Arcane Technologies' Response to OpenGL ARB JavaGL Issues

Alligator Descartes

descarte@arcana.co.uk

November 16, 1998

Arcane Technologies Ltd.
PO Box 3738
Glasgow
G41 4YD
Scotland

T: *+44 141 423 3449*
F: *+44 141 423 3449*
http://www.arcana.co.uk
info@arcana.co.uk

Abstract

This document contains Arcane Technologies' response to the list of issues outlined by the OpenGL Architecture Review Board regarding the Java OpenGL bindings, or *JavaGL*.

We have described in detail the solutions for each issue as implemented within our *Magician* technology. Magician is a mature, high-performance, robust and fully portable product which has been on public release for over a year supporting a wide range of operating systems and Java Virtual Machines.

1 Organization

The first area of specific interest requiring additional information covers the topics of how OpenGL and GLU functions and constant variables are organized within the JavaGL class structure.

1.1 How are the methods and constants...?

Magician implements a clearly defined boundary between the traditional OpenGL functions and constants, prefixed with `gl` and `GL_` respectively, and the GLU functions and constants prefixed with `glu` and `GLU_` respectively. Magician implements OpenGL and GLU constants throught a group of interfaces which also provide the basis for Magician's powerful "*composable pipeline*" architecture.

1.1.1 Constants

The constants defined within the OpenGL Specification are declared within Magician as `static` and `final` integer values within an interface called `GLConstants`.

This interface is the only place in which these constants are defined. Furthermore, by using the `static final` modifiers and placing the declarations within an interface you are assured that the values cannot be modified by end-user applications.

For example, Magician 1.0 simply declared all OpenGL constants and methods within an interface called `GL`. This meant that if you wished to reference OpenGL constants you were forced to prepend each constant with an explicit reference to the `GL` interface. A short example of this form of constant addressing follows.

```
/** Draw a triangle */
glBegin( GL.GL_TRIANGLES );
    glVertex2f( 0.0f, 0.0f );
    glVertex2f( 75.0f, 75.0f );
    glVertex2f( 75.0f, 0.0f );
glEnd();
```

This form of constant addressing is not a major problem in itself, but it does reduce the "*cut and paste-ability*" of porting existing OpenGL code from C/C++ since each variable needs to be prefixed with the `GL.` interface reference.

Magician 1.1.0 introduced a new, but backwardly-compatible, solution with the `GLConstants` interface. By implementing this interface within your programs, you automatically make all the constant values defined within it locally available within the implementing class. This means that you don't need to reference the `GL` interface for constant values.

Additionally, since the `GL` interface is a subclass of `GLConstants`, all the constant values are imported into that interface ensuring that existing Magician 1.0 programs that have explicit class references will continue to work without modification.

A short example of this form of constant addressing follows.

```
/** A short class that uses GLConstants */
public class shortClass implements GLConstants {
    ...
    /** Draw a triangle */
    glBegin( GL_TRIANGLES );
        ...
    glEnd();
```

Therefore, the separation of methods and constants in this way affords you flexibility and power when referencing constant values and greatly speeds up the task of porting existing OpenGL code to Java. The separation of methods and constants into separate interfaces also provides the basis of Magician's powerful and innovative "*composable pipeline*" architecture.

Constant addressing for GLU is identical except the `GLUConstants` and `GLU` interfaces are used.

1.1.2 Functions and Composable Pipelines

Composable pipelines are an innovative feature introduced in Magician to allow runtime switching between different OpenGL pipeline functionality including OpenGL function execution, per-statement tracing, per-statement profiling and automatic error-checking. These facilities provide an extremely fully-featured development environment for you to work within and allows for extremely fast program development.

All the composable pipeline classes implement the `GL` interface which requires that all OpenGL methods are implemented in some pipeline-specific way. This also ensures that pipelines can be cast freely between themselves which allows for per-statement pipeline switching to occur. That is, if you wished to enable profiling for a short portion of code only, you can do so simply by switching to a profiling pipeline instead of the usual recompile-and-execute cycle required with C/C++ programming.

Therefore, all OpenGL methods are defined initially within the `GL` interface and implemented with each pipeline class. I shall discuss this further in Section 1.2.1.

GLU methods are implemented in an identical way in the `GLU` interface and various GLU pipeline classes.

1.2 How are the classes and interfaces...?

A complete JavaGL implementation, such as Magician, requires both mappings between the core OpenGL and GLU methods and constants and mappings for utility classes to handle objects such as GLU quadrics, NURBS and tessellators. This section explicitly excludes any window-system integration classes.

1.2.1 GL and GLU

The core OpenGL and GLU functionality is partitioned into several interfaces upon which the whole composable pipeline architecture is implemented. Figure 1 illustrates this more clearly.

The composable pipelines that we ship as a standard part of Magician are the *core execution pipeline* called `CoreGL`, a *tracing pipeline* called `TraceGL`, a *profiling pipeline* called `ProfileGL` and an *error-checking pipeline* called `ErrorGL`. By splitting OpenGL and GLU functionality in this way, you can quickly "switch

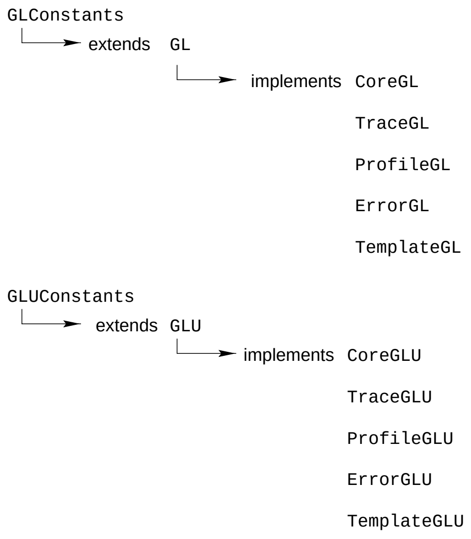

Figure 1: GL and GLU Pipelines

on" functionality that you need to write bullet-proof and performance-tuned applications. However, when you ship your applications, you can simply use the `CoreGL` pipeline for maximum performance. Since the core and extended pipelines all implement the `GL` interface, pipelines can be switched on a statement-to-statement basis without recompilation or any code alteration.

If this functionality wasn't impressive enough, pipelines can be "*stacked*" for multiplied functionality. For example, if you wished to both profile *and* trace your program's execution, you can create a composite pipeline from both `TraceGL` and `ProfileGL`. These composite pipelines behave in exactly the same way as the standard pipelines.

The GLU functions are identically covered by pipelines named `CoreGLU`, `TraceGLU`, `ProfileGLU` and `ErrorGLU` and a base interface called `GLU`.

1.2.2 GLU Quadrics

Magician provides a class called `GLUQuadric` which encapsulates the notion of a GLU quadric object, usually defined as `GLUquadricObj` or `GLUtriangulatorObj` within the C/C++ API.

Quadric manipulation can occur in two ways.

1. *via* GLU, as per the C/C++ API. That is, you can write code in the following manner.

   ```
   GLUQuadric quadric = gluNewQuadric();
   gluDisk( quadric, 1, 10, 10, 10 );
   ```

   which will behave exactly as expected.

2. *via* the `GLUQuadric` object which has methods mapped for each GLU method. For example

   ```
   GLUQuadric quadric = new GLUQuadric();
   quadric.disk( 1, 10, 10, 10 );
   ```

Both of these approaches are functionally identical and generate the same results but offer alternative programming styles to you, the "old-style" and a newer, more object-orientated, style.

1.2.3 GLU Nurbs

Magician offers a full implementation of NURBS by using a similar style of partitioning as implemented with quadrics. In the case of NURBS, a class entitled `GLUNurbs` exists which encapsulates the C/C++ `GLUnurbsObj` data structure.

As with the `GLUQuadric` class, `GLUNurbs` may be used either *via* the GLU pipeline or directly in an object-orientated manner.

1.2.4 GLU Tessellators

Finally, Magician offers a powerful and innovative interface for implementing GLU tessellators within a Java environment.

Magician provides an abstract base class called `GLUTesselator` which handles the basic functionality required by tessellators under Magician. This functionality includes the allocation of internal structures for the tessellators and providing default methods for the tessellator callbacks.

The rationale behind our implementation of tessellators stems from the inablity of Java to do function callbacks or function redirection as used with C/C++ tessellator code. For example, the following C/C++ code will pass all tessellated vertices directly to `glVertex3f()`.

```
gluTessCallback( GLU_VERTEX, (GLvoid (CALLBACK *)())&glVertex3f );
```

This is an extremely powerful system, but one which is not suited to Java. In order to provide an equally powerful Java system, we have decided to implement callbacks that execute defined Java methods within subclasses of the `GLUTesselator` class. This allows you to write customised tessellator callbacks in Java without any difficulty and in a portable manner.

For example, a sample tessellator subclass can be written as

```
public class dinoballTesselator extends GLUTesselator {

    /** Begins tesselation of a polygon */
    public void begin( int mode ) {
        gl.glBegin( mode );
      }

    /** Ends tesselation of a polygon */
    public void end() {
        gl.glEnd();
      }

    /** Handles a single vertex from the tesselator */
    public void vertex( float[] vertex ) {
        gl.glVertex2fv( vertex );
      }

    /** Handles edge flag setting */
    public void edgeFlag( boolean edgeFlag ) {
        gl.glEdgeFlag( edgeFlag );
      }
  }
```

These Java methods are invoked from within the native code when the native OpenGL implementation decides the callback should be invoked. This technique affords identical power to you as if you were writing tessellator code in C/C++. The actual passing of tesselator data can be expressed as going *via* the standard GLU pipeline classes or directly *via* the instantiated tessellator object if you prefer writing object-orientated code as described for GLU quadrics and NURBS.

A final note on tessellator usage is that the GLU Specification defines the generation of primitives depending on which callbacks are registered. That is, if you do not wish to register an *edge flag* callback, the tessellator will attempt to generate triangle strips as opposed to triangles. Since Magician provides no explicit system for registering callbacks as it is done internally, we supply an alternate constructor for the `GLUTesselator` class which allows you to specify which callbacks you wish to be registered. For example, if you only wished to register the bare minimum of callbacks to allow geometry tessellation to occur, the following code stub would do so.

```
/**
 * Create a new dinoballTesselator ( see above ) with non-default
 * callbacks
 */
dinoballTesselator tess =
    new dinoballTesselator( GLUTesselator.BEGIN_CALLBACK |
                            GLUTesselator.END_CALLBACK |
                            GLUTesselator.VERTEX_CALLBACK );
```

Therefore, on high-performance tessellators, it is likely that triangle strips would be generated instead of triangles.

1.3 Should there be separate packages...?

Concerning the question of whether the window-system functionality for JavaGL should be split into separate classes from the core OpenGL and GLU classes, it is our opinion that such a split is necessary if only to follow the precedent of GLX, wgl *et al.*

From our experience of supporting multiple JVMs and operating systems, the main area of abstraction required lies within the window-system interface and not the "core" APIs as these are already platform-independent. As such, tying the window-system code into the core APIs is an overly restrictive and sub-optimal solution.

2 Versioning

This section discusses how OpenGL versioning is tracked and queried from within a JavaGL implementation.

2.1 How is the native OpenGL API support queried?

The version of the native OpenGL that has been loaded by the JavaGL implementation is queryable by using the `glGetString()` method with the argument of `GL_VERSION`.

2.2 How is version compatibility determined?

Magician handles this internally at the moment for use with runtime function linkage checking. This is discussed in more detail in Section 9.3.

OpenGL Type	**Java Type**
`GLvoid`	`void`
`GLbitfield`	`int`
`GLboolean`	`boolean`
`GLbyte`	`byte`
`GLshort`	`short`
`GLint`	`int`
`GLubyte`	`byte`
`GLushort`	`short`
`GLuint`	`int`
`GLsizei`	`int`
`GLfloat`	`float`
`GLclampf`	`float`
`GLdouble`	`double`
`GLclampd`	`double`

Table 1: OpenGL to Java Data Type Mapping

3 Argument Passing

Java supports a far more restricted range of data types that C/C++. Furthermore, since Java is a strongly-typed and type-safe language, pointers, callbacks and function redirection are not supported within the language. Furthermore, Java does not have any notion of `unsigned` data types nor does it have a way of declaring enumerated types. Finally, because Java does not use a preprocessing stage macro definitions and `#define`s are not available.

Since these features are heavily used within OpenGL and GLU, some form of standardised conversion must be decided upon. The following sections illustrate how Magician has worked around these restrictions to be a fully-functional Java OpenGL interface.

3.1 How are Java types mapped to C types?

Magician uses a standard technique to map the OpenGL data types to Java data types as shown in Table 1.

These values are generally passed through Java unaltered and are explicitly type-cast to the expected data type internally before being passed to the native OpenGL implementation. This generally ensures that no data type corruption, such as `signed` *vs.* `unsigned`, occurs causing errors in application execution.

Magician also converts GLboolean to an actual Java boolean value in order to allow you to write slightly more readable application code. This can produce some slight problems if explicit comparisons to GL_TRUE or GL_FALSE are used, but we have checked a large quantity of OpenGL application code and these values are almost never used for comparison tests. Most developers appear to prefer to write code similar to

```
if ( blah ) {
    ...
  }
```

as opposed to the more correct

```
if ( blah == GL_TRUE ) {
    ...
  }
```

As such, we decided that converting GLboolean to a Java boolean was a sensible and useful change.

3.2 How are unsigned types converted?

The conversion of unsigned data types in C/C++ to Java is treated in a straight-forward manner by Magician. We simply use the signed version of a particular data type in Java which is internally re-cast to being unsigned before being dispatched to the native OpenGL implementation.

3.3 How does JavaGL handle pointers?

Java, being a portable language, does not support the concept of pointers or direct memory addressing in any form. This makes the conversion of functions using pointers slightly problematic.

Magician has implemented the solution of using Java arrays of data instead of pointers to blocks of memory containing the data in question. This solution works admirably in all but one case.

For example, the glVertex3fv() function takes the argument of GLfloat * which points at a contiguous block of memory containing three GLfloat values. Magician defines glVertex3fv() as having an argument of float[] which should have a length of 3.

For functions that take GLvoid pointers as arguments, we have implemented all the possible variants for that function for each data type. This allows you to use all of OpenGL that is available to you under C/C++ but has the benefit of being compile-time checkable.

The only situation that this solution does not work is with interleaved data arrays composed of multiple data types such as `GL_C4UB_V2F`. One possible solution we have tested is to create an array of objects each stuffed with appropriate values simulating a C `union`. However, the performance on this solution is so utterly atrocious it completely invalidates the point of using interleaved data arrays in the first place. It also brings up many implementation issues as the physical data location and contiguousness which causes implementation problems on multiple JVMs. Therefore, Magician currently does not support mixed interleaved data but does support non-mixed interleaved data.

3.4 How does JavaGL handle callbacks?

Callbacks are implemented within GLU only for use with quadrics, NURBS and tessellators and are not implemented at all within the core OpenGL.

Magician side-steps the more traditional form of callback registration *via* `gluQuadricCallback()`, `gluNurbsCallback()` and `gluTessCallback()` by simply allowing you to subclass the appropriate base classes of `GLUQuadric`, `GLUNurbs` or `GLUTesselator` accordingly.

For example, if you wished to implement an custom error callback, you could simply subclass `GLUQuadric` and implement the `error()` method. This would ensure that if a quadric generation error occurred, your custom method would be invoked.

GLU tessellator callback handling is far more detailed than the simple callbacks registerable for quadrics and NURBS handling and is detailed in Section 1.2.4 *supra*.

This system is far more powerful than the alternatives for several reasons.

1. It is compile-time checkable. One possible, but rejected, solution we tested registered named Java methods as being the callback to execute when an action occurred. For example,

   ```
   gluTessCallback( tessellator, GLU_ERROR, "error" );
   ```

   The downside to this approach is that the named callback is not compile-time checkable, and if it doesn't actually exist, the JavaGL implementation would error internally in a way that might not be recoverable by the application.

2. It is extremely simple to use within the context of Java compared to fiddling about with the various callback calls which are very C/C++ in style.

3. It defines a standard base behaviour for each callback in the base class which will perform useful callback operations even if not explicitly overridden.

3.5 How does JavaGL handle enumerated types?

As Java does not handle enumerated types, Magician simply converts all the enumerated type values to their corresponding integerial values. This solution requires a certain amount of monitoring as these values may change between OpenGL versions, but doing so would destroy backwards-compatibility in both OpenGL and JavaGL. As such, it's a *potential*, but unlikely, issue.

3.6 How does JavaGL handle `typedefs` and `#defines`?

Again, Java does not support the notion of `typedef`s and has no *standard* preprocessing stage to support `#define` macros.

Magician simply ignores `typedef`s completely as they are not widely used within the `gl.h` and `glu.h` header files other than to map OpenGL data types onto C data types, *e.g.*, `GLfloat` to `float`. Since this data type mapping has already been performed as discussed in 3.1, the `typedef`s are simply not required in JavaGL.

3.7 Are OpenGL functions overloaded or does JavaGL use variants?

Magician implements two strategies regarding OpenGL function naming and access paths.

Standard Functions Magician implements all of the "standard" C/C++ OpenGL functions exactly as they are defined within the OpenGL and GLU Specification documents with variants substituted where `GLvoid *` is used in arguments. This form of function addressing is *required* as far as we are concerned. This provides you with the ability to quickly write JavaGL code if you are used to the C/C++ APIs. This is also the case from the point of view of porting existing C/C++ application code to Java. By supplying these functions, porting is reduced to being 99% cutting and pasting.

Overloaded Functions Magician also implements the notion of polymorphic methods within its core pipelines. For example, the variants of `glVertex[234]*()` such as `glVertex3f()`, `glVertex2d()` *et al* are reduced into a single set of methods called `vertex()` which are multiply defined with differing arguments. That is,

```
vertex( float[] v );
```

```
    vertex( double x, double y, double z, double w );
    vertex( short x, short y );
```

and so on.

Another benefit of this form of addressing is that the explicit namespacing prefix of `gl` can be dropped since the JavaGL namespace within Java is enforced by Java itself rather than the global free-for-all that manifests in C.

We consider this form of function addressing to be *required* by a JavaGL implementation as it greatly improves the readability of OpenGL programs and also uses the functionality that is present within Java to powerful effect.

We also feel this form of addressing is most useful to developers who are starting off with OpenGL using Magician and do not really wish to be totally bewildered by the sheer quantity of OpenGL functions available to them.

Therefore, we would recommend that this form of function addressing is also included within JavaGL to complement and enhance the standard form of function addressing.

4 Java Integration

This section discusses some of the ARB members' concerns about the way in which JavaGL integrates with the Java Virtual Machine technologies provided by third party vendors such as Sun, Microsoft, Symantec and IBM amongst others.

4.1 How does JavaGL access system fonts?

The topic of font access is an extremely tricky one to approach because of the vast difference between the ways in which each operating system implement fonts, *e.g.*, BDF, PCF and SNF format fonts under X11, TrueType and Adobe Type1 font under Windows and MacOS, not to mention the native Windows font format and the standard MacOS formats.

There are several ways in which font access can be implemented. These include

1. The use of Java fonts as defined by the `java.awt.Font` classes. This limits the fonts available to you to the fonts specified within the standard

Java class, *i.e.*, Times-Roman, Helvetica and Courier. This functionality is implemented in all JVMs, but does not implement any functionality to define your own fonts for use.

2. Using Java2D. This solution appears to be a lot better than the former in that custom fonts can be used. There are two major downsides to using Java2D. Firstly, there is a requirement that the fonts be installed on the end-user's machine and secondly that the JVM the end-user is executing the JavaGL application on supports Java2D.

   In the case of fonts being installed, this brings up the problems of font licensing and expected behaviour when the desired fonts are not available.

Magician implements a third strategy that is neither subject to the reliance of non-core JVM functionality such as Java2D nor the reliance on installed fonts on end-users' machines. Our solution also works with the most common font foundries' licensing agreements for the distribution of font info.

We have implemented a framework of classes that allows you to quickly render font glyphs onto an OpenGL drawing surface. We also provide two utility programs to generate the appropriate Java classes that encapsulate the font glyphs suitable for this framework. By using this framework, you can easily integrate font rendering into your software with the following benefits.

1. Font data is encapsulated within Java classes which can be delivered *via* the network or as part of an application. This removes the dependency on assuming that the end-user has the appropriate fonts installed on their machine for the application to look correct.

2. Because the font data is stored in the format of bitmapped data within a Java class, you will not be in breach of font distribution policies which regulate only the distribution of *scalable* font data.

3. The font glyph bitmaps can be used in other ways, for example, textured textual data instead of simple bitmapped data.

The class structure implemented by Magician to support bitmapped font rendering relies on an abstract base class called `GLBitmapFont` which defines methods for rendering individual glyphs and strings of glyphs within the font.

The font-generating utility programs simply generate Java code in the form of subclasses of `GLBitmapFont` which defines the individual glyphs. These classes can be compiled and distributed with your applications allowing them to be sure that the application will look identical and correct on all platforms.

4.2 How does JavaGL handle class loading?

There are a few different approaches that can be taken to handling JavaGL class loading depending on how you may wish multiple JavaGL implementations to interoperate or be selected between.

4.2.1 No Interoperation

If no interoperation is desired between JavaGL implementations, that is, multiple JavaGL implementations may not be installed and used simultaneously, then there are no real additional requirements to handling the bootstrapping of JavaGL.

However, this is possibly not a desirable solution as users may want to install multiple versions of JavaGL simultaneously for either evaluation, or to mix and match optimal aspects of each implementation.

An argument for the unique operation of each implementation is that the boundary between implementations is clearly delineated. For example, a complete implementation is installed and used, but the user also installs a less complete or less powerful implementation which leads the user to construe that the former implementation is at fault. This could increase support for the entirely wrong product and would require some form of monitoring.

4.2.2 Interoperation

To support safe interoperation between JavaGL implementations simultaneously installed on a machine, there are some possible solutions that could be implemented to bootstrap the desired implementation.

All of the following techniques assume the existence of a factory which would be supplied as standard by all implementations. This factory would contain methods such as `createGL()` to allocate OpenGL pipeline classes using the selected implementation and so on for all "objects" defined within the JavaGL Specification.

Arcane Technologies are still evaluating solutions to this issue and the details described in the above section are best regarded as "fluid". There has been discussion on the working group mailing list to this end, but no conclusions have been drawn as yet.

4.3 How are differences in the native JVM handled?

Despite the desire of Sun, many JVM vendors have implemented high-performance proprietary native method interfaces for their JVMs. Most notable of these interfaces is Microsoft's *Raw Native Interface*, or *RNI*. This makes it considerably more difficult to develop a portable and robust JavaGL implementation as each native method interface and operating system will require a separate branch of code.

Magician provides a completely abstracted view of the OpenGL, GLU and various window-system APIs allowing you to write 100% portable code that operates identically on all JVMs and operating systems. The main levels of abstraction exist within the window-system code as opposed to the OpenGL and GLU pipelines. The main differences in the window-system code pertains to the configuration of visuals or device contexts and to this end, Magician defines a class called `GLCapabilities` that allows portable visual configuration. This allows you access to all the functionality that can be accessed *via* the window-system protocols such as GLX and wgl.

In order to support a wide range of platforms and JVMs, Magician internally implements many code branches ensuring that the shipped Java class files are identical for each platform. Only the native library is different. This technique allows us to run code on Linux, Solaris, Irix, Windows95/98/NT, OS/2 and MacOS on many JVMs without modification and ensures identical behaviour and rendering quality on all platforms.

5 Window System

This section discusses the less platform-independent topics of window-system integration with Java's windowing technologies such as AWT and Swing and the underlying operating-system specific protocols such as GLX, wgl, PGL and AGL.

Magician has defined a set of classes that provide a totally platform- and JVM-independent abstracted view of these protocols and window-systems allowing you to write powerful and fully portable code with ease.

Magician provides a class called `GLCapabilities` which is an abstracted view of the visual and device context capabilities that can be set within the most popular window-system and OpenGL windowing protocols. For example, the depth buffer size can be configured, the number of auxiliary buffers, whether or not double-buffering is in operation and so on.

The `GLCapabilities` class provides about 20 common characteristics or capabilities which OpenGL uses to configure the way in which framebuffers are rendered to a window or off-screen buffer. These characteristics can be set and read back *via* accessor methods and it by using `GLCapabilities` that solutions for many of the following questions is implemented.

5.1 How does JavaGL interact with Swing?

Magician interacts seamlessly with Swing through the `JGLComponent` class which is a fully Swing- and AWT-compliant GUI component that can be rendered onto from OpenGL.

`JGLComponent` is supplied separately from the heavyweight `GLComponent` class as the underlying logic for locating drawing surfaces to render onto differs greatly. From an application perspective, there are no differences between the components in terms of API or functionality. The only difference lies in the fact that `JGLComponent` is completely *lightweight*.

Being completely lightweight, `JGLComponent`s can be easily added into complex Swing layout managers such as splitter panes and `JInternalFrame`s without any trouble whatsoever. Other Swing-related problems with heavyweight components such as z-ordering and peerless rendering are side-stepped completely.

5.2 How does JavaGL handle lightweight *vs.* heavyweight?

Magician defines two Java components suitable for rendering onto within a Java GUI framework.

`GLComponent` `GLComponent` is the standard heavyweight drawing surface component supplied with Magician. It is a subclass of `java.awt.Canvas`, which is in itself heavyweight, and uses underlying private window-system resources internally.

`GLComponent` is a fully functional AWT-aware component that can have various "listeners", such as `MouseMotionListener`, attached to it as any other AWT component would have. This design allows OpenGL-aware components to be easily slotted into existing GUIs without requiring elaborate rewrites or redesigns.

`JGLComponent` `JGLComponent`, as discussed *supra*, is a lightweight Swing-aware drawing surface that OpenGL can render onto. Objects instantiated from this class can be used exactly as `GLComponents` *via* AWT or *via* Swing's APIs, for example, tooltips can be attached to a `JGLComponent`.

Additionally, `JGLComponent` can have listeners attached to it in exactly the same way as `GLComponent`s.

You may select which component is required for your particular applications. Heavyweight components are considerably faster and if your applications are not using Swing at all or are using Swing in such a way that z-ordering is not an issue or specialised layout managers are not used then `GLComponent` is the best choice. Otherwise, `JGLComponent` should be used.

Both classes, and the off-screen buffer class, all implement an interface called `GLDrawable` allowing them to be swapped around with minimum fuss and maximum conformance with each other.

5.3 How is Double-buffering handled?

Double-buffering is a windowing function that exists either within the OpenGL window-system protocol, *e.g.*, `glXSwapBuffers()` or `pglSwapBuffers()`, or within the operating system itself, *e.g.*, Win32's `SwapBuffers()` function.

As such, configuration of a double-buffered visual lies within the OpenGL window-system protocol and takes many forms depending on the operating system used.

The `GLCapabilities` class simplifies and unifies the requesting of double- or single-buffered visuals by using the `setDoubleBuffered()` method which takes arguments of either `GLCapabilities.SINGLEBUFFER` or `GLCapabilities.DOUBLEBUFFER`. Single-buffering is enabled by default.

The operation of buffer swapping when a window refresh is required is equally simple. In most cases, `GLComponent` or `JGLComponent` will automatically either buffer swap or flush the pipeline when all OpenGL commands are completed, or you can force a buffer swap at any point by invoking the `swapBuffers()` method defined within the `GLContext` class. Again, this method is totally portable and is translated internally to the appropriate OpenGL window protocol or operating system call.

5.4 How is Off-screen rendering handled?

Magician implements off-screen rendering through the `GLOffscreenBuffer` class which is analogous to `GLComponent` and `JGLComponent` in that it implements the `GLDrawable` interface. Therefore, it is treated by Magician as a valid drawing surface.

Internally, an off-screen buffer behaves almost identically to an on-screen buffer in that a `GLContext` is associated with it. The contents of the off-screen buffer can be read *via* `glReadPixels()` or it can be copied completely to a visible drawing surface by invoking the `GLOffscreenBuffer.swapBuffers()` method which takes a single argument of a `GLDrawable` object. The given object will have the contents of the off-screen buffer copied onto it and be automatically refreshed.

Again, Magician shields the majority of internal work from you and provides a completely portable way of performing off-screen rendering in a simple way that is also seamlessly integrated with Magician's other visible drawing surfaces.

5.5 How are Overlays handled?

Magician has no implemented overlay handling at the moment as of version 1.1.0, purely due to our lack of having a machine capable of supporting overlays. We aim to have overlay support implemented in Magician 1.2.0 which is planned to be available in December 1998.

However, hooks for overlay support already exist within the `GLCapabilities` class.

5.6 How are extended visual capabilities handled?

Extended visual capabilities are handled completely *via* the `GLCapabilities` class as described throughout this section.

5.7 How are Ancillary buffers handled?

Ancillary buffer support is defined within the `GLCapabilities` class by using a group of methods to correctly initialize the underlying visuals.

Depth Buffer Depth buffer configuration is achieved by using the `setDepthBits()` method which takes an integerial argument specifying the number of bits the desired depth buffer should have. The corollary `getDepthBits()` method will return the number of depth bits in a given set of capabilities.

Stencil Buffer Configuration of the stencil buffer can be achieved using the `setStencilBits()` method which takes an integerial argument specifying the number of bits of stencil required. The corollary `getStencilBits()` method will return the number of stencil bits in a given set of capabilities.

Accumulation Buffer Accumulation buffer configuration is performed by using the `setAccumRedBits()`, `setAccumGreenBits()` and `setAccumBlueBits()` methods which configure the depth of the accumulation buffers for each colour. The corollary methods of `getAccumRedBits()`, `getAccumGreenBits()` and `getAccumBlueBits()` are also provided for querying the accumulation buffer configuration.

Auxiliary Buffers Magician allows for easy configuration of the number of desired auxiliary buffers that can be accessed *via* `GLCapabilities`. This can be achieved by invoking the `setAuxiliaryBuffers()` method with the number of required auxiliary buffers as the sole argument. Once the visual is configured, you can query the number of available auxiliary buffers by invoking `getAuxiliaryBuffers()`.

Stereo Buffers Magician has no built-in capabilities for stereo rendering handling as this is typically dependent on platform- and OpenGL-specific extensions.

`GLCapabilities` does not query the underlying OpenGL or operating system for values but merely reflects the values that have been set within an application or the defaults. When a query accessor method is executed, it simply returns the current value. After a `GLContext` has been initialized, the capabilities are *locked* and the `set*()` methods have no effect.

5.8 How are Index *vs.* RGB buffers handled?

The `GLCapabilities` class allows portable and simple configuration of the framebuffer such as the type of addressing used and the depth of the framebuffer.

Configuring the depth of framebuffer can be achieved by using the `setColourBits()` method which takes an integerial argument specifying the depth of the framebuffer.

The type of framebuffer can be configured by using the `setPixelType()` method which takes the arguments of `GLCapabilities.RGBA` for RGB framebuffers and `GLCapabilities.INDEXED` for indexed framebuffers.

5.9 How is integration with other Java applications handled?

The ability to draw onto a Java AWT or Swing component by using the AWT graphics context is only partially available with Magician not because of any restriction Magician enforces but simply because of the differences in the way that AWT and OpenGL render to windows.

The main problem is that the AWT graphics contexts and the OpenGL rendering contexts are completely separate entities that cannot share information. Therefore, to render both types of graphics onto a single drawing surface, one context must be flushed, then the other one typically in the order of OpenGL then Java.

For example, if you rendered some basic shapes to a `GLComponent` using a Java AWT graphics context then flushed the OpenGL framebuffer to that component, the results of the AWT drawing would be totally overwritten because the

OpenGL buffer swap or flushing operation operates on the entire window and does not preserve previous window contents.

However, it is possible to render OpenGL onto a drawing surface then scribble over the top of that with AWT using the `java.awt.Graphics` class since an AWT graphics context does not implicitly clear the context's framebuffer contents unless explicitly asked to do so. This should be done *after* OpenGL has finished rendering, however.

By using Magician's `GLEventListener` mechanism, it will be ensured that the underlying window system and OpenGL have finished writing to the drawing surface and framebuffer by the time you wish to use AWT graphics contexts in your applications. Therefore, there are no real synchronization issues to worry about.

5.10 How is remote rendering handled?

Remote rendering is essentially handled by underlying OpenGL window-system protocols such as GLX. This requires access to underlying window information and target X display information which a portable and abstracted API such as JavaGL cannot directly support.

Similarly, rendering to remote displays is not necessarily possible due to the fact that the OpenGL drawing surfaces will be embedded within a GUI and cannot be simply separated out.

However, it is possible, at least under X Windows, to cause the entire GUI to be displayed on a remote display. This would, however, be extremely slow in comparison to direct rendering through a local framebuffer or shared memory segments.

5.11 Is a new remote rendering protocol required?

A new remote rendering protocol would be required to support an abstracted system such as JavaGL in a portable manner unless some form of extension mechanism was added to JavaGL to implement support on a non-standard basis.

6 Multi-Threading

Multi-threading is a core part of Java rather than a nice feature tacked on *via* an external threading API as in C/C++. This brings to light several extremely tricky problems from an architectural and implementation viewpoint from the high-level "how do we do multi-threaded applications?" to the far more insidious "how do we do multi-threaded applications that work identically on all platforms and JVMs?".

Magician implements its main thread-handling code internally regarding the management of OpenGL resources across thread boundaries, but it also implements a higher-level thread strategy within the `GLComponent`, `JGLComponent` and `GLOffscreenBuffer` classes. This higher-level strategy deals with simple threading at an application level.

6.1 Are Java threads mapped $1:1$ or $n:1$ with OS threads?

This question cannot really be answered in a concrete manner. The way in which Java threads are mapped to OS threads is completely dependent on the JVM being used. For example, the Microsoft JVM maps Java threads to OS threads $1:1$ whereas non-native-thread versions of the Sun JVM map multiple Java threads onto a single native thread *via* Sun's "*Green Threads*" package.

This topic really doesn't have a direct bearing on JavaGL from an application development point of view as you should not have to care how the JVM performs thread mapping. From an implementor's point of view, it's critical to understand and straddle the differences in threading strategies with each JVM. It is in this area that Magician implements its thread boundary strategies within its native code segments as opposed to performing essentially JVM low-level thread operations within Java.

6.2 If not 1:1, how do we create thread-safe JavaGL applications?

This question is difficult to unravel in a simple way. Again, from an application development viewpoint, there should be no requirement to mess about with threads *unless you wish them in your application.* JavaGL should function quite happily on both single- and multiple-threaded applications.

Magician implements a few application level threading strategies that allows you to quickly harness some of the power available to them through multi-threaded rendering. The first strategy is that the `GLComponent`, `JGLComponent` and `GLOffscreenBuffer` classes all define an internal threaded architecture that allows them to be animated by simply invoking the `start()` method. This simply internally issues repeated calls to a registered listener's `display()` method which causes simple and repetitive animation to occur.

The second strategy revolves around the use of application-level threads written by developers. Magician implements several internal mechanisms that ensure that applications can drive rendering contexts safely from multiple threads in a portable manner.

Therefore, Magician is totally thread-safe on all platforms and OpenGL implementations.

7 Extensions

Extensions encompass functionality that has not been as yet voted into the core OpenGL or GLU functionality, functionality that is extremely platform- or machine-specific or functionality that is experimental in nature.

7.1 Is an extension mechanism required?

Yes, we feel that a mechanism to access extensions is required for JavaGL. Support for stereo rendering devices, for example, is generally implemented as a set of extensions rather than being present within core OpenGL.

Similarly, extensions such as multi-texturing are extremely important if JavaGL is to be useful in the arena of games or game editors. Furthermore, access to hardware-specific functionality is a desirable feature that is only available *via* extensions, *e.g.*, PBuffer support.

7.2 Into what class(es) or interface(s) are extensions added?

Magician has implemented a base class called `GLExtension` which supplies hooks for you to add your own extension methods into by subclassing. `GLExtension` handles the basic extension library loading functionality and link checking required to ensure that versioning is correct and that the desired extensions are present and available for use.

By doing so, we have separated extension handling from the standard OpenGL and GLU functionality that is guaranteed to be present[1].

You are required currently to write your own implementations of the extension functions, but we are gathering together implementations of as many extensions as possible for all supported JVMs and operating systems in order to provide a pre-built repository of extension code that your can simply download and use without additional effort.

Furthermore, some extensions require access to window-system information,

[1]Depending on OpenGL version.

such as the stereo rendering support on SGI machines. Magician is supplied with an external piece of software known as the *Extension Developer's Kit* or *EDK*. This gives you access to the internal window-system information of Magician GUI components such as `GLComponent`, `JGLComponent` and `GLOffscreenBuffer` in a thread-safe and portable way.

7.3 How are optional features handled?

Magician treats optional features in exactly the same way as extensions.

8 Context Management

This section discusses the question of how higher granularity access to OpenGL contexts and standard context operations is made available to you.

Magician implements a class called `GLContext` which encapsulates an OpenGL rendering context in an abstract and platform-independent way. `GLContext` objects are associated on a 1 : 1 basis with drawing surfaces such as lightweight and heavyweight components or off-screen buffers.

8.1 How are shared objects handled?

Magician handles the sharing of display lists and texture objects with an extremely straight-forward and completely abstracted way by automatically translating the different techniques found in the various window-systems.

The way in which we handle object sharing is best illustrated by an example. If you have two `GLComponents` of which you wish to share the display lists and texture objects, doing so is as simple as passing the first `GLComponent` to the constructor of the second. This can be written as

```
/** Create the first GLComponent */
GLComponent comp1 =
    GLComponentFactory.createGLComponent( WIDTH, HEIGHT );


/** Create the second GLComponent using the arenas of the first */
GLComponent comp2 =
    GLComponentFactory.createGLComponent( comp1, WIDTH, HEIGHT );

...
```

And that is all there is to it with Magician. Additional components can be added by passing either `comp1` or `comp2` in their constructor. Similarly, the display lists and texture objects of lightweight `JGLComponents` can also be intermixed with `GLComponents` since it is in the internal `GLContext` object that the sharing occurs.

Therefore, Magician implements an extremely simple and powerful mechanism for providing shared object functionality that operates independently of any platform, JVM or OpenGL differences.

9 Integration With Native OpenGL?

This section discusses some details as to how JavaGL integrates with the underlying native OpenGL implementation and how any mismatches in versioning or function linkage can be handled in a sensible way.

9.1 Are calls to JavaGL required to have a 1:1 mapping?

Generally, yes. Using the standard OpenGL functions should result in the corresponding native OpenGL function being called if only for performance reasons. In some cases, such as variant methods, some internal massaging of the data being passed into the function might be required, but this is really an implementation issue rather than a specification issue.

9.2 How are errors handled in JavaGL?

Magician offers two forms of error handling regarding inline OpenGL function execution failure.

`glGetError()` Magician fully support `glGetError()` for testing the error status after any OpenGL or GLU function execution. This operates identically to C/C++ and is implemented by you at an application level.

`ErrorGL / ErrorGLU` Magician ships with two error-checking pipelines called `ErrorGL` and `ErrorGLU` for OpenGL functions and GLU functions respectively. These pipelines will automatically test for errors after *each and every* OpenGL function's execution through those pipelines. If an error is detected, a Java exception of type `OpenGLException` will be thrown detailing exactly where and why the execution of the program failed.

This is an extremely powerful and simple way to debug your applications since you do not need to add any explicit error-handling code into your applications at all in order to debug them. You simply need to switch to an `ErrorGL` or `ErrorGLU` pipeline.

Another benefit to this method of error-checking is that you can simply enable or disable it on a statement-to-statement basis. This allows you to switch it off when high-performance is required, or if you know one particular section of code is prone to failure.

Therefore, Magician implements two powerful forms of error-checking that can be used by both existing programs using their own error-checking routines or new programs that wish to quickly and powerfully test for error conditions.

9.3 What is the behaviour when the underlying OpenGL does not support a feature in JavaGL?

Magician internally probes for function linkage when initially loaded allowing us to check whether or not a particular function is supported in the OpenGL implementation that has been loaded. This is necessary if Magician has been linked against OpenGL 1.1 but is being executed on a machine with only OpenGL 1.0 installed.

If an unsupported function is invoked, Magician throws an exception of type `GLUnsupportedFunctionException` which can be trapped by the application.

10 References

The following texts and software should be referenced to fully understand and appreciate the design of Magician.

Magician 1.1.0 Magician 1.1.0 is available for evaluation download from `http://www.arcana.co.uk/products/magician`

Magician Programmer's Guide This document describes the task of programming with Magician under Java in considerably more detail and provides a complete discussion on the Magician architecture and design. It also available from the URL shown above.

11 Caveats

Some of the features described within this document are not available for public release in Magician 1.1.0. These features include offscreen buffers and Swing components. These are currently in testing for release with Magician 1.2.0.

Contents