Chapter 9

The GLUT API for OpenGL Configuration

GLUT is an older, tried-and-true, cross-platform API for quick and dirty access to OpenGL application infrastructure. GLUT provides a very simple and straightforward CAPI to create windows, manage device input, monitor windowing events, handle a run loop, and so on. It also provides low-level primitive construction elements such as Spheres, Cubes, and Torii. This API is not unique to the Mac: in fact, you'll find it on most every platform. GLUT allows you to quickly prototype some OpenGL code in such a way that you can test it on every platform on which you must deploy. Although not really a good infrastructure for more complex applications, it's a great way to get started. In this chapter, we'll provide only a cursory examination of the API on the Mac because, with only one or two extremely minor exceptions, GLUT on the Mac is the same as GLUT on any platform.

GLUT first arrived in November 1994, as a creation of Mark Kilgard of SGI. It was created as a basic infrastructure for quickly and simply bringing up a window for OpenGL rendering. Over the years, GLUT evolved into a cross-platform API, providing support through its same basic interface to bring windows up for OpenGL rendering on most Unix systems, including Linux, and eventually adding Windows and Mac support. GLUT evolved in scope, too, as it grew beyond its windowing roots to provide a variety of other wrapper functions. These wrapper functions focus on tasks such as device handling, from keyboard and mouse, to SpaceBall, joysticks, and more. There are also wrapper functions for quick and easy creation of objects such as spheres, cones, and cubes. In addition, font handling, video resize functions, render-to-texture capabilities, and basic dynamic function binding are all features that GLUT has acquired over the years.

Although GLUT has evolved to have a lot of capability, the core of what GLUT is remains unchanged: It is a simple and uniform way of bringing up an OpenGL application in a platform-independent way.

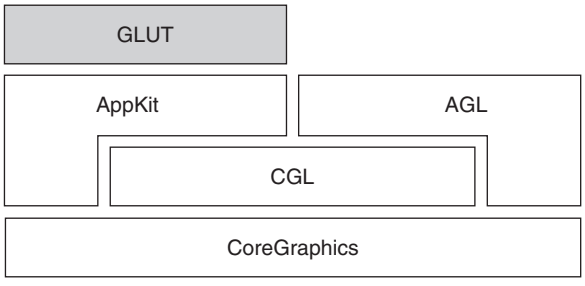

Figure 9-1 GLUT API and Framework in the Overall OpenGL Infrastructure on the Mac

Overview

The GLUT API is part of the overall Apple OpenGL infrastructure. It leverages AppKit for its windowing and event requirements. Figure 9-1 shows where the GLUT API and framework reside relative to other API layers.

The GLUT API lives in your particular SDK framework path or in `/System/Library/Frameworks/GLUT.Framework`. As with other APIs, linking against GLUT requires specification of this framework path (in our code examples, specifying the variable `SDKROOT`). Compiling using headers from the GLUT framework will also require specification of the framework. The relevant locations for building and linking with the GLUT framework are found in Table 9-1.

GLUT is an interface for complete, stand-alone applications that provides a comprehensive set of windowing, event management, device input, OpenGL setup, OpenGL configuration, and a few other miscellaneous functions. If for whatever reason you can't find the interface you need within GLUT, you're best off investigating one layer beneath it, such as Cocoa or CGL. For the most part, GLUT provides a rich feature set that can be used to meet all of your full-screen, windowed, and accelerated off-screen needs. With the fundamentals of GLUT described, and armed with the locations in which to fully explore the framework, let's move directly into GLUT configuration.

Table 9-1 GLUT Headers, Frameworks, and Overview

Framework path	`/System/Library/Frameworks/GLUT.framework`
Build flag	`-framework GLUT`
Header	`#include<GLUT/glut.h>`

Configuration and Setup

Configuring and using GLUT is pretty straightforward, and given what we've covered in prior chapters, it should all feel somewhat familiar. We'll waste no time in this section; we'll just jump right into a code example and cover the only Mac-specific change you'll need to be aware of for GLUT applications on the Mac.

Begin by going to XCode and creating a new project, of type C++ tool, as seen in Figure 9-2. We're choosing a C++ project just because we feel like it and prefer some C++ idioms, rather than because GLUT requires C++. In fact, as mentioned earlier, GLUT is a C-API.

In Figure 9-2, we create a new project; in Figure 9-3, we add the GLUT framework; and in Figure 9-4, we see what the resultant framework should look like. Specifically, in Figure 9-3, navigate to `/System/Library/Frameworks/` and select `GLUT.framework` to add to the project.

Now that we've got a project, we must address the first Mac-specific element—linking against the library. We do that as seen in Figure 9-3, with the result shown in Figure 9-4. This specifies that we will use the GLUT framework to resolve include files and link libraries. The only other Mac-specific element is the way in which we include the headers, as seen in Figure 9-3. On other platforms, the GLUT headers may live in different directories (in fact, they usually

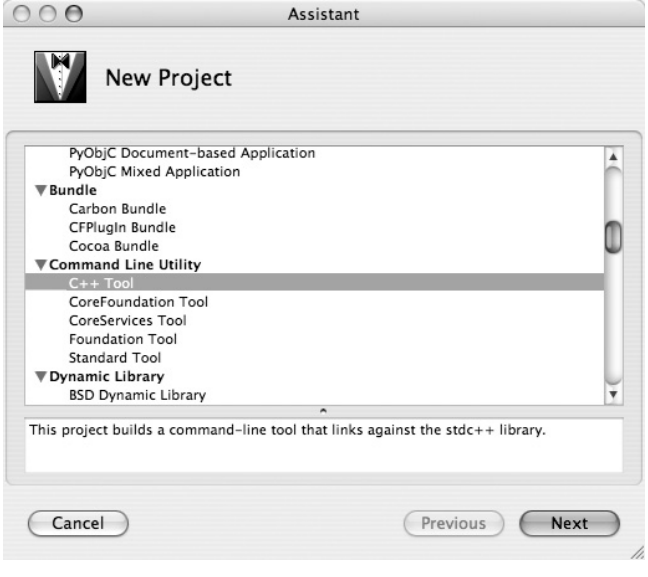

Figure 9-2 New Project Creation for GLUT Application

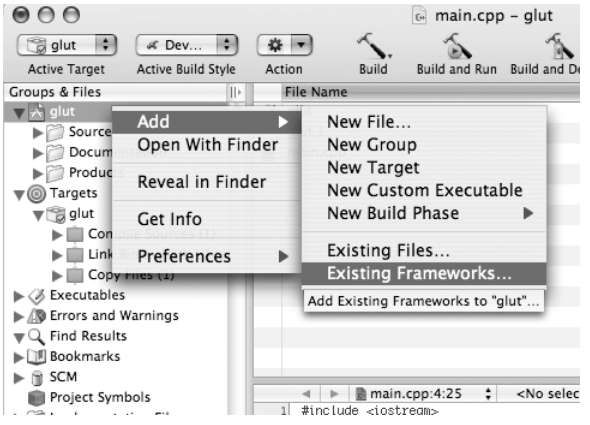

Figure 9-3 Adding a Framework to This Project

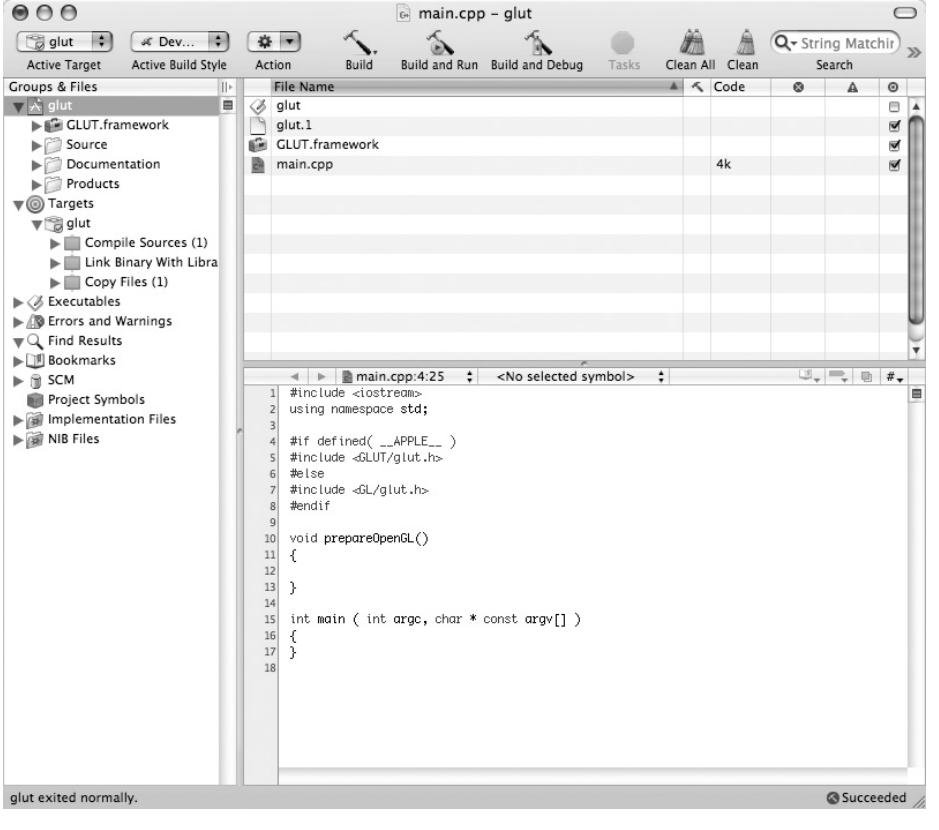

Figure 9-4 Resultant GLUT Project Showing Framework and Sample Code

live in the GL directory), so some wrangling is necessary to ensure that your compiler can find the header file. The code in Example 9-1 performs this operation to include the glut.h header using a preprocessor check to determine whether we're building on the Mac and, if so, to adjust where we find GLUT to use the framework-resolved path. Those are really the only two unique elements to using GLUT on the Mac.

Simple enough. Now let's look at fleshing out this code.

Example 9-1 GLUT Header Inclusion on the Mac

```
#if defined( __APPLE__ )
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

Pixel Format

We'll now look at a complete application, from window creation to GL initialization through swap buffers. This code is presented here for your edification, but not because we plan to explain it in painstaking detail. As we've said before, GLUT is GLUT is GLUT. You'll find that the code we write here will function on many platforms, and the GLUT examples on the Mac are a great way to learn more about how to use the API. In fact, Apple ships a complete set of GLUT examples with its developer tools; you'll find them in /Developer/Examples/OpenGL/GLUT/. Now, let's move on to our code. It renders a

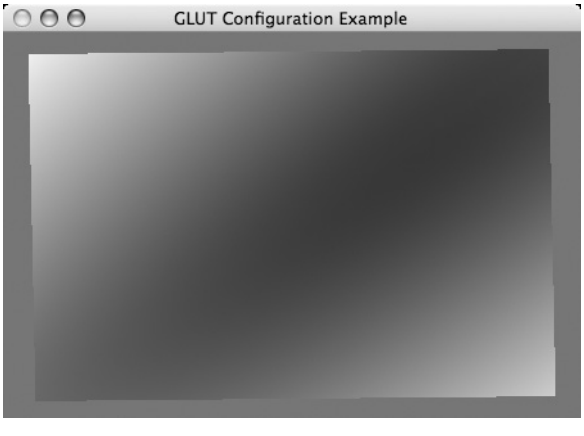

Figure 9-5 GLUT Project Results Showing Visual Selection and Rendered Object

simple animated shape, but doesn't do much else. The results of Example 9-2 are seen in Figure 9-5.

Example 9-2 Basic GLUT Sample Application

```
void prepareOpenGL()
{
    myAngle = 0;
    myTime = 0;
}

void draw()
{

    glClearColor( 0, .5, .8, 1 );
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glRotatef( myAngle, 0, 0, 1 );

    glTranslatef( 0, 0, 1 );
    glColor3f( 0, 1, 0 );
    glBegin( GL_QUADS );
    float ww = .9;
    float hh = .9;
    glTexCoord2f( 0, 0 );
    glVertex3f( -ww, -hh, 0 );
    glTexCoord2f( 1, 0 );
    glVertex3f( ww, -hh, 0 );
    glTexCoord2f( 1, 1 );
    glVertex3f( ww, hh, 0 );
    glTexCoord2f( 0, 1 );
    glVertex3f( -ww, hh, 0 );
    glEnd();

    glutSwapBuffers();
}

void angleUpdate( int delay )
{
    float twopi = 2*M_PI;
    myTime = (myTime>twopi)?0:myTime+.03;
    myAngle = sinf(twopi*myTime);
    glutTimerFunc( delay, angleUpdate, delay );
    glutPostRedisplay();
}

int main ( int argc, char * argv[] )
{
    glutInit( &argc, argv );

    // choose a visual and create a window
    glutInitDisplayString( "stencil>=2 rgb8 double depth>=16" );
```

```
    // this is comparable to glutInitDisplayMode with the
    // tokens below, and achieves a similar effect
    // glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );

    glutInitWindowSize( 450, 300 );
    glutCreateWindow( "GLUT Configuration Example" );

    // initialize our opengl (context is now valid)
    prepareOpenGL();

    // register callback functions
    int delay = 50;
    glutTimerFunc( delay, angleUpdate, delay );
    glutDisplayFunc( draw );
    glutMainLoop();
}
```

GLUT is a good way to bring up a rendering window quickly and efficiently. It also provides a fair degree of specificity for window management. You can use the `glutInitDisplayMode`, as shown in Example 9-2, to specify a variety of flags to set the visual that is used. For example, you can use any combination of the bit flags as described in Table 9-2 to customize which visual you use. These bit fields are described in complete detail in the `glutInitDisplayMode` manual page, and we present only a few in Table 9-2. A simplified version of the use of this visual specification was presented in Example 9-2. This function, along with its bit field settings, allows you a coarse degree of control in the visual qualities of your application.

As we've seen in other chapters, selecting a visual can be a very detailed process, and one that your application needs to specify fully. GLUT provides

Table 9-2 `glutInitDisplayMode` Bit Field Tokens

Token	Description
`GLUT_RGBA` / `GLUT_RGB`	Synonymous tokens to select a visual with RGBA pixel formats. The default if no other format is specified.
`GLUT_SINGLE`	Single-buffered visual token. The default if neither `GLUT_DOUBLE` nor `GLUT_SINGLE` is present.
`GLUT_DOUBLE`	Double-buffered visual token. Has priority if `GLUT_SINGLE` is also present.
`GLUT_ACCUM`	Token for accumulation buffer visuals.
`GLUT_ALPHA`	Token to choose alpha channel visuals.
`GLUT_DEPTH`	Token to select a depth-buffered visual.
`GLUT_STENCIL`	Token to select a stencil-buffered visual.
`GLUT_MULTISAMPLE`	Token to select a multisample visual. Automatically degrades to another visual if multisampling is not available.
`GLUT_STEREO`	Token to select a visual with stereo abilities.

a limited form of this capability through a complementary function called `glutInitDisplayString`. In no way is the GLUT process nearly as complete as the CGL, AGL, or Cocoa methods, but it does allow you to exert a fair degree of control. Among the capabilities exposed through this method, a caller can specify the number of bits in various color or depth channels, the number of samples in multisample visuals, and the policy regarding how to select which visual matches. We present a selection of states that can be specified through such a call in Table 9-3, and a complete description of these flags and their defaults can be found at the manual page: `man glutInitDisplayString`.

So how are these flags used to specify a visual? The tokens in Table 9-3 specify the individual visual elements to be specified. With each, we can also attach an optional policy. The code for doing so requires the use of a standard set of operators with meanings equivalent to those operators' meanings in C code. For example, to specify a visual with all buffer bits, including alpha, of depth 8 or greater, we would write `rgba>=8` as part of our overall string. For other specifications, such as to consider visuals of other constraints, we would use any one of `<`, `>`, `<=`, `>=`, `=`, or `!=`. A final syntax element, the character, is used to specify a match that is greater than or equal to the number specified, but preferring fewer bits rather than more. This is a good way of getting close to your literal specification, but with some fail-over capability, preferring visuals of better quality.

For a complete example of how this specification works, we'll examine a replacement for the call `glutInitDisplayString` in our previous example but now modify it to use this form of visual selection instead. Example 9-3 is set up to try to find a visual with at least 2 bits of stencil precision, double buffered, with an RGBA visual of 8 or greater bits, as closely matching 8 as possible, a 16-bit or greater depth, and multisample anti-aliasing. The results of this change

Table 9-3 `glutInitDisplayString` Policy and Capability Strings

Label	Description
`alpha`	Bits of alpha channel color buffer precision
`red`	Bits of red channel color buffer precision
`green`	Bits of green channel color buffer precision
`blue`	Bits of blue channel color buffer precision
`rgba`	Bits of red, green, blue, and alpha channels color buffer precision
`acca`	Bits of RGBA channels accumulation buffer precision
`depth`	Bits of depth channel buffer precision
`stencil`	Bits of depth channel buffer precision
`single`	Boolean enabling single buffer mode
`double`	Boolean enabling double buffer mode
`stereo`	Boolean enabling quad buffer stereo mode
`samples`	Number of multisamples to use

Figure 9-6 GLUT Project Results Showing Visual Selection and Rendered Object for Anti-Aliased Pixel Format

to Example 9-2 are subtle, because the only differences involve the addition of the stencil and anti-aliasing. The results of the anti-aliasing are visible in Figure 9-6.

For a much more verbose description of these flags, ways to use this initialization call, and more, check the manual page for this call using `man glutInitDisplayString`.

Example 9-3 Initializing a GLUT Visual Using a String

```
glutInitDisplayString("stencil>=2 rgb~8 double depth>=16 samples");
```

Summary

In this chapter, we saw how GLUT works on the Mac, and pointed out the key configuration differences from other platforms. GLUT is useful for rapid prototyping, in that it lets you portably and efficiently bring up a window, configure the visual with a fair degree of specificity, and draw. In essence, this API gets you rendering quickly, although, it doesn't mesh particularly well with the more native ways of integrating OpenGL drawing into a window, especially for the Mac. We devoted the majority of our discussion in this chapter to the minor differences between the Mac and other platforms—specifically, how to include and build with GLUT. This concludes our coverage of GLUT on Mac OS X. We now return you to your regularly scheduled Mac OS X OpenGL programming.