LinuxWorld 2000 in San Jose

Tutorial HL:
Linux 3D Hardware Acceleration

Brad Grantham `(grantham@valinux.com)`

July 2000

This whitepaper complements LinuxWorld 2000 San Jose Tutorial HL with an introduction to OpenGL (a portable 2D and 3D rendering API) and the APIs which allow OpenGL acceleration under Linux, including some high-level toolkits which ease the use of OpenGL from an application developer perspective.

Copyright

Introduction

LinuxWorld 2000 San Jose Tutorial HL, Linux 3D Hardware Acceleration, presents OpenGL 3D acceleration for Linux from the point of view of both application developers and OpenGL driver implementors. Presenters well known in the industry draw on their past experience with OpenGL on Silicon Graphics' Irix, Microsoft Windows, and Linux to provide details about the use and design of OpenGL.

Speakers

David Blythe is a member of the technical staff at RouteFree, Inc. Prior to this, David was chief engineer in the advanced graphics group at SGI. David contributed to the development of the RealityEngine and InfiniteReality graphics systems while at Silicon Graphics. He has worked extensively on implementations of the OpenGL graphics library, OpenGL extension specifications, and high-level toolkits built on top of OpenGL. His other interests include large-scale system design and interactive photorealism. David has been a course presenter at SIGGRAPH '96-'2000 as well as other technical forums. Prior to joining SGI, David was a visualization scientist at the Ontario Centre for Large Scale Computation and a lecturer at the University of Toronto. David received both a B.S. and

M.S. degree in computer science from the University of Toronto.

Brad Grantham is a Senior Software Engineer at VA Linux Systems in Sunnyvale, California. Brad specializes in 3D graphics and OpenGL, and previously worked at Silicon Graphics on object oriented graphics toolkits. Prior to SGI, Brad worked at Tenon Intersystems on a Macintosh UNIX variant and at Recognition Research on image processing. Brad is an Adjunct Lecturer at Santa Clara University and has presented at the SIGGRAPH computer graphics conference for the past three years.

Brian Paul works for Precision Insight, Inc. where he contributes to the development of infrastructure and drivers for 3D hardware acceleration on Linux. He's probably best known for Mesa: his open source implementation of the OpenGL API. Brian has been involved in 3D software development for over 10 years and holds BS and MS degrees in computer science from the University of Wisconsin.

OpenGL under Linux

OpenGL is *the* industry standard 3D graphics API. OpenGL is available on most platforms from Silicon Graphics InfiniteReality to Sun Workstations' Creator3D to Hewlett Packard's Visualize fx6 series, all the way to a five hundred dollar commodity PC that can be bought at a computer superstore. OpenGL 1.2 supports the most common high-end features of graphics cards including multitexture, texture automatic coordinate generation, and geometry and lighting acceleration. More advanced features are easily exposable as OpenGL extensions either by an individual vendor or by the OpenGL architecture Review board. The OpenGL API is not tied to an individual window or operating system and is thus very portable.

OpenGL is used for all types of 3D visualization as well as 2D illustration. Many games available under Windows 98 and most under Linux use OpenGL, including Quake III: Arena, Baldur's Gate II, and Descent 3. OpenGL has long been the choice of API for Scientific Visualization and CAD software. Many animation and visualization packages go so far as to implement their user interface using OpenGL because it allows them to write platform-independent graphics code.

Linux has already established itself as a powerful platform for internet services and computing and is now starting to compete with Windows as a desktop environment for applications. Only time will tell whether Linux can match or even supplant Windows, but support for games and other 3D applications is a prerequisite. OpenGL has been available in an unaccelerated form on Linux for several years, both as a port of Silicon Graphics' OpenGL and as Mesa, a multiplatform open source implementation of the OpenGL 1.2 API.

In the last year and a half, the list of 3D acclerators supported under Linux has grown from one to nearly all of the accelerators supported available for PCs. Matrox, 3dfx, 3Dlabs, S3, Evans and Sutherland, and NVIDIA are all supported in some form of hardware-accelerated Mesa or OpenGL.

Some implementations of the OpenGL API under Linux provide advanced features rivaling those of Windows OpenGL implementations, including multitexture, 3D texture, geometry acceleration, and AGP buffers for DMA for rapid transfer of commands to hardware. Benchmarking by Utah-GLX developers (admittedly biased) shows Linux Mesa and OpenGL to approach or even exceed the performance of OpenGL under Windows 98. (see utah-glx-dev mailing list messages from the 21st of January, 2000 and the 30th of January, 2000)

This tutorial provides an introduction to the use and implementation of the OpenGL API on Linux, and should be valuable both to beginning OpenGL application developers wishing to use Linux and to engineers curious about the foundations of 3D direct rendering using the OpenGL API.

Brief Introduction to OpenGL

This tutorial is not intended to contain a detailed introduction to the features of OpenGL itself. This tutorial instead touches only on the basics of OpenGL to provide a brief introduction. For an in-depth introduction to OpenGL, see the *OpenGL Programming Guide* (also known as the "Red Book", most recently edited by Dave Shreiner), which introduces the reader to each of the OpenGL features in turn and how to use each one. A detailed technical specification for OpenGL is available at www.opengl.org.

OpenGL is an immediate mode 2D and 3D graphics library. An application calls commands in the OpenGL library to change state including colors, lighting, and textures, and to draw primitives. A snippet of code in OpenGL to draw a red lit triangle can look like the following:

```
GLfloat lightPos[4] = [0.707f, 0.707f, 0.0f, 0.0f];
GLfloat redColor[4] = [1.0f, 0.0f, 0.0f, 1.0f];

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, redColor);

glBegin(GL_TRIANGLES);
    glNormal3f(0, 0, 1);
    glVertex3f(-1, 1, 0);

    glNormal3f(0, 0, 1);
    glVertex3f(-1, -1, 0);

    glNormal3f(0, 0, 1);
    glVertex3f(1, -1, 0);
glEnd();
```

Each application thread has a current openGL *context*, which contains drawing state and directs drawing commands to a window. Each of the functions above and most of the functions in OpenGL change the context's state directly and execute immediately from the viewpoint of the application. Thus OpenGL is called an *immediate mode* API. The OpenGL machine can be broken roughly into two parts: transformation and rasterization.

Geometric primitives in OpenGL are specified using 1, 2 or 3 dimensional vertices, for example with glVertex3f or with glInterleavedArrays. OpenGL maintains a stack of matrices for the *modelview* transformation, so that vertices can be modeled in one space, then rotate, scaled, and translated into another, and finally transformed into the camera space, or *eye* space. Because it's a stack, applications can easily apply hierarchical transformations through glMultMatrix, glPushMatrix, and glPopMatrix. Then vertices are lit using an (emperically derived) mathematical model including spotlight effects and surface shininess, often provided using glLightfv. The top of the *projection* matrix stack projects 3D points on to the 2D window.

Once primitives like triangles and lines are lit, projected to the screen and clipped, they are scan-converted. The colors produced at vertices are usually interpolated across the pixels inside the vertex, and this can produce a convincing illusion of surface curvature. A rectangle of pixels (*texture*) can be applied across the primitive to simulate changes in color across a surface or to be used in (potentially complex) pixel operations, using glTexImage and glTexEnv. A small screen aligned *stipple* pattern can be applied to mask pixels out, and there is a *stencil buffer* which can be used as a more general masking or counting register. The distance of the primitive from the eye is also interpolated and can be compared to the *depth buffer* to see if it each pixel not "hidden". Pixels from a primitive (*fragments*) can be blended with the color values already in the frame buffer.

OpenGL also provides functions for reading images from (glReadPixels) and storing images in (glDrawPixels) the framebuffer directly, although many implementations optimize this operation poorly. Images can be converted from or into different pixel formats, and row strides can allow use of a subset of the image. A color lookup can also be applied to color values from the image. Some implementations support the OpenGL 1.2 Imaging subset which allows a rich set of operations including lookups, histograms, and convolution among others.

All OpenGL state can be queried using some variant of glGet, and sets of state variables can be pushed and popped. A sequence of OpenGL commands can also be stored in a *display list* for execution later by glCallList; implementations may choose to compile those commands into a more efficient form for execution.

OpenGL also has some interesting but little used features for returning transformed and clipped vertices to the application and detecting when primitives are rasterized within a rectangular window (*selection*; usually used to detect which scene objects are under a mouse click).

OpenGL does not provide an audio API, a mechanism for playing back video, object-oriented scene manipulation, or windowing or operating system abstractions. OpenGL in general was designed to provide a flexible but detailed programming abstraction for a class of machines that draw raster graphics.

GLU, the OpenGL Utility Library

Because OpenGL is designed to be a clean hardware abstraction layer, it does not provide some obvious facilities for convenience. GLU provides functions for creating a viewing transformation from simple 3D data about the camera position and the object being viewed, drawing curved NURBS or quadric surfaces, and downloading texture data that is not a power of two on a side and filling in the data that OpenGL needs to perform texture antialiasing.

OpenGL and the X Window System

Since OpenGL does not provide a windowing or event API, a separate specification and API details how OpenGL is tied to a specific windowing system. For Linux and the X Window System, this API is *GLX*. (For Windows 98/NT, it's called *WGL*, and for MacOS it's called *AGL*.)

Many developers opt to avoid the complexities of GLX and use a portable toolkit which **does** encapsulate windowing and event systems, like GLUT (mentioned later in these notes). The remainder of this section presents a brief introduction to OpenGL and GLX.

GLX is actually used to describe several aspects of the process of rendering with OpenGL on X. In addition to being the API used to connect OpenGL with X, it is also the protocol used to transmit OpenGL commands over the X wire (socket or UNIX domain pipe). It is also used as the name of an open source package that provides hardware acceleration for OpenGL programs using Mesa and GLX, as described later.

Because the GLX protocol is communicated over the X wire, applications can run on one machine and direct their 3D rendering output to a completely different machine. Most commonly, though, OpenGL applications run on the same machine as the display. An OpenGL library can optionally detect this situation and send OpenGL commands directly to the hardware, known as *direct rendering*. This saves bandwidth and conversion overhead, but requires a great detail of synchronization between the X server and other direct OpenGL clients. Direct rendering in different implementations of the OpenGL API under Linux are described later.

An application requests an X visual (or searches for one itself) that matches its requirements for OpenGL bit-depths, doublebuffering, and other framebuffer attributes. An OpenGL context is created and bound to an X Windows drawable (usually a window). Here’s an example of some code that illustrates this process:

```
static int attributeList[] = {
    GLX_RGBA,
    GLX_DOUBLEBUFFER,
    GLX_DEPTH_SIZE, 16,
    GLX_RED_SIZE, 5,
    GLX_GREEN_SIZE, 5,
    GLX_BLUE_SIZE, 5,
    None
};

XVisualInfo *visInfo =
    glXChooseVisual(display, screen, attributeList);

/* Application opens X Window that matches visual info */

context = glXCreateContext(display, visInfo, 0, GL_TRUE);

glXMakeCurrent(display, window, context);

/* Application renders a bunch of 3D data */

glXSwapBuffers(display, drawable);
```

An application might issue some costly OpenGL operations and then issue X operations as well in a separate region, because OpenGL may be highly pipelined. Because OpenGL rendering is often direct, however, GLX provides functions allowing an application to force either OpenGL or X drawing to complete before proceeding with the other in an overlapping screen region, for example for 3D scenes annotated with X widgets.

GLX also allows for OpenGL rendering in drawables that do not require on-screen real estate, or X Pixmaps, but this operation is limited; the newer GLX 1.3 specification offers *pbuffers* which can be used as offscreen rendering areas that don’t require X semantics and whose contents can be copied to other drawables or texture memory.

GLX of course provides some more functionality than is presented here. Clients can flush the rendering pipeline, request that X fonts be turned into GL display lists for text rendering, move data between contexts, and query extensions to GLX. Users desiring a detailed guide to using GLX and OpenGL should consult Mark Kilgard's book *Programming OpenGL for the X Window System*.

Toolkits

GLUT

GLUT, by Mark Kilgard, (not to be mistaken with GLU) encapsulates window system and event processing in a simple toolkit. GLUT provides facilities to open windows, subwindows, draw simple geometric shapes, draw text, handle pop-up menus, and invoke application callbacks on mouse and keyboard events.

Some application developers use GLUT to rapidly prototype their OpenGL application, because of GLUT's simplicity. Others use it as the basis of their released application because of its portability. Our experience has been that most production-quality interactive OpenGL applications use GLX (or WGL) either directly or through their own proprietary portability layer in order to have more control, but GLUT is an excellent toolkit for whipping together demos and samples.

Here's an example of how an application might set up a GLUT window and a mouse and some standard callbacks:

```
glutInit(&argc, argv);
glutInitWindowSize(512,512);
glutInitDisplayString("samples rgb double depth");
glutCreateWindow("window title goes here");
glutDisplayFunc(redrawFunc);
glutKeyboardFunc(keyboardFunc);
glutMotionFunc(motionFunc);
glutMouseFunc(buttonFunc);
glutReshapeFunc(reshapeFunc);

/* initialize OpenGL here */

glutMainLoop();
```

Here's an example button callback:

```
void button(int b, int state, int x, int y)
{
    if(button == 0) {
        if(state == 0)
            doLeftButtonDownAction();
        else
            doLeftButtonUpAction();
    }
}
```

More information on the functions and features of GLUT can be found at the OpenGL web site, www.opengl.org.

Simple DirectMedia Layer (SDL)

Sam Lantinga has written a portable media API called *Simple DirectMedia Layer* for applications needing audio, input, and framebuffer access on multiple platforms. SDL is used largely for games but can be used to author general multimedia applications that can run on Linux, MacOS, BeOS, and Windows. Here's a snippet of code that creates an SDL window and draws using OpenGL, borrowed from the OpenGL tutorials available at http://www.libsdl.org/opengl/intro.html.

```
/* Initialize SDL for video output */
SDL_Init(SDL_INIT_VIDEO);
SDL_SetVideoMode(640, 480, 0, SDL_OPENGL);

/* Set the title bar in environments that support it */
SDL_WM_SetCaption("SDL Sample", NULL);

/* initialize OpenGL here */

/* Loop, drawing and checking events */
done = 0;
while ( ! done ) {

    /* Draw OpenGL scene here */

    SDL_Event event;
    while ( SDL_PollEvent(&event) ) {
        if ( event.type == SDL_QUIT )
            done = 1;
        if ( event.type == SDL_KEYDOWN ) {
            if ( event.key.keysym.sym == SDLK_ESCAPE )
                done = 1;
        }
    }
}
SDL_Quit();
```

SDL provides access to framebuffer memory (*surfaces*) with optional conversion, for blitting images to the screen. The SMPEG mpeg player uses SDL to display its video frames. Events from the windowing system are provided to the application from keyboard, mouse, and window operations. SDL provides a thread creation API, plus a semaphore facility for thread synchronization, and event processing in SDL is thread-safe. SDL also provides an API for playing 8-audio and 16-bit audio samples and for controlling audio playback from a CD.

SDL is available at http://www.devolution.com/~slouken/SDL/ and is available in a stable production version 1.08.

Open Inventor

Inventor was developed at Silicon Graphics and provides an object-oriented C++ interface for graphics objects and system abstractions. Inventor provides both a run-time API and a file format, and is considered a scene graph, because it stores 3D scene data in a directed graph configuration. In Inventor, state from objects affects the state of other objects "below" and "to the right", thus hierarchies of geometric transformation (rotation, scale, and rotation) can be layered together to create complex scenes. Inventor is used by many in visualization because of its ease-of-use and power.

Inventor provides an extremely rich assortment of tools including scene primitives, window and event management, reading and writing files, and scene operation. Here's a sample of some code that creates a very simple scene graph and displays it in a window, borrowed from *The Inventor Mentor* by Josie Wernecke:

```
Widget myWindow = SoXt::init(argv[0]);

// Create a new scene graph root node
SoSeparator *root = new SoSeparator;
root->ref();

SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
root->addChild(myCamera);
root->addChild(new SoDirectionalLight);

// Add a red material; all subsequent siblings will be affected by
// this red material until the next material
SoMaterial *myMaterial = new SoMaterial;
myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Red
root->addChild(myMaterial);

// Add a cone.
root->addChild(new SoCone);

// Create a renderArea in which to see our scene graph.
// The render area will appear within the main window.
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

// Make myCamera see everything.
myCamera->viewAll(root, myRenderArea->getViewportRegion());

// Put our scene in myRenderArea, change the title
myRenderArea->setSceneGraph(root);
myRenderArea->setTitle("Hello Cone 1");
myRenderArea->show();

// Map the window to the screen and give control to Inventor
SoXt::show(myWindow);  // Display main window
SoXt::mainLoop();      // Main Inventor event loop
```

An Inventor scene graph can be made of a variety of different types of *nodes*, including geometric shapes, texture data, material parameters, cameras, and object manipulators. Some shape nodes are simple predefined shapes like SoCone and SoIndexedFaceSet (Inventor scene objects are prefixed with **So** to differentiate them from application-specific data types). Property nodes change rendering state and include surface appearance changes like SoMaterial and SoTexture2, geometric transformation changes like SoTransform, and other nodes modifying interpretation of shape data like SoComplexity.

There are many more nodes in Inventor than are mentioned here. A quick reference to the nodes in Open Inventor 2.0 is available at http://www.sgi.com/Technology/Inventor/PostScript/quickRef.ps.

Inventor *actions* traverse the scene graph, accumulating, pushing, and popping state. The SoGLRenderAction accumulates render state, like material values and geometric transformations, and renders each shape it encounters.

Each node in Inventor is made up of *fields*. For example, the *shininess* parameter in an SoMaterial is a field of type MFFloat and thus contains one or more floating point numbers. Each field can be *connected* to another field, so that when the first field is changed, the second field changes to the same value. Fields may also be connected to *engines*, which change the field value in some way. Connections and engines are very powerful and can be used to make complex behaviors without application callbacks or intervention.

Inventor also contains a set of *manipulators*, nodes that a user can interact with to change rotations, scales, and translations or material parameters, like SoTabBox, which provides visible "tabs" the user can click on and drag to change the translation or scale provided to subsequent nodes in the scene graph.

Inventor provides a series of prebuilt user-interface widgets. SoXt allows an application to integrate Inventor operations and events with Xt. The SoXtExaminerViewer presents the user with a number of convenient buttons and menus for viewing a scene, including changing the rendering style and moving and rotating a scene.

Inventor specifies its own file format which directly maps to the scene graph data. Thus Inventor scene data can be stored verbatim in a file and read in at a later time to recreate nodes, connections and engines. The Inventor file format was used as the basis for the VRML virtual reality web file format.

Template Graphics (www.tgs.com) sells a port of Inventor version 2 to Linux. Coin, at www.coin3d.org, is a 3D toolkit that uses the Inventor 2.0 API, available in an open source release restricted to noncommercial use and as a commercial product.

IRIS Performer

Performer is another C++ object-oriented scene graph toolkit developed at Silicon Graphics. Performer tends to run more towards high-performance applications rather than ease-of-use; it is targetted at maximizing OpenGL rendering performance through the use of multiple processors for view-frustum culling, sorting primitives to reduce OpenGL state changes, and transferring vertex data using optimized unrolled loops.

Performer provides a full-featured toolkit for building high-performance visualization and simulation programs including scene primitives and a generalized file reader/writer facility, and allows the application a great deal of flexibility regarding how multiprocessing is configured. Here’s a simplified excerpt from the `hello.C` sample program:

```
// Initialize and configure IRIS Performer
pfInit();

pfMultiprocess( PFMP_DEFAULT );

pfConfig();

// Look for files in PFPATH, ".", and  "/usr/share/Performer/data"
pfFilePath(".:/usr/share/Performer/data:../../../../data");

// Create a scene
pfScene *scene = new pfScene;

// Create a lit scene pfGeoState for the scene
pfGeoState *gstate = new pfGeoState;
gstate->setMode(PFSTATE_ENLIGHTING, PF_ON);
// attach the pfGeoState to the scene
```

```
    scene->setGState(gstate);
    scene->addChild(new pfLightSource);

    // Create 3D message and place in scene.
    pfString *str = new pfString;
    pfFont *fnt = pfdLoadFont_type1("Times-Elfin",PFDFONT_EXTRUDED);

    str->setFont(fnt);
    str->setMode(PFSTR_JUSTIFY, PFSTR_MIDDLE);
    str->setColor(1.0f, 0.0f, 0.8f, 1.0f);
    str->setString("Welcome to IRIS Performer");
    str->flatten();

    pfText *text = new pfText;
    text->addString(str);
    scene->addChild(text);

    // Create and configure a pfPipe and pfChannel.
    pfPipe *pipe = pfGetPipe(0);
    pfChannel *chan = new pfChannel(pipe);
    chan->setFOV(60.0f, 0.0f);
    chan->setScene(scene);

    // Determine extent of scene's geometry.
    pfSphere bsphere;
    text->getBound(&bsphere);
    chan->setNearFar(1.0f, 10.0f*bsphere.radius);

    // Spin text for 15 seconds.
    double startTime = pfGetTime();
    double t;
    while ((t = pfGetTime() - startTime) < 15.0f)
    {
        pfCoord    view;
        float      s, c;

        // Compute new view position, rotating around text.
        pfSinCos(45.0f*t, &s, &c);
        view.hpr.set(45.0f*t, -5.0f, 0.0f);
        view.xyz.set(
                     2.0f * bsphere.radius * s,
                     -2.0f * bsphere.radius * c,
                     0.3f * bsphere.radius);
        chan->setView(view.xyz, view.hpr);

        // Initiate cull/draw processing for this frame.
        pfFrame();
    }

    // Terminate parallel processes and exit.
    pfExit();
```

As you can see, Performer allows an application a great deal of control over the scene and processing of the scene, but sometimes at the expense of usability for the beginner.

Like Inventor, Performer scenes are made of nodes in a directed graph, except that changes like rotations and scales are inherited from top to bottom and not from left to right. *Parent* nodes do something to their *children* like grouping them together (pfGroup), applying a geometric transformation, choosing one child out of many, or cycling through child nodes. Leaf nodes are usually pfGeodes, which group together pfGeoSet objects. A pfGeoSet contains a representation of geometric data like a triangle strip or lines and a set of state to apply to the geometry. Performer has traversals to cull the scene graph to only the visible objects, sort objects by graphics state changes, and to draw the scene, but usually these are all invoked by pfFrame, which kicks off multiple processes for operating on the scene graph.

Performer contains a large number of facilities for managing scene complexity and scene rate, like level of detail groups (pfLOD) which contain geometric objects of different complexities which Performer chooses based on the visual size of the object (distant models need less detail) and the current estimate of frame timing. Billboards are pictures which represent objects and are rotated to face the viewer as the viewer moves. Applications may get a detailed set of statistics to use for planning scene geometry through pfFrameStats.

Performer has a facility called libpfdb for providing readers and writers for files of any format; readers that already exist include Inventor, LightScape, and AutoCAD file formats. Most loaders use the libpfdu database utility library to help construct scene graphs and optimize them for efficient traversal.

The libpfu utility library contains various facilities for input handling, GUI widgets drawn in OpenGL, following a geometric path for walkthroughs, and texture loading, among several others.

Performer 2.3 binaries are available for Linux from Silicon Graphics' web site at www.sgi.com. Although Performer takes advantage of features in IRIX that are not available in Linux and other facilities are not yet ported to IRIX (Performer does not run now in multiple processors under Linux), we expect to see a release soon which supports Linux multiprocessing and graphics cards more fully.