White Paper

www.hoops3d.com

Integrating HOOPS with JAVA

1 Introduction 1
2 JAVA Based GUIs and Native Code 2
2.1 JAVA NATIVE INTERFACE BINDINGS FOR NATIVE CODE 3
2.2 INTER-PROCESS SOCKET-BASED MESSAGE PASSING 4
3 The HOOPS/JAVA Integration 5
3.1 BINDING HOOPS 3D GRAPHICS SYSTEM TO JAVA SWING 5
3.2 JNI BINDINGS FOR THE HOOPS/3DGS API 6
3.3 INTEGRATING HOOPS/3DAF AND JAVA USING TWO PROCESSES ... 6
3.3.1 Binding HOOPS/3dGS to a JAVA Swing Canvas 7
3.3.2 Inter-Process Communication via Sockets 7
3.3.3 .. A Messaging Scheme for Connecting HOOPS/MVO to the JAVA Swing GUI 9
4 Summary 11

1 Introduction

This paper discusses the integration of the HOOPS 3D Graphics System (HOOPS/3dGS) and its associated Application Framework (HOOPS/3dAF) with the JAVA programming environment. Primarily this involves connecting the output from the HOOPS 3D Graphics System (HOOPS/3dGS) to a heavyweight canvas object in the Abstract Windowing Toolkit (AWT) upon which the JAVA Swing GUI toolkit is built.

With the HOOPS/JAVA integration application developers are now able to build cross-platform 3D graphics applications using JAVA Swing for the Graphical User Interface (GUI) and the HOOPS 3D Graphics System for the display and manipulation of 3D geometric information.

The paper starts with an overview of the two approaches available for integrating existing non-JAVA based software components, a.k.a. Native Code, with JAVA and then specifically discusses how each of the approaches have been used in the HOOPS/JAVA integration.

www.hoops3d.com

2 JAVA Based GUIs and Native Code

The JAVA programming language provides developers with a programming interface for creating applications that can be executed on any platform that supplies a compliant JAVA Virtual Machine. As such, cross platform application developers are looking to JAVA as a tool for minimizing the amount of platform-specific code required for building applications to be distributed on multiple platforms.

In an ideal world a developer's entire application would be written in JAVA and there would be a compliant JAVA Virtual Machine available on every computer hardware platform. In the reality of today's world (Nov. 1999), the JAVA specification, controlled by SUN Microsystems, is still evolving and the JAVA virtual machines supplied by the various hardware manufacturers are in various stages of compliance.

To further complicate the matter, many existing software applications rely on large code bases written in FORTRAN, C and C++ that are far too large to rewrite in JAVA. These code bases are often referred to as "Native Code". Despite these real-world constraints, JAVA can help the cross platform developer gain some efficiency, specifically in the area of the application's Graphical User Interface (GUI), by using JAVA Swing for the GUI and connecting this to their Native Code.

In the CAD/CAM/CAE software industry, applications rely on solid modelers, mesh generators, solvers and graphics sub-systems that may have taken over 300+ person-years of development effort—their Native Code. The effort required in rewriting these software components in JAVA would be far too cost prohibitive and may actually produce less computationally efficient versions of these software components. Thus, CAD/CAM/CAE software developers looking to JAVA as a means of reducing their cross-platform related implementation expenses must find a way to leverage their existing Native Code within the JAVA programming environment. This requires some communication path between a JAVA-based GUI and their Native Code.

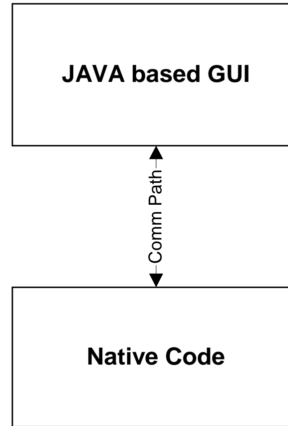

Diagram 1: JAVA based GUI and Native Code Require a Communication Path

There are two methods available for creating this communication path. The developer can either create JAVA Native Interface Bindings (JNI) for their Native Code, or they can use sockets to pass messages between the JAVA GUI and the Native Code. Each of these methods is described in detail below.

2.1 JAVA Native Interface Bindings for Native Code

JAVA Native Interface (JNI) bindings are wrappers around a C, or C++ library's application programming interface that enable the Native Code contained in the library to be called directly from a program written in JAVA. Thus, the JAVA GUI and the Native Code execute in the same process.

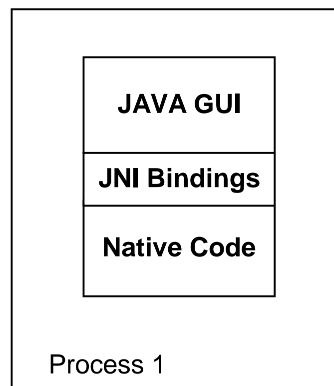

Diagram 2: JNI Bindings for Native Code

The main benefit with this approach is that it provides the programmer with a version of the Native Code's API that can be called directly from a JAVA program.

2.2 Inter-Process Socket-Based Message Passing

If a software component requires large amounts of data to be passed across its boundaries, it may better be to integrate it with a JAVA-based GUI by executing the GUI and the Native Code in separate processes and having them communicate by passing messages via sockets.

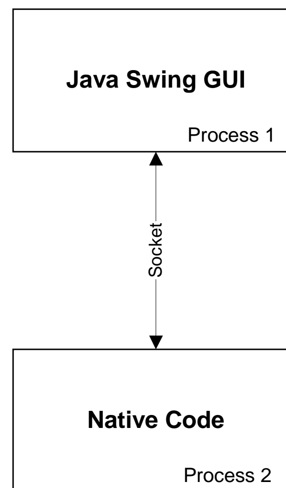

Diagram 3: JAVA Swing GUI and Native Code in Separate Processes

This approach avoids the need to create JNI bindings for the Native Code's API, but does require the developer to design and implement a pair of message passing objects, one for each side of the socket, and a message passing scheme.

The main benefit of this approach is to minimize the amount of data passed across the boundary between the application's GUI and the Native Code, i.e., it maximizes execution-time performance of the application.

www.hoops3d.com

3 The HOOPS/JAVA Integration

The HOOPS/JAVA Integration from Tech Soft America enables software developers to use the HOOPS 3D Graphics System or the HOOPS 3D Application Framework in combination with the JAVA Swing GUI toolkit to write cross platform applications with one code base. The integration was designed with the help of SUN Microsystems with whom Tech Soft America works closely to ensure that the HOOPS/3dGS optimally supports SUN hardware platforms running both the Solaris and Microsoft NT operating systems.

SUN advised using the JAVA AWT heavyweight canvas for connecting the HOOPS/3dGS to the JAVA Swing GUI toolkit. There are two approaches for accomplishing this:

- Build JAVA Native Interface (JNI) Bindings for your Native Code
- Run the GUI and Native Code in separate processes and pass messages back and forth via sockets

JNI Bindings for the HOOPS/3dGS API make it possible for the application developer to write programs in JAVA. The developer will need to design and implement a component for manipulating the HOOPS/3dGS library via its JNI bindings. For applications with a well-defined object layer above their graphics sub-system, or developers wishing to leverage the existing object layer in the HOOPS/3D Application Framework, the two-process approach should be employed.

The HOOPS/JAVA integration supports both approaches by providing JNI bindings for the HOOPS/3dGS as well as a reference implementation of the socket-based message passing approach used to integrate the HOOPS/3dAF with JAVA Swing.

3.1 Binding HOOPS 3D Graphics System to JAVA Swing

HOOPS/3dGS requires a windowing system window into which it can render its output. More specifically, when a HOOPS/3dGS driver segment is instanced it must be passed the resource handle for a window into which it can render along with an associated colormap. The resource handles for the JAVA Swing Canvas' Window ID and Colormap must be extracted and passed to HOOPS/3dGS when instancing a driver segment.

Unfortunately the Window ID and Colormap resource handles are not directly available from the JAVA Swing interface. To access these pieces of information one must turn to the interface layer just below JAVA Swing, the Abstract Window Toolkit (AWT) upon which JAVA Swing is built. Access to the information needed from the AWT toolkit is officially supported in JAVA Version 1.2.

www.hoops3d.com

Both the JNI Bindings for the HOOPS/3dGS API and the integration of the HOOPS/3dAF with JAVA Swing using socket-based messaging encapsulate the process of accessing this information and passing it into the HOOPS/3dGS when creating driver segment instances.

3.2 JNI Bindings for the HOOPS/3dGS API

The HOOPS/3dGS API is a C-language interface. Creating JNI bindings is a fairly straightforward process of writing a new set of routines that can be called from a JAVA program which accept data, reformat it as needed and transmit to or from the native code library. The main consideration in building JNI bindings for a C-based API is how to handle the C language's pointer construct in JAVA. JAVA has no pointers, only objects; thus data passed by reference in C must be buffered up in the JNI binding and passed over to the JAVA side as an entire object.

The JNI bindings for HOOPS exactly mirror the C-based API and as such, programming with this interface is exactly the same as programming with HOOPS in C, C++ or FORTRAN.

3.3 Integrating HOOPS/3dAF and JAVA using Two Processes

Application developers who either have a well-defined object layer above their graphics subsystem or wish to use the HOOPS/MVO class library as their object layer on top of the HOOPS/3dGS should use the socket-based message passing approach to connect their JAVA Swing GUI to their object layer. The alternative would be to create a set of JNI bindings for their object layer (or the HOOPS/MVO Class Library), which would take considerable effort and introduce significant execution time overhead. This section focuses on the integration of the HOOPS/3dAF with JAVA Swing; developers who wish to integrate their own object layer using a similar approach will find this section useful.

Integrating HOOPS/3dAF with JAVA Swing using a two process socket-based approach primarily consists of binding the HOOPS/3dGS to a JAVA Swing heavyweight canvas, creating a pair of communication objects that monitor the sockets and dispatch messages, and designing a set of inter-object messages for connecting the GUI with the HOOPS/MVO object layer.

3.3.1 Binding HOOPS/3dGS to a JAVA Swing Canvas

The HOOPS/JAVA integration's two-process implementation creates a pair of classes called JHCanvas, one in JAVA and one in C++, to manage the binding of HOOPS/3dGS to a JAVA Swing Canvas. The JAVA version derives from the JAVA Swing Canvas Class and, among other things, adds functionality for accessing the window handle resource using the AWT toolkit. This information is passed to the C++ JHCanvas object over the socket and used when creating an instance of a HOOPS 3dGS driver segment. This enables HOOPS/3dGS to render 3D graphics to the JAVA/Swing Canvas using any of the available HOOPS/3dGS output drivers, i.e., OpenGL or X11 on Solaris, and OpenGL or GDI on Windows NT.

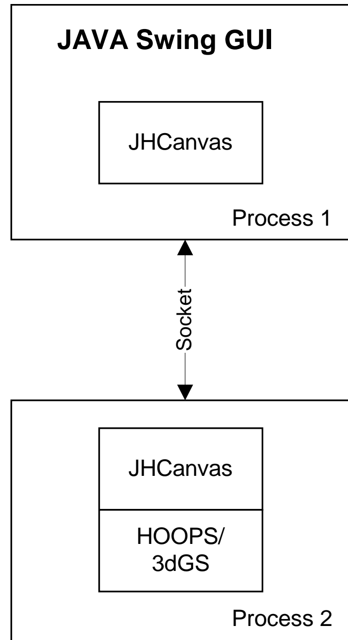

Diagram 4: Parallel JHCanvas Objects used to Connect HOOPS/3dGS to JAVA Swing Canvas

3.3.2 Inter-Process Communication via Sockets

Building an application involves more than binding HOOPS/3dGS to a JAVA Swing Canvas. The HOOPS/JAVA integration provides a reference implementation of a socket based messaging scheme. Application developers may use it as a reference template during implementation or may elect to extend the one provided.

www.hoops3d.com

Since HOOPS/3dGS and JAVA are in separate processes they must communicate via a socket-based communication pathway. The user generated GUI input events must be mapped to messages and sent to the Native Code; this includes direct input events like mouse motion, keyboard presses and window resizing and indirect events generated by direct interaction with other applications on the screen like window expose events.

HOOPS/3dGS resides on the side of the application receiving messages from the JAVA Swing based GUI. This side requires an object layer to receive the messages and turn them into meaningful operations on the 3D data using the API of the HOOPS/3dGS. The HOOPS Model-View-Operator (MVO) Class Library provides this functionality and the HOOPS/JAVA integration leverages HOOPS/MVO in defining its messaging implementation.

Implementing the inter-process communication requires a pair of messaging objects, one in each process, which monitor the socket connection and dispatch incoming messages to the appropriate objects in their process and collect outgoing messages from these objects and send them across the socket for dispatching.

The JAVA based GUI can be thought of as the “Client”, and the HOOPS based 3D graphics along with the other Native Code as the “Server”. The HOOPS/JAVA integration communication objects (classes) are thus named “JHClient” and “JHServer” respectively. JHClient resides in the JAVA Swing GUI process and is implemented as a JAVA class; JHServer resides in the Native Code process with HOOPS/3dGS and is implemented as a C++ class.

It should be noted that while the relationship of the JAVA-Swing GUI to the HOOPS/3dGS can be thought of as a client/server relationship, the two components must execute on the same physical machine.

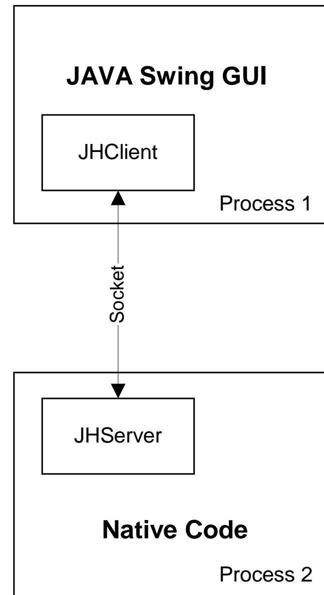

Diagram 5: JHClient and JHServer Objects Monitor the Socket and Dispatch Messages

3.3.3 A Messaging Scheme for Connecting HOOPS/MVO to the JAVA Swing GUI

The HOOPS/Java Integration's Reference Application includes a scheme for passing messages between GUI objects in one process and HOOPS/MVO objects in another process.

On the GUI-side the JAVA Swing Canvas must package up information associated with user generated events such as mouse movement, keyboard presses, and window resizing. Most applications will use other components in the GUI like menubars and icons and messages must be defined for passing across the socket when the user activates these components. These messages are sent from the GUI components to the JHClient object on the GUI side that sends them to the JHServer Object on the Native Code side. The Native Code based JHServer object then dispatches the messages to the appropriate HOOPS/MVO objects which in turn use the API of the HOOPS/3dGS to manipulate the 3D data.

www.hoops3d.com

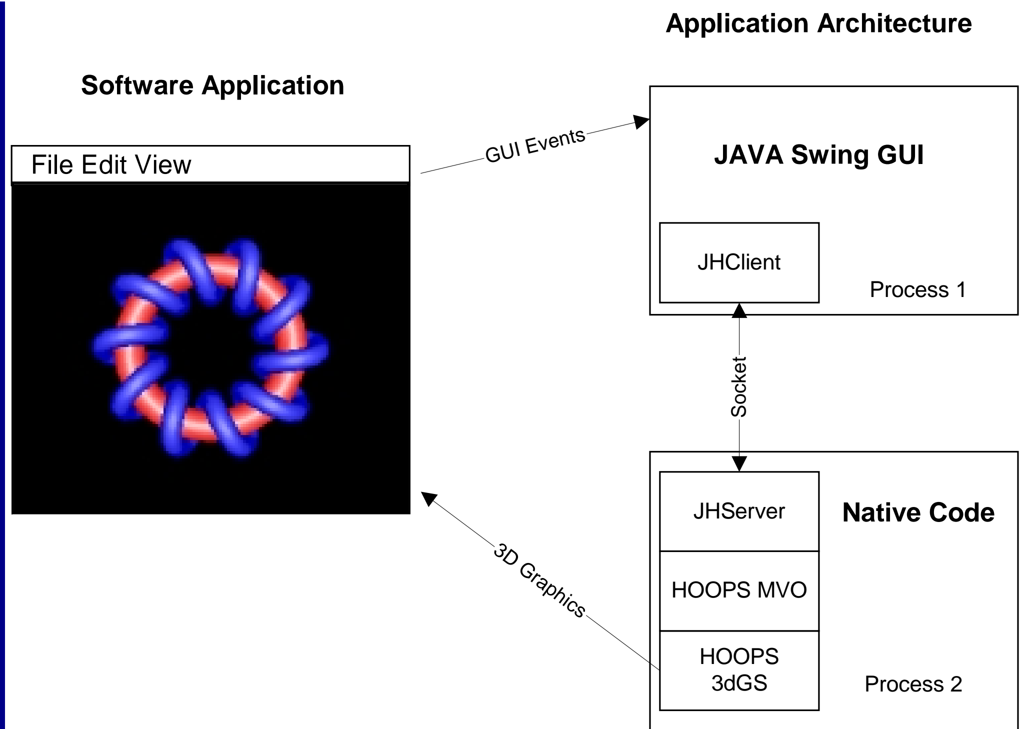

Diagram 6: HOOPS/3dAF and JAVA Swing Socket-based Integration

www.hoops3d.com

www.hoops3d.com

4 Summary

The HOOPS/JAVA integration provides software application developers with two methods for using the HOOPS/3dGS within their JAVA development efforts. Those who wish to write applications completely in JAVA should use the JAVA Native Interface bindings for the HOOPS 3D Graphics system. Those with existing application code or who wish to use the HOOPS/MVO Class Library in the HOOPS 3D Application Framework in conjunction with the HOOPS/3dGS should use the socket-based integration approach.

The HOOPS/JAVA Integration is currently under development and scheduled for general release with HOOPS 5.0 in mid-2000. As of November 1999, the socket-based approach has been implemented and is available in pre-release form. A pre-release version of the JNI bindings are scheduled for availability in Q1 2000

The HOOPS/JAVA integration relies on capabilities in the Abstract Windowing Toolkit exposed by SUN in version 1.2 of the JAVA specification. As of November 1999, the only implementations of the JAVA virtual machine that support the JAVA 1.2 specification are SUN's implementations for the Solaris and Windows NT operating systems.