Appendix A. Using OpenGL with FXRuby

Abstract

FOX provides extensive support for OpenGL through its `FXGLCanvas` and `FXGLViewer` widgets, and FXRuby in turn provides interfaces to those classes. By combining FXRuby with the OpenGL interface for Ruby (described below) you can develop very powerful 3-D graphics applications. This chapter gives you the information you'll need to get started.

What is OpenGL?

OpenGL is a platform-independent API for 2D and 3D graphics. The home page is http://www.opengl.org. Because it's a fairly open standard, highly optimized OpenGL drivers are available for most operating systems (including Windows and Linux).

OpenGL Extensions for Ruby

The ruby-opengl extension module, originally developed by Yoshiyuki Kusano and currently maintained by Alain Hoang, provides interfaces to not only the basic OpenGL API, but also the GLU and GLUT APIs. As of this writing, the currently released version is 0.60.1. You can install it by typing:

```
$ gem install ruby-opengl
```

Please note that since I'm not the maintainer of this particular Ruby extension, so I can't really accept bug fixes for it. But if you're having trouble integrating Ruby/OpenGL with FXRuby, let me know and we'll see what we can do.

The FXGLVisual Class

An `FXGLVisual` object describes the capabilities of an `FXGLCanvas` or `FXGLViewer` window. Typically, an X server supports many different visuals with varying capabilities, but the ones with greater capabilities require more resources than those with fewer capbilities. To construct an `FXGLVisual` object, just call `FXGLVisual.new`:

```
aVisual = FXGLVisual.new(theApp, VISUAL_DOUBLEBUFFER)
```

The first argument to `FXGLVisual.new` is a reference to the application object. The second argument is a set of options indicating the *requested* capabilities for the visual. If one or more of the requested capabilities aren't available, FOX will try to gracefully degrade to a working GL

visual; but if you're counting on a specific capability, be sure to check the returned visual to see if it actually supports that capability. For example, say you request a visual with double-buffering and stereographic capabilities:

```
anotherVisual = FXGLVisual.new(theApp, VISUAL_DOUBLEBUFFER | VISUAL_STEREO)
```

Double-buffering is pretty commonplace these days, but stereo may not be available on the system. We can check to see whether the visual we got supports these capabilities by calling the `FXGLVisual#doubleBuffered?` and `FXGLVisual#stereo?` methods:

```
anotherVisual = FXGLVisual.new(theApp, VISUAL_DOUBLEBUFFER | VISUAL_STEREO)
if anotherVisual.doubleBuffered?
  puts "It's double-buffered."
else
  puts "It's single-buffered."
end
if anotherVisual.stereo?
  puts "It's stereo."
else
  puts "It isn't stereo."
end
```

Some `FXGLVisual` object must be associated with every `FXGLCanvas` or `FXGLViewer` window, but you don't need to have a separate `FXGLVisual` object for each window. For most applications, you can just construct a single `FXGLVisual` object that's shared among all the OpenGL windows.

The FXGLCanvas Class

The `FXGLCanvas` widget provides a very simple OpenGL-capable window with minimal functionality. To construct an `FXGLCanvas`, call `FXGLCanvas.new`:

```
glCanvas = FXGLCanvas.new(parent, vis)
```

The first argument to `FXGLCanvas.new` is the parent (container) widget and the second argument is the `FXGLVisual` that should be used for this window.

OpenGL objects and the FXGLViewer

The `FXGLViewer` widget provides a higher-level OpenGL-capable window with a lot of built-in functionality. To construct an `FXGLViewer`, call `FXGLViewer.new`:

```
glViewer = FXGLViewer.new(parent, vis)
```

The first argument to `FXGLViewer.new` is the parent (container) widget and the second argument is the `FXGLVisual` that should be used for this window.