overlay planes, do the following:
- Create a normal plane window.
- Create a subwindow with an overlay plane visual with the same size and position. Set the background pixel to be the overlay's transparent pixel value.
- Only select for input events in the subwindow.
- For OpenGL rendering, use *glXMakeCurrent* to switch between windows.

Efficient 3D over the network

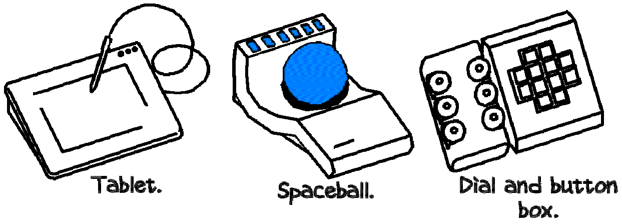
- OpenGL supports non−editable display lists.
- Display lists can reside in the X server to efficiently execute large batches of OpenGL rendering commands.
- Display lists and immediate mode can be mixed.
- If running efficiently over the network is important to your 3D application, use display lists.

Mixing GUI and OpenGL rendering

- Most OpenGL X applications will use Motif or some other X toolkit for their user interface needs.
- The use of OpenGL is generally limited to 3D application windows. The buttons and scroll bars continue to be using core X rendering.
- This makes good sense; segregating OpenGL and X rendering by windows avoids the overhead of synchronizing the OpenGL and X execution streams.

OpenGL and the X Input Extension

- **OpenGL applications often want access to sophisticated input devices like:**



- **The X Input extension provides access to such devices.**

Basics of the X Input extension

- **Distinct extension to the X server. Query for it separately from OpenGL GLX extension.**
- **The X Input extension augments the input events generated by the core X11 protocol.**
- **Header file for the X Input extension API:**

```
#include <X11/extensions/XInput.h>
```

- **Required library link options for using X Input extension:**

```
-lXi -lXext -lX11
```

Using the X Input Extension

- **Query for server's support of the extension using *XGetExtensionVersion*.**
- **List available input devices using *XListInputDevices* and determine what devices to use.**
- **Call *XOpenDevice* to open desired devices.**
- **Determine device event types and classes, then select desired events using *XSelectExtensionEvent*.**
- **Get XInput events by calling *XNextEvent*.**

Next topic:

Advanced Topics: overlays, stereo, etc.

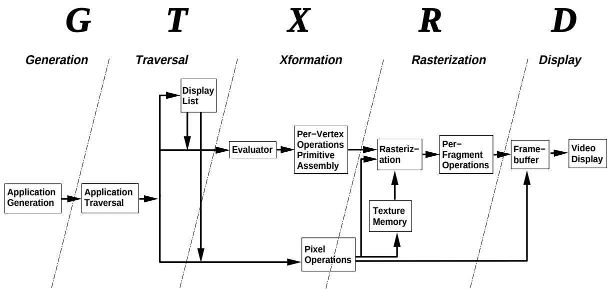
- **Performance, Performance! (not just X)**

Pipeline Oriented Tuning

- **Most computer programs are tuned based on “hot−spots.”**
- **“80% of the time is spent in 20% of the code.”**
- **With graphics hardware, you need to think about “pipeline” oriented tuning...**

The Graphics Pipeline

Akeley’s taxonomy



What work is off−loaded to gfx hardware?

Example Architectures

- **“Dumb frame buffer” (VGA)**

  GTXR−D

- **Silicon Graphics Indy**

  GTX−RD

- **Silicon Graphics Indigo ^{2} IMPACT**

  GT−XRD

Tuning a Pipeline

Two basic ideas:

- **Keep the graphics pipeline busy.**
- **Keep the graphics pipeline balanced.**

Maximizing performance

High−level issue:
How can I structure my application to achieve maximum performance?

Low−level issue:
How do I get the best performance from OpenGL?

Maximizing performance (2)

High−level techniques:
- **Multiprocessing (IRIS Performer)**
- **Image quality vs performance**
- **Culling and level of detail management**

Low−level techniques:
- **Efficient data structures**
- **Efficient traversal**
- **Careful use of OpenGL features**

High−level techniques

Multiprocessing
Perform rendering, computation, database generation in separate threads.

Image quality vs performance
Use high−resolution model and features for static images, low−resolution model and simpler features for animation.

Level of detail management and culling
Monitor application performance and modify database to meet minimum frame rate.

Low−level techniques

Efficient data structures and traversal

Maximize vertices between glBegin/glEnd.

Minimize extraneous code between glBegin/glEnd.

Store vertex data with zero stride in compact representations.

Use efficient forms of glVertex, glColor, etc.

Example: data structs and traversal

Drawing cities for a road map:

```
#define VILLAGE 1
#define CITY 2

struct city {
      float latitude, longitude;
      int size;   /* VILLAGE or CITY */
};
```

Want to draw a small dot for villages and a large dot for cities.

Ex: data structures and traversal (2)

Poor implementation:

```
void draw_cities( int n, struct city list[] )  {
      for (i=0;i<n;i++) {
            glPointSize( list[i].size==CITY ? 3.0, 1.0 );
            glBegin( GL_POINTS );
            glVertex2f( list[i].latitude, list[i].longitude );
            glEnd();
      }
}
```

Ex: data structures and traversal (3)

Better implementation:

```
void draw_cities( int n, struct city list[] )  {
      glPointSize( 1.0 );
      glBegin( GL_POINTS );
      for (i=0;i<n;i++)
            if (list[i].size==VILLAGE)
               glVertex2f(list[i].latitude,list[i].longitude);
      glEnd();
      glPointSize( 3.0 );
      glBegin( GL_POINTS );
      for (i=0;i<n;i++)
            if (list[i].size==CITY)
               glVertex2f(list[i].latitude,list[i].longitude);
      glEnd();
}
```

Ex: data structures and traversal (4)

Better yet− a new data structure and drawing function:

```
float cities[MAX][2];
float villages[MAX][2];

void draw_cities( int n, int size, float position[][2] )  {
      glPointSize( size==CITY ? 3.0, 1.0 );
      glBegin( GL_POINTS );
      for (i=0;i<n;i++)
            glVertex2fv( position[i] );
      glEnd();
}
```

(even better, the vertex array extension)

OpenGL optimization

- **Traversal**
- **Transformation**
- **Rasterization**
- **Texturing**
- **Clearing**
- **Miscellaneous**
- **Window system integration**
- **Mesa−specific**
- **Hardware/implementation−specific**

OpenGL optimization: traversal

Use connected primitives (triangle and line strips).

Store vertex data in consecutive memory locations.

Use the vector versions of glVertex, glNormal, glColor, and glTexCoord.

Use the vertex arrays (extension or OpenGL 1.1).

Reduce the number of primitives (tessellation).

Use display lists.

Don’t specify unneeded per−vertex data. (texcoords)

Minimize extraneous code between glBegin/glEnd.

OpenGL optimization: transformation

Disable normal vector normalization when not needed.

Use long connected primitives such as GL_LINE_STRIP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, and GL_QUAD_STRIP.

Don’t over−tessellate your primitives (NURBS, spheres, etc).

Use efficient forms of glVertex, glNormal, etc. such as glVertex3fv and glNormal3fv.

Disable clipping planes that aren’t needed.

OpenGL optimization: lighting

Avoid positional lights.
Avoid spotlights.
Avoid two−sided lighting.
Avoid negative material and light coefficients.
Avoid using the local viewer option.
Avoid frequent changes to GL_SHININESS.
Some implementations optimized for using a single light source.
Consider pre−lighting your model.
Don’t use too many light sources.
Avoid frequent material changes.

OpenGL optimization: rasterization

- Disable smooth shading when not needed.
- Disable depth testing when not needed.
- Disable dithering if not needed (esp. glClear).
- Use polygon culling whenever possible.
- Use few large polygons rather than many small polygons to reduce raster setup time.
- Avoid extra fragment operations such as scissoring, stenciling, blending, stippling, alpha testing and logic operations.
- Reduce your window size or screen resolution.
- Use integer glPixelZoom() values.
- Antialiased lines of width 1 often optimized.

OpenGL optimization: texturing

- Use efficient image formats such as GL_UNSIGNED_BYTE or one of the internal packed formats optimized for your hardware.
- Use fewer texture color components.
- Encapsulate textures in texture objects or display lists to reduce binding time.
- Use simple sampling functions such as GL_LINEAR and GL_NEAREST.
- Use a simple texture environment function such as GL_DECAL instead of GL_MODULATE for 3−component textures.
- Compile many small textures into one larger texture and use offset texture coordinates to address them.
- Use smaller texture maps.
- Pre−dither or pre−light textures to avoid dithering and lighting.

OpenGL optimization: clearing

- Be aware glClear takes a bitmask; don’t use multiple glClear calls.
- Disable dithering before clearing.
- Use scissoring to limit clearing to subregions.
- Don’t clear the color buffer at all if redrawing the entire window.
- Eliminate depth buffer clearing if redrawing entire window:

```
if (EvenFlag) {
      glDepthFunc( GL_LESS );
      glDepthRange( 0.0, 0.5 );
} else {
      glDepthFunc( GL_GREATER );
      glDepthRange( 1.0, 0.5 );
}
```

OpenGL optimization: misc (1)

- Avoid round trip calls such as glGet*(), glIsEnabled() and glGetString() in your rendering loop.
- Avoid glPushAttrib(), especially with GL_ALL_ATTRIB_BITS.
- Use glColorMaterial() instead of glMaterial() for frequent material changes.
- Avoid using viewports which are larger than the window.
- Check for GL errors during development with glGetError().

OpenGL optimization: misc (2)

Don’t allocate alpha, stencil, accumulation, or overlay planes unless you really need them.

Try implementing transparency with stippling instead of blending.

Avoid using glPolygonMode() for drawing unfilled polygons. glBegin(GL_LINE_LOOP) may be faster.

Group GL state changes together.

Be aware of your depth buffer’s depth (ex 16 vs 32−bit) and your hardware’s optimized configuration.

Optimizations: window system

Minimize calls to the *MakeCurrent* function. Context switching is expensive.

Be aware of tradeoffs in visual/pixel formats with respect to precision (bits) versus speed.

Avoid mixing OpenGL rendering with native window system (X11) rendering in the same window.

Don’t redraw more often than needed. Example: X expose events often come in groups.

Be aware that *SwapBuffers* may stall the graphics pipe until the next vertical retrace.

Mesa optimizations

Double buffering may be implemented with an XImage or Pixmap. Experiment to learn which is faster for you.

Some X visuals can be rendered into quicker than others (8−bit vs 24−bit).

Mesa supports 16 or 32−bit depth buffers. 16−bit is usually faster but may not be not precise enough for some applications.

When drawing constant, flat shaded primitives put the glColor call before the glBegin call.

The GLubyte versions of glColor are the fastest.

The GLfloat versions of glVertex, glNormal, and glTexCoord are the fastest.

See the README file for optimized rendering combinations.

System−specific optimizations

Read your vendor’s release notes and documentation carefully to learn the optimal parameters of your hardware and OpenGL: lengths of triangle strips, texture sizes, texture formats, pixel depths, etc.

Use the glGetString(GL_RENDERER) call to test for specific hardware configurations and use specialized OpenGL code.

Write test programs to determine what’s fast and slow or to compare relative speeds of different code fragments.

OpenGL & Window System Integration

"Most portable 3D, fastest 3D."

OpenGL

Mark J. Kilgard *Silicon Graphics, Inc.*
Brian Paul *Avid Technology*
Nate Robins *SGI, University of Utah, Parametric Technology*

SIGGRAPH '97 Course
August 4, 1997

Nate Robins

My background

- **Worked for Evans & Sutherland in the Graphics Systems Group**
- **Worked for Parametric Technology porting Pro/3DPAINT to Windows NT.**
- **Currently an Intern at SGI**
- **Ported the OpenGL Utility Toolkit (GLUT) to Windows NT/95.**

OpenGL & Win32 Topics

- **A simple example to get started**
- **Processing messages & using menus**
- **Pixel formats & palettes**
- **Overlays & underlays**
- **WGL reference**

A simple example

Three basic steps

- **Create a window**
- **Set the pixel format**
- **Create a rendering**

Creating a window

Two-fold process

- Register a window class
- Create a window in the new class

Registering a window class

What is a window class?

- A template used to create a window in an application
  - specifies certain basic attributes (such as window procedure)
  - identified by character string name
  - every window must be associated with a class

Registering a window class (2)

How do I register a window class?

- Fill in a WNDCLASS structure

```
WNDCLASS wc;

wc.style         = 0;                             /* no special styles */
wc.lpfnWndProc   = (WNDPROC)WindowProc;           /* message handler */
wc.cbClsExtra    = 0;                             /* no extra class data */
wc.cbWndExtra    = 0;                             /* no extra window data */
wc.hInstance     = GetModuleHandle(NULL);         /* instance */
wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO);   /* load a default icon */
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);   /* load a default cursor */
wc.hbrBackground = NULL;                          /* redraw our own bg */
wc.lpszMenuName  = NULL;                          /* no menu */
wc.lpszClassName = "OpenGL";                      /* use a special name */
```

  - The hbrBackground member should be NULL
  - The lpszClassName can be any character string

Registering a window class (3)

How do I register a window class?

- Call the RegisterClass() function

```
RegisterClass(&wc);
```

  - returns TRUE on success, FALSE if an error occurred
  - when the application that registered a window class exits, the window class is destroyed
  - see the course notes for a more complete example

Creating a window from the class

- **Call CreateWindow() function**

```
HWND hWnd;

hWnd = CreateWindow("OpenGL",               /* class */
                    "Simple Example",       /* title (caption) */
                    WS_CLIPSIBLINGS |
                    WS_CLIPCHILDREN,        /* style */
                    x, y, width, height,    /* dimensions */
                    NULL, NULL,             /* no parent, no menu */
                    GetModuleHandle(NULL),  /* instance */
                    NULL);                  /* nothing for WM_CREATE */
```

  - **The style argument must include the WS_CLIPSIBLINGS and WS_CLIPCHILDREN attributes**

- **Final preparation forthe window**

```
ShowWindow(hWnd, SW_SHOW);
/* send an initial WM_PAINT message (expose) */
UpdateWindow(hWnd);
```

Setting the pixel format

What is a pixel format?

- **Specifies properties of a rendering context**
  - **number of color bits**
  - **depth of the Z buffer**
  - **single/double buffered**
  - **number of stencil bits**
  - **etc**

Setting the pixel format (2)

How do I set the pixel format?

- **Simplest method is to fill out a PIXELFORMATDESCRIPTOR and call the ChoosePixelFormat() command**

```
HDC hDC;
PIXELFORMATDESCRIPTOR pfd;

pfd.nSize       = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion    = 1;                  /* version (should be 1) */
pfd.dwFlags     = PFD_DRAW_TO_WINDOW | /* draw to window (not bitmap) */
                  PFD_SUPPORT_OPENGL,  /* draw using opengl */
pfd.iPixelType  = PFD_TYPE_RGBA;       /* PFD_TYPE_RGBA or COLORINDEX */
pfd.cColorBits  = 24;

pf = ChoosePixelFormat(hDC, &pfd);
```

  - **returns a valid pixel format index on success, 0 if none match**
  - **More on this later! (better methods)**

Creating a rendering context

What is a rendering context?

- **Port through which all OpenGL commands pass**
- **Link between OpenGL and Windows NT/95 windowing systems**
- **Win32 context has the type HGLRC (analog in X is the GLXContext)**

Creating a rendering context (2)

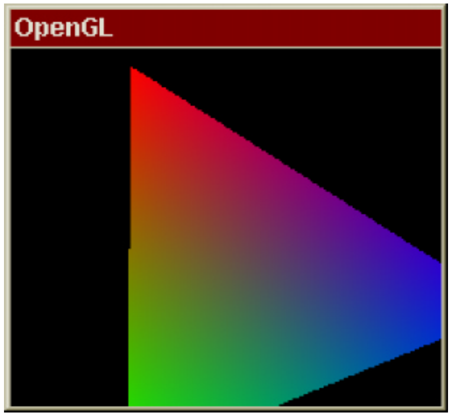
How do I create a rendering context?

- Just wiggle! (Use WGL)

```
HDC        hDC;    /* device context */
HGLRC      hRC;    /* opengl context */

hRC = wglCreateContext(hDC);
wglMakeCurrent(hDC, hRC);
```

- remember to clean up when done (see the course notes for details)

Screenshot of simple.c program



OpenGL & Win32 Topics

- A simple example to get started
- Processing messages & using menus
- Pixel formats & palettes
- Overlays & underlays
- WGL reference

Processing messages & using menus

Topics

- About messages
- Peeking at messages
- Using window procedures
- Using menus

About messages

What is a message?

- Method of communicating user input to an application
- MSG structure contains data pertinent to each message
- Analog of an X Window event

About messages (2)

How do I use messages?

- Two methods will be discussed (peeking & window procedure)
- Structure common to all messages
    - all message names begin with WM_ (can be used in a switch statement)
    - all messages have an lParam and a wParam (long word and word parameter)
    - values in lParam and wParam depend on the message

Peeking at messages

- Keep checking the message queue until a message appears

```
MSG msg;
while (1) {
    /* check for (and process) messages in the queue */
    while(PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE)) {
        switch(msg.message) {
        case WM_KEYDOWN:
            if(msg.wParam == 27) /* ESC */
                /* do something */
            break;
        /* case for other messages */
        default:
            DefWindowProc(hWnd, msg.message,
                                msg.wParam, msg.lParam);
            break;
        }
   }
}
```

- PeekMessage() can’t retrieve all message types

Using window procedures

What is a window procedure?

- Special function registered with the window class designed to handle messages
- Usually has a large switch() statement

```
LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    LONG lRet = 1;

    switch(uMsg) {
    case WM_CREATE:
        break;
    /* other message cases */
    default:
        lRet = DefWindowProc(hWnd, uMsg, wParam, lParam);
        break;
    }

    return lRet;
}
```

Using window procedures (2)

How do I use a window procedure?

- Translate and dispatch messages

```
while(PeekMessage(&msg, hWnd, 0, 0, PM_NOREMOVE)) {
   if(GetMessage(&msg, hWnd, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   } else {
      /* bail out - window was destroyed */
   }
}
```

- If you want to yield until a message appears on the queue, use GetMessage() in place of PeekMessage() and translate/dispatch within the while loop

Using menus

Why use menus?

- Built in to Win32
- Simple to use
- Professional looking :-)

Using menus (2)

How do I create a menu bar?

- Use CreateMenu() then insert items & attach it to the window

```
HMENU        hFileMenu;             /* file menu handle */
HMENU        hMenu;                 /* menu bar */
MENUITEMINFO item;                  /* item info */

hFileMenu = CreateMenu();
hMenu     = CreateMenu();
item.cbSize      = sizeof(MENUITEMINFO);
item.fMask       = MIIM_ID | MIIM_TYPE | MIIM_SUBMENU;
item.fType       = MFT_STRING;
item.hSubMenu    = NULL;
item.wID        = ’x’;
item.dwTypeData = "E&xit";
item.cch        = strlen("E&xit");
InsertMenuItem(hFileMenu, 0, FALSE, &item);

item.wID        = 0;
item.dwTypeData = "&File";
item.cch        = strlen("&File");
item.hSubMenu   = hFileMenu;
InsertMenuItem(hMenu, 0, FALSE, &item);

SetMenu(hWnd, hMenu);
```

Using menus (3)

How do I get messages from a menu?

- All menu items send a WM_COMMAND message to the window when selected

```
case WM_COMMAND:
    switch(LOWORD(wParam)) {
    case ’x’:
        PostQuitMessage(0);
        break;
    }
    break;
```

Using menus (4)

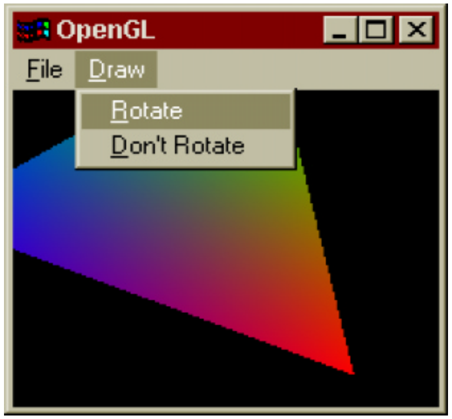
How do I handle a popup menu?

- Create the menu as outlined earlier
- When a mouse button is pressed, call TrackPopupMenu()

```
case WM_RBUTTONDOWN:
        point.x = LOWORD(lParam);
        point.y = HIWORD(lParam);
        ClientToScreen(hWnd, &point);
        TrackPopupMenu(hPopup, TPM_LEFTALIGN, point.x, point.y,
                       0, hWnd, NULL);
        break;
```

  - position of the popup menu must be in screen coordinates
  - use ClientToScreen() function to convert

Screenshot of menu.c program



OpenGL & Win32 Topics

- A simple example to get started
- Processing messages & using menus
- Pixel formats & palettes
- Overlays & underlays
- WGL reference

Pixel formats & palettes

Topics

- The Pixel Format Descriptor
- Using Palettes

The pixel format descriptor

What is a pixel format descriptor?

- A structure whose fields indicate properties of an OpenGL context
- Gateway to choosing a pixel format suitable for a given application
- Similar to an XVisualInfo structure in X Windows but tailored to OpenGL

The pixel format descriptor (2)

How do I use it?

- The simplest way is to fill in the fields of the structure with desired properties & call ChoosePixelFormat() (see simple.c)
- We can do better than ChoosePixelFormat()
- Enumerate all formats and compare against our own criteria
- See the course notes for a detail on each field of the pixel format descriptor

---

The pixel format descriptor (3)

- Use the DescribePixelFormat() function to enumerate all the pixel formats

```
int pf, maxpf;
PIXELFORMATDESCRIPTOR pfd;

/* get the maximum number of pixel formats */
maxpf = DescribePixelFormat(hDC, 0, 0, NULL);

/* loop through all the pixel formats */
for (pf = 1; pf <= maxpf; pf++) {
    DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
    if (pfd.dwFlags & PFD_DRAW_TO_WINDOW &&
        pfd.dwFlags & PFD_SUPPORT_OPENGL &&
        pfd.dwFlags & PFD_DOUBLEBUFFER   &&
        pfd.cDepthBits >= 24)
      {
         /* found a matching pixel format */
      }
 }
```

Using Palettes

Two basic reasons to use a palette

- When exact control over colors is needed or for palette animation (color index mode)
- When Truecolor display can’t be used (must simulate Truecolor with ramp & dithering)

Using color index mode

- Must use a logical palette (user defined table of colors)
- Select and Realize the palette for Win32 to recognize it (next slide)
- Intercept the proper messages

```
case WM_QUERYNEWPALETTE:
    SelectPalette(GetDC(hWnd), hPalette, FALSE);
    lRet = RealizePalette(GetDC(hWnd));
    break;

case WM_PALETTECHANGED:
    if(hWnd == (HWND)wParam) break;
    SelectPalette(GetDC(hWnd), hPalette, FALSE);
    RealizePalette(GetDC(hWnd));
    UpdateColors(GetDC(hWnd));
    lRet = 0;
    break;
```

Using color index mode (2)

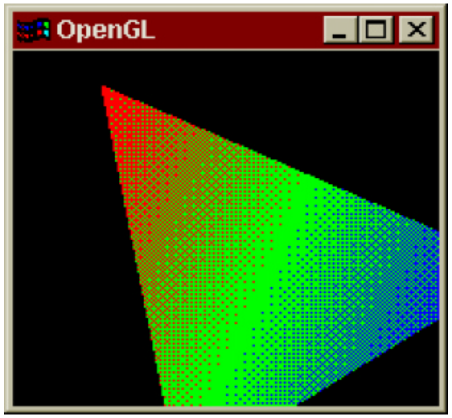
- Create, select and realize a logical palette

```
LOGPALETTE   lgpal;                      /* custom logical palette */
int          nEntries = 5;               /* number of entries in palette */
PALETTEENTRY peEntries[5] = {            /* entries in custom palette */
    0,   0,   0, NULL,                   /* black */
  255,   0,   0, NULL,                   /* red */
    0, 255,   0, NULL,                   /* green */
    0,   0, 255, NULL,                   /* blue */
  255, 255, 255, NULL                    /* white */
};

/* create a logical palette (for color index mode) */
lgpal.palVersion    = 0x300;             /* version should be 0x300 */
lgpal.palNumEntries = nEntries;          /* number of entries in palette */
hPalette = CreatePalette(&lgpal);

SetPaletteEntries(hPalette, 0, nEntries, peEntries);
SelectPalette(hDC, hPalette, TRUE); /* map logical into physical palette */
RealizePalette(hDC);
```

Simulating Truecolor with a palette

- A bit tricky deciding what the palette should look like
- Must have an adequate range of colors
- Functions exist to generate such palettes
- See the course notes for a detailed example

Screenshot of index.c program



OpenGL & Win32 Topics

- **A simple example to get started**
- **Processing messages & using menus**
- **Pixel formats & palettes**
- **Overlays & underlays**
- **WGL reference**

Overlays & Underlays

- **Overlay/Underlay associated with a particular pixel format**
- **Cannot be free floating over any window (as in X Windows)**
- **Same basic process as before, but now must use special WGL functions designed for overlays when setting pixel formats, creating contexts and swapping buffers**

Overlays & Underlays (2)

How do I use overlay/underlays?

- **Same basic process as before, but now must use special WGL functions designed for overlays when setting pixel formats, creating contexts and swapping buffers**

```
int pf, maxpf;
PIXELFORMATDESCRIPTOR pfd;
LAYERPLANEDESCRIPTOR  lpd;          /* layer plane descriptor */

maxpf = DescribePixelFormat(hDC, 0, 0, NULL);
for(pf = 0; pf < maxpf; pf++) {
    DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
    if (pfd.bReserved > 0) {
      /* aha! This format has overlays/underlays */
      wglDescribeLayerPlane(hDC, pf, 1,
                            sizeof(LAYERPLANEDESCRIPTOR), &lpd);
      if (lpd.dwFlags & LPD_SUPPORT_OPENGL &&
          lpd.dwFlags & LPD_DOUBLEBUFFER) /* any other flags */
        {
          /* found one! */
        }
```

Overlays & Underlays (3)

How do I use overlay/underlays?

- **Must ALWAYS set the palette for an overlay/underlay**

```
/* set the pixel format */
if(SetPixelFormat(hDC, pf, &pfd) == FALSE) {
    MessageBox(NULL,
               "SetPixelFormat() failed:  Cannot set format specified.",
               "Error", MB_OK);
    return 0;
}

/* set up the layer palette */
wglSetLayerPaletteEntries(hDC, 1, 0, nEntries, crEntries);

/* realize the palette */
wglRealizeLayerPalette(hDC, 1, TRUE);

/* announce what we’ve got */
printf("Number of overlays = %d\n", pfd.bReserved);
printf("Color bits in the overlay = %d\n", lpd.cColorBits);
```

Screenshot of overlay.c program

OpenGL & Win32 Topics

- A simple example to get started
- Processing Messages & Using Menus
- Pixel Formats & Palettes
- Overlays & Underlays
- WGL Reference

WGL Reference

- Rendering Context Functions
- Font and Text Functions
- Overlay, Underlay and Main Plane Functions
- Miscellaneous Functions

Rendering Context Functions

wglCreateContext
Creates a new rendering context.

wglMakeCurrent
Sets a thread's current rendering context.

wglGetCurrentContext
Obtains a handle to a thread's current rendering context.

wglGetCurrentDC
Obtains a handle to the device context associated with a thread's current rendering context.

wglDeleteContext
Deletes a rendering context.

Font and Text Functions

wglUseFontBitmaps

Creates a set of character bitmap display lists. Characters come from a specified device context's current font. Characters are specified as a consecutive run within the font's glyph set.

wglUseFontOutlines

Creates a set of display lists, based on the glyphs of the currently selected outline font of a device context, for use with the current rendering context. The display lists are used to draw 3−D characters of TrueType fonts.

Overlay, Underlay & Main Plane (1)

wglCopyContext

Copies selected groups of rendering states from one OpenGL rendering context to another.

wglCreateLayerContext

Creates a new OpenGL rendering context for drawing to a specified layer plane on a device context.

wglDescribeLayerPlane

Obtains information about the layer planes of a given pixel format.

wglGetLayerPaletteEntries

Retrieves the palette entries from a given color−index layer plane for a specified device context.

Overlay, Underlay & Main Plane (2)

wglRealizeLayerPalette

Maps palette entries from a given color−index layer plane into the physical palette or initializes the palette of an RGBA layer plane.

wglSetLayerPaletteEntries

Sets the palette entries in a given color−index layer plane for a specified device context.

wglSwapLayerBuffers

Swaps the front and back buffers in the overlay, underlay, and main planes of the window referenced by a specified device context.

Miscellaneous Functions

wglShareLists

Enables a rendering context to share the display−list space of another rendering context.

wglGetProcAddress

Returns the address of an OpenGL extension function for use with the current OpenGL rendering context.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

Comparison of OpenGL Window Sytem Interfaces

Contents

1. Introduction

Since OpenGL is designed to be independent of any window system, integration of OpenGL with a window system is accomplished with a special interface. This interface is dependent on the window system and is typically designed and implemented by the window system vendor.

Though each OpenGL window system interface is different they are all similar in functionality. This document compares the functionality of several interfaces. Programmers writing applications for more than one window systems should find this information especially relevant.

The following interfaces are compared:

- **AGL** for the Apple Macintosh
- **GLX** for the X Window System
- **PGL** for OS/2’s Presentation Manager
- **WGL** for Microsoft Windows ’95 and NT

2. Basic Functionality

There are five basic steps to OpenGL and window system integration in an application:

1. **Test for OpenGL capability** - be sure that the system supports OpenGL rendering.
2. **Select a visual/pixel type** - based on criteria such as RGB vs color index, single vs double buffering, depth buffering, stenciling, etc select a visual/pixel type.
3. **Create an OpenGL rendering context** - create a rendering context for the visual/pixel type selected.
4. **Create a drawable** - create a window or color buffer using the window system’s API. One of the parameters to the window creation function will probably be the visual/pixel type.
5. **Bind the rendering context to the drawable** - binding a context to a drawable activates the context and directs rendering to that drawable.

Note that the rendering context and drawable must usually use the same visual/pixel type. In other words, if you need two rendering windows which don’t share the same visual/pixel type you’ll need to create a separate context for each window.

3. Data types and objects

There are several data types or handles which are used for similar purposes in all the OpenGL interfaces.

Display/Session handle

The notion of a display or drawing/device context.

Datatypes

- AGL: none
- GLX: `Display`
- PGL: `HAB`
- WGL: `HDC`

Visual/Pixel format

The way in which pixel data in a frame buffer is displayed is controlled by a visual or pixel format. OpenGL typically augments a window system’s visuals/pixel formats with information about double buffering, depth buffers, stencil buffers, etc.

Datatypes

- AGL: `AGLPixelFmtID`
- GLX: `XVisualInfo`
- PGL: `PVISUALCONFIG`
- WGL: `an integer pixel format number or a PIXELFORMATDESCRIPTOR structure`

OpenGL rendering context

OpenGL is designed as a state machine. OpenGL state is encapsulated in a context. Multiple contexts may be created but only one may be active at a time. If an application needs to render into several windows, one context may be used for both windows if the windows use the same visual or pixel format. If different pixel formats are used then different OpenGL contexts may be required.

Datatypes

- AGL: `AGLContext`
- GLX: `GLXContext`
- PGL: `HGC`
- WGL: `HGLRC`

Window/drawable

The destination of OpenGL rendering is typically a window on your terminal screen. The OpenGL interface may also allow rendering into an off-screen color buffer. The handle for an off-screen buffer is typically compatible with a window handle.

An OpenGL rendering context is activated by binding a context to a window or drawable.

Datatypes

- AGL: `AGLDrawable`
- GLX: `GLXDrawable (a Window or GLXPixmap)`
- PGL: `HWND`
- WGL: `HDC`

4. Interface Functions

This section presents the major function of the interfaces catagorized according to their purpose.

4.1 Testing for OpenGL availability

At runtime it may be necessary to determine if a display or terminal is capable of OpenGL rendering.

GLX

```
Bool glXQueryExtension( Display *dpy, int *errorBase, int *eventBase )
```

PGL

```
LONG pglQueryCapability( HAB hab )
```

4.2 Getting OpenGL version information

Since OpenGL is an evolving standard it's sometimes useful to be able to determine which version of OpenGL render is being used.

AGL
```
    GLboolean aglQueryVersion( int *major, int *minor )
```

GLX
```
    Bool glXQueryVersion( Display *dpy, int *major, int *minor )
```

PGL
```
    void pglQueryVersion( HAB hab, int *major, int *minor )
```

4.3 Selection of a visual or pixel format

A visual or pixel format describes the frame buffer and ancillary buffers. Attributes include RGB vs color index, bits per color component, single vs double buffered, size of depth buffer, size of stencil buffer, etc.

The application programmer should know what frame buffer attributes are needed and select a visual or pixel format accordingly.

These functions return a visual or pixel format based on a attribute list provided by the programmer.

AGL
```
    AGLPixelFmtID aglChoosePixelFmt( GDHandle *dev, int ndev, int *attribs )
```

GLX
```
    XVisualInfo* glXChooseVisual( Display *dpy, int screen, int *attribList )
```

PGL
```
    PVISUALCONFIG pglChooseConfig( HAB hab, int *attriblist )
```

WGL
```
    int ChoosePixelFormat( HDC hdc, PIXELFORMATDESCRIPTOR *pfd )
```

4.4 Query visual/pixel format attributes

As an alternative to asking the window system for a visual/pixel format which matches an attribute list, one may query the attributes of a particular visual or pixel format. This allows the programmer complete control over visual/pixel format selection. These functions return the value of an attribute for a given visual/pixel format.

AGL
```
    GLboolean aglGetConfig( AGLPixelFmtID pix, int attrib, int *value )
```

GLX
```
    int glXGetConfig( Display *dpy, XVisualInfo *vis, int attrib, int *value )
```

PGL

```
PVISUALCONFIG *pglQueryConfigs( HAB hab )
```

WGL

```
int DescribePixelFormat( HDC hdc, int pixelformat, UINT bytes,
LPPIXELFORMATDESCRIPTOR pfd )
```

4.5 Creating a rendering context

After a visual/pixel format has been selected an OpenGL rendering context may be allocated. Rendering contexts may share display lists and texture maps if the contexts are compatible. Contexts are considered to be compatible if they share the same address space and pixel format and are both direct or indirect.

Direct contexts provide a means of utilizing local graphics hardware in the most efficient means possible. Indirect contexts are used in other situations such as when rendering remotely.

In the case of GLX, a direct context may be used when using local graphics hardware; the GLX protocol encoding/decoding is bypassed. An indirect context allows remote display to X servers which support the GLX extension.

Some OpenGL interfaces make no distinction between direct and indirect rendering.

AGL

```
AGLContext aglCreateContext( AGLPixelFmtID pix, AGLContext shareList )
```

GLX

```
GLXContext glXCreateContext( Display *dpy, XVisualInfo *vis, GLXContext
shareList, Bool direct )
```

PGL

```
HGC pglCreateContext( HAB hab, PVISUALCONFIG pVisualConfig, HGC ShareList, BOOL
IsDirect )
```

WGL

```
HGLRC wglCreateContext( HDC hdc )

BOOL wglShareLists( HGLRC hglrc1, HGLRC hglrc2 )
```

4.6 Destroying a rendering context

When finished with a context it may be destroyed.

AGL

```
GLboolean aglDestroyContext( AGLContext ctx )
```

GLX

```
void glXDestroyContext( Display *dpy, GLXContext ctx )
```

PGL

```
BOOL pglDestroyContext( HAB hab, HGC hgc )
```

WGL

```
wglDeleteContext( HRC hrc )
```

4.7 Context binding

When a rendering context is bound to a window it becomes the *current context*. OpenGL rendering may then begin. Note that it is not until this point that one may test for OpenGL extensions.

AGL

```
GLboolean aglMakeCurrent( AGLDrawable drawable, AGLContext ctx )
```

GLX

```
Bool glXMakeCurrent( Display *dpy, GLXDrawable drawable, GLXContext ctx )
```

PGL

```
BOOL pglMakeCurrent( HAB hab, HGC hgc, HWND hwnd )
```

WGL

```
wglMakeCurrent( HDC hdc, HGLRC hrc )
```

4.8 Copying context state

These functions copy a subset of a context state from one context to another. The mask parameter takes the same values as glPushAttrib().

AGL

```
GLboolean aglCopyContext( AGLContext src, AGLContext dst, GLuint mask )
```

GLX

```
void glXCopyContext( Display *dpy, GLXContext src, GLXContext dst, GLuint mask
)
```

PGL

```
BOOL pglCopyContext( HAB hab, HGC hgc_src, HGC hgc_dst, GLuint attrib_mask )
```

WGL

```
BOOL wglCopyContext( HGLRC hglrcSrc, hglrcDst, UINT mask )
```

4.9 Testing for direct rendering

These functions test if a rendering context is direct.

GLX

```
Bool glXIsDirect( Display *dpy, GLXContext ctx )
```

PGL

```
LONG pglIsIndirect( HAB hab, HGC hgc )
```

4.10 Swapping color buffers

The *swap buffers* operation exchanges the front and back color buffers when double buffering is enabled. The contents of the back buffer become undefined after the swap operation.

AGL
```
GLboolean aglSwapBuffers( AGLDrawable drawable )
```

GLX
```
void glXSwapBuffers( Display *dpy, GLXDrawable drawable )
```

PGL
```
void pglSwapBuffers( HAB hab, HWND hwnd )
```

WGL
```
BOOL SwapBuffers( HDC hdc )
```

4.11 Off-screen rendering

These functions create an off-screen color buffer or pixmap. Be aware that rendering to an off-screen color buffer may not be accelerated by your graphics hardware.

AGL
```
AGLPixmap aglCreateAGLPixmap( AGLPixelFmtID pix, GWorldPtr pixmap )

GLboolean aglDestroyAGLPixmap( AGLPixmap pix )
```

GLX
```
GLXPixmap glXCreateGLXPixmap( Display *dpy, XVisualInfo *vis, Pixmap pixmap )

void glXDestroyGLXPixmap( Display *dpy, GLXPixmap pix );
```

4.12 Bitmap fonts

Fonts provided by the window system may be converted to `glBitmap()` format and stored in display lists. Character strings may then be rendered with `glCallLists()`. These functions convert font glyphys from the window system to a sequence of display lists.

AGL
```
GLboolean aglUseFont( int familyID, int size, int first, int count, int
listBase )
```

GLX
```
void glXUseXFont( Font font, int first, int count, int listBase )
```

PGL
```
BOOL pglUseFont( HAB hab, HPS hps, FATTRS fatAttrs, LONG llcid, int first, int
count, int listbase )
```

WGL
```
BOOL wglUseFontBitmaps( HDC hdc, DWORD first, DWORD count, DWORD listBase )

BOOL wglUseFontOutlines( HDC hdc, DWORD first, DWORD count, DWORD listBase,
FLOAT deviation, FLOAT extrusion, int format, LPGLYPHMETRICSFLOAT lpgmf )
```

4.13 Querying the current context and drawable

The ID of the current rendering context and current window/drawable may be queried with these functions.

AGL

```
AGLContext aglGetCurrentContext( void )

AGLDrawable aglGetCurrentDrawable( void )
```

GLX

```
GLXContext glXGetCurrentContext( void )

GLXDrawable glXGetCurrentDrawable( void )
```

PGL

```
HGC pglGetCurrentContext( HAB hab )

HWND pglGetCurrentWindow( HAB hab )
```

WGL

```
HGLRC wglGetCurrentContext( void )

HDC wglGetCurrentDC( void )

int GetPixelFormat( HDC hdc )
```

4.14 Synchronization

Since both OpenGL and the native window system renderer may both draw into the same window synchronization is needed to be sure operations are performed in the correct order.

GLX

```
void glXWaitGL( void )

void glXWaitX( void )
```

PGL

```
HPS pglWaitGL( HAB hab )

void pglWaitPM( HAB hab )
```

4.15 Miscellaneous

Each OpenGL window system interface has some unique functions. Some of them are described here.

AGL

```
GLenum aglGetError( void )
```

Returns the current error setting or GL_OK if none.

```
int aglListPixelFmts( GDHandle dev, AGLPixelFmtID **fmts )
```

Returns a list of all pixel formats offered for the given device.

`GLboolean aglSetOptions( int options )`

Sets AGL-specific options.

`GLboolean aglUpdateCurrent( void )`

Causes the current context’s state to be updated from the window system. This should be called whenever the window is moved, resized, or the screen resolution or depth is changed.

GLX: (version 1.1)
const char *glXQueryExtensionsString(Display *dpy, int screen)

Returns a list of space separated GLX extensions on the specified display.

`const char *glXGetClientString( Display *dpy, int name )`

Returns a string describing an attribute of the OpenGL client library.

`const char *glXQueryServerString( Display *dpy, int screen, int name )`

Returns a string describing an attribute of the OpenGL display server.

PGL
`INT pglSelectColorIndexPalette( HAB hab, HPAL hpal, HGC hgc )`

This function specifies the color index palette for OpenGL to use when drawing in RGB mode.

`BOOL pglGrabFrontBitmap( HAB hab, HPS phps, HBITMAP phbitmap )`

`BOOL pglReleaseFrontBitmap( HAB hab )`

These functions are used to gain exclusive access to a window.

WGL
`wglCreateLayerContext`, `wglDescribeLayerPlane`, `wglGetLayerPaletteEntries`, `wglSetLayerPaletteEntries`, and `wglSwapLayerBuffers`

Provide support for overlay and underlay color buffers.

5. To learn more

Introduction to OpenGL and X, Part 1: An Introduction (http://www.sgi.com/Technology/openGL/mjk.intro/intro.html) by Mark Kilgard of SGI describes how to get started with OpenGL and the X Window System.

The Unix man pages for GLX and the GLX specification documents describe the GLX functions in detail.

agl.txt describes the AGL interface. This information provided courtesy of Template Graphics Software.

OpenGL for OS/2 including documentation can be obtained from ftp://ftp.austin.ibm.com/pub/developer/os2/OpenGL/.

Using OpenGL in Visual C++ Version 4.x (http://www.iftech.com/oltc/opengl/opengl0.stm) by N. Alan Oursland of Interface Technologies, Inc. describes how to get started using OpenGL with Microsoft’s Visual C++.

OpenGL I: Quick Start (http://www.microsoft.com/msdn/library/technote/gl1.htm) by Dale Rogerson of Microsoft is the first in a series of articles explaining how to use OpenGL with Windows 95 and Windows Nt.

Microsoft’s Developer Studio / Visual C++ product includes online documentation of the WGL interface.

Last edited on April 13, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Application Design and Organization Notes

Contents

1. Introduction

This document presents information which may help you in designing your OpenGL application and organizing its source code such that it may be portable to different window systems or graphics libraries.

Why would we want to do this? One, we may want our application to work on both X and Windows platforms. Two, we may want to support both OpenGL and IRIS GL (or PEX) during a transition period.

2. Organization

The basics:

- isolate window system-dependent code (including WGL and GLX code) in separate modules.
- isolate OpenGL code, and other graphics library code, in separate modules

The practicality of this depends on the nature and size of the application. One one hand, modern window system toolkits are quite similar in that GUIs are designed with the callback/event loop paradigm:

- Create user interface

- Setup callback functions
- Enter event loop

Furthermore, rendering can be encapsulated in wrapper functions which present a higher-level API which is independent of the graphics library.

On the other hand, a complex application may be so tightly integrated with a user interface toolkit or graphics library that it’s impractical to support alternative interfaces or libraries.

3. An example: Vis5D

Vis5D is a system for interactive visualization of three dimensional atmospheric data. It can use OpenGL, IRIS GL, or PEX for 3-D rendering. An Xlib-based GUI toolkit provides the only user interface at this time but it’s quite feasible to write a new one.

OpenGL, IRIS GL and PEX code is isolated into separate source files:

- graphics.ogl.c
- graphics.gl.c
- graphics.pex.c

Each file performs the rendering functions defined by a single header file, graphics.h, defining functions such as:

- create_3d_window()
- clear_3d_window()
- swap_3d_window()
- draw_isosurface()
- draw_trajectory()
- draw_contour_slice()

which graphics.ogl.c, graphics.gl.c and grahics.pex.c each implements in its own way. The Makefile determines which source file is compiled.

The core of Vis5Ds functionality is isolated from the user interface by an internal API. Everything "below" the API is GUI independent. Everything "above" the API is considered user interface code. While Vis5D’s user interface code is substantial, it could be replaced by an alternative toolkit with minimal impact on the rest of the system.

4. Graphics library functionality

When supporting multiple graphics libraries, a difficult problem to deal with is subsetting. While OpenGL mandates that all its features be implemented other graphics libraries aren’t as stringently defined. PEX implementations, for example, vary greatly in terms of what features are implemented.

The simplest solution to this problem is to only use functionality which is common to all libraries. This can actually be quite practical in simple applications which don't require elaborate renderering techniques.

The other solution is to poll the graphics system to determine its capabilities and work around those it doesn't support. Vis5D, for example, offers volume rendering only on systems with alpha blending capability.

5. Multi-window system applications

Suppose your OpenGL application must work on several window system such as X and Microsoft Windows. How can this be accomplished?

5.1 Cross-platform GUIs

Consider using a cross-platform GUI such as GLUT or Tcl/Tk which is available for several window systems. GLUT is appropriate for demos or small applications. Tcl/Tk is appropriate for any size demo or application. Both are free.

5.2 Commercial porting tools

There are commercial solutions which provide Motif emulation for Windows:

- NuTCRACKER from DataFocus, Inc. (http://www.datafocus.com/)
- OpenNT from Softway Systems, Inc. (http://www.softway.com/OpenNT/)
- Exceed from Hummingbird Communications, Ltd. (http://www.hummingbird.com/)

Commercial solutions for porting Windows applications to Unix/X/Motif include:

- Wind/U from Bristol Technology, Inc. (http://www.bristol.com/)
- MainWin Studio from Mainsoft Corporation (http://www.mainsoft.com/)

5.3 Native support for multiple GUIs

Larger applications which use native window system toolkits will have to be partitioned into modules which isolate window and operating system-specific code.

If one is going to use multiple window systems (for example X/Motif and Win32) it's best to first survey the GUIs to determine what they have in common or what is unique to each. It may be wise then to avoid using GUI features which can't be implemented in all window systems.

The OpenGL window system interface (WGL, GLX) calls should be considered window system code and not be put in the OpenGL modules. This includes the swapbuffers operation.

Last edited on April 13, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

Using OpenGL Extensions

Contents

1. Introduction

The designers of OpenGL anticipated the need to extend OpenGL in the future. Thus they clearly defined how extensions are to be implemented and used. To be sure your application is portable it is very important that one uses extensions correctly.

There are three tenets to using extensions:

1. Compile-time extension testing
2. Run-time extension testing
3. Fall-back scenarios

These are discussed below. Furthermore, one may also have to deal with different versions of OpenGL and the GLU and GLX libraries.

We begin with a discussing of extension naming conventions.

2. Naming conventions

OpenGL extension are named according to the convention:

GL_*type_name*

Where *type* is `EXT` or a vendor-specific identifier such as `SGI` or `IBM`.

The `EXT` indentifier generally indicates that an extension has been adopted by at least two vendors.

Vendors may also extend the *type* convention to indicate the class of the extension. Silicon Graphics, for example, use `SGIS` to indicate an extension may only be available on particular systems and `SGIX` to indicate that the extension is experimental.

name is a string of lowercase characters such as `polygon_offset`.

Example extension names:

- `GL_EXT_polygon_offset`
- `GL_SGI_color_table`
- `GL_SGIS_detail_texture`
- `GL_MESA_window_pos`

If an extension defines any new `GLenum` values they will be suffixed with the extension type. For example, the `GL_EXT_blend_minmax` extension adds the following `GLenum` values:

- `GL_FUNC_ADD_`**`EXT`**
- `GL_MIN_`**`EXT`**
- `GL_MAX_`**`EXT`**
- `GL_BLEND_EQUATION_`**`EXT`**

If an extension defines any new API functions they will be suffixed with the extension type as well. For example, the `GL_EXT_polygon_offset` extension adds the function:

```
void glPolygonOffsetEXT( GLfloat factor, GLfloat bias )
```

3. Compile-time extension testing

If an OpenGL extension is supported at compile-time the host's *gl.h* file will define a preprocessor symbol named for that extension. For example, the *gl.h* file will have

```
#define GL_EXT_texture3D 1
```

if the GL_EXT_texture3D extension is supported.

Any references to constants or functions defined by the extension must be surrounded by `#ifdef/#endif`. For example:

```
#ifdef GL_EXT_texture3D
   glTexImage3DEXT(GL_TEXTURE_3D_EXT, 0, format, w, h, d, border,
                   format, type, pixels);
#endif
```

Failure to test for extensions at compile time can result in compilation and linking errors such as `Undefined symbol` or `Undefined function`.

It is critical to properly test for extensions at compile time if you want your application to be recompilable on different systems.

4. Run-time extension testing

We must also test for OpenGL extensions at runtime. There are two reasons for this:

1. An OpenGL application may be dynamically linked to the OpenGL library. When the application is moved to another system with a different OpenGL library there's no guarantee that this library will implement the same extensions as the first library.
2. OpenGL on the X Window System supports remote display and there's no guarantee that any X server's OpenGL renderer will support a given extension.

To test for OpenGL extensions at runtime we must call `glGetString(GL_EXTENSIONS)`. This function returns a list of extensions which are supported by the OpenGL renderer. This list can be searched to determine if a specific extension is supported.

Be aware that `glGetString(GL_EXTENSIONS)` must be called *after* we've established an active OpenGL rendering context. For example, we must call `glXMakeCurrent` or `wglMakeCurrent` before calling `glGetString`. The reason is that OpenGL extensions are dependant on the OpenGL renderer and the renderer isn't bound until MakeCurrent is called.

Be careful *when searching the extensions list!* The C library function `strstr` is not sufficient because it may match a substring of the extension name you're testing for. For example, if you're testing for the `GL_EXT_texture` extension and `glGetString(GL_EXTENSIONS)` returns `"GL_EXT_texture3D"` then simply using `strstr` will incorrectly tell you that `GL_EXT_texture` is supported.

The following function can be used for reliable runtime extension testing:

```
GLboolean CheckExtension( char *extName )
{
   /*
    ** Search for extName in the extensions string.  Use of strstr()
    ** is not sufficient because extension names can be prefixes of
    ** other extension names.  Could use strtok() but the constant
    ** string returned by glGetString can be in read-only memory.
    */
   char *p = (char *) glGetString(GL_EXTENSIONS);
```

```
        char *end;
        int extNameLen;

        extNameLen = strlen(extName);
        end = p + strlen(p);

        while (p < end) {
            int n = strcspn(p, " ");
            if ((extNameLen == n) && (strncmp(extName, p, n) == 0)) {
                return GL_TRUE;
            }
            p += (n + 1);
        }
        return GL_FALSE;
    }
```

5. An extension sampler

This section lists some OpenGL extensions with short descriptions. Many extensions are implemented in groups. For example, the blending extensions are interdependent and usually implemented together. See your OS/OpenGL release notes and man pages for detailed descriptions.

Core extensions

Many of these extensions to OpenGL 1.0 have been incorporated into OpenGL 1.1.

- GL_EXT_abgr - adds the GL_ABGR_EXT pixel format to glDrawPixels, glReadPixels, and glTexImage[12]D. A performance improvement over GL_RGBA on systems designed for IRIS GL.
- GL_EXT_blend_color - adds blending operations with constant colors
- GL_EXT_blend_logic_op - extends `glLogicOp` functionality to RGB blending
- GL_EXT_blend_minmax - adds min/max operators to RGB blending
- GL_EXT_blend_equation - adds subtractive blending equations
- GL_EXT_convolution - adds 1 and 2 dimensional image convolution
- GL_EXT_copy_texture - allows one to load texture images directly from the frame buffer
- GL_EXT_histogram - counts occurances of specific color components during rasterization
- GL_EXT_packed_pixels - adds packed pixel formats for glDrawPixels, glReadPixels, glTexImage, etc.
- GL_EXT_polygon_offset - adds the `glPolygonOffsetEXT` function which displaces the Z value of polygon fragments to facilitate drawing cleanly outlined polygons
- GL_EXT_subtexture - allows subregions of texture images to be replaced
- GL_EXT_texture - adds many packed texture format data types and the texture proxy mechanism
- GL_EXT_texture3D - three dimensional texture image support, useful for volume rendering
- GL_EXT_texture_object - named texture objects; improves performance when multiple textures are needed.
- GL_EXT_vertex_array - specifies geometric primitives with arrays of coordinate data as an alternative to using many glVertex, glColor, glNormal, or glTexCoord calls.

SGI-specific core extensions

- GL_SGI_color_matrix - adds another 4x4 transformation matrix which effects RGBA colors
- GL_SGI_color_table - extends the color lookup table functionality of OpenGL
- GL_SGIX_interlace - causes glDrawPixels and glTexImage to skip rows of pixels (for working with video data (fields vs frames))
- GL_SGIS_sharpen_texture - adds a texture magnification filter which uses extrapolation to improve sharpness of magnified textures
- GL_SGIS_texture_border_clamp - adds a new texture coordinate clamping function which doesn't average the border and edge colors when interpolating samples
- GL_SGIS_texture_color_table - adds a color lookup table to texturing
- GL_SGIS_texture_edge_clamp - adds a new texture coordinate clamping function which prohibits sampling of the texture border color
- GL_SGIS_texture_filter4 - adds support for user-defined 4x4 texture sampling functions

GLX Extensions (see section 8)

- GLX_EXT_import_context - allows multiple X clients to share an indirect rendering context
- GLX_EXT_visual_info - extends RGB mode rendering to PseudoColor, StaticColor, GrayScale, and StaticGray visuals. Also, adds support for transparent overlay pixels.
- GLX_EXT_visual_rating - classifies GLX visuals according to performance and visual quality

SGI-specific GLX extensions

- GLX_SGI_make_current_read - independently set pixel draw and read drawables so, for example, `glCopyPixels` can copy from one window into another
- GLX_SGIS_multisample - an antialiasing mechanism for high-end hardware
- GLX_SGI_swap_control - adds a function to control the rate of glXSwapBuffers and a function for synchronized swapping of multiple displays
- GLX_SGIX_video_source - allows sourcing of pixel data from a video stream
- GLX_SGI_video_sync - provides a way to synchronize with the video frame rate

Microsoft OpenGL Extensions

- GL_WIN_swap_hint - specify a sub-window to swap, rather than the whole window. This is a performance improvement. For more information see http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/ogl/gl/src/glfunc01_1.htm

6. OpenGL 1.1

Many extensions designed for OpenGL 1.0 have been incorporated into OpenGL 1.1 as standard features.

A program written for OpenGL 1.0 which uses no extensions will work with OpenGL 1.1 unchanged. However, a program written for OpenGL 1.0 with extensions may require some modifications to work with OpenGL 1.1.

If you want your program to compile and execute cleanly with either OpenGL 1.0 or OpenGL 1.1 you will need to observe the following guidelines.

Compile time

To detect whether a particular feature is available at compile time you will need to use the C preprocessor to test for either an OpenGL 1.0 extension name or test for the OpenGL 1.1 version symbol: **GL_VERSION_1_1**.

For example:

```
#if defined(GL_EXT_texture_object) || defined(GL_VERSION_1_1)
   your code
#endif
```

Sometime in the future you may need

```
#if defined(GL_EXT_texture_object) || defined(GL_VERSION_1_1) || defined(GL_VERSION_1_2)
   your code
#endif
```

Runtime

At runtime you must check if the renderer supports OpenGL 1.1 or the 1.0 extension:

```
/* After calling MakeCurrent()! */
char *version = (char*) glGetString(GL_VERSION);
GLboolean HaveTexObjExtension;

if (strncmp(version,"1.1",3)==0
    || CheckExtension("GL_EXT_texture_object")) {
    HaveTexObjExtension = GL_TRUE;
}
else {
    HaveTexObjExtension = GL_FALSE;
}
```

Example

Implementing these checks correctly can be a bit complicated. Here’s an approach you may find useful:

Step 1

Declare a boolean variable for each extension or OpenGL 1.1 feature you would like to use:

```
GLboolean HaveTextureObjects = GL_FALSE;
GLboolean HavePolygonOffset  = GL_FALSE;
```

Step 2

Write a function which tests for each feature at runtime. Call it after your first call to `MakeCurrent`.

```
void check_gl_features( void )
{
    char *version = (char*) glGetString(GL_VERSION);
    char *exten = (char*) glGetString(GL_EXTENSIONS);

    if (strncmp(version,"1.1",3)==0) {
        HaveTextureObjects = GL_TRUE;
        HavePolygonOffset  = GL_TRUE;
    }
    else {
        HaveTextureObjects = CheckExtension("GL_EXT_texture_object");
        HavePolygonOffset  = CheckExtension("GL_EXT_polygon_offset");
    }
}
```

Step 3

Write wrapper functions to hide some of the ugliness of dealing with OpenGL 1.1 or 1.0 extensions. For example:

```
/* call to allocate a set of texture objects */
void myGenTextures( GLsizei n, GLuint *textures )
{
    if (HaveTextureObjects) {
#if defined(GL_VERSION_1_1)
        glGenTextures( n, textures );
#elif defined(GL_EXT_texture_object)
        glGenTexturesEXT( n, textures );
#endif
    }
    else {
        /* fallback code: use display lists */
        GLuint first;
        first = glGenLists( n );
        if (first>0) {
            GLuint i;
            for (i=0; i < n; i++) {
                textures[i] = first+i;
            }
        }
    }
}


/* call to start defining a texture object */
void myBeginTexture( GLenum target, GLuint texture )
{
    if (HaveTextureObjects) {
#if defined(GL_VERSION_1_1)
        glBindTexture( target, texture );
#elif defined(GL_EXT_texture_object)
        glBindTextureEXT( texture );
#endif
    }
    else {
        /* fallback code: use display lists */
        glNewList( texture, GL_COMPILE );
```

```
    }
}


/* call to finish defining a texture object */
void myEndTexture( GLenum target )
{
    if (HaveTextureObjects) {
#if defined(GL_VERSION_1_1)
        glBindTexture( target, 0 );
#elif defined(GL_EXT_texture_object)
        glBindTextureEXT( texture, 0 );
#endif
    }
    else {
        /* fallback code: use display lists */
        glEndList();
    }
}


/* call to use a texture object */
void myBindTexture( GLenum target, GLuint texture )
{
    if (HaveTextureObjects) {
#if defined(GL_VERSION_1_1)
        glBindTexture( target, texture );
#elif defined(GL_EXT_texture_object)
        glBindTextureEXT( target, texture );
#endif
    }
    else {
        /* fallback code: use display lists */
        glCallList( texture );
    }
}


/* turn polygon offset on/off */
void myPolygonOffset( GLboolean onoff )
{
    if (HavePolygonOffset) {
#if defined(GL_VERSION_1_1)
        if (onoff) {
            glPolygonOffset( 1.0f, 1.0f );  /* tune this */
            glEnable( GL_POLYGON_OFFSET_FILL );
        }
        else {
            glDisable( GL_POLYGON_OFFSET_FILL );
        }
#elif defined(GL_EXT_texture_object)
        if (onoff) {
            glPolygonOffsetEXT( 1.0f, 0.0001f );  /* tune this */
            glEnable( GL_POLYGON_OFFSET_EXT );
        }
        else {
            glDisable( GL_POLYGON_OFFSET_EXT );
        }
#endif
    }
    else {
```

```
        /* fallback code: no offset */
    }
}
```

When designing wrapper functions it’s usually best to look at the big picture and design simple, high-level wrappers rather than try to make wrappers which directly corresponds to individual OpenGL functions.

A collection of wrappers like these may be put in a separate source file and reused in many applications.

7. GLU extensions and versions

There have been several versions of the GLU (GL Utility) library and the library may have extensions. Again, for safety, the GLU version and extensions should be tested for at compile-time and run-time if you need their specific features. At this time, there are no known GLU extensions.

Compile-time testing

If a GLU extension is available at runtime the *glu.h* file will define a preprocessor symbol with the prefix `GLU_EXT_`. As with OpenGL extensions, there should be `#ifdef/#endif` tests surrounding any references to functions or symbols unique to the extension.

Run-time testing

GLU version 1.0 had no function to call at run-time to query the GLU version or extensions list. GLU version 1.1 added the `gluGetString` function which takes two possible values: `GLU_EXTENSIONS` or `GLU_VERSION`.

Therefore, if you want to get a list of GLU extensions you’ll need to use something like this:

```
char *extensions;
#ifdef GLU_VERSION_1_1
extensions = (char *) gluGetString(GLU_EXTENSIONS);
#else
extensions = "";
#endif
```

Be careful of accidently matching substrings while searching the string.

GLU versions

There have been several versions of the GLU library. As shown above, you can test for the GLU version at compile-time by checking for preprocessor symbols like `GLU_VERSION_1_1` and `GLU_VERSION_1_2`. At run-time you can determine the GLU version by calling `gluGetString(GLU_VERSION)`.

Version 1.1 of GLU only added the `gluGetString` function.

Version 1.2 of GLU introduced a new polygon tessellator. The new tessellator functions all begin with the prefix `gluTess`. For more information about the changes in the GLU tesselator from version 1.0 to 1.1 see http://www.digital.com:80/pub/doc/opengl/opengl_new_glu.html

Note that if the `GLU_VERSION_1_2` symbol is defined then the `GLU_VERSION_1_1` symbol is also defined. One can expect this trend of backward compatibility to continue.

8. GLX extensions

The GLX interface offers extensions in a manner very similar to core OpenGL. Again, extensions must be tested for both at compile-time and run-time. If a GLX extension is not available there should be a fall-back strategy.

Compile-time testing

If a GLX extension is available at runtime the *glx.h* file will define a corresponding preprocessor symbol. For example, if the `GLX_EXT_import_context` extension is available, then *glx.h* (or *glxtokens.h*) will contain

```
#define GLX_EXT_import_context  1
```

Run-time testing

After we've established a connection to an X server we can determine which GLX extensions are available by calling `glXQueryExtensionsString(dpy, screen)`. This function returns a list of supported GLX extensions separated by white space. Again, we have to be careful when searching the extensions list. A function similar to `CheckExtension` should be used.

GLX version testing

There have been several versions of the GLX interface. Version 1.0 was the first version. Version 1.1 added the `glXQueryExtensionsString, glXQueryServerString` and `glXGetClientString` functions. Version 1.2 may include several of the 1.0 and 1.1 GLX extension features.

Testing for the GLX version at runtime involves checking for a preprocessor symbol such as `GLX_VERSION_1_1` or `GLX_VERSION_1_2`.

The GLX version can be determined at runtime by calling `glXQueryVersion`.

9. Fall-back scenarios

Your program should be prepared for the likely situation in which a desired extension is not available.

Depending on the nature of the extension you may elect to limit functionality, fall-back to an equivalent but slower implementation, or to simply abort.

Examples:

- If the GL_SGIS_multisample extension is not present then antialiasing may simply be disabled.
- If the `GL_EXT_vertex_array` extension is not available then you should fall-back to the regular `glVertex/glColor/glNormal` functions at the expense of performance.
- If your application is a 3-D volume rendering program based on the 3-D texture map extension you may have no choice but to abort if the `GL_EXT_texture3D` extension is not available.

Aborting when an extension isn't available is stronly discouraged. In most cases users will prefer reduced performance/functionality over complete failure. At the very least, the user should be informed why an OpenGL application can't operate if an extension isn't present.

10. Using Extensions with Microsoft OpenGL or SGI Cosmo OpenGL

Unfortunately, there is a complication in using OpenGL extensions with Microsoft OpenGL or SGI Cosmo OpenGL.

Instead of simply calling extension functions directly one must use `wglGetProcAddress` to get a pointer to extension functions.

For example, instead of this:

```
#if defined(GL_WIN_swap_hint)
   if (CheckExtension("GL_WIN_swap_hint")) {
      glAddSwapHintRectWIN(x, y, width, height);
   }
#endif
```

One must use:

```
#if defined(WIN32) && defined(GL_WIN_swap_hint)
   if (CheckExtension("GL_WIN_swap_hint")) {
      PFNGLADDSWAPHINTRECTWINPROC glAddSwapHintRectWIN;
      glAddSwapHintRectWIN = (PFNGLADDSWAPHINTRECTWINPROC)
                   wglGetProcAddress("glAddSwapHintRectWIN");
      (*glAddSwapHintRectWIN)(x, y, width, height);
   }
#endif
```

By the way, the `glAddSwapHintRectWIN` function must be called before every `SwapBuffers` call. The rectangle list is lost after `SwapBuffers`.

11. References

Other sources of information about OpenGL extensions can be found at:

- All about OpenGL Extensions from SGI.
  (http://www.sgi.com/Technology/openGL/extensions.html)
- *Programming OpenGL with the X Window System* by Mark Kilgard.
- wglGetProcAddress documentation from
  (http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/ogl/winext/src/ntopnglr_14.htm)
- Using Cosmo OpenGL Extensions
  (http://www.sgi.com/Products/cosmo/opengl/beta2/OpenGLonWin-17.html)

Last edited on April 19, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

GLX Portability Notes

Contents

1. Introduction

GLX is the OpenGL interface to the X Window System. GLX defines both an API and a wire protocol which allows remote display of OpenGL applications on GLX-capable X servers.

Many OpenGL portability problems can be traced to GLX programming errors. The purpose of this document is to help the GLX programmer avoid a number of common problems.

Information relevant to using Mesa is also included. Even if an OpenGL developer isn’t targetting Mesa it’s a good idea to be aware of Mesa’s idiosyncrasies since it will expand the range of systems on which the application can be used.

2. GLX fundamentals

After we’ve established a connection to an X server (perhaps with `XOpenDisplay`) we have to check that the X server actually supports OpenGL and the GLX X server extension.

The `glXQueryExtension(dpy, errorBase, eventBase)` function serves this purpose. The returned `errorBase` and `eventBase` values are usually ignored. If `glXQueryExtension` returns false then the application should inform the user that the display does not support OpenGL.

Next, we’ll proceed with GLX setup which includes selecting a GLX visual, creating a GLX context, selecting a colormap and creating a window.

3. GLX visuals

A GLX visual is basically an X visual augmented with ancillary (depth, stencil, accumulation, etc) buffer information.

A visual is usually chosen with `glXChooseVisual`. Per the OpenGL GLX specification, if an RGB mode is requested, `glXChooseVisual` will return either a `TrueColor` or `DirectColor` visual. Otherwise, a `PseudoColor` or `StaticColor` visual will be returned for color index mode.

Mesa, however, may potentially return any X visual type for RGB mode. This is because some X displays on which Mesa may be used do not have `TrueColor` or `DirectColor` visuals. Mese prefers visual types in the order `TrueColor`, `DirectColor`, `PseudoColor`, `StaticColor`, `GrayScale`, and `StaticGray` and visuals depths from deepest to shallowest. There is one exception: 8-bit `PseudoColor` is preferred over 8-bit `TrueColor`. This is a convention many people prefer for low-end displays which use an 8-bit `PseudoColor` visual for the default and only have one hardware colormap.

Similarly, Mesa may return a `PseudoColor`, `StaticColor`, `GrayScale` or `StaticGray` visual if color index mode is requested.

Mesa violates the GLX specification but allows rendering on more types of displays than OpenGL would.

Dealing with Mesa’s expanded offering of visuals is mostly just a matter of handling colormaps correctly.

A footnote-

I’ve lost count of how many people have reported that depth buffering doesn’t work on system XYZ or doesn’t work with Mesa. In all cases the problem has been that the programmer neglected to specify/request a depth-buffered visual. Many OpenGL servers have depth buffers associated with all GLX visuals so even if a depth buffer isn’t requested one may *get lucky* and get a depth-buffered visual anyway.

The point is: be careful that the attribute list passed to `glXChooseVisual` really specifies what you need.

4. Colormaps

The best way to handle X colormaps depends on whether one is rendering in RGB or color index mode.

4.1 RGB mode colormaps

When rendering in RGB mode the colormap is usually never altered (using a `DirectColor` visual may be an exception). In general we want to share read-only colormaps among windows to minimize colormap flashing. Colormap flashing (aka the technicolor effect) occurs when the demand for colormaps exceeds the hardware’s capacity. As the mouse is moved from window to window different colormaps may be installed; some windows will be forced to use the wrong colormap.

The following algorithm should pick a good RGB colormap in most cases:

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>  /* for XA_RGB_DEFAULT_MAP atom */
#if defined(__vms)
#include <X11/StdCmap.h>  /* for XmuLookupStandardColormap */
#else
#include <X11/Xmu/StdCmap.h>  /* for XmuLookupStandardColormap */
#endif
#include <GL/glx.h>
#include <string.h>

/*
 * Return an X colormap to use for OpenGL RGB-mode rendering.
 * Input:  dpy - the X display
 *         scrnum - the X screen number
 *         visinfo - the XVisualInfo as returned by glXChooseVisual()
 * Return:  an X Colormap or 0 if there’s a _serious_ error.
 */
Colormap
get_rgb_colormap( Display *dpy, int scrnum, XVisualInfo *visinfo )
{
   Atom hp_cr_maps;
   Status status;
   int numCmaps;
   int i;
   XStandardColormap *standardCmaps;
   Window root = RootWindow(dpy,scrnum);
   int using_mesa;

   /*
    * First check if visinfo’s visual matches the default/root visual.
    */
   if (visinfo->visual==DefaultVisual(dpy,scrnum)) {
      /* use the default/root colormap */
      return DefaultColormap( dpy, scrnum );
   }

   /*
    * Check if we’re using Mesa.
    */
   if (strstr(glXQueryServerString( dpy, scrnum, GLX_VERSION ), "Mesa")) {
      using_mesa = 1;
   }
   else {
      using_mesa = 0;
   }

   /*
```

```
    * Next, if we’re using Mesa and displaying on an HP with the "Color
    * Recovery" feature and the visual is 8-bit TrueColor, search for a
    * special colormap initialized for dithering.  Mesa will know how to
    * dither using this colormap.
    */
   if (using_mesa) {
      hp_cr_maps = XInternAtom( dpy, "_HP_RGB_SMOOTH_MAP_LIST", True );
      if (hp_cr_maps
          && visinfo->visual->class==TrueColor
          && visinfo->depth==8) {
         status = XGetRGBColormaps( dpy, root, &standardCmaps,
                                    &numCmaps, hp_cr_maps );
         if (status) {
            for (i=0; i < numCmaps; i++) {
               if (standardCmaps[i].visualid==visinfo->visual->visualid) {
                  Colormap cmap = standardCmaps[i].colormap;
                  XFree(standardCmaps);
                  return cmap;
               }
            }
            XFree(standardCmaps);
         }
      }
   }

   /*
    * Next, try to find a standard X colormap.
    */
#ifndef SOLARIS_BUG
   status = XmuLookupStandardColormap( dpy, visinfo->screen,
                                       visinfo->visualid,
                                       visinfo->depth,
                                       XA_RGB_DEFAULT_MAP,
                                       /* replace */ False,
                                       /* retain */ True);
   if (status == 1) {
      status = XGetRGBColormaps( dpy, root, &standardCmaps,
                                 &numCmaps, XA_RGB_DEFAULT_MAP);
      if (status == 1) {
         for (i = 0; i < numCmaps; i++) {
            if (standardCmaps[i].visualid == visinfo->visualid) {
               Colormap cmap = standardCmaps[i].colormap;
               XFree(standardCmaps);
               return cmap;
            }
         }
         XFree(standardCmaps);
      }
   }
#endif

   /*
    * If we get here, give up and just allocate a new colormap.
    */
   return XCreateColormap( dpy, root, visinfo->visual, AllocNone );
}
```

Basically, we use the default/root colormap if the visual matches the default/root visual. Otherwise we look for a standard colormap. If that fails we must allocate a new, private colormap. If using Mesa on an

8-bit `TrueColor` HP display then we look for a special "Color Recovery" colormap which helps to produce high-quality dithered images.

Caveat: this algorithm may not work on Sun systems due to a bug in the `XmuLookupStandardColormap` function. By defining the `SOLARIS_BUG` symbol the code in question can be omitted.

Finally, if one intends to render into several different windows with the same RGB context those window should share the same colormap. This is required with Mesa and helps to reduce colormap flashing with OpenGL.

4.2 Color index mode colormaps

When designing a color index mode application we must decide if we need a writable colormap and/or need specific colors associated with specific pixel values. For lighting and fog effects to work in color index mode one has to store specific colors in consecutive colormap entries. Therefore, a private, writable colormap is required. It should be allocated/created with `XCreateColormap( dpy, win, visual, AllocAll )`.

Otherwise, if your GLX visual type and depth matches the default/root visual then you can probably use the default/root colormap. To allocate a read/write colorcell from the colormap use `XAllocColorCells`. To allocate read-only cells use `XAllocColor`. In both cases, X will return to you the index of a colorcell.

If `XAllocColor` fails then you may have to search the colormap for a close match. The following function will search a colormap for the closest match to your requested color:

```
#include <X11/Xlib.h>
#include <stdlib.h>

/* A replacement for XAllocColor.
 * This  function should never fail to allocate a color.  When
 * XAllocColor fails, we return the nearest matching color.  If
 * we have to allocate many colors this function isn’t a great
 * solution; the XQueryColors() could be done just once.
 */
static void
noFaultXAllocColor(Display * dpy, Colormap cmap, int cmapSize, XColor * color)
{
    XColor *ctable, subColor;
    int i, bestmatch;
    double mindist;       /* 3*2^16^2 exceeds long int precision. */

    /* First try just using XAllocColor. */
    if (XAllocColor(dpy, cmap, color))
        return;

    /* Retrieve color table entries. */
    /* XXX alloca canidate. */
    ctable = (XColor *) malloc(cmapSize * sizeof(XColor));
    for (i = 0; i < cmapSize; i++)
      ctable[i].pixel = i;
    XQueryColors(dpy, cmap, ctable, cmapSize);

    /* Find best match. */
    bestmatch = -1;
```

```
    mindist = 0.0;
    for (i = 0; i < cmapSize; i++) {
        double dr = (double) color->red - (double) ctable[i].red;
        double dg = (double) color->green - (double) ctable[i].green;
        double db = (double) color->blue - (double) ctable[i].blue;
        double dist = dr * dr + dg * dg + db * db;
        if (bestmatch < 0 || dist < mindist) {
          bestmatch = i;
          mindist = dist;
        }
    }

    /* Return result. */
    subColor.red = ctable[bestmatch].red;
    subColor.green = ctable[bestmatch].green;
    subColor.blue = ctable[bestmatch].blue;
    free(ctable);
    if (!XAllocColor(dpy, cmap, &subColor)) {
      subColor.pixel = (unsigned long) bestmatch;
   }
   *color = subColor;
}
```

If your application needs several color index mode windows it’s a good idea to try to share one colormap among the windows. Finally, be sure that `glXChooseVisual` returns a `PseudoColor` (or for Mesa, `GrayScale`) visual if a writable colormap is needed.

After the colormap has been selected you can create your window, specifying the colormap in the `XSetWindowAttributes` structure passed to `XCreateWindow`.

Furthermore, you should inform the window manager if your top-level window contains children with non-default colormaps. This is done with the `XSetWMColormapWindows` function:

```
    XSetWMColormapWindows( display, top_level_window,
                           &window_list, num );
```

5. Double buffering

Surprisingly, double buffered visuals are not required by OpenGL. If a `glXChooseVisual` request for a double buffered visual fails you should try to get a single buffered visual. Be sure to call `glFlush` to force completion of rendering where `glXSwapBuffers` would have been called.

Similarly, OpenGL does not require single buffered visuals to be offered. If you want a single buffered window but `glXChooseVisual` fails, you should try again specifying double buffering. Then, issue `glDrawBuffer(GL_FRONT)` to direct drawing to the front color buffer.

Be aware that many systems advertised as having 24-bit color, in fact, only offer 12-bit color in double buffer mode. This is because the 24-bit frame buffer is divided into two 12-bit buffers. Dithering usually makes up for the loss of color accuracy.

Suppose you want both double buffering and full 24-bit color in this situation. For example, during animation one may want double buffering but to show a static image a full-color single buffered window would look best.

IRIS GL allowed one to reconfigure a window to single or double buffering on the fly with `doublebuffer`, `singlebuffer` and `gconfig`. This can’t be done with OpenGL. Instead, you can create two subwindows contained by a common parent, one window single buffered and the other window double buffered, and use `XMapWidnow/XUnmapWindow` to display the one you want to use. Remember to use separate contexts for each window since they will have different visuals.

6. GLX Pixmaps

GLX pixmaps are used for off-screen OpenGL rendering. A GLX pixmap is basically an X Pixmap augmented with OpenGL ancillary buffers (depth, stencil, etc). The advantages of GLX pixmaps are they take no screen space, are never damaged, and not constrained by the size of the screen. The disadvantage of GLX pixmaps is that 3-D graphics hardware is often unable to render into them; a software renderer executes the OpenGL instructions.

The usual steps in creating and using a GLX pixmap are:

- Select a visual with `glXChooseVisual`
- Create an X pixmap with `XCreatePixmap` using the depth of the visual returned by `glXChooseVisual`
- Create the GLX pixmap from the X pixmap with `glXCreateGLXPixmap`.
- Create an OpenGL rendering context with `glXCreateContext`, usually specifying the indirect option.
- Bind the context to the GLX pixmap with `glXMakeCurrent`

Notes:

- Since one often wants to render into a GLX pixmap and later copy it to an on-screen window, the X window should have the same depth as the pixmap.
- If one wants to use one context for both GLX pixmap rendering and rendering into a window, the GLX pixmap and window must be created with the same XVisualInfo.
- Direct rendering contexts are usually not supported for pixmap rendering. The only way to determine if direct rendering into GLX pixmaps works is to create a direct context then test if `glXMakeCurrent` succeeds.

There is a special problem in using GLX pixmaps with Mesa in RGB mode. Since Mesa supports RGB mode rendering into any kind of X visual it often needs colormap information so that RGB values can be converted into logical pixel values. The GLX pixmap facility does not provide a way to indicate which X colormap is associated with a GLX pixmap.

Mesa (version 1.2.8 and later) has a GLX extension which lets the user specify the colormap associated with a GLX pixmap. The extension provides a new function very similar to `glXCreateGLXPixmap`:

```
GLXPixmap glXCreateGLXPixmapMESA( Display *dpy, XVisualInfo *visual,
```

```
                                          Pixmap pixmap, Colormap cmap )
```

Strictly speaking, the colormap argument is only needed when rendering in RGB mode into a GLX pixmap which uses a `PseudoColor`, `StaticColor`, `GrayScale` or `StaticGray` visual. If the colormap is not specified but is in fact needed, the `glXMakeCurrent` call will return False.

The proper way to use this function is:

```
    Pixmap p;
    GLXPixmap q;
    ...
    #ifdef GLX_MESA_pixmap_colormap
        q = glXCreateGLXPixmapMESA( display, visual, p, colormap );
    #else
        q = glXCreateGLXPixmap( display, visual, p );
    #endif
```

Since the GLX_MESA_pixmap_color extension symbol is only defined if using Mesa's header files this technique will be portable to any GLX implementation.

7. Mesa-specific

Since Mesa doesn't really implement the GLX protocol it isn't 100% compliant with the GLX specification. Most of the significant differences have been explained above. The remaining differences are discussed here.

7.1 GLX_MESA_release_buffers extension

The first time an X window is specified to Mesa's `glXMakeCurrent` the X window is augmented with ancillary (back color, depth, stencil, etc) buffers. Unfortunately, Mesa's GLX has no way of detecting when the X window is destroyed with `XDestroyWindow`. The best Mesa can do is to check for recently destroyed windows whenever the client calls the `glXCreateContext` or `glXDestroyContext` functions. This may not be sufficient in all situations though. If many windows are used by the application a great deal of memory may be wasted.

The solution is to call the `glXReleaseBuffersMESA` function just before destroying the X window. For example:

```
#ifdef GLX_MESA_release_buffers
    glXReleaseBuffersMESA( dpy, window );
#endif
XDestroyWindow( dpy, window );
```

Last edited on April 13, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL "Gotchas"

Even though OpenGL is a well organized and has a simple API there some common pitfalls which new (and experienced) programmers can run into.

This document describes many such pitfalls and offers explanations or work-arounds.

glDrawPixels problems.

glDrawPixels draws a skewed image

Be sure the GL_UNPACK_ALIGNMENT value is set correctly. The default is four and if you’re drawing GLubyte GL_RGB images it may have to be set to one.

glDrawPixels() draws the wrong colors

Be sure texture mapping is disabled as texturing is applied even to glDrawPixels. Also, be sure you’re using the correct data type for your imagery. A common mistake is to use GLuint instead of GLubyte when drawing images with single-byte red, green, blue and alpha components.

glDrawPixels() of imagery obtained from glReadPixels() looks different than the original image

Try disabling dithering with glDisable(GL_DITHER).

glDrawPixels isn’t as fast as expected

Some older graphics systems handle ABGR-order pixels faster than RGBA-order. Try the GL_EXT_abgr extension. Also, be sure to disable rasterization options such as depth testing, fog, stenciling, scissoring, pixel scaling, dithering and biasing, if you don’t need them. GL_UNSIGNED_BYTE is typically the fastest data type.

How can I make glDrawPixels() draw an image flipped upside down?

Try glPixelZoom(1.0, -1.0). Similarly, an image can be flipped left to right with glPixelZoom(). Note that you may have to adjust your raster position to position the image correctly.

glRasterPos Problems

glRasterPos() doesn’t put the raster position at the window coordinate I specify

glRasterPos transforms coordinates by the modelview and projection matrices just like vertices. Set your matrices appropriately.

Why can't I position a bitmap outside of the window?

If glRasterPos() evaluates to a position outside of the viewport the raster position becomes invalid. Subsequent `glBitmap()` and `glDrawPixels()` calls will have no effect.

Solution; extend the viewport beyond the window bounds or use glBitmap() with an NULL bitmap and your desired delta X,Y movement from the current, valid raster position. Be sure to restore the viewport to a normal position before rendering other primitives.

The following function will set the raster position to an arbitrary window coordinate:

```
void window_pos( GLfloat x, GLfloat y, GLfloat z, GLfloat w )
{
   GLfloat fx, fy;

   /* Push current matrix mode and viewport attributes */
   glPushAttrib( GL_TRANSFORM_BIT | GL_VIEWPORT_BIT );

   /* Setup projection parameters */
   glMatrixMode( GL_PROJECTION );
   glPushMatrix();
   glLoadIdentity();
   glMatrixMode( GL_MODELVIEW );
   glPushMatrix();
   glLoadIdentity();

   glDepthRange( z, z );
   glViewport( (int) x - 1, (int) y - 1, 2, 2 );

   /* set the raster (window) position */
   fx = x - (int) x;
   fy = y - (int) y;
   glRasterPos4f( fx, fy, 0.0, w );

   /* restore matrices, viewport and matrix mode */
   glPopMatrix();
   glMatrixMode( GL_PROJECTION );
   glPopMatrix();

   glPopAttrib();
}
```

The sequence of glRasterPos(), glColor(), glBitmap() doesn't result in the desired bitmap color

Call glColor() before glRasterPos().

Texture Mapping Problems

Texturing just isn't working

There are several possible explanations.

- If texture minification is happening and the GL_MIN_FILTER is not GL_NEAREST or GL_LINEAR then you must have a complete set of mipmaps defined. If you don't it is as if texturing were disabled.

- Be sure your texture sizes are powers of two. Some OpenGL implementations fail to generate an error for this condition.

Textures with borders don't work

Several implementations of OpenGL have bugs which prevent textures with borders from working correctly. OpenGL on SGI Infinite Reality systems is an example.

Texturing isn't working on a Reality Engine 2 system

There's a known bug which requires glEnable(GL_TEXTURE_2D) be called before glTexImage2D() in some situations.

Performance Problems

Overall slow performance

Be sure a direct rendering context is being selected so that graphics hardware is accessed directly.

Motif/OpenGL Problems

Problems with glViewport and window resizing with Motif

In the resize callback for your application you should put a call to `glXWaitX` before the `glViewport` call to be sure the X server has actually resized the window before `glViewport` is called.

Lighting and Coloring Problems

glColor3b(255, 255, 255) doesn't give me white

Be careful with color values and data types. The correct function in this case is glColor3ub(255, 255, 255).

When lighting is enabled, the colors are not what's expected

Try glEnable(GL_NORMALIZE) to scale your normal vectors to unit length. glScale() effects normal vectors, not just vertices.

Lines and points aren't colored as expected

Lighting may be enabled. All vertices are lit if lighting is enabled, even when drawing points and lines.

In color index mode `glClearIndexi(0)` doesn't clear the window to black.

There is no guarantee that color index 0 corresponds to black in the colormap. It is up to you to be sure the colormap entries are correctly loaded in your application.

Miscellaneous Problems

Nothing is drawn when in single-buffer mode

Call glFlush() after rendering. Your drawing commands may accumulate in a buffer and not be executed until you explicitly issue a flush.

How do I draw outlined polygons?

If you’ve tried this you’ve probably seen the "shimmer" effect caused by erroneous depth buffering of the polygon vs the outline. There are several solutions. The polygon offset extension, standard in OpenGL 1.1, is one. A slightly more complex solution is to use stenciling as described in the OpenGL Programming Guide.

Be sure no errors are being generated

Use glGetError() inside your rendering/event loop to catch errors. With Mesa, set the MESA_DEBUG environment variable.

Can I restrict SwapBuffers to a subregion of a window?

No. However, you may be able to use glCopyPixels to copy pixels from the back to front buffer or create subwindows for the regions you want swapped.

Depth testing isn’t working

If you’ve called `glEnable(GL_DEPTH_TEST)` and depth testing still isn’t happening be sure that you’ve requested a visual (GLX) or pixel format (WGL) which has a depth buffer. This is done by specifying the GLX_DEPTH_SIZE parameter to `glxChooseVisual()` or specifying a non-zero `cDepthBits` value in the `PIXELFORMATDESCRIPTOR` structure passed to `ChoosePixelFormat()`.

Last edited on April 20, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Hardcopy

Contents

- 1. Introduction
- 2. Bitmap-based Output
- 3. Vector-based Output
- 4. Microsoft Windows OpenGL Printing

1. Introduction

OpenGL was designed for realtime 3-D raster graphics, which is very different from 2-D printed copy. Nevertheless, many OpenGL applications need hardcopy output. There are basically two approaches:

1. raster/bitmap-based
2. vector-based

The following two sections describe the raster and vector approaches. Microsoft OpenGL users may elect to use the built-in printing support described in the last section.

2. Bitmap-based Output

A simple solution to OpenGL hardcopy is to simply save the window image to an image file, convert the file to Postscript, and print it. Unfortunately, this usually gives poor results. The problem is that a typical printer has much higher resolution than a CRT and therefore needs higher resolution input to produce an image of reasonable size and fidelity.

For example, a raster image of size 1200 by 1200 pixels would more than fill the typical 20-inch CRT but only result in a printed image of only 4 by 4 inches if printed at 300 dpi.

To print an 10 by 8-inch image at 300 dpi would require a raster image of 3000 by 2400 pixels. This is a situation in which off-screen, tiled rendering is useful. For more information see OpenGL/Mesa Offscreen Rendering and TR, a tile rendering utility library for OpenGL.

Once you have a raster image in memory it needs to be written to a file. If printing is the only intended purpose for the image than directly writing an Encapsulated Postscript file is best.

Mark Kilgard's book *Programming OpenGL for the X Window System* contains code for generating Encapsulated Postscript files. The source code may be downloaded from ftp://ftp.sgi.com/pub/opengl/opengl_for_x/xlib.tar.Z.

3. Vector-based Output

In general, high quality vector-style hardcopy is difficult to produce for arbitrary OpenGL renderings. The problem is OpenGL may generate arbitrarily complex raster images which have no equivalent vector representation. For example, how are smooth shading and texture mapping to be converted to vector form?

Getting the highest quality vector output is application dependant. That is, the application should probably generate vector output by examining its scene data structures.

If a more general solution is desired there are at least two utilities which may help:

GLP (http://dns.easysw.com/~mike/glp/) is a C++ class library which uses OpenGL's feedback mechanism to generate Postscript output. GLP is distributed with a GNU copyright.

GLPrint (http://www.ceintl.com/products/GLPrint/) from Computational Engineering International, Inc. is a utility library OpenGL printing. The product is currently in beta release.

4. Microsoft Windows OpenGL Printing

Microsoft's OpenGL support printing of OpenGL images via metafiles. The basic steps are:

1. Call `StartDoc` to associate a print job to your HDC handle
2. Call `StartPage` to setup the document
3. Create a rendering context with `wglCreateContext`
4. Bind the context with `wglMakeCurrent`
5. Do your OpenGL rendering
6. Unbind the context with `wglMakeCurrent(NULL, NULL)`
7. Call `EndPage` to finish the document
8. Call `EndDoc` to finish the print job

This procedure is raster-based and may require much memory. To circumvent this problem, printing is done in bands. This however takes more time.

Last edited on April 22, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Language Bindings

Contents

1. Introduction

The OpenGL API is defined in terms of C/C++ but bindings for several other languages exist.

Fortunately, the OpenGL function parameters are all simple types (boolean, integer, floating point, constants, arrays) so the API translates easily from C to other languages.

The OpenGL Architecture Review Board (ARB) controls the C, C++, Fortran, Pascal and Ada binding specifications at this time.

2. Bindings

C++

> Same as the C bindings. The ARB voted not to use the C++ function overloading facility. Therefore, the C++ OpenGL interface is identical to that for C.

Fortran

> Fortran bindings are shipped by several vendors including SGI. The Fortran API functions are prefixed with f. For example, glVertex3f() becomes fglVertex3f().
>
> The OpenGL constants are not supposed to be prefixed with F (i.e. GL_POLYGON, not FGL_POLYGON) but SGI’s IRIX 5.3 Fortran header file for OpenGL does use the F prefix. The GLUT toolkit includes an fgl.h header file with correctly named constants.

Finally, the maximum length of identifiers varies among Fortran compilers. Since OpenGL has several long (+32 character) identifiers they may be truncated in the header file.

Bill Mitchell of the NIST has written fortran 77 and fortran 90 bindings for OpenGL and Mesa. (http://math.nist.gov/f90gl/)

Ada

Discussed by the ARB, but yet to be implemented by a vendor.

Modula-3

OpenGL bindings for Modula-3 are available from Columbia University. (http://www.cs.columbia.edu:80/graphics/modula3/opengl/)

Pascal

No Pascal bindings for OpenGL are known to exist.

Tcl/Tk

- TIGER (Tcl-based Interpretative Graphics EnviRonment) is a tool for interpretive programming of OpenGL with Tcl. (ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/TIGER1.0)
- TkOGL provides Tcl/Tk wrappers for the OpenGL API (http://aquarius.lcg.ufrj.br/~esperanc/tkogl.html). The following program, for example, draws a triangle:

```
pack [OGLwin .gl]
.gl main -clear colorbuffer \
        -begin triangle \
        -vertex 0 1 0 \
        -vertex -1 -1 0 \
        -vertex 1 -1 0 \
        -end
```

- OGLTK is a Tk widget/shell for OpenGL rendering. (http://www.cs.unm.edu/~bederson/ogl.html)
- Togl is another Tk widget for OpenGL rendering based on OGLTK but with a few more features. (http://www.ssec.wisc.edu/~brianp/Togl.html)

Python

David Ascher at Brown University has information about Python and OpenGL. (http://maigret.cog.brown.edu:80/python/opengl/)

Java

At the time these notes were written the status of official OpenGL / 3D support for Java was still indeterminate. Unfortunatley, It appears that Sun and Silicon Graphics are *not* collaborating further on Cosmo3D.

In the mean time, one is probably best off with the unoffical port of OpenGL to Java by Leo Chan of the University of Waterloo. (ftp://cgl.uwaterloo.ca/pub/software/meta/OpenGL4java.html)

STk (Scheme/Tk)

Carnegie Mellon University has OpenGL bindings for StK, a Scheme interpreter with a Tk interface. Contact James Grandy (jcg@cs.cmu.edu) for more information.

Delphi

Delphi bindings for OpenGL 1.0 (written by Rick Hansen, 71043.2142@compuserve.com) and 1.1 (written by Mike Lischke, Lischke@imib.med.tu-dresden.de) are available from the Delphi Super Page (http://sunsite.icm.edu.pl/delphi/"). Search for *opengl*.

3. Notes

While OpenGL’s API is easily adapted to many languages the same can’t be said of most window system interfaces. For example, a Fortran-based OpenGL application may still need some C code to interface OpenGL with Xlib since there’s no Fortran interface to Xlib.

In some cases, such as Tcl/Tk, a special interface layer written in C may encapsulate the details of the OpenGL window system interface. Another example is GLUT. GLUT hides the details of OpenGL window system integration, providing a simple, window system-independent interface with both C and Fortran bindings.

Last edited on April 14, 1997 by Brian Paul.

The Mesa 3-D Graphics Library

A White Paper

Brian Paul

Second Edition, April 1997

Abstract

Mesa is a free 3-D graphics library which uses the OpenGL API and semantics. It works on most modern computers allowing people without OpenGL to write and use OpenGL-style applications. This paper gives an overview of Mesa and describes a bit of its implementation.

Contents

1. Introduction

Mesa began as an experiment in writing a 3-D graphics library. After about a year of "spare time" development it was released on the Internet. It has since evolved with the help of many contributors to the point where it is a viable and popular alternative to OpenGL.

In the spirit of free software, Mesa is distributed under the terms of the GNU library copyright.

The Mesa distribution includes implementations of the core OpenGL library functions, the GLU utility functions, the aux and tk toolkits, Xt/Motif widgets, drivers for X11, Microsoft Windows ’95/NT and DOS, NeXTStep, and many demonstration programs. Macintosh and Amiga drivers are available separately.

Mesa compiles easily, requiring only an ANSI C compiler and standard development headers and libraries.

From the application programmer’s point of view, Mesa is a nearly seemless replacement for OpenGL. The Mesa header files are named the same as OpenGL’s (GL/gl.h, GL/glu.h, GL/glx.h, etc) and contain equivalent datatypes, constants and function prototypes. The Mesa library files may be renamed to match the typical OpenGL library names and locations. On some operating systems Mesa may be built as a shared library.

After Mesa has been installed most OpenGL applications should compile and execute without modification.

Since version 2.0 of Mesa the OpenGL 1.1 API is implemented.

2. Mesa vs. OpenGL

While Mesa uses the OpenGL API and follows the OpenGL specification very closely, it is important to understand that Mesa is not a true implementation of OpenGL. Official OpenGL products are licensed and must completely implement the OpenGL specification and pass a suite of conformance tests. Mesa meets none of these requirements.

At first, Mesa may seem to be a competitor to official OpenGL products. Actually, Mesa has helped to promote the OpenGL API by expanding the range of computers which may execute OpenGL programs. There are many systems which are not supported by OpenGL vendors but can run Mesa instead. People who are curious about OpenGL may try Mesa at no cost and later purchase an OpenGL implementation which perhaps utilizes 3-D graphics hardware. Mesa has been very popular in computer graphics courses. Many students and colleges without the resources to obtain commercial OpenGL implementations successfully use Mesa instead.

Mesa does not implement the full OpenGL specification. For example, antialiasing, trimmed NURBS, and a few glGet* functions are not yet implemented. The GLX interface is only an emulation; it does not generate GLX protocol. It is expected that these features will eventually be implemented.

Mesa doesn’t typically perform as well as commercial OpenGL implementations for several reasons. First, portability to a wide range of computers is considered more important than optimizing for a particular architecture. Second, the features of the underlying hardware can’t be directly accessed since Mesa exists as a software library above the operating system and window system programming interfaces. And finally, Mesa’s development is not supported by any sort of development team. Only so much can be accomplished by people working in their spare time.

In other respects Mesa has some advantages over OpenGL.

- Mesa is free.
- Mesa works on many computers which lack real OpenGL implementations.
- There is a simple built-in profiling facility which can measure and report performance information.
- There is an option to enable immediate error message reporting. As soon as an error is generated it is printed to the stdout stream.
- Mesa can warn the user when attempting to do illogical things (such as enabling depth testing without a depth buffer).
- Users may attempt to optimize Mesa's source code in areas which impact the performance of their particular application.

3. Implementation

Mesa is written in ANSI C. The core library contains no operating system or window system dependent code which makes it extremely portable. A special device driver interface insulates the core Mesa library from the underlying operating/window system.

3.1 Library State

OpenGL is designed around the concept of a state machine. In Mesa this state is stored in a large C structure. Much of the state is stored in substructures which directly correspond to the attribute groups such as the polygon group, lighting group and texture group. Pushing and popping of attribute groups is just a matter of copying C structs to and from a stack.

Many API functions simply modify state values and produce no output. Before rendering functions are invoked it is often necessary to evaluate the current state to compute derived state values and setup pointers to specific instances of rendering functions. Lazy evaluation is used to updated the state.

For example, Mesa has many instances of specialized polygon drawing functions. The function to use depends on the state of smooth vs flat shading, dithering, depth testing, texturing, etc. When any of these state values are changed the *new state* flag is set. When `glBegin` is called the *new state* flag is tested and if set, the state is evaluated to select the specialized polygon function and the flag is cleared.

3.2 Point, Line and Polygon Rendering

Arguably the most important feature of Mesa is efficient point, line and polygon rendering. The two major components of this are vertex transformation and rasterization.

Vertices specified between `glBegin` and `glEnd` are accumulated in a vertex buffer. When the buffer is full or `glEnd` is called the buffer is processed. Processing the vertex buffer includes transforming vertices from object coordinates to eye coordinates, lighting, transforming eye coordinates to clip coordinates, clip testing, and mapping clip coordinates to window coordinates.

Each transformation and clip test stage is implemented in a tight loop which compilers can unroll for efficient executution. The size of the vertex buffer was chosen so that all vertex data touched in the transformation loops will fit in a 16KB CPU data cache.

Several optimization are used during transformation. The modelview and projection matrices often have particular elements with values of zero or one. These elements are tested to determine if simplified vector/matrix multiplications can be used. Depending on the current lighting parameters, either a full-featured or specialized, optimized lighting function is used. Lookup tables are used to compute the exponential spotlight and material shininess functions.

After a vertex buffer has been processed it is rendered as a set of points, lines or polygons as specified by `glBegin`.

Arrays of points are rendered by either calling a specialized device driver function or by falling back to a core Mesa drawing function. Points whose clip flag is set are discarded.

Line segments are clipped if either endpoint's clip flag is set. Then, the line is rasterized by calling either a specialized device driver function or a fallback Mesa line drawing function. Different line drawing functions are called for flat or smooth shading, RGB or color index mode, texturing, etc.

Polygons are clipped with the Sutherland-Hodgman algorithm if any of the vertex clip flags are set. Next, the equation of the plane containing the polygon is computed. The coefficients of the plane equation ax+by+cz=d are used for determining front/back orientation and implementing the polygon offset feature.

Polygons with more than three vertices are decomposed into triangles. Then, as with line segments, the triangle is rasterized either by a specialized device driver function or by a core fall-back function.

The specialized device driver functions for point, line and triangle rendering take vertices as input and directly modify the frame buffer. Alternatively, the fallback rendering functions in Mesa handle rendering of primitives with arbitrary raster operations. Point, line and bitmap functions generate fragments which are stored in a pixel buffer. The triangle rasterizers and `glDrawPixels` generate horizontal runs of pixels called spans. The pixel buffer and spans are subjected to fragment processing before being written to the frame buffer.

3.3 Fragment processing

Fragments are the pixels generated by rasterization augmented with auxiliary information such as color, depth (Z) and texture coordinates. OpenGL defines an extremely flexible fragment processing pipeline which includes texturing, fogging, clipping, scissoring, alpha testing, stenciling, depth testing, blending, dithering, bitwise logic operations, and masking.

Pixel buffer and span-based fragment processing are very similar, the only difference is that the pixel buffer stores fragments with arbitrary window coordinates while spans are continuous horizontal runs of fragments.

Since fragments may be culled during processing, each fragment has a write flag associated with it. Initially, all fragments have their write flags set to true. Clipping, scissoring, alpha testing, stenciling, and depth testing may set a flag to false to indicate that it should not be considered in further stages. In the end, only those fragments with their flags set are written to the color buffer.

Each stage of fragment processing is implemented in succession with code similar to:

```
if (stage is enabled) {
    for (each fragment in the buffer or span) {
        apply the fragment operation,
            possibly setting some write flags to false
    }
}
```

Finally, fragments are written to the color buffer by device driver functions similar to:

```
for (each fragment) {
    if (fragment flag is true) {
        write fragment color to color buffer
    }
}
```

The special cases of all write flags set to true or false are handled appropriately. Also, optimized code is used when all fragments have the same color.

The only fragment operation which must be handled below the device driver level is dithering. Depth testing, bitwise logic operators and masking may optionally be implemented by the device driver.

3.4 Device Driver Functions

A Mesa driver implements two things:

1. A public OpenGL/window system API (the GLX API, for example)
2. A set of priver driver functions (line and triangle drawing functions, for example)

The device driver interface is a set of function pointers which point to implementations specific to the window system. It includes functions for:

- setting the `glClear` color or index
- clearing the color buffer
- setting the current drawing color or index
- selecting the front or back color buffer as current source or destination
- returning the dimensions of the current color buffer
- drawing points, lines, triangles in specific situations
- implementing `glDrawPixels` for specific situations
- drawing horizontal runs of pixels
- reading horizontal runs of pixels
- drawing arrays of randomly positioned pixels
- reading arrays of randomly positioned pixels
- implementing `glFlush` and `glFinish`
- setting the index and color component write masks
- setting the pixel logic operator
- enabling/disabling dithering
- implementing depth buffer facilities

Some device driver functions are optional. If a particular function isn't implemented by the device driver then we fall back to an internal Mesa function.

The next section explains this in more detail for the X device driver.

3.5 The X Device Driver

The X device driver is the most mature of the Mesa device drivers so it is the example we elaborate upon.

3.5.1 GLX Emulation

Mesa's interface to the X Window System is defined by the X/Mesa interface. There are X/Mesa functions for creating rendering contexts, destroying contexts, binding contexts to windows and pixmaps, swapping color buffers and querying the current context. This interface is not intented for use by application programmers. It's purpose is to support Mesa's GLX emulation.

Mesa only emulates the GLX interface since a true implementation requires hooks into the X server. Mesa and its GLX can be though of as a translator which converts OpenGL API functions to Xlib commands. The nice side-effect of this is that Mesa can remotely render to any X server, even if the X server does not have the GLX server extension. Operating systems which support shared libraries can substitute Mesa for OpenGL at runtime, allowing OpenGL applications to be displayed on non-GLX capable X servers without recompiling.

Since it's an emulation, Mesa's GLX is not 100% compatible with OpenGL's GLX. In several ways is actually superior. For example, while OpenGL only supports RGB rendering into TrueColor or DirectColor X visuals, Mesa allows RGB rendering into virtually any type and depth of X visual. This is an important feature since many X servers don't offer TrueColor or DirectColor visuals. Other visuals are supported by dithering or converting RGB values to gray levels.

This introduces two potential incompatibilities with OpenGL's GLX.

- Rendering into GLX pixmaps requires information about the colormap which isn't normally associated with the pixmap.
- OpenGL applications expecting only TrueColor or DirectColor visuals may fail when Mesa returns a different visual type through the `glXChooseVisual` function.

The first problem is solved with a special Mesa extension to GLX. The second problem can usually be fixed by modifying the application's GLX code.

3.5.2 Pixmaps vs XImages

Images in X can be stored in one of two formats. Pixmaps are stored in the X server and cannot be directly addressed by an X client. XImages are stored in the client's address space and may be directly addressed.

When operating in single buffered mode, rendering is directed into an X window. When operating in double buffered mode, rendering is directed into either a Pixmap or XImage. A Pixmap can be accessed in the same way as a window (both are considered to be *drawables*). Whether a Pixmap or XImage gives best performance depends on a number of factors.

Using a Pixmap can be quite efficient for rendering plain, flat-shaded points, lines and polygons since the intrinsic X point, line and polygon drawing functions can be used. Performance is relatively good whether displaying locally or remotely. However, when using smooth shading or per-pixel fragment operations pixels must be drawn individually with `XSetForeground` and `XDrawPoint` calls. The amount of data transferred from the client to X server is directly proportional to the number of X calls made. For `XSetForeground/XDrawPoint` rendering this is usually unacceptably slow.

In most cases using an XImage yields best performance in double buffer mode. The reason is individual pixels can be directly "poked" into the image since it resides in the client's address space. Front/back buffer swapping is implemented by copying the XImage to the X window. The X Shared Memory extension is used when displaying on the local host to accelerate this operation. In the case of remote display, the amount of data transferred from the client to the X server is directly proportional to the window size and not the number of pixels generated during rendering.

Programmers should note that double buffering using an XImage can be faster than single buffering.

3.5.3 Pixel Processing

The most important factor in device driver performance is efficient access to the frame/image buffer for reading and writing fragments.

The code for writing RGB pixels to the color buffer could be expressed as:

```
for (each pixel i) {
    pixel_value = convert_rgb_to_pixel( red[i], green[i], blue[i] );
    put_pixel( x[i], y[i], pixel_value );
}
```

However, this would be very inefficient since the convert_rgb_to_pixel and put_pixel functions must cope with many types of X visuals and depths. The best method to convert RGB values to pixel values depends on the X visual. The best method to write pixels to the color buffer depends on whether the buffer is implemented as an X Pixmap or XImage. Therefore, almost all inner-loops in the X device driver are optimized for special pixel formats.

For example, there are specialized span and pixel-array writing functions for 24-bit TrueColor, 16-bit TrueColor, 8-bit PseudoColor, N-bit GrayScale, etc. Furthermore, there are many line and triangle rasterizer functions optimized for these pixels formats with popular combination of flat/smooth shading, depth-tested/non-depth-tested rasterization modes.

When the device driver's `UpdateState` state function is called the driver's pointers for span, line and triangle functions are updated to point to the appropriate optimized function. If no optimized function satisfies the current library state then a core Mesa fall-back function is used instead.

The device driver's point, line and triangle functions are also used for hardware acceleration. In this case the driver function will simply set hardware registers and trigger an interupt or DMA to make the hardware render the primitive.

4. Extensions

Mesa implements several popular OpenGL extensions and adds a few of its own.

4.1 OpenGL Extensions

Mesa has the following OpenGL extensions:

- GL_EXT_blend_color
- GL_EXT_blend_minmax
- GL_EXT_blend_logic_op
- GL_EXT_blend_subtract
- GL_EXT_polygon_offset
- GL_EXT_vertex_array
- GL_EXT_texture_object
- GL_EXT_texture3D

Several, such as texture objects and vertex arrays, are also standard OpenGL 1.1 (Mesa 2.x) features. Implementing them both as standard features and as extensions is simply a portability convenience to programmers.

4.2 Mesa Extensions

Like OpenGL, Mesa can have extensions. At this time, Mesa has four unique extensions.

GL_MESA_window_pos

This extension adds the `glWindowPos*MESA` functions. These functions are convenient alternatives to `glRasterPos*` because they set the current raster position to a specific window coordinate, bypassing the usual modelview, projection and viewport transformations. This is especially useful for setting the position for `glDrawPixels` or `glBitmap` to a desired window coordinate.

For `glWindowPosMESA4f(x,y,z,w)` the `x`, `y`, `z`, and `w` parameters directly set the current raster position except that `z` is clamped to the range [0,1]. The current raster position valid flag is always set to true. The current raster distance is set to zero. The current raster color and texture coordinate are updated in the same manner as for `glRasterPos`. In selection mode a hit record is always generated.

Programs using OpenGL, not Mesa, may also use the `glWindowPos*MESA` functions since an implementation of it in terms of standard OpenGL functions is included with Mesa.

Perhaps the GL_MESA_window_pos extension may be incorporated into a future version of OpenGL since it is so convenient.

GL_MESA_resize_buffers

Mesa can’t determine when a window is resized. When the on-screen window is resized the ancillary (depth, stencil, accumulation) buffers should be resized. The work-around is for Mesa to query the window size whenever glViewport is called. This is usually sufficient since glViewport is usually called soon after a window has been resized. When this isn’t sufficent the programmer can include a call to glResizeBuffersMESA() which forces Mesa to query the current window size and resize the ancillary buffers if needed.

GLX_MESA_release_buffers

Mesa can’t determine when an X window has been destroyed. When a window is destroyed the associated ancillary buffers should also be destroyed. As a work-around, Mesa maintains a list of known rendering windows and whenever glXCreateContext or glXDestroyContext are called checks if any of those windows as been recently destroyed. Since this isn’t sufficient in all situations a programmer can explicitly tell Mesa to free the ancillary buffers by calling glXReleaseBuffersMESA just before calling XDestroyWindow.

GLX_MESA_pixmap_colormap

This extension adds the GLX function:

```
GLXPixmap glXCreateGLXPixmapMESA( Display *dpy, XVisualInfo *visual, Pixmap pixmap,
Colormap cmap )
```

It is an alternative to the standard `glXCreateGLXPixmap` function. Since Mesa supports RGB rendering into any X visual, not just TrueColor or DirectColor, Mesa needs colormap information to convert RGB values into pixel values. An X window carries this information but a pixmap does not. This function associates a colormap to a GLX pixmap.

An application using GLX pixmaps should use the following code to associate a colormap with the GLX pixmap when using Mesa.

```
#ifdef GLX_MESA_pixmap_colormap
   glxpixmap = glXCreateGLXPixmapMESA( display, xvisualinfo,
                                       xpixmap, colormap );
#else
   glxpixmap = glXCreateGLXPixmap( display, xvisualinfo, xpixmap );
#endif
```

5. Future Plans

There are a number of things planned in the future for Mesa.

More optimization

Each Mesa release has usually been a bit faster then the previous one. Optimization is an on-going process. Most recently, optimization of vertex transformation, clipping and lighting has been the focus since the rasterization bottleneck is greatly reduced when 3-D hardware is used.

GLX protocol encoding

Steven Parker (sparker@taz.cs.utah.edu) of the University of Utah has written free GLX encoder/decoder software. By integrating the encoder into Mesa, an application linked with Mesa could send true GLX protocol data to a GLX-equipped X server or send ordinary Xlib protocol to non-GLX X servers.

If the GLX X server has 3-D acceleration hardware the Mesa-linked application would use it.

X server integration

Work is underway to integrate Mesa into the XFree86 X server. This implies implementing the GLX decoder and integrating Mesa so that GLX client applications could render to computers running the XFree86 X server.

Hardware acceleration

Recently, 3-D acceleration hardware for personal computers has become very common and affordable. There have been several efforts to support 3-D hardware with Mesa.

The first was a driver for the GLint chipset written by Ken Adams while at Clemson University. Development is now maintained by others at the university. Dr. Robert Geist (rmg@cs.clemson.edu) is the current contact.

The second was a driver for the Cirrus Logic CL5464 chipset written by Peter McDermott while at the University of Texas at Austin. Again, development continues at the university. Contact Adam Seligman (adams@cs.utexas.edu).

The most recent hardware support is for the 3Dfx VooDoo chipset written by David Bucciarelli (tech.hmw@plus.it). This driver is implemented on the 3Dfx GLide rasterization library.

More hardware acceleration projects will probably follow when Mesa has been integrated with XFree86.

Other possibilities

Other long term items for Mesa development include free versions of the GLS (GL Stream encoder/decoder) library, GLC (GL Character rendering) library and the OpenGL debugger. Work has not yet begun on these projects.

6. Summary

Mesa has turned out to be a very useful and popular 3-D library. Its success can be attributed to the fact that the library is free, full featured, reliable, portable and compatible with OpenGL. Many volunteers have contributed to this success.

Mesa has a bright future with many new features planned. No doubt, much of this work will be done by volunteers who share an enthusiasm for computer graphics and free software.

Appendix A

Obtaining Mesa

Mesa can be downloaded via the Mesa home page at
`http://www.ssec.wisc.edu/~brianp/Mesa.html`

Last edited on April 19, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL/Mesa Off-screen Rendering

Contents

1. Introduction

Normally, OpenGL is used for rendering into a window which is displayed on your computer’s screen. But sometimes it’s useful to render into an image buffer which is not displayed. This is called off-screen rendering.

Some uses of off-screen rendering include:

- Generation of intermediate images such as textures
- Batch rendering of non-interactive animations
- High-resolution image generation for hardcopy

Generaly, off-screen rendering is not a core part of OpenGL; it’s provided by an OpenGL window system interface such as GLX or WGL. Some systems have more than one facility for off-screen rendering, each with its own advantages and disadvantages.

The following sections describe the off-screen rendering facilities for WGL, GLX and Mesa with an emphasis on portablity and performance trade-offs.

2. Microsoft OpenGL Off-Screen Rendering

OpenGL for Windows supports off-screen rendering into Windows *device-independent bitmaps*.

Pros:

- A standard WGL feature

Cons:

- Only usable with Windows 95/NT OpenGL

Basically, a bitmap is created with `CreateDIBSection`. A pixel format with the `PFD_DRAW_TO_BITMAP, PFD_SUPPORT_OPENGL, PFD_SUPPORT_GDI` flags must be chosen. After creating a WGL context and binding it, OpenGL rendering can proceed.

3. GLX Pixmaps

A GLX pixmap is an X Pixmap augmented with a set of ancillary buffers such as a depth buffer, stencil buffer or accumulation buffer.

Pros:

- Buffer contents are retained; cannot be damaged like on-screen windows
- GLX pixmaps are a standard part of GLX
- GLX pixmaps can sometimes be larger than an on-screen window

Cons:

- Rendering into GLX pixmaps may not be accelerated with graphics hardware and, in fact, may be rather slow
- Size may be limited to the screen's size
- Connection to X server is required even though rendering is off-screen

The basic steps to create a GLX pixmap are:

1. Call `XOpenDisplay` to open an X display connection.
2. Select an X visual with `glXChooseVisual`.
3. Create an X pixmap with `XCreatePixmap` specifying the depth of the X visual.
4. Create the GLX pixmap with `glXCreateGLXPixmap`.

The `GLXPixmap` handle returned by `glXCreateGLXPixmap` may be passed to `glXMakeCurrent` to bind an OpenGL rendering context to the GLX pixmap. Rendering into the GLX pixmap may then begin.

The contents of a GLX pixmap may be read back with `glReadPixels` or `XGetImage`.

4. SGI pbuffers

Pbuffers are an OpenGL extension available on recent SGI systems. It is an experimental extension- it may be changed in the future. The purpose of pbuffers is to allow hardware accelerated rendering to an off-screen buffer, possibly with pixel formats which aren’t normally supported by the X display.

Pros:

- Hardware accelerated
- May offer pixel formats not available for ordinary windows

Cons:

- Currently only available on recent SGI systems
- May require special X server configuration
- pbuffers contents may be arbitrarily lost at any time
- Connection to X server is required even though rendering is off-screen
- More difficult to use than GLX pixmaps
- Maximum size may be contrained to screen size

If you are using an SGI system and need *accelerated* off-screen rendering then pbuffers should be considered. Otherwise, GLX pixmaps are a more attractive off-screen rendering solution.

With that in mind let us consider pbuffers in more detail.

The pbuffers extension name is `GLX_SGIX_pbuffers`. Prerequisite to the pbuffers extension is the exerimental fbconfig extension (`GLX_SGIX_fbconfig`).

The fbconfig extension was introduced for several reasons:

- It introduces a new way to describe the capabilities of a GLX drawable, that is, to describe the resolution of color buffer components and the type and size of ancillary buffers by providing a GLXFBConfig construct.
- It relaxes the "similarity" requirement when associating a current context with a drawable.
- It supports RGBA rendering to one- and two-component windows and GLX pixmaps as well as pbuffers.

For more information about the fbconfig extension see the fbconfig.txt file.

Pbuffer applications must test for both the GLX_SGIX_pbuffers and GLX_SGIX_fbconfig extensions. See the Using OpenGL Extensions document for details on extension testing. If either extension is not available the application should fall back to using GLX pixmaps.

The basic steps for creating a pbuffer are:

1. Call `XOpenDisplay` to open an X display connection.
2. Get a `GLXFBConfigSGIX` handle by calling `glXChooseFBConfigSGIX`
3. Create a pbuffer by calling `glXCreateGLXPbuffer`

Several difficulties may arise during these seemingly simple steps:

- `glXChooseFBConfigSGIX` returns a sorted list of fbconfigs which match your attribute list. However, some or all of the fbconfigs may not be usable for making a pbuffer.
- The `glXCreateGLXPbuffer` call may fail, generating an X protocol error. You must set up an X error handler to catch this error so your program doesn't exit abnormally.
- You may have to try several different fbconfig attribute lists before you're able to find one which works.

These difficulties basically boil down to the fact that pbuffers are allocated from the frame buffer which is, in general, of fixed size. Also, the fbconfigs may be staticly configured- a particular combination of buffer attributes may not be supported.

As an example, suppose you need a single-buffered RGB pbuffer with a depth buffer. `glXChooseFBConfigSGIX` may return a list of several fbconfig candidates. However, there may not be enough memory available in the frame buffer for some or any of those fbconfigs. There may be enough memory for the color buffer but not the depth buffer, for example. Or, it may not be possible to allocate a single buffered pbuffer; only double buffered pbuffers may exist.

The best approach is a nested loop:

```
let fbAttribs = list of fbconfig attribute lists
foreach fbAttrib in fbAttribs do
    let fbConfigs = list returned by glXChooseFBConfigSGIX(fbAttrib)
    foreach fbConfig in fbConfigs do
        let pBuffer = glXCreateGLXPbufferSGIX(fbConfig)
        if pBuffer then
            SUCCESS!
        endif
    endfor
endfor
```

The course notes CD-ROM includes sample pbuffer code in the pbuffer.trz file. The *pbdemo.c* program illustrates this approach. See the `MakePbuffer` function.

The *pbutil.c* file contains several pbuffer utility functions. The `CreatePbuffer` handles the X protocol error problem.

The *pbinfo.c* program is similar to *glxinfo*. It prints a list of fbconfigs available on your system and whether or not a pbuffer of that config can be created.

System Configuration

Some SGI systems require reconfiguring the display / X server to enable pbuffers (or at least useful pbuffer configurations).

On SGI Impact systems, for example, if you look in the `/usr/gfx/ucode/MGRAS/vof/` directory you will find a list of video output formats supported by the Impact architecture. Look for ones with the `_pbuf` suffix. Use the `setmon -x` utility to configure your X server to use a pbuffer-enabled video format.

5. Auxiliary Buffers

The OpenGL specification includes *auxillary* buffers. These are buffers intended for off-screen rendering. They are addressed via the `glDrawBuffer` and `glReadBuffer` functions. Up to four auxiliary buffers named `GL_AUX0, GL_AUX1, GL_AUX2,` and `GL_AUX3` are available. The actual number of auxiliary buffers available can be queried with `glGetIntegerv(GL_AUX_BUFFERS, numBuffers)`.

Pros:

- A simple off-screen facility standard to OpenGL.

Cons:

- Aux buffers are optional and few implementations of OpenGL support them.

6. Mesa

Mesa includes a special off-screen rendering interface called OSMesa. It's unique in that the interface has no dependencies on any operating system or window system.

Pros:

- No window system or operating system dependencies

Cons:

- Only available in Mesa
- Probably no chance of hardware accelerated rendering

Mesa's off-screen rendering interface is quite simple. Documentation for it may be found in the Mesa README file and there is an example program in the Mesa distribution (demos/osdemo.c).

7. Tiled Rendering

Tiled rendering is a technique in which a large image is produced by tiling together smaller, individually rendered images. It's useful for generating images which are larger than what OpenGL would normally permit.

OpenGL and/or window systems limit the size of rendered imagery in several ways:

- The window system may not allow one to create windows, pixmaps or pbuffers which larger than the screen's size. Typical limits are 1280 by 1024 pixels.
- `glViewport`'s width and height parameters are silently clamped to an implementation-dependant limit. These limits can be queried via `glGetIntegerv` with the argument `GL_MAX_VIEWPORT_DIMS`. Typical limits are 2048 by 2048 pixels.

The basic technique of tiled rendering is to draw your entire scene for each tile, adjusting the projection and viewport parameters such that when the tiles are assembled there are no seams. Unfortunately, this is easier said than done. To make tiled rendering easier I have developed a tile rendering utility library for this course.

Here is a modified excerpt of the *trdemo1.c* example program which demonstrates how to use the tr (tile rendering) library:

```
static void Display(void)
{
   GLubyte *image;
   TRcontext *tr;

   /* allocate final image buffer */
   image = malloc(WindowWidth * WindowHeight * 4 * sizeof(GLubyte));
   if (!image) {
      printf("Malloc failed!\n");
      return;
   }

   /* Setup tiled rendering.  Each tile is TILESIZE x TILESIZE pixels. */
   tr = trNew();
   trTileSize(tr, TILESIZE, TILESIZE);
   trImageSize(tr, WindowWidth, WindowHeight);
   trImageBuffer(tr, GL_RGBA, GL_UNSIGNED_BYTE, image);

   if (Perspective)
      trFrustum(tr, -1.0, 1.0, -1.0, 1.0, 5.0, 25.0);
   else
      trOrtho(tr, -3.0, 3.0, -3.0, 3.0, -3.0, 3.0);

   /* Draw tiles */
   do {
      trBeginTile(tr);
      DrawScene();
   } while (trEndTile(tr));

   trDelete(tr);

   /* 'image' buffer now contains the final image.
    * You could now print it, write it to a file, etc.
    */
}
```

The basic steps are:

1. Allocate memory for the final image.
2. Create a tile rendering context with `trNew`.
3. Call `trTileSize` to specify the tile size.
4. Call `trImageSize` to specify the final image size.

5. Call `trImageBuffer` to specify where the final image is to be stored.
6. Setup a perspective or orthographic projection with `trFrustum` or `trOrtho`.
7. Call the `trBeginTile` and `trEndTile` functions inside a loop which surrounds your scene drawing function until `trEndTile` returns zero.
8. Free the tile rendering context with `trDelete`.

The final image is typically written to a file or sent to a printer.

There is one caveat to this utility library: `glRasterPos, glDrawPixels` and `glBitmap` may be troublesome. The problem is that if `glRasterPos` specifies a coordinate which falls outside the current viewport, the current raster position becomes invalid. If the current raster position is invalid subsequent calls to `glDrawPixels` or `glBitmap` will have no consequence.

The solution to this problem is the `trRasterPos3f` function. It works just like `glRasterPos3f` but doesn't suffer from the invalid raster position problem. See the *trdemo1.c* program for example usage.

The *trdemo2.c* example demonstrates how to generate very large image files without allocating a full-size image buffer.

Last edited on April 29, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Performance Optimization

Contents

1. Hardware vs. Software

OpenGL may be implemented by any combination of hardware and software. At the high-end, hardware may implement virtually all of OpenGL while at the low-end, OpenGL may be implemented entirely in software. In between are combination software/hardware implementations. More money buys more hardware and better performance.

Intro-level workstation hardware and the recent PC 3-D hardware typically implement point, line, and polygon rasterization in hardware but implement floating point transformations, lighting, and clipping in software. This is a good strategy since the bottleneck in 3-D rendering is usually rasterization and modern CPU’s have sufficient floating point performance to handle the transformation stage.

OpenGL developers must remember that their application may be used on a wide variety of OpenGL implementations. Therefore one should consider using all possible optimizations, even those which have little return on the development system, since other systems may benefit greatly.

From this point of view it may seem wise to develop your application on a low-end system. There is a pitfall however; some operations which are cheep in software may be expensive in hardware. The moral is: test your application on a variety of systems to be sure the performance is dependable.

2. Application Organization

At first glance it may seem that the performance of interactive OpenGL applications is dominated by the performance of OpenGL itself. This may be true in some circumstances but be aware that the organization of the application is also significant.

2.1 High Level Organization

Multiprocessing

Some graphical applications have a substantial computational component other than 3-D rendering. Virtual reality applications must compute object interactions and collisions. Scientific visualization programs must compute analysis functions and graphical representations of data.

One should consider multiprocessing in these situations. By assigning rendering and computation to different threads they may be executed in parallel on multiprocessor computers.

For many applications, supporting multiprocessing is just a matter of partitioning the render and compute operations into separate threads which share common data structures and coordinate with synchronization primitives.

SGI’s Performer is an example of a high level toolkit designed for this purpose.

Image quality vs. performance

In general, one wants high-speed animation and high-quality images in an OpenGL application. If you can’t have both at once a reasonable compromise may be to render at low complexity during animation and high complexity for static images.

Complexity may refer to the geometric or rendering attributes of a database. Here are a few examples.

- During interactive rotation (i.e. mouse button held down) render a reduced-polygon model. When drawing a static image draw the full polygon model.
- During animation, disable dithering, smooth shading, and/or texturing. Enable them for the static image.
- If texturing is required, use `GL_NEAREST` sampling and `glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST )`.

- During animation, disable antialiasing. Enable antialiasing for the static image.
- Use coarser NURBS/evaluator tesselation during animation. Use `glPolygonMode( GL_FRONT_AND_BACK, GL_LINE )` to inspect tesselation granularity and reduce if possible.

Level of detail management and culling

Objects which are distant from the viewer may be rendered with a reduced complexity model. This strategy reduces the demands on all stages of the graphics pipeline. Toolkits such as Inventor and Performer support this feature automatically.

Objects which are entirely outside of the field of view may be culled. This type of high level cull testing can be done efficiently with bounding boxes or spheres and have a major impact on performance. Again, toolkits such as Inventor and Performer have this feature.

2.2 Low Level Organization

The objects which are rendered with OpenGL have to be stored in some sort of data structure. Some data structures are more efficient than others with respect to how quickly they can be rendered.

Basically, one wants data structures which can be traversed quickly and passed to the graphics library in an efficient manner. For example, suppose we need to render a triangle strip. The data structure which stores the list of vertices may be implemented with a linked list or an array. Clearly the array can be traversed more quickly than a linked list. The way in which a vertex is stored in the data structure is also significant. High performance hardware can process vertexes specified by a pointer more quickly than those specified by three separate parameters.

An Example

Suppose we're writing an application which involves drawing a road map. One of the components of the database is a list of cities specified with a latitude, longitude and name. The data structure describing a city may be:

```
struct city {
        float latitute, longitude;      /* city location */
        char *name;                     /* city's name */
        int large_flag;                 /* 0 = small, 1 = large */
};
```

A list of cities may be stored as an array of city structs.

Our first attempt at rendering this information may be:

```
void draw_cities( int n, struct city citylist[] )
{
   int i;
   for (i=0; i < n; i++) {
      if (citylist[i].large_flag) {
         glPointSize( 4.0 );
      }
      else {
         glPointSize( 2.0 );
```

```
            }
            glBegin( GL_POINTS );
            glVertex2f( citylist[i].longitude, citylist[i].latitude );
            glEnd();
            glRasterPos2f( citylist[i].longitude, citylist[i].latitude );
            glCallLists( strlen(citylist[i].name),
                         GL_BYTE,
                         citylist[i].name );
         }
      }
```

This is a poor implementation for a number of reasons:

- `glPointSize` is called for every loop iteration.
- only one point is drawn between `glBegin` and `glEnd`
- the vertices aren't being specified in the most efficient manner

Here's a better implementation:

```
      void draw_cities( int n, struct city citylist[] )
      {
         int i;
         /* draw small dots first */
         glPointSize( 2.0 );
         glBegin( GL_POINTS );
         for (i=0; i < n ;i++) {
            if (citylist[i].large_flag==0) {
               glVertex2f( citylist[i].longitude, citylist[i].latitude );
            }
         }
         glEnd();
         /* draw large dots second */
         glPointSize( 4.0 );
         glBegin( GL_POINTS );
         for (i=0; i < n ;i++) {
            if (citylist[i].large_flag==1) {
               glVertex2f( citylist[i].longitude, citylist[i].latitude );
            }
         }
         glEnd();
         /* draw city labels third */
         for (i=0; i < n ;i++) {
            glRasterPos2f( citylist[i].longitude, citylist[i].latitude );
            glCallLists( strlen(citylist[i].name),
                         GL_BYTE,
                         citylist[i].name );
         }
      }
```

In this implementation we're only calling glPointSize twice and we're maximizing the number of vertices specified between `glBegin` and `glEnd`.

We can still do better, however. If we redesign the data structures used to represent the city information we can improve the efficiency of drawing the city points. For example:

```
      struct city_list {
              int num_cities;         /* how many cities in the list */
```

```
        float *position;        /* pointer to lat/lon coordinates */
        char **name;            /* pointer to city names */
        float size;             /* size of city points */
};
```

Now cities of different sizes are stored in separate lists. Position are stored sequentially in a dynamically allocated array. By reorganizing the data structures we've eliminated the need for a conditional inside the `glBegin/glEnd` loops. Also, we can render a list of cities using the `GL_EXT_vertex_array` extension if available, or at least use a more efficient version of `glVertex` and `glRasterPos`.

```
/* indicates if server can do GL_EXT_vertex_array: */
GLboolean varray_available;

void draw_cities( struct city_list *list )
{
    int i;
    GLboolean use_begin_end;

    /* draw the points */
    glPointSize( list->size );

#ifdef GL_EXT_vertex_array
    if (varray_available) {
        glVertexPointerEXT( 2, GL_FLOAT, 0, list->num_cities, list->position );
        glDrawArraysEXT( GL_POINTS, 0, list->num_cities );
        use_begin_end = GL_FALSE;
    }
#else
    use_begin_end = GL_TRUE;
#endif

    if (use_begin_end) {
        for (i=0; i < list->num_cities; i++) {
            glVertex2fv( &position[i*2] );
        }
    }

    /* draw city labels */
    for (i=0; i < list->num_cities ;i++) {
        glRasterPos2fv( list->position[i*2] );
        glCallLists( strlen(list->name[i]),
                        GL_BYTE, list->name[i] );
    }
}
```

As this example shows, it's better to know something about efficient rendering techniques before designing the data structures. In many cases one has to find a compromize between data structures optimized for rendering and those optimized for clarity and convenience.

In the following sections the techniques for maximizing performance, as seen above, are explained.

3. OpenGL Optimization

There are many possibilities to improving OpenGL performance. The impact of any single optimization can vary a great deal depending on the OpenGL implementation. Interestingly, items which have a large impact on software renderers may have no effect on hardware renderers, *and vice versa*! For example, smooth shading can be expensive in software but free in hardware While `glGet*` can be cheap in software but expensive in hardware.

After each of the following techniques look for a bracketed list of symbols which relates the significance of the optimization to your OpenGL system:

- H - beneficial for high-end hardware
- L - beneficial for low-end hardware
- S - beneficial for software implementations
- all - probably beneficial for all implementations

3.1 Traversal

Traversal is the sending of data to the graphics system. Specifically, we want to minimize the time taken to specify primitives to OpenGL.

Use connected primitives

Connected primitives such as `GL_LINES`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, and `GL_QUAD_STRIP` require fewer vertices to describe an object than individual line, triangle, or polygon primitives. This reduces data transfer and transformation workload. [all]

Use the vertex array extension

On some architectures function calls are somewhat expensive so replacing many `glVertex/glColor/glNormal` calls with the vertex array mechanism may be very beneficial. [all]

Store vertex data in consecutive memory locations

When maximum performance is needed on high-end systems it’s good to store vertex data in contiguous memory to maximize through put of data from host memory to graphics subsystem. [H,L]

Use the vector versions of `glVertex`, `glColor`, `glNormal` and `glTexCoord`

The `glVertex`, `glColor`, etc. functions which take a pointer to their arguments such as `glVertex3fv(v)` may be much faster than those which take individual arguments such as `glVertex3f(x,y,z)` on systems with DMA-driven graphics hardware. [H,L]

Reduce quantity of primitives

Be careful not to render primitives which are over-tesselated. Experiment with the GLU primitives, for example, to determine the best compromise of image quality vs. tesselation level. Textured objects in particular may still be rendered effectively with low geometric complexity. [all]

Display lists

Use display lists to encapsulate frequently drawn objects. Display list data may be stored in the graphics subsystem rather than host memory thereby eliminating host-to-graphics data movement. Display lists are also very beneficial when rendering remotely. [all]

Don’t specify unneeded per-vertex information

If lighting is disabled don’t call `glNormal`. If texturing is disabled don’t call `glTexCoord`, etc.

Minimize code between `glBegin/glEnd`

For maximum performance on high-end systems it’s extremely important to send vertex data to the graphics system as fast as possible. Avoid extraneous code between `glBegin/glEnd`.

Example:

```
        glBegin( GL_TRIANGLE_STRIP );
        for (i=0; i < n; i++) {
                if (lighting) {
                        glNormal3fv( norm[i] );
                }
                glVertex3fv( vert[i] );
        }
        glEnd();
```

This is a very bad construct. The following is much better:

```
        if (lighting) {
                glBegin( GL_TRIANGLE_STRIP );
                for (i=0; i < n ;i++) {
                        glNormal3fv( norm[i] );
                        glVertex3fv( vert[i] );
                }
                glEnd();
        }
        else {
                glBegin( GL_TRIANGLE_STRIP );
                for (i=0; i < n ;i++) {
                        glVertex3fv( vert[i] );
                }
                glEnd();
        }
```

Also consider manually unrolling important rendering loops to maximize the function call rate.

3.2 Transformation

Transformation includes the transformation of vertices from `glVertex` to window coordinates, clipping and lighting.

Lighting

- ○ Avoid using positional lights, i.e. light positions should be of the form (x,y,z,0) [L,S]
- ○ Avoid using spotlights. [all]
- ○ Avoid using two-sided lighting. [all]
- ○ Avoid using negative material and light color coefficients [S]
- ○ Avoid using the local viewer lighting model. [L,S]
- ○ Avoid frequent changes to the `GL_SHININESS` material parameter. [L,S]
- ○ Some OpenGL implementations are optimized for the case of a single light source.

- Consider pre-lighting complex objects before rendering, ala radiosity. You can get the effect of lighting by specifying vertex colors instead of vertex normals. [S]

Two sided lighting

If you want both the front and back of polygons shaded the same try using two light sources instead of two-sided lighting. Position the two light sources on opposite sides of your object. That way, a polygon will always be lit correctly whether it's back or front facing. [L,S]

Disable normal vector normalization when not needed

`glEnable/Disable(GL_NORMALIZE)` controls whether normal vectors are scaled to unit length before lighting. If you do not use `glScale` you may be able to disable normalization without ill effects. Normalization is disabled by default. [L,S]

Use connected primitives

Connected primitives such as `GL_LINES`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, and `GL_QUAD_STRIP` decrease traversal and transformation load.

`glRect` usage

If you have to draw many rectangles consider using `glBegin(GL_QUADS)` ... `glEnd()` instead. [all]

3.3 Rasterization

Rasterization is the process of generating the pixels which represent points, lines, polygons, bitmaps and the writing of those pixels to the frame buffer. Rasterization is often the bottleneck in software implementations of OpenGL.

Disable smooth shading when not needed

Smooth shading is enabled by default. Flat shading doesn't require interpolation of the four color components and is usually faster than smooth shading in software implementations. Hardware may perform flat and smooth-shaded rendering at the same rate though there's at least one case in which smooth shading is faster than flat shading (E&S Freedom). [S]

Disable depth testing when not needed

Background objects, for example, can be drawn without depth testing if they're drawn first. Foreground objects can be drawn without depth testing if they're drawn last. [L,S]

Disable dithering when not needed

This is easy to forget when developing on a high-end machine. Disabling dithering can make a big difference in software implementations of OpenGL on lower-end machines with 8 or 12-bit color buffers. Dithering is enabled by default. [S]

Use back-face culling whenever possible.

If you're drawing closed polyhedra or other objects for which back facing polygons aren't visible there's probably no point in drawing those polygons. [all]

The GL_SGI_cull_vertex extension

SGI's Cosmo GL supports a new culling extension which looks at vertex normals to try to improve the speed of culling.

Avoid extra fragment operations

Stenciling, blending, stippling, alpha testing and logic ops can all take extra time during rasterization. Be sure to disable the operations which aren't needed. [all]

Reduce the window size or screen resolution

A simple way to reduce rasterization time is to reduce the number of pixels drawn. If a smaller window or reduced display resolution are acceptable it's an easy way to improve rasterization speed. [L,S]

3.4 Texturing

Texture mapping is usually an expensive operation in both hardware and software. Only high-end graphics hardware can offer free to low-cost texturing. In any case there are several ways to maximize texture mapping performance.

Use efficient image formats

The `GL_UNSIGNED_BYTE` component format is typically the fastest for specifying texture images. Experiment with the internal texture formats offered by the `GL_EXT_texture` extension. Some formats are faster than others on some systems (16-bit texels on the Reality Engine, for example). [all]

Encapsulate texture maps in texture objects or display lists

This is especially important if you use several texture maps. By putting textures into display lists or texture objects the graphics system can manage their storage and minimize data movement between the client and graphics subsystem. [all]

Use smaller texture maps

Smaller images can be moved from host to texture memory faster than large images. More small texture can be stored simultaneously in texture memory, reducing texture memory swapping. [all]

Use simpler sampling functions

Experiment with the minification and magnification texture filters to determine which performs best while giving acceptable results. Generally, GL_NEAREST is fastest and GL_LINEAR is second fastest. [all]

Use a simpler texture environment function

Some texture environment modes may be faster than others. For example, the `GL_DECAL` or `GL_REPLACE_EXT` functions for 3 component textures is a simple assignment of texel samples to fragments while `GL_MODULATE` is a linear interpolation between texel samples and incoming fragments. [S,L]

Combine small textures

If you are using several small textures consider tiling them together as a larger texture and modify your texture coordinates to address the subtexture you want. This technique is a good way to eliminate texture binding time.

Use glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST)

This hint can improve the speed of texturing when perspective- correct texture coordinate interpolation isn't needed, such as when using a glOrtho() projection.

Animated textures

If you want to use an animated texture, perhaps live video textures, don't use `glTexImage2D` to repeatedly change the texture. Use `glTexSubImage2D` or `glTexCopyTexSubImage2D`. These functions are standard in OpenGL 1.1 and available as extensions to 1.0.

3.5 Clearing

Clearing the color, depth, stencil and accumulation buffers can be time consuming, especially when it has to be done in software. There are a few tricks which can help.

Use `glClear` carefully [all]

Clear all relevant color buffers with one `glClear`.

Wrong:

```
glClear( GL_COLOR_BUFFER_BIT );
if (stenciling) {
      glClear( GL_STENCIL_BUFFER_BIT );
}
```

Right:

```
if (stenciling) {
      glClear( GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
}
else {
      glClear( GL_COLOR_BUFFER_BIT );
}
```

Disable dithering

Disable dithering before clearing the color buffer. Visually, the difference between dithered and undithered clears is usually negligable.

Use scissoring to clear a smaller area

If you don't need to clear the whole buffer use `glScissor()` to restrict clearing to a smaller area. [L].

Don't clear the color buffer at all

If the scene you're drawing opaquely covers the entire window there is no reason to clear the color buffer.

Eliminate depth buffer clearing

If the scene you're drawing covers the entire window there is a trick which let's you omit the depth buffer clear. The idea is to only use half the depth buffer range for each frame and alternate between using GL_LESS and GL_GREATER as the depth test function.

Example:

```
int EvenFlag;

/* Call this once during initialization and whenever the window
 * is resized.
 */
void init_depth_buffer( void )
{
     glClearDepth( 1.0 );
     glClear( GL_DEPTH_BUFFER_BIT );
     glDepthRange( 0.0, 0.5 );
     glDepthFunc( GL_LESS );
     EvenFlag = 1;
}

/* Your drawing function */
void display_func( void )
{
     if (EvenFlag) {
         glDepthFunc( GL_LESS );
         glDepthRange( 0.0, 0.5 );
     }
     else {
         glDepthFunc( GL_GREATER );
         glDepthRange( 1.0, 0.5 );
     }
     EvenFlag = !EvenFlag;

     /* draw your scene */
}
```

Avoid glClearDepth(d) where d!=1.0

Some software implementations may have optimized paths for clearing the depth buffer to 1.0. [S]

3.6 Miscellaneous

Avoid "round-trip" calls

Calls such as `glGetFloatv`, `glGetIntegerv`, `glIsEnabled`, `glGetError`, `glGetString` require a slow, round trip transaction between the application and renderer. Especially avoid them in your main rendering code.

Note that software implementations of OpenGL may actually perform these operations faster than hardware systems. If you're developing on a low-end system be aware of this fact. [H,L]

Avoid `glPushAttrib`

If only a few pieces of state need to be saved and restored it's often faster to maintain the information in the client program. `glPushAttrib( GL_ALL_ATTRIB_BITS )` in particular can be very expensive on hardware systems. This call may be faster in software implementations than in hardware. [H,L]

Check for GL errors during development

During development call `glGetError` inside your rendering/event loop to catch errors. GL errors raised during rendering can slow down rendering speed. Remove the `glGetError` call for production code since it's a "round trip" command and can cause delays. [all]

Use `glColorMaterial` instead of `glMaterial`

If you need to change a material property on a per vertex basis, `glColorMaterial` may be faster than `glMaterial`. [all]

`glDrawPixels`

- `glDrawPixels` often performs best with `GL_UNSIGNED_BYTE` color components [all]
- Disable all unnecessary raster operations before calling `glDrawPixels`. [all]
- Use the GL_EXT_abgr extension to specify color components in alpha, blue, green, red order on systems which were designed for IRIS GL. [H,L].

Avoid using viewports which are larger than the window

Software implementations may have to do additional clipping in this situation. [S]

Alpha planes

Don’t allocate alpha planes in the color buffer if you don’t need them. Specifically, they are not needed for transparency effects. Systems without hardware alpha planes may have to resort to a slow software implementation. [L,S]

Accumulation, stencil, overlay planes

Do not allocate accumulation, stencil or overlay planes if they are not needed. [all]

Be aware of the depth buffer’s depth

Your OpenGL may support several different sizes of depth buffers- 16 and 24-bit for example. Shallower depth buffers may be faster than deep buffers both for software and hardware implementations. However, the precision of of a 16-bit depth buffer may not be sufficient for some applications. [L,S]

Transparency may be implemented with stippling instead of blending

If you need simple transparent objects consider using polygon stippling instead of alpha blending. The later is typically faster and may actually look better in some situations. [L,S]

Group state changes together

Try to mimimize the number of GL state changes in your code. When GL state is changed, internal state may have to be recomputed, introducing delays. [all]

Avoid using `glPolygonMode`

If you need to draw many polygon outlines or vertex points use `glBegin` with `GL_POINTS`, `GL_LINES`, `GL_LINE_LOOP` or `GL_LINE_STRIP` instead as it can be much faster. [all]

3.7 Window System Integration

Minimize calls to the *make current* call

The `glXMakeCurrent` call, for example, can be expensive on hardware systems because the context switch may involve moving a large amount of data in and out of the hardware.

Visual / pixel format performance

Some X visuals or pixel formats may be faster than others. On PCs for example, 24-bit color buffers may be slower to read/write than 12 or 8-bit buffers. There is often a tradeoff between performance and quality of frame buffer configurations. 12-bit color may not look as nice as 24-bit color. A 16-bit depth buffer won't have the precision of a 24-bit depth buffer.

The `GLX_EXT_visual_rating` extension can help you select visuals based on performance or quality. GLX 1.2's *visual caveat* attribute can tell you if a visual has a performance penalty associated with it.

It may be worthwhile to experiment with different visuals to determine if there's any advantage of one over another.

Avoid mixing OpenGL rendering with native rendering

OpenGL allows both itself and the native window system to render into the same window. For this to be done correctly synchronization is needed. The GLX `glXWaitX` and `glXWaitGL` functions serve this purpose.

Synchronization hurts performance. Therefore, if you need to render with both OpenGL and native window system calls try to group the rendering calls to minimize synchronization.

For example, if you're drawing a 3-D scene with OpenGL and displaying text with X, draw all the 3-D elements first, call `glXWaitGL` to synchronize, then call all the X drawing functions.

Don't redraw more than necessary

Be sure that you're not redrawing your scene unnecissarily. For example, expose/repaint events may come in batches describing separate regions of the window which must be redrawn. Since one usually redraws the whole window image with OpenGL you only need to respond to one expose/repaint event. In the case of X, look at the count field of the XExposeEvent structure. Only redraw when it is zero.

Also, when responding to mouse motion events you should skip extra motion events in the input queue. Otherwise, if you try to process every motion event and redraw your scene there will be a noticable delay between mouse input and screen updates.

It can be a good idea to put a print statement in your redraw and event loop function so you know exactly what messages are causing your scene to be redrawn, and when.

SwapBuffer calls and graphics pipe blocking

On systems with 3-D graphics hardware the SwapBuffers call is synchronized to the monitor's vertical retrace. Input to the OpenGL command queue may be blocked until the buffer swap has completed. Therefore, don't put more OpenGL calls immediately after SwapBuffers. Instead, put application computation instructions which can overlap with the buffer swap delay.

3.8 Mesa-specific

Mesa is a free library which implements most of the OpenGL API in a compatible manner. Since it is a software library, performance depends a great deal on the host computer. There are several Mesa-specific features to be aware of which can effect performance.

Double buffering

The X driver supports two back color buffer implementations: Pixmaps and XImages. The MESA_BACK_BUFFER environment variable controls which is used. Which of the two that's faster depends on the nature of your rendering. Experiment.

X Visuals

As described above, some X visuals can be rendered into more quickly than others. The `MESA_RGB_VISUAL` environment variable can be used to determine the quickest visual by experimentation.

Depth buffers

Mesa may use a 16 or 32-bit depth buffer as specified in the src/config.h configuration file. 16-bit depth buffers are faster but may not offer the precision needed for all applications.

Flat-shaded primitives

If one is drawing a number of flat-shaded primitives all of the same color the `glColor` command should be put before the `glBegin` call.

Don't do this:

```
glBegin(...);
glColor(...);
glVertex(...);
...
glEnd();
```

Do this:

```
glColor(...);
glBegin(...);
glVertex(...);
...
glEnd();
```

glColor*() commands

The `glColor[34]ub[v]` are the fastest versions of the `glColor` command.

Avoid double precision valued functions

Mesa does all internal floating point computations in single precision floating point. API functions which take double precision floating point values must convert them to single precision. This can be expensive in the case of glVertex, glNormal, etc.

4. Evaluation and Tuning

To maximize the performance of an OpenGL applications one must be able to evaluate an application to learn what is limiting its speed. Because of the hardware involved it's not sufficient to use ordinary profiling tools. Several different aspects of the graphics system must be evaluated.

Performance evaluation is a large subject and only the basics are covered here. For more information see "OpenGL on Silicon Graphics Systems".

4.1 Pipeline tuning

The graphics system can be divided into three subsystems for the purpose of performance evaluation:

- **CPU subsystem** - application code which drives the graphics subsystem
- **Geometry subsystem** - transformation of vertices, lighting, and clipping
- **Rasterization subsystem** - drawing filled polygons, line segments and per-pixel processing

At any given time, one of these stages will be the bottleneck. The bottleneck must be reduced to improve performance. The strategy is to isolate each subsystem in turn and evaluate changes in performance. For example, by decreasing the workload of the CPU subsystem one can determine if the CPU or graphics system is limiting performance.

4.1.1 CPU subsystem

To isosulate the CPU subsystem one must reduce the graphics workload while presevering the application’s execution characteristics. A simple way to do this is to replace `glVertex()` and `glNormal` calls with `glColor` calls. If performance does not improve then the CPU stage is the bottleneck.

4.1.2 Geometry subsystem

To isoslate the geometry subsystem one wants to reduce the number of primitives processed, or reduce the transformation work per primitive while producing the same number of pixels during rasterization. This can be done by replacing many small polygons with fewer large ones or by simply disabling lighting or clipping. If performance increases then your application is bound by geometry/transformation speed.

4.1.3 Rasterization subsystem

A simple way to reduce the rasterization workload is to make your window smaller. Other ways to reduce rasterization work is to disable per-pixel processing such as texturing, blending, or depth testing. If performance increases, your program is *fill limited*.

After bottlenecks have been identified the techniques outlined in section 3 can be applied. The process of identifying and reducing bottlenecks should be repeated until no further improvements can be made or your minimum performance threshold has been met.

4.2 Double buffering

For smooth animation one must maintain a high, constant frame rate. Double buffering has an important effect on this. Suppose your application needs to render at 60Hz but is only getting 30Hz. It’s a mistake to think that you must reduce rendering time by 50% to achive 60Hz. The reason is the swap-buffers operation is synchronized to occur during the display’s vertical retrace period (at 60Hz for example). It may be that your application is taking only a tiny bit too long to meet the 1/60 second rendering time limit for 60Hz.

Measure the performance of rendering in single buffer mode to determine how far you really are from your target frame rate.

4.3 Test on several implementations

The performance of OpenGL implementations varies a lot. One should measure performance and test OpenGL applications on several different systems to be sure there are no unexpected problems.

Last edited on April 14, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Portability Notes

Contents

1. Introduction

Though OpenGL is an extremely portable 3-D graphics API there are some things to be careful of. OpenGL has some built-in limits and there are a number of too-common implementation errors that OpenGL developers should be aware of to ensure portability.

2. OpenGL Limits

The OpenGL specification calls for certain minimum requirements in any OpenGL implementation. These limits may be extended in some implementations but to be safe, developers should be aware of the minimum requirements.

Limits may be queried with the `glGetInteger` and related functions.

Texture Size

Implementations must support textures of at least 64 by 64 texels. Larger textures are usually supported but consider the possibility that you may be limited to 64 by 64. 512 by 512 is a common limit.

The maximum texture size can also depend on whether you’re using texture borders or mipmapping. OpenGL 1.1 and the GL_EXT_texture extension offer proxy textures which better indicate the maximum texture size than `glGet`.

Pixel Maps

Pixel maps (`glPixelMap`) must support at least 32 entries. Larger maps of 256 or 4096 entries are common.

Selection stack
The selection stack may be as small as 64 names.

Evaluators
Evaluators may be limited to 8 control points. A larger number of control points is frequently supported.

Stacks Depths
The MODELVIEW matrix stack size is at least 32 matrices.

The PROJECTION matrix stack size is at least 2 matrices.

The TEXTURE matrix stack size is at least 2 matrices.

The attribute stack size is at least 16. Similarly, the client attribute stack (OpenGL 1.1) size is at least 16.

Point and line sizes
Maximum point size may be 1 pixel. Maximum line width may be 1 pixel. Antialiased points and lines are often limited to one size.

Viewports
The maximum viewport size may be limited to your screen size. Frequently, the maximum viewport size is 2048 by 2048.

Lights
At least eight light sources must be available. Seldom are more supported.

Clipping Planes
At least six user-definable clipping planes must be available. Seldom are more supported.

3. OpenGL Bugs

Unfortunately, OpenGL implementations often have some minor (and occasionally, major) bugs. Typically, these bugs are found in the more obscure corners of OpenGL so they don’t effect most applications.

In some cases the hardware is at fault and the likelihood of a fix is slim, short of hardware redesign. In other cases a subsequent OpenGL software release may fix the problem.

Here are some tips on dealing with OpenGL bugs:

- Read your system’s OpenGL release notes. They often include lists of known bugs and work-arounds.

- Read the man pages for OpenGL commands which you suspect may have bugs. They're often document at the end.
- If you've found an undocumented OpenGL bug check if a new release of the software is available.
- Finally, if you've really found a new bug you should report it to your OpenGL vendor. If you can provide a simple test case with the bug report you'll make it much easier for the vendor to verify and hopefully fix the bug.

Here is a small collection of known OpenGL problems discovered from personal experience. ***Please note*** that the following information may become obsolete at any time upon the release of updated software.

Texture borders

> Texture borders are not supported on some systems such as the SGI Infinite Reality system. Luckily, the functionality provided by texture borders can be achieved with the GL_SGIS_texture_border_clamp and GL_SGIS_texture_edge_clamp extensions.
>
> It's probably best to avoid using OpenGL texture borders in general.

Texture formats

> Several SGI systems (Impact and possibly Reality Engine) don't support GL_ALPHA (internal format) textures.

glTexImage error checking

> glTexImage[12]D doesn't generate an error if the texture sizes are not powers of two on some SGI systems.

Line Stippling

> The line stipple counter isn't reset upon glBegin() on SGI Impact and IR systems.

Texture objects

> Texture objects which are shared by several rendering contexts don't work correctly on SGI Impact systems.

Last edited on April 14, 1997 by Brian Paul.

Togl - a Tk OpenGL widget

Version 1.2

Copyright (C) 1996 Brian Paul and Ben Bederson

Introduction

Togl is a Tk widget for OpenGL rendering. Togl is originally based on OGLTK, written by Benjamin Bederson at the University of New Mexico. Togl adds the new features:

- color-index mode support including color allocation functions
- support for requesting stencil, accumulation, alpha buffers, etc
- multiple OpenGL drawing widgets
- OpenGL extension testing from Tcl
- simple, portable font support

Togl allows one to create and manage a special Tk/OpenGL widget with Tcl and render into it with a C program. That is, a typical Togl program will have Tcl code for managing the user interface and a C program for computations and OpenGL rendering.

Togl is copyrighted by Brian Paul (brianp@elastic.avid.com) and Benjamin Bederson (bederson@cs.unm.edu). See the LICENSE file for details.

The Togl WWW page is available from:

- Wisconsin at http://www.ssec.wisc.edu/~brianp/Togl.html
- New Mexico at http://www.cs.unm.edu/~bederson/Togl.html

Prerequisites

You should have Tcl and Tk installed on your computer, including the Tk source code files. Togl has been tested with Tcl 7.4/Tk 4.0, Tcl 7.5/Tk 4.1 and Tcl 7.6/Tk 4.2 at this time. It is currently configured for Tcl7.6/Tk4.2.

You must also have OpenGL or Mesa (a free alternative to OpenGL) installed on your computer.

One should be familiar with Tcl, Tk, OpenGL, and C programming to use Togl effectively.

Getting Togl

The current version of Togl is 1.2. You may download it from either:

- Wisconsin at ftp://iris.ssec.wisc.edu/pub/misc/Togl-1.2.tar.gz
- New Mexico at ftp://ftp.cs.unm.edu/pub/bederson/Togl-1.2.tar.gz

Togl may also be obtained manually with ftp:

- Host: iris.ssec.wisc.edu
- Login: anonymous
- Password: your email address
- Directory: pub/misc
- File: Togl-1.2.tar.gz

The Makefile included with Togl is configured for SGI systems. It shouldn’t be hard to adapt it for others. In practice, you’ll just add togl.c to your application’s Makefile.

Using Togl With Your Application

Since the Togl code is in just three files (togl.c, togl.h and tkInt.h) it’s probably most convenient to just include those files with your application sources. The Togl code could be made into a library but that’s not necessary.

C Togl Functions

These are the Togl commands one may call from a C program.

```
#include "togl.h"
```

Setup and Initialization Functions

```
int Togl_Init( Tcl_Interp *interp )
```

Initializes the Togl module. This is typically called from the Tk_Main() callback function.

```
void Togl_CreateFunc( Togl_Callback *proc )
void Togl_DisplayFunc( Togl_Callback *proc )
void Togl_ReshapeFunc( Togl_Callback *proc )
void Togl_DestroyFunc( Togl_Callback *proc )
```

Register C functions to be called by Tcl/Tk when a widget is realized, must be redrawn, is resized, or is destroyed respectively.

Each C callback must be of the form:

```
void callback( struct Togl *togl )
{
   ...your code...
}
```

`void Togl_CreateCommand( char *cmd_name, Togl_CmdProc *cmd_proc )`

Used to create a new Togl sub-command. The C function which implements the command must be of the form:

```
int callback( struct Togl *togl, int argc, char *argv[] )
{
   ...your code...
   return TCL_OK or TCL_ERROR;
}
```

Drawing-related Commands

`void Togl_PostRedisplay( struct Togl *togl )`

Signals that the widget should be redrawn. When Tk is next idle the user's C render callback will be invoked. This is typically called from within a Togl sub-command which was registered with Togl_CreateCommand().

`void Togl_SwapBuffers( struct Togl *togl )`

Swaps the front and back color buffers for a double-buffered widget. glFlush() is executed if the window is single-buffered. This is typically called in the rendering function which was registered with Togl_DisplayFunc().

Query Functions

`char *Togl_Ident( struct Togl *togl )`

Returns a pointer to the identification string associated with an Togl widget or NULL if there's no identifier string.

`int Togl_Width( struct Togl *togl )`

Returns the width of the given Togl widget. Typically called in the function registered with Togl_ReshapeFunc().

`int Togl_Height( struct Togl *togl )`

Returns the height of the given Togl widget. Typically called in the function registered with Togl_ReshapeFunc().

`Tcl_Interp *Togl_Interp( struct Togl *togl )`

Returns the Tcl interpreter associated with the given Togl widget.

Color Index Mode Functions

These functions are only used for color index mode.

`unsigned long Togl_AllocColor( struct Togl *togl, float red, float green, float blue )`

Allocate a color from a read-only colormap. Given a color specified by red, green, and blue return a colormap index (aka pixel value) whose entry most closely matches the red, green, blue color. Red, green, and blue are values in [0,1]. This function is only used in color index mode when the

-privatecmap option is false.

`void Togl_FreeColor( struct Togl *togl, unsigned long index )`

Free a color in a read-only colormap. Index is a value which was returned by the Togl_AllocColor() function. This function is only used in color index mode when the -privatecmap option is false.

`void Togl_SetColor( struct Togl *togl, int index, float red, float green, float blue )`

Load the colormap entry specified by index with the given red, green and blue values. Red, green, and blue are values in [0,1]. This function is only used in color index mode when the -privatecmap option is true.

Font Functions

`GLuint Togl_LoadBitmapFont( struct Togl *togl, const char *fontname )`

Load the named font as a set of glBitmap display lists. *fontname* may be one of

- `TOGL_BITMAP_8_BY_13`
- `TOGL_BITMAP_9_BY_15`
- `TOGL_BITMAP_TIMES_ROMAN_10`
- `TOGL_BITMAP_TIMES_ROMAN_24`
- `TOGL_BITMAP_HELVETICA_10`
- `TOGL_BITMAP_HELVETICA_12`
- `TOGL_BITMAP_HELVETICA_18`
- or any X11 font name

Zero is returned if this function fails.

After Togl_LoadBitmapFont() has been called, returning *fontbase*, you can render a string *s* with:

```
glListBase( fontbase );
glCallLists( strlen(s), GL_BYTE, s );
```

`void Togl_UnloadBitmapFont( struct Togl *togl, GLuint fontbase )`

Destroys the bitmap display lists created by by Togl_LoadBitmapFont().

Client Data Functions

`void Togl_SetClientData( struct Togl *togl, ClientData clientData)`

clientData is a pointer to an arbitrary user data structure. Each Togl struct has such a pointer. This function set’s the Togl widget’s client data pointer.

`ClientData Togl_GetClientData( const struct Togl *togl )`

clientData is a pointer to an arbitrary user data structure. Each Togl struct has such a pointer. This function returns the Togl widget’s client data pointer.

Overlay Functions

These functions are modelled after GLUT’s overlay sub-API.

`void Togl_UseLayer( struct Togl *togl, int layer )`

Select the layer into which subsequent OpenGL rendering will be directed. *layer* may be either *TOGL_OVERLAY* or *TOGL_NORMAL*.

`void Togl_ShowOverlay( struct Togl *togl )`

Display the overlay planes, if any.

`void Togl_HideOverlay( struct Togl *togl )`

Hide the overlay planes, if any.

`void Togl_PostOverlayRedisplay( struct Togl *togl )`

Signal that the overlay planes should be redraw. When Tk is next idle the user’s C overlay display callback will be invoked. This is typically called from within a Togl sub-command which was registered with Togl_CreateCommand().

`void Togl_OverlayDisplayFunc( Togl_Callback *proc )`

Registers the C callback function which should be called to redraw the overlay planes. This is the function which will be called in response to Togl_PostOverlayRedisplay(). The callback must be of the form:

```
void RedrawOverlay( struct Togl *togl )
{
   ...your code...
}
```

Tcl Togl commands

These are the Togl commands one may call from a Tcl program.

`togl` *`pathName`* `[options]`

Creates a new togl widget with name *pathName* and an optional list of configuration options. Options include:

Option	Default	Comments
-width	400	Width of widget in pixels.
-height	400	Height of widget in pixels.
-ident	""	A user identification string ignored by togl. This can be useful in your C callback functions to determine which Togl widget is the caller.
-rgba	true	If true, use RGB(A) mode If false, use Color Index mode
-double	false	If false, request a single buffered window If true, request double buffered window
-depth	false	If true, request a depth buffer
-accum	false	If true, request an accumulation buffer

`-alpha`	`false`	`If true and -rgba is true, request an alpha channel`
`-stencil`	`false`	`If true, request a stencil buffer`

`-privatecmap`	`false`	`Only applicable in color index mode.`
		`If false, use a shared read-only colormap.`
		`If true, use a private read/write colormap.`

`-overlay`	`false`	`If true, request overlay planes.`
`-stereo`	`false`	`If true, request a stereo-capable window.`

pathName `configure`

Returns all configuration records for the named togl widget.

pathName `configure` *-option*

Returns configuration information for the specifed *option* which may be one of:

`-width`

Returns the width configuration of the widget in the form:

`-width width Width` *W w*

where *W* is the default width in pixels and *w* is the current width in pixels

`-height`

Returns the height configuration of the widget in the form:

`-height height Height` *H h*

where *H* is the default height in pixels and *h* is the current height in pixels

`-extensions`

Returns a list of OpenGL extensions available. For example: `GL_EXT_polygon_offset GL_EXT_vertex_array`

pathName `configure` *-option value*

Reconfigure an togl widget. *option* may be one of:

`-width`

Resize the widget to *value* pixels wide

`-height`

Resize the widget to *value* pixels high

pathName `render`

Causes the render callback function to be called for *pathName*.

pathName `swapbuffers`

Causes front/back buffers to be swapped if in double buffer mode.

pathName `makecurrent`

Make the widget specified by *pathName* the current one.

Demo programs

There are three demo programs:

- double - compares single vs double buffering with two Togl widgets
- texture - lets you play with texture mapping options
- index - demo of using color index mode

To compile the demos, edit the Makefile to suit your system, then type "make". The Makefile currently works with Linux. To run a demo just type "double" or "texture" or "index".

Reporting Bugs

If you find a bug in Togl please report it to both Ben and Brian. When reporting bugs please provide as much information as possible. Also it's very helpful to us if you can provide an example program which demonstrates the problem.

Version History

Version 1.0, March 1996

- Initial version

Version 1.1 (never officially released)

- Added Togl_LoadBitmapFont function
- Fixed a few bugs

Version 1.2, November 1996

- added swapbuffers and makecurrent Tcl commands
- More bug fixes
- Upgraded to suport Tcl 7.6 and Tk 4.2
- Added stereo and overlay plane support
- Added Togl_Get/SetClientData() functions
- Added Togl_DestroyFunc()

Future plans

- Port to Windows NT

Last edited on December 14, 1996 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL Toolkit Choices

Contents

1. Introduction

A 3-D graphics application has two important components: the graphics library and user interface toolkit. While choosing OpenGL as the graphics library may be an easy choice, the decision of which GUI toolkit to use is not.

A number of factors influence the toolkit selection:

- **Size, complexity and purpose of application**: a simple graphics demo will have different user interface requirements than a 3-D modeller, for example.
- **Target platform**: few toolkits work on more than one operating system or window system.
- **Free vs commercial application**: a commercial application may have more stringent GUI requirements than a free program.
- **Free vs commercial toolkit**: some toolkits are free, others aren’t.

This document presents a survey of toolkit options for the OpenGL application programmer. For each toolkit the following attributes are discussed:

- **Overview**: Basic information about the toolkit or interface.
- **OpenGL integration method**: How does the toolkit/interface work?
- **Appropriate uses**: When is this toolkit most appropriate to use?
- **Advantages**: What are the pros of this toolkit?
- **Disadvanges**: What are the cons of this toolkit?
- **References**: Where to find more information.

2. GLX and Xlib

Overview

GLX is the OpenGL extension to X. It provides the "glue" functions for integrating OpenGL with the X window system in C or C++. GLX is also the protocol which allows remote display of OpenGL on X suitable X servers. While Xlib is not a user interface toolkit, it is a means of integrating OpenGL into an X application.

OpenGL integration method

Functions are provided to select OpenGL-enhanced visuals, create rendering contexts, bind contexts to X windows, synchronize with X, swap color buffers, etc.

Appropriate uses

Any X-based application may use the GLX interface. Toolkits build on X such as Xt/Motif and GLUT are built on top of GLX and hide its details.

Advantages

- Low level: complete access to unique facilities of the hardware (stereo, overlay planes, multi-sampling, etc)
- It’s the standard low level X/OpenGL interface.

Disadvantages

- Low level: does not provide GUI elements such as menus and buttons
- limited to X-based (and usually Unix-based) systems
- requires considerable Xlib knowledge

References:

- Appendix D of the OpenGL Programming Guide from Addison-Wesley
- Introduction to OpenGL and X, Part 1: An Introduction by Mark Kilgard (http://www.sgi.com/Technology/openGL/mjk.intro/intro.html)
- Introduction to OpenGL and X, Part 2: Using OpenGL with Xlib by Mark Kilgard (http://www.sgi.com/Technology/openGL/mjk.xlib/xlib.html)

3. AGL, PGL and WGL (GLX-like) interfaces

Overview

There are OpenGL glue libraries for other window systems such as IBM's Presentation Manager (PGL), Macintosh (AGL), and Microsoft Windows (WGL for NT and '95). These interfaces are similar to GLX in functionality and API design. Function bindings are typically only availabe for C and C++.

OpenGL integration method

Again, functions are provided to select visual/pixel formats, create/bind rendering contexts, synchronize with the window system, swap color buffers, etc.

Appropriate uses

These low level interfaces are often needed for any OpenGL application on a PC or Mac since higher level toolkits don't encapsulate them. Caveat: There is an effort among OS/2 developers to write a PGL wrapper for PM.

Advantages

- provide access to all OpenGL/window system integration features (off-screen rendering, font handling, etc.)

Disadvantages

- requires knowledge of details specific to the window system

References

- IBM's OpenGL for OS/2 (http://www.austin.ibm.com/software/OpenGL/)
- OpenGL for OS/2 FAQ (http://www.utsi.com/~kgl/os2-opengl/faq.html)
- IBM's The OpenGL libraries for OS/2 (ftp://ftp.austin.ibm.com/pub/developer/os2/OpenGL/)
- OpenGL for Microsoft Windows '95 and NT (http://www.sgi.com/Technology/openGL/vendor/microsoft.html)
- OpenGL for the Macintosh from Conix Graphics (http://www.conix3d.com/)

4. Xt/Motif

Overview

Xt is the X Toolkit Instrinsics, a library built on Xlib designed to support user interface toolkits. Motif is a popular widget set built on Xt. Xt and Motif may be used with C/C++.

OpenGL integration method

The special GLwMDrawingArea widget supports OpenGL rendering. The IRIS ViewKit library provides a framework which offers an OpenGL widget as well.

Appropriate uses

Commercial, professional applications for the X environment.

Advantages

- Motif is standardized and full featured.
- Other widget sets are available: Athena, OPEN LOOK.

Disadvantages

- Xt/Motif is large and complicated
- Probably overkill for small applications
- Motif is not free

References

- OpenGL and X, Part 3: Integrating OpenGL with Motif by Mark Kilgard (http://www.sgi.com/Technology/openGL/mjk.motif/motif.html)
- *Programming OpenGL with the X Window System* by Mark Kilgard

5. GLUT

Overview

The GL Utility Toolkit, written by Mark Kilgard, is a free, portable toolkit which provides functions for creating windows, pop-up menus, event handling, drawing simple geometric primitives and much more.

GLUT will replace aux in the next edition of the OpenGL Programming Guide.

OpenGL integration method

GLUT is built on top of OpenGL and the underlying window system. It has a simple C/C++ API. Simply make GLUT calls to create windows and setup event handling then make OpenGL calls to draw your imagery.

Appropriate uses

- applications which don’t require a sophisticated GUI
- teaching, instruction, experimentation
- demos

Advantages

- free
- simple
- portable; operating system and window system independent. Available for Xlib, Windows ’95/NT, and OS/2.
- provides access to advanced input devices, stereo viewing, overlay planes, etc
- the GLUT source code provides excellent examples of programming advanced OpenGL and window system features.

Disadvantages

- doesn’t provide the user interface elements such as buttons and sliders needed for many applications

References

- GLUT 3.0 WWW page by Mark Kilgard (http://reality.sgi.com/employees/mjk_asd/glut3/glut3.html)

- *Programming OpenGL with the X Window System* by Mark Kilgard
- GLUT for Windows '95/NT by Nate Robins. (http://www.cs.utah.edu/~narobins/opengl.html)

6. aux/tk

Overview

aux and tk (not to be confused with Tcl/Tk) are simple OpenGL toolkits developed by SGI for the OpenGL Programming Guide (first edition) and for OpenGL demos. They are very similar to each other, often only different in function prefixes. The major features of aux/tk are window creation and event handling.

These toolkits are very limited in functionality and are not intended for any sort of application development. The GLUT toolkit does everything that aux/tk does plus much more and should be prefered over aux/tk in any situation.

OpenGL integration method

tk is built on top of Xlib/GLX. aux has been implemented on several window systems and in the case of X, implemented on top of tk.

Appropriate uses

Small demo programs and examples from the OpenGL programming guide. ***GLUT is a much better choice***.

Advantages

- small and simple
- aux is available on several operating systems

Disadvantages

- very limited funtionality
- several different API implementations of aux exist
- has no features which GLUT doesn't also provide

References

- OpenGL Programming Guide (first edition) from Addison-Wesley
- A README documentat is included with most implementations

7. Tcl/Tk

Overview

Tcl is a popular, free, interpreted "script" language invented by John Ousterhout. Tk is a graphics user interface toolkit for Tcl. Tcl/Tk handles user interface and event processing while C is used for computation and rendering. Originally designed for X, both are now available for Windows and Macintosh systems.

OpenGL integration method

1. A number of free OpenGL/Tk widgets are available which allow one to create OpenGL "canvases" from Tk. Rendering is done from C code calling the OpenGL API.
2. Another approach taken by several people is to provide Tcl/Tk wrappers for all OpenGL function so an application may be written with Tcl/Tk alone.

Appropriate uses

- Good for demos through large applications.
- Good for experimentation, learning and small programs.

Advantages

- Free
- Easy to learn
- Full featured GUI
- Quick prototyping
- Tcl/Tk applications are portable across Unix, Windows, and Mac.
- hides low level details of GUI/OpenGL integration

Disadvantages

- OpenGL/Tk support not available on Windows or Macintosh at this time.
- Since Tcl is interpreted it may not meet the demands of high performance applications.

References

- TIGER by Ekkehard Beier of the Technical University of Ilmenau, Germany (ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/TIGER1.0)
- TkOGL - a Tk OpenGL widget by Claudio Esperanca of Brazil (http://aquarius.lcg.ufrj.br/~esperanc/tkogl.html)
- OGLTK by Benjamin Bederson of the University of New Mexico (http://www.cs.unm.edu/~bederson/ogl.html)
- Togl (http://www.ssec.wisc.edu/~brianp/Togl.html)

8. XForms

Overview

XForms is a free X-based GUI toolkit written by T. C. Zhao based on the original Forms library by Mark Overmars.

OpenGL Integration method

A special OpenGL canvas can be created for OpenGL rendering.

Appropriate use

Small to large applications and demos.

Advantages

- Free

- Easy to use
- Available for most Unix/X workstations

Disadvantages

- OpenGL integration is minimal, one would have to modify the OpenGL canvas code if you need anything more than double buffered RGB rendering.
- May not be as powerful as Motif

References

- XForms home page (http://bragg.phys.uwm.edu/~zhao/xforms_home.html)

9. Inventor

Overview

A high-level 3-D graphics toolkit for C and C++ built on top of OpenGL. Inventor provides object-oriented database construction, rendering, interaction, file I/O, etc.

OpenGL Integration method

Inventor provides library functions for creating OpenGL- rendering windows. However, lower level window system integration (Xt) is also allowed.

Appropriate uses

Interactive, "object"-oriented graphical applications, possibly in conjuction with a GUI toolkit such as Motif.

Advantages

- provides powerful high-level graphics structures and interaction
- object/model file I/O
- now available on many platforms from vendors such as Template Graphics Software and Portable Graphics

Disadvantages

- not free
- doesn’t in itself provide all the GUI elements needed for full applications

References

- Open Inventor home page at SGI (http://www.sgi.com/Technology/Inventor/)
- Open Inventor Products from Template Graphics Software, Inc. (http://www.sd.tgs.com/)
- The Visual 3Space Browser Control from Template Graphics Software is a 3D/VRML OLD Custom Control for Win32, allowing VRML/Inventor integration into OCX container applications. (http://www.tgs.com/Products/v3space.htm)

10. SGI Performer

Overview

A high-level graphics library built on top of OpenGL designed for high-performance realtime applications such as virtual reality, visual simulation, entertainment. C/C++ language bindings.

OpenGL integration method

Performer 2.0 is built on OpenGL. It also privides a simple set of window management routines (pfWindow).

Appropriate uses

Applications which require maximum interactive performance.

Advantages

- provides high-level graphics structures, interaction, multi-CPU support, scene (LOD) management
- object description file I/O

Disadvantages

- Proprietary
- Targeted to high-end hardware
- doesn’t provide GUI elements

References

- Performer information from SGI (http://www.sgi.com/Technology/Performer/)

11. OpenGL Optimizer

Overview

The OpenGL Optimizer is a toolkit built on top of OpenGL. It’s designed for CAD/CAE and visualization applications which deal with large, complex models. The OpenGL Optimizer offers advanced culling, occlusion testing and NURBS tesselation features.

OpenGL integration method

The OpenGL Optimizer is a C++ toolkit layered upon OpenGL.

Appropriate uses

Applications which deal with large, complicated object models can use the OpenGL Optimizer to simplify their models for faster interactive rendering.

Advantages

- provides performance advantages over straight OpenGL rendering
- adopted as a standard among CAD/CAE vendors/developers

Disadvantages

- The OpenGL Optimizer is a very new product and may not be widely available at this time.

References

- The OpenGL Optimizer home page (http://www.sgi.com/Technology/OpenGL/optimizer/).

12. OpenGL++ / OpenGL Scene Graph

Overview

At the time of this writing, OpenGL++ (aka the OpenGL Scene Graph) is under development by the OpenGL ARB. The purpose of OpenGL++ is to provide a higher-level toolkit for OpenGL which manages a scene graph with facilities for interaction, compilation, culling, multi-processing, sorting, etc.

OpenGL integration method

OpenGL++ will likely have C++ and Java APIs built upon OpenGL (or possibly other low-level 3-D APIs).

Appropriate uses

OpenGL++, like Open Inventor or Performer, will be appropriate for applications which require higher-level functionality than what OpenGL provides.

Advantages

- Will relieve the application programmer of low-level OpenGL concerns.
- Will provide high-level 3-D features such as scene-graph management, interaction, culling, LOD management, etc.

Disadvantages

- May not be available for some time.

References

- OpenGL ARB meeing notes from February 17-19, 1997 (http://www.sgi.com/Technology/openGL/arb-feb.html)

13. Others

Python

While still a work in progress there is some information available from Brown University regarding OpenGL/Python integration. (http://maigret.cog.brown.edu:80/python/opengl/)

Java

There is an unofficial port of OpenGL to Java. (ftp://cgl.uwaterloo.ca/pub/software/meta/OpenGL4java.html)

MET++

MET++ is an extension to the ET++ Application Framework, an object-oriented class library that integrates interface building blocks, basic data structures, input/output, printing, and high-level application framework components. The MET++ extensions include PEX, GL, and OpenGL support. (http://www.ifi.unizh.ch/groups/mml/projects/met++/met++.html)

On a related note, Steven Baum maintains a nice list of free GUI development systems (http://www-ocean.tamu.edu/~baum/graphics-GUI.html) and graphics/visualization software (http://www-ocean.tamu.edu/~baum/ocean_graphics.html) at Texas A&M University.

Last edited on April 14, 1997 by Brian Paul.

TR - OpenGL Tile Rendering Library

Version 1.0

Copyright (C) 1997 Brian Paul

Introduction

The TR (Tile Rendering) library is an OpenGL utility library for doing tiled rendering. Tiled rendering is a technique for generating large images in pieces (tiles).

TR is memory efficient; arbitrarily large image files may be generated without allocating a full-sized image buffer in main memory.

The TR library is copyrighted by Brian Paul. See the LICENSE file for details.

You may download TR 1.0 by SHIFT-clicking on one of the following:

- tr-1.0.tar.gz (10Kbytes)
- tr-1.0.zip (10Kbytes)

Prerequisites

TR works with any version of OpenGL or Mesa. No extensions are necessary and there are no dependencies on GLX, WGL or any other window system interface.

TR is written in ANSI C and may be used from C or C++.

The TR demo programs require Mark Kilgard’s GLUT.

Users should have intermediate experience with OpenGL.

Example

The following image is divided into four rows and three columns of tiles. Note that the image does not have to be divided into equally sized tiles. The TR library handles the situation in which the top row and right column are a fraction of the full tile size.

Also note that the tiles do not have to be square.

This is a small example. In reality, one may use tiles of 512 by 512 pixels and the final image may be 4000 by 3000 pixels (or larger!).

Using the Library

Ordinarily, OpenGL can’t render arbitrarily large images. The maximum viewport size is typically 2K pixels or less and the window system usually imposes a maximum color buffer size.

To overcome this limitation we can render large images in pieces (tiles).

To render each tile we must carefully set the viewport and projection matrix and render the entire scene. The TR library hides the details involved in doing this. Also, TR can either automatically assemble the final image or allow the client to write the image, row by row, to a file.

The basic steps in using TR are as follows:

1. Determine where you’ll render the tiles

Tiles may be rendered either in a window (front or back buffer) or in an off-screen buffer. The choice depends on your application. It doesn’t matter to the TR library since TR just retrieves image tiles with `glReadPixels`. Just be sure `glDrawBuffer` and `glReadBuffer` are set to the same buffer.

2. Determine the destination for the final image

The final, large image may either be automatically assembed in main memory by TR or you may elect to process tiles yourself, perhaps writing them to an image file.

3. Centralize your drawing code

It should be a simple matter to completely re-render your OpenGL scene. Ideally, inside the tile rendering loop you should be able to make one function call which clears the color (and depth, etc) buffer(s) and draws your scene. If you’re using a double buffered window you should not call `SwapBuffers` since `glReadBuffer`, by default, specifies the back buffer.

4. Allocate a TR context

Every TR function takes a `TRcontext` pointer. A TR context encapsulates the state of the library and allows one to have several TR contexts simultaneously. TR contexts are allocated with `trNew`.

5. Set the image and tile sizes

Call `trImageSize` to set the final image size, in pixels. Optionally, call `trTileSize` to set the tile size. Currently, the default tile size is 256 by 256 pixels. Generally, larger tiles are better since fewer tiles (and rendering passes) will be needed.

6. Specify an image or tile buffer

If you want TR to automatically assemble the final image you must call `trImageBuffer` to specify an image buffer, format, and pixel type. The format and type parameters directly correspond to those used by `glReadPixels`.

Otherwise, if you want to process image tiles yourself you must call `trTileBuffer` to specify a tile buffer, format, and pixel type. The `trEndTile` function will copy the tile image into your buffer. You may then use or write the tile to a file, for example.

7. Optional: set tile rendering order

Since OpenGL specifies that image data are stored in bottom-to-top order TR follows the same model. However, when incrementally writing tiles to a file we usually want to do it in top-to-bottom order since that's the order used by most file formats.

The `trRowOrder` function allows you to specify that tiles are to be rendering in `TR_TOP_TO_BOTTOM` order or `TR_BOTTOM_TO_TOP` order. The later is the default.

8. Specify the projection

The projection matrix must be carefully controlled by TR in order to produce a final image which has no cracks or edge artifacts.

OpenGL programs typically call `glFrustum`, `glOrtho` or `gluPerspective` to setup the projection matrix. There are three corresponding functions in the TR library. One of them *must* be called to specify the projection to use. The arguments to the TR projection functions exactly match the arguments to the corresponding OpenGL functions.

9. Tile rendering loop

After the tile size and image size are specified the TR library computes how many tiles will be needed to produce the final image.

The tiles are rendered inside a loop similar to this:

```
int more = 1;
while (more)
{
        trBeginTile(tr);
        DrawScene();
        more = trEndTile(tr);
}
```

This should be self-explanatory. Simply call `trBeginTile`, render your entire scene, and call `trEndTile` inside a loop until `trEndTile` returns zero.

10. Query functions

The `trGet` function can be called to query a number of TR state variables such as the number of rows and columns of tiles, tile size, image size, currently rendered tile, etc. See the detailed description of `trGet` below.

11. glRasterPos problem

The `glRasterPos` function is troublesome. The problem is that the current raster position is invalidated if `glRasterPos` results in a coordinate outside of the window. Subsequent `glDrawPixels` and `glBitmap` functions are ignored. This will frequently happen during tiled rendering resulting in flawed images.

TR includes a substitute function: `trRasterPos3f` which doesn't have this problem. Basically, replace calls to `glRasterPos` with `trRasterPos`. See the included demo programs for example usage.

12. Compilation

Include the *tr.h* header file in your client code.

Compile and link with the *tr.c* library source file. There is no need to compile TR as a separate library file.

API Functions

Creating and Destroying Contexts

`TRcontext *trNew(void)`
Return a pointer to a new TR context and initialize it. Returns NULL if out of memory.

`void trDelete(TRcontext *tr)`
Deallocate a TR context.

Image and Tile Setup Functions

`void trTileSize(TRcontext *tr, GLint width, GLint height)`
Specifies size of tiles to generate. This is generally the size of your window or off-screen image buffer.

`void trImageSize(TRcontext *tr, GLint width, GLint height)`
Specifies size of final image to generate.

`void trTileBuffer(TRcontext *tr, GLenum format, GLenum type, GLvoid *image);`
This is an optional function. After a tile is rendered (after `trEnd`) it will be copied into the buffer

specified by this function.
`image` must point to a buffer large enough to hold an image equal to the tile size specified by `trTileSize`.
`format` and `type` are interpreted in the same way as `glReadPixels`.

`void trImageBuffer(TRcontext *tr, GLenum format, GLenum type, GLvoid *image);`

This is an optional function. This specifies a buffer into which the final image is assembled. As tiles are generated they will automatically be copied into this buffer. The image will be complete after the last tile has been rendered.
`image` must point to a buffer large enough to hold an image equal to the size specified by `trImageSize`.
`format` and `type` are interpreted in the same way as `glReadPixels`.

Note: `trImageBuffer` and `trTileBuffer` are the means by which image data is obtained from the TR library. You must call one (or both) of these functions in order to get output from TR.

`void trRowOrder(TRcontext *tr, TRenum order)`

Specifies the order in which tiles are generated.
`order` may take one of two values:

- `TR_BOTTOM_TO_TOP` - render tiles in bottom to top order (the default)
- `TR_TOP_TO_BOTTOM` - render tiles in top to bottom order

Projection Setup Functions

`void trOrtho(TRcontext *tr, GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`

Specify an orthographic projection as with `glOrtho`.
Must be called before rendering first tile.

`void trFrustum(TRcontext *tr, GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`

Specify a perspective projection as with `glFrustum`.
Must be called before rendering first tile.

`void trPerspective(TRcontext *tr, GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar );`

Specify a perspective projection as with `gluPerspective`.
Must be called before rendering first tile.

Tile Rendering Functions

`trBeginTile(TRcontext *tr)`

Begin rendering a tile.

`int trEndTile(TRcontext *tr)`

End rendering a tile.
Return 0 if finished rendering image.
Return 1 if more tiles remain to be rendered.

The `trBeginTile` and `trEndTile` functions are meant to be used in a loop like this:

```
int more = 1;
while (more)
{
        trBeginTile(tr);
        DrawScene();
        more = trEndTile(tr);
}
```

`DrawScene` is a function which renders your OpenGL scene. It should include `glClear` but not `SwapBuffers`.

Miscellaneous Functions

`GLint trGet(TRcontext *tr, TRenum param)`

Query TR state. `param` may be one of the following:

- `TR_TILE_WIDTH` - returns tile buffer width
- `TR_TILE_HEIGHT` - returns tile buffer height
- `TR_IMAGE_WIDTH` - returns image buffer width
- `TR_IMAGE_HEIGHT` - returns image buffer height
- `TR_ROW_ORDER` - returns `TR_TOP_TO_BOTTOM` or `TR_BOTTOM_TO_TOP`
- `TR_ROWS` - returns number of rows of tiles in image
- `TR_COLUMNS` - returns number of columns of tiles in image
- `TR_CURRENT_ROW` - returns current tile row. The bottom row is row zero.
- `TR_CURRENT_COLUMN` - returns current tile column The left column is column zero.
- `TR_CURRENT_TILE_WIDTH` - returns width of current tile
- `TR_CURRENT_TILE_HEIGHT` - returns height of current tile

Note the difference between `TR_TILE_WIDTH/HEIGHT` and `TR_`**`CURRENT`**`_TILE_WIDTH/HEIGHT`. The former is the size of the tile buffer. The later is the size of the *current* tile which can be less than or equal to the `TR_TILE_WIDTH/HEIGHT`. Unless the final image size is an exact multiple of the tile size, the last tile in each row and column will be smaller than `TR_TILE_WIDTH/HEIGHT`.

`void trRasterPos3f(TRcontext *tr, GLfloat x, GLfloat y, GLfloat z)`

This function is a replacement for `glRasterPos3f`. The problem with the OpenGL RasterPos functions is that if the resulting window coordinate is outside the view frustum then the raster position is invalidated and `glBitmap` becomes a no-op.

This function avoids that problem.

You should replace calls to `glRasterPos` with this function. Otherwise, `glRasterPos/glBitmap` sequences won’t work out correctly during tiled rendering.

Unfortunatley, `trRasterPos3f` can’t be saved in a display list.

Demonstration Programs

The TR distribution includes two GLUT-based demo programs:

- trdemo1 - renders a window-size image in tiles
- trdemo2 - produces a large PPM file incrementally

You'll probably have to edit the Makefile for your computer. Compiling the demos is very simple though since they only require OpenGL and GLUT.

Contributors

- Robin Syllwasschy - provided much helpful feedback for the initial version of TR.

Version History

Version 1.0 - April 1997

- Initial version

Last edited on April 27, 1997 by Brian Paul.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

Graphics Library Transition Notes

Contents

1. Introduction

OpenGL is now the predominate 3-D graphics library and there are reasons to port many existing applications from older libraries:

- To take advantage of new graphics hardware
- To keep up with evolving operating systems
- To broaden the range of system supported

Porting graphics applications can take a lot of effort; there are no silver bullets. This document outlines several techniques and hints.

2. PEX to OpenGL

PEX is a 3-D graphics extension to the X Window System. The API is similar to Xlib in that there are many pointers, structures and complicated function calls. OpenGL by comparison is much cleaner and simpler. Feature-wise, PEX offers much of the functionality of OpenGL 1.0.

Here are the highlights of PEX vs OpenGL and porting:

- PEX is more of a protocol specification than API specification. That is, there are several interfaces to PEX functionality. OpenGL on the other hand, is defined in terms of an API and not a protocol.

- Since PEX relies on the same Xlib window management and event handling code as OpenGL for X (GLX), much of the user interface code may be quite portable.

- PEX's data structures for describing geometry are of coarse granularity while OpenGL geometry is described in in fine granularity. That is, the data structures for PEX can be easily rendered by OpenGL since OpenGL specifies primitives a vertex at a time rather than as large arrays or structures.

  One may be able to continue using PEX-style data structures in your application and render them using OpenGL commands.

- PEX's notion of attribute "bundles" can be replaced with OpenGL display lists.

- The problem of dealing with PEX subsetting largly disappears with since the OpenGL specification mandates full implementation.

- A PEX application which uses multiple rendering contexts may be especially difficult to port to OpenGL since most PEX API functions explicitly specify the context while in OpenGL the context is implicit. Context switching in OpenGL may be considerably more expensive than it is with PEX.

- PEX has a lot of support for fonts and text drawing which may be difficult to translate to OpenGL.

- PEX has several primitive such as quadrilateral meshes which aren't directly offered by OpenGL but can be implemented without too much trouble.

- PEX supports editable display lists while OpenGL doesn't. Nested OpenGL display lists may be a suitable work around.

- Though PEX and GLX both are built on Xlib, visual selection and window creation code will have to be reimplemented for OpenGL.

3. IRIS GL to OpenGL

Since OpenGL's roots are in IRIS GL one may expect porting from IRIS GL to OpenGL to be easy. Conceptually, IRIS GL and OpenGL are very similar, but in practice porting is not an easy job. Many of the IRIS GL function calls directly map to OpenGL. On the other hand, many features such as lighting and texturing are implemented quite differently.

SGI's OpenGL Porting Guide is a good place to begin a porting project. The *toogl* utility partially automates the conversion of programs from IRIS GL to OpenGL. It is included with the IRIX IDO option.

Below are the highlights of the similarities and differences in OpenGL and IRIS GL.

3.1 Similarities

Basic Rendering

OpenGL and IRIS GL are very similar in how they specify geometric primitives; both use the begin/vertex/color/normal/end paradigm. In many cases, IRIS GL drawing commands directly map to OpenGL equivalents.

Transformation and viewing

OpenGL and IRIS GL use similar functions for coordinate transformation and viewing. Both have modelview and projection matrices which can be built up from simple transformation calls (scale, translate, rotate). Be aware that OpenGL’s projection functions such as glOrtho() and glFrustum() are multiplied onto the projection matrix rather than replace the projection matrix as IRIS GL’s ortho() and window() do. You should first load an identity matrix.

Immediate mode rendering and display lists

Immediate mode rendering and display list are supported by both libraries. OpenGL, however, does not support editing display lists as IRIS GL does. Nested/hierarchal OpenGL display lists may replace editing.

Picking and feedback

Picking (selection) works similar in OpenGL and IRIS GL; both use a name stack. Feedback in OpenGL is nicer than IRIS GL because OpenGL feedback is identical on all implementations, while IRIS GL implemented it differently on some systems.

Depth testing, blending, stenciling, accumulation

Depth (Z) buffering, alpha blending, stencil buffers and accumulation buffers are all implemented similarly in OpenGL and IRIS GL. In many cases there is a direct mapping of functions between the libraries.

2.2 Differences

OpenGL contains no window system functions like IRIS GL

If your IRIS GL program is a "mixed model" program, using IRIS GL for rendering but X for window/event handling, then most of your even processing code should work fine with OpenGL.

If your IRIS GL program makes heavy use of IRIS GL’s input devices, window management, pop-up menus, etc porting will be more difficult. One possibility is to use GLUT. GLUT provides much of the IRIS GL functionality which OpenGL lacks.

Lighting

While OpenGL and IRIS GL lighting are functionally similar, the implementations are quite different. IRIS GL’s lmdef() and lmbind() functions are replaced by separate functions for setting light, material, and lighting model parameters in OpenGL. The tables of IRIS GL lighting parameters one might be using can be replaced by display lists in OpenGL.

Texture mapping

IRIS GL supports defining tables of textures, one of which can be bound at a time with texbind(). OpenGL only directly supports one texture map definition at a time. However, the texture object extension or display lists can be used to simulate the IRIS GL texture system.

No subsetting of OpenGL

One especially nice difference between IRIS GL and OpenGL is the fact that OpenGL does not allow subsetting. That is, the entire functionality of OpenGL will always be implemented. IRIS GL unfortunately implemented different features on different systems.

These points only describe the high-level differences in the graphics libraries. As mentioned above, the OpenGL Porting Guide goes into much more detail.

Last edited on April 13, 1997 by Brian Paul.

PUBLISHED IN THE
JANUARY/FEBRUARY 1994 ISSUE OF *The X Journal*.

OPENGL$^{\mathrm{TM}}$ AND X, PART 2: USING OPENGL WITH XLIB

Mark J. Kilgard [*]
Silicon Graphics Inc.

$Revision: 1.22$

May 7, 1997

Abstract

This is the second article in a three-part series about using the OpenGL$^{\mathrm{TM}}$ graphics system and the X Window System. A moderately complex OpenGL program for X is presented. Depth buffering, back-face culling, lighting, display list modeling, polygon tessellation, double buffering, and shading are all demonstrated. The program adheres to proper X conventions for colormap sharing, window manager communication, command line argument processing, and event processing. After the example, advanced X and OpenGL issues are discussed including minimizing colormap flashing, handling overlays, using fonts, and performing animation. The last article in this series discusses integrating OpenGL with the Motif toolkit.

1 Introduction

In the first article in this series, the OpenGL$^{\mathrm{TM}}$ graphics system was introduced. Along with an explanation of the system's functionality, a simple OpenGL X program was presented and OpenGL was compared to the X Consortium's PEX extension. In this article, a more involved example of programming OpenGL with X is presented. The example is intended to demonstrate both sophisticated OpenGL functionality and proper integration of OpenGL with the X Window System.

This article is intended to answer questions from two classes of programmers: first, the X programmer wanting to see OpenGL used in a program of substance; second, the OpenGL or IRIS GL programmer likely to be unfamiliar with the more mundane window system setup necessary when using the X Window System at the Xlib layer.

The example program called `glxdino` renders a 3D dinosaur model using OpenGL. Hidden surfaces are removed using depth buffering. Back-face culling improves rendering performance by not rendering back-facing polygons. Hierarchical modeling is used to construct the dinosaur and render it via OpenGL display lists. The OpenGL Utility Library (GLU) polygon tessellation routines divide complex polygons into simpler polygons renderable by OpenGL. Sophisticated lighting lends realism to the dinosaur. If available, double buffering smoothes animation.

The program integrates well with the X Window System. The program accepts some of the standard X command line options: `-display`, `-geometry`, and `-iconic`. The user can rotate the model using mouse motion. Top-level window properties specified by the Inter-Client Communication Convention Manual (ICCCM) are properly set up to communicate with the window manager. Colormap sharing is done via ICCCM conventions. And the proper way of communicating to the window manager a desire for a constant aspect ratio is demonstrated.

A walk through of the `glxdino` source code is presented in Section 2. While `glxdino` tries to demonstrate a good number of OpenGL features and many of the issues concerning how X and OpenGL integrate, it is only an example. Section 3 explores more of the issues encountered when writing an advanced OpenGL program using Xlib. The third and last article in this series discusses how to integrate OpenGL with the Motif toolkit.

2 Example Walk Through

The source code for `glxdino` can be found in Appendix A. I will refer to the code repeatedly throughout this section. Figure 1 shows a screen snapshot of `glxdino`.

2.1 Initialization

The program's initialization proceeds through the following steps:

1. Process the standard X command line options.
2. Open the connection to the X server.

---

[*] Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

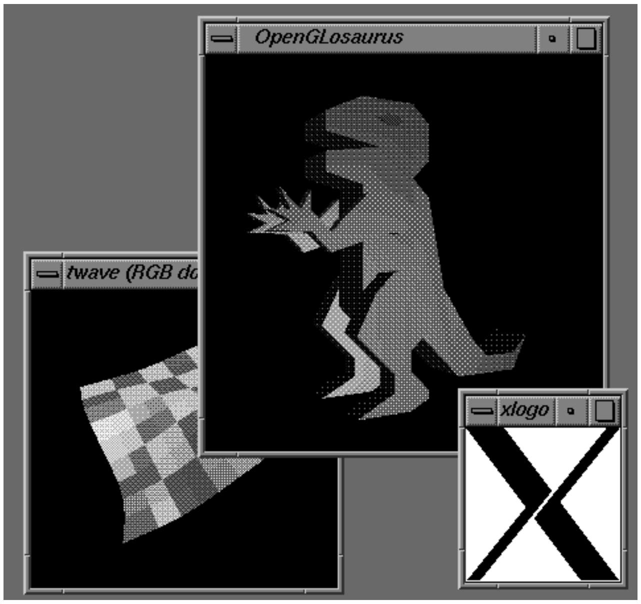


Figure 1: Screen snapshot of `glxdino`.

3. Determine if OpenGL's GLX extension is supported.
4. Find the appropriate X visual and colormap.
5. Create an OpenGL rendering context.
6. Create an X window with the selected visual and properly specify the right ICCCM properties for the window manager to use.
7. Bind the rendering context to the window.
8. Make the display list hierarchy for the dinosaur model.
9. Configure OpenGL rendering state.
10. Map the window.
11. Begin dispatching X events.

Comments in the code correspond to these enumerated steps.

In the program's `main` routine, the first task is to process the supported command line arguments. Users of the X Window System should be familiar with `-display` which specifies the X server to use, `-geometry` which specifies the initial size and location of the program's main window, and `-iconic` which requests the window be initially iconified. Programmers used to the IRIS GL (the predecessor to OpenGL) may not be familiar with these options. While nothing requires an X program to accept standard X options, most do as a matter of consistency and convenience. Most X toolkits automatically understand the standard set of X options

The `-keepaspect` option is not a standard X command line option. When specified, it requests that the window manager ensure that the ratio between the initial width and height of the window be maintained. Often for 3D programs, the programmer would like a constant aspect ratio for their rendering window. In IRIS GL, a call named `keepaspect` is available. Maintaining the aspect ratio of a window is something for the window system to do so there is no call analogous to IRIS GL's `keepaspect` in OpenGL. Remember that the core OpenGL Application Programmer Interface (API) attempts to be window system independent. IRIS GL programmers used to the IRIS GL interface will need to become aware of X functionality to do things that used to be done with IRIS GL calls.

Normally `glxdino` tries to use a double buffered window but will use a single buffered window if a double buffered visual is not available. When the `-single` option is present, the program will look only for a single buffered visual. On many machines with hardware double buffering support, color resolution can be traded for double buffering to achieve smooth animation. For example, a machine with 24 bits of color resolution could support 12 bits of color resolution for double buffered mode. Half the image bit-planes would be for the front buffer and half for the back buffer.

Next, a connection to the X server is established using `XOpenDisplay`. Since `glxdino` requires OpenGL's GLX extension, the program checks that the extension exists using `glXQueryExtension`. The routine indicates if the GLX extension is supported or not. As is convention for X routines that query extensions, the routine can also return the *base error code* and *base event code* for the GLX extension. The current version of GLX supports no extension events (but does define eight protocol errors). Most OpenGL programs will need neither of these numbers. You can pass in `NULL` as `glxdino` does to indicate you do not need the event or error base.

OpenGL is designed for future extensibility. The `glXQueryVersion` routine returns the major and minor version of the OpenGL implementation. Currently, the major version is 1 and the minor version is 0. `glxdino` does not use `glXQueryVersion` but it may be useful for programs in the future.

2.1.1 Choosing a Visual and Colormap

The GLX extension overloads X visuals to denote supported frame buffer configurations. Before you create an OpenGL window, you should select a visual which supports the frame buffer features you intend to use. GLX guarantees at least two visual will be supported. An RGBA mode visual with a depth buffer, stencil buffer, and accumulation buffer must be supported. Second, a color index mode visual with a depth buffer and stencil buffer must be available. More and less capable visuals are likely to also be supported depending on the implementation.

To make it easy to select a visual, `glXChooseVisual` takes a list of the capabilities you are requesting and returns an `XVisualInfo*` for a visual meeting your requirements. `NULL` is returned if a visual meeting your needs is not available. To ensure your application will run with any OpenGL GLX server, your program should be written to support the base line required GLX visuals. Also you should only ask for the minimum set of frame buffer capabilities you require. For example, if your program never uses a stencil buffer, you will possibly waste resources if you request one anyway.

Since `glxdino` rotates the dinosaur in response to user input, the program will run better if double buffering is available. Double buffering allows a scene to be rendered out of view and then displayed nearly instantly to eliminate the visual artifacts associated with watching a 3D scene render. Double buffering helps create the illusion of smooth animation. Since double buffering support is not required for OpenGL implementations, `glxdino` resorts to single buffering if no double buffer visuals are available. The program's `configuration` integer array tells what capabilities `glXChooseVisual` should look for. Notice how if a double buffer visual is not found, another attempt is made which does not request double buffering by starting after the `GLX_DOUBLBUFFER` token. And when the `-single` option is specified, the code only looks for a singled buffered visual.

`glxdino` does require a depth buffer (of at least 16 bits of accuracy) and uses the RGBA color model. The RGBA base line visual must support at least a 16 bit depth buffer so `glxdino` should always find a usable visual.

You should not assume the visual you need is the default visual. Using a non-default visual means windows created using the visual will require a colormap matching the visual. Since the window we are interested in uses OpenGL's RGBA color model, we want a colormap configured for using RGB. The ICCCM establishes a means for sharing RGB colormaps between clients. `XmuLookupStandardColormap` is used to set up a colormap for the specified visual. The routine reads the ICCCM `RGB_DEFAULT_MAP` property on the X server's root window. If the property does not exist or does not have an entry for the specified visual, a new RGB colormap is created for the visual and the property is updated (creating it if necessary). Once the colormap has been created, `XGetRGBColormaps` finds the newly created colormap. The work for finding a colormap is done by the `getColormap` routine.

If a standard colormap cannot be allocated, `glxdino` will create an unshared colormap. For some servers, it is possible (though unlikely) a `DirectColor` visual might be returned (though the GLX specification requires a `TrueColor` visual be returned in precedence to a `DirectColor` visual if possible). To shorten the example code by only handling the most likely case, the code bails if a `DirectColor` visual is encountered. A more portable (and longer) program would be capable of initializing an RGB `DirectColor` colormap.

2.1.2 Creating a Rendering Context

Once a suitable visual and colormap are found, the program can create an OpenGL rendering context using `glXCreateContext`. (The same context can be used for different windows with the same visual.)

The last parameter allows the program to request a direct rendering context if the program is connected to a local X server. An OpenGL implementation is not required to support direct rendering, but if it does, faster rendering is possible since OpenGL will render directly to the graphics hardware. Direct rendered OpenGL requests do not have to be sent to the X server. Even when on the local machine, you may not want direct rendering in some cases. For example, if you want to render to X pixmaps, you must render through the X server.

GLX rendering contexts support sharing of display lists among one another. To this end, the third parameter to `glXCreateContext` is another already created GLX rendering context. `NULL` can be specified to create an initial rendering context. If an already existent rendering context is specified, the display list indexes and definitions are shared by the two rendering contexts. The sharing is transitive so a share group can be formed between a whole set of rendering contexts.

To share, all the rendering contexts must exist in the *same* address space. This means direct renderers cannot share display lists with renderers rendering through the X server. Likewise direct renderers in separate programs cannot share display lists. Sharing display lists between renderers can help to minimize the memory requirements of applications that need the same display lists.

2.1.3 Setting Up a Window

Because OpenGL uses visuals to distinguish various frame buffer capabilities, programmers using OpenGL need to be aware of the required steps to create a window with a non-default visual. As mentioned earlier a colormap created for the visual is necessary. But the most irksome thing to remember about creating a window with a non-default visual is that the border pixel value *must* be specified if the window's visual is not the same as its parent's visual. Otherwise a `BadMatch` is generated.

Before actually creating the window, the argument to the `-geometry` option should be parsed using `XParseGeometry` to obtain the user's requested size and location. The size will be needed when we create the window. Both the size and location are needed to set up the ICCCM size hints for the window manager. A fixed aspect ratio is also requested by setting up the right size hints if the `-keepaspect` option is specified.

Once the window is created, `XSetStandardProperties` sets up the various standard ICCCM properties including size hints, icon name, and window name. Then the ICCCM window manager hints are set up to indicate the window's initial state. The `-iconic` option sets the window manager hints to indicate the window should be initially iconified. `XAllocWMHints` allocates a hints structure. Once filled in, `XSetWMHints` sets up the hint property for the window.

The final addition to the window is the `WM_PROTOCOLS` property which indicates window manager protocols the client understands. The most commonly used protocol defined by

ICCCM is `WM_DELETE_WINDOW`. If this atom is listed in the `WM_PROTOCOLS` property of a top-level window, then when the user selects the program be quit from the window manager, the window manager will politely send a `WM_DELETE_WINDOW` message to the client instructing the client to delete the window. If the window is the application's main window, the client is expected to terminate. If this property is not set, the window manager will simply ask the X server to terminate the client's connection without notice to the client. By default, this results in Xlib printing an ugly message like:

```
X connection to :0.0 broken
(explicit kill or server shutdown).
```

Asking to participate in the `WM_DELETE_WINDOW` protocol allows the client to safely handle requests to quit from the window manager.

The property has another advantage for OpenGL programs. Many OpenGL programs doing animation will use `XPending` to check for pending X events and otherwise draw their animation. But if all a client's animation is direct OpenGL rendering and the client does not otherwise do any X requests, the client never sends requests to the X server. Due to a problem in `XPending`'s implementation on many Unix operating systems,[1] such an OpenGL program might not notice its X connection was terminated for sometime. Using the `WM_DELETE_WINDOW` protocol eliminates this problem because the window manager notifies the client via a message (tripping `XPending`) and the client is expected to drop the connection.

Using the `WM_DELETE_WINDOW` protocol is good practice even if you do not use `XPending` and the Xlib message does not bother you.

All these steps (besides creating a window with a non-default visual) are standard for creating a top-level X window. A top-level window is a window created as a child of the root window (the window manager may choose to reparent the window when it is mapped to add a border). Note that the properties discussed are placed on the *top-level* window, not necessarily the same window that OpenGL renders into. While `glxdino` creates a single window, a more complicated program might nest windows used for OpenGL rendering inside the top-level window. The ICCCM window manager properties belong on top-level windows only.

An IRIS GL programmer not familiar with X will probably find these details cumbersome. Most of the work will be done for you if you use a toolkit layered on top of Xlib.

Now a window and an OpenGL rendering context exist. In OpenGL (unlike Xlib), you do not pass the rendering destination into every rendering call. Instead a given OpenGL rendering context is bound to a window using `glXMakeCurrent`. Once bound, all OpenGL rendering calls operate using the current OpenGL rendering context and the current bound window. A thread can only be bound to one window and one rendering context at a time. A context can only be bound to a single thread at a time. If you call `glXMakeCurrent` again, it unbinds from the old context and window and then binds to the newly specified context and window. You can unbind a thread from a window and a context by passing `NULL` for the context and `None` for the drawable.

2.2 The Dinosaur Model

The task of figuring out how to describe the 3D object you wish to render is called *modeling*. Much as a plastic airplane model is constructed out of little pieces, a computer generated 3D scene must also be built out of little pieces. In the case of 3D rendering, the pieces are generally polygons.

The dinosaur model to be displayed is constructed out of a hierarchy of display lists. Rendering the dinosaur is accomplished by executing a single display list.

The strategy for modeling the dinosaur is to construct solid pieces for the body, arms, legs, and eyes. Figure 2 shows the 2D sides of the solids to construct the dinosaur. Making these pieces solid is done by *extruding* the sides (meaning stretching the 2D sides into a third dimension). By correctly situating the solid pieces relative to each other, they form the complete dinosaur.

The work to build the dinosaur model is done by the routine named `makeDinosaur`. A helper routine `extrudeSolidFromPolygon` is used to construct each solid extruded object.

2.2.1 The GLU Tessellator

The polygons in Figure 2 are irregular and complex. For performance reasons, OpenGL directly supports drawing only convex polygons. The complex polygons that make up the sides of the dinosaur need to be built from smaller convex polygons.

Since rendering complex polygons is a common need, OpenGL supplies a set of utility routines in the OpenGL GLU library which make it easy to *tessellate* complex polygons. In computer graphics, tessellation is the process of breaking a complex geometric surface into simple convex polygons.

The GLU library routines for tessellation are:

`gluNewTess` - create a new tessellation object.

`gluTessCallback` - define a callback for a tessellation object.

`gluBeginPolygon` - begin a polygon description to tessellate.

`gluTessVertex` - specify a vertex for the polygon to tessellate.

`gluNextContour` - mark the beginning of another contour for the polygon to tessellate.

---

[1] Operating systems using `FIONREAD ioctl` calls on file descriptors using Berkeley non-blocking I/O cannot differentiate no data to read from a broken connection; both conditions cause the `FIONREAD ioctl` to return zero. MIT's standard implementation of `XPending` uses Berkeley non-blocking I/O and `FIONREAD ioctls`. Eventually, Xlib will do an explicit check on the socket to see if it closes but only after a couple hundred calls to `XPending`.

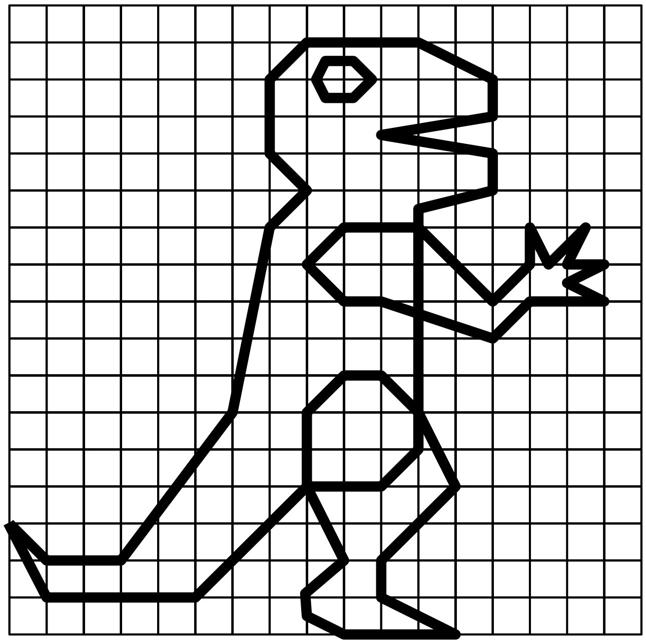


Figure 2: 2D complex polygons used to model the dinosaur's arm, leg, eye, and body sides.

`gluEndPolygon` - finish a polygon being tessellated.

`gluDeleteTess` - destroy a tessellation object.

These routines are used in the example code to tessellate the sides of the dinosaur. Notice at the beginning of the program static arrays of 2D vertices are specified for the dinosaur's body, arm, leg, and eye polygons.

To use the tessellation package, you first create a tessellation object with `gluNewTess`. An object of type `GLUtriangulatorObj*` is returned which is passed into the other polygon tessellation routines. You do not need a tessellation object for every polygon you tessellate. You might need more than one tessellation object if you were trying to tessellate more than one polygon at a time. In the sample program, a single tessellation object is used for all the polygons needing tessellation.

Once you have a tessellation object, you should set up callback routines using `gluTessCallback`. The way that the GLU tessellation package works is that you feed in vertices. Then the tessellation is performed and your registered callbacks are called to indicate the beginning, end, and all the vertices for the convex polygons which correctly tessellate the points you feed to the tessellator.

Look at the `extrudeSolidFromPolygon` routine which uses the GLU tessellation routines. To understand exactly why the callbacks are specified as they are, consult the OpenGL Reference Manual [4]. The point to notice is how a single tessellation object is set up once and callbacks are registered for it. Then `gluBeginPolygon` is used to start tessellating a new complex polygon. The vertices of the polygon are specified using `gluTessVertex`. The polygon is finished by calling `gluEndPolygon`.

Notice the code for tessellating the polygon lies between a `glNewList` and `glEndList`; these routines begin and end the creation of a display list. The callbacks will generate `glVertex2fv` calls specifying the vertices of convex polygons needed to represent the complex polygon being tessellated. Once completed, a display list is available that can render the desired complex polygon.

Consider the performance benefits of OpenGL's polygon tessellator compared with a graphics system that supplies a polygon primitive that supports non-convex polygons. A primitive which supported complex polygons would likely need to tessellate each complex polygon on the fly. Calculating a tessellation is not without cost. If you were drawing the same complex polygon more than once, it is better to do the tessellation only once. This is exactly what is achieved by creating a display list for the tessellated polygon. But if you are rendering continuously changing complex polygons, the GLU tessellator is fast enough for generating vertices on the fly for immediate-mode rendering.

Having a tessellation object not directly tied to rendering is also more flexible. Your program might need to tessellate a polygon but not actually render it. The GLU's system of callbacks just generate vertices. You can call OpenGL `glVertex` calls to render the vertices or supply your own special callbacks to save the vertices for your own purposes. The tessellation algorithm is accessible for your own use.

The GLU tessellator also supports multiple contours allowing disjoint polygons or polygons with holes to be tessellated. The `gluNextContour` routine begins a new contour.

The tessellation object is just one example of functionality in OpenGL's GLU library which supports 3D rendering without complicating the basic rendering routines in the core OpenGL API. Other GLU routines support rendering of curves and surfaces using Non-Uniform Rational B-Splines (NURBS) and tessellating boundaries of solids such as cylinders, cones, and spheres. All the GLU routines are a standard part of OpenGL.

2.2.2 Hierarchical Display Lists

After generating the complex polygon display list for the sides of a solid object, the `extrudeSolidFromPolygon` routine creates another display list for the "edge" of the extruded solid. The edge is generated using a `QUAD_STRIP` primitive. Along with the vertices, normals are calculated for each quad along the edge. Later these normals will be used for lighting the dinosaur. The normals are computed to be unit vectors. Having normals specified as unit vectors is important for correct lighting. An alternative would be to use `glEnable(GL_NORMALIZE)` which ensures all normals are properly normalized before use in lighting calculations. Specifying unit vectors to begin with and not using `glEnable(GL_NORMALIZE)` saves time during rendering. Be careful when using scaling transformations (often set up using `glScale`) since scaling transformations will scale normals too. If you are using scaling transformations, `glEnable(GL_NORMALIZE)` is almost always required for correct lighting.

Once the edge and side display lists are created, the solid is formed by calling the edge display list, then filling in the solid by calling the side display list twice (once translated over by the width of the edge). The `makeDinosaur` routine will use `extrudeSolidFromPolygon` to create solids for each body part needed by the dinosaur.

Then `makeDinosaur` combines these display lists into a single display list for the entire dinosaur. Translations are used to properly position the display lists to form the complete dinosaur. The body display list is called; then arms and legs for the right side are added; then arms and legs for the left side are added; then the eye is added (it is one solid which pokes out either side of the dinosaur's head a little bit on each side).

2.2.3 Back-face Culling

A common optimization in 3D graphics is a technique known as *back-face culling*. The idea is to treat polygons as essentially one-sided entities. A front facing polygon needs to be rendered but a back-facing polygon can be eliminated.

Consider the dinosaur model. When the model is rendered, the back side of the dinosaur will not be visible. If the direction each polygon "faced" was known, OpenGL could simply eliminate approximately half of the polygons (the back-facing ones) without ever rendering them.

Notice the calls to `glFrontFace` when each solid display list is created in `extrudeSolidFromPolygon`. The argument to the call is either `GL_CW` or `GL_CCW` meaning clockwise and counter-clockwise. If the vertices for a polygon are listed in counter-clockwise order and `glFrontFace` is set to `GL_CCW`, then the generated polygon is considered front facing. The static data specifying the vertices of the complex polygons is listed in counter-clockwise order. To make the quads in the quad strip face outwards, `glFrontFace(GL_CW)` is specified. The same mode ensures the far side faces outward. But `glFrontFace(GL_CCW)` is needed to make sure the front of the other side faces outward (logically it needs to be reversed from the opposite side since the vertices were laid out counter-clockwise for both sides since they are from the same display list).

When the static OpenGL state is set up, `glEnable(GL_CULL_FACE)` is used to enable back-face culling. As with all modes enabled and disabled using `glEnable` and `glDisable`, it is disabled by default. Actually OpenGL is not limited to back-face culling. The `glCullFace` routine can be used to specify either the back or the front should be culled when face culling is enabled.

When you are developing your 3D program, it is often helpful to disable back-face culling. That way both sides of every polygon will be rendered. Then once you have your scene correctly rendering, you can go back and optimize your program to properly use back-face culling.

Do not be left with the misconception that enabling or disabling back-face culling (or any other OpenGL feature) must be done for the duration of the scene or program. You can enable and disable back-face culling at will. It is possible to draw part of your scene with back-face culling enabled, and then disable it, only to later re-enable culling but this time for front faces.

2.3 Lighting

The realism of a computer generated 3D scene is greatly enhanced by adding lighting. In the first article's sample program, `glColor3f` was used to add color to the faces of the 3D cube. This adds color to rendered objects but does not use lighting. In the example, the cube moves but the colors do not vary the way a real cube might as it is affected by real world lighting. In this article's example, lighting will be used to add an extra degree of realism to the scene.

OpenGL supports a sophisticated 3D lighting model to achieve higher realism. When you look at a real object, its color is affected by lights, the material properties of the object, and the angle at which the light shines on the object. OpenGL's lighting model approximates the real world.

Complicated effects such as the reflection of light and shadows are not supported by OpenGL's lighting model though techniques and algorithms are available to simulate such effects. Environment mapping to simulate reflection is possible using OpenGL's texturing capability. OpenGL's stencil buffers and blending support can be used to create shadows, but an explanation of these techniques is beyond the scope of this article. (See the topics in the final chapter of the *OpenGL Programming Guide*).

2.3.1 Types of Lighting

The effects of light are complex. In OpenGL, lighting is divided into four different components: emitted, ambient, diffuse, and specular. All four components can be computed independently and then added together.

Emitted light is the simplest. It is light that originates from an object and is unaffected by any light sources. Self-luminous objects can be modeled using emitted light.

Ambient light is light from some source that has been scattered so much by the environment that its direction is impossible to determine. Even a directed light such as a flashlight may have some ambient light associated with it.

Diffuse light comes from some direction. The brightness of the light bouncing off an object depends on the light's angle of incidence with the surface it is striking. Once it hits a surface, the light is scattered equally in all directions so it appears equally bright independent of where the eye is located.

Specular light comes from some direction and tends to bounce off the surface in a certain direction. Shiny metal or plastic objects have a high specular component. Chalk or carpet have almost none. Specularity corresponds to the everyday notion of how shiny an object is.

A single OpenGL light source has a single color and some combination of ambient, diffuse, and specular components.

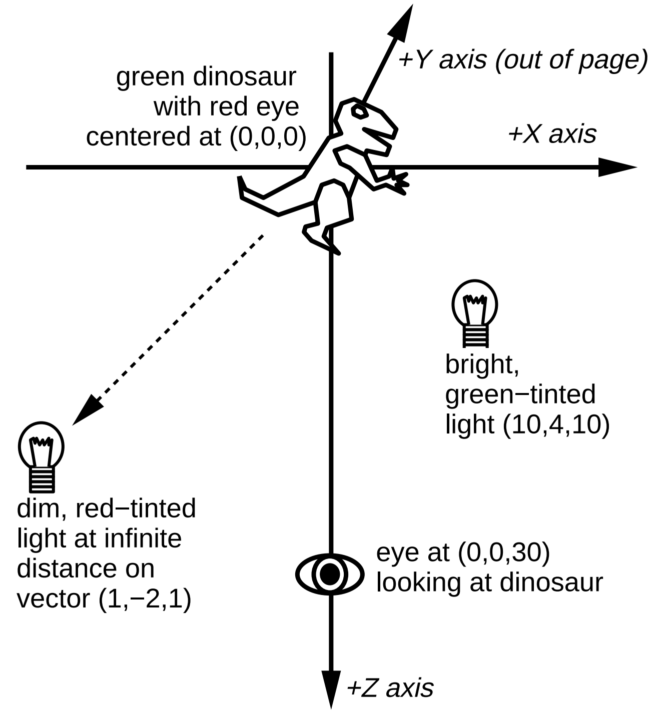


Figure 3: Arrangement of lights, eye, and dinosaur in modeling space.

OpenGL supports multiple lights simultaneously. The programmer can control the makeup of a light as well as its position, direction, and attenuation. Attenuation refers to how a light's intensity decreases as distance from the light increases.

2.3.2 Lighting in the Example

The example uses two lights. Both use only the diffuse component. A bright, slightly green-tinted *positional* light is to the right, front of the dinosaur. A dim, red-tinted *directional* light is coming from the left, front of the dinosaur. Figure 3 shows how the dinosaur, the lights, and the eye-point are arranged. A positional light is located at some finite position in modeling space. A directional light is considered to be located infinitely far away. Using a directional light allows the OpenGL to consider the emitted light rays to be parallel by the time the light reaches the object. This simplifies the lighting calculations needed to be done by OpenGL.

The `lightZeroPosition` and `lightOnePosition` static variables indicate the position of the two lights. You will notice each has not three but four coordinates. This is because the light location is specified in *homogeneous* coordinates. The fourth value divides the X, Y, and Z coordinates to obtain the true coordinate. Notice how `lightOnePosition` (the infinite light) has the fourth value set to zero. This is how an infinite light is specified.[2]

The dinosaur can rotate around the Y axis based on the user's mouse input. The idea behind the example's lighting arrangement is when the dinosaur is oriented so its side faces to the right, it should appear green due to the bright light. When its side faces leftward, the dinosaur should appear poorly lighted but the red infinite light should catch the dinosaur's red eye.

Section 9 of the program initialization shows how lighting is initialized. The `glEnable(GL_LIGHTING)` turns on lighting support. The lights' positions and diffuse components are set using via calls to `glLightfv` using the `GL_POSITION` and `GL_DIFFUSE` parameters. The lights are each enabled using `glEnable`.

The attenuation of the green light is adjusted. This determines how the light intensity fades with distance and demonstrates how individual lighting parameters can be set. It would not make sense to adjust the attenuation of the red light since it is an infinite light which shines with uniform intensity.

Neither ambient nor specular lighting are demonstrated in this example so that the effect of the diffuse lighting would be clear. Specular lighting might have been used to give the dinosaur's eye a glint.

Recall when the edge of each solid was generated, normals were calculated for each vertex along the quad strip. And a single normal was given for each complex polygon side of the solid. These normals are used in the diffuse lighting calculations to determine how much light should be reflected. If you rotate the dinosaur, you will notice the color intensity changes as the angle incidence for the light varies.

Also notice the calls to `glShadeModel`. OpenGL's shade model determines whether flat or smooth shading should be used on polygons. The dinosaur model uses different shading depending on whether a side or edge is being rendered. There is a good reason for this. The `GL_SMOOTH` mode is used on the sides. If flat shading were used instead of smooth, each convex polygon composing the tessellated complex polygon side would be a single color. The viewer could notice exactly how the sides has been tessellated. Smooth shading prevents this since the colors are interpolated across each polygon.

But for the edge of each solid, `GL_FLAT` is used. Because the edge is generated as a quad strip, quads along the strip share vertices. If we used a smooth shading model, each edge between two quads would have a single normal. Some of the edges are very sharp (like the claws in the hand and the tip of the tail). Interpolating across such varying normals would lead to an undesirable visual effect. The fingers would appear rounded if looked at straight on. Instead, with flat shading, each quad gets its own normal and there is no interpolation so the sharp angles are clearly visible.

---

[2] Actually all coordinates are logically manipulated by OpenGL as three-dimensional homogeneous coordinates. The *OpenGL Programming Guide*'s Appendix G [3] briefly explains homogeneous coordinates. A more involved discussion of homogeneous coordinates and why they are useful for 3D computer graphics can be found in Foley and van Dam [1].

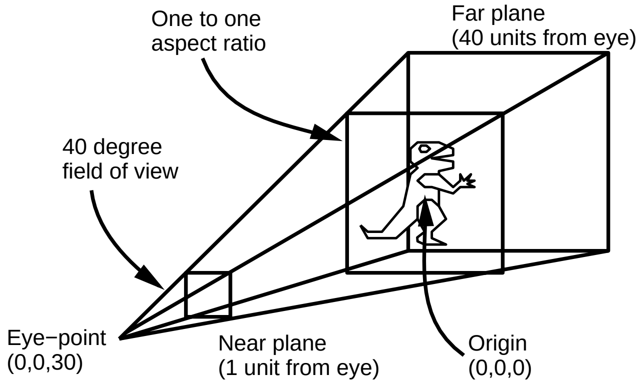


Figure 4: Static view for `glxdino`.

2.4 View Selection

In 3D graphics, *viewing* is the process of establishing the perspective and orientation with which the scene should be rendered. Like a photographer properly setting up his camera, an OpenGL programmer should establish a view. Figure 4 shows how the view is set up for the example program.

In OpenGL, establishing a view means loading the projection and model-view matrices with the right contents. To modify the projection matrix, call `glMatrixMode(GL_PROJECTION)`. Calculating the right matrix by hand can be tricky. The GLU library has two useful routines that make the process easy.

GLU's `gluPerspective` routine allows you to specify a field of view angle, an aspect ratio, and near and far clipping planes. It multiplies the current projection matrix with one created according to the routine's parameters. Since initially the projection matrix is an identity matrix, `glxdino`'s `gluPerspective` call effectively loads the projection matrix.

Another GLU routine, `gluLookAt`, can be used to orient the eye-point for the model-view matrix. Notice how `glMatrixMode(GL_MODELVIEW)` is used to switch to the model-view matrix. Using `gluLookAt` requires you to specify the eye-point's location, a location to look at, and a normal to determine which way is up. Like `gluPerspective`, `gluLookAt` multiplies the matrix it constructs from its parameters with the current matrix. The initial model-view matrix is the identity matrix so `glxdino`'s call to `gluLookAt` effectively loads the model-view matrix.

After the `gluLookAt` call, `glPushMatrix` is called. Both the model-view and projection matrices exist on stacks that can be pushed and popped. Calling `glPushMatrix` pushes a copy of the current matrix onto the stack. When a rotation happens, this matrix is popped off and another `glPushMatrix` is done. This newly pushed matrix is composed with a rotation matrix to reflect the current absolute orientation. Every rotation pops off the top matrix and replaces it with a newly rotated matrix.

Notice that the light positions are not set until after the model-view matrix has been properly initialized.

Because the location of the viewpoint affects the calculations for lighting, separate the projection transformation in the projection matrix and the modeling and viewing transformations in the model-view matrix.

2.5 Event Dispatching

Now the window has been created, the OpenGL renderer has been bound to it, the display lists have been constructed, and OpenGL's state has been configured. All that remains is to request the window be mapped using `XMapWindow` and begin handling any X events sent to the program.

When the window was created, four types of window events were requested to be sent to our application: `Expose` events reporting regions of the window to be drawn, `ButtonPress` events indicating mouse button status, `KeyPress` events indicating a keyboard key has been presed, `MotionNotify` events indicating mouse movement, and `ConfigureNotify` events indicating the window's size or position has changed.

X event dispatching is usually done in an infinite loop. Most X programs do not stop dispatching events until the program terminates. `XNextEvent` can be used to block waiting for an X event. When an event arrives, its type is examined to tell what event has been received.

2.5.1 Expose Handling

For an `Expose` event, the example program just sets a flag indicating the window needs to be redrawn. The reason is that `Expose` events indicate a single sub-rectangle in the window that must be redrawn. The X server will send a number of `Expose` events if a complex region of the window has been exposed.

For a normal X program using 2D rendering, you might be able to minimize the amount needed to redraw the window by carefully examining the rectangles for each `Expose` event. For 3D programs, this is usually too difficult to be worthwhile since it is hard to determine what would need to be done to redraw some sub-region of the window. In practice the window is usually redrawn in its entirety. For the dinosaur example, redrawing involves calling the dinosaur display list with the right view. It is not helpful to know only a sub-region of the window actually needs to be redrawn. For this reason, an OpenGL program should not begin redrawing until it has received all the expose events most recently sent to the window. This practice is known as *expose compression* and helps to avoid redrawing more than you should.

Notice that all that is done to immediately handle an expose is to set the `needRedraw` flag. Then `XPending` is used to determine if there are more events pending. Not until the stream of events pauses is the `redraw` routine really called (and the `needRedraw` flag reset).

The `redraw` routine does three things: it clears the image and depth buffers, executes the dinosaur display list, and either calls `glXSwapBuffers` on the window if double buffered or calls `glFlush`. The current model-view matrix determines in what orientation the dinosaur is drawn.

2.5.2 Window Resizing

The X server sends a `ConfigureNotify` event to indicate a window resize. Handling the event generally requires changing the viewport of OpenGL windows. The sample program calls `glViewport` specifying the window's new width and height. A resize also necessitates a screen redraw so the code "falls through" to the expose code which sets the `needRedraw` flag.

When you resize the window, the aspect ratio of the window may change (unless you have negotiated a fixed aspect ratio with the window manager as the `-keepaspect` option does). If you want the aspect ratio of your final image to remain constant, you might need to respecify the projection matrix with an aspect ratio to compensate for the window's changed aspect ratio. The example does not do this.

2.5.3 Handling Input

The example program allows the user to rotate the dinosaur while moving the mouse by holding down the first mouse button. We record the current angle of rotation whenever a mouse button state changes. As the mouse moves while the first mouse button is held down, the angle is recalculated. A `recalcModelView` flag is set indicating the scene should be redrawn with the new angle.

When there is a lull in events, the model-view matrix is recalculated and then the `needRedraw` flag is set, forcing a redraw. The `recalcModelView` flag is cleared. As discussed earlier, recalculating the model-view is done by popping off the current top matrix using `glPopMatrix` and pushing on a new matrix. This new matrix is composed with a rotation matrix using `glRotatef` to reflect the new absolute angle of rotation. An alternative approach would be to multiply the current matrix by a rotation matrix reflecting the change in angle of rotation. But such a relative approach to rotation can lead to inaccurate rotations due to accumulated floating point round-off errors.

2.5.4 Quitting

Because the `WM_DELETE_WINODW` atom was specified on the top-level window's list of window manager protocols, the event loop should also be ready to handle an event sent by the window manager asking the program to quit. If `glxdino` receives a `ClientMessage` event with the first data item being the `WM_DELETE_WINDOW` atom, the program calls `exit`.

In many IRIS GL demonstration programs, the Escape key is used by convention to quit the program. So `glxdino` shows a simple means to quit in response to an Escape key press.

3 Advanced Xlib and OpenGL

The `glxdino` example demonstrates a good deal of OpenGL's functionality and how to integrate OpenGL with X but there are a number of issues that programmers wanting to write advanced OpenGL programs for X should be aware of.

3.1 Colormaps

Already a method has been presented for sharing colormaps using the ICCCM conventions. Most OpenGL programs do not use the default visual and therefore cannot use the default colormap. Sharing colormaps is therefore important for OpenGL programs to minimize the amount of colormaps X servers will need to create.

Often OpenGL programs require more than one colormap. A typical OpenGL program may do OpenGL rendering in a subwindow but most of the program's user interface is implemented using normal X 2D rendering. If the OpenGL window is 24 bits deep, it would be expensive to require all the user interface windows also to be 24 bits deep. Among other things, pixmaps for the user interface windows would need to be 32 bits per pixel instead of the typical 8 bits per pixel. So the program may use the server's (probably default) 8 bit `PseudoColor` visual for its user interface but use a 24 bit `TrueColor` visual for its OpenGL subwindow. Multiple visuals demand multiple colormaps. Many other situations may arise when an OpenGL program needs multiple colormaps within a single top-level window hierarchy.

Normally window managers assume the colormap that a top-level window and all its subwindows need is the colormap used by the top-level window. A window manager automatically notices the colormap of the top-level window and tries to ensure that that colormap is installed when the window is being interacted with.

With multiple colormaps used inside a single top-level window, the window manager needs to be informed of the other colormaps being used. The Xlib routine `XSetWMColormapWindows` can be used to place a standard property on your top-level window to indicate all the colormaps used by the top-level window and its descendants.

Be careful about using multiple colormaps. It is possible a server will not have enough colormap resources to support the set of visuals and their associated colormaps that you desire. Unfortunately, there is no standard way to determine what sets of visuals and colormaps can be simultaneously installed when multiple visuals are supported. Xlib provides two calls, `XMaxCmapsOfScreen` and `XMinCmapsOfScreen`, but these do not express hardware conflicts between visuals.

Here are some guidelines:

- If `XMaxCmapsOfScreen` returns one, you are guaranteed a single hardware colormap. Colormap flashing is quite likely. You should write your entire application to use a single colormap at a time.

- If an 8 bit `PseudoColor` visual and a 24 bit `TrueColor` visual are supported on a single screen, it is extremely likely a different colormap for each of the two visuals can be installed simultaneously.

- If `XMaxCmapsOfScreen` returns a number higher than one, it is possible that the hardware supports multiple colormaps for the same visual. A rule of thumb is the higher the number, the more likely. If the number is higher than the total number of visuals on the screen, it must be true for at least one visual (but you cannot know which one).

Hopefully multiple hardware colormaps will become more prevalent and perhaps a standard mechanism to detect colormap and visual conflicts will become available.

3.2 Double Buffering

If you are writing an animated 3D program, you will probably want double buffering. It is not always available for OpenGL. You have two choices: run in single-buffered mode or render to a pixmap and copy each new frame to the window using `XCopyArea`.

Note that when you use `glXChooseVisual`, booleans are matched exactly (integers if specified are considered minimums). This means if you want to support double buffering but be able to fall back to single buffering, two calls will be needed to `glXChooseVisual`. If an OpenGL application has sophisticated needs for selecting visuals, `glXGetConfig` can be called on each visual to determine the OpenGL attributes of each visual.

3.3 Overlays

X has a convention for supporting overlay window via special visuals [2]. OpenGL can support rendering into overlay visuals. Even if an X server supports overlay visuals, you will need to make sure those visuals are OpenGL capable. The `glXChooseVisual` routine does allow you to specify the frame buffer layer for the visual you are interested in with the `GLX_LEVEL` attribute. This makes it easier to find OpenGL capable overlay visuals.

IRIS GL programmers are used to assuming the transparent pixel in an overlay visual is always zero. For X and OpenGL, this assumption is no longer valid. You should query the transparent mode and pixel specified by the `SERVER_OVERLAY_VISUALS` property to ensure portability.

IRIS GL programmers are also used to considering overlay planes as being “built-in” to IRIS GL windows. The X model for overlay planes considers an overlay window to be a separate window with its own window ID. To use overlays as one does in IRIS GL, you need to create a normal plane window, then create a child window in the overlay planes with the child’s origin located at the origin of the parent. The child should be maintained to have the same size as the parent. Clear the overlay window to the transparent pixel value to see through to the parent normal plane window. Switching between the overlay and normal planes windows requires a `glXMakeCurrent` call.

It is likely that the overlay visuals will not support the same frame buffer capabilities as the normal plane visuals. You should avoid assuming overlay windows will have frame buffer capabilities such as depth buffers, stencil buffers, or accumulation buffers.

3.4 Mixing Xlib and OpenGL Rendering

In IRIS GL, rendering into an X window using core X rendering after IRIS GL was bound to the window is undefined. This precluded mixing core X rendering with GL rendering in the same window. OpenGL allows its rendering to be mixed with core X rendering into the same window. You should be careful doing so since X and OpenGL rendering requests are logically issued in two distinct streams. If you want to ensure proper rendering, you *must* synchronize the streams. Calling `glXWaitGL` will make sure all OpenGL rendering has finished before subsequent X rendering takes place. Calling `glXWaitX` will make sure all core X rendering has finished before subsequent OpenGL rendering takes place. These requests do not require a protocol round trip to the X server.

The core OpenGL API also includes `glFinish` and `glFlush` commands useful for rendering synchronization. `glFinish` ensures all rendering has appeared on the screen when the routine returns (similar to `XSync`). `glFlush` only ensures the queued commands will eventually be executed (similar to `XFlush`).

Realize that mixing OpenGL and X is not normally necessary. Many OpenGL programs will use a toolkit like Motif for their 2D user interface component and use a distinct X window for OpenGL rendering. This requires no synchronization since OpenGL and core X rendering go to distinct X windows. Only when OpenGL and core X rendering are directed at the same window is synchronization of rendering necessary.

Also OpenGL can be used for extremely fast 2D as well as 3D. When you feel a need to mix core X and OpenGL rendering into the same window, consider rendering what you would do in core X using OpenGL. Not only do you avoid the synchronization overhead, but you can potentially achieve faster 2D using direct rendered OpenGL compared to core X rendering.

3.5 Fonts

Graphics programs often need to display text. You can use X font rendering routines or you can use the GLX `glXUseXFont` routine to create display lists out of X fonts.

Neither of these methods of font rendering may be flexible enough for a program desiring stroke or scalable fonts or having sophisticated font needs. In the future, an OpenGL font manager will be available to meet these needs. In the meantime, you can use `glXUseXFont` or X font rendering or roll your own font support. An easy way to do this is to convert each glyph of your font into a display list. Rendering text in the font becomes a matter of executing the display list corresponding to each glyph in the string to display.

3.6 Display Lists

OpenGL supports immediate mode rendering where commands can be generated on the fly and sent directly to the screen. Programmers should be aware that their OpenGL programs might be run indirectly. In this case, immediate mode rendering could require a great deal of overhead for transport to the X server and possibly across a network.

For this reason, OpenGL programmers should try to use display lists when possible to batch rendering commands. Since the display lists are stored in the server, executing a display list has minimal overhead compared to executing the same commands in the display list immediately.

Display lists are likely to have other advantages since OpenGL implementations are allowed to compile them for maximum performance. Be aware you can mix display lists and immediate mode rendering to achieve the best mix of performance and rendering flexibility.

4 Conclusion

The `glxdino` example demonstrates the basic tasks that must be done to use OpenGL with X. The program demonstrates sophisticated OpenGL features such as double buffering, lighting, shading, back-face culling, display list modeling, and polygon tessellation. And the proper X conventions are followed to ensure `glxdino` works well with other X programs.

The `glxdino` example program and the hints for advanced OpenGL programming should provide a good foundation for understanding and programming OpenGL with Xlib. The next article will explain how to integrate OpenGL with the Motif toolkit.

A glxdino.c

```
/* compile: cc -o glxdino glxdino.c -lGLU -lGL -lXmu -lX11 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>               /* for cos(), sin(), and sqrt() */
#include <GL/glx.h>             /* this includes X and gl.h headers */
#include <GL/glu.h>             /* gluPerspective(), gluLookAt(), GLU polygon
                                 * tesselator */
#include <X11/Xatom.h>          /* for XA_RGB_DEFAULT_MAP atom */
#include <X11/Xmu/StdCmap.h>    /* for XmuLookupStandardColormap() */
#include <X11/keysym.h>         /* for XK_Escape keysym */

typedef enum {
    RESERVED, BODY_SIDE, BODY_EDGE, BODY_WHOLE, ARM_SIDE, ARM_EDGE, ARM_WHOLE,
    LEG_SIDE, LEG_EDGE, LEG_WHOLE, EYE_SIDE, EYE_EDGE, EYE_WHOLE, DINOSAUR
}               displayLists;

Display *dpy;
Window win;
GLfloat angle = -150;   /* in degrees */
GLboolean doubleBuffer = GL_TRUE, iconic = GL_FALSE, keepAspect = GL_FALSE;
int W = 300, H = 300;
XSizeHints sizeHints = {0};
GLdouble bodyWidth = 2.0;
int configuration[] = {GLX_DOUBLEBUFFER, GLX_RGBA, GLX_RED_SIZE, 1, GLX_DEPTH_SIZE, 16, None};
GLfloat body[][2] = { {0, 3}, {1, 1}, {5, 1}, {8, 4}, {10, 4}, {11, 5},
    {11, 11.5}, {13, 12}, {13, 13}, {10, 13.5}, {13, 14}, {13, 15}, {11, 16},
    {8, 16}, {7, 15}, {7, 13}, {8, 12}, {7, 11}, {6, 6}, {4, 3}, {3, 2},
    {1, 2} };
GLfloat arm[][2] = { {8, 10}, {9, 9}, {10, 9}, {13, 8}, {14, 9}, {16, 9},
    {15, 9.5}, {16, 10}, {15, 10}, {15.5, 11}, {14.5, 10}, {14, 11}, {14, 10},
    {13, 9}, {11, 11}, {9, 11} };
GLfloat leg[][2] = { {8, 6}, {8, 4}, {9, 3}, {9, 2}, {8, 1}, {8, 0.5}, {9, 0},
    {12, 0}, {10, 1}, {10, 2}, {12, 4}, {11, 6}, {10, 7}, {9, 7} };
GLfloat eye[][2] = { {8.75, 15}, {9, 14.7}, {9.6, 14.7}, {10.1, 15},
    {9.6, 15.25}, {9, 15.25} };
GLfloat lightZeroPosition[] = {10.0, 4.0, 10.0, 1.0};
GLfloat lightZeroColor[] = {0.8, 1.0, 0.8, 1.0}; /* green-tinted */
GLfloat lightOnePosition[] = {-1.0, -2.0, 1.0, 0.0};
GLfloat lightOneColor[] = {0.6, 0.3, 0.2, 1.0};  /* red-tinted */
GLfloat skinColor[] = {0.1, 1.0, 0.1, 1.0}, eyeColor[] = {1.0, 0.2, 0.2, 1.0};
GC gc;
XGCValues gcvals;

void
fatalError(char *message)
{
    fprintf(stderr, "glxdino: %s\n", message);
    exit(1);
}

Colormap
getColormap(XVisualInfo * vi)
{
    Status          status;
    XStandardColormap *standardCmaps;
    Colormap        cmap;
    int             i, numCmaps;
```

```

    /* be lazy; using DirectColor too involved for this example */
    if (vi->class != TrueColor)
        fatalError("no support for non-TrueColor visual");
    /* if no standard colormap but TrueColor, just make an unshared one */
    status = XmuLookupStandardColormap(dpy, vi->screen, vi->visualid,
        vi->depth, XA_RGB_DEFAULT_MAP, /* replace */ False, /* retain */ True);
    if (status == 1) {
        status = XGetRGBColormaps(dpy, RootWindow(dpy, vi->screen),
                             &standardCmaps, &numCmaps, XA_RGB_DEFAULT_MAP);
        if (status == 1)
            for (i = 0; i < numCmaps; i++)
                if (standardCmaps[i].visualid == vi->visualid) {
                    cmap = standardCmaps[i].colormap;
                    XFree(standardCmaps);
                    return cmap;
                }
    }
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    return cmap;
}

void
extrudeSolidFromPolygon(GLfloat data[][2], unsigned int dataSize,
    GLdouble thickness, GLuint side, GLuint edge, GLuint whole)
{
    static GLUtriangulatorObj *tobj = NULL;
    GLdouble        vertex[3], dx, dy, len;
    int             i;
    int             count = dataSize / (2 * sizeof(GLfloat));

    if (tobj == NULL) {
        tobj = gluNewTess();    /* create and initialize a GLU polygon
                                 * tesselation object */
        gluTessCallback(tobj, GLU_BEGIN, glBegin);
        gluTessCallback(tobj, GLU_VERTEX, glVertex2fv); /* semi-tricky */
        gluTessCallback(tobj, GLU_END, glEnd);
    }
    glNewList(side, GL_COMPILE);
        glShadeModel(GL_SMOOTH); /* smooth minimizes seeing tessellation */
        gluBeginPolygon(tobj);
            for (i = 0; i < count; i++) {
                vertex[0] = data[i][0];
                vertex[1] = data[i][1];
                vertex[2] = 0;
                gluTessVertex(tobj, vertex, &data[i]);
            }
        gluEndPolygon(tobj);
    glEndList();
    glNewList(edge, GL_COMPILE);
        glShadeModel(GL_FLAT);  /* flat shade keeps angular hands from being
                                 * "smoothed" */
        glBegin(GL_QUAD_STRIP);
        for (i = 0; i <= count; i++) {
            /* mod function handles closing the edge */
            glVertex3f(data[i % count][0], data[i % count][1], 0.0);
            glVertex3f(data[i % count][0], data[i % count][1], thickness);
            /* Calculate a unit normal by dividing by Euclidean distance. We
             * could be lazy and use glEnable(GL_NORMALIZE) so we could pass in
```

```
             * arbitrary normals for a very slight performance hit. */
            dx = data[(i + 1) % count][1] - data[i % count][1];
            dy = data[i % count][0] - data[(i + 1) % count][0];
            len = sqrt(dx * dx + dy * dy);
            glNormal3f(dx / len, dy / len, 0.0);
        }
        glEnd();
    glEndList();
    glNewList(whole, GL_COMPILE);
        glFrontFace(GL_CW);
        glCallList(edge);
        glNormal3f(0.0, 0.0, -1.0); /* constant normal for side */
        glCallList(side);
        glPushMatrix();
            glTranslatef(0.0, 0.0, thickness);
            glFrontFace(GL_CCW);
            glNormal3f(0.0, 0.0, 1.0); /* opposite normal for other side */
            glCallList(side);
        glPopMatrix();
    glEndList();
}

void
makeDinosaur(void)
{
    GLfloat         bodyWidth = 3.0;

    extrudeSolidFromPolygon(body, sizeof(body), bodyWidth,
        BODY_SIDE, BODY_EDGE, BODY_WHOLE);
    extrudeSolidFromPolygon(arm, sizeof(arm), bodyWidth / 4,
        ARM_SIDE, ARM_EDGE, ARM_WHOLE);
    extrudeSolidFromPolygon(leg, sizeof(leg), bodyWidth / 2,
        LEG_SIDE, LEG_EDGE, LEG_WHOLE);
    extrudeSolidFromPolygon(eye, sizeof(eye), bodyWidth + 0.2,
        EYE_SIDE, EYE_EDGE, EYE_WHOLE);
    glNewList(DINOSAUR, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, skinColor);
        glCallList(BODY_WHOLE);
        glPushMatrix();
            glTranslatef(0.0, 0.0, bodyWidth);
            glCallList(ARM_WHOLE);
            glCallList(LEG_WHOLE);
            glTranslatef(0.0, 0.0, -bodyWidth - bodyWidth / 4);
            glCallList(ARM_WHOLE);
            glTranslatef(0.0, 0.0, -bodyWidth / 4);
            glCallList(LEG_WHOLE);
            glTranslatef(0.0, 0.0, bodyWidth / 2 - 0.1);
            glMaterialfv(GL_FRONT, GL_DIFFUSE, eyeColor);
            glCallList(EYE_WHOLE);
        glPopMatrix();
    glEndList();
}

void
redraw(void)
{
static int x = 0;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glCallList(DINOSAUR);
    if (doubleBuffer)
```

```
        glXSwapBuffers(dpy, win);       /* buffer swap does implicit glFlush */
    else glFlush();             /* explicit flush for single buffered case */
#if 1
    XDrawLine(dpy, win, gc, 10+x, 10, 40+x, 40);
    x+=8;
    XSync(dpy, 0);
#endif
}

void
main(int argc, char **argv)
{
    XVisualInfo    *vi;
    Colormap        cmap;
    XSetWindowAttributes swa;
    XWMHints       *wmHints;
    Atom            wmDeleteWindow;
    GLXContext      cx;
    XEvent          event;
    KeySym          ks;
    GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
    char           *display = NULL, *geometry = NULL;
    int             flags, x, y, width, height, lastX, i;

    /*** (1) process normal X command line arguments ***/
    for (i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-geometry")) {
            if (++i >= argc)
                fatalError("follow -geometry option with geometry parameter");
            geometry = argv[i];
        } else if (!strcmp(argv[i], "-display")) {
            if (++i >= argc)
                fatalError("follow -display option with display parameter");
            display = argv[i];
        } else if (!strcmp(argv[i], "-iconic")) iconic = GL_TRUE;
        else if (!strcmp(argv[i], "-keepaspect")) keepAspect = GL_TRUE;
        else if (!strcmp(argv[i], "-single")) doubleBuffer = GL_FALSE;
        else fatalError("bad option");
    }

    /*** (2) open a connection to the X server ***/
    dpy = XOpenDisplay(display);
    if (dpy == NULL) fatalError("could not open display");

    /*** (3) make sure OpenGL’s GLX extension supported ***/
    if (!glXQueryExtension(dpy, NULL, NULL))
        fatalError("X server has no OpenGL GLX extension");

    /*** (4) find an appropriate visual and a colormap for it ***/
    /* find an OpenGL-capable RGB visual with depth buffer */
    if (!doubleBuffer) goto SingleBufferOverride;
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), configuration);
    if (vi == NULL) {
      SingleBufferOverride:
        vi = glXChooseVisual(dpy, DefaultScreen(dpy), &configuration[1]);
        if (vi == NULL)
            fatalError("no appropriate RGB visual with depth buffer");
        doubleBuffer = GL_FALSE;
    }
    cmap = getColormap(vi);
```

```
    /*** (5) create an OpenGL rendering context  ***/
    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi, /* no sharing of display lists */ NULL,
                           /* direct rendering if possible */ GL_TRUE);
    if (cx == NULL) fatalError("could not create rendering context");

    /*** (6) create an X window with selected visual and right properties ***/
    flags = XParseGeometry(geometry, &x, &y,
        (unsigned int *) &width, (unsigned int *) &height);
    if (WidthValue & flags) {
        sizeHints.flags |= USSize;
        sizeHints.width = width;
        W = width;
    }
    if (HeightValue & flags) {
        sizeHints.flags |= USSize;
        sizeHints.height = height;
        H = height;
    }
    if (XValue & flags) {
        if (XNegative & flags)
            x = DisplayWidth(dpy, DefaultScreen(dpy)) + x - sizeHints.width;
        sizeHints.flags |= USPosition;
        sizeHints.x = x;
    }
    if (YValue & flags) {
        if (YNegative & flags)
            y = DisplayHeight(dpy, DefaultScreen(dpy)) + y - sizeHints.height;
        sizeHints.flags |= USPosition;
        sizeHints.y = y;
    }
    if (keepAspect) {
        sizeHints.flags |= PAspect;
        sizeHints.min_aspect.x = sizeHints.max_aspect.x = W;
        sizeHints.min_aspect.y = sizeHints.max_aspect.y = H;
    }
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | StructureNotifyMask |
        ButtonPressMask | Button1MotionMask | KeyPressMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen),
                        sizeHints.x, sizeHints.y, W, H,
                        0, vi->depth, InputOutput, vi->visual,
                        CWBorderPixel | CWColormap | CWEventMask, &swa);
    gcvals.line_width = 5;
    gcvals.foreground = 45;
    gc = XCreateGC(dpy, win, GCForeground|GCLineWidth, &gcvals);
    XSetStandardProperties(dpy, win, "OpenGLosaurus", "glxdino",
        None, argv, argc, &sizeHints);
    wmHints = XAllocWMHints();
    wmHints->initial_state = iconic ? IconicState : NormalState;
    wmHints->flags = StateHint;
    XSetWMHints(dpy, win, wmHints);
    wmDeleteWindow = XInternAtom(dpy, "WM_DELETE_WINDOW", False);
    XSetWMProtocols(dpy, win, &wmDeleteWindow, 1);

    /*** (10) request the X window to be displayed on the screen ***/
    XMapWindow(dpy, win);
    sleep(1);
```

```

    /*** (7) bind the rendering context to the window ***/
    glXMakeCurrent(dpy, win, cx);

    /*** (8) make the desired display lists ***/
    makeDinosaur();

    /*** (9) configure the OpenGL context for rendering ***/
    glEnable(GL_CULL_FACE);     /* ˜50% better perfomance than no back-face
                                 * culling on Entry Indigo */
    glEnable(GL_DEPTH_TEST);    /* enable depth buffering */
    glEnable(GL_LIGHTING);      /* enable lighting */
    glMatrixMode(GL_PROJECTION);/* set up projection transform */
    gluPerspective( /* field of view in degree */ 40.0, /* aspect ratio */ 1.0,
                    /* Z near */ 1.0, /* Z far */ 40.0);
    glMatrixMode(GL_MODELVIEW); /* now change to modelview */
    gluLookAt(0.0, 0.0, 30.0,   /* eye is at (0,0,30) */
              0.0, 0.0, 0.0,    /* center is at (0,0,0) */
              0.0, 1.0, 0.);    /* up is in postivie Y direction */
    glPushMatrix();             /* dummy push so we can pop on model recalc */
    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 1);
    glLightfv(GL_LIGHT0, GL_POSITION, lightZeroPosition);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05);
    glLightfv(GL_LIGHT1, GL_POSITION, lightOnePosition);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, lightOneColor);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);        /* enable both lights */

    /*** (11) dispatch X events ***/
    while (1) {
        do {
            XNextEvent(dpy, &event);
            switch (event.type) {
            case ConfigureNotify:
                glViewport(0, 0,
                    event.xconfigure.width, event.xconfigure.height);
                /* fall through... */
            case Expose:
                needRedraw = GL_TRUE;
                break;
            case MotionNotify:
                recalcModelView = GL_TRUE;
                angle -= (lastX - event.xmotion.x);
            case ButtonPress:
                lastX = event.xbutton.x;
                break;
            case KeyPress:
                ks = XLookupKeysym((XKeyEvent *) & event, 0);
                if (ks == XK_Escape) exit(0);
                break;
            case ClientMessage:
                if (event.xclient.data.l[0] == wmDeleteWindow) exit(0);
                break;
            }
        } while (XPending(dpy));/* loop to compress events */
        if (recalcModelView) {
            glPopMatrix();      /* pop old rotated matrix (or dummy matrix if
                                 * first time) */
```

```
            glPushMatrix();
            glRotatef(angle, 0.0, 1.0, 0.0);
            glTranslatef(-8, -8, -bodyWidth / 2);
            recalcModelView = GL_FALSE;
            needRedraw = GL_TRUE;
        }
        if (needRedraw) {
            redraw();
            needRedraw = GL_FALSE;
        }
    }
}
```

References

[1] James Foley, Andries van Dam, Steven Feiner, and John Hughes, *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley Publishing, 1990.

[2] Mark Kilgard, "Programming X Overlay Windows," *The X Journal*, SIGS Publications, July 1993.

[3] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.

[4] OpenGL Architecture Review Board, *OpenGL Reference Manual: The official reference document for OpenGL, Release 1*, Addison Wesley, 1992.

PUBLISHED IN THE
JULY/AUGUST ISSUE OF *The X Journal*.

OPENGL$^{\text{TM}}$ AND X, PART 3: INTEGRATING OPENGL WITH MOTIF

Mark J. Kilgard [*]
Silicon Graphics Inc.

$Revision : 1.19$

May 7, 1997

Abstract

The OpenGL$^{\text{TM}}$ graphics system can be integrated with the industry-standard OSF/Motif user interface. This article discusses how to use OpenGL within a Motif application program. There are two approaches to using OpenGL with Motif. One is to render into a standard Motif drawing area widget, but this requires each application window to use a single visual for its window hierarchy. A better approach is to use the special OpenGL drawing area widget allowing windows used for OpenGL rendering to pick freely an appropriate visual without affecting the visual choice for other widgets. An example program demonstrates both approaches. The X Toolkit's work procedure mechanism animates the example's 3D paper airplanes. Handling OpenGL errors is also explained.

1 Introduction

OSF/Motif is the X Window System's industry-standard programming interface for user interface construction. Motif programmers writing 3D applications will want to understand how to integrate Motif with the OpenGL$^{\text{TM}}$ graphics system. This article, the last in a three-part series about OpenGL, describes how to write an OpenGL program within the user interface framework provided by Motif and the X Toolkit.

Most 3D applications end up using 3D graphics primarily in one or more "viewing" windows. For the most part, the graphical user interface aspects of such programs use standard 2D user interface objects like pulldown menus, sliders, and dialog boxes. Creating and managing such common user interface objects is what Motif does well. The "viewing" windows used for 3D are where OpenGL rendering happens. These windows for OpenGL rendering can be constructed with standard Motif drawing area widgets or OpenGL-specific drawing area widgets. Bind an OpenGL rendering context to the window of a drawing area widget and you are ready for 3D rendering.

Programming OpenGL with Motif has numerous advantages over using "Xlib only" as described in the first two articles in this series [2, 3]. First and most important, Motif provides a well-documented, standard widget set that gives your application a consistent look and feel. Second, Motif and the X Toolkit take care of routine but complicated issues such as *cut and paste* and window manager conventions. Third, the X Toolkit's work procedure and timeout mechanisms make it easy to animate a 3D window without blocking out user interaction with your application.

This article assumes you have some experience programming with Motif and you have a basic understanding of how OpenGL integrates with X.

Section 2 describes how to use OpenGL rendering with either a standard Motif drawing area widget or an OpenGL-specific drawing area widget. Section 3 discusses using X Toolkit mechanisms for seamless animation. Section 4 provides advice on how to debug OpenGL programs by catching OpenGL errors. Throughout the discussion, a Motif-based OpenGL program named `paperplane` is used as an example. The complete source code for `paperplane` is found in the appendix. The program animates the 3D flight paths of virtual paper airplanes. The user can interact with the program via Motif controls. The program can be compiled to use either a standard Motif drawing area widget or an OpenGL-specific drawing area widget. Figure 1 shows `paperplane` running.

2 OpenGL with Widgets

Your application's 3D viewing area can be encapsulated by an X Toolkit widget. There are two approaches to rendering OpenGL into a widget. You can render OpenGL into a standard Motif drawing area widget, or you can use a special OpenGL drawing area widget.

---

[*] Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

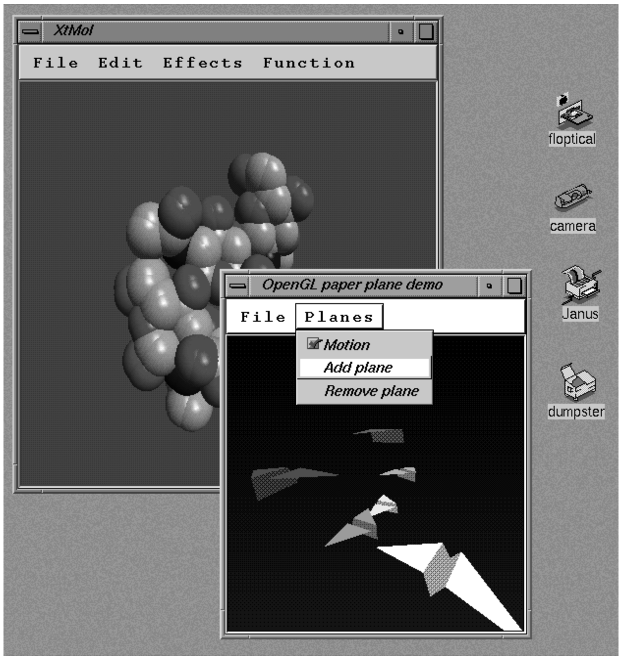


Figure 1: Screen snapshot of `paperplane` with another OpenGL Motif program for molecular modeling.

The Motif drawing area widget would seem a natural widget for OpenGL rendering. Unfortunately, the X Toolkit's design (upon which Motif relies) allows programmers to specify a widget's visual only if its class is derived from the shell widget class. Shell widgets are often called "top level" widgets because they are designed to communicate with the window manager and act as containers for other widgets. Non-shell widgets inherit the depth and visual of their parent widget. The Motif drawing area widget class (like most widget classes) is not derived from the shell widget class. It is impossible (without resorting to programming widget internals) to set the visual of a standard non-shell Motif widget differently than its ancestor shell widget.

But in OpenGL, the X notion of a visual has expanded importance for determining the OpenGL frame buffer capabilities of an X window. In many cases, an application's 3D viewing area is likely to demand a deeper, more capable visual than the default visual which Motif normally uses.

There are two options:

1. Use the standard Motif drawing area widget for your OpenGL rendering area and make sure that the top level shell widget is created with the desired visual for OpenGL's use.
2. Use an OpenGL drawing area widget that is specially programmed to overcome the limitation on setting the visual and depth of a non-shell widget.

Either approach works.

The `paperplane` example in the appendix is written to support either scheme depending on how the code is compiled. By default, the code compiles to use the OpenGL-specific widget. If the `noGLwidget` C preprocessor symbol is defined, the standard Motif drawing area widget will be used, forcing the use of a single visual throughout the example's widget hierarchy. The code differences between the two schemes in the `paperplane` example constitute seven changed lines of code.

The preferable approach is to use the OpenGL-specific widget, since you can run most of the application's user interface in the default visual and use a deeper, more capable visual only for 3D viewing areas. Limiting the use of deeper visuals can save memory and increase rendering speed for the user interface windows. If you use a 24-bit visual for your 3D viewing area and use the same visual for your entire application, that means that the image memory for pixmaps used by non-OpenGL windows is four times what it would be for an 8-bit visual.[1] Some X rendering operations might also be slower for 24-bit windows compared with 8-bit windows.

There can be advantages to running your entire application in a single visual. Some workstations with limited colormap resources might not be capable of showing multiple visuals without colormap flashing. Such machines which support OpenGL should be rare. Even if running in a single visual is appropriate, nothing precludes doing so using an OpenGL-specific widget.

2.1 The OpenGL-specific Widget

There are two OpenGL-specific drawing area widget classes. One is derived from the Motif primitive widget class (*not* the Motif drawing area widget class). The other is derived from the X Toolkit core widget class. Both have the same basic functionality; the main difference is that the Motif-based widget class gains capabilities of the Motif primitive widget class. If you use Motif, you should use the Motif OpenGL widget. If you use a non-Motif widget set, you can use the second widget for identical functionality.

The Motif OpenGL widget class is named `glwMDrawingAreaWidgetClass`; the non-Motif OpenGL widget class is named `glwDrawingAreaWidgetClass` (the difference is the lack of an `M` in the non-Motif case). Since the Motif OpenGL widget is subclassed from the Motif primitive widget class, the Motif OpenGL widget inherits the capabilities of the primitive class like a help callback and keyboard traversal support (keyboard traversal is disabled by default for the Motif OpenGL widget). The `paperplane` example uses the Motif widget by default but the non-Motif widget can be used by defining the `noMotifGLwidget` C preprocessor symbol when compiling `paperplane.c`. The difference is two changed lines of code with no functional difference in the program.

When you create either type of widget, you need to specify the visual to use by supplying the widget's `GLwNvisualInfo` resource. The attribute is of type `XVisualInfo*` making it easy to find an appropriate visual using `glXChooseVisual`

---

[1] Even though a 24-bit pixel requires only three bytes of storage, efficient manipulation of the pixels demands each pixel is stored in an even 4 bytes.

which returns a `XVisualInfo*` for a visual with the capabilities you request.

Although this practice is not recommended, the widgets also allow you to specify the OpenGL capabilities you desire for the widget directly using widget resources. Because the X Toolkit widget creation process is not expected to fail, there is no way for a widget creation routine to indicate failure. If a visual that matches the desired OpenGL capabilities cannot be found, the widget code prints an error and exits without giving the program a chance to handle the failure. If you request a specific `XVisualInfo*` that has already been determined to be acceptable using `glXChooseVisual` or calls to `glXGetConfig`, you will not have this problem. As a rule, always specify the visual using the `GLwNvisualInfo` resource.

The OpenGL widgets also do extra work that might go unnoticed. Because the OpenGL widget uses a different visual, the widget's creation code creates a colormap matching the visual. It also posts an ICCCM `WM_COLORMAP_WINDOWS` top level window property to let the window manager know that the program uses multiple colormaps.

More information about the OpenGL widgets can be found in the Silicon Graphics *OpenGL Porting Guide* [4] and the widgets' man pages.[2]

2.2 The Motif Drawing Area Widget

Using the standard Motif drawing area widget with OpenGL has some extra caveats. The main caveat is that you must create the top level widget with the correct visual for the program's OpenGL rendering.

When you start a widget program, there is generally a call to `XtAppInitialize` to establish the connection to the X server and create the top level widget. Both steps are done in the same routine. So how can we call `glXChooseVisual` to know what visual the top level widget should use until we have established a connection to the X server?

It would appear that it is impossible to create the top level widget with an appropriate visual for OpenGL. `XtAppInitialize` connects to the X server and creates the top level widget, but it does not *realize* the top level widget. The X window for the top level widget is not created until `XtRealizeWidget` is called. This allows `XtSetValues` to be used after the top level widget's creation (and before its realization) to specify the widget's visual. The `paperplane` sample code in the non-OpenGL widget case demonstrates this.

A second caveat is due to the X Toolkit's inconsistent inheritance of the visual, depth, and colormap widget resources. The default visual of a widget's window is copied from its *parent window*'s visual. But the default colormap and depth of a widget are copied from the widget's *parent widget*.[3]

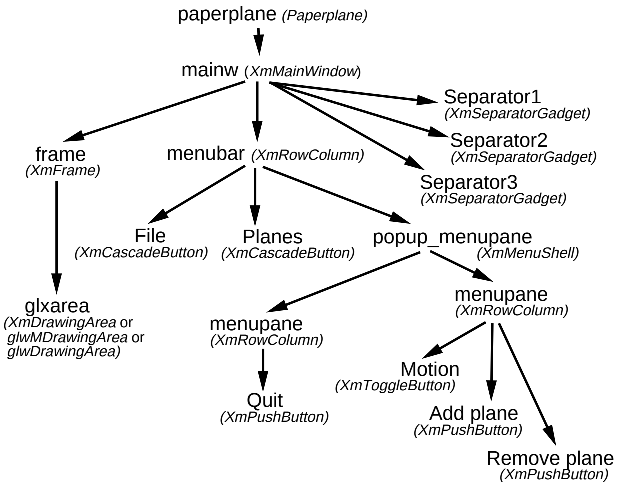


Figure 2: Diagram of the widget hierarchy for `paperplane`. The `glxarea` `XmDrawingArea` widget is the only widget rendered using OpenGL.

This means that if you create a widget derived from the shell widget and the widget's parent uses a non-default depth or colormap for a non-default visual, you will need to specify the same visual as the new widget's parent widget. If you do not, a `BadMatch` X protocol error will result. For this reason the `paperplane` example's `XmCreatePulldownMenu` calls specify the visual of the created widget's parent widget in the Motif drawing area version of `paperplane`.

Realize that it is not possible to bind an OpenGL rendering context to a widget's window until the widget has been realized. Until the widget is realized, the widget's window does not yet exist. Notice `paperplane` does not call `glXMakeCurrent` until after `XtRealizeWidget` has been called.

To see how the 3D viewing area widget fits into the `paperplane` widget hierarchy example, Figure 2 shows the complete hierarchy including widget class names.

These caveats are not unique to OpenGL. The problem comes from using non-default visuals with the X Toolkit. PEXlib 5.1 programs have a similar need for non-default visuals and require the same jumping through hoops[1]. Fortunately, if you use the OpenGL drawing area widgets, you can avoid the caveats of using the standard Motif drawing area.

2.3 Drawing Area Callbacks

Applications using the Motif drawing area widget or the OpenGL drawing area widgets for their 3D rendering will want to register routines to handle expose, resize, and input callbacks using `XtAddCallback`. In `paperplane.c`, the `draw`, `resize`, and `input` routines handle these callbacks.

`paperplane`'s drawing area adjusts OpenGL's viewport by calling `glViewport`. Note how the `made_current` variable is used to protect against calling `glViewport` before we have done the `glXMakeCurrent` to bind to the draw-

---

[2] The *official* standard location for the OpenGL widget headers is `<X11/GLw/GLwDrawA.h>` and `<X11/GLw/GLwMDrawA.h>`. In IRIX 5.2, these headers are mistakenly located at `<GL/GLwDrawA.h>` and `<GL/GLwMDrawA.h>`.

[3] If the widget has no parent, the depth and colormap are determined by the default depth and colormap of the screen.

ing area window. In the X Toolkit, the resize callback can be called before the `XtRealizeWidget` routine returns. Since the program does not call `glXMakeCurrent` until after the program returns from `XtRealizeWidget`, the OpenGL rendering context would not be bound. Calling an OpenGL routine before a context is bound has no effect but generates an ugly warning message.[4] An example of when the resize callback can be called before `XtRealizeWidget` returns is when a `-geometry` command line option is specified.

Note that `glXMakeCurrent` is defined to set a context's viewport to the size of the first window it is bound to. (This happens only on the context's first bind.) This is why `paperplane.c` makes no initial call to `glViewport`; `glXMakeCurrent` sets the viewport implicitly.

The `paperplane` example uses a single window for OpenGL rendering. For this reason, `glXMakeCurrent` is called only once to bind the OpenGL context to the window. In a program with multiple OpenGL windows, each expose and resize callback should make sure that `glXMakeCurrent` is called so that OpenGL rendering goes to the correct window.

The `draw` callback routine issues the OpenGL commands to draw the scene. If the window is double buffered, `glXSwapBuffers` swaps the window's buffers. If the context is not direct, `glFinish` is called to avoid the latency from queuing more than one frame at a time; interactivity would suffer if we allowed more than one frame to be queued. Direct rendering involves direct manipulation of the hardware so it generally has less latency than a potentially networked indirect OpenGL context.

Note that you can render OpenGL into *any* widget (as long as it is created with an OpenGL capable visual). There is nothing special about the Motif or OpenGL-specific drawing area widgets, though drawing area widgets tend to be the most appropriate widget type for a 3D viewing area.

2.4 Handling Input

The `input` routine handles X events for the drawing area. Input events require no special handling for OpenGL. But remember that the coordinate systems for X and OpenGL are distinct, so pointer locations need to be mapped into OpenGL's coordinate space. OpenGL generally assumes that the origin is in the lower left-hand corner, while X always assumes an origin at the upper left-hand corner.

3 Animation Via Work Procedures

The X Toolkit's work procedure facility makes it easy to integrate continuous OpenGL animation with Motif user interface operation. Work procedures are application supplied routines that execute while the application is idle waiting for events. Work procedures should be used to do small amounts of work; if too much time is spent in a work procedure, X events will not be processed and program interactivity will suffer.

Rendering a single frame of OpenGL animation is a good use for work procedures. `XtAppAddWorkProc` and `XtRemoveWorkProc` are used to add and remove work procedures. `XtAppAddWorkProc` is passed a function pointer for the routine to be called as a work procedure. The function to be called returns a `Boolean`. If the function returns `True`, the work procedure should be removed automatically; returning `False` indicates the work procedure should remain active. `XtAppAddWorkProc` returns an ID of type `WorkProcId` which can later be given to `XtRemoveWorkProc` to remove the work procedure.

The `paperplane` example uses a work procedure to manage the update of its 3D scene. In response to changing the state of the "Motion" toggle button on the "Planes" pulldown menu, the `toggle` callback routine will add and remove the `animate` work procedure.

The `animate` routine calls `tick` which advances the position of each active plane; `animate` then calls `draw` to redraw the scene with the new plane locations. Finally, `animate` returns `False` to leave the work procedure installed so that the animation will continue.

Because `paperplane` uses a work procedure, animation of the scene does not interfere with window resizing and user input. The X Toolkit manages both the animation and events from the X server.

3.1 Handling Iconification

When the `paperplane` window is open, we want the `animate` work procedure to update the 3D scene continuously. If the user iconifies the window, it would be wasteful to continue animating a no longer visible scene. To avoid wasting resources rendering to an unmapped window, `paperplane` installs an event handler called `map_state_changed` for the top-level widget to notice `UnmapNotify` and `MapNotify` events. The handler makes sure the work procedure is removed or added to reflect the map state of the window.

3.2 Timeouts

X Toolkit timeouts are similar to work procedures, but instead of being activated whenever event dispatching is idle, they are called when a given period of time has expired. The `XtAppAddTimeout` and `XtRemoveTimeOut` routines can be used to add and remove X Toolkit timeouts.

OpenGL programmers may find timeouts useful to maintain animation at rates slower than "as fast as OpenGL will render." Timeouts can be used to give animation a sustained frame rate. Timeouts can also be used to redraw a scene with higher detail when the user has stopped interacting with the program. For example, a 3D modeling program might redraw its model with lighting enabled and finer tessellation after the program has

---

[4] The exact behavior is undefined by the OpenGL specification.

been idle for two seconds. Timeouts can also be used to trigger simple real-time state changes useful for visual simulation.

4 Debugging Tips

As well as demonstrating the use of widgets with OpenGL, `paperplane` also demonstrates detection of OpenGL errors for debugging purposes. Some debugging code has been added to the bottom of `paperplane`'s `draw` function to test for any OpenGL errors. A correct OpenGL program should not generate any OpenGL errors, but while debugging it is helpful to check explicitly for errors. A good time to check for errors is at the end of each frame. Errors in OpenGL are not reported unless you explicitly check for them, unlike X protocol errors which are always reported to the client.

OpenGL errors are recorded by setting "sticky" flags. Once an error flag is set, it will not be cleared until `glGetError` is used to query the error. An OpenGL implementation may have several error flags internally that can be set (since OpenGL errors might occur in different stages of the OpenGL rendering pipeline). When you look for errors, you should call `glGetError` repeatedly until it returns `GL_NO_ERROR` indicating that all of the error flags have been cleared.

The OpenGL error model is suited for high performance rendering, since error reporting does not slow down the error-free case. Because OpenGL errors should not be generated by bug-free code, you probably want to remove error querying from your final program since querying errors will slow down your rendering speed.

When an OpenGL error is generated, the command which generated the error is not recorded, so you may need to add more error queries into your code to isolate the source of the error.

The `gluErrorString` routine in the OpenGL Utility library (GLU) converts an OpenGL error number into a human readable string and helps you output a reasonable error message.

5 Conclusion

OpenGL and Motif are a powerful combination. Using both APIs allow X applications programmers to get the most out of both Motif and OpenGL.

Still another way to integrate OpenGL rendering with widgets is the Open Inventor object-oriented 3D graphics toolkit which renders using OpenGL and integrates with X Toolkit widgets. Open Inventor allows you to specify 3D scenes in an object-oriented fashion instead of low-level OpenGL rendering primitives. If you are interested in object-oriented 3D, check out the recently published *Inventor Mentor* [5].

The source code presented in this series is available by anonymous ftp to `sgigate.sgi.com` in the `pub/opengl/xjournal` directory.

Acknowledgments

Writing these three articles on OpenGL required the assistance from numerous engineers and managers at Silicon Graphics. In particular I would like to thank Kurt Akeley, David Blythe, Simon Hui, Phil Karlton, Mark Segal, Kevin Smith, Joel Tesler, Tom Weinstein, Mason Woo, and David Yu.

A paperplane.c

```c
/*
 * paperplane can be compiled to use a "single visual" for the entire window
 * hierarchy and render OpenGL into a standard Motif drawing area widget:
 *
 *  cc -o sv_paperplane paperplane.c -DnoGLwidget -lGL -lXm -lXt -lX11 -lm
 *
 * Or paperplane can be compiled to use the default visual for most of
 * the window hierarchy but render OpenGL into a special "OpenGL widget":
 *
 *  cc -o glw_paperplane paperplane.c -lGLw -lGL -lXm -lXt -lX11 -lm
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <Xm/MainW.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <Xm/ToggleB.h>
#include <Xm/CascadeB.h>
#include <Xm/Frame.h>
#ifdef noGLwidget
#include <Xm/DrawingA.h>        /* Motif drawing area widget */
#else
/** NOTE: in IRIX 5.2, the OpenGL widget headers are mistakenly in   **/
/** <GL/GLwDrawA.h> and <GL/GlwMDraw.h> respectively.  Below are the **/
/** _official_ standard locations.                                   **/
#ifdef noMotifGLwidget
#include <X11/GLw/GLwDrawA.h>   /* pure Xt OpenGL drawing area widget */
#else
#include <X11/GLw/GLwMDrawA.h>  /* Motif OpenGL drawing area widget */
#endif
#endif
#include <X11/keysym.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>

static int dblBuf[] = {
    GLX_DOUBLEBUFFER, GLX_RGBA, GLX_DEPTH_SIZE, 16,
    GLX_RED_SIZE, 1, GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1,
    None
};
static int *snglBuf = &dblBuf[1];
static String   fallbackResources[] = {
#ifdef IRIX_5_2_or_higher
    "*sgiMode: true",           /* try to enable IRIX 5.2+ look & feel */
    "*useSchemes: all",         /* and SGI schemes */
#endif
    "*title: OpenGL paper plane demo",
    "*glxarea*width: 300", "*glxarea*height: 300", NULL
};
Display     *dpy;
GLboolean    doubleBuffer = GL_TRUE, moving = GL_FALSE, made_current = GL_FALSE;
XtAppContext app;
XtWorkProcId workId = 0;
Widget       toplevel, mainw, menubar, menupane, btn, cascade, frame, glxarea;
GLXContext   cx;
```

```
XVisualInfo *vi;
#ifdef noGLwidget
Colormap     cmap;
#endif
Arg          menuPaneArgs[1], args[1];

#define MAX_PLANES 15

struct {
    float           speed;      /* zero speed means not flying */
    GLfloat         red, green, blue;
    float           theta;
    float           x, y, z, angle;
} planes[MAX_PLANES];

#define v3f glVertex3f /* v3f was the short IRIS GL name for glVertex3f */

void draw(Widget w)
{
    GLfloat         red, green, blue;
    int             i;

    glClear(GL_DEPTH_BUFFER_BIT);
    /* paint black to blue smooth shaded polygon for background */
    glDisable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glBegin(GL_POLYGON);
        glColor3f(0.0, 0.0, 0.0);
        v3f(-20, 20, -19); v3f(20, 20, -19);
        glColor3f(0.0, 0.0, 1.0);
        v3f(20, -20, -19); v3f(-20, -20, -19);
    glEnd();
    /* paint planes */
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
    for (i = 0; i < MAX_PLANES; i++)
        if (planes[i].speed != 0.0) {
            glPushMatrix();
                glTranslatef(planes[i].x, planes[i].y, planes[i].z);
                glRotatef(290.0, 1.0, 0.0, 0.0);
                glRotatef(planes[i].angle, 0.0, 0.0, 1.0);
                glScalef(1.0 / 3.0, 1.0 / 4.0, 1.0 / 4.0);
                glTranslatef(0.0, -4.0, -1.5);
                glBegin(GL_TRIANGLE_STRIP);
                    /* left wing */
                    v3f(-7.0, 0.0, 2.0); v3f(-1.0, 0.0, 3.0);
                    glColor3f(red = planes[i].red, green = planes[i].green,
                              blue = planes[i].blue);
                    v3f(-1.0, 7.0, 3.0);
                    /* left side */
                    glColor3f(0.6 * red, 0.6 * green, 0.6 * blue);
                    v3f(0.0, 0.0, 0.0); v3f(0.0, 8.0, 0.0);
                    /* right side */
                    v3f(1.0, 0.0, 3.0); v3f(1.0, 7.0, 3.0);
                    /* final tip of right wing */
                    glColor3f(red, green, blue);
                    v3f(7.0, 0.0, 2.0);
                glEnd();
            glPopMatrix();
        }
```

```
    if (doubleBuffer) glXSwapBuffers(dpy, XtWindow(w));
    if(!glXIsDirect(dpy, cx))
        glFinish(); /* avoid indirect rendering latency from queuing */
#ifdef DEBUG
    { /* for help debugging, report any OpenGL errors that occur per frame */
        GLenum error;
        while((error = glGetError()) != GL_NO_ERROR)
            fprintf(stderr, "GL error: %s\n", gluErrorString(error));
    }
#endif
}

void tick_per_plane(int i)
{
    float theta = planes[i].theta += planes[i].speed;
    planes[i].z = -9 + 4 * cos(theta);
    planes[i].x = 4 * sin(2 * theta);
    planes[i].y = sin(theta / 3.4) * 3;
    planes[i].angle = ((atan(2.0) + M_PI_2) * sin(theta) - M_PI_2) * 180 / M_PI;
    if (planes[i].speed < 0.0) planes[i].angle += 180;
}

void add_plane(void)
{
    int i;

    for (i = 0; i < MAX_PLANES; i++)
        if (planes[i].speed == 0) {

#define SET_COLOR(r,g,b) \
        planes[i].red=r; planes[i].green=g; planes[i].blue=b; break;

            switch (random() % 6) {
            case 0: SET_COLOR(1.0, 0.0, 0.0); /* red */
            case 1: SET_COLOR(1.0, 1.0, 1.0); /* white */
            case 2: SET_COLOR(0.0, 1.0, 0.0); /* green */
            case 3: SET_COLOR(1.0, 0.0, 1.0); /* magenta */
            case 4: SET_COLOR(1.0, 1.0, 0.0); /* yellow */
            case 5: SET_COLOR(0.0, 1.0, 1.0); /* cyan */
            }
            planes[i].speed = (random() % 20) * 0.001 + 0.02;
            if (random() & 0x1) planes[i].speed *= -1;
            planes[i].theta = ((float) (random() % 257)) * 0.1111;
            tick_per_plane(i);
            if (!moving) draw(glxarea);
            return;
        }
    XBell(dpy, 100); /* can’t add any more planes */
}

void remove_plane(void)
{
    int             i;

    for (i = MAX_PLANES - 1; i >= 0; i--)
        if (planes[i].speed != 0) {
            planes[i].speed = 0;
            if (!moving) draw(glxarea);
            return;
        }
```

```
    XBell(dpy, 100); /* no more planes to remove */
}

void resize(Widget w, XtPointer data, XtPointer callData)
{
    if(made_current) {
#ifdef noGLwidget
        Dimension width, height;

        /*
         * Silly drawing area resize callback doesn’t give
         * height and width via its parameters!
         */
        XtVaGetValues(w, XmNwidth, &width, XmNheight, &height, NULL);
        glViewport(0, 0, (GLint) width, (GLint) height);
#else
        GLwDrawingAreaCallbackStruct *resize =
            (GLwDrawingAreaCallbackStruct*) callData;

        glViewport(0, 0, (GLint) resize->width, (GLint) resize->height);
#endif
    }
}

void tick(void)
{
    int i;

    for (i = 0; i < MAX_PLANES; i++)
        if (planes[i].speed != 0.0) tick_per_plane(i);
}

Boolean animate(XtPointer data)
{
    tick();
    draw(glxarea);
    return False;              /* leave work proc active */
}

void toggle(void)
{
    moving = !moving; /* toggle */
    if (moving)
        workId = XtAppAddWorkProc(app, animate, NULL);
    else
        XtRemoveWorkProc(workId);
}

void quit(Widget w, XtPointer data, XtPointer callData)
{
    exit(0);
}

void input(Widget w, XtPointer data, XtPointer callData)
{
    XmDrawingAreaCallbackStruct *cd = (XmDrawingAreaCallbackStruct *) callData;
    char            buf[1];
    KeySym          keysym;
    int             rc;

```

```
    if(cd->event->type == KeyPress)
        if(XLookupString((XKeyEvent *) cd->event, buf, 1, &keysym, NULL) == 1)
            switch (keysym) {
            case XK_space:
                if (!moving) { /* advance one frame if not in motion */
                    tick();
                    draw(w);
                }
                break;
            case XK_Escape:
                exit(0);
            }
}

void map_state_changed(Widget w, XtPointer data, XEvent * event, Boolean * cont)
{
    switch (event->type) {
    case MapNotify:
        if (moving && workId != 0) workId = XtAppAddWorkProc(app, animate, NULL);
        break;
    case UnmapNotify:
        if (moving) XtRemoveWorkProc(workId);
        break;
    }
}

main(int argc, char *argv[])
{
    toplevel = XtAppInitialize(&app, "Paperplane", NULL, 0, &argc, argv,
                               fallbackResources, NULL, 0);
    dpy = XtDisplay(toplevel);
    /* find an OpenGL-capable RGB visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), dblBuf);
    if (vi == NULL) {
        vi = glXChooseVisual(dpy, DefaultScreen(dpy), snglBuf);
        if (vi == NULL)
            XtAppError(app, "no RGB visual with depth buffer");
        doubleBuffer = GL_FALSE;
    }
    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi, /* no display list sharing */ None,
        /* favor direct */ GL_TRUE);
    if (cx == NULL)
        XtAppError(app, "could not create rendering context");
    /* create an X colormap since probably not using default visual */
#ifdef noGLwidget
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    /*
     * Establish the visual, depth, and colormap of the toplevel
     * widget _before_ the widget is realized.
     */
    XtVaSetValues(toplevel, XtNvisual, vi->visual, XtNdepth, vi->depth,
                  XtNcolormap, cmap, NULL);
#endif
    XtAddEventHandler(toplevel, StructureNotifyMask, False,
                      map_state_changed, NULL);
    mainw = XmCreateMainWindow(toplevel, "mainw", NULL, 0);
    XtManageChild(mainw);
    /* create menu bar */
```

```
    menubar = XmCreateMenuBar(mainw, "menubar", NULL, 0);
    XtManageChild(menubar);
#ifdef noGLwidget
    /* Hack around Xt’s unfortunate default visual inheritance. */
    XtSetArg(menuPaneArgs[0], XmNvisual, vi->visual);
    menupane = XmCreatePulldownMenu(menubar, "menupane", menuPaneArgs, 1);
#else
    menupane = XmCreatePulldownMenu(menubar, "menupane", NULL, 0);
#endif
    btn = XmCreatePushButton(menupane, "Quit", NULL, 0);
    XtAddCallback(btn, XmNactivateCallback, quit, NULL);
    XtManageChild(btn);
    XtSetArg(args[0], XmNsubMenuId, menupane);
    cascade = XmCreateCascadeButton(menubar, "File", args, 1);
    XtManageChild(cascade);
#ifdef noGLwidget
    menupane = XmCreatePulldownMenu(menubar, "menupane", menuPaneArgs, 1);
#else
    menupane = XmCreatePulldownMenu(menubar, "menupane", NULL, 0);
#endif
    btn = XmCreateToggleButton(menupane, "Motion", NULL, 0);
    XtAddCallback(btn, XmNvalueChangedCallback, (XtCallbackProc)toggle, NULL);
    XtManageChild(btn);
    btn = XmCreatePushButton(menupane, "Add plane", NULL, 0);
    XtAddCallback(btn, XmNactivateCallback, (XtCallbackProc)add_plane, NULL);
    XtManageChild(btn);
    btn = XmCreatePushButton(menupane, "Remove plane", NULL, 0);
    XtAddCallback(btn, XmNactivateCallback, (XtCallbackProc)remove_plane, NULL);
    XtManageChild(btn);
    XtSetArg(args[0], XmNsubMenuId, menupane);
    cascade = XmCreateCascadeButton(menubar, "Planes", args, 1);
    XtManageChild(cascade);
    /* create framed drawing area for OpenGL rendering */
    frame = XmCreateFrame(mainw, "frame", NULL, 0);
    XtManageChild(frame);
#ifdef noGLwidget
    glxarea = XtVaCreateManagedWidget("glxarea", xmDrawingAreaWidgetClass,
                                      frame, NULL);
#else
#ifdef noMotifGLwidget
    /* notice glwDrawingAreaWidgetClass lacks an ’M’ */
    glxarea = XtVaCreateManagedWidget("glxarea", glwDrawingAreaWidgetClass,
#else
    glxarea = XtVaCreateManagedWidget("glxarea", glwMDrawingAreaWidgetClass,
#endif
                                      frame, GLwNvisualInfo, vi, NULL);
#endif
    XtAddCallback(glxarea, XmNexposeCallback, (XtCallbackProc)draw, NULL);
    XtAddCallback(glxarea, XmNresizeCallback, resize, NULL);
    XtAddCallback(glxarea, XmNinputCallback, input, NULL);
    /* set up application’s window layout */
    XmMainWindowSetAreas(mainw, menubar, NULL, NULL, NULL, frame);
    XtRealizeWidget(toplevel);
    /*
     * Once widget is realized (ie, associated with a created X window), we
     * can bind the OpenGL rendering context to the window.
     */
    glXMakeCurrent(dpy, XtWindow(glxarea), cx);
    made_current = GL_TRUE;
    /* setup OpenGL state */
```

```
    glClearDepth(1.0);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 20);
    glMatrixMode(GL_MODELVIEW);
    /* add three initial random planes */
    srandom(getpid());
    add_plane(); add_plane(); add_plane();
    /* start event processing */
    XtAppMainLoop(app);
}
```

References

[1] Tom Gaskins, “Using PEXlib with X Toolkits,” *PEXlib Programming Manual*, O’Reilly & Associates, Inc., 1992.

[2] Mark Kilgard, “OpenGL and X, Part 1: An Introduction,” *The X Journal*, SIGS Publications, Nov/Dec 1993.

[3] Mark Kilgard, “OpenGL and X, Part 2: Using OpenGL with Xlib,” *The X Journal*, SIGS Publications, Jan/Feb 1994.

[4] Silicon Graphics, *The OpenGL Porting Guide*, supplied with the IRIX 5.2 development option, 1994.

[5] Josie Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1994.

OpenGL Graphics with the X Window System

Phil Karlton
Revised by: Paula Womack

Copyright © 1992, 1993, 1994, 1995, 1996 Silicon Graphics, Inc.

This document contains unpublished information of Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

OpenGL is a registered trademark of Silicon Graphics, Inc.
X is a registered trademark of the Massachussetts Institute of Technology
Unix is a registered trademark of A T & T Bell Laboratories.

1 Overview

This document describes GLX, the OpenGL extension to the X Window System. It refers to concepts discussed in the OpenGL specification, and may be viewed as an X specific appendix to that document. Parts of the document assume some acquaintance with both the OpenGL and X.

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense: connection and authentication are accomplished with the normal X mechanisms. As with other X extensions, there is a defined network protocol for the OpenGL rendering commands encapsulated within the X byte stream.

Since performance is critical in 3D rendering, there is a way for OpenGL rendering to bypass the data encoding step, the data copying, and interpretation of that data by the X server. This *direct rendering* is possible only when a process has direct access to the graphics pipeline. Allowing for parallel rendering has affected the design of the GLX interface. This has resulted in an added burden on the client to explicitly prevent parallel execution when that is inappropriate.

X and the OpenGL have different conventions for naming entry points and macros. The GLX extension adopts those of the OpenGL.

2 GLX Operation

2.1 Rendering Contexts and Drawing Surfaces

The OpenGL specification is intentionally vague on how a rendering context (an abstract OpenGL state machine) is created. One of the purposes of GLX is to provide a means to create an OpenGL context and associate it with a drawing surface.

In X, a rendering surface is called a `Drawable`. `Window`s, one type of `Drawable`, are associated with a `Visual`.[*] The X protocol allows for a single `VisualID` to be instantiated at multiple depths. The GLX bindings allow only one depth for an OpenGL renderer for any given `VisualID`. In GLX the definition of `Visual` has been extended to include the types, quantities and sizes of the ancillary buffers (depth, accumulation, auxiliary, and stencil). Double buffering capability is also fixed by the `Visual`.[†] The ancillary buffers have no meaning within the core X environment. The set of extended `Visual`s is fixed at server startup time. One result is that a server can export multiple `Visual`s that differ only in the extended attributes.

The other type of X `Drawable` is a `Pixmap`, a drawing surface that is maintained off screen. The GLX equivalent to an X `Pixmap` is a `GLXPixmap`. A `GLXPixmap` is created using the `Visual` along with its extended attributes. The `Visual` is used to define the type and size of the Ancillary buffers associated with the `Pixmap`. The `Pixmap` is used as the front-left color buffer. A `GLXDrawable` is the union {`Window`, `GLXPixmap`}.

Ancillary buffers are associated with a `GLXDrawable`, not with a rendering context. If several OpenGL renderers are all writing to the same window, they will share those buffers. Rendering operations to one window never affect the unobscured pixels of another window, or of the corresponding pixels of ancillary buffers of that window. If an `Expose` event is received by the client, the values in the ancillary buffers and in the back buffers for regions corresponding to the exposed region become undefined.

---

[*] The association is with a {`Visual`, `screen`, `depth`} triple. An `XVisualInfo` is used by GLX functions since it can be interpreted unambiguosly.

[†] Any rendering system is free to use the ancillary buffers as long as it uses them in a manner consistent with the use by the OpenGL.

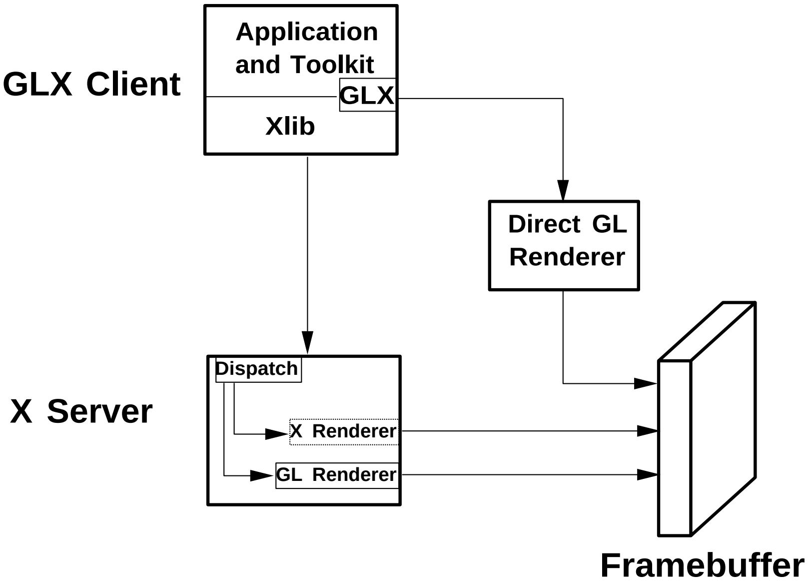


Figure 1: Direct Rendering Block Diagram.

A rendering context can be used with multiple `GLXDrawable`s as long as those `Drawable`s are *similar*. Similar means that the rendering contexts and `GLXDrawable`s are created with the same `XVisualInfo`.

An application can use any rendering context (subject to the restrictions discussed in the section on address spaces) to render into any similar `GLXDrawable`. An implication is that multiple applications can render into the same window, each using a different rendering context.

2.2 Using Rendering Contexts

OpenGL defines both client state and server state. Thus a rendering context consists of two parts: one to hold the client state and one to hold the server state. The client is responsible for creating a rendering context and a drawable; defaults are not supplied.

Each thread can have at most one current rendering context. In addition, a rendering context can be current for only one thread at one time.

Issuing OpenGL commands may cause the X buffer to be flushed. In particular, calling **glFlush()** will flush both the X and OpenGL rendering streams.

Some state is shared between the OpenGL and X. The pixel values in the X frame buffer are shared. The X double buffer extension (DBE) has a definition for which buffer is currently the displayed buffer. This information is shared with GLX. The state of which buffer is displayed tracks in both extensions, independent of which extension initiates a buffer swap.

2.3 Direct Rendering and Address Spaces

One of the basic assumptions of the X protocol is that if a client can name an object, then it can manipulate that object. GLX introduces the notion of an *Address Space*. A GLX object cannot be used outside of the address space in which it exists.

In a classic UNIX environment, each process is in its own address space. In a multi-threaded environment, each of the threads will share a virtual address space which references a common data region.

An OpenGL client that is rendering to a graphics engine directly connected to the executing CPU may avoid passing the tokens through the X server. This generalization is made for performance reasons. The model described here specifically allows for such optimizations, but does not mandate that any implementation support it.

When direct rendering is occurring, the address space of the renderer is that of the direct process; when direct rendering is not being used, the address space of the renderer is that of the X server. The client has the ability to reject the use of direct rendering, but there may be a performance penalty in doing so.

In order to use direct rendering, a client must create a direct rendering context. Both the client context state and the server context state of a direct rendering context exist in the client's address space; this state cannot be shared by a client in another process. With indirect rendering contexts, the client context state is kept in the client's address space and the server context state is kept in the address space of the X server. In this case the server context state is stored in an X resource; it has an associated XID and may potentially be used by another client process.

2.4 OpenGL Display Lists

Most OpenGL state is small and easily retrieved using the **glGet*** commands. This is not true of OpenGL display lists, which are used, for example, to encapsulate a model of some physical object. First, there is no mechanism to obtain the contents of a display list from the rendering context. Second, display lists may be large and numerous. It may be desirable for multiple rendering contexts to share display lists rather than replicating that information in each context.

GLX provides for limited sharing of display lists; the lists can be shared only if the server state for the contexts share a single address space. Using this mechanism, a single set of lists can be used, for instance, by a context that supports color index rendering and a context that supports RGBA rendering.

A group of shared display lists exists until the last referencing rendering context is destroyed. All rendering contexts have equal access to using lists or defining new lists. Implementations sharing contexts must handle the case where one rendering context is using a display list when another rendering context destroys that list.

When display lists are shared between OpenGL contexts, the sharing extends only to the display lists themselves and the information about which display list numbers have been allocated. In particular, the value of the base set with **glListBase** is not shared.

In general, OpenGL commands are not atomic. **glEndList** and **glDeleteLists** are exceptions. The list named in a **glNewList** call is not created or superseded until **glEndList** is called. If one rendering context is sharing a display list with another, it will continue to use the existing definition while the second context is in the process of re-defining it.

2.5 Texture Objects

OpenGL texture state can be encapsulated in a named texture object. A texture object is created by binding an unused name to one of the texture targets (TEXTURE_1D or TEXTURE_2D) of a rendering context. When a texture object is bound, OpenGL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object.

Texture objects may be shared by rendering contexts, as long as the server portion of the contexts share the same address space. OpenGL makes no attempt to synchronize access to texture objects. If a texture object is bound to more than one context, then it is up to the programmer to ensure that the contents of the object are not being changed via one context while another context is using the texture object for rendering. The results of changing a texture object while another context is using it are undefined.

A texture object will not be deleted until it is no longer bound to any rendering context.

2.6 Aligning Multiple Drawables

A client can create one window with an overlay `Visual` and a second with a main plane `Visual` and then move them independently or in concert to keep them aligned. This is a major change between the OpenGL and the previous SGI proprietary GL: allocation of overlay planes and main planes for every window is no longer done automatically. To accomplish what was done by a **drawmode/gconfig** pair in previous versions of the SGI proprietary GL, the OpenGL client can use the following paradigm:

- Make the windows which are to share the same screen area children of a single window (that will never be written). Size and position the children to completely occlude their parent. When the window combination must be moved or resized, perform the operation on the parent.
- Make the subwindows have a background of `None` so that the X server will not paint into the shared area when you restack the children.
- Select for device-related events on the parent window, not on the children. Since device-related events with the focus in one of the child windows will be inherited by the parent, input dispatching can be done directly without reference to the child on top.

2.7 Multiple Threads

It is possible to create a version of the client side library that is protected against multiple threads attempting to access the same connection. This is accomplished by having appropriate definitions for **LockDisplay** and **UnlockDisplay**. Since there is some performance penalty for doing the locking, it is implementation-dependent whether a thread safe version, a non-safe version, or both versions of the library are provided. Interrupt routines may not share a connection (and hence a rendering context) with the main thread. An application may be written as a set of co-operating processes.

X has atomicity (between clients) and sequentiality (within a single client) requirements that limit the amount of parallelism achievable when interpreting the command streams. GLX relaxes these requirements. Sequentiality is still guaranteed within a command stream, but not between the X and the OpenGL command streams. It is possible, for example, that an X command issued by a single threaded client after an OpenGL command might be executed before that OpenGL command.

The X specification requires that commands are atomic:

> If a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams).

OpenGL commands are not guaranteed to be atomic. Some OpenGL rendering commands might otherwise impair interactive use of the windowing system by the user. For instance calling a deeply nested display list or rendering a large texture mapped polygon on a system with no graphics hardware could prevent a user from popping up a menu soon enough to be usable.

Synchronization is in the hands of the client. It can be maintained with moderate cost with the judicious use of the **glFinish**, **glXWaitGL**, **glXWaitX**, and **XSync** commands. OpenGL and X rendering can be done in parallel as long as the client does not preclude it with explicit synchronization calls. This is true even when the rendering is being done by the X server. Thus, a multi-threaded X server implementation may execute OpenGL rendering commands in parallel with other X requests.

Some performance degradation may be experienced if needless switching between OpenGL and X rendering is done. This may involve a round trip to the server, which can be costly.

3 Functions and Errors

3.1 Errors

Where possible, as in X, when a request terminates with an error, the request has no side effects.

The error codes that may be generated by a request are described with that request. The following table summarizes the GLX-specific error codes that are visible to applications:

`GLXBadContext` A value for a `Context` argument does not name a `Context`.

`GLXBadContextState` An attempt was made to switch to another rendering context while the current context was in `RenderMode` `GL_FEEDBACK` or `GL_SELECT`, or a call to **glXMakeCurrent** was made between a **glBegin** and the corresponding call to **glEnd**.

`GLXBadCurrentWindow` The current `Drawable` of the calling thread is a window that is no longer valid.

`GLXBadDrawable` The `Drawable` argument does not name a `Drawable` configured for OpenGL rendering.

`GLXBadPixmap` The `Pixmap` argument does not name a `Pixmap` that is appropriate for OpenGL rendering.

`GLXUnsupportedPrivateRequest` May be returned in response to either a glXVendorPrivate request or a glXVendorPrivateWithReply request.

The following error codes may be generated by a faulty GLX implementation, but would not normally be visible to clients:

`GLXBadContextTag` A rendering request contains an invalid context tag. (Context tags are used to identify contexts in the protocol.)

`GLXBadRenderRequest` A glXRender request is ill-formed.

`GLXBadLargeRequest` A glXRenderLarge request is ill-formed.

3.2 Functions

GLX functions should not be called between **glBegin** and **glEnd** operations. If a GLX function is called within a **glBegin**/**glEnd** pair, then the result is undefined; however, no error is reported.

3.2.1 Initialization

To ascertain if the GLX extension is defined for an X server, use

`Bool` **glXQueryExtension**`(  Display` **dpy*, `int` **error_base*, `int` **event_base* `)`
`;`

dpy specifies the connection to the X server. `False` is returned if the extension is not present. *error_base* is used to return the value of the first error code. The constant error codes should be added to this base to get the actual value.

event_base is included for future extension. GLX does not currently define any events.

The GLX definition exists in multiple versions. Use

`Bool` **glXQueryVersion**`(  Display` **dpy*, `int` **major*, `int` **minor* `) ;`

to discover which version of GLX is available. Upon success, *major* and *minor* are filled in with the major and minor versions of the extension implementation. If the client and server both have the same major version number then they are compatible and the minor version that is returned is the minimum of the two minor version numbers.

major and *minor* do not return values if they are specified as **NULL**.

glXQueryVersion returns **True** if it succeeds and **False** if it fails. If it fails, *major* and *minor* are not updated.

3.2.2 Configuration Management

The constants shown in Table 1 are passed to **glXGetConfig** and **glXChooseVisual** to specify which attributes are being queried.

`GLX_BUFFER_SIZE` gives the total depth of the color buffer in bits. For **PseudoColor** and **StaticColor** visuals, this is equal to the depth value reported in the core X11 Visual. For **TrueColor** and **DirectColor** visuals, `GLX_BUFFER_SIZE` is the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, `GLX_BLUE_SIZE`, and `GLX_ALPHA_SIZE`. Note that this value may be larger than the depth value reported in the core X11 visual since it may include alpha planes that may not be reported by X11. Also, for `TrueColor` visuals, the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, and `GLX_BLUE_SIZE` may be larger than the maximum depth that core X11 can support.

To obtain a description of an OpenGL attribute exported by a `Visual` use

`int` **glXGetConfig**`(  Display` **dpy*, `XVisualInfo*` **visual*, `int` *attribute*, `int`
**value* `) ;`

glXGetConfig returns through *value* the value of the *attribute* of *visual*.

glXGetConfig returns one of the following error codes if it fails, and Success otherwise:

`GLX_NO_EXTENSION` *dpy* does not support the GLX extension.

`GLX_BAD_SCREEN` screen of *visual* does not correspond to a screen.

`GLX_BAD_ATTRIBUTE` *attribute* is not a valid GLX attribute.

`GLX_BAD_VISUAL` *visual* does not support GLX and an attribute other than `GLX_USE_GL` was specified.

`GLX_BAD_VALUE` parameter invalid

Attribute	Type	Notes
`GLX_USE_GL`	boolean	**True** if OpenGL rendering supported
`GLX_BUFFER_SIZE`	integer	depth of the color buffer
`GLX_LEVEL`	integer	frame buffer level
`GLX_RGBA`	boolean	**True** if RGBA rendering supported
`GLX_DOUBLEBUFFER`	boolean	**True** if color buffers have front/back pairs
`GLX_STEREO`	boolean	**True** if color buffers have left/right pairs
`GLX_AUX_BUFFERS`	integer	number of auxiliary color buffers
`GLX_RED_SIZE`	integer	number of bits of Red in the framebuffer
`GLX_GREEN_SIZE`	integer	number of bits of Green in the framebuffer
`GLX_BLUE_SIZE`	integer	number of bits of Blue in the framebuffer
`GLX_ALPHA_SIZE`	integer	number of bits in the destination alpha buffer
`GLX_DEPTH_SIZE`	integer	number of bits in the depth buffer
`GLX_STENCIL_SIZE`	integer	number of bits in the stencil buffer
`GLX_ACCUM_RED_SIZE`	integer	number Red bits in the accumulation buffer
`GLX_ACCUM_GREEN_SIZE`	integer	number Green bits in the accumulation buffer
`GLX_ACCUM_BLUE_SIZE`	integer	number Blue bits in the accumulation buffer
`GLX_ACCUM_ALPHA_SIZE`	integer	number Alpha bits in the accumulation buffer

Table 1: Configuration attributes.

A GLX implementation may export many visuals that support OpenGL. These visuals support either color index or RGBA rendering. Currently RGBA rendering can be supported only by Visuals of type **TrueColor** or **DirectColor** and color index rendering can be supported only by Visuals of type **PseudoColor** or **StaticColor**.

Servers are required to export at least one visual that supports RGBA rendering. At least one of the visuals that supports RGBA rendering must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer; alpha bitplanes are optional. The color buffer size for this visual must be as large as that of the deepest **TrueColor**, **DirectColor**, **PseudoColor**, or **StaticColor** visual supported on framebuffer level zero (the main image planes), and it must be available on framebuffer level zero.

If the X server exports a **PseudoColor** or **StaticColor** visual on framebuffer level 0, a visual that supports color index rendering is also required. If color index rendering is supported then one of the visuals that supports color index rendering must have at least one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. It also must have as many color bitplanes as the deepest **PseudoColor** or **StaticColor** visual supported on framebuffer level zero, and it must itself be made available on level zero.

glXChooseVisual is used to find a visual that matches the client's specified attributes.

`XVisualInfo*` **glXChooseVisual**`(` `Display` **dpy*, `int` *screen*, `int` **attrib_list*
`) ;`

glXChooseVisual returns a pointer to an `XVisualInfo` structure describing the visual that best matches the specified attributes. If no matching visual exists, **NULL** is returned.

The attributes are matched in an attribute-specific manner, as shown in Table 2. Some of the attributes, such as `GLX_LEVEL`, must match the specified value exactly; others, such as, `GLX_BUFFER_SIZE` and `GLX_RED_SIZE` must meet or exceed the specified minimum values. In the case of `GLX_BUFFER_SIZE`,

Attribute	Default	Selection Criteria
`GLX_USE_GL`	**True**	exact
`GLX_BUFFER_SIZE`	0	minimum, smallest
`GLX_LEVEL`	0	exact
`GLX_RGBA`	**False**	exact
`GLX_DOUBLEBUFFER`	**False**	exact
`GLX_STEREO`	**False**	exact
`GLX_AUX_BUFFERS`	0	minimum, smallest
`GLX_RED_SIZE`	0	minimum, largest
`GLX_GREEN_SIZE`	0	minimum, largest
`GLX_BLUE_SIZE`	0	minimum, largest
`GLX_ALPHA_SIZE`	0	minimum, largest
`GLX_DEPTH_SIZE`	0	minimum, largest
`GLX_STENCIL_SIZE`	0	minimum, smallest
`GLX_ACCUM_RED_SIZE`	0	minimum, largest
`GLX_ACCUM_GREEN_SIZE`	0	minimum, largest
`GLX_ACCUM_BLUE_SIZE`	0	minimum, largest
`GLX_ACCUM_ALPHA_SIZE`	0	minimum, largest

Table 2: Defaults and selection criteria used by **glXChooseVisual**.

preference is given based on how close the visual's attribute value is to the specified value. (Attributes that are matched in this manner have `minimum, smallest` listed as their selection criteria in Table 2.) In the case of `GLX_RED_SIZE`, if the specified value is non-zero, then preference is given to visuals with the largest value for this attribute; otherwise preference is given to visuals with the smallest value. (Attributes that are matched in this manner have `minimum, largest` listed as their selection criteria in Table 2.)

If `GLX_RGBA` is in *attrib_list* then the resulting visual will be `TrueColor` or `DirectColor`. If all other attributes are equivalent, then a `TrueColor` visual will be chosen in preference to a `DirectColor` visual.

If `GLX_RGBA` is not in *attrib_list* then the returned visual will be `PseudoColor` or `StaticColor`. If all other attributes are equivalent then a `PseudoColor` visual will be chosen in preference to a `StaticColor` visual.

If an attribute is not specified in *attrib_list*, then the default value is used. See Table 2 for a list of defaults.

Default specifications are superseded by the attributes included in *attrib_list*. Integer attributes are immediately followed by the corresponding desired value. Boolean attributes appearing in *attrib_list* have an implicit **True** value; such attributes are *never* followed by an explicit **True** or **False** value. The list is terminated with `None`.

To free the data returned, use **XFree**.

NULL is returned if an undefined GLX attribute is encountered.

3.2.3 Off Screen Rendering

To create an off screen rendering area, first create an X `Pixmap` of the depth specified by the desired `Visual`, then call

`GLXPixmap` **glXCreateGLXPixmap**`(` `Display` **dpy*, `XVisualInfo*` *visual*, `Pixmap` *pixmap* `) ;`

glXCreateGLXPixmap creates an off screen rendering area and returns its XID. Any GLX rendering context created with respect to *visual* can be used to render into this off screen area.

pixmap is used for the RGB planes of the front-left buffer of the resulting GLX off screen rendering area. The alpha buffer and ancillary buffers specified by *visual* are created without externally visible names. GLX pixmaps may be created with a *visual* that includes back buffers and stereoscopic buffers. However, **glXSwapBuffers** is ignored for these pixmaps.

A direct rendering context might not be able to be made current with a `GLXPixmap`.

If the depth of *pixmap* does not match the depth value reported by core X11 for *visual*, or if *pixmap* was not created with respect to the same screen as *visual*, then a `BadMatch` error is generated. If *visual* is not valid (e.g., if GLX does not support it), then a `BadValue` error is generated. If *pixmap* is not a valid pixmap id, then a `BadPixmap` error is generated. Finally, if the server cannot allocate the new GLX pixmap, a `BadAlloc` error is generated.

A `GLXPixmap` is destroyed by calling

`void` **glXDestroyGLXPixmap**`(` `Display` **dpy*, `GLXPixmap` *pixmap* `) ;`

This request deletes the association between the resource ID *pixmap* and the GLX pixmap. The storage will be freed when it is not current to any client.

If *pixmap* is not a valid GLX pixmap then a `GLXBadPixmap` error is generated.

3.2.4 Rendering Contexts

To create an OpenGL rendering context call

`GLXContext` **glXCreateContext**`(` `Display` **dpy*, `XVisualInfo*` *visual*, `GLXContext` *share_list*, `Bool` *direct* `) ;`

glXCreateContext returns **NULL** if it fails. If **glXCreateContext** succeeds, it initializes the rendering context to the default OpenGL state and returns a handle to it. This handle can be used to render to both windows and GLX pixmaps.

If *share_list* is not **NULL**, then all display lists and texture objects except texture objects named 0 will be shared by *share_list* and the newly created rendering context. An arbitrary number of **GLXContexts** can share a single display list and texture object space. All sharing contexts must also share a single address space or a `BadMatch` error is generated.

If *direct* is true, then a direct rendering context will be created if the implementation supports direct rendering and the connection is to an X server that is local. If *direct* is **False**, then a rendering context that renders through the X server is created.

Direct rendering contexts may be a scarce resource in some implementations. If *direct* is true, and if a direct rendering context cannot be created, then **glXCreateContext** will attempt to create an indirect context instead.

glXCreateContext can generate the following GLX extension errors: `GLXBadContext` if *share_list* is neither zero nor a valid GLX rendering context; `BadValue` if *visual* is not a valid X Visual or if GLX does not support it; `BadMatch` if *share_list* defines an address space that cannot be shared with the newly created context or if *share_list* was created on a different screen than the one referenced by *visual*; `BadAlloc` if the server does not have enough resources to allocate the new context.

To determine if an OpenGL rendering context is direct call

`Bool` **glXIsDirect**`(` `Display` **dpy*, `GLXContext` *ctx* `) ;`

glXIsDirect returns **True** if *ctx* is a direct rendering context, **False** otherwise. If *ctx* is not a valid GLX rendering context, a `GLXBadContext` error is generated.

An OpenGL rendering context is destroyed by calling

`void` **glXDestroyContext**`(` `Display` **dpy*, `GLXContext` *ctx* `) ;`

If *ctx* is still current to any thread, *ctx* is not destroyed until it is no longer current. In any event, the associated XID will be destroyed and *ctx* cannot subsequently be made current to any thread.

glXDestroyContext will generate a `GLXBadContext` error if *ctx* is not a valid rendering context.

To copy OpenGL rendering state from one context to another, use

`void` **glXCopyContext**`(` `Display` **dpy*, `GLXContext` *source*, `GLXContext` *dest*, `unsigned long` *mask* `) ;`

glXCopyContext copies selected groups of state variables from *source* to *dest*. *mask* indicates which groups of state variables are to be copied; it contains the bitwise OR of the symbolic names for the attribute groups. The symbolic names are the same as those used by **glPushAttrib**, described in the OpenGL Specification. Also, the order in which the attributes are copied to *dest* as a result of the **glXCopyContext** operation is the same as the order in which they are popped off of the stack when **glPopAttrib** is called. The single symbolic constant `GL_ALL_ATTRIB_BITS` can be used to copy the maximum possible portion of the rendering state. It is not an error to specify *mask* bits that are undefined.

If *source* and *dest* do not share an address space or were not created on the same screen, a `BadMatch` error is generated. (*source* and *dest* may be based on different X visuals and still share an address space; **glXCopyContext** will work correctly in such cases.) If the destination context is current for some thread then a `BadAccess` error is generated. If the source context is the same as the current context of the calling thread, and the current drawable of the calling thread is a window that is no longer valid, a `GLXBadCurrentWindow` is generated. Finally, if either *source* or *dest* is not a valid GLX rendering context, a `GLXBadContext` error is generated.

glXCopyContext performs an implicit **glFlush()** if *source* is the current context for the calling thread.

Only one rendering context may be in use, or *current*, for a particular thread at a given time. The minimum number of current rendering contexts that must be supported by a GLX implementation is one. (Supporting a larger number of current rendering contexts is essential for general-purpose systems, but may not be necessary for turnkey applications.)

To make a context current, call

`Bool` **glXMakeCurrent**`(` `Display` **dpy*, `GLXDrawable` *drawable*, `GLXContext` *ctx* `) ;`

If the calling thread already has a current rendering context, then that context is flushed and marked as no longer current. *ctx* is made the current context for the calling thread.

If the *drawable* and *ctx* are not similar, a `BadMatch` error is generated. If *ctx* is current to some other thread, then **glXMakeCurrent** will generate a `BadAccess` error. `GLXBadContextState` is generated if there is a current rendering context and its render mode is either **GL_FEEDBACK** or **GL_SELECT**. `GLXBadContextState` will also be generated if **glXMakeCurrent** is called between a **glBegin** and its corresponding **glEnd**. If *ctx* is not a valid GLX rendering context, `GLXBadContext` is generated. If *drawable* is not a valid GLX drawable, a `GLXBadDrawable` error is generated. If the previous context of the calling thread has unflushed commands, and the previous drawable is a window that is no longer valid, `GLXBadCurrentWindow` is generated. Finally, note that the ancillary buffers for *drawable* need not be allocated until they are needed. A `BadAlloc` error will be generated if the server does not have enough resources to allocate the buffers.

If *drawable* is destroyed after **glXMakeCurrent** is called then subsequent rendering commands will behave as if *drawable* is bound to the NULL clip. The commands will be processed and the context state will be updated, but no output will appear on the display.

To release the current context without assigning a new one, use `NULL` for *ctx* and `None` for *drawable*. If *ctx* is `NULL` and *drawable* is not `None`, or if *drawable* is `None` and *ctx* is not `NULL`, then a `BadMatch` error will be generated.

The first time *ctx* is made current to a `GLXDrawable`, its initial viewport is set. That viewport must be reset by the client when *ctx* is subsequently made current.

Note that when multiple threads are using their current contexts to render to the same drawable, OpenGL does not guarantee atomicity of fragment update operations. In particular, programmers may not assume that depth-buffering will automatically work correctly; there is a race condition between threads that read and update the depth buffer. Clients are responsible for avoiding this condition. They may use vendor-specific extensions or they may arrange for separate threads to draw in disjoint regions of the framebuffer, for example.

glXGetCurrentContext returns the current context.

`GLXContext` **glXGetCurrentContext**`( void ) ;`

If there is no current context, **NULL** is returned.

glXGetCurrentDrawable returns the XID of the current drawable.

`GLXDrawable` **glXGetCurrentDrawable**`( void ) ;`

If there is no current drawable, `None` is returned.

To get the display associated with the current context and drawable, call

`Display*` **glXGetCurrentDisplay**`( void ) ;`

If there is no current context, **NULL** is returned. This routine is available only if the GLX version is 1.2 or later.

glXGet* calls retrieve client-side state and do not force a round trip to the X server. Unlike most X calls (including the **glXQuery*** calls) that return a value, these calls do not flush any pending requests.

3.2.5 Synchronization Primitives

To prevent X requests from executing until any outstanding OpenGL rendering is done, call

`void` **glXWaitGL**`( void ) ;`

OpenGL calls made prior to **glXWaitGL** are guaranteed to be executed before X rendering calls made after **glXWaitGL**. While the same result can be achieved using **glFinish**, **glXWaitGL** does not require a round trip to the server, and is therefore more efficient in cases where the client and server are on separate machines.

glXWaitGL is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

To prevent the OpenGL command sequence from executing until any outstanding X requests are completed, call `void` **glXWaitX**`( void ) ;`

X rendering calls made prior to **glXWaitX** are guaranteed to be executed before OpenGL rendering calls made after **glXWaitX**. While the same result can be achieved using **XSync**, **glXWaitX** does not require a round trip to the server, and may therefore be more efficient.

glXWaitX is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

3.2.6 Double Buffering

For drawables that are double buffered, the contents of the back buffer can be made potentially visible (i.e., become the contents of the front buffer) by calling

`void` **glXSwapBuffers** `( Display` **dpy*, `GLXDrawable` *drawable* `) ;`

The contents of the back buffer then become undefined. This operation is a no-op if *drawable* was created with a non-double-buffered visual, or if *drawable* is a `GLXPixmap`.

All GLX rendering contexts share the same notion of which are front buffers and which are back buffers for a given drawable. This notion is also shared with the X double buffer extension (DBE).

When multiple threads are rendering to the same drawable, only one of them need call `glXSwapBuffers` and all of them will see the effect of the swap. The client must synchronize the threads that perform the swap and the rendering, using some means outside the scope of GLX, to insure that each new frame is completely rendered before it is made visible.

If *dpy* and *drawable* are the display and drawable for the calling thread's current context, `glXSwapBuffers` performs an implicit `glFlush()`. Subsequent OpenGL commands can be issued immediately, but will not be executed until the buffer swapping has completed, typically during vertical retrace of the display monitor.

If *drawable* is not a valid GLX drawable, **glXSwapBuffers** generates a `GLXBadDrawable` error. If *dpy* and *drawable* are the display and drawable associated with the calling thread's current context, and if *drawable* is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

3.2.7 Access to X Fonts

A shortcut for using X fonts is provided by the command

`void` **glXUseXFont**`( Font` *font*, `int` *first*, `int` *count*, `int` *list_base* `) ;`

count display lists are defined starting at *list_base*, each list consisting of a single call on **glBitmap**. The definition of bitmap *list_base* + i is taken from the glyph *first* + i of *font*. If a glyph is not defined, then an empty display list is constructed for it. The `width`, `height`, `xorig`, and `yorig` of the constructed bitmap are computed from the font metrics as `rbearing-lbearing`, `ascent+descent`, `-lbearing`, and `descent-1` respectively. `xmove` is taken from the `width` metric and `ymove` is set to zero.

Note that in the direct rendering case, this requires that the bitmaps be copied to the client's address space.

glXUseXFont performs an implicit `glFlush()`.

glXUseXFont is ignored if there is no current GLX rendering context. `BadFont` is generated if *font* is not a valid X font id. `GLXBadContextState` is generated if the current GLX rendering context is in display list construction mode. `GLXBadCurrentWindow` is generated if the drawable associated with the calling thread's current context is a window and is no longer valid.

3.2.8 GLX Versioning

The following functions are available only if the GLX version is 1.1 or later.

`const char*` **glXQueryExtensionsString**`(` `Display` **dpy*, `int` *screen* `) ;`

glXQueryExtensionsString returns a pointer to a string describing which GLX extensions are supported on the connection. The string is zero-terminated and contains a space-seperated list of extension names. The extension names themselves do not contain spaces. If there are no extensions to GLX, then the empty string is returned.

`const char*` **glXGetClientString**`(` `Display` **dpy*, `int` *name* `);`

glXGetClientString returns a pointer to a static, zero-terminated string describing some aspect of the client library. The possible values for *name* are `GLX_VENDOR`, `GLX_VERSION`, and `GLX_EXTENSIONS`. If *name* is not set to one of these values then **NULL** is returned. The format and contents of the vendor string is implementation dependent, and the format of the extension string is the same as for **glXQueryExtensionsString**. The version string is laid out as follows:

<major_version.minor_version><space><vendor-specific info>

Both the major and minor portions of the version number are of arbitrary length. The vendor-specific information is optional. However, if it is present, the format and contents are implementation specific.

`const char*` **glXQueryServerString**`(` `Display` **dpy*, `int` *screen*, `int` *name* `)`
`;`

glXQueryServerString returns a pointer to a static, zero-terminated string describing some aspect of the server's GLX extension. The possible values for *name* and the format of the strings is the same as for **glXGetClientString**. If *name* is not set to a recognized value then **NULL** is returned.

4 Encoding on the X Byte Stream

In the remote rendering case, the overhead associated with interpreting the GLX extension requests must be minimized. For this reason, all commands have been broken up into two categories: OpenGL and GLX commands that are each implemented as a single X extension request and OpenGL rendering requests that are batched within a `GLXRender` request.

4.1 Requests that hold a single extension request

Each of the commands from glx.h (that is, the **glX*** commands) is encoded by a separate X extension request. In addition, there is a separate X extension request for each of the OpenGL commands that cannot be put into a display list. That list consists of all the **glGet*** commands plus

glAreTexturesResident
glDeleteLists
glDeleteTextures
glEndList
glFeedbackBuffer
glFinish

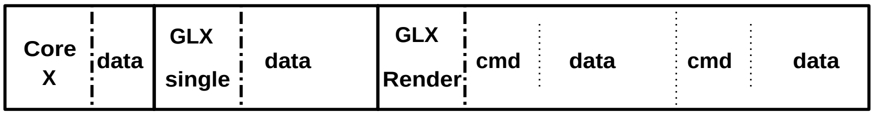


Figure 2: GLX byte stream.

glFlush
glGenLists
glGenTextures
glIsEnabled
glIsList
glIsTexture
glNewList
glPixelStoref
glPixelStorei
glReadPixels
glRenderMode
glSelectBuffer

The two **PixelStore** commands (**glPixelStorei** and **glPixelStoref**) are exceptions. These commands are issued to the server only to allow it to set its error state appropriately. Pixel storage state is maintained entirely on the client side. When pixel data is transmitted to the server (by **glDrawPixels**, for example), the pixel storage information that describes it is transmitted as part of the same protocol request. Implementations may not change this behavior, because such changes would cause shared contexts to behave incorrectly.

4.2 Request that holds multiple OpenGL commands

The remaining OpenGL commands are those that may be put into display lists. Multiple occurrences of these commands are grouped together into a single X extension request (**GLXRender**). This is diagrammed in Figure 4.2.

The grouping minimizes dispatching within the X server. The library packs as many OpenGL commands as possible into a single X request (without exceeding the maximum size limit). No OpenGL command may be split across multiple **GLXRender** requests.

For long OpenGL commands (those longer than a maximum X request size), a series of **GLXRenderLarge** commands is issued. The structure of the OpenGL command within **GLXRenderLarge** is the same as for **GLXRender**.

Note that it is legal to have a **glBegin** in one request, followed by **glVertex** commands, and eventually the matching **glEnd** in a subsequent request. A command is not the same as an OpenGL primitive.

4.3 Wire representations and byte swapping

Unsigned and signed integers are represented as they are represented in the core X protocol. Single and double precision floating point numbers are sent and received in IEEE floating point format. The X byte stream and network specifications make it impossible for the client to assure that double precision floating point numbers will be naturally aligned within the transport buffers of the server. For those architectures that require it, the server or client must copy those floating point numbers to a properly aligned buffer before using them.

Byte swapping on the encapsulated OpenGL byte stream is performed by the server using the same rule as the core X protocol. Single precision floating point values are swapped in the same way that 32-bit integers are swapped. Double precision floating point values are swapped across all 8 bytes.

4.4 Sequentiality

There are two sequences of commands: the X stream, and the OpenGL stream. In general these two streams are independent: Although the commands in each stream will be processed in sequence, there is no guarantee that commands in the separate streams will be processed in the order in which they were issued by the calling thread.

An exception to this rule arises when a single command appears in *both* streams. This forces the two streams to rendezvous.

Because the processing of the two streams may take place at different rates, and some operations may depend on the results of commands in a different stream, we distinguish between commands assigned to each of the X and OpenGL streams.

The following commands are processed on the client side and therefore do not exist in either the X or the OpenGL stream:

glXGetClientString
glXGetCurrentContext
glXGetCurrentDisplay
glXGetCurrentDrawable
glXGetConfig

The following commands are in the X stream and obey the sequentiality guarantees for X requests:

glXCreateContext
glXDestroyContext
glXMakeCurrent
glXIsDirect
glXQueryExtensionsString
glXQueryServerString
glXQueryVersion
glXWaitGL
glXCreateGLXPixmap
glXDestroyGLXPixmap
glXChooseVisual
glXSwapBuffers (but see below)
glXCopyContext (see below)

glXSwapBuffers is in the X stream if and only if the display and drawable are not those belonging to the calling thread's current context; otherwise it is in the OpenGL stream. **glXCopyContext** is in the X stream alone if and only if its source context differs from the calling thread's current context; otherwise it is in both streams.

Commands in the OpenGL stream, which obey the sequentiality guarantees for OpenGL requests are:

glXWaitX
glXSwapBuffers (see below)
All OpenGL Commands

glXSwapBuffers is in the OpenGL stream if and only if the display and drawable are those belonging to the calling thread's current context; otherwise it is in the X stream.

Commands in both streams, which force a rendezvous are:

glXCopyContext (see below)
glXUseXFont

glXCopyContext is in both streams if and only if the source context is the same as the current context of the calling thread; otherwise it is in the X stream only.

5 Extending OpenGL

OpenGL is extended by adding new GLX requests, OpenGL requests or additional enumerated values to the OpenGL requests. The OpenGL Architectural Review Board maintains a registry of indexes for each vendor to use as they wish.

New names must clearly indicate to clients whether some particular feature is in the core OpenGL or is vendor specific. To make a vendor-specific name, append a company identifier (in upper case) and any additional vendor-specific tags (e.g. machine names). For instance, SGI might add new commands and manifest constants of the form **glNewCommandSGI** and **GL_NEW_DEFINITION_SGI**. If SGI wanted to provide extensions that were specific to its Reality Engine, then the names might be of the form **glNewCommandSGIre** and **GL_NEW_DEFINITION_SGI_RE**. If two or more licensees agree in good faith to implement the same extension, and to make the specification of that extension publicly available, the procedures and tokens that are defined by the extension can be suffixed by **EXT**.

6 Glossary

Address Space the set of objects or memory locations accessible through a single name space. In other words, it is a data region that one or more processes may share through pointers.

Client an X client. An application communicates to a server by some path. The application program is referred to as a client of the window system server. To the server, the client is the communication path itself. A program with multiple connections is viewed as multiple clients to the server. The resource lifetimes are controlled by the connection lifetimes, not the application program lifetimes.

Connection a bidirectional byte stream that carries the X (and GLX) protocol between the client and the server. A client typically has only one connection to a server.

(Rendering) Context a OpenGL rendering context. This is a virtual OpenGL machine. All OpenGL rendering is done with respect to a context. The state maintained by one rendering context is not affected by another except in case of shared display lists.

GLXContext a handle to a rendering context. Rendering contexts consist of client side state and server side state.

Similar a potential correspondence among `GLXDrawable`s and rendering contexts. `Window`s and `GLXPixmap`s are similar to a rendering context are similar if, and only if, they have been created with respect to the same `VisualID` and root window.

Thread one of a group of processes all sharing the same address space. Typically, each thread will have its own program counter and stack pointer, but the text and data spaces are visible to each of the threads. A thread that is the only member of its group is equivalent to a process.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL and Win32

A Simple Example

In order to use OpenGL with Win32 to render images, there are some initialization steps that must be taken. These steps are outlined below.

Creating a Window
Setting the Pixel Format
Creating a Rendering Context

Example source code:
simple.c

Create a Window

Before creating a window, a *window class* must be registered. A window class is a basic template that is used to create a window in an application. Every window is associated with a window class. To register a window class, a `WNDCLASS` structure is filled out with the desired settings and then the Win32 function `RegisterWindowClass()` is called with a pointer to this structure as an argument. Multiple windows can be associated with a single class. When the application that registered a window class exits, the window class is destroyed. A window class can be identified by its class name (a character string).

The window class contains the *window procedure*. A window procedure is a callback function that is used by Win32 to notify the application of messages that should be processed by the window. A window procedure must have the form: `LONG WINAPI WindowProc(HWND, UINT, WPARAM, LPARAM)`. See the next section on messages for more information about window procedures.

The following code fragment shows how to register a new window class.

code fragment from oglCreateWindow() function in simple.c

```
/* oglCreateWindow
 *  Create a window suitable for OpenGL rendering
 */
HWND oglCreateWindow(char* title, int x, int y, int width, int height)
{
    WNDCLASS  wc;
    HWND      hWnd;
    HINSTANCE hInstance;

    /* get this modules instance */
    hInstance = GetModuleHandle(NULL);

    /* fill in the window class structure */
    wc.style         = 0;                           /* no special styles */
    wc.lpfnWndProc   = (WNDPROC)WindowProc;         /* event handler */
    wc.cbClsExtra    = 0;                           /* no extra class data */
    wc.cbWndExtra    = 0;                           /* no extra window data */
    wc.hInstance     = hInstance;                   /* instance */
    wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO); /* load a default icon */
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW); /* load a default cursor */
    wc.hbrBackground = NULL;                        /* redraw our own bg */
    wc.lpszMenuName  = NULL;                        /* no menu */
    wc.lpszClassName = title;                       /* use a special class */

    /* register the window class */
    if (!RegisterClass(&wc)) {
      MessageBox(NULL,
                 "RegisterClass() failed:  Cannot register window class,",
                 "Error", MB_OK);
        return NULL;
    }

    . . .

}
```

Although the settings above should be sufficient for many applications, there are many values each field of the `WNDCLASS` structure can assume. For more information on the `WNDCLASS` structure and its options, see the Microsoft Developer Studio InfoViewer topic *`WNDCLASS`*.

Once a window class has been successfully registered, a new window can be created. When creating a window suitable for OpenGL rendering, the window style must have the `WS_CLIPSIBLINGS` and `WS_CLIPCHILDREN` attribute bits set.

The following code shows how to create a window.

code fragment from oglCreateWindow() function in simple.c

```
/* oglCreateWindow
 *  Create a window suitable for OpenGL rendering
 */
HWND oglCreateWindow(char* title, int x, int y, int width, int height)
{
    WNDCLASS  wc;
    HWND      hWnd;
    HINSTANCE hInstance;

    . . .
```

```
    /* create a window */
    hWnd = CreateWindow(title,          /* class */
                        title,          /* title (caption) */
                        WS_CLIPSIBLINGS | WS_CLIPCHILDREN,  /* style */
                        x, y, width, height, /* dimensions */
                        NULL,           /* no parent */
                        NULL,           /* no menu */
                        hInstance,      /* instance */
                        NULL);          /* don't pass anything to WM_CREATE */

    /* make sure we got a window */
    if (hWnd == NULL) {
        MessageBox(NULL,
                   "CreateWindow() failed:  Cannot create a window.",
                   "Error", MB_OK);
        return NULL;
    }

    /* show the window (map it) */
    ShowWindow(hWnd, SW_SHOW);

    /* send an initial WM_PAINT message (expose) */
    UpdateWindow(hWnd);

    return hWnd;
}
```

A common style attribute which is used quite often (and bears mentioning here) is the `WS_OVERLAPPEDWINDOW` style. This creates a window that has resize handles and a system menu as well as the three icons (minimize, maximize and close) common to most Win32 windows in the upper right hand corner of the title (caption) bar. In the next section on messages, there are some example programs that use this style. Another style that can be used allows for the window to take up the whole screen. See the fullscrn.c program for an example of this style.

While in the example we only use the minimum style options necessary for OpenGL (`WS_CLIPCHILDREN` and `WS_CLIPSIBLINGS`), there are many options that can be used when creating a window. See the Microsoft Developer Studio InfoViewer topic *CreateWindow* for a list of all the available options.

After creating a new window it must be shown if the rendering is to be seen. It is also a good idea (though not strictly necessary) to force an initial paint by making a call to the window procedure in order to "prime the message pump". This is accomplished by calling the `ShowWindow()` and `UpdateWindow()` functions as shown in the example above.

Set the Pixel Format

After a window class has been registered and a new window has been successfully created, the *pixel format* must be set. The simplest way to set the pixel format is to use the `ChoosePixelFormat()` function. More sophisticated methods for choosing the pixel format will be discussed in a later section.

The pixel format specifies several properties of an OpenGL context. Common properties are depth of the Z buffer, whether a stencil buffer exists or not, whether the framebuffer is double buffered and many others.

In order to specify the many properties available, a `PIXELFORMATDESCRIPTOR` structure is employed. The members of this structure correspond to different properties. In order to set these properties, the corresponding field is set in the `PIXELFORMATDESCRIPTOR` structure and a format that best fits the criteria defined by the `PIXELFORMATDESCRIPTOR` structure is selected by the `ChoosePixelFormat()` function. The "best fit" is somewhat ambiguous and methods for finding exactly the pixel format desired are covered, as mentioned above, in a later section.

The following code fragment illustrates how to set the pixel format.

code defining the oglSetPixelFormat() function in simple.c

```
/* oglPixelFormat()
 *  Sets the pixel format for the context
 */
int oglSetPixelFormat(HDC hDC, BYTE type, DWORD flags)
{
    int pf;
    PIXELFORMATDESCRIPTOR pfd;

    /* fill in the pixel format descriptor */
    pfd.nSize        = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion     = 1;                   /* version (should be 1) */
    pfd.dwFlags      = PFD_DRAW_TO_WINDOW | /* draw to window (not bitmap) */
                       PFD_SUPPORT_OPENGL | /* draw using opengl */
                       flags;               /* user supplied flags */
    pfd.iPixelType   = type;                /* PFD_TYPE_RGBA or COLORINDEX */
    pfd.cColorBits   = 24;
    /* other criteria here */

    /* get the appropriate pixel format */
    pf = ChoosePixelFormat(hDC, &pfd);
    if (pf == 0) {
       MessageBox(NULL,
                  "ChoosePixelFormat() failed:  Cannot find format specified.",
                  "Error", MB_OK);
       return 0;
    }

    /* set the pixel format */
    if (SetPixelFormat(hDC, pf, &pfd) == FALSE) {
        MessageBox(NULL,
                   "SetPixelFormat() failed:  Cannot set format specified.",
                   "Error", MB_OK);
        return 0;
    }

    return pf;
}
```

Note that `type` is one of `PFD_TYPE_RGBA` for non-paletted or `PFD_COLORINDEX` for paletted (indexed) display mode. `flags` is a bitwise OR (|) of several options. We’ll use only `PFD_DOUBLEBUFFER` which selects a doublebuffered framebuffer for these simple examples. For more information on what other values it can assume, see the next section on pixel formats or the Microsoft Developer Studio InfoViewer topic *`PIXELFORMATDESCRIPTOR`*.

Create a Rendering Context

The final step in setting up for OpenGL rendering is to create the OpenGL context. An OpenGL rendering context in Win32 has the type `HGLRC`. All OpenGL rendering must go through an HGLRC. A context must be current for OpenGL calls to affect to it.

The procedure for creating and making a context current is shown below.

code from main() function in simple.c

```
/* main()
 *  Entry point
 */
int main(int argc, char** argv)
{
    HDC       hDC;                       /* device context */
    HGLRC     hRC;                       /* opengl context */
    HWND      hWnd;                      /* window */

    . . .

    /* create an OpenGL context */
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);

    /* now we can start changing state & rendering */
    while (1) {
        /* rotate a triangle around */
        glClear(GL_COLOR_BUFFER_BIT);
        glRotatef(1.0, 0.0, 0.0, 1.0);
        glBegin(GL_TRIANGLES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2i( 0,  1);
        glColor3f(0.0, 1.0, 0.0);
        glVertex2i(-1, -1);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2i( 1, -1);
        glEnd();
        glFlush();
        SwapBuffers(hDC);                /* nop if singlebuffered */
    }

    /* clean up */
    wglMakeCurrent(NULL, NULL);          /* make our context 'un-'current */
    ReleaseDC(hDC, hWnd);                /* release handle to DC */
    wglDeleteContext(hRC);               /* delete the rendering context */
    DestroyWindow(hWnd);                 /* destroy the window */

    return 0;
}
```

After this is done, OpenGL calls can be made to change state and render to the context as shown in the example above. In order to clean up the resources allocated for OpenGL rendering, first make the HGLRC 'un'-current, release the HDC and delete the context. Lastly, destroy the window.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL and Win32

Processing Messages & Using Menus

Win32 Messages and Menus allow for processing of user input. Methods for intercepting and responding to messages as well as methods for using menus is presented below.

Peeking at Messages
Using Message Procedures
Using Menus

Example source code:
peek.c
msgproc.c
menu.c

Peeking at Messages

While the simple example presented in the last section got us started with OpenGL, it was very limited in that it didn’t provide for any user input. *Messages* are the standard method to receive and process user input in Win32. An easy way to check for messages is presented below. This approach is very simple and limited. There are more sophisticated methods for processing messages which will be covered later in this document.

code defining the main() function in msgproc.c

```
/* main()
 *  Entry point
 */
int main(int argc, char** argv)
{
    HDC       hDC;                      /* device context */
    HGLRC     hRC;                      /* opengl context */
```

```
HWND      hWnd;                     /* window */
MSG       msg;                      /* message */

/* create a window */
hWnd = oglCreateWindow("OpenGL", 0, 0, 200, 200);
if (hWnd == NULL)
  exit(1);

/* get the device context */
hDC = GetDC(hWnd);

/* set the pixel format */
if (oglSetPixelFormat(hDC, PFD_TYPE_RGBA, 0) == 0)
  exit(1);

/* create an OpenGL context */
hRC = wglCreateContext(hDC);
wglMakeCurrent(hDC, hRC);

/* now we can start changing state & rendering */
while (1) {
    /* first, check for (and process) messages in the queue */
    while(PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE)) {
        switch(msg.message) {
        case WM_LBUTTONDOWN:
            printf("WM_LBUTTONDOWN: %d %d %s %s %s %s %s\n",
                   LOWORD(msg.lParam), HIWORD(msg.lParam),
                   msg.wParam & MK_CONTROL ? "MK_CONTROL" : "",
                   msg.wParam & MK_LBUTTON ? "MK_LBUTTON" : "",
                   msg.wParam & MK_RBUTTON ? "MK_RBUTTON" : "",
                   msg.wParam & MK_MBUTTON ? "MK_MBUTTON" : "",
                   msg.wParam & MK_SHIFT   ? "MK_SHIFT"   : "");
            break;
        case WM_MOUSEMOVE:
            printf("WM_MOUSEMOVE: %d %d\n",
                   LOWORD(msg.lParam), HIWORD(msg.lParam));
            break;
        case WM_KEYDOWN:
            printf("WM_KEYDOWN: %c\n", msg.wParam);
            if(msg.wParam == 27) /* ESC */
                goto quit;
            break;
        default:
            DefWindowProc(hWnd, msg.message, msg.wParam, msg.lParam);
            break;
        }
    }

    /* rotate a triangle around */
    glClear(GL_COLOR_BUFFER_BIT);
    glRotatef(1.0, 0.0, 0.0, 1.0);
    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2i( 0,  1);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2i(-1, -1);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2i( 1, -1);
    glEnd();
    glFlush();
    SwapBuffers(hDC);               /* nop if singlebuffered */
```

```
    }

quit:

    /* clean up */
    wglMakeCurrent(NULL, NULL);         /* make our context 'un-'current */
    ReleaseDC(hDC, hWnd);               /* release handle to DC */
    wglDeleteContext(hRC);              /* delete the rendering context */
    DestroyWindow(hWnd);                /* destroy the window */

    return 0;
}
```

There are many other messages that can be checked for and processed. See the macros defined in the `winuser.h` include file for a full listing, or look at Microsoft Developer Studio InfoViewer topics beginning with *WM_*. The method presented above is limited in that some messages must be "translated" before they can be processed. The method presented next takes care of these messages as well.

Message Procedure

A much more effective way of processing messages is to use a *window procedure*. Every window must have a window procedure associated with it (actually, the window procedure is associated with the window class, but since every window has a class, every window also has a window procedure). The window procedure is called whenever there are messages for the window in the message queue.

The code for a typical window procedure follows.

code defining the WindowProc() function in msgproc.c

```
/* WindowProc()
 *  Minimum Window Procedure
 */
LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    LONG        lRet = 1;
    PAINTSTRUCT ps;

    switch(uMsg) {
    case WM_CREATE:
        break;

    case WM_DESTROY:
        break;

    case WM_PAINT:
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;

    case WM_LBUTTONDOWN:
        printf("WM_LBUTTONDOWN: %d %d %s %s %s %s %s\n",
               LOWORD(lParam), HIWORD(lParam),
               wParam & MK_CONTROL ? "MK_CONTROL" : "",
               wParam & MK_LBUTTON ? "MK_LBUTTON" : "",
```

```
                wParam & MK_RBUTTON ? "MK_RBUTTON" : "",
                wParam & MK_MBUTTON ? "MK_MBUTTON" : "",
                wParam & MK_SHIFT   ? "MK_SHIFT"   : "");
            break;

        case WM_MOUSEMOVE:
            printf("WM_MOUSEMOVE: %d %d\n", LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_CHAR:
            printf("WM_CHAR: %c\n", wParam);
            if(wParam == 27)                /* ESC */
                PostQuitMessage(0);
            break;

        case WM_SIZE:
            printf("WM_SIZE: %d %d\n", LOWORD(lParam), HIWORD(lParam));
            glViewport(0, 0, LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_CLOSE:
            printf("WM_CLOSE\n");
            PostQuitMessage(0);
            break;

        default:
            lRet = DefWindowProc(hWnd, uMsg, wParam, lParam);
            break;
        }

        return lRet;
}
```

Each `case` in the `switch` statement processes one type of message. As mentioned above, there are many types of messages. The ones presented in this code fragment are some of the more common ones. Notice that the `default` action is to call a `DefWindowProc()` function. This passes on any messages that the user doesn't intercept to the system message processing function.

The translation and dispatch of messages must be done explicitly. The following code illustrates a method of doing this.

code defining the main() function in msgproc.c

```
/* main()
 *  Entry point
 */
int main(int argc, char** argv)
{
    HDC       hDC;                      /* device context */
    HGLRC     hRC;                      /* opengl context */
    HWND      hWnd;                     /* window */
    MSG       msg;                      /* message */

    /* create a window */
    hWnd = oglCreateWindow("OpenGL", 0, 0, 200, 200);
    if (hWnd == NULL)
      exit(1);

    /* get the device context */
```

```
    hDC = GetDC(hWnd);

    /* set the pixel format */
    if (oglSetPixelFormat(hDC, PFD_TYPE_RGBA, 0) == 0)
      exit(1);

    /* get the device context */
    hDC = GetDC(hWnd);

    /* create an OpenGL context */
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);

    /* now we can start changing state & rendering */
    while (1) {
        /* first, check for (and process) messages in the queue */
        while(PeekMessage(&msg, hWnd, 0, 0, PM_NOREMOVE)) {
            if(GetMessage(&msg, hWnd, 0, 0)) {
                TranslateMessage(&msg); /* translate virtual-key messages */
                DispatchMessage(&msg);  /* call the window proc */
            } else {
                goto quit;
            }
        }

        /* rotate a triangle around */
        glClear(GL_COLOR_BUFFER_BIT);
        glRotatef(1.0, 0.0, 0.0, 1.0);
        glBegin(GL_TRIANGLES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2i( 0,  1);
        glColor3f(0.0, 1.0, 0.0);
        glVertex2i(-1, -1);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2i( 1, -1);
        glEnd();
        glFlush();
        SwapBuffers(hDC);               /* nop if singlebuffered */
    }

quit:

    /* clean up */
    wglMakeCurrent(NULL, NULL);         /* make our context ’un-’current */
    ReleaseDC(hDC, hWnd);               /* release handle to DC */
    wglDeleteContext(hRC);              /* delete the rendering context */
    DestroyWindow(hWnd);                /* destroy the window */

    return 0;
}
```

The `TranslateMessage()` function breaks down virtual-key messages into character messages. The `DispatchMessage()` function dispatches a message to the window procedure, which means it calls the window procedure with the correct arguments for the given message.

Menus

Another common method for obtaining user input in Win32 is through menus. Setting up and managing a menu is very simple. The following example shows how to create a menu bar.

code defining the menubar() function in menu.c

```
/* globals */
HMENU hPopup = NULL;                    /* popup menu */

  . . .

/* menubar()
 * create a menubar for the window
 */
void menubar(HWND hWnd)
{
    HMENU        hFileMenu;             /* file menu handle */
    HMENU        hDrawMenu;             /* draw menu handle */
    HMENU        hMenu;                 /* menu bar handle */
    MENUITEMINFO item;                  /* item info */

    /* create the menus */
    hMenu     = CreateMenu();
    hFileMenu = CreateMenu();
    hDrawMenu = CreateMenu();

    /* fill up the file menu */
    item.cbSize      = sizeof(MENUITEMINFO);
    item.fMask       = MIIM_ID | MIIM_TYPE | MIIM_SUBMENU;
    item.fType       = MFT_STRING;
    item.hSubMenu    = NULL;

    item.wID        = ’x’;
    item.dwTypeData = "E&xit";
    item.cch        = strlen("E&xit");
    InsertMenuItem(hFileMenu, 0, FALSE, &item);

    /* now do the draw menu */
    item.wID        = ’r’;
    item.dwTypeData = "&Rotate";
    item.cch        = strlen("&Rotate");
    InsertMenuItem(hDrawMenu, 0, FALSE, &item);
    item.wID        = ’s’;
    item.dwTypeData = "&Don’t Rotate";
    item.cch        = strlen("&Don’t Rotate");
    InsertMenuItem(hDrawMenu, 1, FALSE, &item);

    /* now do the main menu */
    item.wID        = 0;
    item.dwTypeData = "&File";
    item.cch        = strlen("&File");
    item.hSubMenu   = hFileMenu;
    InsertMenuItem(hMenu, 0, FALSE, &item);
    item.wID        = 0;
    item.dwTypeData = "&Draw";
    item.cch        = strlen("&Draw");
    item.hSubMenu   = hDrawMenu;
    InsertMenuItem(hMenu, 1, FALSE, &item);

    /* attach the menu to the window */
    SetMenu(hWnd, hMenu);
```

```
    /* use the draw menu as a popup menu */
    hPopup = hDrawMenu;
}
```

The above code creates all the menus needed in the program. It also attaches the menus to the menubar at the top of the window just under the title (caption) bar. An ampersand in a string used as a dwTypeData causes an underscore beneath the following letter to be printed, and uses that letter as the accelerator key.

All menus send a `WM_COMMAND` message to the window that they are attached to. The low word of the wParam sent to the message procedure indicates what item was selected from the menu. The following code handles the actions attached to each menu. It should be inserted into the window procedure of an application.

code defining the menubar() function in menu.c

```
/* globals */
BOOL  Rotate = TRUE;                    /* rotating? */

    . . .

/* WindowProc()
 *  Minimum Window Procedure
 */
LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    . . .

    case WM_COMMAND:
        printf("WM_COMMAND: %c\n", LOWORD(wParam));
        switch(LOWORD(wParam)) {
        case ’s’:
            Rotate = FALSE;
            break;
        case ’r’:
            Rotate = TRUE;
            break;
        case ’x’:
            PostQuitMessage(0);
            break;
        }
        break;

    . . .
}
```

A popup menu is one that is attached to a certain mouse button. When the button is pressed inside the window, the menu should "pop-up" right below where the mouse was pressed. These type of menus take an additional step to set up. Since they are triggered when a mouse button is pressed, the corresponding message must be reacted to.

The following code explains how to react to mouse messages for popup menus. It should be inserted in the window procedure of the application.

```
/* globals */
HMENU hPopup = NULL;                    /* popup menu */
```

```
    . . .

/* WindowProc()
 *  Minimum Window Procedure
 */
LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    POINT       point;

    . . .

    case WM_RBUTTONDOWN:
        point.x = LOWORD(lParam);
        point.y = HIWORD(lParam);
        ClientToScreen(hWnd, &point);
        TrackPopupMenu(hPopup, TPM_LEFTALIGN, point.x, point.y,
                       0, hWnd, NULL);
        break;

    . . .
}
```

Note that the x and y location of the menu must be in screen coordinates, not window coordinates. The conversion is facilitated by the `ClientToScreen()` function.

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL and Win32

Pixel Formats & Palettes

Pixel formats specify the properties of OpenGL contexts. Pixel formats in conjunction with palettes are the gateway through which an appropriate context for an application is created. Their use is described below.

Pixel Format Descriptor
Using Palettes

Example source code:
wglinfo.c
index.c

Pixel Format Descriptor

Setting the pixel format seems to be one of the more tricky parts of programming with OpenGL in Win32. This section should dispel most of the mystery surrounding the *pixel format descriptor* and the setting of pixel formats. A pixel format descriptor is the key to getting and setting pixel formats.

There are several functions that are used to manipulate pixel formats. They are as follows:

Function	**Description**
ChoosePixelFormat	Obtains the device context’s pixel format that is the closest match to a specified pixel format.
SetPixelFormat	Sets a device context’s current pixel format to the pixel format specified by a pixel format index.
GetPixelFormat	Obtains the pixel format index of a device context’s current pixel format.
DescribePixelFormat	Given a device context and a pixel format index,

fills in a `PIXELFORMATDESCRIPTOR` data structure
with the pixel format’s properties.

A lot of the time, the `ChoosePixelFormat()` function will be adequate to choose a pixel format, but when more precision in pixel format choice is needed, other methods must be employed. An excellent method of selecting a pixel format with specific properties is to enumerate them all and compare them against your own criteria. When one fits all the criteria, stop examining the rest of the formats (if any) and use the one that fit. Weights can even be added to certain criteria if need be. For example, if it was absolutely necessary that a color depth of 24 bits be used, but not so necessary that the depth buffer be 24 bits, the weights could be set accordingly. The following code illustrates this method. It only prints out information for those pixel formats that are OpenGL capable. Of course, when choosing a visual to render with, more criteria should probably be used (such as color depth, z-buffer depth and single/doublebuffering -- all the possible criteria are outlined below).

code defining the VisualInfo() function in wglinfo.c

```
/* VisualInfo()
 *  Shows a graph of all the visuals that support OpenGL and their
 *  capabilities.  Just like (well, almost) glxinfo on SGI’s.
 */
void VisualInfo(HDC hDC)
{
    int i, maxpf;
    PIXELFORMATDESCRIPTOR pfd;

    /* calling DescribePixelFormat() with NULL args return maximum
       number of pixel formats */
    maxpf = DescribePixelFormat(hDC, 0, 0, NULL);

    /* print the table header */
 printf("   visual  x  bf lv rg d st  r  g  b a  ax dp st accum buffs  ms \n");
 printf(" id dep cl sp sz l  ci b ro sz sz sz sz bf th cl  r  g  b  a ns b\n");
 printf("-----------------------------------------------------------------\n");

    /* loop through all the pixel formats */
    for(i = 1; i <= maxpf; i++) {

        DescribePixelFormat(hDC, i, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

        /* only describe this format if it supports OpenGL */
        if(!(pfd.dwFlags & PFD_SUPPORT_OPENGL))
            continue;

        /* other criteria could be tested here for actual pixel format
           choosing in an application:

           for (...each pixel format...) {

             if (pfd.dwFlags & PFD_SUPPORT_OPENGL &&
                 pfd.dwFlags & PFD_DOUBLEBUFFER &&
                 pfd.cDepthBits >= 24 &&
                 pfd.cColorBits >= 24)
                 {
                    goto found;
                 }
           }
           ... not found so exit ...
```

```
   found:
   ... found so use it ...
*/

/* print out the information for this pixel format */
printf("0x%02x ", i);

printf("%2d ", pfd.cColorBits);
if(pfd.dwFlags & PFD_DRAW_TO_WINDOW)      printf("wn ");
else if(pfd.dwFlags & PFD_DRAW_TO_BITMAP) printf("bm ");
else printf(".  ");

/* should find transparent pixel from LAYERPLANEDESCRIPTOR */
printf(" . ");

printf("%2d ", pfd.cColorBits);

/* bReserved field indicates number of over/underlays */
if(pfd.bReserved) printf(" %d ", pfd.bReserved);
else printf(" . ");

printf(" %c ", pfd.iPixelType == PFD_TYPE_RGBA ? ’r’ : ’c’);

printf("%c ", pfd.dwFlags & PFD_DOUBLEBUFFER ? ’y’ : ’.’);

printf(" %c ", pfd.dwFlags & PFD_STEREO ? ’y’ : ’.’);

if(pfd.cRedBits)        printf("%2d ", pfd.cRedBits);
else printf(" . ");

if(pfd.cGreenBits)      printf("%2d ", pfd.cGreenBits);
else printf(" . ");

if(pfd.cBlueBits)       printf("%2d ", pfd.cBlueBits);
else printf(" . ");

if(pfd.cAlphaBits)      printf("%2d ", pfd.cAlphaBits);
else printf(" . ");

if(pfd.cAuxBuffers)     printf("%2d ", pfd.cAuxBuffers);
else printf(" . ");

if(pfd.cDepthBits)      printf("%2d ", pfd.cDepthBits);
else printf(" . ");

if(pfd.cStencilBits)    printf("%2d ", pfd.cStencilBits);
else printf(" . ");

if(pfd.cAccumRedBits)   printf("%2d ", pfd.cAccumRedBits);
else printf(" . ");

if(pfd.cAccumGreenBits) printf("%2d ", pfd.cAccumGreenBits);
else printf(" . ");

if(pfd.cAccumBlueBits)  printf("%2d ", pfd.cAccumBlueBits);
else printf(" . ");

if(pfd.cAccumAlphaBits) printf("%2d ", pfd.cAccumAlphaBits);
else printf(" . ");

/* no multisample in Win32 */
```

```
        printf(" . .\n");
    }

    /* print table footer */
 printf("-----------------------------------------------------------------\n");
 printf("   visual  x  bf lv rg d st  r  g  b a  ax dp st accum buffs  ms \n");
 printf(" id dep cl sp sz l  ci b ro sz sz sz sz bf th cl  r  g  b  a ns b\n");
 printf("-----------------------------------------------------------------\n");

}
```

Following is a detailed description of the `PIXELFORMATDESCRIPTOR` structures fields as shown in the Microsoft Developer Studio InfoViewer topic *PIXELFORMATDESCRIPTOR*.

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerType;
    BYTE  bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

Members

nSize
Specifies the size of this data structure. This value should be set to sizeof(PIXELFORMATDESCRIPTOR).

nVersion
Specifies the version of this data structure. This value should be set to 1.

dwFlags
A set of bit flags that specify properties of the pixel buffer. The properties are generally not mutually exclusive; you can set any combination of bit flags, with the exceptions noted. The following bit flag constants are defined.

Value	**Meaning**
PFD_DRAW_TO_WINDOW	The buffer can draw to a window or

	device surface.
PFD_DRAW_TO_BITMAP	The buffer can draw to a memory bitmap.
PFD_SUPPORT_GDI	The buffer supports GDI drawing. This flag and PFD_DOUBLEBUFFER are mutually exclusive in the current generic implementation.
PFD_SUPPORT_OPENGL	The buffer supports OpenGL drawing.
PFD_GENERIC_ACCELERATED	The pixel format is supported by a device driver that accelerates the generic implementation. If this flag is clear and the PFD_GENERIC_FORMAT flag is set, the pixel format is supported by the generic implementation only.
PFD_GENERIC_FORMAT	The pixel format is supported by the GDI software implementation, which is also known as the generic implementation. If this bit is clear, the pixel format is supported by a device driver or hardware.
PFD_NEED_PALETTE	The buffer uses RGBA pixels on a palette-managed device. A logical palette is required to achieve the best results for this pixel type. Colors in the palette should be specified according to the values of the cRedBits, cRedShift, cGreenBits, cGreenShift, cBluebits, and cBlueShift members. The palette should be created and realized in the device context before calling wglMakeCurrent.
PFD_NEED_SYSTEM_PALETTE	Used with systems with OpenGL hardware that supports one hardware palette only. For such systems to use hardware acceleration, the hardware palette must be in a fixed order (for example, 3-3-2) when in RGBA mode or must match the logical palette when in color-index mode. When you set this flag, call SetSystemPaletteUse in your program to force a one-to-one

	mapping of the logical palette and the system palette. If your OpenGL hardware supports multiple hardware palettes and the device driver can allocate spare hardware palettes for OpenGL, you don't need to set PFD_NEED_SYSTEM_PALETTE. This flag is not set in the generic pixel formats.
PFD_DOUBLEBUFFER	The buffer is double-buffered. This flag and PFD_SUPPORT_GDI are mutually exclusive in the current generic implementation.
PFD_STEREO	The buffer is stereoscopic. This flag is not supported in the current generic implementation.
PFD_SWAP_LAYER_BUFFERS	Indicates whether a device can swap individual layer planes with pixel formats that include double-buffered overlay or underlay planes. Otherwise all layer planes are swapped together as a group. When this flag is set, wglSwapLayerBuffers is supported.

You can specify the following bit flags when calling `ChoosePixelFormat()`.

Value	**Meaning**
PFD_DEPTH_DONTCARE	The requested pixel format can either have or not have a depth buffer. To select a pixel format without a depth buffer, you must specify this flag. The requested pixel format can be with or without a depth buffer. Otherwise, only pixel formats with a depth buffer are considered.
PFD_DOUBLEBUFFER_DONTCARE	The requested pixel format can be either single- or double-buffered.
PFD_STEREO_DONTCARE	The requested pixel format can be either monoscopic or stereoscopic.

With the `glAddSwapHintRectWIN` extension function, two new flags are included for the `PIXELFORMATDESCRIPTOR` pixel format structure.

Value	**Meaning**
PFD_SWAP_COPY	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the content of the back buffer to be copied to the front buffer. The content of the back buffer is not affected by the swap. PFD_SWAP_COPY is a hint only and might not be provided by a driver.
PFD_SWAP_EXCHANGE	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the exchange of back buffer's content with the front buffer's content. Following the swap, the back buffer's content contains the front buffer's content before the swap. PFD_SWAP_EXCHANGE is a hint only and might not be provided by a driver.

iPixelType
Specifies the type of pixel data. The following types are defined.

Value	**Meaning**
PFD_TYPE_RGBA	RGBA pixels. Each pixel has four components in this order: red, green, blue, and alpha.
PFD_TYPE_COLORINDEX	Color index pixels. Each pixel uses a color-index value.

cColorBits
Specifies the number of color bitplanes in each color buffer. For RGBA pixel types, it is the size of the color buffer, excluding the alpha bitplanes. For color index pixels, it is the size of the color-index buffer.

cRedBits
Specifies the number of red bitplanes in each RGBA color buffer.
cRedShift
Specifies the shift count for red bitplanes in each RGBA color buffer.
cGreenBits
Specifies the number of green bitplanes in each RGBA color buffer.
cGreenShift
Specifies the shift count for green bitplanes in each RGBA color buffer.
cBlueBits
Specifies the number of blue bitplanes in each RGBA color buffer.
cBlueShift

Specifies the shift count for blue bitplanes in each RGBA color buffer.
cAlphaBits
Specifies the number of alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.
cAccumBits
Specifies the total number of bitplanes in the accumulation buffer.
cAccumRedBits
Specifies the number of red bitplanes in the accumulation buffer.
cAccumGreenBits
Specifies the number of green bitplanes in the accumulation buffer.
cAccumBlueBits
Specifies the number of blue bitplanes in the accumulation buffer.
cAccumAlphaBits
Specifies the number of alpha bitplanes in the accumulation buffer.
cDepthBits
Specifies the depth of the depth (z-axis) buffer.
cStencilBits
Specifies the depth of the stencil buffer.
cAuxBuffers
Specifies the number of auxiliary buffers. Auxiliary buffers are not supported.
iLayerType
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.
bReserved
Not used. Must be zero.
dwLayerMask
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.
dwVisibleMask
Specifies the transparent color or index of an underlay plane. When the pixel type is RGBA, dwLayerMask is a transparent RGB color value. When the pixel type is color index, it is a transparent index value.
dwDamageMask
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.

Note that in the documentation above, when it says "not supported" it means not supported in the generic implementation of OpenGL supplied by Microsoft. Different hardware types may well support some of these options (such as alpha bitplanes, or auxiliary buffers).

Here’s a short code fragment which finds a pixel format that is OpenGL capable, draws to a window, has a depth buffer greater than or equal to 24 bits and is double buffered:

code fragment defining oglPixelFormatExact() in exact.c

```
/* oglPixelFormatExact()
 *  Sets the pixel format for the context
 */
int oglSetPixelFormatExact(HDC hDC)
{
    int pf, maxpf;
    PIXELFORMATDESCRIPTOR pfd;

    /* get the maximum number of pixel formats */
```

```
    maxpf = DescribePixelFormat(hDC, 0, 0, NULL);

    /* loop through all the pixel formats */
    for (pf = 1; pf <= maxpf; pf++) {
        DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
        if (pfd.dwFlags & PFD_DRAW_TO_WINDOW &&
            pfd.dwFlags & PFD_SUPPORT_OPENGL &&
            pfd.dwFlags & PFD_DOUBLEBUFFER   &&
            pfd.cDepthBits >= 24)
          {
              /* found a matching pixel format */

              /* set the pixel format */
              if (SetPixelFormat(hDC, pf, &pfd) == FALSE) {
                MessageBox(NULL,
                   "SetPixelFormat() failed:  Cannot set format specified.",
                   "Error", MB_OK);
                return 0;
              }

              return pf;
          }
    }

    /* couldn’t find one, bail out! */
    MessageBox(NULL,
               "Fatal Error:  Failed to find a suitable pixel format.",
               "Error", MB_OK);
    return 0;
}
```

Using Palettes

Up to this point, we’ve neglected a very important part of the integration of OpenGL with Win32 -- *palettes*. A palette is a table of colors used when a Truecolor display can’t be used or when the application wants exact control over what colors are available (for example, in a height field), or when palette animation functionality is desired.

There are two situations that arise regarding palettes when using OpenGL and Win32. The first is trying to use a color-index context. A discussion of this follows. The second is a bit harder -- using an RGBA context in a paletted mode.

When using a color-index context, a *logical palette* must be created. A logical palette is a table of colors that is *selected* and *realized* into a device context. This just means that the user defines a table of colors, then forces windows to use those colors. On a Truecolor display, this isn’t a problem, but on a paletted display, Windows must try to match up the system and logical palettes the best it can. Sometimes there is a "flashing" that occurs because of this palette switching.

The following code shows how to initialize a logical palette.

code defining the oglSetPalette() function in index.c

```
/* globals */
HPALETTE     hPalette;                  /* handle to custom palette */

. . .

/* oglSetPalette()
 *  Sets the palette
 */
BOOL oglSetPalette(HDC hDC)
{
    LOGPALETTE   lgpal;                 /* custom logical palette */
    int          nEntries = 5;          /* number of entries in palette */
    PALETTEENTRY peEntries[5] = {       /* entries in custom palette */
        0,   0,   0, NULL,              /* black */
      255,   0,   0, NULL,              /* red */
        0, 255,   0, NULL,              /* green */
        0,   0, 255, NULL,              /* blue */
     255, 255, 255, NULL                /* white */
    };

    /* create a logical palette (for color index mode) */
    lgpal.palVersion    = 0x300;        /* version should be 0x300 */
    lgpal.palNumEntries = nEntries;     /* number of entries in palette */
    if((hPalette = CreatePalette(&lgpal)) == NULL) {
        MessageBox(NULL,
                   "CreatePalette() failed:  Cannot create palette.",
                   "Error", MB_OK);
        return FALSE;
    }

    /* set the palette entries */
    SetPaletteEntries(hPalette, 0, nEntries, peEntries);

    /* select the palette */
    SelectPalette(hDC, hPalette, TRUE); /* map logical into physical palette */

    /* realize the palette */
    RealizePalette(hDC);

    return TRUE;
}
```

In addition to the initialization code, there are some messages that must be dealt with when using palettes. The following shows these messages and the reaction to them.

code fragment from WindowProc() function in index.c

```
/* globals */
HPALETTE     hPalette;                  /* handle to custom palette */

. . .

/* WindowProc()
 *  Minimum Window Procedure
 */
LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    LONG        lRet = 1;
    PAINTSTRUCT ps;
```

```
    switch(uMsg) {
    . . .

    case WM_QUERYNEWPALETTE:
        SelectPalette(GetDC(hWnd), hPalette, FALSE);/* select custom palette */
        lRet = RealizePalette(GetDC(hWnd));
        break;

    case WM_PALETTECHANGED:
        if(hWnd == (HWND)wParam)        /* make sure we don’t loop forever */
            break;
        SelectPalette(GetDC(hWnd), hPalette, FALSE);/* select custom palette */
        RealizePalette(GetDC(hWnd));    /* remap the custom palette */
        UpdateColors(GetDC(hWnd));
        lRet = 0;
        break;

    . . .

    }

    return lRet;
}
```

This next section is very tricky. Palette management in general is tricky, but even more so when trying to simulate Truecolor with a palette. The basic idea is to create a palette that has an adequate range of colors so that a Truecolor display can be simulated with the aid of dithering. There are many ways to generate such a palette. For a full example, see the Microsoft Developer Studio InfoViewer topic *RGBA Mode and Windows Palette Management*. We’ll use a simple palette derived from the example cited above.

Note that this operation need only be done if the `dwFlags` member of the `PIXELFORMATDESCRIPTOR` structure has the `PFD_NEED_PALETTE` bit set.

Following is the code required to setup a new palette for RGBA rendering in a paletted display mode.

code from the GLUT for Win32 sources

```
static HPALETTE ghpalOld, ghPalette = (HPALETTE) 0;

static unsigned char threeto8[8] = {
  0, 0111>>1, 0222>>1, 0333>>1, 0444>>1, 0555>>1, 0666>>1, 0377
};

static unsigned char twoto8[4] = {
  0, 0x55, 0xaa, 0xff
};

static unsigned char oneto8[2] = {
  0, 255
};

static int defaultOverride[13] = {
  0, 3, 24, 27, 64, 67, 88, 173, 181, 236, 247, 164, 91
};

static PALETTEENTRY defaultPalEntry[20] = {
  { 0,   0,   0,    0 },
```

```
  { 0x80,0,   0,    0 },
  { 0,   0x80,0,    0 },
  { 0x80,0x80,0,    0 },
  { 0,   0,   0x80, 0 },
  { 0x80,0,   0x80, 0 },
  { 0,   0x80,0x80, 0 },
  { 0xC0,0xC0,0xC0, 0 },

  { 192, 220, 192,  0 },
  { 166, 202, 240,  0 },
  { 255, 251, 240,  0 },
  { 160, 160, 164,  0 },

  { 0x80,0x80,0x80, 0 },
  { 0xFF,0,   0,    0 },
  { 0,   0xFF,0,    0 },
  { 0xFF,0xFF,0,    0 },
  { 0,   0,   0xFF, 0 },
  { 0xFF,0,   0xFF, 0 },
  { 0,   0xFF,0xFF, 0 },
  { 0xFF,0xFF,0xFF, 0 }
};

static unsigned char ComponentFromIndex(int i, UINT nbits, UINT shift) {
  unsigned char val;

  val = (unsigned char) (i >> shift);
  switch (nbits) {
  case 1:
    val &= 0x1;
    return oneto8[val];

  case 2:
    val &= 0x3;
    return twoto8[val];

  case 3:
    val &= 0x7;
    return threeto8[val];

  default:
    return 0;
  }
}


HPALETTE CreateRGBPalette(HDC hDC) {
  PIXELFORMATDESCRIPTOR pfd;
  LOGPALETTE *pPal;
  int n, i;

  n = GetPixelFormat(hDC);
  DescribePixelFormat(hDC, n, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

  if (pfd.dwFlags & PFD_NEED_PALETTE) {
    n = 1 << pfd.cColorBits;
    pPal = (PLOGPALETTE)LocalAlloc(LMEM_FIXED, sizeof(LOGPALETTE) +
                                   n * sizeof(PALETTEENTRY));
    pPal->palVersion = 0x300;
    pPal->palNumEntries = n;
    for (i=0; ipalPalEntry[i].peRed =
```

```
        ComponentFromIndex(i, pfd.cRedBits, pfd.cRedShift);
      pPal->palPalEntry[i].peGreen =
        ComponentFromIndex(i, pfd.cGreenBits, pfd.cGreenShift);
      pPal->palPalEntry[i].peBlue =
        ComponentFromIndex(i, pfd.cBlueBits, pfd.cBlueShift);
      pPal->palPalEntry[i].peFlags = 0;
    }

    /* fix up the palette to include the default GDI palette */
    if ((pfd.cColorBits == 8)                           &&
        (pfd.cRedBits   == 3) && (pfd.cRedShift   == 0) &&
        (pfd.cGreenBits == 3) && (pfd.cGreenShift == 3) &&
        (pfd.cBlueBits  == 2) && (pfd.cBlueShift  == 6)
        ) {
      for (i = 1 ; i <= 12 ; i++)
        pPal->palPalEntry[defaultOverride[i]] = defaultPalEntry[i];
    }

    ghPalette = CreatePalette(pPal);
    if(!ghPalette)
      __glutFatalError("CreatePalette() failed:  Cannot create palette.");
    LocalFree(pPal);

    ghpalOld = SelectPalette(hDC, ghPalette, FALSE);
    n = RealizePalette(hDC);
  }

  return ghPalette;
}
```

As you can see, it is very messy and very tricky. However, for the most part, this code can simply be "cut and pasted" into an application. When it is determined that the application needs an RGB palette (if the `PFD_NEED_PALETTE` bit is set as described above), call the `CreateRGBPalette()` function.

In addition to the initialization code, there are some windows messages that must now be intercepted.

code from the GLUT for Win32 sources

```
    case WM_QUERYNEWPALETTE:
     if (ghPalette) {
        SelectPalette(GetDC(hWnd), hPalette, FALSE);/* select custom palette */
        lRet = RealizePalette(GetDC(hWnd));
     }
     break;

    case WM_PALETTECHANGED:
     if (ghPalette) {
        if(hWnd == (HWND)wParam)        /* make sure we don’t loop forever */
            break;
        SelectPalette(GetDC(hWnd), hPalette, FALSE);/* select custom palette */
        RealizePalette(GetDC(hWnd));    /* remap the custom palette */
        UpdateColors(GetDC(hWnd));
        lRet = 0;
     }
     break;
```

SIGGRAPH ’97

Course 24: OpenGL and Window System Integration

OpenGL and Win32

Overlays & Underlays

Overlays and underlays are often used in applications for rendering above (or below) the main OpenGL context. Setup and use of overlays and underlays is discussed below.

Overlays & Underlays

Example source code:
overlay.c

Overlays

Some pixel formats include an overlay or underlay plane. If overlay or underlay planes are desired, a pixel format with these must be selected. You cannot have free-floating overlay windows that can move over other windows. Overlay planes have a transparent color to allow things drawn ’beneath’ them to show through. Every layer has a palette associated with it.

Unlike main plane pixel formats, overlay and underlay plane formats don’t have an equivalent `ChoosePixelFormat()`, so a method similar to that described in the pixel format section must be employed to find an appropriate format.

The following code will setup the pixel format to use an overlay plane if available. Note that it looks very similar to the pixel format choosing code developed in the last section. Notable differences are the `wglDescribeLayerPlane()` function call in place of the `DescribePixelFormat()` call in the previous example.

code defining oglPixelFormat() function in overlay.c

```
/* oglPixelFormat()
 *  Sets the pixel format for the context
 */
```

```
int oglSetPixelFormatOverlay(HDC hDC, BYTE type, DWORD flags)
{
    int pf, maxpf;
    PIXELFORMATDESCRIPTOR pfd;
    LAYERPLANEDESCRIPTOR  lpd;          /* layer plane descriptor */
    int   nEntries = 2;                 /* number of entries in palette */
    COLORREF crEntries[2] = {           /* entries in custom palette */
      0x00000000,                       /* black (ref #0 = transparent) */
      0x00ff0000,                       /* blue */
    };

    /* get the maximum number of pixel formats */
    maxpf = DescribePixelFormat(hDC, 0, 0, NULL);

    /* find an overlay layer descriptor */
    for(pf = 0; pf < maxpf; pf++) {
        DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

        /* the bReserved field of the PIXELFORMATDESCRIPTOR contains the
           number of overlay/underlay planes */
        if (pfd.bReserved > 0) {
          /* aha! This format has overlays/underlays */
          wglDescribeLayerPlane(hDC, pf, 1,
                                sizeof(LAYERPLANEDESCRIPTOR), &lpd);
          if (lpd.dwFlags & LPD_SUPPORT_OPENGL &&
              lpd.dwFlags & flags)
            {
              goto found;
            }
        }
    }
    /* couldn’t find any overlay/underlay planes */
    MessageBox(NULL,
               "Fatal Error:  Hardware does not support overlay planes.",
               "Error", MB_OK);
    return 0;

found:
    /* now get the "normal" pixel format descriptor for the layer */
    DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

    /* set the pixel format */
    if(SetPixelFormat(hDC, pf, &pfd) == FALSE) {
        MessageBox(NULL,
                   "SetPixelFormat() failed:  Cannot set format specified.",
                   "Error", MB_OK);
        return 0;
    }

    /* set up the layer palette */
    wglSetLayerPaletteEntries(hDC, 1, 0, nEntries, crEntries);

    /* realize the palette */
    wglRealizeLayerPalette(hDC, 1, TRUE);

    /* announce what we’ve got */
    printf("Number of overlays = %d\n", pfd.bReserved);
    printf("Color bits in the overlay = %d\n", lpd.cColorBits);

    return pf;
}
```

Now simply create an overlay context in much the same way that you create a main plane context. The number passed in to the `wglCreateLayerContext()` function is the layer number.

code fragment from the main() function in overlay.c

```
/* main()
 *  Entry point
 */
int main(int argc, char** argv)
{
    HWND      hWnd;                     /* window */
    MSG       msg;                      /* message */

    /* create a window */
    hWnd = oglCreateWindow("OpenGL", 0, 0, 200, 200);
    if (hWnd == NULL)
      exit(1);

    /* get the device context */
    hDC = GetDC(hWnd);

    /* set the pixel format */
    if (oglSetPixelFormatOverlay(hDC, PFD_TYPE_RGBA, LPD_DOUBLEBUFFER) == 0)
      exit(1);

    /* get the device context */
    hDC = GetDC(hWnd);

    /* create an OpenGL overlay context */
    hOverlayRC = wglCreateLayerContext(hDC, 1);

    . . .
}
```

When rendering to the overlay, be sure to set it current. Also be sure to swap the correct plane if using double buffering. Note that you must also swap the main plane with `wglSwapLayerBuffers()`, NOT `SwapBuffers()` when using overlay or underlay planes. Pass in `WGL_SWAP_MAIN_PLANE` as the second argument to `wglSwapLayerBuffers()` to swap the main plane, and `WGL_SWAP_OVERLAYi` where i is the overlay number to swap an overlay buffer.

code fragment from the main() function in overlay.c

```
/* main()
 *  Entry point
 */
int main(int argc, char** argv)
{
    HWND      hWnd;                     /* window */
    MSG       msg;                      /* message */

    . . .

    /* create an OpenGL overlay context */
    hOverlayRC = wglCreateLayerContext(hDC, 1);

    /* create an OpenGL context */
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);
```

```
    /* now we can start changing state & rendering */
    while(1) {
        /* first, check for (and process) messages in the queue */
        while(PeekMessage(&msg, hWnd, 0, 0, PM_NOREMOVE)) {
            if(GetMessage(&msg, hWnd, 0, 0)) {
                TranslateMessage(&msg); /* translate virtual-key messages */
                DispatchMessage(&msg);  /* call the window proc */
            } else {
                goto quit;
            }
        }

        /* make current and draw a triangle */
        wglMakeCurrent(hDC, hRC);
        glClear(GL_COLOR_BUFFER_BIT);
        glRotatef(1.0, 0.0, 0.0, 1.0);
        glBegin(GL_TRIANGLES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2i( 0,  1);
        glColor3f(0.0, 1.0, 0.0);
        glVertex2i(-1, -1);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2i( 1, -1);
        glEnd();
        glFlush();
        wglSwapLayerBuffers(hDC, WGL_SWAP_MAIN_PLANE);

        /* make current and draw a triangle */
        wglMakeCurrent(hDC, hOverlayRC);
        glClear(GL_COLOR_BUFFER_BIT);
        glRotatef(-1.0, 0.0, 0.0, 1.0);
        glBegin(GL_TRIANGLES);
        glIndexi(1);
        glVertex2i( 0,  1);
        glVertex2i(-1, -1);
        glVertex2i( 1, -1);
        glEnd();
        glFlush();
        wglSwapLayerBuffers(hDC, WGL_SWAP_OVERLAY1);
    }

quit:

    /* clean up */
    wglMakeCurrent(NULL, NULL);         /* make our context ’un-’current */
    ReleaseDC(hDC, hWnd);               /* release handle to DC */
    wglDeleteContext(hRC);              /* delete the rendering context */
    wglDeleteContext(hOverlayRC);       /* delete the overlay context */
    DestroyWindow(hWnd);                /* destroy the window */

    return TRUE;
}
```

SIGGRAPH '97

Course 24: OpenGL and Window System Integration

OpenGL and Win32

WGL Reference

WGL (pronounced "wiggle") is the glue that binds OpenGL and the Win32 API together.

Rendering Context functions
Font and Text functions
Overlay, Underlay and Main Plane functions
Miscellaneous functions

Rendering Context Functions

Function	Description
wglCreateContext	Creates a new rendering context.
wglMakeCurrent	Sets a thread's current rendering context.
wglGetCurrentContext	Obtains a handle to a thread's current rendering context.
wglGetCurrentDC	Obtains a handle to the device context associated with a thread's current rendering context.
wglDeleteContext	Deletes a rendering context.

See the source code referenced in previous sections for examples of the use of each of these functions.

Font and Text functions

Function	Description

wglUseFontBitmaps	Creates a set of character bitmap display lists. Characters come from a specified device context’s current font. Characters are specified as a consecutive run within the font’s glyph set.
wglUseFontOutlines	Creates a set of display lists, based on the glyphs of the currently selected outline font of a device context, for use with the current rendering context. The display lists are used to draw 3-D characters of TrueType fonts.

example from Microsoft Developer Studio topic *wglUseFontBitmaps*

```
HDC    hdc;
HGLRC  hglrc;

// create a rendering context
hglrc = wglCreateContext (hdc);

// make it the calling thread’s current rendering context
wglMakeCurrent (hdc, hglrc);

// now we can call OpenGL API

// make the system font the device context’s selected font
SelectObject (hdc, GetStockObject (SYSTEM_FONT));

// create the bitmap display lists
// we’re making images of glyphs 0 thru 255
// the display list numbering starts at 1000, an arbitrary choice
wglUseFontBitmaps (hdc, 0, 255, 1000);

// display a string:
// indicate start of glyph display lists
glListBase (1000);
// now draw the characters in a string
glCallLists (24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World");
```

example from Microsoft Developer Studio topic *wglUseFontOutlines*

```
HDC    hdc;  // A TrueType font has already been selected
HGLRC  hglrc;
GLYPHMETRICSFLOAT agmf[256];

// Make hglrc the calling thread’s current rendering context
wglMakeCurrent(hdc, hglrc);

// create display lists for glyphs 0 through 255 with 0.1 extrusion
// and default deviation. The display list numbering starts at 1000
// (it could be any number)
wglUseFontOutlines(hdc, 0, 255, 1000, 0.0f, 0.1f,
            WGL_FONT_POLYGONS, &agmf);

// Set up transformation to draw the string
glLoadIdentity();
glTranslate(0.0f, 0.0f, -5.0f)
glScalef(2.0f, 2.0f, 2.0f);
```

```
// Display a string
glListBase(1000); // Indicates the start of display lists for the glyphs
// Draw the characters in a string
glCallLists(24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World.");
```

Overlay, Underlay and Main Plane functions

Function	**Description**
wglCopyContext	Copies selected groups of rendering states from one OpenGL rendering context to another.
wglCreateLayerContext	Creates a new OpenGL rendering context for drawing to a specified layer plane on a device context.
wglDescribeLayerPlane	Obtains information about the layer planes of a given pixel format.
wglGetLayerPaletteEntries	Retrieves the palette entries from a given color-index layer plane for a specified device context.
wglRealizeLayerPalette	Maps palette entries from a given color-index layer plane into the physical palette or initializes the palette of an RGBA layer plane.
wglSetLayerPaletteEntries	Sets the palette entries in a given color-index layer plane for a specified device context.
wglSwapLayerBuffers	Swaps the front and back buffers in the overlay, underlay, and main planes of the window referenced by a specified device context.

See the overlay.c program for examples of how to use the functions above.

Miscellaneous Functions

Function	**Description**
wglShareLists	Enables a rendering context to share the display-list space of another rendering context.
wglGetProcAddress	Returns the address of an OpenGL extension function for use with the current OpenGL rendering context.