Chapter 3 System Architecture

3.1. An Overview of DIVINE

DIVINE – **D**ebugging **I**nformation **VI**sualizatio**N** in 3D **E**nvironment is a debugging information visualization system that visualizes the debugging information received from HILCADT –**HI**gh-**L**evel **C**omputer-**A**ided **D**ebugging **T**ool. While debugging a program (called a "debuggee") with HILCADT, a user firstly loads the debuggee into HILCADT. Then, DIVINE connects to HILCADT via TCP/IP and executes remote debugging even DIVINE and HILCADT are both on local machine. After establishing the connection, DIVINE sends debugging commands to HILCADT, i.e., setting breakpoints and watching values of variables, and receives information from HILCADT. After receiving the debugging information, DIVINE visualizes the information and users can interact with the visualized elements in DIVINE in 2D or 3D user interface. While users interact with the visualized elements, each action may be translated into a sequence of debugging commands and sent to HILCADT without users knowing it.

3.2. System Architecture of DIVINE

Overall, DIVINE is a prototype tool under construction and planning. In this section, we describe the overall system architecture of DIVINE. The framework

presented in this thesis is an important component in the system. As the introduction in section 1.2, DIVINE connects to HILCADT and sends debugging commands for remote debugging via TCP. In other words, we will have a simple graphical debugger for loading source programs and provide a simple interface for functions like setting breakpoint and watching variables. The graphical interface has been implemented in HILCADT. Alternatively, users can choose to visualize some objects and variables in DIVINE. The user can debug in a 3D environment by wearing a HMD and data glove. Figure 3-1 illustrates the architecture of DIVINE and the areas marked in gray are the main contributions of the thesis.

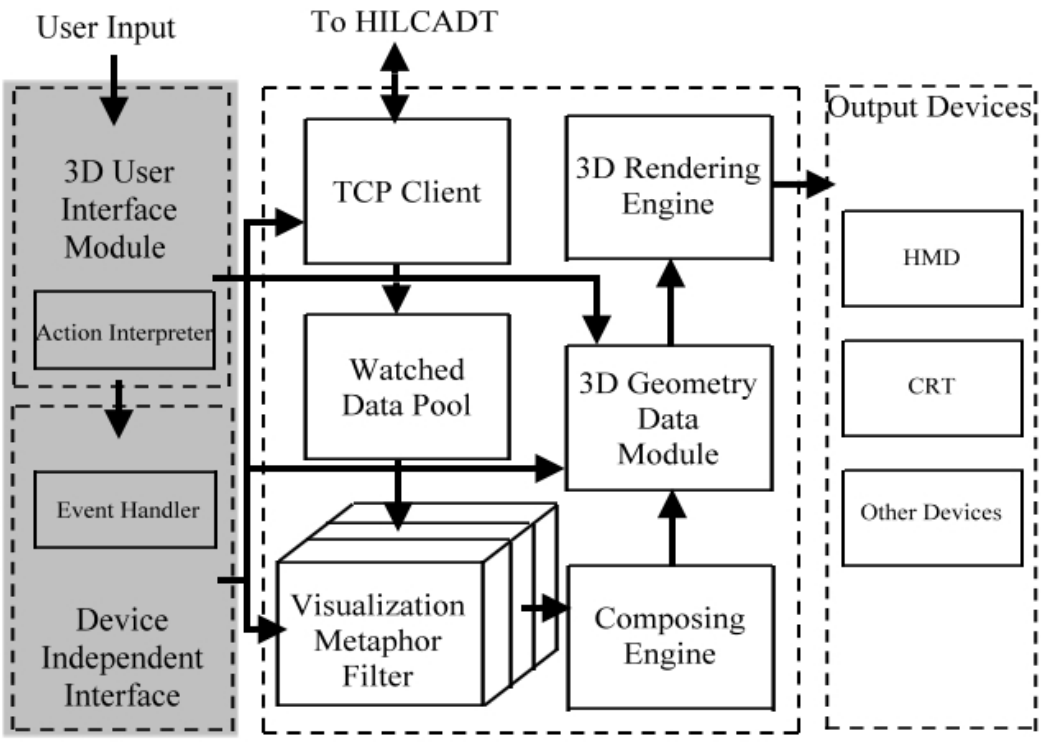

Figure 3-1: System architecture of DIVINE.

Referring to Figure 3-1, all objects or variables received from HILCADT are collected in a managed module – *watched data pool*. Every watched object or variable can associate with a valid *visualization metaphor*, and then the *composing engine* integrates all visualization metaphors of an object into one or many 3D geometries, which are rendered by a *3D rendering engine*. While composing individual metaphors into a composite metaphors (such as spatial layout for graph visualization), composing engine may apply another visualization metaphor. The rendering result will be sent to output devices such as HMD, CRT display, and etc. To overcome the limitation in 2D display, we choose a HMD with a 3-DOF tracker as our input and output device which makes users have more feeling of presence.

3.3. Device-independent Interface and 3D User Interface Module

The key component in the architecture is the *3D user interface module*. This module is responsible for transforming the input of various input devices into a set of *device-independent interface* which interacts with the other modules in DIVINE. So, users can interact with visualized metaphors via either VR devices or traditional desktop input devices like keyboards and mice but in the mean time, the programming interface of visualization metaphor does not change.

There is a set of unified *abstract action code* defined in the device-independent

interface. All the input of various devices will be interpreted into an abstract action code by a particular *action input interpreter* of every input device. The target visualized metaphor will react according to the received abstract action code from the *event handler* which is in charge of the delivery of abstract action code. Different input devices may have several ways to implement the same abstract action code. By using the device-independent interface, DIVINE has high flexibility to use various input device for interaction. More implementation details about abstract action code and input interpreter will be discussed in section 4.5.

As shown in Figure 3-1, the device-independent interface will interact with three modules. When a user doing an action such as picking by some device, the device-independent interface will change the state of the 3D geometry, i.e., highlight the picked object. If the action of picking involves expanding all children nodes of the object, the interface then interacts with the TCP client module to get more debugging information by sending debugging commands. Finally, a user can select the association between an object and a metaphor via the interface, which further changes the way of visualization.

3.3.1. The framework of Event Listeners and User-defined VM Object

Most modern programming languages provide object-oriented but

device-dependent design of event listeners. Taking Java programming for example, programmers may develop an application which can interact with the mouse by implementing the interfaces, like *MouseListener* and *MouseMotionListener*, provided in Java. To design the reactions while receiving events triggered by the mouse, programmers have to implement the virtual functions of the interface, such as *mouseClicked*, *mouseReleased*, and etc. These functions then will be invoked while the mouse doing relative actions on the elements which the listeners are registered to process events.

In thesis, the main contribution of our work is to design a framework in Figure 3-2. In the future of this project, some people are responsible for designing visualization metaphor (VM) to render watched objects in DIVINE., However, if VM programmers have to design a specified event listener for each kind of particular input device, which is extremely complex and difficult, especially in 3D environment. To master programming in 3D or 3D device is not an easy job.

In order to make VM programmers concentrate on the design of visualization elements with a device-independent framework, we add an abstract action layer between input devices and event listeners which can be implemented by VM objects. Figure 3-2 indicates the concept of this abstract action layer. With the abstract action layer, DIVINE provides a high-level device-independent and object-oriented

framework.

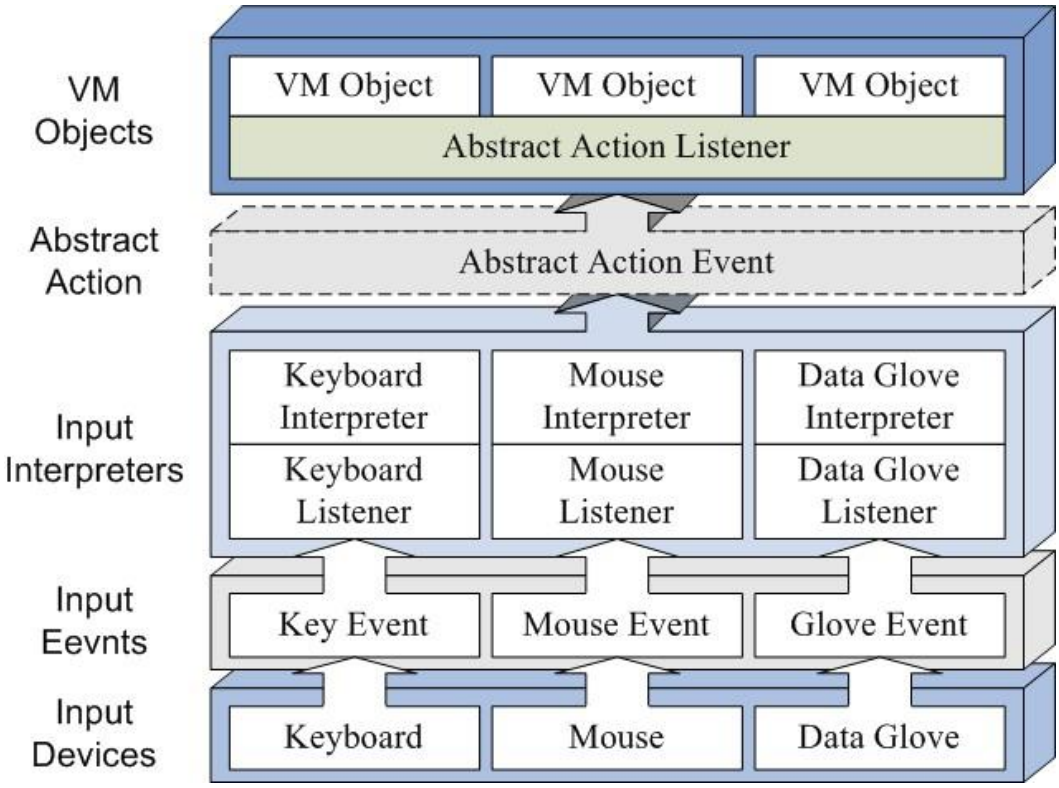

Figure 3-2: An abstract action layer between input devices and VM objects.

In the paragraph, we give an example that shows how to use the device-independent and object-oriented framework. Considering the class diagram in Figure 3-3, *DIVINEVMObject* is the base class which has defined the basic attributes and behaviors of a VM object in DIVINE. *AbstractActionListener* is an interface which must be implemented in user-defined VM object class to receive and process abstract actions. Abstract actions are interpreted from different input devices by input interpreters. *MyVMObject* which inherits from *DIVINEVMObject* class is the rendering class a VM programmer has to implement in DIVINE. In each *MyVMObject*, VM

programmers are responsible for implementing the event listener defined in *AbstractActionListener*. For example, in the *AbstractActionListener,* an action is called *actionPicked()*. This event listener is invoked by our module when a VM object is picked where the "picked" action can be made by clicking a mouse on the 3D VM object or by touching the 3D VM object using data glove.

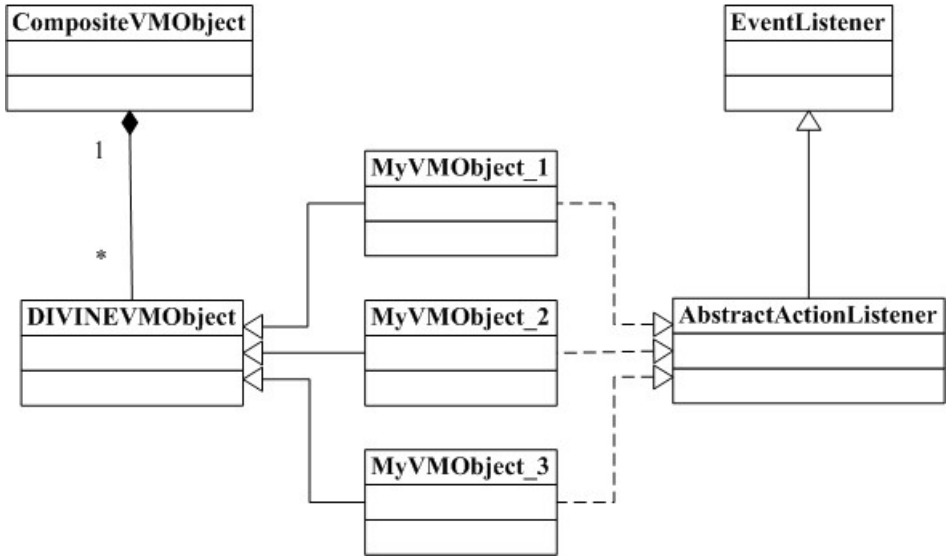

Figure 3-3: The class diagram of user defined VM object.

Figure 3-4 shows the class diagram of listener classes and the derived classes of interpreters. The listener classes are all abstract classes that derived from *EventListener* class which is an abstract class in OGRE. To realize the functions in listener classes, we have interpreter classes to inherit from these abstract classes. Thus, different input devices have particular interpreters which are responsible for the translation of device inputs to abstract actions. The interpreted abstract actions will be sent to the object which implements *AbstractActionListener*.

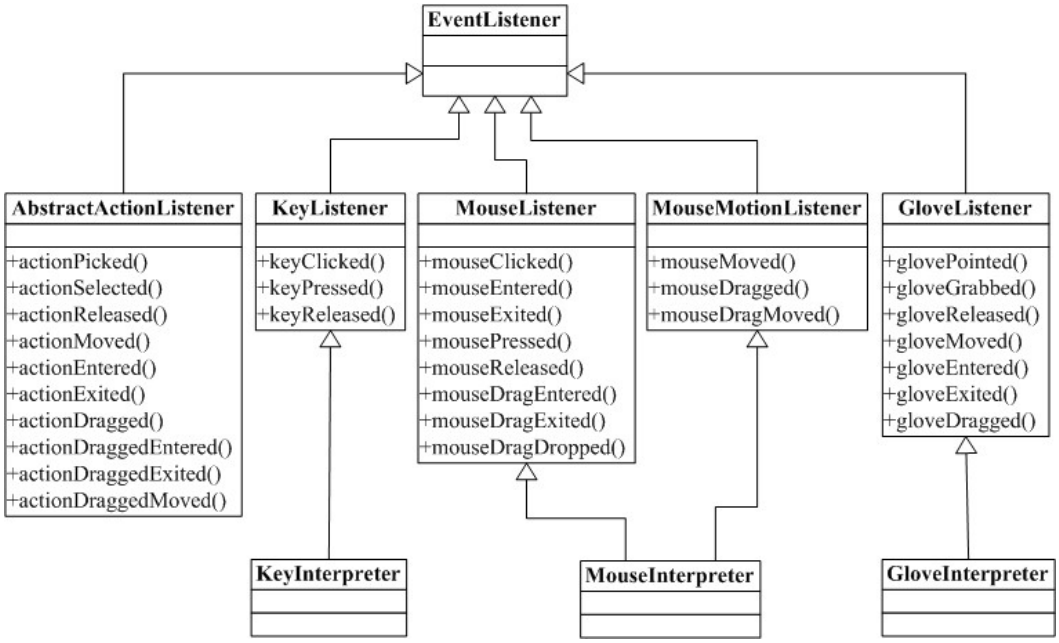

Figure 3-4: The class diagram of listener and interpreter classes.

In the future, we may use another input device to interact with the VM elements in DIVINE. With the framework, what we only have to do is creating another listener class and a particular interpreter class that implements the listener class. *AbstractActionListener* class in the left of the diagram will be implemented by user defined VM object classes as mentioned in previous paragraph.

3.3.2. An Example of Event Process

As demonstrated in Figure 3-5, the sequence diagram presents an example of picking a VM element by using the data glove. In the beginning, a user may do a pointing gesture at a VM element and then *DIVINEEventProcessor* is triggered to update and capture the current state of *OgreHand* which is our virtual hand model used in DIVINE. After *OgreHand* recognized the pointing gesture, the gesture is processed

and then a glove-pointed event is created. Next, *OgreHand* will check if the fingers collide with any VM objects. If there are collisions occurred between *OgreHand* and some VM elements, the glove-pointed event is then sent to our glove interpreter which translates the glove-pointed event into action-picked event, one of the abstract action events. Finally, the target VM element receives the action-picked event and invokes the code in event listener.

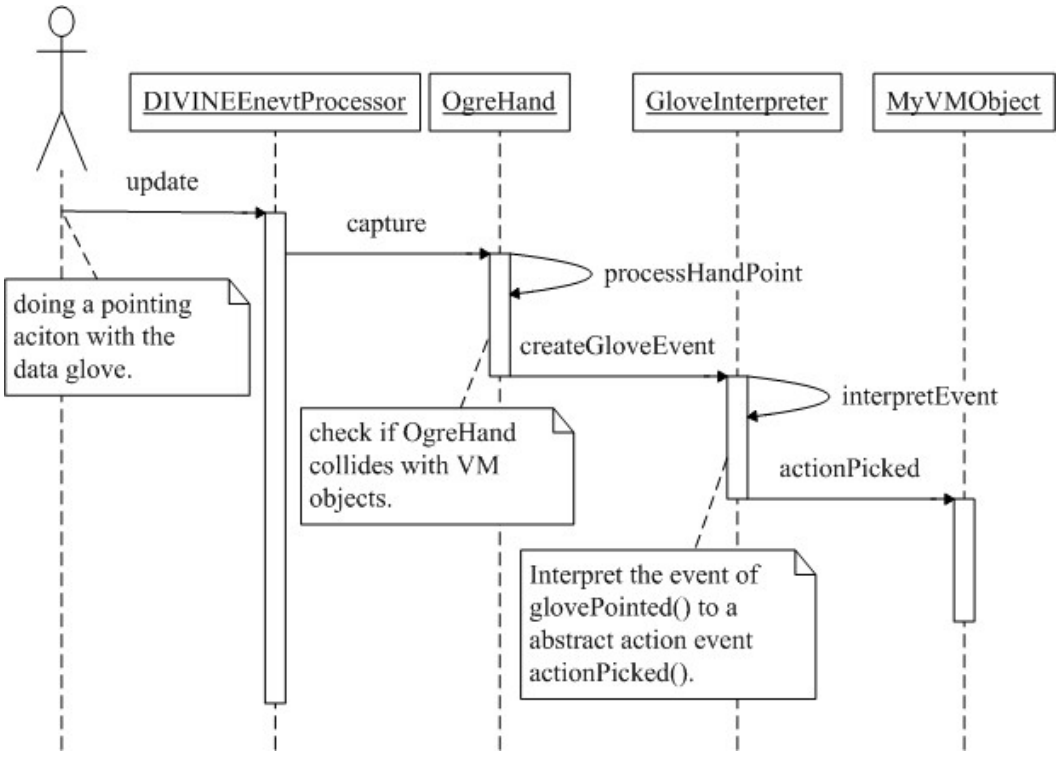

Figure 3-5: The sequence diagram of picking a VM.

With the object-oriented framework and device-independent interface, VM programmers can focus on the design of VM elements without deepening themselves into the details of programming with input devices.