TIDL: Mixed Presence Groupware Support for Legacy and Custom Applications

Peter Hutterer[1,2], Benjamin S. Close[1,2], Bruce H. Thomas[1,2]

[1]Wearable Computer Laboratory
School of Computer and Information Science,
University of South Australia
Mawson Lakes SA 5095

[2]National ICT Australia
Australia Technology Park
Bay 15 Locomotive Workshop
Eveleigh NSW 1430

{peter|cisbjc|thomas}@cs.unisa.edu.au

Abstract

In this paper, we present a framework to use an arbitrary number of mouse and keyboard input devices controlling Swing based Java applications. These devices can be distributed amongst any number of host computers on a network. We use this framework to provide independent input devices to a number of users on different host computers. These users can then work collaboratively on applications.

A major limitation for current real-time groupware is that contemporary graphic environments do not support more than one system cursor and keyboard. The Transparent Input Device Layer (TIDL) is a framework we have developed that provides an easy-to-use API for Java applications to gain support for multiple independent input devices. We have also created a wrapper application to retrofit legacy applications with support for multiple distributed input devices at runtime. This support can be injected without altering or recompiling the application's source code. TIDL allows multiple devices to work across window and application boundaries. Applications supporting multiple input devices can employ features such as simultaneous drag-and-drop and the entry of text in multiple textboxes. In addition, different applications running simultaneously can use multi-device support independently and at the same time. We present four applications that use TIDL to enable distributed groups to work collaboratively. One of these applications has been developed to make active use of TIDL, the other three applications are applications we have found on the web and gain support for multiple independent devices through the wrapper application.

Keywords: Mixed Presence Groupware, CSCW, Graphical User Interfaces.

1 Introduction

Research and development in Computer Supported Collaborative Work (CSCW) has made substantial progress in non-real-time groupware, resulting in successful applications such as Microsoft Exchange or Lotus Notes. These two applications allow users to work in an asynchronous fashion, but do not support real-time collaboration. Similarly, version control systems like CVS, Subversion, or Microsoft SourceSafe are widely used amongst developers, but are also limited to non-real-time collaboration. The domain of real-time groupware outside research institutions is limited to video conferencing. The widespread use Microsoft NetMeeting allows the sharing of the display of arbitrary applications but has a strict floor control policy and does not allow multiple users to collaborate simultaneously. The proper software support for real-time CSCW applications is still an open research question.

We aim to support real-time collaboration in legacy applications as well as in newly written ones. Using and developing collaborative applications require significantly more effort than single-user applications (Grudin 1988), and there is limited support from current user interface toolkits. We postulate that real-time collaboration is a natural way of working together. This is demonstrated by the fact that people still congregate in groups to perform collaborative work practices, without any explicit knowledge of collaborative activities. To state the obvious, people work together in collaborative groups because it is an effective problem solving activity.

Mixed Presence Groupware (MPG) connects both co-located and distributed collaborators and their disparate displays via a common shared virtual workspace. In this paper, we describe the Transparent Input Device Layer (TIDL), an implementation of a MPG framework for Java applications. TIDL supports an arbitrary number of users on a number of different host computers. The TIDL framework may be employed for new applications but also for legacy applications without recompilation or code alteration. A major research goal is to support the developers of MPG applications with only minimal additional effort.

We will describe previous research in the domain of real-time collaborative software in Section 2. Section 3 describes our implementation of a MPG framework, and Section 4 will detail some of the implementation challenges we experienced and our solutions. We describe four applications using the framework in Section 5, and finish with some concluding remarks in Section 6.

Copyright © 2006, Australian Computer Society, Inc. This paper appeared at the Seventh Australasian User Interface Conference (AUIC2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 50. Wayne Piekarski, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Related Work

Real-time groupware domains are divided by the number of users and host computers that can be simultaneously supported. Single Display Groupware (SDG) supports an arbitrary number of users on a single host computer; Distributed Groupware (DG) or distributed groupware connects single users on individual host computers over a network to share one desktop or application. Mixed Display Groupware (MPG) is a combination of SDG and DG and allows an arbitrary number of users on each host computer. Figure 1 depicts an example MPG session with a total of eight users on three host computers.

Stewart et al. (Stewart 1999, Stewart 1998) found that users collaborating on a single display prefer independent input devices instead of sharing one input device. Several projects have investigated supporting multiple input devices by different methods. One of the earliest SDG applications is the Multi-Device Multi-User Multi-Editor (MMM) (Bier 1991) that supports multiple users in either a drawing or a text editor. MMM features private areas for each user and differentiation between location independent global menus and user-specific context-sensitive menus. The PEBBLES project (Myers 1998) uses PDAs instead of the traditional input devices, keyboards and mice. The PDA provides each user with a small display area that can be used for private data. The DiamondTouch touch input device (Dietz 2001) can identify two touch-points simultaneously and delivers this information to the application.

The MID Java package (Hourcade 1999) employs calls to the Microsoft Windows 98 API to provide multiple input device support but is limited to only one host computer. MIDDesktop (Shoemaker 2001), which is based on MID, supports multiple Java applets running simultaneously on one desktop. A similar project based on C# is the SDGToolkit (Tse 2004), which has a focus on tabletop displays and allows users to rotate the cursors to match their position around the table. However, these projects support multiple devices on one host computer but not across different host computers. Moreover, while MID and SDGToolkit are toolkits to provide application developers support when developing real-time groupware, these toolkits do not support legacy applications.

The main technical challenges for a SDG application are retrieving data from input devices, limiting floor control and handling screen real estate. Current graphical environments do not support more than one system mouse or keyboard, and getting data from different devices has to be performed at the lower level of the operating system. Floor control becomes important with an increasing number of users sharing a single display. GUI elements often do not support multiple users simultaneously and strict floor control permits only one user to access a specific part of the GUI. Finally, screen real estate can become a limitation to the maximum number of users as applications may require user-dependent menus.

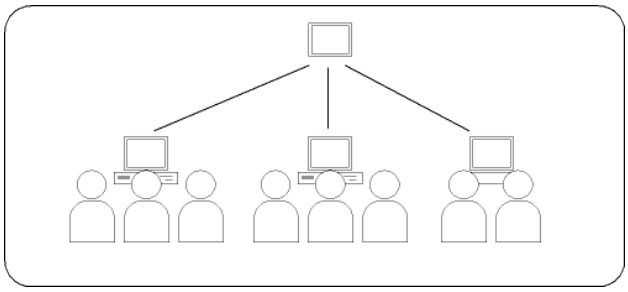

Figure 1. Layout of an MPG session. An arbitrary number of users share one common display across multiple host computers.

The two commonly used architectures for DG are a *centralised* and a *replicated* architecture. In the centralised architecture, the application executes on one machine but the GUI is distributed amongst all host computers. In the replicated architecture, the application is executed on each host computer and the user's input events are distributed amongst all host computers.

One of the earliest CSCW conferencing systems supporting DG is Rapport (Ahuja 1988) that allows legacy applications to be executed in a shared environment. Rapport provides virtual meeting rooms for the participants and support for telephone lines for audio transmission. MMConf (Crowley 1990) is a CSCW conferencing system employing a fully replicated approach for group conferences. GroupKIT (Roseman 1992) is a toolkit for the development of real-time groupware with shared displays. GroupKIT deals with synchronisation issues, registration and concurrency, thus reducing the development work required for application developers in building groupware applications.

DG faces three challenges SDG does not have to face: synchronisation, view sharing and telepresence. As network latencies cause race conditions, and events from different systems may arrive out of order, synchronisation becomes important to ensure consistent application states on each host computer. The centralised architecture may be employed to address the synchronisation but applications then need to implement methods to share the view amongst different host computers. The centralised architecture also suffers from fault tolerance concerns since the central server is the single point of failure. Finally, increasing telepresence is important for DG. There is no physical awareness of the remote peers, so the software has to provide means of replacing this missing awareness. Video conferencing or audio streams can improve awareness, and many DG toolkits and applications provide telepointers to indicate the remote users' actions.

MPG is a very young domain for CSCW. An early example is Tang et al.'s implementation of a MPG drawing editor (Tang 2004), MPGSketch, based on SDGToolkit and Collabrary (Boyle 2002). Multiple users on different sites can draw onto the editor's surface simultaneously. They found that the perception of remote groups is significantly different from the perception of co-located peers. In their work, they tried to increase telepresence by drawing digital arm shadows onto the application to show remote user's actions and gestures.

3 TIDL Framework

Our TIDL framework consists of three parts: the platform-specific Multiple Direct Device Interface (MDDI), the Transparent Input Device Layer (TIDL), and TIDLInject, the wrapper application to inject multi-device support into legacy applications. MDDI reads data directly from the input devices using low-level OS interfaces and passes it on to the TIDL abstraction. TIDL then distributes those events across host computers to be injected into the application. In this section, we will describe both MDDI and TIDL in detail and explain how TIDL supports both legacy applications via TIDLInject and applications developed directly with the TIDL Framework.

3.1 MDDI

Current graphics environments only support one system cursor. Modern operating systems generally allow multiple pointing devices, but when multiple pointing devices are connected to a host computer, the data from the different pointing devices is merged into a single cursor. Current Java implementations do not provide support to query each connected device independently or to query events for their originating device.

MDDI is a library to query the operating system for device data and to pass this information on to an application or another library. MDDI currently runs on Microsoft Windows XP and Linux, employing the operating systems' interfaces to query all connected devices for data. The data is then wrapped into Java objects using JNI and then passed on to the TIDL abstraction. MDDI uses the Raw Input API[1] via JNI under Windows XP and the virtual device files in the */dev*[2] directory under Linux.

The Windows XP Raw Input API supplies an application with events including a handle to the input device generating the specific event. From this handle a unique device ID is created which is used in the object passed to the Java implementation of MDDI.

In the Linux implementation of MDDI, we obtain a list of all connected devices as maintained by the Linux kernel, from the */dev* directory. A mouse is represented as a file in the notion of */dev/input/mouseN* where N is the number of the device. To gain access to the keyboards, the event module has to be loaded by the kernel. Similarly to the mouse, a keyboard is represented as */dev/input/eventN*, where N is again the number of the device. However, with the event module loaded, mice are not only represented as a mouse, but also as an event device. To find out which event device represents a mouse and which device a keyboard, MDDI parses the */proc/bus/input/devices* file. This file lists all connected devices and the associated file handlers.

The mouse device files in Linux provide raw data streams in the native mouse protocol that have to be parsed to convert them into mouse events. Because of security restrictions, accessing the device files is only allowed as root. The PS/2 protocol requires a mouse driver to write data to the mouse to reset the device and to query whether it is plain PS/2 or extended PS/2 with an extra data byte representing the mouse wheel motion. To avoid setting the files globally writeable or executing the Java application as root, we decided to use a Python script running as root to read the files, convert the bytes into a string representation of the event and then provide the data on a TCP socket. The MDDI Java implementation then accesses this socket to parse these events into Java events.

Both interfaces allow MDDI to assign a device ID to each event. MDDI encapsulates the data into Java objects and these objects can be used for any application that needs to query data from multiple devices. However, mouse events represent relative coordinates, and not the absolute coordinates commonly used in GUI APIs.

3.2 TIDL

TIDL receives events from MDDI via a listener interface and transmits them across a TCP/IP network to all connected host computers. TIDL employs a combination of a replicated and a centralised architecture, running the application on each host computer and distributing only input events to each node in the TIDL setup (Figure 2). If an event occurs (dotted line), TIDL forwards this event to the central server (dashed line), which then redistributes this event to every application instance (solid lines). We chose this over a pure centralised architecture because TIDL is designed to work with any application even if the MPG functionality is injected at runtime. TIDL does not have knowledge of the internals of an application and a centralised architecture needs either active support from the application or from the underlying graphical environment to distribute the GUI. A centralised approach is preferable over a replicated approach if there is a high bandwidth network because it is easier to maintain a consistent state on all nodes. In a replicated approach, where applications are executed on each site, race conditions occur due to network latencies, leading to inconsistent behaviour and difficulties synchronising applications once they are in different states. To avoid these inconsistencies, TIDL uses a centralised event distribution model. Although the application is executed on each host computer, the events are sent to and

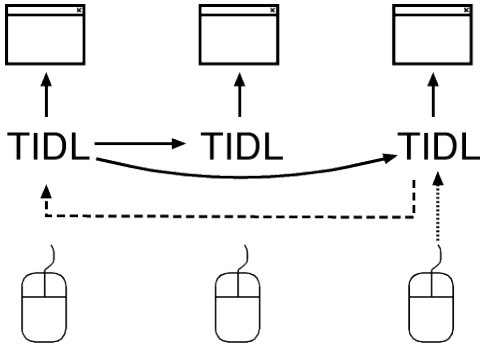

Figure 2. TIDL event distribution.

[1] http://msdn.microsoft.com

[2] http://www.pathname.com/fhs/

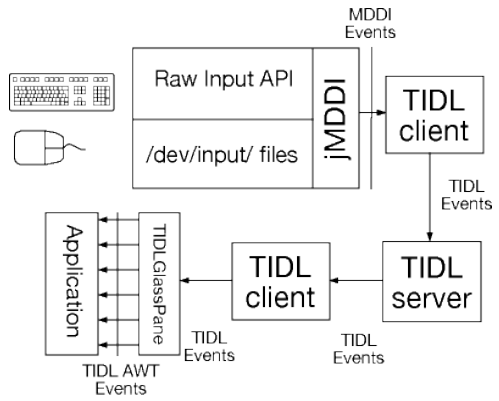

Figure 3. Event flow in the MDDI and TIDL system.

redistributed from a central server, thus ensuring the order of events is identical on each host computer. Previous implementations of similar toolkits treated local devices differently to remote devices. As every event is received from and transmitted to the central server, all sites are truly equal. There is no notion of a local device in TIDL. For an application, it is transparent which devices are connected to the local machine and which devices are remote.

The central element of TIDL is the TIDLGlassPane, a subclass of Java's GlassPane. The GlassPane is a Java concept allowing a transparent layer on top of an application. In Java Swing, a GlassPane can be used by any *javax.swing.RootPaneContainer*. A GlassPane is first to receive events generated by the Java AWT event system and can thus be used to intercept all AWT events and dispose of the events or replace them with custom events. The TIDLGlassPane receives the events from the TIDL subsystem and converts them into subclasses of AWT Events. These events are then passed on to the application, see Figure 3. The TIDLGlassPane draws the user's mouse cursors in distinct colours and maintains each user's colour across applications.

3.3 The TIDL API

TIDL has a strong focus on enabling multi-user support in legacy applications. However, an application can be developed using the TIDL API to gain additional collaboration support including simultaneous drag-and-drop, device level processing, and fine-grained annotation support. Applications may use the TIDLGlassPane in two ways: they can instantiate the TIDLGlassPane and assign this object to the application window or use the TIDLInject wrapper application to do so. The preferable way is to not let the application instantiate the TIDLGlassPane by itself. Instead, all applications should be started up through the TIDLInject wrapper application, as this allows legacy and newly created applications to be run simultaneously. Applications using the TIDL API to gain additional collaboration support will still gain this functionality with the use of the TIDLInject wrapper application.

The API we have developed is designed to enrich Java's own API with additional information. Because the API is an extension to the AWT class library, the complexity of learning to develop applications with TIDL is greatly reduced for experienced AWT and Swing developers. The AWT Event API provides developers with methods that are called when events occur on the GUI. A mouse event contains the coordinates of the mouse pointer, the component which is about to receive the event, button states and other information that may be used by the application. Similarly, a keyboard event contains the scancode and the character of the key, and whether it was a key press or release event. TIDL extends those events and each event passed on to the application contains a TIDLDeviceId instance that contains two properties, the host computer ID and the number of the device on the specific host computer. An application may use this information to monitor the users' interactions with input devices and host computers.

TIDL assigns all keyboards and mice to TIDLUser objects, where each TIDLUser object defines one mouse and keyboard for a user. From within the application, this object can be used to redirect a certain user's input event or display additional information (such as cursor colour). This TIDLUser object may be employed to support floor control policies on the application's components. Moreover, a user will always have the same cursor colour in any application unless the devices are physically disconnected from and reconnected to a different host computer. Disconnecting and reconnecting from a host computer causes the devices to appear as new devices.

The TIDL framework can be extended with *pluggable modules*. A pluggable module is a class that shares the graphic context with the TIDLGlassPane, thus everything a module draws on this context appears to be on top of the application. Modules receive events before the application does, allowing the following: 1) intercept the events and then discard them or 2) extract information from the event (i.e. which component the event is sent to) and modify the application using this information. However, components can be registered to be not affected by a specific module. They are therefore called a *veto component*.

One example for a module is the annotation layer. The annotation layer is a transparent layer atop the application and can work without the knowledge of the application. The users can use this layer to annotate in distinct colours without affecting the application. However, an application can purposely switch the annotation layer on and off as required. Switching the annotation on using a GUI button component introduces problems when a GUI element must be used to switch the annotation layer off again. As the annotation layer covers the whole application window, the button would not receive events anymore. Instead, the users would annotate on the button instead of switching the layer off. To control the annotation layer when the layer is activated, a GUI element has to be registered as veto component to be able.

Managing floor control in groupware applications is a complex challenge and each application has different requirements. TIDL's floor control module can be activated with a keyboard shortcut or through the TIDL API and restricts the application to only one user at a time. This restriction is useful for legacy applications that

rely heavily on drag-and-drop. When the floor restriction is active, users can still use the annotation layer. Applications using the TIDL API can extract the user information and implement a floor control policy that matches the application's requirements more closely.

3.4 Supporting Legacy Applications

We have created a wrapper application, TIDLInject, to insert support for multiple independent input devices into an application at runtime. The application needs no knowledge of the TIDL library at creation time; the TIDLInject wrapper application provides the collaboration support without the need for code modification or recompilation. TIDLInject has two main features: 1) to inject the TIDLGlassPane into an application at runtime and 2) to merge the GUI's of all open applications to be within a single frame. This section gives an overview of how we inject TIDL's functionality into pre-existing Swing applications.

TIDLInject supports Swing applications that use the Java AWT event queue. If the application is started up through TIDLInject, TIDLInject displays a window with a button to *grab* the application. Once the application has started up, the user then presses the *grab* button to direct TIDLInject to grab application's GUI. TIDLInject then queries the Java Virtual Machine for all open Frames by employing the *java.awt.Frame.getFrames()* method. This method returns an array with all open frames. If an application only opens a single window, TIDL can then insert the TIDLGlassPane into this JFrame. If the application opens up more than one window or if multiple applications are started up through TIDLInject, TIDLInject creates a desktop-sized JFrame and one JInternalFrame for each application window returned by the *getFrames()* method. Swing applications add components to ContentPanes, and TIDL obtains these content panes for each window of the application and/or for each application and assigns them to the respective JInternalFrames. This effectively copies the GUI of the application into an internal window. TIDL can then use the TIDLGlassPane on the desktop-wide JFrame, thus enabling support for multiple independent input devices across multiple application windows and even across different applications.

As mentioned before, the TIDLGlassPane converts TIDL events into TIDLAWTEvents. These events are subclassed from standard AWT events but are enriched with a unique device ID containing a device number and the originating host computer. Because the events are subclassed from the AWT events, the TIDLGlassPane can pass on TIDLAWTEvents to applications that do not support multiple devices actively. The application treats TIDLAWTEvents as standard AWTEvents and processes them unaware of the extra data. The TIDLGlassPane does not have knowledge of the application and acts identically regardless of the underlying application.

The functionality TIDLInject gives to any application without the application's active support is: 1) uniquely coloured mouse cursors for each connected mouse, 2) a user-dependent annotation layer, 3) simple floor control, and 4) independent keyboard foci for each connected keyboard and thus the ability to send keyboard events to different components (i.e. the ability to enter into different textfields) simultaneously.

3.5 Support for Latecomers and Drop-Outs

An important element for real-time groupware is support for latecomers and users who leave the session early. TIDL has latecomer support for users operating on host computers that are already connected. Users can connect new mice and keyboards at runtime and immediately receive their own cursor. However, this is limited to host computers already connected to the central server. While it is possible to connect host computers at a later time of the session, it is not recommended, as the replicated approach does not guarantee application consistency. TIDL does not have knowledge about an application's internals, and does not stop the user from connecting.

TIDL at present does not have specific support for drop-outs, if users leave the session their cursors simply cease to move any more. No notification is sent to the other users that this user dropped out. If a host computer disconnects, all cursors originating from this host computer cease to operate. However, because of the central server approach in our event distribution system, the central server must not drop out.

4 Implementation Challenges

During the development process, we faced several challenges and limitations. The most notable challenges were to support drag-and-drop for multiple devices and to reduce the delays between the events and their effects on the GUI. This section gives an overview about those challenges and TIDL's limitations.

4.1 Supporting Drag-And-Drop

Java has two APIs to support drag-and-drop in applications. The AWT event model features a *mouseDragged()* method in the *MouseMotionListener* interface in the *java.awt.event* package. This method is invoked when the mouse is moved while one or more buttons are pressed. In connection with the *java.awt.event.MouseListener* interface, which is used to notify an application of a mouse button release, this can be used for drag-and-drop in an application. TIDL adds the device ID to each *java.awt.event.MouseEvent* and applications can use this information to support simultaneous drag-and-drop by multiple users.

Java also supports a more sophisticated drag-and-drop API with the *java.awt.dnd* package. An application does not need to handle each mouse event but instead only designates a source for dragging objects (*java.awt.dnd.DragSource*) and a target for dropping objects (*java.awt.dnd.DropTarget*). The JVM then handles the drag-and-drop process without affecting the application using a *java.awt.dnd.DragGestureRecognizer* and a *java.awt.dnd.DragGestureListener*. The former initiates a gesture if the mouse is moved more than a certain threshold and at least one of the buttons is pressed. The latter is then notified of this gesture.

However, supporting the *java.awt.dnd* package in TIDL is problematic. The drag-and-drop process does not use the Java event queue and it is not possible to add the device ID. Moreover, while it is possible to query a component if it is registered as a mouse listener, the Java API does not provide methods to query when a DragGestureRecognizer is created or whether a component has a DragGestureListener associated with it. As our focus is on legacy applications, we cannot expect applications to use TIDL's API to initiate the drag-and-drop process. Instead, we need to replace the DragSource class with our own one. To support drag-and-drop in TIDL we need to intercept the instantiation of a DragGestureRecognizer.

Java supplies the "bootclasspath" commandline switch to replace system classes with user-defined classes. On startup, TIDL extracts the DragSource class from the standard libraries and modifies its package definition to be in the *real.java.awt.dnd* package. TIDL's own DragSource call is then inserted into the *java.awt.dnd* package and acts as a proxy to the original *java.awt.dnd.DragSource* class which is now in a different package. When an application creates a DragSource, TIDL's replacement DragSource class is invoked. For each method called on the replacement DragSource, the real method on the original class is invoked, with the return value passed back to the application. This allows TIDL to replace the standard DragGestureRecognizer with a custom written TIDLMouseDragGestureRecognizer. The latter generates events if the mouse is pressed; however, it enriches the event with the device ID that caused the drag-and-drop process.

Sun's Java license prohibits rewriting system classes and generating classes residing in any package with a package-name starting with "java". We obtained a Java Research License, which is necessary to implement and use the described method.

4.2 Speed Improvements

In our first implementations, we experienced low response times in the user interfaces. Several reasons for this could be identified. We were using the JADE agent framework[3] to transmit the mouse events to remote host computers. JADE is an agent-based framework, and agents were operating on the different host computers. Each of these agents has several different behaviours, where a behaviour receives and/or sends messages and processes them accordingly. JADE encodes messages in a custom format containing meta-information about the originating agent, and transmits them as a string representation to the remote agents. This encoding and the general overhead of JADE proved to be too time-consuming for real-time events such as mouse events or key events. We switched to a simpler network handling technique by utilising Java's standard TCP libraries and a protocol where the events are sent as strings of the length of only a few bytes. This new technique resulted in faster response time. JADE did prove to be a useful prototyping tool for the early versions of TIDL however.

A second performance problem was a severe delay caused by repaints of the GUI. Initially, every mouse movement caused a repaint on the TIDLGlassPane where the cursors where visualised. However, a repaint on a GlassPane causes a repaint on the underlying components. If multiple mice are moved simultaneously this can lead to hundreds of repaints per seconds. Usually a repaint is a costly operation and excessive use of this method should be avoided. The TIDLGlassPane now limits repainting to a maximum of 30 frames per second to reduce the number of repaints. This limitation contributed to a much faster response time.

A final performance improvement was to limit the area of the repaint. Instead of just calling the repaint() method on the TIDLGlassPane, the rectangle surrounding the cursor's old and new position is given as an argument. However, this gives only small speed benefits as Java queues repaint requests and then executes the actual repaint on the smallest rectangle including all areas given in all requests. This can be an issue if one repaint requests the top left corner of a window and another one the bottom right corner. If those requests are queued and accumulated, Java repaints the whole window. This queue forces TIDL to perform poorly on applications with very complex repaints. However, response times during the use of ViSOR (Takatsuka 2005) (a complex full-screen visualisation application) displaying thousands of data points were less than half a second.

4.3 Limitations

There are a number of limitations to TIDL. The concept of the GlassPane is limited to one GlassPane per JFrame, effectively denying the application to use a GlassPane for its own. Although we demonstrated a method of using the GlassPane across multiple frames, this method does not work if the application creates new frames at runtime (i.e. if the application pops up a dialog box). Similarly, if the application disposes the main frame containing the TIDLGlassPane, it loses support for multiple independent input devices. However, this limitation may be mitigated by continuously polling the JVM for open frames and resembling newly created frames in the desktop-wide JFrame.

Low-level mouse events can cause different reactions if the host computers have mixed setups. To safely support different resolutions, TIDL would need semantic knowledge about the underlying application. However, TIDL works safely in mixed setups if applications do not depend on the screen resolution to set window dimensions.

As the Java Swing graphical environment does not support multiple cursors, all connected mice contribute to the system cursor's motions. If all users move their respective mice, the system cursor's movements are unpredictable. This results in the cursor being likely to move outside of the application window. If one of the users then performs a mouse click, the application may lose its focus and all consecutive events would then be

[3] http://jade.tilab.com

sent to a different application or to the windowing environment. To avoid this, we lock the system cursor in a fixed position inside the application window.

Java does not use the AWT event queue for ActionEvents. If a user performs a button click or an action which causes an ActionEvent to be sent to the component, the TIDLGlassPane cannot intercept and discard this event. If the TIDLGlassPane were to then create its own ActionEvent (with the additional device ID of the user who performed the click), the component would receive two ActionEvents, the original one and the TIDL ActionEvent. Due to this, if an application needs multi-user support on buttons, it has to implement the *java.awt.event.MouseListener* methods.

5 Applications

We have employed TIDL for a number of applications. First, we will describe three applications, JavaChess, AllLights and JTans, that gain multi-user support at runtime without recompilation. Then we will demonstrate MPGCoast, a multi-user enhancement to a military scheduling application and explain the features MPGCoast gains by actively supporting the TIDL API. Using the TIDLInject wrapper application, all applications can be executed simultaneously and work independently of each other.

5.1 JavaChess

JavaChess is a freely available Java Swing application[4]. It is a chess game fully implemented in Java and features a mode where the user can play versus a computer player. By starting JavaChess through the TIDLInject wrapper application, we gain multiplayer support where multiple players can play together in a combined effort against the computer. Chess is ideally suited for collaborative purposes, as it allows users to combine their strategies to besiege the computer player. TIDL's annotation layer allows for the discussion of strategies atop the application.

JavaChess can be started up through TIDLInject without modifications. However, it makes extensive use of dialog boxes to inform users about invalid moves. For our use of JavaChess, we altered one line of code to display an error on the console instead of displaying the dialog box.

5.2 AllLights

AllLights[5] is a puzzle game that displays a 5x5 matrix of squares, representing lights which can be switched on and off. Each light affects its immediate neighbours and changes their states. The goal is to switch all lights on.

Using TIDL to enrich AllLights with support for multiple input devices allows an arbitrary number of users to solve the puzzles. AllLights only uses mouse moves and mouse click events with no events requiring tracking the state of the mouse (as drag-and-drop would). Because of this, playing AllLights with an arbitrary number of users has a similar feeling as playing it in single-user mode.

5.3 JTans

JTans[6] is a free Java implementation of the popular Tangram game. The Tangram game requires players to rebuild a given figure with a set of geometric shapes. JTans can be retrofitted with TIDL support at runtime and supports an arbitrary number of users. However, because JTans makes heavy use of drag-and-drop it shows the limitations of injecting TIDL into legacy application. Although it receives events tagged with the device ID, it treats them as coming from only one device. It is therefore not possible to drag or rotate multiple pieces simultaneously. However, with a small code alteration to support TIDL, it would be possible to have any number of users moving pieces at one time.

5.4 MPGCoast

MPGCoast is an GUI extension to the Australian Defence Science and Technology Organisation's COAST application (Zhang 2004). COAST provides military commanders with a tabular based application to determine a course of action to achieve a specified military goal. Although the development of these courses of actions is done in collaborative group, the current COAST implementation does not support multiple users simultaneously. The users enter tasks in text-based GUIs; the GUI itself makes heavy use of pop-up windows. These pop-up windows occlude other parts of the GUI and impede real-time collaboration.

We have built MPGCoast as a GUI extension to represent the tasks as a directed graph view. The users can add and remove tasks; a task's dependencies are shown as links to other nodes of the graph, see Figure 4. As with JavaChess, one of the main benefits for users is the annotation layer, which is integrated into the GUI using the previously mentioned veto-components. Users may switch to a text-based representation of the tasks at any point in time.

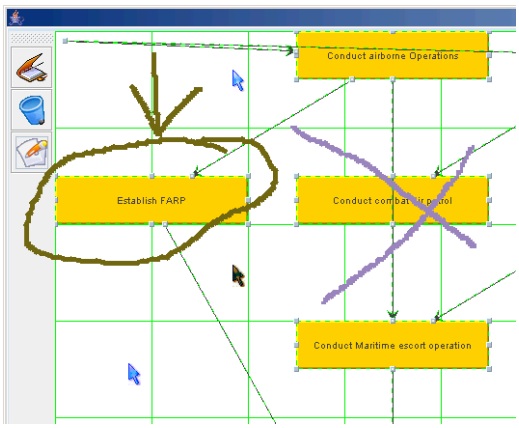

Figure 4. MPGCoast uses the TIDL API to support multiple input devices.

[4] http://www.java-chess.de

[5] http://www.ability.org.uk/alllights.html

[6] http://jtans.sourceforge.net

6 Conclusion

In this paper we presented TIDL, a toolkit to use multiple independent input devices in Swing based Java applications. The applications use the AWT event API but receive events that are enriched with a device ID to identify the originating user and host computer. This information can then be employed to enable features such as simultaneous drag-and-drop. The input devices can be connected to any host computer in the network, with each host computer supporting an arbitrary number of devices. This domain is known as Mixed Presence Groupware and allows distributed groups to work collaboratively on a shared application.

Our implementation is an improvement over previous toolkits as it is possible to use multiple input devices on legacy applications and not only with applications developed with the API. We described a wrapper application, TIDLInject, which can modify applications at runtime to support a large subset of the multi-user features TIDL provides. This wrapper application also allows the use of multiple input devices across application windows and different applications simultaneously. An application does not have to know about TIDL at any time, it can be retrofitted with this support without code alteration or the need for recompilation.

7 Acknowledgements

We give our thanks to the ViCAT team, mainly Peter Eades, Julien Epps, Masahiro Takatsuka and Mike Wu for their feedback during the development. Additional thanks go to Wolfgang Hochleitner, Aaron Stafford, Stewart Itzstein and Wayne Piekarski for improving this paper. Also, we want to thank the DSTO for providing us access to the COAST application. Finally, we thank Sun for providing a Java Research License.

8 References

Ahuja, S. R., Ensor, J. R., and Horn, D. N. (1988): *The rapport multimedia conferencing system.* SIGOIS Bull. **9**(2): 1-8.

Bier, E. A. and Freeman, S. (1991): MMM: a user interface architecture for shared editors on a single screen. In *Proc. of the 4th annual ACM symposium on User interface software and technology,* 79-86, Hilton Head, South Carolina, United States, ACM Press.

Boyle, M. and Greenberg, S. (2002): *GroupLab Collabrary: A toolkit for multimedia groupware.* J. Patterson (Ed.) ACM CSCW Workshop on Networking Services for Groupware.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R. (1990): MMConf: an infrastructure for building shared multimedia applications. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work,* 329-342, Los Angeles, California, United States, ACM Press.

Dietz, P. and Leigh, D. (2001): DiamondTouch: a multi-user touch technology. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology,* 219-226, Orlando, Florida, ACM Press.

Grudin, J. (1988): Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work,* 85-93, Portland, Oregon, USA, ACM Press.

Hourcade, J. P. and Bederson, B. B. (1999): *Architecture and Implementation of a Java Package for Multiple Input Devices (MID).* College Park, MD 20742, USA.

Myers, B. A., Stiel, H., and Gargiulo, R. (1998): Collaboration using multiple PDAs connected to a PC. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work,* 285-294, Seattle, Washington, United States, ACM Press.

Roseman, M. and Greenberg, S. (1992): GROUPKIT: a groupware toolkit for building real-time conferencing applications. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work,* 43-50, Toronto, Ontario, Canada, ACM Press.

Shoemaker, G. B. D. and Inkpen, K. M. (2001): *MIDDesktop: An Application Framework for Single Display Groupware Investigations.* Burnaby, BC, Canda.

Stewart, J., Bederson, B. B., and Druin, A. (1999): Single display groupware: a model for co-present collaboration. In *Proc. of the SIGCHI conference on Human factors in computing systems,* 286-293, Pittsburgh, Pennsylvania, United States, ACM Press.

Stewart, J., Raybourn, E. M., Bederson, B., and Druin, A. (1998): When two hands are better than one: enhancing collaboration using single display groupware. In *CHI '98: CHI 98 conference summary on Human factors in computing systems,* 287-288, Los Angeles, California, United States, ACM Press.

Takatsuka, M. (2005): *A component-oriented software authoring system for exploratory visualization.* Future Generation Computer Systems: Journal of Grid Computing: Theory, Methods and Applications, **21**(7): 1213-1222.

Tang, A., Boyle, M., and Greenberg, S. (2004): Display and presence disparity in Mixed Presence Groupware. In *CRPIT '28: Proceedings of the fifth conference on Australasian user interface,* 73-82, Dunedin, New Zealand, Australian Computer Society, Inc.

Tse, E. and Greenberg, S. (2004): Rapidly prototyping Single Display Groupware through the SDGToolkit. In *Proceedings of the fifth conference on Australasian user interface,* 101-110, Dunedin, New Zealand, Australian Computer Society, Inc.

Zhang, L., Kristensen, L. M., Mitchell, B., Gallash, G., Mechlenborg, P., and Janczura, C. (2004): COAST - An Operational Planning Tool for Course of Action Development and Analysis. In *Proceedings of the 9th International Command and Control Research and Technology Symposium.*