To be published in the
January/February issue of *The X Journal.*

OpenGL™ and X, Part 2: Using OpenGL with Xlib

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.21

October 28, 1993

Abstract

This is the second article in a three-part series about using the OpenGL™ 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™ 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.

*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`

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`.

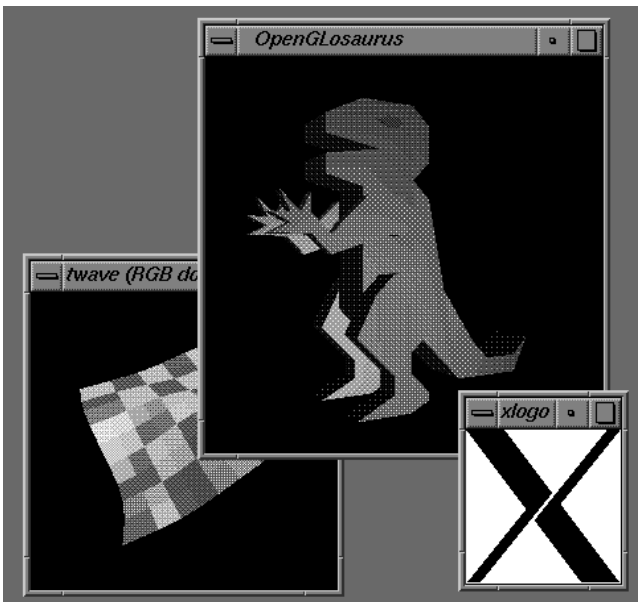

Figure 1: Screen snapshot of `glxdino`.

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.
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 sup-

ports 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

[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 ioctl`s. Eventually, Xlib will do an explicit check on the socket to see if it closes but only after a couple hundred calls to `XPending`.

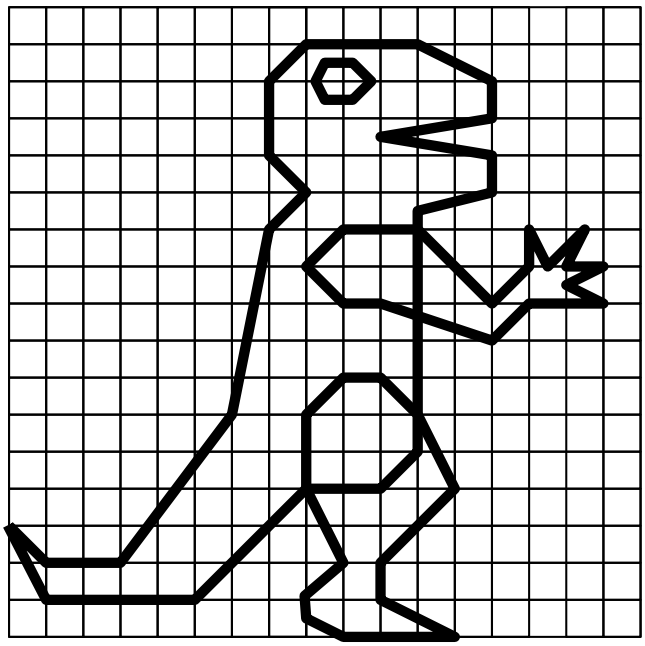

Figure 2: 2D complex polygons used to model the dinosaur's arm, leg, eye, and body sides.

`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.

`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 clock-wise 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. 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

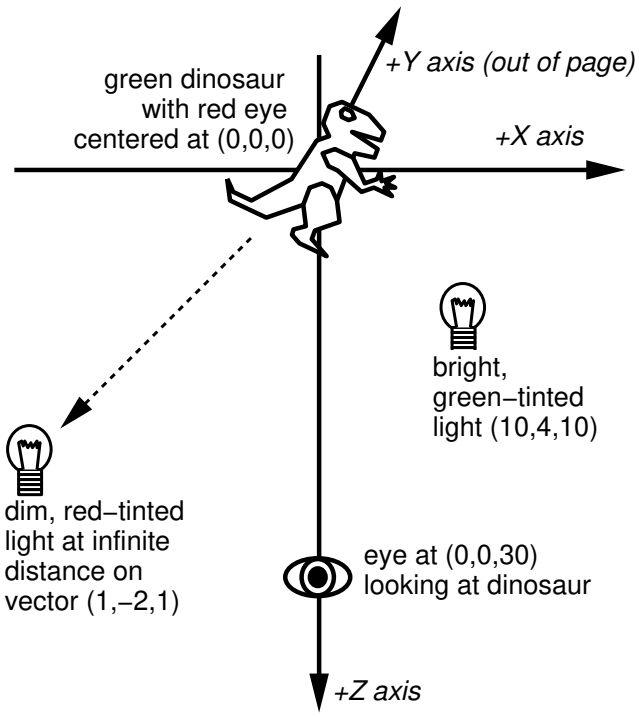

Figure 3: Arrangement of lights, eye, and dinosaur in modeling space.

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 com-

[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].

ponents 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.

2.4 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

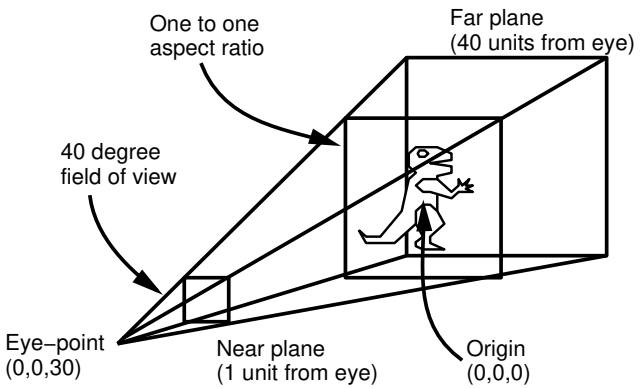

Figure 4: Static view for `glxdino`.

`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

```
1  /* compile: cc -o glxdino glxdino.c -lGLU -lGL -lXmu -lX11 */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>               /* for cos(), sin(), and sqrt() */
6  #include <GL/glx.h>             /* this includes X and gl.h headers */
7  #include <GL/glu.h>             /* gluPerspective(), gluLookAt(), GLU polygon
8                                   * tesselator */
9  #include <X11/Xatom.h>          /* for XA_RGB_DEFAULT_MAP atom */
10 #include <X11/Xmu/StdCmap.h>    /* for XmuLookupStandardColormap() */
11 #include <X11/keysym.h>         /* for XK_Escape keysym */

12 typedef enum {
13     RESERVED, BODY_SIDE, BODY_EDGE, BODY_WHOLE, ARM_SIDE, ARM_EDGE, ARM_WHOLE,
14     LEG_SIDE, LEG_EDGE, LEG_WHOLE, EYE_SIDE, EYE_EDGE, EYE_WHOLE, DINOSAUR
15 }               displayLists;

16 Display *dpy;
17 Window win;
18 GLfloat angle = -150;   /* in degrees */
19 GLboolean doubleBuffer = GL_TRUE, iconic = GL_FALSE, keepAspect = GL_FALSE;
20 int W = 300, H = 300;
21 XSizeHints sizeHints = {0};
22 GLdouble bodyWidth = 2.0;
23 int configuration[] = {GLX_DOUBLEBUFFER, GLX_RGBA, GLX_DEPTH_SIZE, 16, None};
24 GLfloat body[][2] = { {0, 3}, {1, 1}, {5, 1}, {8, 4}, {10, 4}, {11, 5},
25     {11, 11.5}, {13, 12}, {13, 13}, {10, 13.5}, {13, 14}, {13, 15}, {11, 16},
26     {8, 16}, {7, 15}, {7, 13}, {8, 12}, {7, 11}, {6, 6}, {4, 3}, {3, 2},
27     {1, 2} };
28 GLfloat arm[][2] = { {8, 10}, {9, 9}, {10, 9}, {13, 8}, {14, 9}, {16, 9},
29     {15, 9.5}, {16, 10}, {15, 10}, {15.5, 11}, {14.5, 10}, {14, 11}, {14, 10},
30     {13, 9}, {11, 11}, {9, 11} };
31 GLfloat leg[][2] = { {8, 6}, {8, 4}, {9, 3}, {9, 2}, {8, 1}, {8, 0.5}, {9, 0},
32     {12, 0}, {10, 1}, {10, 2}, {12, 4}, {11, 6}, {10, 7}, {9, 7} };
33 GLfloat eye[][2] = { {8.75, 15}, {9, 14.7}, {9.6, 14.7}, {10.1, 15},
34     {9.6, 15.25}, {9, 15.25} };
35 GLfloat lightZeroPosition[] = {10.0, 4.0, 10.0, 1.0};
36 GLfloat lightZeroColor[] = {0.8, 1.0, 0.8, 1.0}; /* green-tinted */
37 GLfloat lightOnePosition[] = {-1.0, -2.0, 1.0, 0.0};
38 GLfloat lightOneColor[] = {0.6, 0.3, 0.2, 1.0};  /* red-tinted */
39 GLfloat skinColor[] = {0.1, 1.0, 0.1, 1.0}, eyeColor[] = {1.0, 0.2, 0.2, 1.0};

40 void
41 fatalError(char *message)
42 {
43     fprintf(stderr, "glxdino: %s\n", message);
44     exit(1);
45 }

46 Colormap
47 getColormap(XVisualInfo * vi)
48 {
49     Status          status;
50     XStandardColormap *standardCmaps;
51     Colormap        cmap;
52     int             i, numCmaps;

53     /* be lazy; using DirectColor too involved for this example */
```

```
54      if (vi->class != TrueColor)
55          fatalError("no support for non-TrueColor visual");
56      /* if no standard colormap but TrueColor, just make an unshared one */
57      status = XmuLookupStandardColormap(dpy, vi->screen, vi->visualid,
58          vi->depth, XA_RGB_DEFAULT_MAP, /* replace */ False, /* retain */ True);
59      if (status == 1) {
60          status = XGetRGBColormaps(dpy, RootWindow(dpy, vi->screen),
61                                  &standardCmaps, &numCmaps, XA_RGB_DEFAULT_MAP);
62          if (status == 1)
63              for (i = 0; i < numCmaps; i++)
64                  if (standardCmaps[i].visualid == vi->visualid) {
65                      cmap = standardCmaps[i].colormap;
66                      XFree(standardCmaps);
67                      return cmap;
68                  }
69      }
70      cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
71          vi->visual, AllocNone);
72      return cmap;
73  }

74  void
75  extrudeSolidFromPolygon(GLfloat data[][2], unsigned int dataSize,
76      GLdouble thickness, GLuint side, GLuint edge, GLuint whole)
77  {
78      static GLUtriangulatorObj *tobj = NULL;
79      GLdouble        vertex[3], dx, dy, len;
80      int             i;
81      int             count = dataSize / (2 * sizeof(GLfloat));

82      if (tobj == NULL) {
83          tobj = gluNewTess();    /* create and initialize a GLU polygon
84                                   * tesselation object */
85          gluTessCallback(tobj, GLU_BEGIN, glBegin);
86          gluTessCallback(tobj, GLU_VERTEX, glVertex2fv); /* semi-tricky */
87          gluTessCallback(tobj, GLU_END, glEnd);
88      }
89      glNewList(side, GL_COMPILE);
90          glShadeModel(GL_SMOOTH); /* smooth minimizes seeing tessellation */
91          gluBeginPolygon(tobj);
92              for (i = 0; i < count; i++) {
93                  vertex[0] = data[i][0];
94                  vertex[1] = data[i][1];
95                  vertex[2] = 0;
96                  gluTessVertex(tobj, vertex, &data[i]);
97              }
98          gluEndPolygon(tobj);
99      glEndList();
100     glNewList(edge, GL_COMPILE);
101         glShadeModel(GL_FLAT);  /* flat shade keeps angular hands from being
102                                  * "smoothed" */
103         glBegin(GL_QUAD_STRIP);
104         for (i = 0; i <= count; i++) {
105             /* mod function handles closing the edge */
106             glVertex3f(data[i % count][0], data[i % count][1], 0.0);
107             glVertex3f(data[i % count][0], data[i % count][1], thickness);
108             /* Calculate a unit normal by dividing by Euclidean distance. We
109              * could be lazy and use glEnable(GL_NORMALIZE) so we could pass in
110              * arbitrary normals for a very slight performance hit. */
111             dx = data[(i + 1) % count][1] - data[i % count][1];
```

```
112             dy = data[i % count][0] - data[(i + 1) % count][0];
113             len = sqrt(dx * dx + dy * dy);
114             glNormal3f(dx / len, dy / len, 0.0);
115         }
116         glEnd();
117     glEndList();
118     glNewList(whole, GL_COMPILE);
119         glFrontFace(GL_CW);
120         glCallList(edge);
121         glNormal3f(0.0, 0.0, -1.0); /* constant normal for side */
122         glCallList(side);
123         glPushMatrix();
124             glTranslatef(0.0, 0.0, thickness);
125             glFrontFace(GL_CCW);
126             glNormal3f(0.0, 0.0, 1.0); /* opposite normal for other side */
127             glCallList(side);
128         glPopMatrix();
129     glEndList();
130 }

131 void
132 makeDinosaur(void)
133 {
134     GLfloat         bodyWidth = 3.0;

135     extrudeSolidFromPolygon(body, sizeof(body), bodyWidth,
136         BODY_SIDE, BODY_EDGE, BODY_WHOLE);
137     extrudeSolidFromPolygon(arm, sizeof(arm), bodyWidth / 4,
138         ARM_SIDE, ARM_EDGE, ARM_WHOLE);
139     extrudeSolidFromPolygon(leg, sizeof(leg), bodyWidth / 2,
140         LEG_SIDE, LEG_EDGE, LEG_WHOLE);
141     extrudeSolidFromPolygon(eye, sizeof(eye), bodyWidth + 0.2,
142         EYE_SIDE, EYE_EDGE, EYE_WHOLE);
143     glNewList(DINOSAUR, GL_COMPILE);
144         glMaterialfv(GL_FRONT, GL_DIFFUSE, skinColor);
145         glCallList(BODY_WHOLE);
146         glPushMatrix();
147             glTranslatef(0.0, 0.0, bodyWidth);
148             glCallList(ARM_WHOLE);
149             glCallList(LEG_WHOLE);
150             glTranslatef(0.0, 0.0, -bodyWidth - bodyWidth / 4);
151             glCallList(ARM_WHOLE);
152             glTranslatef(0.0, 0.0, -bodyWidth / 4);
153             glCallList(LEG_WHOLE);
154             glTranslatef(0.0, 0.0, bodyWidth / 2 - 0.1);
155             glMaterialfv(GL_FRONT, GL_DIFFUSE, eyeColor);
156             glCallList(EYE_WHOLE);
157         glPopMatrix();
158     glEndList();
159 }

160 void
161 redraw(void)
162 {
163     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
164     glCallList(DINOSAUR);
165     if (doubleBuffer)
166         glXSwapBuffers(dpy, win);       /* buffer swap does implicit glFlush */
167     else glFlush();             /* explicit flush for single buffered case */
168 }
```

```
169 void
170 main(int argc, char **argv)
171 {
172     XVisualInfo    *vi;
173     Colormap        cmap;
174     XSetWindowAttributes swa;
175     XWMHints       *wmHints;
176     Atom            wmDeleteWindow;
177     GLXContext      cx;
178     XEvent          event;
179     KeySym          ks;
180     GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
181     char           *display = NULL, *geometry = NULL;
182     int             flags, x, y, width, height, lastX, i;

183     /*** (1) process normal X command line arguments ***/
184     for (i = 1; i < argc; i++) {
185         if (!strcmp(argv[i], "-geometry")) {
186             if (++i >= argc)
187                 fatalError("follow -geometry option with geometry parameter");
188             geometry = argv[i];
189         } else if (!strcmp(argv[i], "-display")) {
190             if (++i >= argc)
191                 fatalError("follow -display option with display parameter");
192             display = argv[i];
193         } else if (!strcmp(argv[i], "-iconic")) iconic = GL_TRUE;
194         else if (!strcmp(argv[i], "-keepaspect")) keepAspect = GL_TRUE;
195         else if (!strcmp(argv[i], "-single")) doubleBuffer = GL_FALSE;
196         else fatalError("bad option");
197     }

198     /*** (2) open a connection to the X server ***/
199     dpy = XOpenDisplay(display);
200     if (dpy == NULL) fatalError("could not open display");

201     /*** (3) make sure OpenGL's GLX extension supported ***/
202     if (!glXQueryExtension(dpy, NULL, NULL))
203         fatalError("X server has no OpenGL GLX extension");

204     /*** (4) find an appropriate visual and a colormap for it ***/
205     /* find an OpenGL-capable RGB visual with depth buffer */
206     if (!doubleBuffer) goto SingleBufferOverride;
207     vi = glXChooseVisual(dpy, DefaultScreen(dpy), configuration);
208     if (vi == NULL) {
209       SingleBufferOverride:
210         vi = glXChooseVisual(dpy, DefaultScreen(dpy), &configuration[1]);
211         if (vi == NULL)
212             fatalError("no appropriate RGB visual with depth buffer");
213         doubleBuffer = GL_FALSE;
214     }
215     cmap = getColormap(vi);

216     /*** (5) create an OpenGL rendering context  ***/
217     /* create an OpenGL rendering context */
218     cx = glXCreateContext(dpy, vi, /* no sharing of display lists */ NULL,
219                           /* direct rendering if possible */ GL_TRUE);
220     if (cx == NULL) fatalError("could not create rendering context");

221     /*** (6) create an X window with selected visual and right properties ***/
```

```
222     flags = XParseGeometry(geometry, &x, &y,
223         (unsigned int *) &width, (unsigned int *) &height);
224     if (WidthValue & flags) {
225         sizeHints.flags |= USSize;
226         sizeHints.width = width;
227         W = width;
228     }
229     if (HeightValue & flags) {
230         sizeHints.flags |= USSize;
231         sizeHints.height = height;
232         H = height;
233     }
234     if (XValue & flags) {
235         if (XNegative & flags)
236             x = DisplayWidth(dpy, DefaultScreen(dpy)) + x - sizeHints.width;
237         sizeHints.flags |= USPosition;
238         sizeHints.x = x;
239     }
240     if (YValue & flags) {
241         if (YNegative & flags)
242             y = DisplayHeight(dpy, DefaultScreen(dpy)) + y - sizeHints.height;
243         sizeHints.flags |= USPosition;
244         sizeHints.y = y;
245     }
246     if (keepAspect) {
247         sizeHints.flags |= PAspect;
248         sizeHints.min_aspect.x = sizeHints.max_aspect.x = W;
249         sizeHints.min_aspect.y = sizeHints.max_aspect.y = H;
250     }
251     swa.colormap = cmap;
252     swa.border_pixel = 0;
253     swa.event_mask = ExposureMask | StructureNotifyMask |
254         ButtonPressMask | Button1MotionMask | KeyPressMask;
255     win = XCreateWindow(dpy, RootWindow(dpy, vi->screen),
256                         sizeHints.x, sizeHints.y, W, H,
257                         0, vi->depth, InputOutput, vi->visual,
258                         CWBorderPixel | CWColormap | CWEventMask, &swa);
259     XSetStandardProperties(dpy, win, "OpenGLosaurus", "glxdino",
260         None, argv, argc, &sizeHints);
261     wmHints = XAllocWMHints();
262     wmHints->initial_state = iconic ? IconicState : NormalState;
263     wmHints->flags = StateHint;
264     XSetWMHints(dpy, win, wmHints);
265     wmDeleteWindow = XInternAtom(dpy, "WM_DELETE_WINDOW", False);
266     XSetWMProtocols(dpy, win, &wmDeleteWindow, 1);

267     /*** (7) bind the rendering context to the window ***/
268     glXMakeCurrent(dpy, win, cx);

269     /*** (8) make the desired display lists ***/
270     makeDinosaur();

271     /*** (9) configure the OpenGL context for rendering ***/
272     glEnable(GL_CULL_FACE);     /* ~50% better perfomance than no back-face
273                                    * culling on Entry Indigo */
274     glEnable(GL_DEPTH_TEST);    /* enable depth buffering */
275     glEnable(GL_LIGHTING);      /* enable lighting */
276     glMatrixMode(GL_PROJECTION);/* set up projection transform */
277     gluPerspective( /* field of view in degree */ 40.0, /* aspect ratio */ 1.0,
278                     /* Z near */ 1.0, /* Z far */ 40.0);
```

```
279     glMatrixMode(GL_MODELVIEW); /* now change to modelview */
280     gluLookAt(0.0, 0.0, 30.0,   /* eye is at (0,0,30) */
281               0.0, 0.0, 0.0,    /* center is at (0,0,0) */
282               0.0, 1.0, 0.);    /* up is in postivie Y direction */
283     glPushMatrix();             /* dummy push so we can pop on model recalc */
284     glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 1);
285     glLightfv(GL_LIGHT0, GL_POSITION, lightZeroPosition);
286     glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
287     glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1);
288     glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05);
289     glLightfv(GL_LIGHT1, GL_POSITION, lightOnePosition);
290     glLightfv(GL_LIGHT1, GL_DIFFUSE, lightOneColor);
291     glEnable(GL_LIGHT0);
292     glEnable(GL_LIGHT1);        /* enable both lights */

293     /*** (10) request the X window to be displayed on the screen ***/
294     XMapWindow(dpy, win);

295     /*** (11) dispatch X events ***/
296     while (1) {
297         do {
298             XNextEvent(dpy, &event);
299             switch (event.type) {
300             case ConfigureNotify:
301                 glViewport(0, 0,
302                     event.xconfigure.width, event.xconfigure.height);
30
```