MASTER

Multimouse framework
multiple user interaction

Schakenbos, R.P.H.

Award date:
2008

Link to publication
TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER’S THESIS

MultiMouse framework: multiple user interaction

by

R.P.H. Schakenbos

Supervisor: dr. C. Huizing

Eindhoven, August 2008
Abstract

Single Display Groupware (SDG) is the category of software applications with which multiple co-present users can simultaneously interact. A common problem for aspiring SDG developers is that current operating systems do not provide support for multiple individual input sources.

The MultiMouse framework presented in this thesis provides the technology to process input from multiple mice separately. Input is separated by using an alternative input mechanism to what the operating system delivers by default. This input allows the MultiMouse framework to manage a set of cursors that interact with software applications. A benefit of this MultiMouse framework is that developers will be able to focus immediately on developing SDG applications, instead of having to deal with the technical challenges first.
Acknowledgements

I would like to thank my supervisor, dr. Kees Huizing for his guidance, understanding and support. His passionate discussions about user interaction were a inspiration for my work and his insights were of great value.

Furthermore, I would like to express my gratitude towards my parents for supporting me in every possible way during my time at the university.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Overview</td>
<td>2</td>
</tr>
<tr>
<td>2 Problem description</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Context</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Goal</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Research questions</td>
<td>4</td>
</tr>
<tr>
<td>2.4 Requirements</td>
<td>4</td>
</tr>
<tr>
<td>2.5 Related work</td>
<td>5</td>
</tr>
<tr>
<td>2.5.1 MID</td>
<td>5</td>
</tr>
<tr>
<td>2.5.2 SDGToolkit</td>
<td>5</td>
</tr>
<tr>
<td>2.5.3 MultiPoint</td>
<td>6</td>
</tr>
<tr>
<td>2.5.4 Multi-Pointer X Server</td>
<td>6</td>
</tr>
<tr>
<td>3 Design decisions</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Framework introduction</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Hardware</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Graphical user interface</td>
<td>8</td>
</tr>
<tr>
<td>3.3.1 Programming language</td>
<td>9</td>
</tr>
<tr>
<td>3.3.2 UI Components</td>
<td>9</td>
</tr>
<tr>
<td>3.3.3 Swing’s take on the Model-View-Controller pattern</td>
<td>10</td>
</tr>
<tr>
<td>3.3.4 Layout management</td>
<td>10</td>
</tr>
<tr>
<td>3.3.5 Layered architecture</td>
<td>10</td>
</tr>
<tr>
<td>3.3.6 Event handling</td>
<td>11</td>
</tr>
<tr>
<td>3.3.7 Concurrency</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Hardware communication</td>
<td>12</td>
</tr>
<tr>
<td>3.4.1 Motivation for USB usage</td>
<td>12</td>
</tr>
<tr>
<td>3.4.2 USB overview</td>
<td>13</td>
</tr>
<tr>
<td>3.4.3 Java’s USB support</td>
<td>14</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the understanding that cooperation and teamwork is key for consistent good results, we explore in this thesis possibilities for computer aided collaboration and present the design of a framework that enables such collaboration. This chapter will provide a background and motivation for the research, followed by an overview of the rest of this thesis.

1.1 Background

Since their invention, computers have been used to increase productivity. Whether this increase comes from pure calculational power or from a reduction in workforce, the underlying drive of achievements made in computer technology has always been to support the manner in which people work. As technology advances, the focus of much research and development has shifted to supporting groups rather than individuals. Software and technology that provide means for human collaboration can be referred to as groupware. This term has many definitions and ideal feature sets, but the primary focus of groupware applications is to connect people and allow them to work, learn, and create together.

Especially the user interface of a computer application plays a central role in the interaction between humans and computers. This user interface consists of both the software and the hardware that allow people to operate a computer. The combination of a mouse and a keyboard has become the standard for home and office use, but other input devices such as touch-screens, joysticks and graphic tablets are rather popular as well. In contrast to typical usage of desktop computers, we describe in this thesis research done towards a framework that enables multiple mice to be used simultaneously within an application.

1.2 Motivation

The motivation for this thesis emerged from a situation that any public speaker may be familiar with. Giving a presentation, the speaker often has a hard time estimating the rate in which the audience keeps up with the story. Because an understanding of the basic principles
that are covered early in the presentation is essential for progressing to more advanced topics, a method for getting feedback from the audience is desired.

One way to get feedback is by periodically testing the audience’s understanding of the presented material. The output of such a test could be a true or false, or a multiple choice question, to which each individual in the audience should respond. Based on the individual or combined results the speaker may adjust his story or go over a topic, which has been covered before, in more detail.

The result of a previous master’s thesis [H02] is a framework for live voting that led to the realisation of a computer application called TeleKnijp, which enables the interaction between a speaker and his audience. With TeleKnijp, the speaker can prepare a multiple-choice question on a host computer and broadcast it over the network to the notebooks of each person in the room. The individual answers are then in turn collected at the host computer, providing an overview of the results for the speaker. Although this solution works fine technically, a side-effect of using notebooks is that the audience tends to be distracted easily.

A new approach to overcome this problem is to restrict the input mechanism of the audience to mouse devices only, i.e. each member of the audience only may control a mouse to provide feedback to the speaker when requested. In this context we will proceed to the goals and requirements for this thesis.

1.3 Overview

The following chapter describes the problem addressed in this thesis and poses research questions and requirements. The third chapter discusses the design decisions for the framework. Chapter four introduces the architecture of the MultiMouse framework. The applications that are developed as proof of concept are presented in chapter five. The final chapter presents conclusions, as well as recommendations for future work.
Problem description

In this chapter we will formulate our goal and research questions. Next we establish the requirements of the framework, followed by a discussion of related work.

2.1 Context

A large body of applications specially designed to promote collaboration between people are already well established, like e-mail, chat systems and shared whiteboards.

An early approach that enables co-present collaboration is introduced in [SBD99] as Single Display Groupware. As the name implies, Single Display Groupware is a category of applications where multiple users share the same display simultaneously. In this paper, the authors argue that Single Display Groupware (SDG) is potentially useful in at least the following domains:

- **Creative domain.** “Creative projects often benefit from group activity and input, but the restrictive nature of current systems can limit expression. The potential benefits of using SDG in this domain include being able to work more effectively by working in parallel and eliminate unnecessary turn taking.”

- **Learning domain.** “Learning around current computer systems can create an inequality in the partners due to the difference in their skills and the restrictions of only having a single input device. Potential benefits of using SDG in this domain include more effective learning by being able to work at the same time with the same objects, reducing the cognitive difference between partners by giving each parallel access.”

- **Instruction domain.** “One user is more experienced than the other and has skill or knowledge to impart such as training to use software, peer teaching in a classroom, or informal help from an instructor.”

- **Sales domain.** “A sales person and customer could configure items together.”

In the remainder of this thesis we will focus on SDG applications.
2.2 Goal

A typical operating system is able to handle input from multiple input devices connected to the computer it runs on. However, the paradigm of current operating systems is based on a single mouse cursor, which behaviour is influenced by the combined input of all devices with effective dimensional motion (such as mice, touch-pads, tablets) attached to the computer. The goal of this project is to provide software developers a means to create applications to be used by multiple co-present users simultaneously and independently.

To demonstrate the interoperability of the framework with applications, two software applications will be constructed that serve as proof of concept.

2.3 Research questions

When a person interacts with the system, will he receive feedback in the form of visual cues or other means from the actions he takes? If so, how can this person distinguish the results from his actions from the other users? In some situations it may be that people have different roles, for instance in a teacher/students setting.

• How can users identify themselves within a multi-mouse application?

The problem with popular operating systems is that when multiple mice are connected, the input of all mice is merged into one single system pointer. In Windows or Mac OS for instance it is not possible to identify the original input source once the input is merged.

• How can we process input from multiple mice separately?

Abstracting from the interactive presentation application mentioned in paragraph 1.2 we want to design a framework that handles input from multiple mice in a way that it is not of concern to the underlying application.

• Can we design a multi-mouse framework that enables an application to use multiple mice?

To show the capabilities of the framework, software applications will be implemented.

• What kind of applications are suitable for a multi-mouse framework?

2.4 Requirements

There are a few requirements that limit the range of choices we can make in our design decisions, because the implementation should be feasible and easily approachable. These requirements are:

• Inexpensive hardware. Because this framework is intended to make SDG applications easily available to interested parties, there should be no overhead in the form of expensive or custom equipment. In particularly we would like to use off-the-shelf components like standard low-end mice and computers.
• **Limited platform dependency.** With the intent of making easily shareable and portable software, we aim at a solution that is as much independent of the operating system it runs on as possible.

• **Individual input capturing for multiple mice in SDG applications.** Applications that are specially developed as SDG should be able to process input from each mouse separately.

• **Legacy application support.** Applications that initially were not designed as SDG should be easily extendible to work with multiple cursors.

• **Reasonable performance.** The performance of the framework should be such that a user who controls a mouse has the feeling that his actions correspond with the change in the application. The delay should be 0.5 seconds at most.

### 2.5 Related work

#### 2.5.1 MID

A common issue that Single Display Groupware developers face is that all mice connected to a computer control one single system cursor, instead of a cursor for each mouse. This poses a problem if you want individual interaction support for each mouse in an application. Multiple Input Devices (MID) [HB99] is a framework for Windows that addresses this problem and offers an architecture to access advanced events through Java. This architecture consists of two layers. One is a cross-platform Java layer that is available to SDG developers. This layer is based on the Java event mechanism and provides custom mouse events and mouse listeners. The other layer is platform-specific and handles the transformation of separate mouse input into Java events. MID relies on DirectInput for receiving device specific input. However, DirectInput was last supported in Windows ME.

#### 2.5.2 SDGToolkit

The main focus of many research projects is the design and evaluation of SDG interaction techniques. However, a lot of time and effort goes into the development of the infrastructure that makes the development of SDG applications possible. [TG04] describes the effort of Tse and Greenberg to develop the SDG Toolkit, which purpose is to enable rapid prototyping of SDG applications.

This toolkit aims to provide solutions for challenges posed by current windowing systems: identifying multiple input sources, multiple cursors, table orientation (support for tabletop SDG applications that are displayed on a horizontal screen where people can sit around), and custom controls that distinguish between input sources. These are important issues we also face in the development of our framework.
2.5.3 MultiPoint

The MultiPoint project\(^1\) is a result of a study by Microsoft researchers on the topic of computer education in developing countries [PPT06]. They observed that the typical student-to-computer ratio is high; one PC may be shared by up to ten children at the same time. While only one of them (usually the oldest or smartest) is in control of the mouse, the others stand around pointing and offering verbal advise. In order to increase the level of involvement of the other students, the research team came up with the solution to provide a mouse for each student and let them share one computer.

For this purpose the MultiPoint Software Development Kit (SDK) has been developed. Because SDGToolkit provides a well-designed toolkit for creating SDG applications, MultiPoint has been built on this API. They developed two additional features: button controls that are aware of multiple mice, and different coloured cursors to allow children to identify their own cursor.

This project is especially valuable to us because of the experiments they carried out. Using the MultiPoint framework, they built applications that are tailored to the educational needs of various primary schools throughout China and India [MPK07]. The drawback of this project is its dependency on the Windows operating system.

2.5.4 Multi-Pointer X Server

Current operating systems allow for only one system cursor. To enable multiple cursors, one could explicitly draw these cursors on screen. This approach is indeed followed by earlier discussed toolkits. However, this is a work-around solution. Instead it would be desirable if the operating system would support this by design. One such design is the Multi-Pointer X Server [HT07]. In Figure 2.1 the two approaches are visualised. Although support for multiple mice is desirable in an operating system, this approach does not work with the current popular operating systems.

\(^{1}\) see: [http://research.microsoft.com/users/udaip/multipoint.htm](http://research.microsoft.com/users/udaip/multipoint.htm)
Chapter 3

Design decisions

This chapter is an introduction to the MultiMouse framework. We also address design issues and trade-offs that were made that led to the eventual design of the framework.

3.1 Framework introduction

Developing software that accommodates collaboration between multiple people has been a popular topic of interest for a long time. In the specific domain of Single Display Groupware, a lot of time and energy goes into creating the foundations that make collaborative software possible. The MultiMouse framework presented in this thesis is an effort to relieve developers from the task of creating such foundation and provide an environment in which they can focus on creating the SDG applications. The approach to the design of the MultiMouse framework is to analyse of which parts it should consist, discuss adequate alternatives for each part and decide on a solution. The final design will be highlighted in more detail in the framework architecture, which is presented in the next chapter.

Enrichment of applications with multi-mouse support has implications for both the end users and the developers. End users may be concerned with the features that the framework offers, like how do multiple cursors behave compared to the system cursor they are used to, how will they able to keep track of their cursor, and whether or not that amount of cursors in one screen leads to chaotic user interface? The developers on the other hand may be more concerned about how to integrate the MultiMouse framework in their application, what role each part of the framework has, and what parts of the framework can be customised to suit their application. Both aspects will be discussed in this chapter and the next chapter about the architecture of the framework.

During the design of the framework, a number of research topics arise. These topics serve as a guideline for this chapter and consist of:

- Determine the applications that the framework should support.
- Separate input from different mice, so each mouse can control a cursor.
• Process low-level mouse events to fit the application.
• Visualise multiple mouse cursors.
• Establish the ability to manage multiple mice.

3.2 Hardware

The requirement for inexpensive hardware poses the question of what input devices to use. There are many available options for creating a framework that allow multiple people to collaborate. Some of these options are impractical, custom hardware or expensive, like tabletop touch screens. With the MultiMouse framework we aim at a cost efficient solution, because we want to make it easily accessible. A simple solution is to use mice, because they are easy to use and inexpensive. Nowadays most people know how to handle a mouse, so intuitively people know what to expect when you give them control over a mouse.

The commercial price of one low-budget mouse in 2008 is around 6 euro, meaning that you can set up a multi-mouse aware system for a reasonably small amount of money. If we assume the availability of a computer, the hardware for realising a multi-mouse aware system that supports 25 mice consists of the mice, hubs and a power socket, which can be purchased for around 230 euro (Table 3.1). This is only a small investment compared to for instance a network of connected computers. The only way to connect a large number of mice is by using hubs.

<table>
<thead>
<tr>
<th>item</th>
<th>amount</th>
<th>price</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB mouse</td>
<td>25</td>
<td>€6</td>
<td>€150</td>
</tr>
<tr>
<td>7-port (USB) hub</td>
<td>3</td>
<td>€20</td>
<td>€60</td>
</tr>
<tr>
<td>master-slave power socket</td>
<td>1</td>
<td>€20</td>
<td>€20</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td></td>
<td></td>
<td><strong>€230</strong></td>
</tr>
</tbody>
</table>

Table 3.1: Cost estimation of required hardware.

3.3 Graphical user interface

The MultiMouse framework is middleware, meaning that it is software that can be used by other developers to enhance their own applications. One aim of the framework is directed towards improving functionality of a graphical user interface and therefore we shall limit ourselves to a specific subset of Java applications that rely on a certain type of graphical user interface (GUI). The motivation for choosing Swing will be discussed in context of the MultiMouse framework requirements. In this paragraph we will examine Swing more closely, as it is of influence to other design decisions made later on.

Swing is a component of the Java Foundation Classes (JFC) that make up the core of the Java 2 platform. JFC are a comprehensive set of GUI components and services to simplify the development and deployment of commercial desktop applications. The JFC consists of
five components in total: Abstract Window Toolkit (AWT), Java 2D, Accessibility, Drag and Drop and Swing. Accessibility provides support for people with disabilities (e.g., in the form of screen magnifiers and screen readers), Java 2D is a model for advanced 2D graphics and imaging, and Drag and Drop provides advanced drag and drop support for Swing components. The focus of this paragraph, however, is Swing and by implication AWT, since Swing integrates and extends parts of AWT.

### 3.3.1 Programming language

The choice for Swing automatically implies the choice for Java, which suits the framework requirements well. We aim to make the implementation as much independent of a particular operating system as possible. This can be achieved with the Java programming language, because Java code can be executed on every computer that has a Java virtual machine installed. This also makes Java applications easy to distribute to users. Java applications can be compiled into so-called Jar files, which can be run on any computer that has a Virtual Machine installed, regardless of the operating system.

Due to its licensing system, Java makes a particularly good choice for educational and non-profit purposes, because there is a large variety of open source libraries available.

### 3.3.2 UI Components

AWT, now part of the JFC, is Java’s original user interface toolkit. From the start, AWT was designed to have a platform-independent API and yet to preserve each platform’s look and feel. AWT achieves its goals by providing classes, called components, which have a platform-independent API, but that make use of platform-specific implementations (peers). This means that the look and behaviour of the components is delegated to the operating system, making a Button look like a regular Windows button in Windows. Examples of AWT components are Buttons, Lists and Labels. Because of their dependency on the operating system, AWT components are called *heavyweight*. In practice, the AWT features appeared too restrictive, because it may not be desirable to have an application appear differently on each operating system.

Swing was developed to overcome shortcomings of AWT and to extend its features. In contrast to AWT, Swing is implemented purely in Java without any native code. Swing also provides a set of components, consisting of a lightweight alternative for every heavyweight AWT component and new exclusive Swing components. The term *lightweight* is used for components that do not have a native peer and which appearance and behaviour is controlled by the Java application itself. With Swing, only the top-level components are heavyweight: JApplet, JFrame, JDialog, and JWindow. There are generally two kinds of components: containers and the rest. Containers are special components to which other components can be added, typically enhanced with a layout manager that handles the place and size of the added components.
3.3.3 Swing’s take on the Model-View-Controller pattern

The Swing architecture is rooted in the Model-View-Controller (MVC) design pattern[1][2]. The MVC architecture calls for a visual application to be broken up into three separate parts: A model that represents the data for the application, the view that is the visual representation of that data and a controller that takes user input on the view and translates that to changes in the model. Out of practical reasons, Swing uses a modified version of this MVC pattern, by separating the data from a UI delegate, which combines both the view and controller. Although only loosely based on the traditional MVC design, the choice serves at least one goal of separating model from the component, which facilitates model-driven programming. This allows the ability to tie multiple views to a single model.

The appearance of components can be changed individually by assigning different borders or background colours. Also one feature of Swing is also the ability to assign a pluggable look-and-feel (PLAF) to an application. Using customisable PLAFs, the appearance of Swing applications can for instance be changed to a cross-platform look called Metal, or to mimic a Windows look-and-feel.

3.3.4 Layout management

Every container in Swing, such as JPanel, has a layout manager. A layout manager takes care of positioning components, regardless of the platform or screen size. Using a layout manager eliminates the need to compute component placement manually, which would be hard to do, since the size required for a component depends on the platform the application is deployed on and the look-and-feel that is used.

Two common layout managers are FlowLayout and BorderLayout. FlowLayout is the default layout manager for Swing containers and adds component to the container in rows, typically starting at the left. When a component does not fit anymore, it starts a new row, like a word processor with word wrap enabled. BorderLayout distributes a container in area’s, providing a flexible way to align components along the edges of the container. Components can for example be assigned to the north or west edge, or to the center area that stretches when the container is resized. Although we shall not go into detail about all the available layout managers, it is important to understand that the user interface layout is not manually computed.

3.3.5 Layered architecture

From Figure 3.1 the components have the following responsibilities:

- **Heavyweight component**: This is the only component that has a native peer and hosts the main window of the application. The root pane it contains does all the work.
- **Root pane**: This pane contains the layered pane and a glass pane on top of that.
- **Layered pane**: The layered pane consists of multiple layers, adding depth to a Swing container. These layers are numbered, where components in a higher layer appear on
3.3. GRAPHICAL USER INTERFACE

Figure 3.1: Swing layout structure.

top of components of lower layers. This layered pane contains both a content pane and optionally a menu bar.

- Menu bar: This component contains a traditional menu bar, which pops up when selected, and is located top of the root pane.

- Content pane: Acts as the parent for all components added to the container; maintains a layout manager that you can set and add components to.

- Glass pane: A component object that sits on top of all other components in the root pane. Because this pane covers the entire top-level container, it is suited for custom painting. Painting on this pane is ideal for visualising multiple cursors.

3.3.6 Event handling

Interaction with Swing applications is based on events. When certain properties are changed, keys are pressed or components are clicked, the corresponding components will fire events. These events are data objects that contain information about the component that fired it and details of why it was created. For each type of event, there is an associated listener. In order to receive a notification of an event, a listener for that type of event has to be registered to a component. For instance, a mouse movement listener can be registered to a button and will from then on be notified when the cursor moves over the buttons visible boundaries.

Events can be divided into two groups: low-level events and semantic events. Low-level events represent window-system occurrences or low-level input. Everything else is a semantic event. Examples of low-level events include mouse and key events, both of which result directly from user input. Examples of semantic events include action and item events. A semantic event might be triggered by user input; for example, a button customarily fires an action event when the user clicks it, and a text field fires an action event when the user presses Enter. However, some semantic events are not triggered by low-level events at all. For example, a table-model event might be fired when a table model receives new data from a database.
3.3.7 Concurrency

Developers should not use independent threads to change model states in components. Instead, once a component has been painted to the screen (or is about to be painted), updates to its model state should occur only from the event-dispatching queue. The event-dispatching queue is a system thread used to communicate events to other components. It posts GUI events, including those that repaint components.

3.4 Hardware communication

3.4.1 Motivation for USB usage

There are a number of advantages to using specifically USB connections over other connection types.

- **Cross-platform.** PS/2 is a popular connector type for mouse devices, however it is increasingly becoming replaced by USB. Also PS/2 is not supported by Apple computers, whereas USB is. USB devices are recognised on all popular operating systems, including Windows, Mac and Linux.

- **Scalability.** The USB protocol consists of a certain hierarchy, where at the root of a USB connection lies a host controller. Typically this host controller is integrated in the computer to which the USB plugs are directly connected. By chaining USB devices together, using intermediate USB hubs, each of these host controllers can manage 127 USB devices (hubs included). Such a chain of hubs can go up to 5 levels deep.

- **Power supply.** The USB standards provides also power for low-consumption devices without the need for an external power supply. Although this power will not be enough
to supply a whole chain of connected mice, only the hubs need to be powered to provide enough for the connected mice.

- **Hot plugging.** An important feature of USB is the ability to easily plug-and-play, by supporting hot plugging. This means that devices can be connected and disconnected while the computer is running. In contrast, standards that do not support hot plugging either do not register the connection when a device is plugged in, or even worse cause physical damage to the motherboard. This hot plugging ability will allow a dynamic arrangement of hubs and mice.

### 3.4.2 USB overview

The USB specifications introduces terms that are used throughout USB literature. This paragraph provides an overview of these terms.

The abbreviation USB stands for "universal serial bus", meaning the specification of a serial bus for easy connection of peripheral devices to the PC. The PC, referred to as host, may have multiple host controllers. A host controller consists of both the hardware and the software that allows USB devices to be attached to and communicate with the host. The hardware of this host controller consists of an embedded hub, called the root hub, which is a communication distribution center that provides multiple ports to connect USB devices. These devices can be I/O devices, such as mice, keyboards, faxes or printers, or they can be hubs, to allow even more devices to connect to the bus.

![USB designation](a) USB designation

![USB device](b) USB device

**Figure 3.3: USB overview.**

A single physical device may consist of several logical sub-devices that are referred to as **device functions**, because an individual device may provide multiple functions, such as a webcam with a built-in microphone. Communication between USB devices is based on logical channels, called pipes. These pipes are connections from the host controller to a point on the device where data enters or leaves, called an endpoint. Each endpoint can transfer data in one direction only and to ensure reliable data delivery, a typical connection needs multiple endpoints. The collection of endpoints that are associated with a device function is called the **interface** of that device function. Each interface corresponds with a device driver on the
host controller. A collection of interfaces is called a configuration of a device, of which only one can be active at a time.

The pipes connection in Figure 3.3 is a simplification of how a device really communicates with the host. The host runs software that supports initialisation and monitoring of USB devices. Figure 3.4 ([Sta03]) shows how this software is layered. A distinction is made between user and kernel mode. Software written in user mode cannot harm the system, while software like device drivers are executed in kernel mode and have to be verified to be error free. On the top is the client software that wants to achieve the desired USB device functionality. Class libraries provide developers access to class drivers that run in kernel mode. A class is a group of devices with similar characteristics, like mass-storage or HID (human-interface devices), which can be controlled by a generic class device driver. If a device cannot be categorised into such a class, a specific device driver for that device has to be provided. The next layer is the USB system software that takes care of the enumeration, monitoring, and recognising removal and attachment of devices. The last layer, the USB host controller represents the hardware and software that allows USB devices to be connected to the host.

### 3.4.3 Java’s USB support

Since USB is bus driven, you cannot directly hook to a USB port and start listening, in contrast to traditional serial (COM) ports. As described in 3.4.2, a device driver is needed to communicate with USB devices. Two prominent projects were involved in the introduction of USB to the Java community. These projects provide libraries that allow communication from Java with the USB subsystem of the operating system.
The first project is Java Specification Request 80 (JSR080). This project has officially been assigned the java package `javax.usb`. The package consists of three parts: the USB API, a common OS independent reference implementation, and an OS specific implementation. The USB API and the common implementation have been fully released, but the OS specific implementation has only been completed for Linux and for BSD. There are two efforts done towards an implementation for the Windows platform, but both attempts are unfinished at this time.

The second project is jUSB. jUSB also provides an API, a common implementation, and a platform specific implementation. Unfortunately there only exists a Linux implementation of the OS specific part for this project as well. A Windows implementation has been researched and implemented partially, but turned out to be impractical and incomplete.

A disadvantage of this approach is that it requires special drivers to be installed in the operating system for each device, requiring its standard device drivers to be uninstalled first. Because we aim to provide support for a large group of users, this means that every time we connect the mice to the host, these drivers have to be installed before we can use the application.

### 3.4.4 HID protocol

The Human-interface device (HID) protocol is a layer on top of the USB protocol. Windows, Linux and Mac OS all provide an alternative input model for communication with HID devices.

- Windows provides the raw input API, which is developed as an alternative to the original Microsoft Windows input model for the keyboard and mouse. With the original input model, applications receive device independent input in the form of messages that are sent to its windows. For raw input, applications must register a device to an application in order to receive input data. This data will be in the form of WM_INPUT messages, which contain the identifier of the originating device. So even if there are more devices of the same class, we can still determine what device the messages originate from.

- For Mac OS there is the HID Manager API, which allows developers to directly open and communicate with HID devices.

- Linux has special file streams in the file system to which all the input from the devices are appended.

An advantage of this approach is that there is a solution available for the three most popular operating systems at this time, and another advantage is that there are no special device drivers required. Therefore we will use this approach in our final framework design.

\[1\] see [http://javax-usb.org/](http://javax-usb.org/)

\[2\] see [http://jusb.sourceforge.net/](http://jusb.sourceforge.net/)
3.5 Multiple mouse cursors

The traditional way of controlling a computer is by using a mouse and keyboard. Moving the mouse around will navigate the system cursor within the windowing system. This means that when people take control of a mouse, they identify themselves with the cursor on the screen. Intuitively, or by experience, they know how to select items, navigate through menus and operate applications. In a Single Display Groupware application, multiple users will share the same environment and therefore the experience of controlling the system cursor does not apply anymore. Relating the actions of a user to the interaction with the application suddenly becomes an issue, which we will discuss in this paragraph.

A cursor is an indication of the user’s focus within a computer screen. With the MultiMouse framework, a cursor will be assigned to each user. This paragraph discusses how users can keep track of their assigned cursor. Also, since we replace the system cursor with a custom set of cursors we should decide what to do with the system pointer.

3.5.1 Cursors

In the MultiMouse framework we assign a cursor to each user. Having multiple cursors floating around can become quite a chaos, especially if all these cursors look alike. One way for users to keep track of their cursor is to use a different cursor for each of them. But even when a user knows which cursor is his, it can be hard not to lose track when the cursors resemble each other too much. This problem can be distinguished in two parts: how to keep cursors well distinguishable and how to assign a cursor to a user.

Cursor appearance

A typical cursor in a windowing environment is arrow shaped, pointing to the hot spot of the cursor, usually in the upper-left corner. When there is more than one cursor present, we need variation in order to distinguish between them.

One kind of variation is assigning a different color to each cursor. In a group of 20 cursors some colours are bound to be similar, but the difference would still be noticeable. It is always easier to keep 2 similar coloured cursors apart than 20 equally coloured ones. Another variation is to change the shape of the cursor from the familiar arrow-shape into a different form that stands out from the others. A third variation is to add a label to a cursor, like the name of the user. A fourth variation could be to use variations in dynamics, like rotating under a certain frequency or blinking (or coupled to an action like a mouse button click).

The Nintendo Wii console interface is an example of how multiple cursors are distinguished by the combination of both a variation in color and the assignment of labels. Up to four Wii remote controllers can be connected to the console and by pointing the device at the screen will display a cursor in the shape of hand on that position. The controller contains a display with four LED lights, of which at most one is active. The LED on the controller corresponds to the number of the player, e.g. if the first LED light is on, the cursor on screen will have a blue colour and the number 1 is displayed inside the hand shape (see Figure 3.5).
Because different cursors may be appropriate for different occasions, developers should have the ability to dynamically change the appearance of cursors. This is supported by our framework, but as an example we decided on a set of cursors that is used by default in the MultiMouse framework. This icon set, consisting of 24 different images, is created by Charlotte Schmidt and can be used freely for non-commercial purposes. As can be seen in Figure 3.6, not only are the shapes of these cursors very recognisable and easily distinguishable from the others, they also use a large variety in bright colours.

Cursor assignment

Having multiple cursors in one screen can be confusing for the users if they do not know which one they control. There are a few approaches to assigning a cursor to a user, which come down to either predetermine which cursor corresponds to each device or to let the users choose their cursor at the initialisation of the application.

Our preferred solution would be to mark each mouse with the image of the cursor it controls, so it is automatically clear for each user on which cursor to focus. Our decision to use USB mouse devices prevent us from establishing this however. As described in 3.4.2 the enumeration of USB devices is carried out in kernel mode by the operating system. Unfortunately, this enumeration process is unpredictable. Every time the operating system reboots, or the topology of USB devices is changed, the devices are assigned a new device id. When a USB device is connected, the host controller requests a device descriptor, which contains specific information about the device and includes a product id and vendor id. The combination of product and vendor id is not enough to uniquely identify a device, because multiple devices of the same brand and type have the same product and vendor ids. The USB specification allows devices to specify a serial id as well, however unfortunately this option is not used for mouse devices. Thus, although we are able to separate the mouse input and process it
individually, we cannot determine from which physical device it originates. As long as the
topology of the connected USB devices stays the same, the device ids will remain the same.
So one solution would be to prepare the setup in advance, check which cursor each mouse
corresponds to and mark it accordingly.

Another option to assign cursors is by making verbal agreement to let the users take turns in
choosing a cursor. One way to establish this is to hide the cursors by default. Then as soon
as one of the users clicks on a mouse button, a general cursor pops up with which this user
can select a custom cursor from an available set of cursors. This can even be performed in
small groups in which the chance of confusion is little. The success of this method depends
on the ability of the audience to participate.

The MultiMouse framework will just assign a default cursor to each mouse. The first time
a particular setup is used it will take the group a little while to figure out which cursor
corresponds to each user, but once this is determined this cursor will be associated with
that user for the rest of the session. Optionally, the ability to hide certain cursors may help
speeding up this process, for instance by hiding a cursor when a user has found his.

3.5.2 Handling the system cursor

The MultiMouse framework will replace the system cursor with its own set of cursors. The
system cursor not only interacts with the Java application, but also with the rest of the
operating system. This creates unwanted side-effects if we do not restrict it, because all
the connected mice contribute to the actions of this system cursor. If one user moves his
mouse upwards and another user clicks a button while the system cursor is outside of the
Java application window, the application will lose it focus. In this paragraph we research four
methods to restrict the system cursor.

Capturing the system cursor

One solution would be to capture the system cursor inside the application window for as long
as the application is active and then disabling it. The problem with this approach is that
Java is an abstract language that only refers to application windows on a conceptual level
without revealing the platform dependent details. There is no support in Java to capture the
system cursor within the boundaries of a JFrame for instance. Windows supports capturing
the system cursor within a specified rectangular area using the ClipCursor method available
through the user32.dll library. Even reverting to native code, however, this poses the problem
that the application window handle is needed for this method and there is no documented
way to retrieve it in Java.

An alternative could be the java.awt.Robot class. This class is used to generate native system
input events for the purposes of for instance test automation. It has the ability to control the
system cursor, so it can put it at any required position. Using a Robot to set the cursor to a
specific location in a mouse event listener could prevent the cursor from leaving the window
boundaries. In practice this is not feasible, since the Robot only corrects the position when an
event is received. The system cursor reacts to the combined input of all devices and if a large
number of mice are moved upwards, the cursor may have moved outside the window before a
3.5. MULTIPLE MOUSE CURSORS

mouse event is received. This approach will work most of the time, but will accidentally lead to unintended behaviour.

Using the system cursor

Instead of disabling the system cursor we could use the cursor, by binding the system cursor to one of the MultiMouse cursors, or by time-slicing the cursor. Binding the system cursor to one of the cursors would allow us to distinguish the events of that one mouse from the others and treat that cursor as a master cursor with more privileges. Unfortunately, the input from all mice contribute to the system cursor and to follow one would require either filtering out all other mouse input from the windowing system. Because we do not make use of custom mouse drivers or mouse filter drivers, we cannot achieve this. Another way to let the system cursor follow one of the other cursors is using the Robot, but the disadvantages mentioned in the previous paragraph remain. Button clicks of all other mice will still be processed as well.

Time-slicing the cursor would make the system cursor switch between all the available MultiMouse cursors very rapidly in a way that it would not be noticeable by the users. This technique has been used in projects that dealt with multiple mouse cursors as well ([WBLA04], [West02]), but is dependent on the operating system. Another drawback of time-slicing is that atomic actions like single clicks work well, but fails for drag events.

Full-screen

The third approach is the only solution that requires no support from the operating system. Running an application full-screen makes it the only application that can get the focus, since the system cursor cannot move outside the boundaries. By hiding the system cursor and filtering out all its messages, we can effectively disable the system cursor.

There are some drawbacks associated with running an application full-screen. For one, having an application run full-screen means that only one application can be active at the time. But since we design the framework to be used to enhance one single Java application with multi-mouse support, we do not regard this as a major drawback, since we want to reserve as much screen space as possible for the application anyway. Another drawback is that modal dialogs do not work, which somewhat limits the content to be used in a MultiMouse extended application. However, since this is the most feasible solution, we decided to run applications full-screen.

3.5.3 Mouse management

Having multiple users interact with the same application at once can create a chaotic experience. Whereas it is common in a face-to-face meeting to take turns and provide nonverbal cues to indicate when someone is expected to participate, this becomes harder in a virtual environment with limited interaction methods. To make the experience of MultiMouse software enjoyable, there should be a mechanism to provide direction to the interaction. This management encompasses two topics: managing the status of each cursor, and managing conflicting user actions.
In most meetings there is a chairman who leads the conversation and in class settings it is the teacher who tells a student to answer a question. In a similar fashion the MultiMouse framework will provide a so called master cursor, which has a higher priority than the other cursors, such as accessing certain GUI elements reserved for this role. This will be combined with a management interface that allows the master cursor to manage the state of the other cursors. To simulate turn taking and to avoid unnecessary chaos, the master mouse should be able to hide cursor and/or disable clicking for cursors that have no purpose at that time.

Conflicting user actions can arise when two cursors interact with a GUI object at the same time. Because there is no clear agreement of what it means if two cursors drag an object simultaneously, we either have to specify this behaviour or prevent this from happening. This topic shall be discussed in more detail in the next chapter.

### 3.6 Conclusion

In this chapter we described the limitations and possibilities of the framework. Furthermore we presented research topics that were encountered during the framework design phase. For each topic we provided a decision for the final architecture as described in the next chapter.

- As hardware input device standard low-end mice are supported.
- For communication with input devices the USB protocol is used.
- We chose to support Java Swing applications for the MultiMouse framework for the reasons that they are easily distributable, OS independent and the Swing framework lends itself for providing multiple cursors support. The glass pane that is part of Swing top-level containers will be used to paint cursors.
- The event handling mechanism of Swing will be used to allow each cursor to interact with the underlying application.
- For separating mouse input we will use the available HID service (or comparable) that each operating system offers.
- A management functionality will be part of the framework, in the form of one master mouse with more privileges and a management panel for controlling the other cursors.
In this chapter we will discuss the design and implementation details of the MultiMouse framework. The framework architecture consists of four main parts: a communication component, a transformation component, a data component, and a GUI component.

4.1 Overview

In this chapter we discuss the architecture of the MultiMouse framework. The framework consists of four separate components that each fulfil a certain role. Figure 4.1 provides an overview of how these components relate to each other, to the hardware, and to the application. The communication layer processes the input of multiple mice in an alternative way than the windowing system does and converts input signals from the mice to low-level mouse events. These low-level mouse events are processed by the transformation layer and depending on the context creates mouse events that are sent to the application. The data layer provides data structures for the administration and state management of available mice. The GUI enhances the application with multiple mouse cursors and overlay panels for managing the collection of mice.

In the next paragraphs we discuss the design of each component individually and point out the challenges and limitations that were of influence.

4.2 Communication

The communication layer processes the input of multiple mice in an alternative way than the windowing system does and converts input signals from the mice to low-level mouse events. The system cursor reacts to the combined input of all connected mouse devices, but in the process from the USB signal to the reaction of the cursor, information about the origin of the signal gets lost. The reason is that the mouse related part of the Java event handling mechanism is related to the system cursor. Therefore, mouse events that are generated when
the system cursor interacts with a component do not contain any information about the specific device it originates from.

In this paragraph we introduce an alternative method for processing mouse input in a way that information about the originating device is preserved. Since there is no pure Java solution that enables us to do this, the communication layer of the framework consists of two parts: a native part and a Java part. The native part communicates with the operating system and produces low-level mouse events. These events serve as the input for the Java part that processes them into Swing-compatible mouse events.

4.2.1 Native part

The idea of using an alternative method for processing input has been inspired by other projects. There exists an emulation scene for which emulator applications are designed to recreate the hardware of game consoles and arcade systems in software. Many of the games that can be played with these emulators feature multiplayer capability that requires individual processing of multiple input sources. Two projects that offer the core functionality needed for this multiplayer capability are RawMouse \(^1\) and ManyMouse \(^2\). Both projects are open source software that permit adjustment of the code where needed, providing a wealth of information. Both projects have contributed to the eventual design of the native part of the MultiMouse framework. Most notably the ManyMouse API proved to be adequate for our purpose, especially the specific details about communication with the operating system.

To limit the dependency on operating system, the native part of the communication layer is restricted to only the necessary tasks that cannot be performed in pure Java. This native part is constructed as a library, which handles the low-level details about the interaction with

---

\(^1\)see [http://jstookey.com/arcade/rawmouse/](http://jstookey.com/arcade/rawmouse/)

\(^2\)see [http://icculus.org/manymouse/](http://icculus.org/manymouse/)
the operating system and offers the Java part of the communication layer a set of methods. The Java part can call these methods without being aware of the implementation details, as long as the native part has been implemented for the particular operating system it runs on.

The API that the native part provides covers the following aspects:

- **Initialisation.** Each operating system has its own management system for USB input devices. Before it is possible to communicate with a specific device, some initialisation steps are required. This involves requesting an inventory of the available input devices and associating a unique identifier to each mouse device. The initialisation also covers preparations for receiving input from these devices.

- **Requesting the name of a mouse device.** Each mouse is registered in the operating system with an associated name. This name is fixed. The operating system also assigns an id to a mouse during the USB enumeration. Every time the computer is rebooted or the USB topology is changed this id changes. This poses a problem, because only the id is included in the mouse events, but not the name. The MultiMouse framework provides functionality to request the name of a mouse. This is useful for finding out which device corresponds to what id.

- **Checking for new events.** When this library is initialised it will create a buffer where the incoming events of registered mouse devices are stored. When requested the events stored in this buffer are sent and removed from the buffer.

- **Quitting.** Once the application that depends on this library is closed, or no longer in need of mouse input, the library can free up any claimed resources that are no longer needed.

Regardless of which implementation is used, the native part will produce the same mouse events for the Java part. These mouse events are based on the input types that a mouse device can provide: *absolute motion*, *relative motion*, *button clicks*, *scrolling* and also an event to inform the *disconnection* of a mouse. The mouse event (Figure 4.2) contains a device id, an item and a value that is specific for each type of mouse event (see Table 4.1).

The following paragraph discusses the implementation details of the native part of the communication layer for Windows.

### Windows

For the implementation of the native communication part in the Windows environment we decide on using the raw input API, which is available since Windows XP. The raw input model differs from the original Windows input model for keyboard and mouse. A Windows application always receives device-independent input messages from the operating system, but in order to receive raw input messages an application needs to register devices it requires input from. In paragraph 4.2.1 we proposed the API for the native library as four methods
Table 4.1: Corresponding item and value for each event type.

<table>
<thead>
<tr>
<th>event type</th>
<th>item</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute motion</td>
<td>0 : x-axis</td>
<td>The exact position on the axis that item denotes.</td>
</tr>
<tr>
<td></td>
<td>1 : y-axis</td>
<td></td>
</tr>
<tr>
<td>relative motion</td>
<td>0 : x-axis</td>
<td>The distance on the axis that item denotes since the last relative mouse event from this mouse.</td>
</tr>
<tr>
<td></td>
<td>1 : y-axis</td>
<td></td>
</tr>
<tr>
<td>button click</td>
<td>0 : left button</td>
<td>The number of consecutive clicks. Since every button click sends a separate event this will be 1.</td>
</tr>
<tr>
<td></td>
<td>1 : middle button</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 : right button</td>
<td></td>
</tr>
<tr>
<td>scrolling</td>
<td>0 : scrolling up</td>
<td>The relative scroll position in the direction that item denotes.</td>
</tr>
<tr>
<td></td>
<td>1 : scrolling down</td>
<td></td>
</tr>
</tbody>
</table>

data that perform: initialisation, checking for events, requesting a device name, and quitting. For the implementation of the library the C language is used, because this allows easy access to the required Windows system libraries, of which the user32.dll library contains methods for handling raw input. In the rest of this paragraph we will present an overview of how each method is implemented.

Initialisation (Init). The initialisation method does all the hard work. Its main tasks consist of detecting available mouse devices on the computer and making preparations to start receiving raw input from these devices.

There are also two important variables initialised during this phase, an events array that is treated as a ring buffer for storing mouse events (Figure 4.2), and a mice array that contains a name and handle for each mouse. This handle is a unique identifier assigned by the operating system to each HID device and is contained in every incoming raw input message.

```c
MouseEvent events[MAX_EVENTS];
MouseStruct mice[MAX_MICE];
```

Detecting mouse devices is done in two steps:

1. Requesting a list of raw input devices. This is achieved by calling the method GetRawInputDeviceList from the user32.dll, resulting in a list that contains a handle and type of every active raw input device.

2. Filtering out the mouse devices. From the list of raw input devices, only the ones of the type RIM_TYPEMOUSE are accepted. For each of these, their name as known in the operating system is retrieved by calling the method GetRawInputDeviceInfo. This name corresponds to the devices entry in the Windows registry (e.g. `\\??\HID#Vid_046d&Pid_c00c#5&188d28c9&0&0000#{378de44c-56ef-11d1-bc8c-00a0c91405dd}`).

This line could be traced back to the corresponding registry key to retrieve more information about the device, such as a more readable name for this device like “Logitech USB WheelMouse”. In the MultiMouse framework we work with a large number of mice, however, and chances are that a number of them are the same brand and type
that all have the same description. To make it possible for developers to identify which mouse corresponds to a certain mouse id, we decided to return the original registry key-like name instead. The device handle and the device name are added to the mice array.

Making preparations to start receiving raw input requires three steps:

1. Defining a window procedure that defines what to do with incoming messages. In Windows, each application window has a method WndProc that can be implemented to handle all input directed at the window. To receive raw input messages, we need a window to register the mouse devices to. Because of the platform independency, the handle of a Java application window cannot be retrieved through its API, so as an alternate approach is needed. Therefore we will create a new window, as discussed in the second step. What the processing of messages contains will be discussed in the "Checking for events" description below.

2. Creating a message window. The only reason we need a new window is to register mouse devices to. Fortunately, the user32.dll allows us to create an invisible window that simply dispatches messages (a so called message-only window) by calling CreateWindowEx. A parameter of this method is the window procedure that we previously declared, which will be invoked every time the window receives input from a registered device.

3. Registering mouse devices to the window. Before the window receives any input, we need to register the mouse devices. This is done by calling the method RegisterRawInputDevices, which requires a specification of which devices we want to register. Such a specification is done in a RAWINPUTDEVICE object as follows:

```c
RAWINPUTDEVICE rid;
ZeroMemory(&rid, sizeof (rid));

rid.usUsagePage = 0x01;
rid.usUsage = 0x02;
rid.dwFlags = RIDEV_INPUTSINK;
rid hwndTarget = hwnd;
```

In this specification, the so called top-level usage page is 0x01 for most HID devices including keyboards and mice. The usage stands for the specific type of device, in this case 0x02 for mice. The flag RIDEV_INPUTSINK indicates that the window will always receive input, even without focus. And the target is the handle of the window we created in step 2.

**Checking for events (PollEvent).** The message-only window will have an associated procedure that is invoked whenever the window receives mouse input. Once this procedure is called, a user32.dll method GetRawInputData is used to retrieve the message data in the form of a RAWINPUT data structure. This data contains the handle of the originating device, as well as specific details about the kind of message. This RAWINPUT data is then processed into events as described in Table 4.1. The produced events are queued into the

---

3 for a full list see [http://www.microsoft.com/whdc/archive/HID_HWID.mspx](http://www.microsoft.com/whdc/archive/HID_HWID.mspx)
events buffer that fills up until the checking for events method is called. During the processing, one problem surfaces as a result of the message-only window approach: absolute motion is not received properly, because the window has no dimensions and therefore input from an absolute motion device cannot be mapped properly to it. This implementation for the MultiMouse framework does therefore not allow this type of input. But since the framework is designed to work with USB mice (which produce relative motion input), this is acceptable. When his method is called, the events buffer will be checked for unprocessed events and if there are, these will be returned.

Requesting a device name (DeviceName). This method simply returns the name from the mice list for a specified id.

Quitting (Quit). This method will clean up all the resources that are no longer needed. It also unregisters the mouse devices for the message-only window, using the same method RegisterRawInputDevices, but this time using the flag RIDEV_REMOVE.

4.2.2 Java part

The Java part of the communication layer has two tasks: providing an interface to the native library, and polling the library for new events.

Interface to native methods

The native part of the communication layer is created as a dynamic library. The rest of the MultiMouse framework is pure Java. Therefore, to make use of the library we need a solution to bridge the gap. A convenient approach is using the Java Native Access (JNA) library 4. This library provides easy access to native shared libraries, without having to write non-Java code. Access to the native methods can be achieved by subclassing the com.sun.jna.Library interface and declaring the methods that the library exports.

Polling for new events

The MultiMouseLib provides an interface to the native library. The rest of this communication layer has the task of retrieving buffered low-level mouse events from the library and delivering them to the transformation layer. Retrieving low-level events is done in a separate thread that periodically calls the PollEvent method from the library.

4.3 Data

Before we can describe the process transforming low-level input messages into Java events, we need to describe the available data structures within the framework first. The data

4see: https://jna.dev.java.net/
consists of a data model for the mouse information, and special mouse event data that allows applications to handle each cursor differently.

### 4.3.1 Mouse data

![Diagram of Mouse data and MouseModel class]

Each mouse that we connect to the framework is represented by an object of a Mouse class, which will contain the necessary information to allow this mouse to act as an individual mouse cursor in our framework. This information consists of an id, a position, a status and context information.

At the initialisation of the framework, a request to the operating system is sent for a list of connected mouse devices. This will be an enumeration id’s and names per mouse. The framework will assign each id and name to the corresponding Mouse object.

Each time a mouse moves, the movement events that are generated only contain relative movement information. Relative movement is the distance since the last mouse motion event. If we want to represent a mouse as a cursor, we need to map this relative input into an absolute coordinate system. Therefore we maintain an absolute position for each mouse, which is measured relatively to the application window. In the case of the MultiMouse framework, this corresponds to the screen coordinates.

For each Mouse object, the status consists of two conditions: whether it is enabled or disabled, and whether it is visible or hidden. Enabled or disabled will affect the ability of a mouse to interact with the underlying application and although the cursor may be visible it cannot process mouse clicks and drag events. Visible or hidden will only affect whether the cursor is visible or not. By setting a mouse to both disabled and hidden the device will be ignored entirely by the framework. To monitor the status of each mouse, the Mouse class is also an Observable. The GUI parts that handle the visualisation of the mouse cursors, and the input controller that handles the distribution of mouse events need to be registered as Observers of each Mouse object.

To be able to generate the right mouse events, we have to keep track of the component that the cursor last hovered over. This will be explained in more detail in paragraph 4.5.
The low-level mouse events that enter our system are either move, click or scroll events, but no combined events such as drag events. A drag event is a move event while one mouse button is pressed, so therefore when a mouse click event is received we register the component below that cursor as the dragging object. When successive move events are received, while the button is still pressed, the MultiMouse framework will send drag events instead of move events.

To manage the collection of Mouse objects in our framework, we use a MouseModel that contains a list of Mouse objects. We can use the MouseModel to gain access to a specific mouse, or collectively change the status of all mice at once.

4.3.2 Mouse events

Because Java is event-based, we use the same mechanism to pass our mouse events to the application. Our custom MultiMouseEvent is based on the regular MouseEvent class, but because we need the mouse id to know which device triggered the event, we include this id. To make a distinction between mouse events coming from the regular users and the mouse events coming from the master mouse, we generate MasterMouseEvents for this mouse and because it is a subclass of MultiMouseEvent components the master mouse acts as a normal mouse as well.

As an alternative approach, we could have made special listeners for our mouse events in a similar fashion as MouseListener and MouseMotionListener are able to process MouseEvents. This would make it easy for applications to register to only regular mouse events, our MultiMouse events, or both and handle them in a separately. We decided not to use custom listeners for our MultiMouse events, because of the following reasons:

- We filter out regular mouse events when the application is wrapped in our framework to block out the system cursor events entirely. So the MultiMouse events will be a substitute for the regular mouse events and not an extra type of events.
- By subclassing both MultiMouseEvent and MasterMouseEvent from MouseEvent, applications that are unaware of the MultiMouse framework can process the events as regular MouseEvents.

This approach fulfills the requirement for both supporting applications that are designed to be used with the framework, and supporting existing applications that are unaware of multiple input sources.

As mentioned in paragraph 3.3.6, there is a difference between so called low-level Java events (e.g. Mouse events) and high-level Java events (e.g. Action events). Mouse events are regarded low-level, while Action events are high-level events. It is recommended practice to
use Action listeners, because these disregard the source that triggered the event. In order to
distinguish between mice, however, Mouse listeners will have to be used.

4.4 GUI

One of the requirements of our framework is to provide multiple cursors simultaneously on
screen. Because only one system cursor is supported by the windowing system, the framework
will have to provide more cursors by painting them. A benefit of handling the cursor painting
in the framework is the ability to easily control the shape and status of each cursor. In this
paragraph we will first discuss how the cursors are drawn and then discuss how we can provide
control over these cursors.

4.4.1 Multiple cursors

Independent of the application that is enhanced by the MultiMouse framework, we need the
cursors to be drawn on top of it. We do not want situations in which application components
hide cursors from view. If for instance one of the cursors clicks a menu item, the menu
should not cover a cursor. Swing offers a solution in the form of a glass pane (as discussed
in paragraph 3.3) that lies as a transparent layer on top of the application and can be used
to draw on. Each mouse available to the framework is represented by a Mouse object (see
paragraph 4.3.1), which contains its position within the screen. At the initialisation of the
framework, a cursor, in the form of an image, is assigned to each of the available Mouse
objects. As long as the application is active, the cursors are then periodically drawn at their
respective positions on the CursorPane, which is the part of the glass pane that is used for
drawing the cursors. Because we cannot assign the master cursor to one particular mouse
device (as described in 3.5), the master cursor is determined by entering a button sequence
at the initialisation of the framework (e.g. Right button - Left button - Left button - Right
button).

Each time a mouse device moves, the data of the corresponding Mouse object is updated. To
visualise this, the framework should redraw the cursors. If the CursorPane would be refreshed
on each received event, however, the frequency would not be consistent and often so high that
the performance of the application would noticeably suffer. The implication of using the
glass pane to draw cursors on is that every time it is repainted, not only this glass pane,
but the entire application is repainted. To ensure an acceptable and stable performance, the
framework uses a dedicated thread that periodically requests a repaint of the CursorPane
(and implicitly the application as well).

4.4.2 Cursor management

Having a large number of cursors moving through the screen can be more of a distraction
than a help in some cases. Especially with tasks where only one user or a small group of
users are required, the other cursors quickly become an obstacle. To cope with this problem,
the MultiMouse framework supports cursor management in the form of a ManagementPane.
This ManagementPane features two sections: a section to change the status for each mouse in general and a section to change the status for each individual mouse (see paragraph 4.3.1). This management pane can only be accessed by the master cursor.

The ManagementPane is located just below the CursorPane. It is hidden by default and can be made visible by pressing the right arrow key on the keyboard. When visible, the left arrow key will hide it again. Processing keyboard input in Swing is event driven as well.

One of the mice available to the framework will be the master mouse, which has a higher priority with regard to accessing certain GUI elements. Each cursor has its own set of buttons in the ManagementPane. One button contains its image and is used to disable or enable button clicks for this cursor, and the other button has a pair of glasses on it to hide or show the cursor. The general section above the individual cursor buttons are three buttons to change the status of all cursors at once.

### 4.5 Transformation

The purpose of the transformation layer is to generate Java mouse events from the low-level mouse events that are sent by the communication layer. There is a semantic difference between these two types of events: a low-level mouse event is a result from a signal from the mouse device itself, while a Java mouse event is generated by a component (in reaction to the mouse cursor). The Swing event handling model is designed to generate events based on the system cursor, but the framework replaces the system cursor with multiple cursors. Each of these cursors will have to trigger mouse events in a similar fashion as the system cursor does. The transformation process is best described by analysing how mouse events are generated in Swing applications and then propose the steps it takes to create similar events, in the form of MultiMouseEvents (see paragraph 4.3.2), from an incoming low-level event.

For this discussion we use an abstraction of a Java mouse event, as a collection of:
4.5. TRANSFORMATION

<table>
<thead>
<tr>
<th>source</th>
<th>the component that fired the event.</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>the type of mouse event.</td>
</tr>
<tr>
<td>location</td>
<td>the position in the component where the event originated.</td>
</tr>
<tr>
<td>click count</td>
<td>the number of times a mouse clicked in the same position.</td>
</tr>
<tr>
<td>button</td>
<td>indicates which mouse button was clicked (if any).</td>
</tr>
</tbody>
</table>

4.5.1 Moving

Mouse devices produce so called relative motion low-level events when they are moved. In contrast to absolute motion, relative motion events only contain the positional distance since the last relative motion event. The MultiMouse framework maintains a position for all cursors and each time a relative motion event is received, the position of the corresponding cursor is updated. Maintaining cursor position means that the framework uses an internal absolute coordinate system to which it maps each cursor position. Because MultiMouse applications always run in full-screen, the coordinate system corresponds to the dimensions of the application. This coordinate system not only allows drawing the cursors the right position on the screen, but is also used to determine which application components the cursors hover over.

Swing provides four types of mouse events that relate to mouse movement:

- **MOUSE_MOVED.** This event is generated when the cursor moves within the boundaries of the source component with no buttons pressed.
- **MOUSE_DRAGGED.** This event is generated when one or more buttons are pressed while moving within the boundaries of the source component, or outside the boundaries as long as the buttons remain pressed.
- **MOUSE_ENTERED.** Fired when the cursor enters the boundaries of the source component.
- **MOUSE_EXITED.** Fired when the cursor leaves the boundaries of the source component.

![Figure 4.6: Button states.](image)

Figure 4.6 shows the effect that the mouse events have on the state of a JButton. Each button has a default state and will highlight when a cursor enters it boundaries. If a mouse button is pressed, or the cursor is dragged within the components boundaries, it appears pressed down. When the cursor is dragged outside the buttons borders, the button will remain pressed but the appearance will change once more. The actual appearance may differ between the various look-and-feels, but it gives us an idea of when to generate a kind of mouse event.

For our MultiMouse cursors we need to simulate this behaviour. Each time we receive a relative motion event from the communication layer, we need to transform it into MultiMouse events. These low-level events do not contain any contextual information, like whether there
a button pressed or not. Each mouse has an associated Mouse object, of which the cursor is a visual representation. This Mouse object contains information about the current component under the cursor and whether the mouse is dragging or moving. With this information, we simulate the behaviour for our MultiMouse cursors as follows when a relative motion event is received:

![Figure 4.7: Activity graph of motion event processing](image)

### 4.5.2 Clicking

Java has three types of mouse events that relate to button activity:

- **MOUSE_PRESSED.** This event is generated when one of the buttons is pressed. This also registers the mouse as **dragging**, meaning that when the mouse is moved before a MOUSE_RELEASED event is received, MOUSE_DRAGGED events will be sent instead of MOUSE_MOVED.

- **MOUSE_RELEASED.** This event is generated when a pressed button is released.

- **MOUSE_CLICKED.** When a mouse button is pressed and then released at the same position within a very short time frame, this event is fired as well.

This means that when a user clicks a mouse button, three mouse events are generated. On a double click this adds up to six events. At the beginning of paragraph 4.5 we introduced the properties of a mouse event and one of these was **click count**. After a user clicks once and then presses the mouse button on the same location, the click count of the MOUSE_PRESSED, MOUSE_RELEASED and MOUSE_CLICKED events will become 2 and this adds up for as many times as the user clicks on that spot.

The low-level events that are received from the communication layer are separate events that denote either a button press or a button release. The transformation into MultiMouse events comes down to the process as illustrated in Figure 4.8.
4.5.3 Scrolling

Because a mouse wheel has a special relationship to scrolling components, Java introduces an event class purely for wheel events: MouseWheelEvent, which is in turn a subclass of MouseEvent. This is because MouseWheelEvents are delivered differently than other MouseEvents. Regular MouseEvents usually affect the component directly under the cursor, but a mouse wheel is often not scrolled directly over a scroll bar, rather over the area that is contained by the scroll pane.

With MouseWheelEvents it is not necessarily the case that the component directly below the cursor fires the event. If this component does not contain a MouseWheelEventListener, the event is delivered to the first ancestor container that does (usually a scroll pane). The source and the location of the event will then be relative to this container.

4.5.4 Dispatching events

We want applications enhanced with the MultiMouse framework to behave in similar fashion as original Swing applications. However, there is an issue that needs to be resolved that originates from our choice to use a glass pane for painting mouse cursors. The glass pane we use has the property of blocking all mouse events from passing through it. Therefore the component below it is unaware of the presence of the cursor. The framework solves this problem by dispatching the events to the right component below the cursor.

Dispatching events is done in two steps: finding the right component for the event, and dispatching the event to this component. The SwingUtilities provides methods to establish both steps. SwingUtilities can be used to determine the right component for a certain event by using the getDeepestComponentAt method. This method finds the deepest component at a supplied position in a container. Calling this method on the root pane of the application finds the current visual component under the cursor, which is used as the source for the event. Because Swing is not thread safe, updates done to the GUI should be handled through the
event-dispatching queue. SwingUtilities class also provides the `invokeLater` method, which allows adding events to the event-dispatching queue.

### 4.5.5 Multiple cursor conflicts

The interaction of one mouse with Swing components is well defined. For each move, click or scroll there is a series of appropriate events that are triggered by the receiving component. The Swing framework is never intended for multiple cursors and this may cause unexpected behaviour. Atomic events like a button click or a mouse scroll are not a problem, because these are sent as individual events. Dragging poses a problem in that it requires a context, like which component is dragged, and what the original and the current location of the dragged component is. If two mouse cursors start dragging the same component, this results in contradicting information. There are two ways to deal with this: specify semantics for multiple cursor dragging, and preventing multiple mouse dragging.

There is no established semantics for dragging with multiple mice. Dragging an object with two cursors at the same time may mean resizing in one application, or in another application may mean copying multiple instances of the object. Although this is an interesting research topic, the Swing architecture does not lend itself well for this support. The layout of Swing GUI’s is controlled by layout managers instead of by the developer. Making default components dynamically customisable in running applications, like resizing a button, goes against the design philosophy of Swing. It could be achieved by using custom defined and painted objects that have integrated support for multiple mouse dragging, like photo objects that can be rotated, resized and moved within a JPanel. This may be achieved with an extra abstraction layer on top of the MultiMouseFramework, but this has not been researched in this project.

The alternative to prevent unpredictable behaviour is preventing dragging of components by multiple cursors at the same time. This can be achieved by keeping a list of the components that are dragged by each cursor. If a mouse button is pressed, the component below the cursor will be registered as being dragged until the mouse button is released. This works well with components like buttons, scroll bars and sliders. Some components lend themselves well for custom painting and interaction (e.g. in games), like panels. This poses a problem if multiple cursors are expected to interact with objects that are custom drawn on this panel. With the locking mechanism, the first cursor that drags on it will lock the panel and thus permits other cursors to interact. A solution for this problem is to allow developers to specify components that should be excluded from the locking mechanism. It is the assumed that the developer manages dragging in his application.

JDesktopPane is a Swing component that makes extensive use of dragging. In Appendix A we discuss the possibility of supporting multiple applications with MultiMouse framework at the same time. Due to dragging and focus issues we decided not to support this component in the framework.
4.6 Integration of the framework

Developers may be interested in how this framework can be integrated into their applications. The integration process is made easy by a wrapper class that serves as an entry point to the framework. The wrapper class, called MultiMouseWrapper, contains a constructor that requires a JFrame as a parameter. The assumption of the framework is that each application runs inside a JFrame and by passing this frame to the wrapper, it will take care of the integration. The following steps describe this process:

1. Initialisation of the input, the data model and transformation module.
2. Filter out all regular mouse messages.
3. Set the MultiMousePane as glass pane for the application.
4. Run the application in full-screen.

The MultiMouseWrapper object can then be accessed to change the appearance or the status of the mouse cursors if desired.

4.7 Conclusion

In this chapter we presented the architecture of the MultiMouse framework. It consists of four different layers: communication, transformation, data and GUI.

- The communication layer provides an alternative way to process mouse input by using the raw input API in Windows. This has the advantage of not requiring special drivers. Low-level events are sent to the transformation layer.
- Transformation of the input is context sensitive, relying on both the state of the cursor and the underlying application. In this layer, the low-level events are handled and which Java events are generated.
- The data layer contains mouse information and mouse events that are introduced in the framework to support multiple cursors. The separation of the data from the GUI facilitates model driven programming.
- The GUI provides painting of multiple cursors and a management pane that can be used by the master cursor to change cursor statuses.

The final implementation of the framework works well with most Swing applications. There are, however, some limitations posed by the design choices. These may be of relevance to developers. A summary of these limitations:

- Only full-screen display. This means that pop-up menu’s, dialogs or multiple frames do not work, because the graphics device claims the main application window exclusively.
- Only MouseListeners are aware of the mouse identity. Action events generated by clicking a button do not contain this information. This implies programming on a lower abstraction level than programmers originally intended.
• No JDesktopPanes and InternalFrames, since dragging is not properly supported. Even subclassing the DesktopManager is inconvenient, since the problem is rooted deeply and will require overwriting all the methods.

• No graphics tablets, touch screens or other devices that generate absolute motion, because the absolute values do not come through properly.
Applications

In this chapter we present two applications that serve as proof of concept of the MultiMouse framework. The first application is developed for an educational context, to be used as a voting tool in a class setting. The other application is a game, a genre that lends itself well for a multi-user environment.

5.1 Education

The direct motivation for this project is a desire for a voting tool to be used in a classroom setting. During a presentation or class, a teacher cannot always tell how well the students keep up with the story that is presented. With this voting tool a teacher can periodically start a vote, consisting of a question and possible answers, to which each student should answer. This gives an overall representation of the audience’s knowledge level on the topic. The idea is that this tool will not only be an aid for the teacher to better pace his presentation, but also to increase the participation level of the students by involving them in the presentation with an interactive task.

Requirements for this application are:

- Display one question with corresponding potential answers at a time.
- It should be clear how an input device should be used to choose an answer.
- During and after the voting, the teacher should have an overview of the results.

5.1.1 Design

For the design of the voting application we choose a specific format, allowing a teacher to formulate a multiple choice question with three possible answers to choose from. The visualisation of the application should be clear, meaning that the text is clearly readable and users should intuitively know how to interact with the application.
The application contains three phases: a preparation phase, a voting phase and an overview phase. In the preparation phase the teacher can edit the question text and choices text. Once he is ready, he can press a ‘play’ button, which activates the voting and starts registering the replies from the users. During the voting phase the list of intermediate results can be shown or hidden. When everyone has voted, or if the teacher decides the time is up, he can stop the voting phase and continue to the overview phase, where for each choice a percentage is shown of how much each choice was chosen. The teacher may then choose to go on to a next question, which resets the screen and the data of the previous question is added to the history in the data model.

The data model and the GUI are separated for this application. The data model consists of the text of the question, the text of the choices, and also the replies from the users. Replies are sent through the MultiMouse framework in the form of a mouse click event. A left mouse button click represents the first choice, a middle mouse button click represents the second choice and the right mouse button click represents the third choice. The teacher is able to change the data, by clicking on a text area in the GUI, like the question text field. It becomes editable and the teacher can use his keyboard to enter the question. Once the ‘Enter’ key is pressed or the teacher clicks another text field, the text is send to the data model. The GUI is updated in turn using the Observer/Observable pattern, meaning that whenever one of the text values in the data model is changed, all components that registered as an observer are notified of this change. Because the GUI is registered as an observer, the text labels will be updated accordingly.
5.1.2 MultiMouse framework integration

The MultiMouse framework provides support for both the teacher and the students interaction. The role for the teacher is to navigate through the interface, click certain text fields and change the status of the application using the buttons at the bottom. The role of the students is to click one of the mouse buttons when the voting is in progress.

The behavior of the teacher, who controls the master cursor, is equivalent to the behaviour of the system cursor without the framework. To get the voting to work for the students cursors, we needed to adapt the application a bit. Although the students cursors are invisible they are not disabled, because otherwise the application would not be able to register the button clicks. This poses the problem that although the cursors are invisible, they still move around in the application and when a student clicks a button it will be dispatched by the framework to the underlying component. This could cause unwanted behaviour such as accidental button clicks by students. Moreover, the way listeners work is that in order to receive events it needs to register to a component, which means that we would have to register all visible components to a mouse listener. The way this problem is solved in this application is to lay a transparent layer on top of the application and below the glass pane, that registers as a mouse listener. All incoming events from the master mouse are dispatched to the application below and all the incoming events from the students cursors are processed as incoming replies to the voting process.

5.1.3 Evaluation

The application has been tested with two groups of students during a Java programming class. Technically the application performed well and the features were clear, except for a minor confusion about the difference between the ‘next question’ button and the ‘play’ button. In practice the application was used in a different way than intended, namely the question written down on paper instead of typed in and the choice alternatives were written on the blackboard. One reason for this was because it was faster and another reason was that because it was a programming lecture the text did not consist of natural language all the time.

The students were all excited to actively be involved in the lecture and responded well to the voting concept. There were some technical issues that involved the hardware. For this particular system it appeared to be impossible to chain three USB hubs together. The USB specification allows for four external hubs to be chained and this issue could not be reproduced on another computer. Another issue was that this operating system required a driver to be installed for each mouse. The reason for this issue is probably the tight restrictions posed on this particular system, such as not being able to write into the Windows register.

For a future project it would be interesting to combine elements of an authoring application with this direct approach and integrate support for mathematical formulas or program code, which could for instance be realised by integrating \LaTeX{} support (a document markup language) into the application.
5.1.4 Alternative designs

During the design of this application, a number of alternatives have been considered. To motivate the eventual design and to encourage others to participate in designing Single Display Groupware applications in an educational context, we present three approaches in this paragraph. The first one is a client-server based approach, the second is an authoring environment and the third is a direct approach that allows for quick and spontaneous use.

Client-server architecture

The requirement for a teacher to have an overview of the results during and after the vote led to a design idea that involves two computers connected by a network. One of the computers is used by the teacher to create questions with a corresponding set of answers, and to provide him an overview of the vote results. The other computer runs a MultiMouse enhanced application that communicates with the audience by showing a question and processing the input that represents the votes. The teacher can prepare a vote in advance on his computer and send it to the main computer at the right moment a network that connects the two computers. This requires a protocol that specifies how the data, containing the questions and set of answers, is sent to the main computer and how the individual responses to a question are sent back to the teachers computer.

The advantage of this method is that the teacher has the ability to prepare a question with answers before presenting it to the class, because it can be awkward to have people watching you as you type. This method allows the teacher to prepare his next vote during the time that another vote is in progress. Another advantage is that a teacher can watch the results of the votes in private instead of in public, because seeing the results of people who have already voted may influence the decision of the people who have not yet voted.

The disadvantage of this method is that it requires two computers that are connected over a network. Not only does this impose impractical hardware requirements, but would require configuration of the client and server as well.

Authoring system

By an authoring system, we refer to a separate editorial environment besides a presentation environment in which a teacher can prepare his voting questions in advance. Such an editor allows a teacher to create new voting questions and store them in a database or on a file in the file system, or edit already existing questions.

Separating the editor from the presentation application allows the preparation of questions in an environment that is better adjusted to editing the content instead of editing in the actual presentation layout. This also makes it possible to add and view extra information that has no place in the presentation environment, like the order in which voting questions should be presented, or meta information about the context of the question that may be important future reference. From the set of voting questions, higher level data structures could be created in the form of sets of questions, called a lesson. Lessons would be a sequential series of questions
about one topic for instance. This enables a teacher to compose a lesson in advance from the
available questions in the database and just load that lesson into the presentation application.

A drawback of such an approach is that a teacher needs to invest time in preparing the
questions and lessons, which restrict the spontaneity somewhat. The goal for such a voting
system is to poll how well students keep up with the class and it is hard to anticipate what
topics to question about, or when to do it.

## 5.2 Entertainment

As another demonstration of how the MultiMouse framework can be put into practice, a
game has been designed. The initial idea for this game came from an evaluation of the voting
application that was tested with two groups of students. During this evaluation, the students
decided that a soccer-like game would be fun. Such a game also takes more advantage of the
movement dimension that the framework offers, instead of just clicking as used in the voting
application.

The genre of gaming lends itself well for a multi-user environment. The popularity of the
Nintendo Wii is largely based on the availability of ‘party games’, in which multiple people
typically compete against each other using pointing devices for input. This framework allows
even a larger group of people to play simultaneously on one single screen. This rest of this
paragraph describes the design of the game and discusses the integration of the MultiMouse
framework.

### 5.2.1 Conceptual design

The initial concept for the game was a soccer game. Not only is such a game suitable for
multiple players, but it is also possible to provide a single view that contains all players at
the same time.

First, we briefly explain the conceptual design of the game. A soccer game contains a play
field with two goals, two teams of players and a ball. Each user controls one member of a
team (referred to as \textit{player}) and each player is able to interact with the ball and the other
players. A team that shoots the ball in a goal scores a point. As trivial as this sounds, there
are a few decisions to be made regarding the appearance and mechanics of the game.

### Field

There are multiple ways to display a play field: a 3D view that can be rotated, a isometric
view that is a 2D view that simulates depth and perspective, or a top-down view that is plain
2D. We chose a top-down view of the play field, because adding perspective to the game might
improve its visual appeal, but may present problems to users with mapping the movements
of their mice to the movements of their players on the screen. A top-down view offers a user
a direct correspondence between moving his mouse and the movement of his player.
Player movement

This correspondence between the mouse and the place raises the issue of how users can control their player. This issue consist of how this player is represented and how the player moves according to the mouse movements. Some soccer games use realistic representations of real soccer players that move and act in a way that resembles human motion. Using only a mouse, we propose three different ways to move a player around: 1) move the player around just like a cursor without limitations on speed or direction, 2) use movement of the mouse to turn a player around his axis and moving him forward when pressing a button, 3) moving a cursor around and let the player move toward that cursor as like a spring is attached between the cursor and the player, while restricting movement of the player to maintain a realistic feel. We decided on the first option, because our aim is not to create a realistic game, but to create a fun and fast paced experience.

Moving each player around like a cursor feels more like a game of air hockey. Air hockey is played on a flat table, with a goal on each side, one puck that floats on a layer of air that streams on top of the table and, and two players that each try to bounce the puck into the goal of the opponent directly or using the walls.

Game mechanics

Instead of going for a soccer game, we go for a game that is based on a soccer game combined with the feeling of air hockey. There are still two teams of multiple players, but the ball (or puck) is bounced around by players bumping into it or by hitting a wall. To give these mechanics the right feeling, we want to add realistic physics to the game. Players should be able to bump into each other without going straight through each other, and the puck should bounce against players and walls realistically.

5.2.2 Implementation

The initialisation phase of the game is used to identify the participating users and assign each user to a team. Using a start-up screen consisting of two buttons lets users choose between the two teams. To ensure an evenly spread distribution of teams, the two buttons will alternately become active, so one team will contain at most one member more than the other. Once the teams are set, the master cursor can start the game and the start-up screen will disappear, making room for the game field.

To realise the physics that resemble the air hockey mechanics, we experimented with a physics library called JBox2D. This JBox2D is a pure Java implementation of a C++ physics engine called Box2D. For this physics engine a virtual world is defined in which objects can be created that have properties like weight and size. Forces can be applied to objects or to the entire world, like gravity (which we do not set, since this we use a top-down view). We make the game field by creating immovable walls, leaving a hole on each side that serve as the goals. The puck is created as a circle shaped object with a relative low weight and giving it properties that simulate hard material. The players are also created as circular objects with a larger weight. These player objects react to the input of the mice by binding a so called
mouse joint to each cursor and its associated player object. When a user moves his mouse, the joint makes the player object follow the cursor within the limitations of the game world. When a cursor moves over a wall, the player object will stick behind that wall until the cursor moves to a position the player object can reach.

The visualisation of the game is mostly done by custom painting. This game field consists of a JPanel on which all game components are painted. All the objects that exist in the physics world need to be painted on screen, which is done by mapping the dimensions and locations of the physical objects to the available screen space. The puck is represented by a small black circle and the players are larger circles in the colour of their team and contain the image of their original cursor (which is hidden when the game starts). The players should be distinguishable and because users already identify themselves with a certain cursor, using the image of the cursor in the player retains this association.

5.2.3 MultiMouse framework integration

By clicking on one of the team buttons, a user chooses the team to play in. Once the master cursor clicks the ready button, all the cursors that chose a team are assigned a player object (and their cursor become hidden). This play object contains the image of their cursor. This is done by requesting the current cursor set from the MultiMouseWrapper, which is the entry point for the framework. With the cursor set and the list of mouse ids in every team, the player object can be created.

The application contains a mouse movement listener that is registered to the fields JPanel. Every time a mouse moves, the listener receives a MultiMouseEvent. This event contains a mouse id for which the corresponding player object can be retrieved and updated.

This game needs no GUI update thread of itself. The MultiMouse framework already has an own GUI thread that periodically updates the screen. If the game would also contain an update thread, this would cause unnecessary repainting.
Figure 5.2: Airhockey stages.
Chapter 6

Conclusion

In this thesis we researched the design of the MultiMouse framework. This concluding chapter gives an overview of the results and includes a discussion of future work.

6.1 Results

The result of this thesis is the design of a framework that supports multiple co-present users to simultaneously interact with software applications. We made design choices that led to a four layered framework architecture.

What are the target applications that the framework will be used with? The Swing architecture lends itself well for a framework the provides multi-user interaction, which makes the MultiMouse framework an extension of Swing. Therefore applications that use the Swing GUI are suited to be used with the MultiMouse framework. The design decisions and technical limitations restrict some of the aspects of Swing applications however. To cope with the system cursor, the decision to run applications in full-screen makes using modal dialogs impossible. Another issue related to dragging, is the use of a JDesktopPane, because its Desktop Manager only supports one cursor. Multiple cursors work well with other Swing components.

How can we process input from multiple mice separately? For the separation of mouse input, we decided on using the raw input API that is accessible in Microsoft Windows. This method provides an efficient method to access HID devices, in our case mice, without the need to install drivers.

How can we provide multiple cursors? With separate input obtained from the communication layer of the framework, the source of mouse events can be identified and used to update the position of each cursor. Using the glass pane that fits as a transparent layer over Swing applications, we can draw cursors their respective position. Interaction with the application

How can users identify themselves within a multi-mouse application? Using a creative set of cursors makes it easy for a user to distinguish himself from the others once he knows which one he controls. Reserving a few minutes the first time a MultiMouse application is run with
in a certain setup allows users to play around and find out which cursor they control. The ability to hide cursors may help in this process.

Furthermore, two applications have been developed that demonstrate the applicability of the MultiMouse framework. The concept of multiple cursors was adopted quickly and managing cursors with a master cursor was received well. The performance of the framework was such that there was hardly any noticeable delay between mouse movement and cursor response.

6.2 Future work

The work presented in this thesis is a foundation for a larger research area. This includes topics that deal with collaboration and development of serious SDG applications. This paragraph will further present ideas about these two topics that are not elaborately researched, but serve as a guide for future work in this area.

6.2.1 Focus on collaboration

Collaboration between multiple mouse cursors is an interesting topic. With recent technologies (anno 2008) like the "multi-touch" interface as used in Apples iPhone, a whole spectrum of new interaction methods presents itself. The ability to scale, rotate and move pictures using a touch-screen with multiple fingers is a technique that would translate well to the MultiMouse environment.

Before it is possible to implement such a technique, however, we need a specification of the semantics of these new interaction methods. What does it mean when two cursors simultaneously drag a button in an opposite direction? This could be interpreted as resizing the button both ways, or as each cursor dragging a copy of this button to another location. To allow both interpretations, a palette could be designed that enables switching between various modes.

The features of multi-touch would work well in an environment where GUI elements can be dynamically customised. A component like a JDesktopPane allows moving and resizing of components it contains, but other GUI elements in Swing are managed by layout managers. This makes it impossible to dynamically manipulate GUI elements in runtime.

A suggestion for an approach that enables multi-touch features includes four adjustments to the current situation:

- Specification of a new high-level Java events (e.g. Collaboration events) similar to Action events. These new Collaboration events should represent a set of multi-touch actions, like resize, rotate or separate. Associated listeners should be constructed to process these events as well.
- Designing a new layout manager that enables free placements of components within a container, similar to how JDesktopPane manages its JInternalFrames.
- Designing a set of special components that are able to interpret collaborate interaction and generate Collaboration events.
6.2.2 SDG applications

The realisation of the MultiMouse framework opens up possibilities for developers to design SDG applications without having to deal with technical issues. As mentioned in chapter 2, educational or creative design applications make good candidates for multi-user interaction.

In the extend of the application proposed in paragraph 5.1 an interactive application could be designed that not only incorporates the power of a document markup language like \LaTeX, but also includes a whole spectrum of different interaction methods. Suggestions for other interaction methods are:

- Multiple choice that allows four choices. In order for users not to influence each others choices, the cursors will have to be hidden. One way to allow four alternative answers is to reserve a corner area of the screen for each of them. Instead of button clicks, mouse motion can be used to choose between four alternatives. If users move their mouse in the direction of one of the corners, the hidden cursors will eventually be located on one of the alternatives. By using a timer, the results can be counted as soon as the time for the questions runs out.

- Open ended questions. Answering an open question involves typing an answer. Since the MultiMouse framework only provides support for multiple mice, this answer will have to be entered by mouse. A virtual keyboard, such as recently introduced for the Wii (see Figure 6.1), makes this a possible. Using this for a large group will be impractical, but when only one or a few cursors are active this could work well.

![Virtual Keyboard](Figure 6.1: Virtual keyboard)
Appendix A

Multiple applications support

Current operating systems support multitasking, which makes running multiple programs at the same time a common occurrence. During the research phase for the framework design, the possibility of providing multi-mouse support for multiple applications has been researched. In this appendix we will discuss a method to establish this, the challenges that it raised and ultimately whether we will support this or not.

A.1 Approach to encapsulate multiple applications

Multiple Swing applications each run in their own window. This window is the only native component of a Swing application and is typically represented by a JFrame, JDialog or JApplet. Using Java, we do not have the tools for drawing outside of the applications boundaries, thus the requirement of visualising multiple cursors implies that these cursors are drawn within one window. If we want to support multiple applications, these applications will need to share a cursor space and solve the issue that we cannot draw outside window boundaries.

A feasible approach involves the use of a JDesktopPane, a container which serves as a virtual desktop. In a similar fashion as new windows are opened for every new application in an operating system, this special component is able to create multiple floating JInternalFrames. A JInternalFrame is the only lightweight top-level container that Swing provides and are conceptually similar to JFrames, but do not have a native peer and therefore only exist within the Java application.

We could support multi-mouse interaction for multiple applications if we design the framework as an application with a JDesktopPane. Then we could create a separate JInternalFrame for each application that needs multi-mouse support in this desktop pane. The cursors could be drawn on the glass pane that covers the entire application, so they can interact with all the individual applications in the desktop pane. There are a few issues with this approach, which we will cover next.
A.2 Issues

The desktop pane creates an environment in which it is possible to resize and rearrange internal frames, which creates a dynamic experience, but at the same time introduces the problem that all of these windows share the same desktop space. This means that each internal frame covers only a part of the entire screen or is either obstructed by another internal frame.

In a multi-user environment, the contribution of the desktop pane is that multiple applications can be accessed at the same time. Because an internal frame can be moved around and resized, it can cover up another, making parts of the covered application inaccessible to the users without bringing it on top again. This not only creates tension between users who want to interact with different applications, but also introduces the problem of losing focus.

Internal frames, in a similar fashion as windows in an operating system, are active simultaneously, but only one of them has the current focus. This focus directs the input from external devices to the currently active internal frame. If there exist two internal frames and the user turns a scroll wheel, then the internal frame that is active will process this event and move the scroll pane up or down if it is present.

The moving of the internal frames is done by dragging the mouse and this poses a problem in itself. The root of the problem lies in the DesktopManager that manages the position and size of all the internal frames in the desktop pane. Dragging an internal frame with one cursor is not a problem, but the framework offers multiple cursors.

The dragging and focus issues led to the decision not to support JDesktopPanes and multiple applications.
Bibliography


