Magician Programmer's Guide For Java

Alligator Descartes

descarte@arcana.co.uk

September 7, 1999

"The Magician Programmer's Guide" was written, illustrated and prepared by Alligator Descartes for Arcane Technologies Ltd. using the LaTeX typesetting environment.

Contents

List of Figures

Preface

Magician is an interface or *binding* that enables developers to write high-performance 3D applications in Java using the API defined by *OpenGL*. OpenGL is a platform-independent API defined by the *OpenGL Architecture Review Board*, or *ARB*, for writing 3D applications.

This guide serves as an explanation of how Magician implements OpenGL and how you can write OpenGL applications using Magician. This guide is not currently designed to be an introduction to, or explanation of, OpenGL. A more detailed function-by-function reference guide for Magician can be found in downloadable and on-line formats at the Magician WWW site and also on the Magician CD.

Highlights of Magician

Magician leverages functionality from Java and integrates this new network-centric functionality into OpenGL, *e.g.*, for transparently loading texture data from arbitrary URLs. Using extensions of standard AWT components, OpenGL-aware objects can be used within your GUIs and can be manipulated in the same way as the bundled AWT classes. Furthermore, Magician preserves complete portability between both platforms and Java Virtual Machines enabling you to deploy applications on over 90% of desktops on the planet immediately. You will also benefit from the applet features of Java enabling you to deploy full-blown OpenGL applications on the World Wide Web.

Magician also enhances OpenGL functionality by providing dynamic access to extensions, simple and powerful multi-threaded rendering using Java's in-built threading and in-built debugging and profiling streams.

Getting Magician

Magician is distributed from Arcane Technologies WWW site at:

```
http://www.arcana.co.uk/products/magician
```

The WWW site also contains information on Magician, on-line API documentation and this manual, mailing list archives and *Frequently Asked Questions* on Magician.

Magician can be downloaded for evaluation purposes, and as such is "locked" by slowing down after a number of screen refreshes. Unlocked versions are distributed upon licensing Magician from us. For more information on our licensing policies, please read the Licensing pages on the WWW site.

Recommended Reading

The following book should be purchased as an excellent explanation of how OpenGL works. This programmer's guide does not explain how OpenGL can be programmed simply how Magician implements OpenGL. If you do not know how to use OpenGL, buy this book!

```
''OpenGL Programming Guide''
      -- Mason Woo, Tom Davis, Jackie Neider
         Published by Addision Wesley, 2nd edition, 1996
         ISBN 0-201-46138-2
```

There are 3 other books in this series[1] that may be perused for additional information on the workings of OpenGL for different platforms.

The "Blue Book" is

```
''OpenGL Reference Manual''
      -- OpenGL ARB
         Published by Addison Wesley, 2nd Edition, 1996
         ISBN 0-201-46140-4
```

[1]The above book is known as the "Red Book". The others are the "Blue Book", "Green Book" and "White Book" (or "Alpha Book")!

and is a function-by-function reference to OpenGL and GLU. The book is invaluable for quick reference when writing OpenGL-based applications.

The “Green Book” is a book specific to using OpenGL under X Windows and contains lots of information on special effects and bizarre input devices that can input data to OpenGL. This book is the most advanced in the series.

```
‘‘OpenGL Programming for the X Window System’’
      -- Mark Kilgard
         Published by Addison Wesley, 1st Edition, 1996
         ISBN 0-201-48359-9
```

The final book in the series is the “White Book” and relates to programming OpenGL under Win32 for Windows 95 and NT. The book contains a gentle introduction to OpenGL and 3D graphics and discusses using OpenGL and MFC. This is an excellent beginners book for OpenGL programming, but not particularly in-depth.

```
‘‘OpenGL Programming for Windows 95 and Windows NT’’
      -- Ron Fosner
         Published by Addison Wesley, 1st Edition, 1996
         ISBN 0-201-40709-4
```

All of the above books can be ordered from Arcane Technologies in conjunction with Amazon.com at the following URL

```
http://www.arcana.co.uk/books.html
```

Supported Configurations

Magician currently supports a wide variety of operating systems and Java Virtual Machines. The Magician WWW pages at Arcane Technologies Ltd.’s site should be consulted for up-to-the-minute information on supported configurations.

Operating System	**Java Virtual Machine**	**OpenGL**
Windows 95/98/NT	Microsoft IE 4	SGI OpenGL / Microsoft OpenGL
	Microsoft IE 3	SGI OpenGL / Microsoft OpenGL
	Microsoft JVM 2.x, 3.x	SGI OpenGL / Microsoft OpenGL
	Sun JDK-1.1.x	SGI OpenGL / Microsoft OpenGL
	Sun JDK-1.2 / Java-2	SGI OpenGL / Microsoft OpenGL
	Netscape Navigator 4.04+	SGI OpenGL / Microsoft OpenGL
	Symantec Visual Cafe V2.5+	SGI OpenGL / Microsoft OpenGL
	SuperCede V2.04+	SGI OpenGL / Microsoft OpenGL
Linux	Sun JDK-1.1.x	Mesa
	RedHat Sun JDK-1.1.x	Mesa
	Cambridge OpenGroup JVM	Mesa
SPARC/Solaris	Sun JDK-1.1.x	Sun OpenGL
	Sun JDK-1.2 / Java-2	Sun OpenGL
Intel/Solaris	Sun JDK-1.1.x	Mesa
	Sun JDK-1.2 / Java-2	Mesa
Irix	Sun JDK-1.1.x	SGI OpenGL
OS/2	IBM JDK-1.1.x	IBM OpenGL
MacOS 8	Apple MRJ 2.1	Conix/Apple OpenGL / Mesa
	Metrowerks	Conix/Apple OpenGL / Mesa
AIX	IBM JDK-1.1.x	IBM OpenGL / Mesa

Acknowledgments

The author would like to thank (in no particular order) Jason "Mr. Wiggles" Osgood, Rob DeMillo, Brian Hook, Martin McCarthy, Adrian Cook, Gary McTaggart, Rob Povey, Craig Setera, Tom Lasseter, Jeff Meredith, Jeff White and all other users of Magician for their support. They will, however, be damned for all eternity for reporting bugs! `:-)`

Brian Paul should be awarded beer for writing and maintaining the excellent "OpenGL-a-like" *Mesa* library. Michael Gold requires thanks for providing support on the nVidia OpenGL drivers. Mark Kilgard will have a star named after him for allowing usage of the conversions of his demonstration programs bundled with Magician and for his excellent examples of advanced OpenGL techniques.

Documentation Feedback

If you find any parts of this manual confusing, misleading or downright wrong, please let us know about it! You should address all documentation feedback to

```
magician-docs@arcana.co.uk
```

Please state the date from the front of your manual as well as any other pertinent page information.

Chapter 1

An Overview of Magician

The simplest way to introduce Magician to you is to present a short example program then explain each aspect of that program. The subsequent chapters in this book will then explain each individual aspect of Magician in more detail.

The example we shall be exploring is the oft-seen "white rectangle" example that is presented near the beginning of the "OpenGL Programming Guide". This program simply draws a white rectangle on a black background and the output can be seen in Figure 1.1.

The White Rectangle Demo

First off, we need to declare some basic Java constructs for identifying the class file, putting the class in a package and importing various other packages that we'll be using in our application. All the "core" Magician classes live in the `com.hermetica.magician` namespace with "utility" classes belonging to the `com.hermetica.util3d` package. You will always have to import the former package.

```
/** whiteRectangle.java -- Demonstration Magician program */

/** Import classes from some necessary places */
import java.awt.*;
import com.hermetica.magician.*;
```

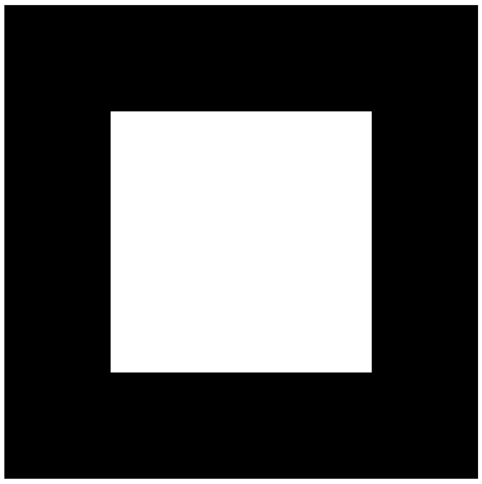

Figure 1.1: Drawing A White Rectangle

```
/**
 * This is a short demo illustrating the basic structure
 * of a Magician application. This doesn't handle keyboard
 * or window events.
 */
public class whiteRectangle extends Frame
    implements GLEventListener {
    .
    .
    .
```

This class will create a new window *via* the standard Java AWT `Frame` class. The code has also implemented the `GLEventListener` interface which requires that we implement the four core methods defined in this interface. The `GLEventListener` interface gives Magician applications a standard structure and form.

The next chunk of code sets up some basic variables that we will be using in the course of the program namely a `GLComponent` for rendering onto and a `CoreGL` object that encapsulates the OpenGL rendering pipeline. The program is not using any of the functions defined within GLU, so there's no need to have a GLU encapsulation object at all.

.

```
    .
    .

/** OpenGL pipelines used to render the scene */
GL gl_ = null;
CoreGL coregl_ = new CoreGL();

/** OpenGL rendering context */
GLComponent glc = null;

/** Kicks the whole thing off */
public static void main( String argv[] ) {
    whiteRectangle t = new whiteRectangle();
  }

public whiteRectangle() {
    /** Assign the OpenGL pipeline */
    gl_ = coregl_;

    /**
     * Set the layout manager for the Frame. This is BorderLayout
     * which will cause the children ( the GLComponent ) to
     * be resized to fill the Frame upon resizing the Frame.
     */
    setLayout( new BorderLayout() );

    /**
     * Create a new GLComponent from the factory
     * at size ( 200x200 )
     */
    glc = (GLComponent)GLDrawableFactory.createGLComponent( 200, 200 );

    /** Add the GLComponent to the Frame and display it */
    add( "Center", glc );
    pack();
    show();
    .
    .
    .
```

At this stage, your application would pop up a new window with a blank canvas within it since your `GLComponent` has neither a *rendering context* nor a `GLEventListener` associated with it. The rendering context is basically a link between the OpenGL pipeline (a `CoreGL` object) and the window onto which the rendering will occur (a `GLComponent`).

This next section of code will set up the context *via* the `GLCapabilities` mechanism and also register a `GLEventListener` with the `GLComponent`. Finally, you can use the `initialize()` method defined with `GLComponent` to tell the component to start processing events and initialize itself.

```
          .
          .
          .
        /**
         * Set up the capabilities of the OpenGL
          * rendering context
          */
        GLCapabilities cap = glc.getContext().getCapabilities();
        cap.setDoubleBuffered( GLCapabilities.DOUBLEBUFFER );
        cap.setDepthBits( 12 );
        cap.setColourBits( 24 );
        cap.setPixelType( GLCapabilities.RGBA );

        /**
         * Add the GLEventListener to handle GL events
         * and initialize!
         */
        glc.addGLEventListener( this );
        glc.initialize();
      }
      .
      .
      .
```

Each `GLComponent` is created with a `GLContext` associated with it. It is generally easiest to use this context and manipulate it *via* the `getContext()` method rather than create your own contexts and associate them with `GLComponents`.

The `GLCapabilities` class allows you to specify what visual capabilities the rendering context should have. For example, is it *double-buffered* or *single-buffered*? Will it be rendering *true colour* or *indexed colour* images? The permutations of visual quality that can be addressed by *GLCapabilities* is vast! Each `GLContext` object has a `GLCapabilities` object pre-created and associated with it. Therefore, using the `getCapabilities()` method is the quickest way to set up visual capabilities. However, you can do it manually in cases where you might wish to initialize three identical rendering contexts.

Since this class implements the GLEventListener interface, you must also provide at least stub implementations of each of the methods that this interface defines. These methods are display(), reshape() and initialize() which handle the three basic fundamental operations that OpenGL programs exhibit, that is, what happens when the window is drawn, what happens when the window is resized and what should happen when the window is first created and initialized. There is also a fourth GLEventListener method that allows you to more closely control the internal operations of the drawing surface called getGL(). Almost every OpenGL application will perform these functions and the GLEventListener interface simply formalises this into a structure that should be followed.

```
    .
    .
    .
  /**
   * Executed exactly once to initialize the
   * associated GLComponent
   */
  public void initialize( GLDrawable component ) {
      /**
       * Set the background colour when the GLComponent
       * is cleared
       */
      gl_.glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
    }

  /** Handles resizing of the GLComponent */
  public void reshape( GLDrawable component, int x, int y,
                       int width, int height ) {
      gl_.glViewport( component, x, y, width, height );
      gl_.glMatrixMode( GL.GL_PROJECTION );
      gl_.glLoadIdentity();
      gl_.glOrtho( 0.0, 1.0, 0.0, 1.0, -1.0, 1.0 );
      gl_.glMatrixMode( GL.GL_MODELVIEW );
    }

  /** This method handles the painting of the GLComponent */
  public void display( GLDrawable component ) {
      /** Clear the colour buffer */
      gl_.glClear( GL.GL_COLOR_BUFFER_BIT );

      /** Set the drawing colour to white */
      gl_.glColor3f( 1.0f, 1.0f, 1.0f );
```

```
            /** Draw a square */
            gl_.glBegin( GL.GL_POLYGON );
                gl_.glVertex3f( 0.25f, 0.25f, 0.0f );
                gl_.glVertex3f( 0.75f, 0.25f, 0.0f );
                gl_.glVertex3f( 0.75f, 0.75f, 0.0f );
                gl_.glVertex3f( 0.25f, 0.75f, 0.0f );
            gl_.glEnd();
          }

        /**
         * Returns a valid OpenGL pipeline when asked
         * by GLComponent
         */
        public GL getGL() {
            return gl_;
          }
      }
```

This demonstration is extremely straight-forward, but illustrates many of the powerful features inherent in Magician.

OpenGL-aware AWT Components

The `GLComponent` class is a special AWT component that Magician provides to allow you to display 3D graphics within standard applications but can be manipulated in the same way as any other AWT component class such as `Button` or `Canvas`. This feature lets you create potentially complex GUIs that have windows in which rendering is occurring.

Magician also provides an event handling mechanism in the style of Java 1.1 *listeners* called `GLEventListener`. This allows you to quickly route certain types of event to a particular `GLComponent`.

For example, when you resize the window in the above application, the AWT `BorderLayout` layout manager will automatically resize all the components within that window to fill the window. This causes the `reshape()` method in any `GLEventListener`s registered against any `GLComponent` objects present in that window to be invoked.

Similarly, if a repaint of the window is required because you have dragged the window about or moved the window in front of other windows, AWT will defaultly call a method called `repaint()` in all affected components.

Within Magician, `repaint()` calls against `GLComponent` objects are intercepted and the `display()` method in any registered `GLEventListener`s are invoked.

Magician also allows you to build animation into your applications extremely quickly. Invoking the `start()` method against a `GLComponent` will initialize a thread internal to the `GLComponent` object which regularly makes calls to the `GLEventListeners`' `display()` methods. Therefore, if you place any animation update logic into the `display()` method, it will animate the objects being rendered.

For example, if you wanted to make the rectangle rotate you could rewrite the `display()` method as

```
public void display( GLDrawable component ) {
    /** Clear the framebuffer */
    gl_.glClear( GL.GL_COLOR_BUFFER_BIT );

    /** Set the drawing colour */
    gl_.glColor3f( 1.0f, 1.0f, 1.0f );

    /** Save the current viewpoint settings */
    gl_.glPushMatrix();
        /** Rotate the rectangle by ``angle'' degrees */
        gl_.glRotatef( angle, 0, 0, 1 );

        /** Draw the rectangle */
        gl_.glBegin( GL.GL_POLYGON );
            gl_.glVertex3f( 0.25f, 0.25f, 0.0f );
            gl_.glVertex3f( 0.75f, 0.25f, 0.0f );
            gl_.glVertex3f( 0.75f, 0.75f, 0.0f );
            gl_.glVertex3f( 0.25f, 0.75f, 0.0f );
        gl_.glEnd();
    /** Restore the saved viewpoint settings */
    gl_.glPopMatrix();

    /** Update the angle to spin the rectangle about */
    angle++;
    if ( angle >= 360.0f ) {
        angle -= 360.0f;
      }
  }
```

The update logic is simply to increment the `angle` variable. Therefore, upon repeated calls of the `display()` method, the angle increases causing the rectangle to spin. You can implement more complex time-critical animations in Magician, but this is a simple way to perform animation of a non-complex nature.

Composable Pipelines

Magician defines a group of classes known as the *pipeline* classes. These classes define the actual OpenGL and GLU functions that form the heart of your programs.

Magician treats these functions in an unusual way in that the OpenGL and GLU functions are declared within an *interface* that is implemented by any pipeline classes. This ensures that all the functions that need to be present to form a complete OpenGL and GLU implementation have been defined and it also gives us the flexibility to *compose* pipelines.

For example, Magician uses a class called `CoreGL` to execute the OpenGL functions[1]. In the example shown above, an instance of `CoreGL` is constructed and all OpenGL statements are executed against that.

However, we might also construct an object from the class `TraceGL` which is another Magician pipeline class. This class simply prints the name of each method being executed but does not execute the corresponding OpenGL function. Therefore, if we were to change `CoreGL` to `TraceGL` in the above example, the OpenGL function names would be printed, but nothing would be drawn on the screen.

Since both of these pipelines are inherently useful, combining them to both print the name of the OpenGL function being called and execute the OpenGL function itself would be a real boon to you for debugging applications.

To create a `TraceGL` pipeline that also executes the OpenGL functions, you simply need to write

```
TraceGL gl_ = new TraceGL( new CoreGL() );
```

Furthermore, Magician provides the functionality to swap pipelines on a statement to statement basis. For example, if you wished to only trace one

[1]There is also a corresponding `CoreGLU` class for executing GLU functions.

particular section of your code, you might wish to swap the trace pipeline with the core pipeline then swap it back again.

Ordinarily that code might look like

```
 CoreGL coregl_ = new CoreGL();
 TraceGL tracegl_ = new TraceGL( coregl_ );

 /** Execute some code with CoreGL */
 coregl_.glBegin( GL.GL_POLYGON );
    /** Now, trace the vertex calls */
    tracegl_.glVertex3f( 0.25f, 0.25f, 0.0f );
    tracegl_.glVertex3f( 0.75f, 0.25f, 0.0f );
    tracegl_.glVertex3f( 0.75f, 0.75f, 0.0f );
    tracegl_.glVertex3f( 0.25f, 0.75f, 0.0f );
/** Switch back to coregl */
coregl_.glEnd();
```

This is just silly and is also compile-time specific. Since each pipeline class in Magician implements the GL interface, you can runtime switch pipelines in the following way

```
GL gl_ = null;
CoreGL coregl_ = new CoreGL();
TraceGL tracegl_ = new TraceGL( coregl_ );

/** Execute some code with CoreGL */
gl_ = coregl_;
gl_.glBegin( GL.GL_POLYGON );
    /** Switch to tracing */
    gl_ = tracegl_;
    gl_.glVertex3f( 0.25f, 0.25f, 0.0f );
    gl_.glVertex3f( 0.75f, 0.25f, 0.0f );
    gl_.glVertex3f( 0.75f, 0.75f, 0.0f );
    gl_.glVertex3f( 0.25f, 0.75f, 0.0f );
    /** Switch back to core */
    gl_ = coregl_;
gl_.glEnd();
```

This code may not look radically different, but imagine if you wished to enable tracing from a menu in your application? The second approach allows you to switch pipelines at *runtime* whereas the first method is compile-time restricted.

I shall be discussing pipelines in more detail in Chapter 3.

Associated Magician Demo Programs

testGL.java This demonstration program simply creates a new window and draws a shaded black to blue polygon within it.

mtpaperplane.java This demonstration shows simple animation using the in-built animation features of `GLComponent` and the use of profiling pipelines.

Chapter 2

OpenGL Architecture

The architecture of Magician has been largely defined by the open architecture that OpenGL implements. OpenGL has a fairly simple design, but one which allows software and hardware developers to wield considerable power.

Similarly, there are extensions and abstractions present in the architecture of Magician that allow transparently portable use of all aspects of the OpenGL architecture especially within a multi-threaded and object-orientated environment such as Java. A balance between feature-richness and "recognisable" OpenGL has to be found.

Introduction to OpenGL

OpenGL is the industry standard API for 3D graphics development. The standards for API development are driven by a Consortium formed by many of the major players in the 3D graphics arena including Silicon Graphics, Sun Microsystems, Microsoft and Evans and Sutherland. The benefits of this process are mainly that no one company controls the API and that the API design benefits from the shared knowledge of many individuals[1]. The

[1]There is a downside to this. Software designed by committee does not have a rapid development cycle so changes to the OpenGL API may take time. However, OpenGL has been remarkably well designed from the outset so, in this case, development lag is not really an issue. Additionally, vendors may define *extensions* to OpenGL that do not need to be ratified by the ARB.

body which regulates the development of OpenGL is the *OpenGL Architecture Review Board* or *OpenGL ARB* who meet on a quarterly basis to discuss extensions and enhancements to OpenGL.

The design of the OpenGL API revolves around 4 main areas

- "Core" OpenGL API
- "Core" OpenGL Utility API or *GLU*
- Extensions to the OpenGL API[2]
- Window-system specific interfaces such as GLX

The first two APIs are completely standard and identical across all operating systems implementing OpenGL. These two APIs essentially *are* OpenGL. They provide what can be classed as being an OpenGL *pipeline* through which data is transformed, manipulated and *rasterized* into images.

However, in order to remain operating-system neutral, the core OpenGL and GLU pipelines do not actually produce any images. Output of rendered images and input into software using OpenGL is strictly regulated by the window-system specific functions including the `glX*` functions for X Windows-based systems and the `wgl*` functions for Windows 95/NT systems.

The functions defined in these protocols act as a conduit between the input and output devices of the user's computer and the OpenGL and GLU rendering pipelines. Therefore, the user can perform some action in the application using the mouse or keyboard[3] which affects the operation of the core OpenGL and GLU pipelines in some defined way. The application may then re-render the scene and the window-system specific protocols return an image that can be displayed on the display device[4].

[2]Extensions are one way in which new additions to the core OpenGL API may be tested and revised prior to merging into the core API. This also allows additional functionality to be added quickly without requiring OpenGL ARB activity.

[3]Or any other relevant input device such as a graphics tablet, VR headset or whatever.

[4]In common parlance, the OpenGL and GLU pipelines produce an *opaque* framebuffer of data which is a standard definition of what has been rendered. X Windows uses `XImage` or `Pixmap` data types to copy image data onto the screen and Windows 95/NT use bitmaps. Therefore, the window-system specific protocols *automagically* convert the opaque framebuffers into a usable image format for the pertinent window system.

Extensions are a slightly more complicated beast as there are several extensions which are standard across platforms providing extensions to the core OpenGL API. There are also extensions to each window-system specific protocol which are locked to that particular window-system. However, to make matters even more complicated an extension to a particular window-system, *e.g.*, X Windows, may not be supported across all X Windows platforms. A good example of this is the relatively new VideoBuffer extension which requires several pieces of SGI-specific hardware to run. Therefore, even though both SGI Irix and Linux run X Windows, only the SGI version of OpenGL will support the VideoBuffer extension.

The Architecture Of Magician

The architecture of Magician has been designed with two main goals. Firstly, to provide a familiar programming environment to developers used to programming OpenGL in C. Secondly, to harness the power inherent in the additional features of the Java programming language such as multi-threading and object-orientation.

In addition to the above benefits, ensuring cross-platform and window-system portability was a must to allow developers to write Java software that was truly "write once, run anywhere". The inclusion of platform- or window-system specific information would render Magician far less portable.

Magician is designed around a small set of classes and interfaces that provide either direct access to OpenGL or an abstracted API to system-specific functions, such as `glX*` and `wgl*` functions. There are also some auxiliary classes that are supplied for developer usage, but are not considered part of the core architecture.

OpenGL and GLU Pipelines

The central classes in Magician define the access paths into the native OpenGL implementation, *i.e.*, functions such as `glBegin()`, `glVertex3f()` and so on. There is a separate class for the GLU functions.

The GL and GLU classes are implemented as *interfaces* which define all the OpenGL and GLU functions available within standard OpenGL 1.1. There are also *polymorphic* methods for all the OpenGL functions. For example, the functions that define 3-value vertices are:

```
glVertex3s( short, short, short );
glVertex3f( float, float, float );
```

To make life easier for you and to use a powerful feature of Java, there exists a method called `vertex()` that has *overloaded arguments*, that is, if you invoke the `vertex()` method with `short` values as arguments, Magician will route the call down to the underlying `glVertex3s()` method. Similarly, if you invoke `vertex()` with `float` arguments, Magician will call `glVertex3f()` for you. This feature makes type-matching your functions a lot simpler than explicit specification as in the standard OpenGL 1.1 specification.

Additionally, because Magician implements the core OpenGL and GLU functions as an interface, you can mix and match different pipeline classes within your program. Magician has several of these alternative pipelines distributed with its core including tracing and profiling pipelines. This powerful feature lets you swap pipelines on a statement to statement basis allowing you to finely tune hotspots in your code or debug sections that have bugs present in them.

Rendering Contexts & Visual Capabilities

Rendering contexts belong in the category of the specification that is not platform- and architecture-neutral. Each window system has its own protocol for linking the window system to the OpenGL pipeline. Therefore, to be truly portable, Magician must subsume the varying window system differences internally and provide a unified, abstracted API to the developer. This will not only allow you to write immediately portable code without needing to know or care about the underlying target platforms but also enables end-users to use the same application across a variety of platforms from Windows PC to high-end SGI workstations.

The manner in which rendering contexts are created is fairly standard across platforms in that you specify what sort of *capabilities* you wish the rendering context to have. The underlying window-system then checks to see if an appropriate context is acquirable or not. In Magician, the underlying window-system code is hidden and a standard API for capability specification and context operations is provided.

To effect this, Magician has two separate classes that it uses. These are `GLCapabilities` which defines a set of methods for setting and querying capability values and `GLContext` which defines methods for manipulating the rendering context itself. The `GLContext` uses a `GLCapabilities` object associated with it to specify what sort of context to use.

Drawing Surfaces

Every rendering pipeline terminates at the stage at which an image is displayed on a screen[5]. The window, or drawing surface, that the rendering context spits data onto is perhaps the most platform-specific aspect of graphics software as it can vary wildly between platforms.

Magician extracts the core functionality that a drawing surface requires and blends it with the functionality that the Java AWT provides to form the `GLComponent` class. A further benefit is that the Java AWT Event handling mechanisms are similar to those implemented by the popular *GLUT* library written by Mark Kilgard. This helps ensure that developers accustomed to using GLUT will feel at home using Magician.

The encapsulation of a drawing surface as a Java AWT `Component` is quite a powerful tool as it can be included within other Java AWT `Container` objects and treated in exactly the same way as any other `Component`. In this way, powerful GUIs may be built with in-built 3D rendering support that can be interacted with using standard GUI components.

Magician also supports an offscreen drawing surface called `GLOffscreenBuffer` which allows you to render into a hidden framebuffer which can be copied at some stage to a visible drawing surface.

Furthermore, Magician defines an interface called `GLDrawable` which all drawing surface classes such as `GLComponent` and `GLOffscreenBuffer` implements. This allows you to attach `GLEventListener`s to any sort of drawing surface that Magician supports without changing your code.

[5]Or rendered to an offscreen image or other storage.

Magician Architecture Summary

The Magician architecture can be illustrated in the Figure 2.1. This shows that within any application, only one OpenGL “state machine” is created. That is, the application will set various states within OpenGL which are *sticky*. Since a simple architecture involving only a state machine would only allow for one active window within an application, the concept of rendering contexts exists allowing rendering to occur onto multiple windows within a single application.

A rendering context contains a copy of the state of the OpenGL state

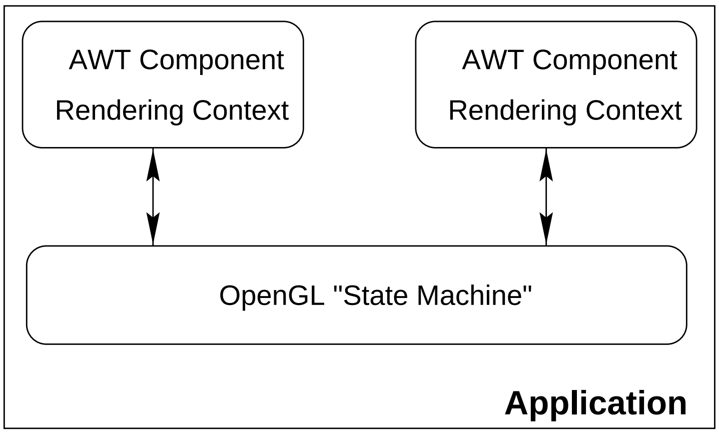

Figure 2.1: Magician Architecture

machine at any given time. Each window within Magician has an associated context such that when that window wishes to render something, the associated context is copied into the state machine and used. When another window wishes to perform some drawing, the previous context is saved and the new context copied in place. This preserves the state of the OpenGL state machine for each window and ensures that they do not interfere with each other.

Therefore, all OpenGL and GLU methods operate on the *current context* in the state machine. Through sophisticated context management, Magician allows you to perform true multi-threaded rendering with OpenGL.

In Magician, the OpenGL state machine corresponds to the composable OpenGL and GLU pipeline classes; the rendering contexts are encapsulated by the `GLContext` class and the drawing surfaces, `GLComponent`. We shall discuss each of these aspects in subsequent chapters.

Magician and the OpenGL ARB

One of the questions we tend to get asked most frequently is "what's happening with Magician and standardisation within the OpenGL ARB?". Another popular one is "if the ARB standardise on something that isn't Magician, what'll happen to Magician?".

Firstly, Arcane are part of a working group established by the ARB to define a specification for Java OpenGL bindings. We have archived the mailing list discussions of this group on the Magician WWW site for your perusal if you're interested in seeing what's being talked about.

Secondly, we're confident that Magician is the best solution available for writing OpenGL programs in Java because of our stability, performance and innovative architecture that no other Java OpenGL bindings even come close.

Chapter 3

OpenGL and GLU Composable Pipelines

The OpenGL and GLU pipeline classes contain the declarations for the portable functions defined within the OpenGL 1.1 Specification. That is, anything that's not either an extension or a window-system specific function.

GLU is not actually part of the "core" OpenGL, but contains functions used so commonly that virtually all OpenGL implementations are distributed with an implementation of the GLU library. To discern between core OpenGL and GLU, Magician defines them in different classes.

"Composable" Pipelines

Magician is unusual in that it supports the concept of *composable pipelines*. Composable pipelines are basically the core OpenGL functions with additional transparent functionality, for example, in-built tracing or profiling.

The base Magician pipeline classes are implemented as Java *interfaces* which define all of the required functions that any pipeline implementation *must* support. By using interfaces, developers have the ability to transfer control flow between different types of pipeline on an instantaneous basis as well as write their own pipeline implementations that will be guaranteed to be compliant with the core Magician pipelines.

The OpenGL and GLU pipeline classes can also be *stacked* on top of each other to combine various aspects of functionality. For example, you might wish error checking enabled *via* the `ErrorGLU` and `ErrorGLU` classes, but you also might want to profile your code *via* the `ProfileGL` and `ProfileGLU` classes. Magician allows you to do both by pipeline stacking.

The general concept of this is that there exists a “stack” off which the `CoreGL` and `CoreGLU` classes are always the base. However, on top of those classes any other pipeline classes can be added by creating them as *children* of the preceding class. Now, when you execute an OpenGL or GLU function the function call executes the corresponding method in the given pipeline class, then executes the corresponding method within that class’ parent and so on until the *actual* OpenGL or GLU function is called in `CoreGL` or `CoreGLU`. Figure 3.1 illustrates the operation of this feature of Magician and I shall be discussing the dynamic switching and extending of pipelines in Section 3.

In addition to the standard core OpenGL and GLU functions, the OpenGL

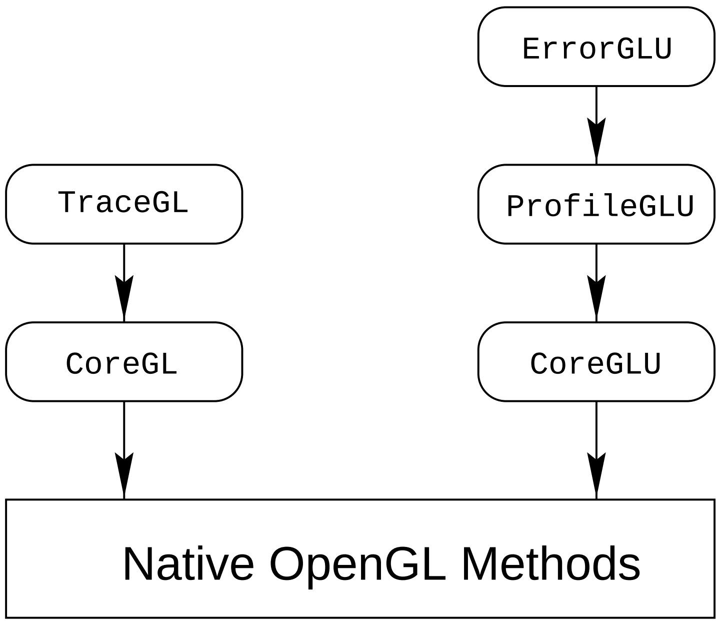

Figure 3.1: OpenGL and GLU Pipeline Stacks

and GLU interfaces also define *polymorphic*, or *overloaded* methods for all the standard OpenGL calls. For example, to specify a two-dimensional point in space in OpenGL, there are a battery of methods that can be used depending on the data type with which you want to express the coordinates with. A few of these methods are:

```
glVertex2s( short x, short y );
glVertex2f( float x, float y );
glVertex2b( byte x, byte y );
```

To remember all these functions can be a bit laborious so, to sidestep this, Magician uses the powerful Java feature of "overloading methods". Using this feature, all the vertex-specification methods can be accessed using a single method called `vertex()`. The neat trick is that this one method will accept argument of many different types. For example,

```
vertex( short x, short y );
vertex( float x, float y );
vertex( byte x, byte y );
```

is a lot easier to remember! The overloaded methods are simply wrappers around the underlying original methods and are therefore a little slower to execute.

Finally, the interfaces define *alternative names* for each OpenGL method. In the C implementation of OpenGL, each function is prefixed with `gl` to uniquely identify that function as being part of the OpenGL namespace. However, in an object-orientated environment such as Java, manual imposition of a namespace is not required. However, to ensure a smooth transition from C and C++ to Java, Magician retains the original OpenGL function names. This also makes porting between C/C++ and Java a less time-consuming operation.

However, to also appease enthusiasts of namespace purity, each OpenGL function is also available with the leading `gl` stripped off and the first letter lower-cased. These generally conform to the overloaded methods discussed *supra.*

OpenGL and GLU Constants

One of the prickly problems regarding porting existing C/C++ code to Java is that you cannot simply use global variables or defined values in the same way as you can with the former languages.

For example, a fragment of C OpenGL code might look like

```
glBegin( GL_TRIANGLES );
    glVertex3f( 100.0, 100.0, 100.0 );
    glVertex3f( 200.0, 200.0, 50.0 );
    glVertex3f( 500.0, 1.0, -400.0 );
glEnd();
```

where the `GL_TRIANGLES` value is a globally available constant defined by the OpenGL include files. The corresponding Java code would generally require that the `GL_TRIANGLES` value be defined within a particular Java class and be referred to as `GL.GL_TRIANGLES` or similar.

This is the approach that we took for the *1.0.0* release of Magician, with all the OpenGL constants being defined with the `GL` interface and the GLU constants in the `GLU` interface. However, we were fairly sure we could do something more to make porting from C/C++ much simpler for you.

Magician *1.1.0* introduces two new interface classes in the Magician namespace called `GLConstants` and `GLUConstants` which contain all the constant definition for OpenGL and GLU respectively. To access these constants, you need only implement the appropriate interface within your class. For example

```
public class myGLProgram extends Frame
    implements GLConstants {

    .
    .
    .

    glBegin( GL_TRIANGLES );
    .
    .
    .
```

The implementation of this interface lets you access the constants without the class prefix. However, we have also made the `GL` interface implement the `GLConstants` interface which means that your existing Magician applications won't break and, if you really like the explicit class prefix, you can keep using it. Similarly, the `GLU` interface implements the `GLUConstants` interface allowing you quick and easy access to the GLU constants.

I shall now more fully discuss each of the pipeline classes distributed with the Magician core classes that define all of the methods declared *abstractly* within the `GL` and `GLU` classes.

`CoreGL` and `CoreGLU`

The implementations of the abstracted functions in `CoreGL` and `CoreGLU` simply execute the correct OpenGL functions by dispatching them to "*native methods*" that interface directly with the underlying OpenGL implementation on your machine.

To put it simply, `CoreGL` and `CoreGLU` give you the results you would get by writing software in C with OpenGL. The following code stub sets up a single polygon and smoothly colours it from top to bottom using a `CoreGL` pipeline.

```
/** Create a new GL pipeline using the core functionality */
CoreGL gl_ = new CoreGL();

/** Draw a polygon */
gl_.glBegin( GL.GL_POLYGON );
    gl_.glVertex2f( 0, 0 );
    gl_.glVertex2f( 10, 0 );
    gl_.glVertex2f( 10, 10 );
    gl_.glVertex2f( 0, 10 );
gl_.glEnd();
```

This example illustrates that an OpenGL or GLU pipeline is instantiated as an object, or instantiation, of a particular pipeline class. Therefore, any OpenGL or GLU functions that you wish to call are done as *instance methods* on a particular object. This is slightly different to "pure" OpenGL since we require to route all function calls through an object, but the effect is the same. In actual fact, the creation of a `CoreGL` object is essentially syntactic sugar since it has no member variables to store any sort of information in. The only reason for the existence of this syntax is to enable the transferrence of pipelines between pipeline classes which I shall elaborate on *infra*.

`TraceGL` and `TraceGLU`

The `TraceGL` and `TraceGLU` pipeline classes are *extensions* of the `CoreGL` and `CoreGLU` classes in that they add to the core functionality defined within the core pipeline classes.

The basic function of these classes are to provide tracing information on OpenGL and GLU functions being called as they are called. The two classes operate in two different modes, *verbose* and *summary* which give differing levels of information to you.

Summary mode simply prints the name of each method as it is called from within a Java program. This enables the developer to see exactly where the program is reaching before some sort of problem manifests itself. The use of `TraceGL` and `TraceGLU` acts as a proactive debugger backtrace! If you use these classes, you can see where something has crashed rather than have to debug it after the fact.

The output of summary mode when run against the short code listed *supra* looks like:

```
glBegin()
glVertex2f()
glVertex2f()
glVertex2f()
glVertex2f()
glEnd()
```

Verbose mode acts in a similar manner to summary mode but it actually dumps the values of the arguments for each OpenGL and GLU function before they are called. This can be useful in two important ways. Firstly, you can use Magician as a code generator. Secondly, you can also use this feature to sanity check the values that you're passing into methods that are giving out suspect results. For example, if you make a calculation prior to setting the viewport *via* `glViewport()`, you may find that you have forgotten to cast the dimensions of the viewport to `float` datatypes prior to dividing them to calculate an aspect ratio[1]. Verbose mode of the `TraceGL` pipeline would show you what the values being passed into `glViewport` were and you would locate the bug more quickly.

The same example code run in verbose mode would generator output corresponding to

[1]Which would result in the aspect ratio of 0 in Java.

```
    glBegin( 9 )
    glVertex2f( 0, 0 )
    glVertex2f( 10, 0 )
    glVertex2f( 10, 10 )
    glVertex2f( 0, 10 )
    glEnd()
```

Arrays are written as being contained within square brackets (`[ ]`) and only the first 8 elements of the array are written.

The mode of the tracing pipelines can be switched at any time with the `setMode()` method and will take immediate effect. The valid values that can be passed to `setMode()` are `TraceGL.SUMMARY` and `TraceGL.VERBOSE` or `TraceGLU.SUMMARY` and `TraceGLU.VERBOSE`.

Similarly, you can alter the parent pipeline of any pipeline object at any time during the program's execution by using the `setParent()` and `getParent()` methods defined within each pipeline class.

`ProfileGL` and `ProfileGLU`

The `ProfileGL` and `ProfileGLU` classes add the ability for developers to profile their programs at run-time. This functionality is extremely useful for identifying bottlenecks within your software in terms of OpenGL performance and also can be used to identify redundant function calls or OpenGL state changes.

As with the `TraceGL` and `TraceGLU` classes, these classes operate in two different modes, *summary* and *verbose*. The duration from immediately before dropping into the parent pipeline class[2] until immediately after returning from the parent pipeline class is timed in *microseconds*[3].

Summary mode operates by accumulating the times that each function has taken to execute and presents the sum total upon request. For example, it will report the total amount of time every called OpenGL or GLU function has taken and also the number of times the function has been called. This

[2]The parent pipeline of a `ProfileGL` or `ProfileGLU` object should be `CoreGL` and `CoreGLU` for the most accurate results, otherwise the activities of the parent pipeline might prejudice the timing calculations.

[3]Or as close to microseconds that the CPU can provide. Magician provides a utility class called `MicroTimer` that provides portable timing across different operating-systems and takes the CPU granularity into account.

is extremely useful for spotting redundant state changes and also analyzing which OpenGL and GLU functions are the most performance intensive. With more fine-tuned profiling, performance bottlenecks and hotspots can be flattened out.

Summary mode will only display the actual methods called when requested, not all the OpenGL and GLU methods. This saves you having to wade through hundreds of functions that have never been used and allows you to focus on the data that you're interested in.

The summarised profiling output of the short OpenGL stub listed *supra* looks like

```
glBegin() was called 1 time, totalling 12us
glVertex2f() was called 4 times, totalling 112us
glEnd() was called 1 time, totalling 14us
```

Verbose mode displays the time taken for a particular OpenGL function to execute immediately after it has executed. The name of the function is also displayed in the style of the `TraceGL` and `TraceGLU` classes although the values of the arguments is not displayed.

The verbose profiling output of the OpenGL code is

```
glBegin() took 12us to execute
glVertex2f() took 32us to execute
glVertex2f() took 24us to execute
glVertex2f() took 30us to execute
glVertex2f() took 26us to execute
glEnd() took 14us to execute
```

`ErrorGL` and `ErrorGLU`

In order to maintain extremely high-performance within Java programs, Magician does not, by default, perform any error handling. Within the base interface classes, however, each OpenGL and GLU function is declared to *throw* an exception object of class `OpenGLException`. In the normal rendering pipelines, this exception will never be thrown allowing you to avoid setting up costly `try ...catch` clauses in your programs.

However, should you wish to enable error checking during the development stages of your program, or to extract detailed error logs from a deployed program, you can use the `ErrorGL` and `ErrorGLU` classes which will test for an error condition having been flagged after *each* OpenGL and GLU call. If an error has been detected, an `OpenGLException` will be thrown.

By testing for errors after each call, debugging is extremely finely-grained with errors being flagged immediately upon triggering them. However, this flexibility comes at a considerable performance penalty and should be used sparingly.

In addition to enhanced error-checking functionality, `ErrorGL` and `ErrorGLU` also test for a current rendering context prior to executing the method. This check is useful in cases where you may have forgotten to switch in a rendering context prior to executing OpenGL methods. In cases where this is true, a `GLNoCurrentContextException` will be thrown allowing you to fix your code quickly and accurately.

As with all other pipeline extension classes, these error-checking classes can be stacked on top of other extension classes. For example, you might wish to combine the error-checking pipelines with the tracing pipelines for maximum debugging capabilities.

Swapping Pipelines

Now that we have looked at each of the pipeline classes that are provided with the base installation of Magician, I shall explain exactly how these can be used to afford maximum flexibility and power to you in your programs.

The main concept of the pipeline classes is to provide functionality to dynamically change the way in which OpenGL functions. Using the powerful interface mechanism in Java, this is trivial to add into your programs and allows you to affect any quantity of code from the entire program to single lines.

In normal operation with the `CoreGL` class, you would simply allocate a new `CoreGL` object and route all OpenGL function calls through that. For example

```
.
.
.
CoreGL gl_ = new CoreGL();
```

```
.
.
.
gl_.glBegin( GL.GL_QUADS );
    gl_.glVertex2f( 0, 0 );
    gl_.glVertex2f( 10, 0 );
    gl_.glVertex2f( 10, 10 );
    gl_.glVertex2f( 0, 10 );
gl_.glEnd();
```

This is a perfectly acceptable way to do things if you simply want to use the basic rendering functionality of OpenGL. However, to change the pipeline to provide tracing information, you would need to recompile your program. That's not particularly useful to you. What you really want to be able to do is dynamically change the pipeline in mid-stride.

Accomplishing this is actually quite easy. Instead of routing all the OpenGL function calls through an object of class `CoreGL`, we can instead route the calls through an reference of class `GL`, *i.e.*, the interface. Since each pipeline extension class *implements* this interface, they can be freely cast between themselves. Therefore, if you wished to perform some basic rendering using a `CoreGL` object then switch on tracing for the final functions, you could do

```
.
.
.

/**
 * This is initialized to null since interfaces cannot
 * be instantiated.
 */
GL gl_ = null;

/** Core OpenGL pipeline object */
CoreGL coregl_ = new CoreGL();

/**
 * Tracing OpenGL pipeline object with the core
 * pipeline as a parent
 */
TraceGL tracegl_ = new TraceGL( coregl_ );

.
.
.
```

```
/** Assign the standard pipeline to the reference */
gl_ = coregl_;

/** Render a square */
gl_.glBegin( GL.GL_QUADS );
    gl_.glVertex2f( 0, 0 );
    gl_.glVertex2f( 10, 0 );

    /** Switch the rendering pipeline to trace! */
    gl_ = tracegl_;

    /** Draw the last two points... */
    gl_.glVertex2f( 10, 10 );
    gl_.glVertex2f( 0, 10 );
gl_.glEnd();
```

This would result in a square being drawn, as expected. However, since tracing was turned on half-way through the rendering process, you should expect to see the following output appear on your screen as well.

```
glVertex2f()
glVertex2f()
glEnd()
```

which reflects the OpenGL functions that were called after the pipeline was switched to trace OpenGL calls. We could also switch on verbose tracing by doing the line

```
.
.
.

/** Switch the rendering pipeline to trace verbosely! */
tracegl_.setMode( TraceGL.VERBOSE );
gl_ = tracegl_;
```

Running within the same context, this would produce output of

```
glVertex2f( 10, 10 )
glVertex2f( 0, 10 )
glEnd()
```

illustrating the code generating capabilities of the `TraceGL` class.

If you wanted to be truly disturbed, you can happily stack a `ProfileGL` object on top of a TraceGL pipeline. This would give you the ability to both trace and profile your program simultaneously.

To effect this, when creating the ProfileGL object, instead of setting the parent object to be the instance of CoreGL, set it to your TraceGL object then assign the ProfileGL object to the interface reference. For example

```
.
.
.

GL gl_ = null;

/** Create a new standard OpenGL pipeline */
CoreGL coregl_ = new CoreGL();

/** Create a new tracing pipeline as a child of the core */
TraceGL tracegl_ = new TraceGL( coregl_ );

/** Create a new profiling pipeline as a child of the tracer */
ProfileGL profilegl_ = new ProfileGL( tracegl_ );

/** Render with the profiler and tracing combined pipeline */
gl_ = profilegl_;

.
.
.
```

The order in which you assign parents for pipelines is important especially in the case of when you are using the ProfileGL or ProfileGLU classes. These classes time the execution of the same method *within the parent pipeline class* so, if you have a parent of something other than CoreGL or CoreGLU, the timings will include not only the actual OpenGL function execution time but also the time taken to execute whatever additional operations that the parent pipeline performs. This has the potential to dramatically mess up your timings and should be watched out for.

If you fail to provide suitable parents for all the pipeline classes right down to an instance of CoreGL and CoreGLU your code will still execute correctly, but execution will stop at the point that the parent link is broken.

A final note on pipeline objects is that you can safely create as many pipeline objects as you wish. Pipelines are *stateless* and contain no useful or persistent information whatsoever. Therefore, if you have multiple subclasses that need to use OpenGL, you could either pass a pipeline object down into the subclass or just create a new local one and use that. Both solutions are possible in Magician.

Pipeline Conclusions

The extended pipeline functionality provided in the core Magician classes is a powerful aid to developing bug-free and high-performance OpenGL applications in Java. Template classes, called `TemplateGL` and `TemplateGLU`, are also provided with Magician that have all the methods required by the OpenGL and GLU interfaces implemented as stubs for developers to adapt if they need pipeline functionality not provided by the bundled pipelines. `TemplateGL` and `TemplateGLU` are also fully commented in order to produce high-quality *Javadoc* API documentation.

However, pipelines on their own are not particularly useful and form only one third of the requirements of a complete OpenGL implementation. The remaining aspects are closely interlinked and I shall discuss each in turn.

Chapter 4

Components, Contexts and Listeners

AWT provides a platform-independent window toolkit that you can use to write fairly complicated GUIs. The platform independence is achieved by extracting the core principles from Windows 95, X Windows and MacOS amongst others and turning them into an abstracted toolkit that can translate all the AWT methods and classes into the appropriate native window system functions. The presence of such an abstracted toolkit is extremely useful since you can write portable code that will have identical functionality on any platform that supports AWT.

Magician leverages the notion of platform-independent rendering and seamlessly integrates with the AWT implementation on a particular platform to provide *drawing surfaces*, or windows, onto which OpenGL can render. This functionality makes it trivial to write fully functional 3D applications that will run automatically on different platforms but look and behave identically.

Using OpenGL Drawables

In Magician, all references to drawing surfaces or rendering contexts are made *via* the `GLDrawable` interface which is implemented by all Magician drawing surface classes.

Magician provides a class called `GLComponent` that provides a drawing surface, or window, for OpenGL to render onto which implements `GLDrawable`. This class is also a standard AWT component which can be treated as any other AWT component and uses standard AWT Component event handling mechanisms. This functionality gives you the ability to embed GUI components into your application that perform OpenGL rendering directly using the exact same function calls as if you were manipulating stock AWT objects such as `Labels`, `Checkboxes` and `TextFields`.

As with the standard AWT classes, the `GLComponent` class abstracts all the underlying window-handling functions that differ on a per-platform basis into a unified API that will work portably on all platforms.

In addition to this, `GLComponent` also internally handles the various *events* that AWT components can receive in a way that is meaningful in an OpenGL context, for example, resizing the component, iconifying the component and handling repainting of newly exposed components.

Magician also provides an offscreen drawing surface class called `GLOffscreenBuffer`, which implements `GLDrawable`, which can be used to perform intermediate or hidden operations before copying the finished result to an onscreen buffer for viewing.

Creating Drawing Surfaces

In Magician, creating drawing surfaces is very simple. You simply ask for a desired object from a *factory.* The factory approach allows Magician to standardise drawing surfaces such that they take the same arguments and are created in exactly the same way.

The factory in question that creates Magician drawing surfaces for you is called `GLDrawableFactory`[1] and returns an object of type `GLDrawable`. This requires you to cast the returned drawable to the correct type before you can use it. For example

```
int width = 200,
    height = 200;
```

[1]This class did not exist in Magician 1.*x* and was called `GLComponentFactory` instead which only returned `GLComponent` objects. Magician 2.*x* ships with a `GLComponentFactory` class for backwards-compatibility.

```
GLComponent component =
    (GLComponent)GLDrawableFactory.createGLComponent( width, height );
```

If the factory cannot create a new drawing surface, it will return `null`.

Creating an offscreen buffer is also possible through the factory by calling the `createGLOffscreenBuffer()` method.

```
int width = 200,
    height = 200;

GLOffscreenBuffer offscreenBuffer =
    (GLOffscreenBuffer)GLDrawableFactory.createGLOffscreenBuffer( width, height )
```

As an aside, the `GLDrawableFactory` class is used to not only provide a single place to create any sort of Magician drawing surface, but it also glosses over differences in AWT design and implementation between Java vendors. Using a factory allows Magician to internally dispatch you a `GLDrawable` object for the correct Java virtual machine[2].

The `GLComponent` in this state can be manipulated in the usual ways that standard AWT components can, most importantly, they can be added into container components in your GUI for layout. Offscreen drawables cannot be added to GUIs as they exist simply as an in-memory structure through which rendering can take place.

To be useful within the context of OpenGL, you need to *associate* the `GLDrawable` with a `GLContext`. This completes the architecture of OpenGL rendering in that the results of the rendering pipeline, as defined by a rendering context, are funneled through the context onto the drawing surface which is encapsulated as the `GLDrawable` object.

By default, every `GLDrawable` has a new `GLContext` created for it, although you can pass a pre-created context to the `GLDrawableFactory` call if you wish when creating a new drawing surface.

[2]This strategy was implemented because of differences between the AWT implementations of Sun and Microsoft. However, this design also future-proofs Magician in case other virtual machine vendors decide to de-standardise their AWT implementations in the future. Your code shouldn't require any alteration to work on these new platforms since Magician does the work.

Sharing Display Lists and Texture Objects

OpenGL also features the ability to share *display lists* and *texture objects* between rendering contexts to reduce memory overheads. This is an extremely useful feature. For example, in a 3D editor that has three orthographic views of an object, if you had stored the object within a display list, you would need to create a separate display list for *each window*. This is both expensive in terms of memory consumption and also tricky to synchronize between windows. By sharing display lists, only one component need make changes to the list. Similarly, sharing texture objects reduces memory consumption and makes textures available to all components that are participating in the sharing.

Magician abstracts the platform-specific functionality used to share display lists and texture objects into a single, easy-to-use mechanism. Firstly, you should create a standard `GLDrawable` object. Then you create your other `GLDrawables` passing the first `GLDrawable` as an argument to the constructor. For example

```
/** Create the front view */
GLComponent frontComponent =
    (GLComponent)GLDrawableFactory.createGLComponent( 200, 200 );

/**
 * Create the plan view and share the front view's
 * display lists
 */
GLComponent planComponent =
    (GLComponent)GLDrawableFactory.createGLComponent( frontComponent,
                                                      200, 200 );

/**
 * Create the side view and share the front view's
 * display lists
 */
GLComponent sideComponent =
    (GLComponent)GLDrawableFactory.createGLComponent( frontComponent,
                                                      200, 200 );
```

After these components have initialized, the display lists and texture objects are *pooled* in that if any component creates a display list or texture objects any other components participating in the share can use it. There is no restriction that the "first" `GLDrawable` create the objects for use by the other components.

It is also perfectly acceptable to share `GLComponent` objects with `GLOffscreenBuffer` objects and *vice versa*.

Fullscreen `GLComponents`

Some popular 3D hardware accelerator cards provide *fullscreen*-only rendering, that is, these cards cannot render into a window as is the standard behaviour with Magician applications. That said, Magician applications being run on a fullscreen accelerator will still continue to function correctly and rendering will occur. The major drawbacks are that the main GUI is completely hidden by the output from the accelerator card. In fact, your entire desktop or window manager will be completely obscured by the output of the renderer.

Within Magician applications, this is not a major problem but should you move your mouse out of the boundaries of the actual AWT window, AWT events may not be delivered to that window. Therefore, a simple solution has been implemented within Magician that automatically sizes up a `GLComponent` to being the size of the screen.

To use this feature, your application should have only one `GLComponent` within it. The only change required by you to your application in this case is to make the immediate parent container of the `GLComponent` be of class `GLFullScreenFrame` instead of, say, `Frame`. The simplest way to do this is as follows

```
public class myDemoApplication extends GLFullScreenFrame {

    GLComponent glc =
        (GLComponent)GLDrawableFactory.createGLComponent( 200, 200 );

    .
    .
    .

    this.setLayout( new BorderLayout() );
    this.add( "Center", glc );

    .
    .
    .
```

When using a `GLFullScreenFrame`, the width and height arguments given to the `GLDrawableFactory` are ignored and the component is resized automatically.

There is one major caveat with this approach. Your desktop will most probably not be the same size as the rendering output generated by your hardware acclerator. For example, if your desktop is sized to 1024×768 and you are using a Voodoo Graphics accelerator, the actual rendering output will be 640×480, not 1024×768. Therefore, even though AWT events are being correctly trapped the dimensions of the underlying window will not necessarily match the 3D output. In this example, any mouse motion should be divided by 8/5 to ensure the AWT window and 3D output match up.

Destroying `GLDrawables`

Finally, if you have completely finished using a particular `GLDrawable`, you can shut it down completely and deallocate all the internal resources it's using by invoking the `destroy()` method against it. This will *not* remove the Java AWT component from the GUI if the drawable is a `GLComponent`, but will allow you to safely remove the `GLComponent` without causing any strange problems.

Rendering Contexts

In the section discussing the architecture of OpenGL, I touched briefly on the concept of "*rendering contexts*". Rendering contexts act as a conduit, or link, between a "drawing surface" and an OpenGL pipeline.

Magician encapsulates the rendering context as a class called `GLContext` which provides methods for performing most "context-ish" operations such as context switching, font handling and swapping frame-buffers. This class is simply created through a standard constructor, for example

```
GLContext context = new GLContext();
```

However, it is more common to use a context that is allocated by default when a new `GLDrawable` is created. A reference to this context can be found by invoking the `getContext()` method within the `GLDrawable` interface. For example,

```
/** Create a new GLComponent */
GLComponent glc =
    (GLComponent)GLDrawableFactory.createGLComponent( width, height );
    /** Reference the GLContext associated with the drawing surface */
    GLContext context = glc.getContext();
```

This context object will not automatically or unexpectedly change during the execution of a Magician application, so, to save on unnecessary method invocations of `getContext()`, you might wish to store the value in a global variable. Of course, if you are using multiple `GLDrawable`s within your application, this is highly unrecommended.

Context Capabilities

When a rendering context is being created, it is defined to have certain *capabilities* as to how rendering is funnelled through it. For example, for smoothly animated applications, you might wish to use a technique called *double-buffering* where OpenGL actually renders the image to an *offscreen framebuffer* instead of the screen. You then perform a *buffer-swap* which copies the contents of the offscreen framebuffer to the drawing surface. There is a corollary capability known as *single-buffering* in which the rendering pipeline draws directly onto the drawing surface, but this produces tremendously flickery updates especially when repainting the drawing surface or animating the scene.

The type of buffering used within a context is one capability that rendering contexts support. Other common ones pertain to the number of bits in the depth buffer[3], the colour bias of your framebuffer, the size of stencil and accumulation buffers & many other different configuration options.

These options are configurable by the developer and are generally chosen to suit the capabilities of the user's display and video card. For example, many users may still be using video cards capable of only 256 colour display whereas other high-powered users might be using a Silicon Graphics workstation that supports a 32-bit hardware depthbuffer, 32-bit colour, hardware stencilling and overlay / underlay planes.

There is a default "safe" set of capabilities that most computers can use provided with each `GLContext` that you can use. This `GLCapabilities` object can be accessed *via* the `getCapabilities()` method. However, if you wish to exercise the full benefits of OpenGL's acclaimed image quality, you'll probably want to hand-tune these capabilities to suit better hardware.

[3]The size of depthbuffer dictates exactly how accurate your depth calculation is likely to be. For example, a small depthbuffer in a huge world will cause *depth artifacts* where the depth order of pixels is wrong causing objects that should be hidden to be visible.

To do this, you simply acquire a reference to the `GLCapabilities` object associated with a context and alter the capabilities to suit.

```
/** Create a new GLComponent */
GLComponent glc =
    GLComponentFactory.createGLComponent( width, height );

/**
 * Get a reference to the capabilities for the context
 * associated with the GLComponent
 */
GLCapabilities cap = glc.getContext().getCapabilities();

/** Set the depthbuffer size to 32 bits */
cap.setDepthBits( 32 );

/** Set the colour depth to 24 bits */
cap.setColourBits( 24 );

/** Set the pixel type to RGBA */
cap.setPixelType( GLCapabilities.RGBA );

/** Make the context double-buffered */
cap.setDoubleBuffered( GLCapabilities.DOUBLE_BUFFER );

.
.
.
```

Some video cards might not be able to cope with a given set of capabilities and in this case a `GLContextInitializationException` will be thrown when either the first call to `GLContext.makeCurrent()` or `GLComponent.initialize()` is made. These methods will be elaborated on *infra*, but if you decide to catch this exception, you can re-specify some advanced aspects of the requested capabilities to be more conservative.

After the context has been internally created, you will not be allowed to set any of the capabilities of the context as they will have been internally locked. In addition to locking the capabilities, the `GLCapabilities` object will be re-populated with the exact values used by the underlying window system code.

Another aspect of the `GLCapabilities` class is that for every "setter" method for all the capabilities, there is a corollary "getter" method that allows you to query back the values currently defined. This is quite useful in cases where you are requesting an unusual capability and you wish to try a new one relative to the original. For example, if your request for a 32-bit depthbuffer fails, you might wish to try 24. If that fails, try 16 bits. If that fails, fallback to 12 which is your last usable depthbuffer size before saying that your application simply can't run on such poor hardware!

Enumerating Available Capabilities

Specifying `GLCapabilities` values in a way that is guaranteed to maximise image quality across a variety of end-user machines is quite an art form since some users may be blessed with hardware-accelerated 24-bit colour framebuffers and some users may be cursed with software-only 8-bit framebuffers.

Trying various `GLCapabilities` settings until one successfully initializes is one way of ensuring your application starts up but requires considerably more application code in the `GLCapabilities` configuration stage.

The other option is to *enumerate* all the capabilities available on you machine and pick one. This can be effected by invoking the static method called `GLCapabilities.enumerateCapabilities()` which returns an array of `GLCapabilities` object initialized with the appropriate values from querying the underlying window-system.

The following stub code illustrates the use of `enumerateCapabilities()`

```
/** Enumerate the OpenGL capabilities */
GLCapabilities[] caps = GLCapabilities.enumerateCapabilities();
for ( int i = 0 ; i < caps.length ; i++ ) {
    System.err.println( "Capabilities[" + i + "]: " );
    System.err.println( caps[i].toString() + "\n" );
  }
```

which generates the output of

```
Capabilities[0]:
GLCapabilities@135049751
    Pixel Type: RGBA
    Frame Buffer Size: 8 bits
    Depth Buffer Size: 16 bits
    Double-Buffered? Yes
    Stereo Capable? No
    Direct Rendering? Yes
    Red/Green/Blue/Alpha: 2/3/3/0
    Accumulation Red/Green/Blue/Alpha: 16/16/16/16
    Stencil Buffer Size: 8 bits
    Auxiliary Buffers: 0

Capabilities[1]:
GLCapabilities@135049749
    Pixel Type: RGBA
    Frame Buffer Size: 8 bits
    Depth Buffer Size: 16 bits
    Double-Buffered? Yes
    Stereo Capable? No
    Direct Rendering? Yes
    Red/Green/Blue/Alpha: 2/3/3/0
    Accumulation Red/Green/Blue/Alpha: 16/16/16/16
    Stencil Buffer Size: 8 bits
    Auxiliary Buffers: 0
```

and so on for all the configurations that OpenGL can use on your machine. Magicians ships with a small demo program called `com.hermetica.magician.demos.enumCa` that dumps a table containing all your visual configurations.

Re-using Capabilities

In cases where you might wish to create several `GLDrawable`s within one application that all have the same visual capabilities, you can re-use the same `GLCapabilities` object to initialize each `GLContext` instead of manually setting the capabilities of each one using the `setCapabilities()` method. For example, a 3D editor might have 3 window for the orthographic projections of the front view, side view and plan view. You can initialize each of these windows to have the same capabilities by writing

```
GLComponent planComponent = ...;
GLComponent sideComponent = ...;
GLComponent frontComponent = ...;

/** Set up the first context's capabilities */
GLCapabilities cap =
    planComponent.getContext().getCapabilities();
cap.setDepthBits( 12 );
cap.setDoubleBuffered( GLCapabilities.DOUBLE_BUFFERED );
cap.setColourBits( 24 );
cap.setPixelType( GLCapabilities.RGBA );
```

```
/** Set up the other two context's with identical capabilities */
sideComponent.getContext().setCapabilities( cap );
frontComponent.getContext().setCapabilities( cap );
```

When creating the actual rendering contexts internally, Magician will convert the values set in the appropriate `GLCapabilities` object into values meaningful for each platform. This again abstracts all the unportable and unpleasant initialization of various window-system structures away from you.

Context Currency

Rendering contexts contains a snapshot of the state of the OpenGL state machine at a given point in time. Therefore, from the point of view of a rendering context, it is "*made current*", *i.e.*, the snapshot of "OpenGL state" is copied back into the OpenGL state machine, OpenGL functions are called and then the context is "*swapped out*" or "*switched out*" leaving the OpenGL state machine in a potentially undefined state.

When the context is switched out, the current state of the OpenGL state machine is copied back into the context which allows us to provide consistency when multiple rendering contexts are in use. The process of switching contexts in and out is known as "context switching". This is typically quite an expensive operation since it involves the copying of a reasonably large quantity of state.

This leads to an interesting problem in that only *one* rendering context may be active at any given time. This is an obvious progression of the idea that a snapshot of "state" is copied to and from the current context. If a second context was to be made current at the same time as another context was current, corruption of at least one, if not both, contexts would be inevitable.

With Magician, multiple contexts are handled transparently and safely within a multi-threaded environment by sophisticated high-speed internal locking code. However, knowing the facts about contexts can help you in writing applications that minimises on context switching which can boost performance by appreciable amounts.

The corollary issue that context switching brings to light is that without a current rendering context, the results of executing any OpenGL or GLU methods are undefined. Some implementations of OpenGL are more tolerant than others and you will see the expected output, but others may cause immediate crashes or otherwise undefinable results.

In a multi-threaded GUI environment such as Java and AWT, it becomes more problematic to keep track of where and when context switching occurs and where and when it *should* occur. Magician simplifies the issue through the `GLEventListener` interface which places a structure onto Magician applications. Each of the methods defined by `GLEventListener` and implemented in `GLDrawable` *will automatically perform context switching for you.* That is, you do not need to perform any explicit context switching in your own applications in these methods.

However, any other method not directly invoked by any of the `GLEventListener` methods in your application code that performs any OpenGL or GLU activity should perform context switching. AWT event listener methods are key places in which this activity should be performed.

For example, we might wish to jazz up our white rectangle example such that when the user presses a key the colour of the rectangle changes to a new random colour. The AWT event listener method that handles this activity simply reads

```
/** Handles keyboard events */
public void keyPressed( KeyEvent evt ) {
    /**
     * Check if we’ve pressed ’c’ or ’C’. If so, change
     * the colour!
     */
    if ( evt.getKeyChar() == ’c’ || evt.getKeyChar() == ’C’ ) {
        /** Set the new colour */
        gl_.glColor3d( Math.random(),
                       Math.random(),
                       Math.random() );
        return;
      }
  }
```

This example would not work well depending on which OpenGL implementation you were using. You forgot to make a context current! To correct this method, it should be written as

```
/** Handles keyboard events */
public void keyPressed( KeyEvent evt ) {
    /**
     * Check if we’ve pressed ’c’ or ’C’. If so, change
     * the colour!
     */
```

```
        if ( evt.getKeyChar() == 'c' || evt.getKeyChar() == 'C' ) {
            /** Make a rendering context current */
            ((GLComponent)evt.getComponent()).getContext().makeCurrent();

            /** Set the new colour */
            gl_.glColor3d( Math.random(),
                           Math.random(),
                           Math.random() );

            /** Release the rendering context */
            ((GLComponent)evt.getComponent()).getContext().unlock();

            return;
          }
    }
```

This might look really weird at first glance, but it's actually pretty straightforward. The bizarre line involving `makeCurrent()` is using the object-orientation features within Java's AWT event listeners to ensure that the correct rendering context is made current. You could use a global `GLComponent` variable, for example, `glc` as

```
glc.getContext().makeCurrent();
```

but this would not necessarily be accurate in applications where multiple `GLDrawable`s exist. If the colour was set in the wrong context, very strange effects might arise!

The `makeCurrent()` method simply attempts to switch in the given rendering context. If another context is in operation, the `makeCurrent()` call will sit and wait indefinitely until the other context switches out. At this point, the waiting context will switch in and processing will continue. This behaviour explains the `unlock()` call that the program makes after the colour has been set in that the context is switched out allowing other contexts to switch in and process.

Context Switching, Locks and Deadlocks

Since Magician is operating within the multi-threaded and asynchronous environment of Java and AWT, extra-special care must be taken to ensure that rendering contexts are not switched in when others are still being used. Single-threaded are generally not subject to this problem illustrated in Figure 4.1.

Magician enforces the idea that two contexts cannot be simultaneously

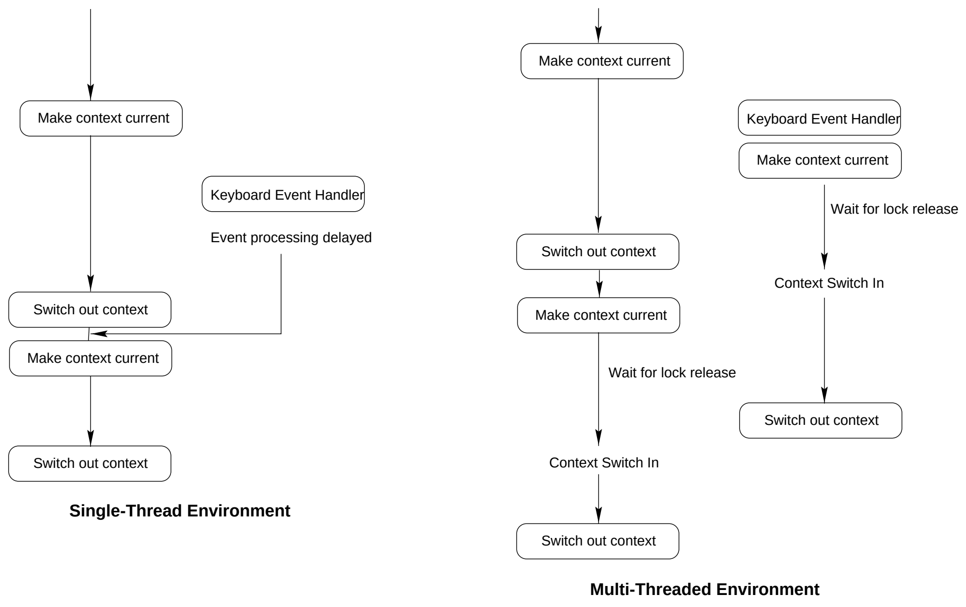

Figure 4.1: Multi-threaded and Single-threaded Context Switching

current by using an internal *mutual exclusion lock* or *mutex*. This mutex is *acquired* by a context when `makeCurrent()` or `lock()` is called and is released when `swapBuffers()` or `unlock()` is called. The mutex is shared between *all* `GLContexts` within a given application and is called the *global context lock*.

The aforementioned functions are split into two groups which are `makeCurrent()` and `swapBuffers()` and `lock()` and `unlock()`. Both groups of functions acquire and release the mutex but `lock()` only acquires the mutex and does not perform a context switch in. `unlock()` is a bit more complicated in that it will both release the mutex *and* switch out the current rendering context.

In the example above, the code used `makeCurrent()` to acquire the mutex and make a rendering context current since an OpenGL operation needed to be executed. However, we used `unlock()` to release the mutex. Why not use `swapBuffers()` instead? Well, `swapBuffers()` does exactly what it says it does. It will flip the frame-buffers of the rendering context over and refresh the `GLDrawable` as well as switching out the context and releasing the mutex. This is not perhaps what you want to happen in a lot of cases so the `unlock()` method will do the same things as `swapBuffers()` but invisibly.

The `lock()` method is not really just there for symmetry but can be used to "choke" context switching for whatever reason. A potential use for this is to prevent AWT event listeners from updating things when "dangerous" operations are happening.

For example, you might have a stock market visualization application that does real-time 3D modelling of stock market fluctuations. A Java thread continuously renders the datasets. However, the main program thread also continuously reads the stock price changes over a network and re-calculates the datasets. You obviously don't want the rendering to occur when the datasets are being recalculated or the image output will be completely wrong. To stop the rendering occurring during dataset calculation, you could simply call `lock()` and acquire the context lock which would prevent the rendering thread from running until you had finished calculating the datasets. At that point, you can call `unlock()` to start rendering again.

Therefore, the mechanism that Magician uses to ensure context switching is a safe operation is extremely powerful and flexible. However, there is a fairly deadly caveat known as *deadlock*.

Deadlocking occurs when a mutex has been acquired but not released and the process or thread currently acquiring the mutex is waiting on another resource and therefore cannot release the original mutex.

The simplest way in which deadlock can occur is to *overlock* in one thread. That is, after calling `lock()` or `makeCurrent()`, you call it again from the same thread. The following code will cause an overlock.

```
/** Handles key presses */
public void keyPressed( KeyEvent evt ) {
    /** Switch in a context */
    ((GLComponent)evt.getComponent()).getContext().makeCurrent();

    if ( evt.getKeyChar() == 'c' || evt.getKeyChar() == 'C' ) {
        /** Switch in the context */
        ((GLComponent)evt.getComponent()).getContext().makeCurrent();
        .
        .
        .
```

The second invocation to `makeCurrent()` would indefinitely wait on the global context lock being released. Given that it had been acquired previously in the same method, this is unlikely to occur! At this point, your application would hang. The only solution in this case is for another thread to release the lock.

Overlocking is quite common when first programming with Magician. As I mentioned in a previous section, the methods defined in `GLEventListener` and implemented by `GLDrawable` perform internal context switching. However, you might inadvertently add your own context switching which would cause an overlock to occur. You should be extremely careful if you wish to perform manual context switching within any of the `GLEventListener` methods!

A true deadlock is a far nastier problem in which two threads have locked each out. Two resources are usually required here and therefore is quite unusual to occur within a Magician application. The basic premise here is that thread A acquires lock A and requests lock B whereas simultaneously thread B acquires lock B and requests lock A. Since each thread is waiting on the other's lock, neither can ever free their locks.

Lock-based systems nowadays sometimes have *deadlock resolution* algorithms which force one thread to "back off" and release their lock allowing the other thread to complete. Since deadlocking is an extremely unusual, and user-induced, occurrence within Magician applications, Magician does not implement any deadlock resolution procedures.

The class used to implement the mutexes is supplied with Magician and is a utility class called `CriticalSection`. This can be enabled to verbosely trace locking operations and is detailed in Chapter 8.

"I'm listening."

Java 1.1 features a radically different AWT event-handling interface to Java 1.0 in that the concepts of *listeners* and *adapters* are now being used in place of the old explicit event handlers.

Listeners are defined as interfaces for which you must implement suitable method bodies. For example, if you wished to handle keyboard input, you would declare a class as implementing KeyListener. This would require you to implement three methods

```
public void keyPressed( KeyEvent event )
public void keyReleased( KeyEvent event )
public void keyTyped( KeyEvent event )
```

Additionally, within the body of your code you would *register* the listener with the AWT component which you wish to handle the events of using the event handles defined within this listener.

Adding and Removing Listeners

Magician provides an interface called GLEventListener[4] that defines methods to handle various standard procedures that GLDrawables will need to handle including window resizing, window repainting and initialization of the GLDrawable. The methods currently defined within GLEventListener are

```
/**
 * Is called upon registration of the listener via
 * GLComponent.initialize()
 */
public void initialize( GLDrawable component )

/**
 * Is called when the window has become exposed and
 * requires redrawing
 */
public void display( GLDrawable component )

/** Is called when the window is resized */
public void reshape( GLDrawable component, int x, int y,
                     int width, int height )

/** Returns an OpenGL pipeline object to the listener */
public GL getGL()
```

These methods provide a framework in which the most standard building blocks of an OpenGL program can be placed. For example, the initialize() method might be implemented to contain lighting setup and global OpenGL

[4]This interface was called GLComponentListener in *Magician 1.0.0*, but has been renamed for releases *1.1.0* upwards to be applicable to components other than AWT-based GLComponents.

state setup since this may only be required once within the lifetime of the application since `initialize()` is invoked *once only*. The `display()` method is called by either AWT when the `GLDrawable` becomes partially or completely visible and the screen requires redrawing or it can be called manually by you *via* the standard AWT `repaint()` method to force a screen redraw. This is most likely in cases where you are driving animated scenes within your applications. The `reshape()` method is invoked when the `GLDrawable` requires resizing. This usually occurs when the container containing the `GLDrawable` is resized and the AWT `LayoutManager` associated with the container is set to make child components automatically fill the available space, for example, as exhibited by the `BorderLayout` manager. Finally, the `getGL()` method is invoked when a `GLDrawable` object needs to use OpenGL internally. This is usually just to flush OpenGL drawing commands *via* `glFlush()`. By forcing you to return an OpenGL pipeline object, you will be able to maintain any tracing or profiling information or use your own custom pipelines internally instead of having the drawing surface behave as a black box.

Simple applications such as the "white rectangle" demo usually don't implement a large number of listeners which makes it perfectly acceptable to implement the code for the `GLEventListener` methods within the main Java program.

To add the class containing these methods to the `GLDrawable` as a listener, the class must be declared as *implementing* the `GLEventListener` interface. For example, the `whiteRectangle` demo class is declared as

```
public class whiteRectangle extends Frame
    implements GLEventListener, ... {
```

Therefore, to register this class as being a listener, you only need invoke the `addGLEventListener()` method against a `GLDrawable` object. In the case of the `whiteRectangle` class, this is achieved by doing

```
glc.addGLEventListener( this );
```

Once the `initialize()` method has been invoked against that `GLDrawable` object, the methods registered in each listener will be invoked as needed.

It is also possible to remove registered `GLEventListener`s from a `GLDrawable` by invoking the `removeGLEventListener()` method. This removes the listener given as the argument from the list of listeners registered against the given drawing surface.

In order to be able to support advanced multi-pass rendering techniques, Magician also supports the notion that a `GLDrawable` can have *multiple listeners* associated with it. Therefore, one listener might handle the rendering of polygons whereas a second listener might handle computation of lighting, view volume culling or feedback or selection buffer operations.

Within the context of offscreen drawing surfaces represented by the `GLOffscreenBuffer` class, the `reshape()` and `display()` methods have no real meaning in GUI terms. However, both methods are called when the `GLOffscreenBuffer` is initialized after `initialize()` as per usual. Furthermore, if you are animating a `GLOffscreenBuffer`, the `display()` method will continue to be called. Therefore, it is adviseable to implement `GLEventListener`s for offscreen drawing surfaces in exactly the same way as you would for onscreen drawing surfaces.

Internal Context Switching and Automatic Repainting

All the methods provided by the `GLEventListener` interface are implemented within the `GLDrawable` interface as performing the appropriate context switching for you. Thus, when you implement your listener methods in your program, you need not worry about any context switching operations at all. These are handled internally for you. Of course, this does not restrict you from manually managing context switches if you so desired.

The functionality inherent within the `display()` handling code operates in a way that automatically updates the `GLDrawable` by buffer flushing or swapping. This may not be desired if you are drawing, say, a rubber-banded box in a 3D editing tool. You might wish to render the rubber-band over the scene in an overlay or underlay plane without causing automatic repaints to occur.

This automatic behaviour can be disabled by using the `setFlushOnRepaint()` method defined within `GLDrawable`. This method takes a `boolean` value as an argument. A value of `true` specifies the default behaviour of automatic buffer updating whereas a value of `false` will disable buffer flushing completely for that `GLDrawable`. You must now manually invoke `swapBuffers()` to update the component.

A final note regarding listeners is that even though they are a Java 1.1

feature, Magician has implemented listeners in such a way that they can be used within Java 1.0 compliant browsers such as Microsoft Internet Explorer 3. This helps ensure that applications that you write using Magician will work portably on a large number of machines.

Driving Animations with `GLDrawable`

The basic functionality of the `GLDrawable` and `GLEventListener` classes is quite likely to satisfy most application requirements. However, if your application is in some way dynamic in that it animates the scene in some way, then can `GLDrawable` cope?

By default, `GLDrawable` objects *react* to events generated against them, for example, reshape and redraw events. However, in the case of animation scene refreshes need to take place on a regular basis.

There are two ways in which you can enable animation in your applications, the simple way and the complicated, but more powerful, way. The differences between these are that the simple method uses the `GLDrawable` to drive animation automatically and the second method requires that you set up your own thread to drive component repainting on a scheduled basis. I shall explore the subject of complex animations in Chapter 4.

Repetitive, Continuous Animation

The simplest and most portable way to drive animated scenes is to let `GLDrawable` drive it for you. This method requires little additional coding to your applications and is guaranteed to perform optimally and safely.

To switch a `GLDrawable` into a mode that can drive animations, you simply need to invoke the method `start()` against a `GLDrawable` after you invoke `initialize()`. This initializes a thread within the component which repeatedly causes scene redraws to occur at regular intervals. What actually happens in these cases is that the `display()` method of any registered `GLEventListener`s of that component is called regularly.

Therefore, to implement animation, you can simply put any application logic that moves objects in the scene into the `display()` method, preferably after the rendering has taken place. For example, if you wanted the white rectangle demo to spin the square, you can implement the `display()` method as

```
    /** Renders the scene */
    public void display( GLDrawable component ) {
        /** Clear the window */
        gl_.glClear( GL.GL_COLOR_BUFFER_BIT );

        /** Set the drawing colour to white */
        gl_.glColor3f( 1.0f, 1.0f, 1.0f );

        /** Draw the rectangle */
        gl_.glPushMatrix();
            gl_.glRotatef( angle, 0.0f, 0.0f, 1.0f );
            gl_.glBegin( GL.GL_POLYGON );
                gl_.glVertex3f( 0.25f, 0.25f, 0.0f );
                gl_.glVertex3f( 0.75f, 0.25f, 0.0f );
                gl_.glVertex3f( 0.75f, 0.75f, 0.0f );
                gl_.glVertex3f( 0.25f, 0.75f, 0.0f );
            gl_.glEnd();
        gl_.glPopMatrix();

        /** Update the spin angle */
        angle += 5;
        if ( angle >= 360.0f ) {
            angle -= 360.0f;
          }
      }
```

This example demonstrates that animation-based applications can be written quickly and easily with Magician.

Starting and Stopping Animation

The ability to drive animation from within `GLDrawable` is only one facet of the overall issue. It is also necessary to able to shut off the animation. For example, if the application has been minimized you may not want the animation to continue running sucking up processor time when no one is watching!

In addition to the `start()` method, the `GLDrawable` class defines three other methods that can be used to control the thread internal to each component. The `stop()` method completely shuts down the rendering thread and reverts the `GLDrawable` back to being purely *reactive.* A good example of this is in the "Molecule Viewer" demonstration program bundled with Magician in that when you spin a molecule about it will automatically continue spinning at that angle. This is easily achieved by invoking `start()` on the `GLDrawable` when the mouse is released thus starting the spinning.

Once the mouse is clicked once more, the `stop()` method is invoked ceasing the spinning of the molecule. Simple and powerful.

However, in cases such as when a window containing a `GLDrawable` is minimized, use of `stop()` and `start()` to control the rendering thread execution is quite hefty as these require safe shutdown and creation and initialization of `Thread` objects. A far quicker and cheaper alternative is to use the `suspend()` and `resume()` methods that simply "pause" the animation.

For example, minimization of the white rectangle demonstration program could be implemented quite easily by adding a `WindowListener` to the `Frame` which contains the `GLDrawable` and implementing the `windowIconified()` method as

```
/** This method is invoked when the window is minimized */
public void windowIconified( WindowEvent evt ) {
    /** Assuming we have a GLComponent called ``glc'' */
    glc.suspend();
  }
```

Similarly, the `windowDeIconified()` method which is invoked when the window is restored can be implemented as

```
/** This method is invoked when the window is restored */
public void windowDeIconified( WindowEvent evt ) {
    /** Assuming we have a GLComponent called ``glc'' */
    glc.resume();
  }
```

These methods are extremely useful when Magician-enhanced applets are being used within web pages. For example, say you have an applet containing the spinning white rectangle on a web page. If you move from that page, if you don't do something about the animation, it will continue to run in the background. Imagine that you have visited several pages that each have animations running. Before long your machine will be running like treacle under the weight of numerous applets animating with no one to watch them!

The `Applet` class defines several methods that are invoked at various stages within the applet's life-cycle and these can be used to control the animation of your applets. The methods are

`init()` Invoked once when the applet is created.

`destroy()` Invoked once when the applet is destroyed. This is usually when the browser is exited.

`start()` This is called whenever the applet is started after initialization. Also, if you have moved from the page and returned to it, `start()` is called again.

`stop()` This is called when you "leave" the web page with the applet on it.

With these method descriptions available to you, it's actually quite easy to see how `GLDrawable` animation controls can be added in. Each method can be implemented as follows

```
public void init() {
    /** No action required for init() */
  }

public void destroy() {
    /** We want to completely zap the GLDrawable here */
    glc.destroy();
  }

public void start() {
    /** Resume the GLDrawable's thread, if it exists */
    glc.resume();
  }

public void stop() {
    /** Pause the GLDrawable's thread if it exists */
    glc.suspend();
  }
```

A final note on the `stop()`, `suspend()` and `resume()` methods is that if `start()` has not been invoked against that `GLDrawable`, these methods will have no effect whatsoever! It's perfectly safe (although completely pointless!) to invoke them against a non-animating `GLDrawable`.

There are some utility methods that you can use to detect the current status of a `GLDrawable`, *i.e.*, whether it has had `start()` called or whether it is suspended or not. These methods are

`isInitialized()` Returns `true` or `false` signifying whether or not `initialize()` has been invoked against this drawable.

`isSuspended()` Returns `true` or `false` signifying whether or not the drawable is suspended, *i.e.*, has `suspend()` been called.

`isRunning()` Returns `true` or `false` signifying whether or not `start()` has been invoked against this drawable.

The animation capabilities provided within `GLDrawable` are therefore extremely simple and repetitive and are invoked on a reasonably regular, but not time-accurate, basis. If you require accurately timed updates for key-frame animations or video stream editing, this form of driving animation is probably too simplistic for your needs.

However, if you simply require things to move about using these capabilities is an extremely easy way to achieve it.

Tuning Automatic Repainting

By default, using the built-in functionality of `GLDrawable` to drive animations uses some internal code that works out how to nicely schedule threads smoothly across each JVM implementation to reduce “choppiness”.

In general, each frame of animation delays by *15ms* to avoid CPU hogging. If you don’t want your animations to be particularly equitable and are looking for the full speed-kick available to you, you can reduce this value all the way down to *0ms*, *i.e.*, no delay at all. To effect this, you can use the `setSleepDuration()` and `getSleepDuration()` methods defined within `GLDrawable` which set and return the current delay respectively.

“Manual” Thread Animations

The most common way in which you are likely to implement animations within Magician, without using the internal functionality, is *via* multi-threading. That is, a new thread of execution will be created to drive your animation.

This technique also incurs difficulties regarding thread management across different Java VMs. For example, Java 1.2 does not support various thread management methods that Java 1.1 supported. Similarly, Netscape has never supported certain thread methods such as `suspend()` and `resume()` which implies that you must be very careful about the way in which you shutdown your animation threads or pause animation.

However, you may have complex time-based systems that require more accurate, fine-grained control over the animation updates than the internal `GLDrawable`-driven functionality can provide.

Multi-threading

The main way in which you may handle animation with Java's multi-threading is to update the animation values from within the `run()` method supplied by the `Runnable` interface, then call `repaint()` against the component or components in your application. For example

```
/** The thread to drive the animation */
private Thread aThread =
    new Thread( this, "animation thread" );

/** The OpenGL Component */
GLComponent component = ...;

aThread.setPriority( Thread.MIN_PRIORITY );
aThread.start();

/** Implements the method required by the Runnable interface */
public void run() {
    while ( aThread != null ) {
        /** Update spin angle */
        angle++;
        if ( angle >= 360.0 ) {
            angle -= 360.0;
          }

        /** Repaint the component */
        component.repaint();
      }
  }
```

This approach is fairly simplistic and will partially emulate the capability available within `GLDrawable`. To extend this to handle more accurate timing, you can simply sleep the thread for a given period of time after the `repaint()` method has been invoked. For example,

```
/** Sleep for 1 second */
try {
    Thread.currentThread().sleep( 1000 );
  } catch ( InterruptedException e ) {
    e.printStackTrace();
  }
```

This will give you a fairly accurate once-per-second update and drive the animation accordingly.

There are several downsides to this approach that are worth mentioning. Firstly, the accuracy of thread timing may differ slightly between Java Virtual Machines leading to slightly different results. Secondly, you will have to perform your own thread management and finally, you cannot necessarily guarantee that a thread will be restarted by the thread scheduler at any given time.

This problem is quite a tricky one to circumvent. To give your animation thread a better chance to switch back in again, you might wish to assign a higher thread priority to it. This will certainly help it be selected by the thread scheduler, but it may also swamp the system and block out other important threads such as those that drive AWT event handling. Therefore, your application may start processing your animations 100% of the time but the GUI will be unresponsive and, worse still, the results of the rendering are unlikely to ever appear!

A saner solution is to calculate the difference between the last update and the current update in terms of time and sleep for that variable amount. This will smooth out the frame rate of the application somewhat and still be gentle on other threads within the system.

When To Context Switch

The second way in which you might wish to use Java's multi-threading to drive animation is to execute OpenGL commands from within the animation loop instead of simply updating variables and letting the `display()` method take care of things.

This technique is slightly more tricky as it involves additional context switching work by you. However, if you have read the appropriate sections on context switching in this guide, you'll have no problems!

The previous example simply invoked the `repaint()` method to render the scene. This is perfectly safe as `repaint()` invokes the `display()` methods of each `GLEventListener` attached to the component and this manages its own context switching. However, say you wish to change the state of OpenGL in your animation? What do you need to do in addition?

The only additions you need make are to make a context current prior to executing any OpenGL commands and switching the context out *after* all the OpenGL commands have executed but *before* you call `repaint()`. For example

```
/** Implemented by the Runnable interface */
public void run() {
    while ( aThread != null ) {
        /** Make the context current */
        component.getContext().makeCurrent();

        /** Do some OpenGL stuff */
        gl_.glColor3f( 1.0, 0.0, 0.0 );
        gl_.glRotatef( angle, 0.0, 1.0, 0.0 );

        /** Release the context lock */
        component.getContext().unlock();

        /** Refresh the scene */
        component.repaint();

        /** Update the angle of rotation */
        angle++;
        if ( angle >= 360.0 );
            angle -= 360.0;
          }
      }
  }
```

Explicit Redraw Control

After `repaint()` has returned, you can make the OpenGL context current again to perform other operations if you so desired given you the ability to perform multiple updates within one animation loop. You may wish to use this functionality with multi-pass rendering techniques or stereo rendering. For example, we might wish to render once into the left buffer and once into the right buffer for stereo rendering. However, when we call `repaint()`, we don't actually want a buffer swap to occur until after *both* buffers have been drawn into. In this scenario, we would invoke `setFlushOnRepaint()` with a parameter of `false` against the `GLDrawable` to disable automatic buffer swapping and manually call `swapBuffers()`. The following code illustrates the process

```
/** Implemented by the Runnable interface */
public void run() {
    while ( aThread != null ) {
        /** Switch off automatic buffer swapping */
        component.setFlushOnRepaint( false );

        /** Make the context current */
        component.getContext().makeCurrent();

        /** Select the left buffer */
        gl_.glDrawBuffer( GL.GL_LEFT );

        /** Switch out the context */
        component.getContext().unlock();

        /** Draw the left scene */
        component.repaint();

        /** Switch in the context again */
        component.getContext().makeCurrent();

        /** Select the right buffer */
        gl_.glDrawBuffer( GL.GL_RIGHT );

        /** Switch out the context */
        component.getContext().unlock();

        /** Redraw the scene into the right buffer */
        component.repaint();

        /** Flush the complete scene */
        component.getContext().makeCurrent();
        component.getContext().swapBuffers();
      }
  }
```

Thus, Magician can be used to exert powerful control over the updating of scenes through animation and Java's multi-threading model.

Associated Magician Demo Programs

enumCapabilities.java This demonstration lists all the visual configurations on your machine.

testGL.java This demonstration program simply creates a new window and draws a shaded black to blue polygon within it.

mtpaperplane.java This demonstration shows simple animation using the in-built animation features of `GLDrawable` and the use of profiling pipelines.

multiView.java This demonstration creates three components and drives animations in all three simultaneously.

molview/molview.java This demonstration illustrates the use of `GLDrawable`-driven animation to start and stop automatic spinning of molecules.

hud/hud.java This demonstration illustrates the use of multiple `GLEventListener`s attached to a single `GLComponent` allowing you to perform layered multi-pass rendering extremely simply.

Chapter 5

Geometry Producers

The GLU defines three groups of functions that can be used to generate complex geometry in a simple way. The three types of geometry in question are *Quadric Objects* which are shapes that can be expressed with a quadric equation, *NURBS* surfaces and *tesselated* polygons.

These three function groups are quite discrete and are represented in Magician in two different ways. Each group has the methods related to it implemented within one of three classes `GLUQuadric`, `GLUNurbs` and `GLUTesselator`. Similarly, these methods are also implemented within the GLU pipeline classes to maintain compatibility with the GLU specification. However, both access paths end up using the method defined within the special classes.

GLU Quadric Rendering

Quadric objects are geometrical shapes that can be described with a quadratic equation. This is outside the scope of this guide, but it allows us to create various shapes easily such as cones, spheres and disks.

Magician provides a special class called `GLUQuadric` that encapsulates an object described by a quadratic equation. The `GLUQuadric` class is used internally by the `GLU` pipelines. GLU defines various methods for manipulating quadric objects and it is these methods that are replicated into the `GLUQuadric` class. Therefore, when the GLU method `gluNewQuadric()` is invoked, it actually internally routes the call to the constructor to the

`GLUQuadric` class. This allows you to therefore take full advantage of any overloaded GLU pipelines when using quadric objects.

The following example program demonstrates the use of `GLUQuadric` objects with Magician and renders the output shown in Figure 5.1.

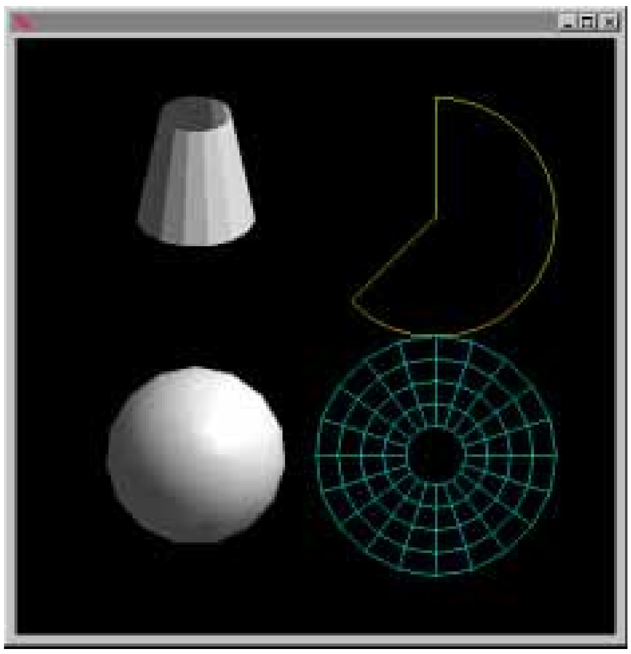

Figure 5.1: GLU Quadric Objects

```
.
.
.

/** Initialization stuff */
public void initialize( GLDrawable component ) {

    /** Build some quadrics! */
    GLUQuadric qobj = glu_.gluNewQuadric();

    /** Define the sphere */
    glu_.gluQuadricDrawStyle( qobj, GLU.GLU_FILL );
```

```
        glu_.gluQuadricNormals( qobj, GLU.GLU_SMOOTH );
        glu_.gluSphere( qobj, 0.75, 15, 10 );

        /** Define the cylinder */
        glu_.gluQuadricDrawStyle( qobj, GLU.GLU_FILL );
        glu_.gluQuadricNormals( qobj, GLU.GLU_FLAT );
        glu_.gluCylinder( qobj, 0.5, 0.3, 1.0, 15, 5 );

        /** Define the disk */
        glu_.gluQuadricDrawStyle( qobj, GLU.GLU_LINE );
        glu_.gluQuadricNormals( qobj, GLU.GLU_NONE );
        glu_.gluDisk( qobj, 0.25, 1.0, 20, 4 );

        /** Define the partial disk */
        glu_.gluQuadricDrawStyle( qobj, GLU.GLU_SILHOUETTE );
        glu_.gluQuadricNormals( qobj, GLU.GLU_NONE );
        glu_.gluPartialDisk( qobj, 0.0, 1.0, 20,
                             4, 0.0, 225.0 );
    }
```

Using GLU NURBS

NURBS (Non-Uniform Rational B-Spline) are either curves or surfaces that can be described mathematically using evaluators. These describe polynomial or rational splines or surfaces of any degree and cover Bézier splines and surfaces and Hermite splines. The mathematics of this is beyond the scope of this book.

To the programmer, NURBS can be used to represent surfaces and curves that may be subdivided mathematically to produce "finer" surfaces with a higher degree of tesselation.

OpenGL, through the GLU interface, provides you with a set of functions that allow you to describe and generate 1- and 2-dimensional NURBS, that is, curves and surfaces.

As with quadrics, Magician encapsulates a NURBS object within the `GLUNurbs` class. This class is used within the Magician GLU interface as `GLUQuadric` is for quadrics. The following source code shows how `GLUNurbs` can be used for generating and rendering a 2-dimensional NURBS surface, the output of which can be seen in Figure 5.2.

.

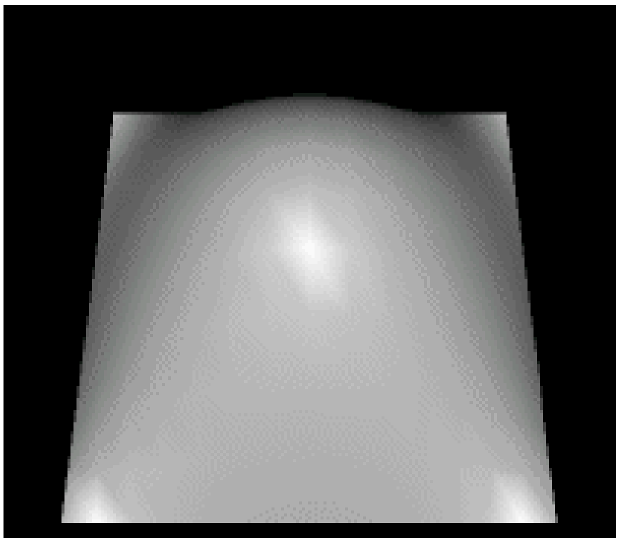

Figure 5.2: A NURBS Surface

```
.
.

/** Control points for the bezier surface */
float[][][] ctlpoints = new float[4][4][3];

/** NURBS renderer object */
GLUNurbs theNurb = glu_.gluNewNurbsRenderer();

/** Initialization stuff */
public void initialize( GLDrawable component ) {

    /** Create the control points for the surface */
    for ( int u = 0 ; u < 4 ; u++ ) {
        for ( int v = 0 ; v < 4 ; v++ ) {
            ctlpoints[u][v][0] = (float)( 2.0 * ( u - 1.5 ) );
            ctlpoints[u][v][1] = (float)( 2.0 * ( v - 1.5 ) );

            if ( ( u == 1 || u == 2 ) &&
                 ( v == 1 || v == 2 ) ) {
                ctlpoints[u][v][2] = 3.0f;
```

```
              } else {
                ctlpoints[u][v][2] = -3.0f;
              }
          }
      }

    /** Setup the NURBS state */
    theNurb.nurbsProperty( GLU.GLU_SAMPLING_TOLERANCE,
                           25.0f );
    theNurb.nurbsProperty( GLU.GLU_DISPLAY_MODE,
                           GLU.GLU_FILL );
  }

/** Draws the scene */
public void display( GLDrawable component ) {
    float knots[] =
        { 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f };

    /** Do the OpenGL stuff */
    gl_.glClear( GL.GL_COLOR_BUFFER_BIT |
                 GL.GL_DEPTH_BUFFER_BIT );

    gl_.glPushMatrix();
        gl_.glRotatef( 330.0f, 1.0f, 0.0f, 0.0f );
        gl_.glScalef( 0.5f, 0.5f, 0.5f );

        theNurb.beginSurface();
        theNurb.nurbsSurface( 8, knots, 8, knots,
                              4 * 3, 3, ctlpoints,
                              4, 4, GL.GL_MAP2_VERTEX_3 );
        theNurb.endSurface();

        /** Render the vertices, if desired */
        if ( showPoints ) {
            gl_.glPointSize( 5.0f );
            gl_.glDisable( GL.GL_LIGHTING );
            gl_.glColor3f( 1.0f, 1.0f, 1.0f );
            gl_.glBegin( GL.GL_POINTS );
                for ( int i = 0 ; i < 4 ; i++ ) {
                    for ( int j = 0 ; j < 4 ; j++ ) {
                        gl_.glVertex3f( ctlpoints[i][j][0],
                                        ctlpoints[i][j][1],
                                        ctlpoints[i][j][2] );
                      }
                }
              gl_.glEnd();
              gl_.glEnable( GL.GL_LIGHTING );
            }

      gl_.glPopMatrix();
    }
```

Using GLU Tesselators

OpenGL is restricted in the types of polygon it can render to triangles and *convex* polygons. That is, if you draw a line joining any two points in the polygon, that line must not intersect with any other lines. If you look at the top-right polygon in Figure 5.3, you can see that a line drawn between two points has a good chance of intersecting with the polygon shape.

Under OpenGL, the resulting shape drawn by a *concave* or *non-convex*

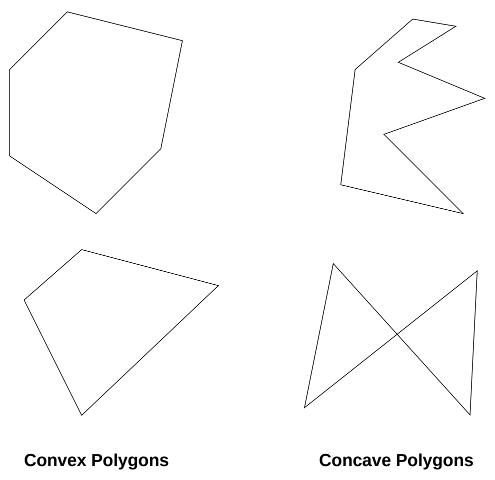

Figure 5.3: Polygons

polygon is undefined. It may draw the shape correctly, or it might draw part of the shape or nothing at all. To render these polygons correctly, you need to split them down into either triangles or convex polygons as shown in Figure 5.4. This procedure is known as *tesselation*, or if the non-convex polygons are split completely into triangles, *triangulation*.

Fortunately, OpenGL provides a tesslelator within GLU that can be used

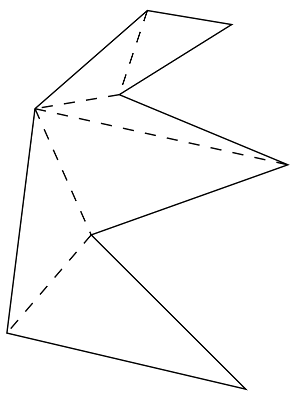

Figure 5.4: Split Polygons

to perform tesselation for you. This operates in C by allocating a tesselator object, specifying polygons and feeding vertices into the tesselator. You also register *callback* functions for different aspects of tesselation which are called by the tesselator as it's working.

The basic tesselator callbacks are called when a new polygon is to be written, when a vertex is being written and when a polygon definition ends. These functions usually correspond to `glBegin()`, `glVertex3f()` and `glEnd()` being called. For example, code to initialize a simple tesselator in C might look something like

```
/** Create a new tesselator and setup the callbacks */
GLUtriangulatorObj *tobj;
tobj = gluNewTess();
gluTessCallback( tobj, GLU_BEGIN, glBegin );
gluTessCallback( tobj, GLU_VERTEX, glVertex3Dv );
gluTessCallback( tobj, GLU_END, glEnd );

/** Start a new polygon definition to tesselate */
gluBeginPolygon( tobj );
```

This system is extremely powerful as any callback function can be used which can arbitrarily warp the tesselator vertex data if desired before sending to OpenGL.

However, callbacks can be extremely fiddly to use and there is no similar corresponding functionality within Java that can be used.

Magician takes a simpler approach to the problem and has defined a special tesselator class called `GLUTesselator`. When created, this class automatically registers default callbacks for each of the standard tesselator functions. These special callbacks, instead of routing data directly to OpenGL, pass the data back into a Java class which can use the standard Magician OpenGL pipelines to render the tesselated data. There are methods corresponding to each standard callback being `begin()`, `end()`, `vertex()`, `error()` and `edgeFlag()`.

For example, the above C code can be re-written as

```
GLUTesselator tobj = new GLUTesselator();
glu_.gluBeginPolygon( tobj );
```

However, using the standard `GLUTesselator` class will result in messages such as

```
GLUTesselator: begin(): override this method!
```

being displayed as tesselation occurs. The `GLUTesselator` class itself should not be used for tesselation but should be subclassed and the 5 standard methods overridden with your own functionality. For example, a Magician tesselator object corresponding to the C fragment above can be written as

```
public class SomeTesselator extends GLUTesselator {

    /** The functionality for the GLU_BEGIN callback */
    public void begin( int mode ) {
        gl.glBegin( mode );
      }

    /** The functionality for the GLU_VERTEX callback */
    public void vertex( float[] data ) {
        gl.glVertex3fv( data );
      }

    /** The functionality for the GLU_END callback */
    public void end() {
        gl.glEnd();
      }
  }
```

and the C code fragment re-implemented as

```
/** Create a new tesselator */
SomeTesselator tobj = new SomeTesselator();

/** Start polygon tesselation */
glu_.gluBeginPolygon( tobj );
```

The GLU tesselators operate in subtly different modes depending on what data is pushed into them and which callbacks are registered. If possible, the tesselator will tesselate the incoming vertex data into triangle strips instead of triangles which are then passed to the appropriate callbacks for processing. However, this behaviour only occurs when the "edge flag" callback isn't registered at all.

By using the default `GLUTesselator` constructor, all the GLU 1.1 callbacks are registered being, `begin`, `end`, `vertex`, `edgeFlag` and `error`. However, you can specify which callbacks you wish to register in a new `GLUTesselator` constructor which takes an integer bitmask as an argument. For example, the following code snippet only registers the `begin`, `end` and `vertex` callbacks, *i.e.*, the bare minimum to actually tesselate something!

```
/** Create a new tesselator with only the given callbacks */
someTesselator tess = new someTesselator( BEGIN_CALLBACK |
                                          END_CALLBACK |
                                          VERTEX_CALLBACK );
```

A point of interest regarding the OpenGL pipeline object that is used within the tesselator implementation example above is worth mentioning. The `GLUTesselator` base class contains `CoreGL` and `CoreGLU` objects called `gl` and `glu` that can be used within any subclasses of `GLUTesselator` to route OpenGL and GLU methods through. This saves you declaring your own, although if you want to take advantage of a composable pipeline, you can declare your own locally within your tesselator subclass and use those instead.

A final point is that Magician currently only *portably* implements the GLU 1.1 tesselator specification although handling for GLU 1.2 is present and can be used for, say, combine callbacks and boundary tesselation.

The Shapes Utility Class

Magician also supplies a utility class called `com.hermetica.util3d.shapes` which defines methods that will generate many common geometrical shapes that you can use within your programs. C programmers will be familiar with these shapes as these routines were implemented firstly in the *tk* toolkit that historically was supplied with some implementations OpenGL and latterly in *GLUT*.

The various shapes that can be generated are the *cone*, *cube*, *tetrahedron*, *dodecahedron*, *icosahedron*, *octahedron*, *sphere*, *torus* and *teapot*. These functions operate in two different modes that generate either *solid* or *wire-frame* objects. Therefore, to generate a solid teapot you would call the `solidTeapot()` method with appropriate arguments. A wire-frame teapot can be generated by invoking `wireTeapot()`. All the methods defined within this class are declared as being `static`. Therefore, to use them, simply invoke `shapes.wireTeapot()` and so on. You must remember to have a current rendering context before you invoke any of these methods or unpredictable results may ensue.

The methods within the `shapes` class are all written in pure Java which ensures that they will operate portably across all platforms Magician supports. This also implies that on unoptimized Java VMs, performance on complex objects such as the teapot may be poor. To avoid repeated performance impacts, it is wise to create a display list encapsulating each object that you plan on using. This will reduce the quantity of Java code that needs to be interpreted in order to produce the shape. For example, encapsulating a solid teapot in a display list can be written as

```
/** Generate a new display list identifier */
int teapotList = gl_.glGenLists( 1 );

/** Create the display list with the teapot */
gl_.glNewList( teapotList, GL.GL_COMPILE );
    shapes.solidTeapot();
gl_.glEndList();
```

Some of the shapes take arguments allowing you to specify the fineness of *tesselation*. That is, a higher degree of tesselation will produce a finer shape using many triangles whereas a lower degree of tesselation will produce a rougher shape but with far less triangles. For example, you might wish to render a highly tesselated sphere for close-up objects and a roughly tesselated sphere for distant objects. This will give you maximum flexibility of the tradeoff between visual quality and rendering speed. The "Molecule

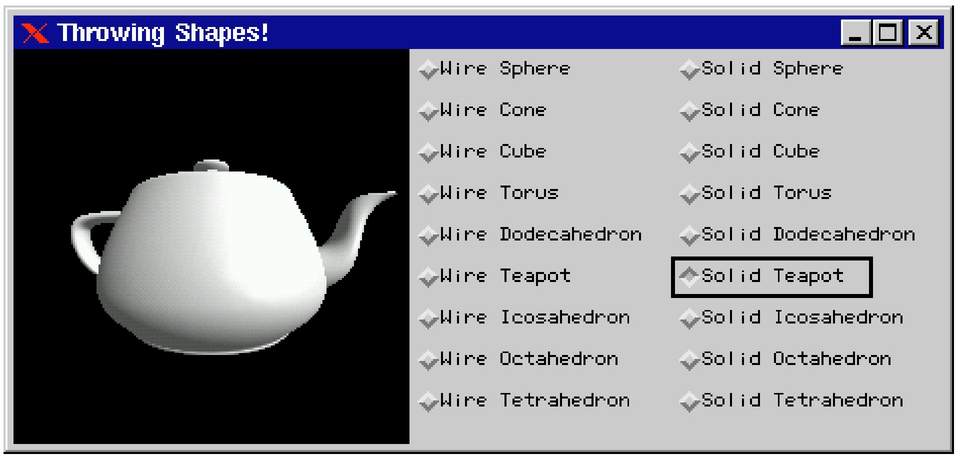

Figure 5.5: "Throwing Shapes" Demo

Viewer" demo bundled with Magician illustrates this technique.

The shapes generated with the `shapes` class also generally automatically generate texture coordinates that can be used to map textures onto the shapes. The texture coordinate mapping can be altered by manipulating the OpenGL texture stack as normal. The "USS Enterpoop" demo bundled with Magician demonstrates warping textures on a cube.

Finally, the "Throwing Shapes" demo demonstrates the visual qualities of all the shapes that can be generated with the `shapes` class. An example of this in action can be seen in Figure 5.5. The full source code for this demonstration is available with Magician and demonstrates the ability to encapsulate shapes within display lists and to scale the shapes while ensuring the normals are still correct amongst other things.

Associated Magician Demo Programs

advanced/boundary.java This demonstration illustrates the use of GLU tesselators to calculate the silhouette outline of arbitrary 3D shapes in realtime

advanced/shadowfun.java This demonstration uses GLU tesselators to calculate the silhouette of objects in realtime which is then used to construct a shadow volume to produce realtime complex shadowing.

redbook/quadric.java This demonstration illustrates the use of GLU Quadric objects within Magician applications

redbook/bezcurve.java This demonstration uses 1D NURBS evaluators to draw a single curve and its control points.

redbook/bezsurf.java This demonstration draws a wireframe 2D NURBS surface

redbook/bezmesh.java This demonstration draws a shaded 2D NURBS surface

redbook/texturesurf.java This demonstration draws a textured 2D NURBS surface where the texture has also been generated using the NURBS evaluator functions

redbook/surface.java This demonstration draws a NURBS surface with toggleable control point display

glut/tessdemo.java This demonstration allows you to draw complex polygons on a grid and then tesselate those polygons for rendering by OpenGL

glut/dinoball.java This demonstration tesselates complex polygons into a three-dimensional dinosaur!

glut/dinoshade.java This demonstration tesselates complex polygons into a three-dimensional dinosaur and also allows you to control various environmental and rendering effects such as reflection and shadow-casting all in real-time!

throwShapes.java This demonstration allows you to selectively display each of the primitive available through the utility `shapes` class.

enterprise.java This demonstration draws the USS Enterpoop chasing a Borg cube over a planet's surface. The planet's surface and Borg cube are both textured from images stored at URLs.

Chapter 6

Image Input and Output

This chapter discusses several advanced features of Magician that can be used to provide powerful texture-map handling and the production of high-quality output from rendered scenes.

Texture Maps

Texture-mapping is an extremely common activity in 3D applications today as the speed of processors increases and dedicated graphics cards become cheaper and more widely available.

OpenGL features several functions that allow for the mapping of textures declared as blocks of data to be mapped onto the surface of geometries in a scene. However, there are no standard image loading mechanisms defined in OpenGL for loading external images that can be converted into data blocks suitable for texture-mapping. This requires that OpenGL developers using C or C++ must either write their own image decoding routines or integrate existing image decoding libraries into their applications.

Magician provides a more seamless method to load external image data for texture-mapping that uses the existing image decoding routines that standard Java uses. Java features the notion of an abstracted `Image` class that encapsulates images in many common formats such as *GIF* and *JPEG*. Java images also have the ability to be loaded over the network by specifying their location as a URL instead of simply as a filename on a local disk.

Magician uses this remote image loading and decoding functionality in the `com.hermetica.util3d.Texture` class which provides a portable way to load texture-map data. This class can load and decode images from any network location *via* URLs and produces data in the format that can be used by OpenGL's texture-mapping routines. This functionality allows developers to write network-aware OpenGL applications and standard OpenGL applications that both use texture-mapping with minimum fuss. For example, a short code stub to load texture data from a URL can be written as

```
Texture tableTexture = null;

if ( tableTexture == null ) {
    try {
        Image tableImage =
            Toolkit.getDefaultToolkit().getImage(
                new URL( "http://www.arcana.co.uk/img/logo.gif" ) );
        tableTexture =
            new Texture( tableImage, glc, Texture.SCALE_NEAREST );
      } catch ( Exception e ) {
        e.printStackTrace();
      }
  }

while ( tableTexture.isValid() == FALSE ) {
    /** Wait 100ms and retest */
    Thread.currentThread().sleep( 100 );
  }

.
.
.
```

OpenGL has the limitation that texture-map data must be dimensioned as being to the power of 2 on each axis. For example, a texture-map of 129×140 would be illegal whereas a texture-map of 128×256 would be legal. The `Texture` class features the ability to automatically scale textures as they are downloaded to provide legal texture-map data to you.

To access this functionality, an additional argument may be passed in the constructor of a new `Texture` object. There are several options to scaling the texture.

`SCALE_DONT` This value specifies that your texture data is already correctly dimensioned and that the `Texture` class should not attempt to internally scale the texture. Using this value on pre-scaled textures will result in much faster processing of the texture as the texture is downloaded.

`SCALE_NEAREST` This value specifies that each axis of the image should be independently scaled to the nearest power of two. For example, an image originally sized at 129×255 will be scaled to 128×256. If the image is already correctly scaled, no scaling will occur. This option is the default if no overriding value is set *via* the `Texture` constructor.

`SCALE_MAGNIFY` This value scales both axes of the image up to the next power of two. If the image is already scaled correctly, no scaling will happen. For example, an image of dimension 129×257 will be scaled to 256×512 using this setting.

`SCALE_MINIFY` This value is the corollary of `SCALE_MAGNIFY` in that both axes of the original image are scaled down to the nearest power of two. For example, an original image of dimensions 127×255 will be scaled to 64×128. Again, if the original image is already correctly scaled, no rescaling will occur.

Since texture loading uses the underlying Java AWT Image mechanisms, the actual retrieval and decoding of the texture data uses the asynchronous `ImageProducer` interface. This has the knock-on effect that the texture data may not be ready for fetching for some seconds after you invoke the `Texture` constructor. To alleviate problems that may occur with trying to use texture data that has not completely downloaded, the `Texture` class features a method `isValid()` which returns a `boolean` value signifying whether the texture data has been downloaded and validated. Only when `isValid()` returns `true` should the texture data be used.

Other useful methods defined within the `Texture` class for ascertaining information on the downloaded texture are `getWidth()` and `getHeight()` which return the size of the texture *after* any scaling has occurred. These methods can be used to return the size of the texture for passing into OpenGL texture-mapping routines.

These aspects of texture handling can be used easily within your code when setting up texture objects. For example, if you wished to use `gluBuild2DMipmaps()` to create a group of mip-mapped texture from your texture data, you only need to write

```
Texture texture = null;
```

```
/** Fetch the texture */

/** Setup texturing in OpenGL */
if ( texture.isValid() ) {
    glu_.gluBuild2DMipmaps( GL.GL_TEXTURE_2D, 4,
                            texture.getWidth(),
                            texture.getHeight(),
                            GL.GL_RGBA, GL.GL_UNSIGNED_BYTE,
                            texture.getTexture() );
    /** Other texture setup here, e.g., glTexParameteri() */
  }
```

Once the `Texture` object has been downloaded and validated, it may be re-used over and over again as the texture is now managed within OpenGL. If you are finished with a texture completely, you should firstly invoke `glDeleteTextures()` within OpenGL to clear the texture cache, then set the `Texture` object to `null` to ensure it is garbage-collected.

Image and PostScript Production

Magician can be used to directly dump images, or snapshots, of the framebuffer as it is rendering through the extension of the standard Java `ImageProducer` interface. Similarly, high-quality PostScript output can be easily generated using the same interface.

The `GLDrawable` classes implements the Java `ImageProducer` interface meaning that you may request any `GLDrawable` to transmit image data, in this case the framebuffer contents, to registered `ImageConsumer`s which operate in some way upon the data. Figure 6.1 illustrates the principles involved.

File Output Writers

Magician provides two output writer classes with the standard distribution which write *Portable Pixmap*, or *PPM*, format images and *Encapsulated PostScript* documents. Adding extra writer formats is extremely straightforward and the source code of the two supplied writers should be used as guidelines.

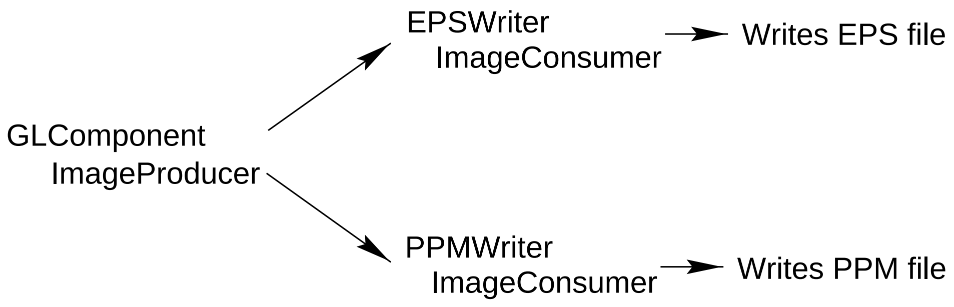

Figure 6.1: The `ImageProducer / ImageConsumer` Architecture

These two classes simply create a new file within the current directory from which the Magician application was started. The `EPSWriter` class writes images in the format of `epswriter?.eps` where `?` is replaced with a unique integerial index. Similarly, the `PPMWriter` class writes files with the naming convention of `ppmwriter?.ppm`. The standard prefix for filenames can be set from the constructor of each class.

Using these classes within your Magician applications is extremely straightforward and generally involves adding a single line of code! For example, to write out a defaultly named PPM file containing the contents of a `GLComponent`s framebuffer, you need only write code like

```
/** Render the scene */
public void display( GLDrawable component ) {
    /** Render the scene */
    ...

    /** Write out the buffer contents */
    PPMWriter writer = new PPMWriter( component );
  }
```

This will cause the contents of the back-buffer to be written out to a uniquely named file. The `EPSWriter` class can be used in exactly the same way.

By default, the back buffer is used to read the framebuffer contents from to ensure a cleaner image grab, however, this behaviour can be changed by invoking the `GLComponent` method named `setReadBuffer()` with an argument value of the buffer from which you wish the reading to occur. For example, if you wished to perform stereo reading, you might wish to create images from the left and right buffers. The code to accomplish this is

```
/** Do left-stereo read into files prefixed with leftbuffer */
component.setReadBuffer( GL.GL_BACK_LEFT );
PPMWriter leftWriter = new PPMWriter( component, "leftbuffer" );

/** Do right-stereo read into files prefixed with rightbuffer */
component.setReadBuffer( GL.GL_BACK_RIGHT );
PPMWriter rightWriter = new PPMWriter( component, "rightbuffer" );
```

This would then write out two sets of images named `leftbuffer?.ppm` and `rightbuffer?.ppm` containing the appropriate buffer date.

The functionality to switch the target read buffer is also pertinent in cases where a single-buffered OpenGL context is being used instead of a double-buffered context. As no back buffer will exist in single-buffered contexts, the results of reading from such a buffer, as happens by default, will be undefined leading to potentially strange results. In single-buffered contexts, the front buffer should be used for reading, but be careful to ensure that all the OpenGL commands have been flushed by calling `glFlush()` or `glFinish()` before creating the `EPSWriter` or `PPMWriter`.

Associated Magician Demo Programs

mesa/reflect.java This demonstration reads an image from a URL and texture-maps it onto a spinning surface. The two geometric shapes are reflected on the surface in real-time.

enterprise.java This demonstration draws the USS Enterpoop chasing a Borg cube over a planet's surface. The planet's surface and Borg cube are both textured from images stored at URLs.

Chapter 7

Fonts

The window-system specific OpenGL protocols, such as GLX and WGL all allow developers to use system fonts as bitmaps that can be rendered onto the drawing surface.

This is a major problem within a portable environment such as Magician in that there is no straight-forward way to abstract the way in which fonts are represented and manipulated on different operating-systems. For example, X Windows specifies fonts as massive strings such as

```
-adobe-times-medium-r-normal--12-120-75-75-p-64-iso8859-1
```

whereas Windows requires you to populate C structures with the desired font characteristics. In addition to this, neither operating system seems to share much information and use different methods of specifying font characteristics, for example, character width and height, baseline positioning and so on.

A problem closer to the heart of the developer is guaranteeing that a particular font is present on the machine upon which the application is running. For example, you might have written an application that displayed illuminated manuscripts using a half-Uncial font. After shipping this application, you discover that if the appropriate half-Uncial font isn't available on a user's machine, it defaults to Courier! This somewhat ruins the effect and visual splendour invested in the application as can be seen in Table 7.

Instead of even attempting to work out some abstracted API for font lorem ipsum

`lorem ipsum...ack!`

Table 7.1: Fallback Fonts

access, Magician side-steps the issue and provides a slightly different but more powerful solution. Magician generates Java classes containing bitmap data representing each character in a particular font at a given font size.

This has several immediate benefits to you. Firstly, it allows you to distribute the exact fonts that you wish to use within your application with your application. Secondly, you can distribute these fonts over networks in exactly the same way in which any other Java classes can be distributed.

Generating Fonts

Magican is supplied with utility programs that can be used on various operating systems to automatically generate Java code that encapsulates a font at a given point size. These programs generate *bitmap fonts*, that is each character is represented as a small picture of the character at the given font size. It is important to rememeber that if you wish to use a larger font, it is better to generate a new Java class file for the font at the larger point size otherwise the bitmap representing the font at the smaller point size will start to pixelate as it is made larger.

The program shipped with Magician for generating the Java code encapsulating fonts is called `generate-xfont` and runs on X Windows platforms. `generate-xfont` connects to the X server specified by the `DISPLAY` environment variable and generates the appropriate bitmaps from the X fonts available to you. You can check which fonts are installed by running `xlsfonts` or browse them with `xfontsel`.

Using Bitmapped Fonts

Magician is also supplied with Java classes encapsulting several common fonts at commonly used point size. These fonts can be found in the `com.hermetica.magician.fonts` package. The correlation between fonts and class names are as follows

`GL8x13BitmapFont`	X Windows 8x13 font
`GL9x15BitmapFont`	X Windows 9x15 font
`GLHelvetica10BitmapFont`	10 point Helvetica / Arial
`GLHelvetica12BitmapFont`	12 point Helvetica / Arial
`GLHelvetica18BitmapFont`	18 point Helvetica / Arial
`GLTimesRoman10BitmapFont`	10 point Times Roman
`GLTimesRoman24BitmapFont`	24 point Times Roman

All Magician font classes are subclasses of the `GLBitmapFont` class and can therefore all adhere to the same format and structure and can be used interchangably. Within each font, each character is defined as a chunk of data that is used with `glBitmap()` to render the character. However, each font contains a `drawString()` method that enables you to simply pass the string you wish to draw as an argument and the string will be rendered at the current screen position in the appropriate font.

For example, rendering a string to the screen in 24 point Times Roman can be achieved by doing

```
/** TimesRoman24 */
GLTimesRoman24BitmapFont font =
    new GLTimesRoman24BitmapFont();

/** Draw the string */
font.drawString( "A String" );
```

Switching between fonts on the fly is also extremely easy and can be effected by writing

```
/** Generic font */
GLBitmapFont font = null;

/** TimesRoman24 */
GLTimesRoman24BitmapFont timesRoman24 =
    new GLTimesRoman24BitmapFont();

/** 18 point Helvetica */
```

```
GLHelvetica18BitmapFont helvetica18 =
    new GLHelvetica18BitmapFont();

/** Render the first string in Times Roman */
font = timesRoman24;
font.drawString( "String 1 is Times Roman 24" );

/** Render the second string as Helvetica 18 */
font = helvetica18;
font.drawString( "String 2 is Helvetica 18" );
```

A downside of rendering fonts in this manner is that they cannot be rotated or projected onto the screen. This is because `glBitmap()` is used to draw the characters when `drawString()` is invoked and this always draws characters "flat" onto the screen.

Font Copyright

It is perhaps worth mentioning some of the legal issues regarding the generation and inclusion of font data within your applications for distribution.

Within the USA, the position appears to be that typefaces are not copyrightable, bitmapped fonts are not copyrightable but scalable fonts *are* subject to copyright. With regards to the font data generated for use with Magician, the information generated is that of a *bitmap* font, not a scalable font. Therefore, it would appear that distribution of encapsulated bitmap font data with your applications is perfectly acceptable under general rules. The position in Europe or other continents may be somewhat different.

You should also check the distribution agreements that are distributed with purchased fonts. The author believes that Adobe, for example, allow bitmapped font distribution, but *always double-check.* This situation may differ on a font-by-font basis.

Arcane Technologies Ltd. do not bundle any non-public-domain bitmapped fonts with Magician and cannot be held responsible for any legal issues arising from the use of Magician regarding breach of font copyright.

Associated Magician Demo Programs

glut/bitfont.java This demonstration draws strings on the window using various bitmapped fonts.

glut/fontdemo.java This demonstration draws various strings on the window. You may also select different strings to display, change the colour of the strings and the font used to render them.

solarSystem.java This demonstration uses Magician's bitmapped font capabilities to display the current selected planet in the solar system. Moving your mouse over a planet will display its name.

Chapter 8

Miscellaneous Utility Classes

Magician comes bundled with a group of utility classes that in some cases are used internally by Magician. Other classes are provided as conveniences to you for programming OpenGL applications.

All these classes are contained within the `com.hermetica.util3d` package and serve a wide variety of tasks from high-speed thread locking to calculating view volume jittering for accumulation rendering.

`CriticalSection` – Mutual Exclusion Lock

The `CriticalSection` class has been provided as a utility class to allow developers to take advantage of a high-performance *mutual exclusion lock* in Java. This solution is advantageous as it removes the requirement to declare methods requiring synchronization as `synchronized`.

Several of Magician's core classes use `CriticalSection` objects to handle high-speed internal locking, for example, `GLContext` and `GLComponent` both use internal locking.

New lock objects can be created by executing the default constructor of `CriticalSection`. For example,

```
    CriticalSection lock = new CriticalSection();
```

This will create an anonymous lock that is currently unlocked. For more accurate tracing and debugging, it is recommended that you name your locks. This can be accomplished by either setting the lock name *via* a constructor

```
CriticalSection lock = new CriticalSection( "context lock" );
```

or *via* the `setName()` method.

```
CriticalSection lock = new CriticalSection();

lock.setName( "context lock" );
```

Using a lock object is simplicity itself. To attempt to acquire a lock, you simply need to invoke the `lock()` method. If the lock is not currently held, the lock will be acquired and processing will continue. However, if the lock is currently held your code will *spin* on the lock until the lock is released.

Similarly, to release a lock you need to call the `unlock()` method. This will immediately free the resource and notify any threads waiting on the lock. The waiting process that acquires the lock cannot be guaranteed.

If you experience problems in your code such as *deadlock*, you can enable tracing on all operations being carried out by the lock. This can be done by invoking the `setTraceStatus()` method with the argument `true` to enable locking and `false` to disable it. The information generated by tracing is extremely informative and can pinpoint any locking problems you may be having. For example, when a thread acquires a lock, a message similar to

```
-> GLComponent@0-choke: Thread[GLComponent@0,1,main]
                        acquired lock
```

will be printed to your console. This shows that the thread has now entered a critical section of code that can only execute *exclusively.* Similarly, when a thread releases a lock, a message similar to

```
<- GLComponent@0-choke: Thread[GLComponent@0,1,main]
                        unlocking mutex
```

will be displayed. This show that the thread has exited the section of code that must be executed exclusively. A third type of tracing message may be displayed if a thread spins on a lock. This message takes the form of

```
-| GLComponent@0-choke: Thread[GLComponent@0,1,main]
                        waiting on lock acquisition
```

Therefore, the correct sequence of messages that will occur in code where the locking is correctly functioning would be

```
-> ... acquired lock
-| ... waiting on lock acquisition
<- ... unlocking mutex
```

with the middle message being optional.

`FrameRateComponent` – Measuring Frame Rates

The `FrameRateComponent` class is provided to enable you to fairly accurately calculate the *frame rate* of your Magician applications. This activity is similar to using the `ProfileGL` or `ProfileGLU` pipelines but provides a more general figure rather than microsecond accurate timings.

The class functions by taking a number of "samples" across a number of iterations of a rendering and averages the times. The default number of samples to take is `16` but this can be changed if machine performance is "spiky" and a smoother average across a higher number of samples is desired. Generally speaking, the more samples you take, the better the results.

This utility class has two ways in which it can be used namely as a graphical representation of frame rates that can be embedded within your GUI or it can simply print its results to a standard `PrintStream` object.

Using a `FrameRateComponent` is extremely easy to add into your program. The two main methods that you will be using are `start()` and `stop()` which signify when timing is to start and stop respectively. The timing sample is taken as being the cumulative time that all the statements between these two calls take to execute. After the number of samples has been taken, the frame rate is either printed to the output stream or the graphical component is updated automatically.

`MicroTimer` – Microsecond Timer

Within the `java.lang.System` class, Java provides a method called `currentTimeMillis()` which can be used for millisecond accurate timing of programs. For timing large, time-consuming operations this method of time measurement is generally acceptable and will give fairly accurate results. A good example of this is the timing of frame rates where a sample of timings are taken cumulatively and the frame rate averaged over the number of samples.

However, Magician provides the `ProfileGL` and `ProfileGLU` pipelines which allow you to time *each* OpenGL command. In most cases, these operations can take tiny fractions of milliseconds to complete which would result in completely incorrect timing information being reported. For example, the output of verbose profiling might read

```
glBegin() took 0ms to execute
glVertex2f() took 0ms to execute
glVertex2f() took 0ms to execute
glVertex2f() took 0ms to execute
glVertex2f() took 0ms to execute
glEnd() took 0ms to execute
```

This would result in the program taking a total of $0ms$ to execute. Pretty fast software, huh?

This is quite obviously wrong and destroys the point of being able to profile things. The solution to this problem is to use a more accurate timer which will give better results when measureing extremely short time quantums.

The `MicroTimer` class is a small Java class that implements its main functionality in platform-specific native code. That is, it uses the underlying timing mechanisms available for each platform in order to measure more accurate time quantums. There is a caveat here in that the timing functions for each operating systems may themselves be slightly inaccurate due to the clock speed of the processor. This is known as the *granularity* of measurement and `MicroTimer` takes this into account when computing timings.

Using a `MicroTimer` in your own software is very easy. You simply create a new `MicroTimer` object and invoke the `start()` method when you wish timing to begin. When you wish it to stop timing, invoke the `stop()` method. The *delta*, or difference, between starting and stopping can be retrieved by invoking `getDelta()` which returns the number of microseconds timing was enabled. For example

```
/** Create a new MicroTimer */
private MicroTimer timer = new MicroTimer();

/** Start timing! */
timer.start();

/** Do some really pointless stuff */
int total = 0;
```

```
for ( int i = 0 ; i < 100000 ; i++ ) {
    total += 2;
  }

/** Stop timing! */
timer.stop();

/** How long did this take? */
System.out.println( "The useless loop took " +
                    timer.getDelta() +
                    " microseconds to execute." );
```

If you are so inclined, you can also retrieve the granularity at which your processor is operating by invoking the `getFrequency()` method. This is not guaranteed to return anything useful on some platforms and will return `0` in these cases. This value represents the smallest number of microseconds that the operating system's timing mechanisms can differentiate between. For example, a value of `17` would indicate that the operating system's timer "ticks" every 17 microseconds. This implies that operations taking less than 17 microseconds will return slightly incorrect measurements for the same reasons that millisecond timing is wrong.

`accum` – Accumulation Jittering

OpenGL can be used to perform sophisticated multipass rendering that can generate the effects known as *scene anti-aliasing.* This effect renders the given scene several times slightly moving, or *jittering*, the view frustum each render to move the objects in the scene around by very small amounts. Each render of the scene is *accumulated* in the accumulation buffer and by the final render, the slight movement of the view volume softens or *anti-aliases* the sharp edges within the scene given a more realistic image.

Magician provides the `accum` class that has several methods and variables that can be used to fractionally jitter the view frustum to produce scene anti-aliased images. The following example illustrates the use of these methods.

```
/** Setup the accumulation buffer with GLCapabilities */
GLCapabilities cap = glc.getContext().getCapabilities();
cap.setAccumRedBits( 4 );
cap.setAccumGreenBits( 4 );
cap.setAccumBlueBits( 4 );
```

```
/** Draws the scene */
private void displayObjects() {

    float[] torus_diffuse = { 0.7f, 0.7f, 0.0f, 1.0f };
    float[] cube_diffuse = { 0.0f, 0.7f, 0.7f, 1.0f };
    float[] sphere_diffuse = { 0.7f, 0.0f, 0.7f, 1.0f };
    float[] octa_diffuse = { 0.7f, 0.4f, 0.4f, 1.0f };

    /** Do the OpenGL stuff */
    gl_.glPushMatrix();
        gl_.glTranslatef( 0.0f, 0.0f, -5.0f );
        gl_.glRotatef( 30.0f, 1.0f, 0.0f, 0.0f );
        gl_.glPushMatrix();
            gl_.glTranslatef( -0.80f, 0.35f, 0.0f );
            gl_.glRotatef( 100.0f, 1.0f, 0.0f, 0.0f );
            gl_.glMaterialfv( GL.GL_FRONT, GL.GL_DIFFUSE,
                              torus_diffuse );
            shapes.solidTorus( 0.275, 0.85, 16, 16 );
        gl_.glPopMatrix();

        gl_.glPushMatrix();
            gl_.glTranslatef( -0.75f, -0.50f, 0.0f );
            gl_.glRotatef( 45.0f, 0.0f, 0.0f, 1.0f );
            gl_.glRotatef( 45.0f, 1.0f, 0.0f, 0.0f );
            gl_.glMaterialfv( GL.GL_FRONT, GL.GL_DIFFUSE,
                              cube_diffuse );
            shapes.solidCube( 1.5 );
        gl_.glPopMatrix();

        gl_.glPushMatrix();
            gl_.glTranslatef( 0.75f, 0.60f, 0.0f );
            gl_.glRotatef( 30.0f, 1.0f, 0.0f, 0.0f );
            gl_.glMaterialfv( GL.GL_FRONT, GL.GL_DIFFUSE,
                              sphere_diffuse );
            shapes.solidSphere( 1.0, 16, 16 );
        gl_.glPopMatrix();

        gl_.glPushMatrix();
            gl_.glTranslatef( 0.70f, -0.90f, 0.25f );
            gl_.glMaterialfv( GL.GL_FRONT, GL.GL_DIFFUSE,
                              octa_diffuse );
            shapes.solidOctahedron();
        gl_.glPopMatrix();
    gl_.glPopMatrix();
```

```
  }

/** Renders the scene with jittering */
public void display( GLComponent component ) {
    int[] viewport = new int[4];
    int jitter = 0;

    /** Render the scene */
    gl_.glGetIntegerv( GL.GL_VIEWPORT, viewport );

    gl_.glClear( GL.GL_ACCUM_BUFFER_BIT );

    for ( jitter = 0 ; jitter < ACCUM_SIZE ; jitter++ ) {
        gl_.glClear( GL.GL_COLOR_BUFFER_BIT |
                     GL.GL_DEPTH_BUFFER_BIT );
        accum.accPerspective( 50.0,
                              (double)( viewport[2] /
                                        viewport[3] ),
                              1.0, 15.0,
                              accum.j8[jitter][0],
                              accum.j8[jitter][1],
                              0.0, 0.0, 1.0 );
        displayObjects();
        gl_.glAccum( GL.GL_ACCUM,
                     (float)( 1.0 / ACCUM_SIZE ) );
      }
    gl_.glAccum( GL.GL_RETURN, 1.0f );
    gl_.glFlush();
  }
```

The "OpenGL Programming Guide" contains a good section of scene anti-aliasing and frustum jittering and should be consulted closely for more information on this topic.

`trackball` – Quaternion-based Trackball

One of the more common capabilities that 3D graphical applications uses is the ability to spin objects about by clicking and dragging the mouse in a window. This is used in many places, for example, molecular modelling, the "examiner" viewers in VRML browsers, perspective views in 3D modelling applications and sundry others.

This functionality is normally implemented by using a "*virtual trackball*" which effectively tightly encloses a 3D object in a sphere which you "roll around" with your mouse[1]

Magician provides the utility class called `trackball` which provides exactly this functionality in an easy to use fashion. The `trackball` class uses *quaternions* to represent the rotations generated by the movement of your mouse and these are used to generate rotational matrices that can be applied to your scenes.

The first stage in using a trackball within your applications is to link it with a `MouseMotionListener` which will give you the ability to track the movement of the mouse around an AWT component such as a `GLComponent`. Using a `MouseListener` also gives you the ability to handle miscellaneous mouse events.

You then need to declare three variables within your class for the trackball itself, the current rotation represented as a quaternion and the 4×4 matrix representing the quaternion. It's also useful to have a couple of variables for storing the previous mouse coordinates. These can be declared as follows

```
/** The trackball */
private trackball ball = new trackball();

/** The quaternion -- This is always formed from 4 values */
private float[] curQuat = new float[4];

/** The rotational 4x4 matrix */
private float[][] m = new double[4][4];

/** Previous mouse position for tracking motion deltas */
private int prevx = 0,
            prevy = 0;
```

The next operation that needs to be carried out is to initialize the quaternion currently associated with the trackball. This is done *via* the `buildQuaternion()` method provided in the `trackball` class and can be written as

```
curQuat = ball.buildQuaternion( 0.0f, 0.0f, 0.0f, 0.0f );
```

Now we're ready to start handling mouse motion and scene rotating!

The two mouse event handles that are of pertinent interest are `mousePressed()`

[1]Afficionados of the arcade machine "Centipede" will know what I'm talking about.

and `mouseDragged()` which belong to the `MouseListener` and `MouseMotionListener` interfaces respectively. `mousePressed()` is of primary use in setting the coordinates at which the mouse was clicked, *i.e.*, the starting point of the mouse drag. This method can be implemented as

```
/** Handles mouse clicking and sets the coordinates */
public void mousePressed( MouseEvent evt ) {
    prevx = evt.getX();
    prevy = evt.getY();
  }
```

The `mouseDragged()` method is slightly more involved as it will be performing the actual quaternion calculations and using the trackball. The basic premise here is that a new quaternion is created using `buildQuaternion()` calculated from the size of the viewport and the amount the mouse has been dragged. This new quaternion is added to the current quaternion which results in the new rotation of the object. This code can be implemented as[2]

```
/** Handles mouse dragging and calculates quaternions */
public void mouseDragged( MouseEvent evt ) {
    /** Get the dimensions of the source component */
    int width = evt.getComponent().getSize().width;
    int height = evt.getComponent().getSize().height;

    /** Get the current mouse coordinates */
    int x = evt.getX();
    int y = evt.getY();

    /** Calculate the new quaternion... */
    float[] tmpQuat =
        ball.buildQuaternion( (float)( 2.0f * prevx - width ) /
                                     (float)width,
                              (float)( height - 2.0f * beginy ) /
                                     (float)height,
                              (float)( 2.0f * x - width ) /
                                     (float)width,
                              (float)( height - 2.0f * y ) /
                                     (float)height );

    /** Add the new quaternion to the current one */
    curQuat = ball.addQuats( tmpQuat, curQuat );

    /** Repaint the GLComponent */
```

[2]And fortunately, this code is extremely cut-and-pastable!

```
      ((GLComponent)evt.getComponent()).repaint();

      /** Update the mouse coordinates */
      prevx = x;
      prevy = y;
    }
```

That piece of code has now calculated the new quaternion representing the current rotation of the trackball. The final stage is to rotate the object that the trackball is applied to by that quaternion and to do this you need to convert the quaternion into a 4×4 matrix. This matrix will then be multiplied onto the matrix stack. Therefore, the `display()` method of this application can be written as

```
  /** Renders the scene */
  public void display( GLDrawable component ) {
      /** Clear the frame and depth buffers */
      gl_.glClear( GL.GL_COLOUR_BUFFER_BIT |
                   GL.GL_DEPTH_BUFFER_BIT );

      /**
       * Push the current matrix onto the matrix stack
       * for safety.
       */
      gl_.glPushMatrix();

      /** Convert the quaternion rotation into a matrix */
      /** ``m'' was declared above as float[4][4] */
      m = ball.buildMatrix( curQuat );

      /** Multiply this rotation with the current rotation */
      gl_.glMultMatrixf( m );

      /** Render the objects! */

      /** Restore the saved matrix */
      gl_.glPopMatrix();
    }
```

It is also possible to easily implement the ability to continue spinning an object along the same rotation automatically using quaternions. Once you have rendered the scene, you can simply add the new quaternion calculated in the `mouseDragged()` method to the current quaternion again. This will continue the rotation. For example,

```
      ...

      /** Restore the saved matrix */
      gl_.glPopMatrix();

      /** Add the new quaternion again to rotate the scene again */
      curQuat = ball.addQuats( tmpQuat, curQuat );
    }
```

For this to work correctly, the `tmpQuat` variable would also need to be declared globally to the class.

The "Molecule Viewer" demonstration program bundled with Magician uses the technique to automatically continue spinning a molecule and uses the animation features of `GLDrawable` to drive the movement.

Associated Magician Demo Programs

molview/molview.java This demonstration uses a trackball to allow you to spin molecules about.

redbook/accanti.java This demonstration uses the accumulation buffer to perform scene anti-aliasing and uses the `accum` utility class.

redbook/accpersp.java This demonstration uses the accumulation buffer to perform scene anti-aliasing and uses the `accum` utility class.

redbook/dof.java This demonstration uses the `accum` class to perform multi-pass rendering to produce a "depth-of-field" effect which simulates lens focus.

mtpaperplane.java This demonstration uses the `FrameRateComponent` class to produce accurate frame rates for this demonstration.

objViewer.java This demonstration uses the `triReader` class to load triangle data files over the WWW and allows you to spin them about with your mouse.

Chapter 9

Extensions, Legacy Code and Magician

As we have discussed in previous chapters, Magician is an extremely powerful toolkit that can be used to write high-performance and totally portable OpenGL code. However, real world situations dictate that Java does not in most cases perform as well as a compiled language such as C. Additionally, organizations have already invested vast amounts of programmer hours in writing applications using OpenGL and another language.

Furthermore, these applications might take advantage of OpenGL implementation-specific *extensions*. How does Magician help you in these cases? The following sections in this chapter examine the relationship that your legacy code and Magician can form giving you possible migration paths from legacy code to Java and how specific extensions can be accessed and used through Magician.

Legacy Code and Magician

Hopefully by this chapter, you're completely sold on writing code using Magician in order to take advantage of all the powerful features that are on offer. There is, however, one drawback to embracing Magician whole-heartedly.

Legacy code. You've got *millions* of lines of legacy code written in C, C++, FORTRAN, ADA or something even stranger. Now, much as we'd love you to start converting all this code into Java, we're not so naïve to think that you would. Nor do we think you should. Therein lies the path to madness.

However, a less ambitious, but equally useful exercise might be to port the GUI aspects of your code to Magician and leave the back-end processing using the core OpenGL and GLU functions in whatever language you've used. After all, those core functions are completely portable and have no platform- or window-system-specific aspects to them anyway.

Therefore, if you let Magician handle any context switching and drawing surfaces in your application, your legacy code can be wrapped with native methods and called directly from Java as needed. For example, if you have an original piece of code for X Windows that draws a white rectangle to the screen, it might look something like

```
/** X Display */
Display dpy;

/** Window to draw onto.. */
Drawable w;

/** OpenGL rendering context */
GLXContext context;

/** Make the context current somewhere */
glXMakeCurrent( dpy, w, context );

/** Draws a rectangle */
void drawRectangle() {

    glBegin( GL_POLYGON );
        glVertex3f( 0.25, 0.25, 0.0 );
        glVertex3f( 0.25, 0.75, 0.0 );
        glVertex3f( 0.75, 0.75, 0.0 );
        glVertex3f( 0.75, 0.25, 0.0 );
    glEnd();
  }
```

With Magician, you don't need to replace the `drawRectangle()` method (imagine it's 2000 lines long!), but you could easily declare a native method within a Java class such as

```
private native void drawRectangle();
```

This method can then be called within your `display()` method. Because Magician automatically acquires contexts for you, you are *always* guaranteed to be given the correct context and drawing surface to draw onto. Therefore, when you drop into the legacy code to execute `drawRectangle()`, you can be assured that the results will be the same as if you had drawn it in your original program.

This approach allows you to selectively replace parts of your legacy code with Java code and slowly phase out the legacy code in favour of new portable sections.

A secondary use for this approach is to implement large, expensive sections of rendering code in C or C++ which will give you slightly better performance than Java for large execution blocks[1].

For more information on Java native method programming, the book "*Java Native Method Programming*" written by the author and published by O'Reilly & Associates should be consulted[2].

Extensions

OpenGL features a powerful way to arbitrarily extend the functionality of OpenGL without requiring large OpenGL specification changes. This allows vendors implementing OpenGL to make implementation- or hardware-specific capabilities available quickly without causing inconsistencies in the core OpenGL specification.

Extensions, by their very nature, are generally not available on every platform and implementation of OpenGL which implies that a mechanism to access these extensions must be available within Magician in a way that does not compromise its platform-neutrality.

There are two realistic approaches to handling extensions from within Magician. Firstly, to provide standard implementations of all extensions within

[1]This is totally dependent on the speed of your Java Virtual Machine. Some JVMs are approaching the speed of compiled C, whereas others, typically those on UNIX platforms, tend to be quite sluggish compared to compiled code.

[2]Currently not published.

the Magician core with standard access paths. Secondly, to provide an "*Extension Developer's Kit*" allowing developers to add handling for specific extensions to Magician themselves.

The first approach has the merit of providing tested and guaranteedly available extension support but is unwieldy in that it is highly unlikely that Magician could ever support all extensions available on all supported platforms. Additionally, it is unlikely that Arcane Technologies Ltd. would be capable of tracking each an every extension change or implementation on all supported platforms.

Therefore, to ensure that you can always get access to the extensions you need when you need them, Magician has a small set of files distributed with it known as the "*Extension Developer's Kit*" (*EDK*). This kit contains two main elements being a C include file containing some functions and a small program that is used to help you auto-generate Java method declarations and their associated native methods.

However, after reading the following sections on writing extensions, you might decide that you don't want to bother. To cover these occasions, Arcane Technologies Ltd. will gather together pre-generated and pre-compiled versions of as many extensions as they can for easy download and installation.

The Theory of Extension Access

Accessing OpenGL extensions from Magician can be a slightly convoluted process due to the ways in which OpenGL extensions are added and the limitations of the Java language.

The theory behind extension access is fairly simple in that we wish to declare a Java method that drops into native code and accesses the extension. In order for this to work you need to firstly write the Java method declaration and secondly write the native method body that implements that method.

For example, say we wished to provide access to the polygon offset extension. This extension is accessed *via* the `glPolygonOffsetEXT()` function and takes arguments of two `floats`. Therefore, you could write the Java declaration of this method as

```
    /** Java method that enables access to glPolygonOffsetEXT() */
```

```
public native void glPolygonOffsetEXT( float factor, float bias );
```

As this method has been declared as being implemented *natively*, that is, with compiled code, you need to write an appropriate native method body to complement it. For JNI-based Java virtual machines, this can be written as

```
/** Native method implementation of glPolygonOffsetEXT() */
JNIEXPORT void JNICALL
Java_aClass_glPolygonOffsetEXT( JNIEnv *env, jobject arg,
                                jfloat factor, jfloat bias ) {
    /** Make the actual glPolygonOffsetEXT() call */
#ifdef GL_EXT_polygon_offset
    glPolygonOffsetEXT( factor, bias );
#endif /** GL_EXT_polygon_offset */
  }
```

Therefore, when you wished to perform a polygon offset operation, you could now call the new Java method that drops down into native code and executes the correct operation for you.

This process is complicated by the fact that you might have, for example, a set of native methods that handle extensions under X Windows. However, not all OpenGL implementations on X Windows supports these particular extensions. These instances are negotiated around with the `#define` statements in the small code snippet above. Furthermore, it is good practice to “probe” for extensions prior to attempting to execute them within your code. Magician provides such probing methods for you and these will be detailed *infra*.

Fortunately, much as this looks absolutely ghastly, the EDK comes with a small program that will take the arguments that you feed into it and produces all the appropriate Java and native code stub functions that you’ll need.

The Extension Functions

The include file bundled with the EDK defines certain functions that can be used to extract window system-specific information from Magician that may be required to use certain extensions. The polygon offset extension detailed above is quite straight-forward and doesn’t require any information or arguments specific to any platform, but some extensions, for example, stereo rendering, do.

```
/** Sets the SGI stereo buffer to render into */
XSGISetStereoBuffer( Display dpy, Window win,
                     int stereoBuffer );
```

The `Display` and `Window` parameters are both X Windows-specific. How can this be translated to being useful within Magician and, more importantly, portable? Furthermore, doesn't this sort of information make it difficult to change Magician internals at a later date?

The solution we provide with Magician is both portable and abstracts the internal workings of Magician away from your extensions ensuring that your code will always work with any modifications between Magician versions. Similarly, your extensions will work on multiple platforms should multiple platforms support the extensions in question.

The EDK defines 3 functions that can be used by you within your extension native method bodies to extract various pieces of window system information. To use these functions, simply pass a `GLComponent` object into the native method as well as the parameters required by that extension. The EDK functions operate on `GLComponent` objects and extract the low-level information from there.

The EDK functions and their platform-dependent return types are shown in the following table

EDK function	**X Windows**	**Windows95/98/NT**	**OS/2**
`edkGetWindow()`	`Window / Drawable`	`HWND`	`HWND`
`edkGetWidget()`	`Widget`	`HDC`	`HAB`
`edkGetGLContext()`	`GLXContext *`	`HGLRC`	`HGC`

These functions do not necessarily return all the sorts of useful information that you might require, but you may use the window system-specific functions to extract those. For example, the EDK does not define a function to fetch the X `Display` of a `GLComponent`. You can easily find that by fetching the `Widget` then call `XtDisplay( widget )`.

Therefore, the Java method declaration for `XSGISetStereoBuffer()` would be

```
/** Java declaration for XSGISetStereoBuffer() */
public native void XSGISetStereoBuffer( GLComponent component,
                                        int stereoBuffer );
```

Notice how the two window system-specific arguments have been replaced by a single `GLComponent`. The native code can be written fairly simply as

```
/** Native implementation of XSGISetStereoBuffer() */
JNIEXPORT void JNICALL
Java_aClass_XSGISetStereoBuffer( JNIEnv *env, jobject arg,
                                 jobject component,
                                 jint stereoBuffer ) {
    /** Make the extension call */
#ifdef GL_XSGI_stereo_buffer
    XSGISetStereoBuffer( XtDisplay( edkGetWidget( component ) ),
                         edkGetWindow( component ),
                         stereoBuffer );
#endif
  }
```

Thus, this technique retains platform-independence and abstracts the internal functionality of Magician away from you but gives you access to all the pertinent information you might require when implementing extensions.

Using the EDK Extension Functions

The functions detailed above are to be found in the C++ include file bundled with the EDK called `edk.h`. This file should be included by any native C++ files that you have written that require access to the EDK functions.

Furthermore, a small C++ source file called `edk.cpp` is also distributed with the EDK that you must link into your own extension code. This source file contains the implementations of the various EDK functions and interfaces directly with Magician's internals. You should *always* use the supplied EDK functions for accessing Magician internals as the internal access path may be subject to change at any time and should not be directly used.

These two EDK files use preprocessor `#defines` to regulate the way in which they operate. If you are building extensions for use with a JNI-based Virtual Machine, you should define the `HAVE_JNI` symbol in your project or Makefile prior to compilation. Similarly, if you are using Microsoft's Virtual Machine, the `HAVE_RNI` symbol should be defined.

Furthermore, you should also indicate which platform you are building extensions for by defining either `HAVE_LIBX11` if you are using X Windows or `WIN32` if you are building for Windows95/NT.