USING CURT FOR CURSOR AND KEYBOARD REDIRECTION IN MULTI-DISPLAY ENVIRONMENTS

BY

ANDREW MCGOVERN

B.A., DePauw University, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Collaboration with multiple users and multiple machines in MDEs (multi-display environments) is lacking today in several aspects. One such aspect is that only one cursor can be present on a machine at a given time. If multiple cursors, each controlled by a different user, could be present on a machine at the same time, there would be potential for more productive collaboration between users. The fundamental problem is that current operating systems, such as Microsoft Windows, are limited to one cursor on a screen at a time. CuRT (Cursor Redirection Toolkit) provides a workaround to this, allowing multiple, functional cursors, as well as multiple keyboards, to interact with one machine simultaneously, with a few minor limitations. CuRT is not meant to be a fully functional, standalone program to allow cursor and keyboard redirection; rather, it is a toolkit that is intended to be used by developers of MDE systems who wish to include cursor redirection as one collaborative component. CuRT was intended to facilitate studies involving cursor/keyboard redirection in MDEs so that someday, the presence of multiple cursors on a single machine might be allowed at the operating system level.

Table of Contents

List of Figures

1 Introduction

People have been collaborating on projects using multiple computers for decades. The functionality provided to collaborate across machines, both in software and in hardware, however, always seems to be several steps behind what users would like to be able to do in a collaborative setting. To compensate, users find ways to work around the limitations. For example, instead of collaborating directly on a document, one user might email a copy of the document to another user, allowing both users to collaborate on it in a turn-taking fashion. Or one user might pass the mouse and keyboard to another user so that they can both collaborate on the same version of the document at (almost) the same time, but again in a turn-taking fashion. Each of these actions facilitates collaboration, but they are both workarounds. One possible solution to these workarounds is the MDE (multi-display environment). An MDE is a system where multiple computers (which could be laptops, large screens, PDAs, tabletop displays, etc.) and multiple people are arranged in an environment that is conducive to collaboration on a particular task or tasks. One basic collaborative feature of MDEs that would aid in decreasing the need for workarounds is that of cursor/keyboard redirection. Cursor/keyboard redirection is simply the ability to redirect one's cursor movements, clicks, drags, and key presses from one machine to another, thus allowing the input devices on one machine to directly control another. This has been implemented in a few systems, but always in a one-to-one relationship. In other words, one machine sends its cursor/keyboard input to one other machine, but the machine receiving the input must necessarily disable the input of its local machine. This facilitates remote control of a desktop, but it does not facilitate full collaboration. This one-to-one input redirection limitation is due to the fact that at the operating system level (in Microsoft Windows, for example), only one cursor device is allowed on a machine at a time; this is one of the main limitations of the potential of MDEs today. The CuRT (Cursor Redirection Toolkit) system provides a clean workaround to this limitation, allowing multiple cursors to exist on and interact with one single machine. The CuRT system is not a fully functional MDE collaboration system, but rather a toolkit that allows other systems to include arbitrary cursor/keyboard redirection as a collaborative feature. The many-to-one relationship of input redirection offered by CuRT facilitates a higher level of collaboration on a single machine. The cursor redirection offered by CuRT simulates true many-to-one cursor redirection on a one-cursor operating system, and

as a result, it has a few limitations. For example, no two cursors can perform a drag or open two different menus at the same time (see Section 5.4.2). Despite this, the CuRT system does provide true functionality of multiple cursors on the same screen at the same time, and social protocol is typically effective in mediating the types of lower-level conflicts that may arise. This functionality, despite its limitations, can lead to many interesting user studies related to this type of collaboration. Depending on the results of these studies, there may be a push in the future to include functionality for multiple cursors at the level of the operating system.

2 Design Goals

The inspiration for this project arose out of a desire not only to create multi-display environment (MDE) systems where multiple users can collaborate on the same work at the same time, but also to set up controlled user studies to determine just how people would best perform this collaboration, given that they are provided with the appropriate tools. Of the possible tools that could be used for collaboration in MDEs, cursor/keyboard redirection (i.e. allowing cursor and keyboard input to be dynamically sent from one machine to another, over a network, at runtime) is one of the most promising in terms of allowing multiple users to truly collaborate on a common project.

Following are the primary design goals of CuRT.

- To set up and perform user studies involving cursor redirection, a completely flexible and customizable toolkit is needed. More important than flexibility from the user's standpoint, the toolkit should be flexible from the developer's standpoint. That way, tightly controlled experiments can be set up to determine the MDE environments that will best support collaboration, using cursor/keyboard redirection as an integral component. The toolkit should function as a completely flexible API (Application Programming Interface), where every possible action related to cursor/keyboard redirection can be performed arbitrarily using method calls.
- A system that manages MDEs clearly needs to enforce protocols about which machines can send their input to which other machines, about how the decision is made to do so, etc.
- The MDE system that encompasses input redirection needs to maintain some type of spatial representation of the displays in the environment.
- The MDE system needs to determine how exactly a user goes about sending his cursor/keyboard input to another machine, and how he retrieves it later.
- The MDE system needs to determine how multiple cursors on the same screen are differentiated (by color, for example).
- The CuRT toolkit provides low-level functionality, through method calls

in the API, to allow all of these components of cursor/keyboard redirection mentioned above to be maintained by the MDE system. Used irresponsibly, the basic cursor/keyboard redirection functionality of the CuRT system could introduce grave vulnerabilities to the machines in an MDE. It is the responsibility of the MDE system itself to ensure that cursor/keyboard redirection can only occur on public or other previously approved machines, for example.

The overall goal of this project was to create a toolkit to enable cursor/keyboard redirection in a MDE system as described above. A natural next step would be to conduct user studies conducted to understand how people use input redirection as part of a larger collaborative effort in MDEs.

3 Related Work

3.1 Large Screens

The need for multiple users to collaborate on a single task has caused a push for the development of large screen environments, where multiple users can look at the same task on the same screen at the same time [13][10][29][13].

3.2 Multiple Desktops and Multiple Display Environments (MDEs)

Large screen environments are beneficial because multiple users can comfortably look at the same screen at the same time for collaborative tasks, but they generally do not offer higher resolution as compared to displays of normal size. When multiple monitors are not available, special techniques can be applied to maximize productivity with a one-monitor system [15][9][16][33][34]. But when multiple monitors are available, there is a push to create MDEs (multi-display environments), environments where multiple computers with multiple displays are integrated into one single system. Courtyard [39] was an early system that integrated individual displays with large screens. iRos [17] is an event heap that creates an "interactive workspace" that allows multiple devices to function as a single, connected workspace. iRos was used to create iRoom [11], which is a system that allows collaboration with mobile devices, such as PDAs. iRoom is limited to mobile devices, and it is application-specific; applications must be tailor-made or modified to support the iRoom framework. Gaia [35] is "middleware" that allows communication between "active spaces"; again, applications must be specifically designed to work with Gaia. The CuRT system differs from this in that it is completely application-independent. It communicates with applications through nothing more than mouse clicks and keystrokes, which are universal to all applications. It also provides a toolkit approach to input redirection; CuRT does not manage the MDE itself; rather, it provides a straightforward API so that existing MDE managers can easily include input redirection as a component of their systems.

3.3 "Reaching" with Large Displays

One important issue regarding large displays (and tabletop displays) is that of being able to "reach" certain areas of the screen. This areas can be difficult to reach either because the screen is too large for the user to physically reach with a stylus, or because the MDE contains so many desktops that it is a time-consuming process to drag the cursor across them [31][28][19][27][1][26]. The CuRT system avoids these issues by allowing a user to send his local cursor to any arbitrary display in the system–assuming that within the larger MDE system that is using CuRT as an API (see Section 5.2.2.2), he has the proper permissions to do so. There is no issue of reaching in this case, since the user can control the redirected cursor from the comfort of his own machine. And regarding the issue of dragging the cursor across several desktops, assuming the larger MDE provides an easy method to allows the user to redirect his local cursor to a specific screen (without first passing through intermediary screens), the issue of dragging the cursor across multiple screens is also resolved. Similarly, the Spotlight [20] project was designed to allow a user to highlight a specific region of a large screen used for collaborative tasks. Using CuRT, a cursor can be redirected to the large screen and pointed to the specified region to achieve the same result.

3.4 Creating Views into Application Windows

Several interesting, task-specific systems have been built on top of the basic concept of the MDE. The VNC system [41] is a cross-platform utility that provides a user on one machine with a window view of the entire desktop of a user on another machine. Similarly, the X window system [36] is a cross-platform utility that allows an application running on one machine to be displayed in a window on another machine. Wincuts [38] extends this functionality to allow an arbitrary rectangular region of an application running on one machine to be displayed in a window on another machine. MightyMouse [5] also extends VNC's functionality to incorporate turn taking and user/application preferences when performing collaborative tasks across multiple machines. A secondary portal into an application window can also be customized to provide restricted-access, useful for situations such as when a speaker would like to provide a document to the audience, but with certain aspects of the document blocked or restricted [2]. In all of these cases, however, only one cursor/keyboard can interact with a given application window at a given time. The CuRT system improves on this by allowing multiple cursors to interact with an application window simultaneously.

3.5 Application-Specific Collaboration

Early work in application-specific multi-user collaboration includes [12], which allows each user to have his own view into a common application, and Quilt [22], which was an early collaborative document editing tool that allowed tailored views based on a permission hierarchy for the users of the system. CoWord [44] is a more recent project that has extended Microsoft Word to allow multiple users to edit the same Word document simultaneously. Each of these applications is application-specific, while CuRT is application-independent.

"Intelligent collaboration transparency" [23] is an approach that attempts to allow users to perform a collaborative task on heterogeneous applications but within the same application domain (for example, word processing). Again, this approach is limiting because a separate framework must be designed for each application domain. CuRT is not only application-independent, it is application-domain-independent.

3.6 Input Redirection

The x2x [43] and x2vnc [14] systems extend the X window system to allow mouse and keyboard control to be redirect from one machine to another. This functionality is X-window-specific, and it allows only a one-to-one input redirection, where CuRT does not require X windows and provides many-to-one input redirection.

The Mouse Anywhere component of Easy Living [6] allows for input redirection to the screen that is physically closest to the user. Again, this is limited to one-to-one input redirection.

PebblesDraw is a drawing program that uses the PebblesCommander system [25]. It allows multiple cursors, each controlled by a PDA, to interact with the drawing program at the same time. Each cursor can represent a different drawing tool, and they can all use their drawing tools simultaneously on the screen. This is necessarily application-specific.

The SDG system in [40] allows for simultaneous input, but only on one specific machine, and only in a specially-designed Java drawing application used within a user study environment.

The InfoTable and InfoWall [32] allow input redirection from laptops to other laptops and other types of displays, but only when using specialized Java applications that use the Java RMI to transfer the pointing and dragging events. The CuRT system supports native Windows applications, without any modifications or special setup required.

The PointRight system [18] allows for users on multiple machines to direct their

input to other machines, using edge detection to send a cursor from one machine to the next (the keyboard input follows the mouse input, as is the case with the CuRT system). However, it does not allow for multiple cursors to be present on the same screen at the same time.

The system whose functionality is closest to that of CuRT is [42]. It uses the X window system to provide cursor redirection functionality, but it also allows for multiple cursors to control the same machine at the same time. It does this in much the same way that CuRT does, by using floating windows to simulate the additional cursors. Based on the implementation description of this system, it appears that when one cursor is performing a drag, it does not lock out the other cursors present on the same screen; instead, it places their actions into an event queue, which may cause undesired operations to occur after the first drag is done. Second, it appears that this application was made for a specific purpose, that of control room collaboration. No API is provided to allow this tool to be used to provide input redirection functionality in other applications.

3.7 Interfaces for Managing Applications in an MDE

Several interfaces are available to manage applications in an MDE, including iCrafter [30], ARIS [3], and SEAPort [4]. One of their primary limitations is that none of them currently supports cursor redirection.

Specifically, SEAPort [4] is a tool that provides the ability to relocate applications from one machine to another in an MDE. SEAPort maintains a spatial map of the multiple, heterogeneous devices in the MDE. A graphical version of this mapping, along with iconic representations of the currently running applications on each machine, is provided to the user as a means to interact with the system. The user can perform a drag-and-drop action to relocate an application from one machine to another. Currently, applications can be relocated arbitrarily from one machine to another, but there is no way for multiple users to interact with one instance of a running application at the same time. The CuRT system was designed with the intention of providing SEAPort (or an extension of SEAPort) with the ability to perform input redirection, through the use of the CuRT API.

4 Usage Scenarios

This section describes four possible user scenarios that could be facilitated by either the CuRT system as a standalone application, or the CuRT system as part of a larger program.

4.1 A Single User Giving a Presentation on a Large Display

A user connects both his personal laptop and the machine running a shared large display to the CuRT system. The user wishes to control both machines from one set of input devices. He can control his own machine locally, but then he can send his mouse/keyboard input to the large display to manage the presentation. See Figure 4.1.

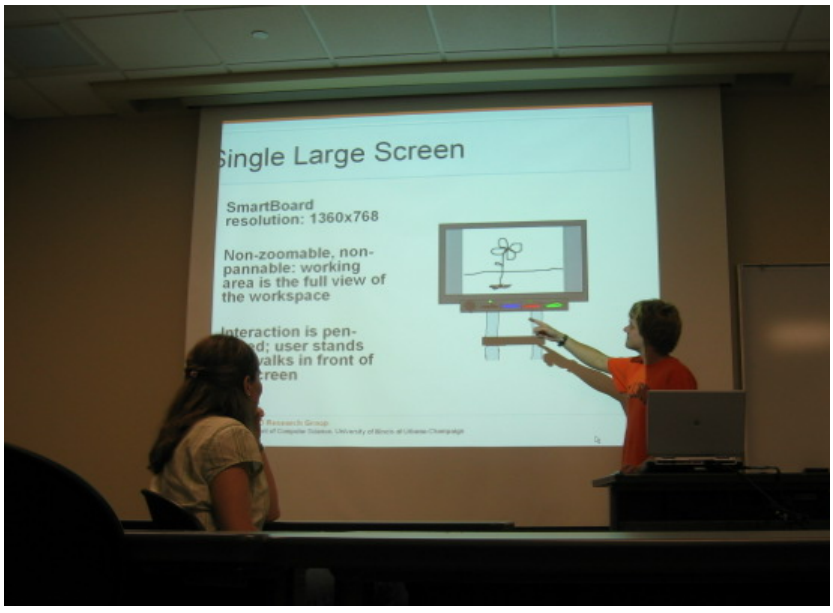

Figure 4.1: This is a photo of a user controlling a presentation on a large screen using input redirection from his personal laptop.

4.2 Multiple Users Doing Early Sketching to Create a Prototype

Each user connects his personal laptop to the CuRT system, and the machine running a shared large display is also connected. Each user can do his own sketching on his private laptop display, but he can also easily send his local input to the shared large display. When multiple cursors are present in the drawing/sketching program on the shared display, social protocol must take over. If one user is currently sketching (i.e. enacting a mouse drag with his virtual cursor), all other cursors are locked out, and the other users must wait until he finishes. When he is done, another user may make a sketch while the others wait. This turn-taking continues until the sketch is complete.

4.3 Multiple Users Co-Writing a Paper

Each user connects his personal laptop to the CuRT system, and the machine running a shared large display is also connected to the CuRT system. One user will most likely be the primary writer and will begin to write the document on the shared large display. This provides a persistent and peripheral view of the document-in-progress for the other users; as they do their own internet research, brainstorming, or prewriting on their local machines, they can look up at the master document whenever they choose. Occasionally, a user may want to take control of the master document temporarily. He simply sends his mouse/keyboard input to the shared display, and he can begin writing from his own machine. Again, social protocol takes over when multiple users are remoted to the same machine, and they must take turns. If two users try to type to the same machine at the same time, chaos ensues because both streams of characters are sent to the same window at the same time (see Section 6.1 under "Simultaneous keyboard input on two separate forms"). The CoWord project [44] provides similar functionality. See Figure 4.2.

4.4 One Continuous Desktop Comprised of Multiple Desktops

As demonstrated in the PointRight system [18], another benefit of cursor redirection is the ability to create one "superdesktop" out of several desktops tiled together. This requires one significant extension to the CuRT system: a larger system, using the CuRT API, would need to provide a listener for the CuRT edge detection. The larger system would also need to maintain a logical topography of the screens in the physical space. When the system detects than an

Figure 4.2: This is photo of two users co-writing a paper on a shared public display while doing research on their own personal laptops.

.

edge has been reached by a cursor, it simply relocates that cursor to the appropriate screen. This is similar to an extended desktop on a single machine, but there are several advantages to using cursor redirection. First, there is theoretically no limit to the number of desktops that can be connected to the system. Second, when multiple users are participating, the protocol for which user can be connected to which machines can be arbitrarily complex. For example, one user, when his cursor reaches the edge of machine A, could be sent directly to machine B, whose screen is physically located next to machine A. But a second user, who does not have access to machine B, may skip directly from machine A to machine C when making the same move as the first user. These access rules would be maintained by the application that is extending the CuRT system.

5 CuRT Implementation

As previously mentioned, the Windows operating system supports exactly one mouse cursor on the screen at a given time. The mouse operations are processed a very low level in the system so as to keep the mouse from tying up the computer's higher-level resources, and also to allow the mouse to continue to function without lag when the computer's resources are in heavy use. Due to this, it is fundamentally impossible to develop a system in Windows that allows multiple mouse cursors to function on the same machine at the same time. The CuRT system provides a workaround, where "virtual cursors" are used to simulate actual cursors. Virtual cursors are simply small Windows forms that have the same size and shape as actual cursors. They can move around the screen on machine A, receiving input from the actual mouse on machine B, with no problem. The problem arises when a virtual cursor must enact a click or a drag on the machine where it is currently being remoted. To simulate this, the virtual cursor must "borrow" the machine's local cursor. For a single or double click, this is just a matter of sending the local cursor to the location of the virtual cursor for a split second, just long enough to enact the click, and then sending it back to its original location. This causes minimal disruption to the user of the machine's local cursor. The situation is more complicated when a virtual cursor needs to perform a drag action. In order to perform a drag, the local cursor must be completely disabled during the entire duration of the drag, since the cursor cannot be both places at the same time. The situation becomes even more complicated when multiple virtual cursors are located on a screen at the same time (see Chapter 4 for specific scenarios involving multiple cursors). For now, when the machine's actual cursor is "borrowed" to enact a virtual cursor's click or drag, the actual cursor and the virtual cursor are both visually present at the location where the click or drag is taking place. Depending on the actual cursor's orientation, it may or may not obscure the virtual cursor. The primary benefit of both cursors appearing on the same spot, at least when the virtual cursor is not obscured, is that the virtual cursor remains identified by its color, and the actual "borrowed" cursor often changes to represent the action that is currently taking place; for example, a text cursor to indicate text selection, or a pencil to indicate a drawing tool. See Section 6.2 under "Customizable cursor icons" for possible future work regarding cursor images.

While not ideal, the above scenario is the best approximation to a multiple-

cursor environment, given the current limitations. It should be noted that keyboard redirection faces fewer limitations than cursor redirection. Whenever any key is pressed on any machine whose input is currently being redirected to a given machine, that key press is immediately enacted on the given machine. This would obviously cause chaos if two users are trying to type at the same time. However, the likely result would be that social protocol would take over; the users would simply end up taking turns.

This chapter describes the implementation of the CuRT system as summarized above, focusing primarily on interesting challenges that were overcome when simulating cursor and keyboard redirection.

5.1 Development Environment and System Requirements

The CuRT system was written in Microsoft Visual C# on Microsoft Visual Studio 2005 Professional Edition. Based on testing on various machines, 256 MB of RAM is necessary, and 512 MB of RAM is recommended to run the `CuRTClient` or the `CuRTServer`. 512 MB of RAM would probably be required to run the `CuRTClient` and the `CuRTServer` on the same machine at the same time, and more RAM may be required if the `CuRTClient` is used as an API (Application Programming Interface) within a larger application. Also, several functions were imported from `user32.dll` in several classes in the CuRT system to provide the following functionality: registering/releasing hotkeys, listening for cursor/keyboard events, blocking cursor/keyboard input, and getting/setting the local cursor position on the screen.

5.2 Basic Functionality

5.2.1 Server/Client Architecture

To begin, the `CuRTServer` executable must be run on a machine with a network connection. The same machine can also have a `CuRTClient` running on it, if desired. The `CuRTServer` offers no options to set, and no interaction. It simply runs in a console window and provides feedback when users connect. Users running the `CuRTClient` (either as a standalone executable or as an instantiated `CuRTClient` object within a larger application) must connect using the IP address or hostname of the `CuRTServer` on a fixed port (this fixed port cannot be seen or modified by the user). The `SocketModule` object that is part of the `CuRTClient` is responsible for all network connection initializations, all network listeners, and all network communication. (It can be assumed that

any network operations described as taking place in the `CuRTClient` are actually taking place in the `SocketModule` that is part of the `CurTClient`.) Each `CuRTClient` also manages a `CursorModule`, which is responsible for monitoring all mouse and keyboard activity, displaying cursors on the local machine that have been remoted from other machines, etc.

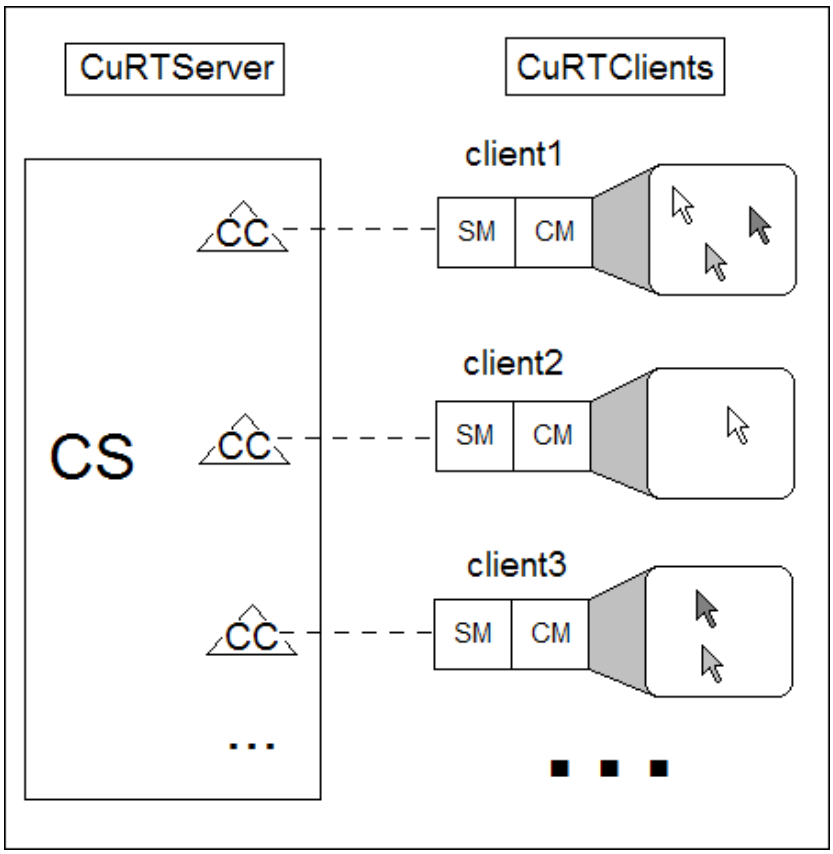

Figure 5.1: This diagram illustrates the CuRT server/client architecture. *CS = CuRTServer; CC = client communicator; SM = socket module; CM = client module; dotted lines = UDP network communication*

The `CuRTServer` has a (TCP) transmission control protocol listener that continually listens for new connections on the fixed port. Once a connection is made, the `CuRTServer` sends a message to the `CuRTClient` indicating the new port to which it should connect; this port is incremental for each new client that attempts to connect. Then, the initial connection is broken. The `CuRTServer` then creates a new `ClientCommunicator` object that is specifically designated for the client that is making the connection. This `ClientCommunicator` belongs to an array of `ClientCommunicators` for all currently connected clients. The `ClientCommunicator` immediately starts a TCP listener on the new, incremental port, and the `CuRTClient` initiates a TCP connection using this port. Then, a UDP (user datagram protocol) connection is established. A UDP listener is initialized in both the `ClientCommunicator` and the `CuRTClient`, and each is

able to send UDP messages to the other. This is how all communication occurs between sever and client (and thus between client and client, as all client-to-client communication must first pass through the server). Both the TCP and UDP connections persist until the `CuRTServer` is closed, the `CuRTClient` is closed, or the network connection is lost. The only purpose of the TCP listener is to throw an exception on the client side when the connection between the client and the server is broken. This exception is caught by the client, and the GUI is updated to inform the user that the connection has been dropped. Other than that, the UDP connection is used for all communication. While the UDP connection theoretically could allow messages to be dropped, it has performed perfectly in all testing done so far with the CuRT system. It was chosen because an extremely fast message pipeline is required to send cursor coordinates with minimum lag. (See Figure 5.1 for a diagram of the server/client architecture.)

5.2.2 CuRT as a Standalone Application Versus CuRT as an API

The CuRT system was designed to perform two primary functions. First, it was designed to be a light, easy-to-use standalone application with a basic GUI that people can use to redirect their cursor/keyboard input from their own machine to another machine connected to the same system. Its primary intended use, however, is as an API. Instead of executing the application and interacting with it directly, a developer can instantiate a `CuRTClient` object within a larger program. This section describes how both modes of operation work.

5.2.2.1 Using the CuRT System as a Standalone Application

The standalone application provides extremely basic input redirection functionality. When the application is first executed, a connection box appears. The user must specify a username for his client machine, the IP address or hostname of the CuRT server (which must already be running when the local user tries to connect), and a hotkey, to be used to send the local cursor/keyboard input back to the local machine when the input is being remoted. When the local user tries to connect to the system, all the different components are initialized, including the network connection to the server, the mouse/keyboard hooks on the local machine, the virtual cursors on the local machine, and the hotkey. If any of these initializations fails, the local user is given an error message to indicate where the failure occurred. Otherwise, the connection box disappears and the main GUI window appears.

The GUI provides a dropdown menu that lists the usernames of all clients that are currently connected to the system. A user simply chooses the client to which he wants to send his local machine's cursor/keyboard input and clicks

the "Send" button. As long as the selected machine is not his own, his local cursor/keyboard input is immediately redirected to the other machine. Only by pressing the hotkey (which was specified by the user when he first connected) can the user return his local cursor/keyboard input to his local machine.

There is no way to enforce a protocol regarding which clients can connect to other clients in the system. For example, if two personal laptops are connected to the system, the user of one could easily send his cursor/keyboard input to the other, without any confirmation. The CuRT system as a standalone application was meant to be used experimentally, as a proof of concept, and not in a normal collaborative setting (see Section 6.2 under "More advanced standalone application").

5.2.2.2 Using the CuRT System as an API

When CuRT is used as an API, the instantiated `CuRTClient` object connects to the CuRT server just as the standalone application would, but the developer has complete control over the actions of the cursor redirection. Instead of the user interacting with the GUI to redirect his cursor/keyboard input to other clients in the system, method calls are made by the developer (please see the Appendix, Chapter 8, for a complete list of available `CuRTClient` method calls). Since the CuRT system allows any client to redirect its cursor/keyboard input arbitrarily to any other client, the CuRT API provides the lowest layer of input redirection. The larger program that controls the `CuRTClient` object is then free to establish more advanced input redirection functionality. One possibility is a more advanced and more powerful GUI that a user can utilize to have more control over the input redirection. Another possibility is to implement a protocol dictating which clients are allowed to redirect input to which other clients (only to shared, public screens for example). This protocol can be arbitrarily complicated and can include authorization requests. For example, both laptop A and laptop B can redirect their own cursor/keyboard input to the public large display; laptop A cannot redirect its cursor/keyboard input to laptop B; laptop B can redirect its cursor/keyboard input to laptop A, but only after requesting authorization, and laptop A can revoke authorization at any time. A third possibility for a more sophisticated cursor redirection system is to implement a feature that "jumps" a cursor from one screen to another when the cursor comes close to the edge of the first screen (presumably, the screens would be physically located next to each other). This feature would make use of the edge detection system that is part of the CuRT system (see Section 5.3.2). For more information regarding the potential expansion of the functionality of the CuRT system within a larger system, see Section .

5.2.3 Loading and Unloading the Hotkey

In both the standalone application version and the API version of the `CuRTClient`, a hotkey must be defined. The hotkey is used to send a user's cursor/keyboard back to his own machine when it is currently being remoted to another machine. In the case of the standalone application, the user chooses the hotkey manually, and in the case of the API, the hotkey is defined programmatically and is passed as a parameter to the `CuRTClient` constructor. In both cases, the hotkey can be and of the three keys `ALT`, `CTRL`, or `WINKEY`, combined with any key `A-Z` or `F1-F12`. Also in both cases, the system attempts to register the hotkey using the `RegisterHotKey` function imported from `user32.dll`. If the hotkey cannot be registered, meaning that the hotkey is already registered on the system to perform another function, the initialization of the `CuRTClient` fails. If the hotkey registration is successful, then the `CuRTClient` can continue to initialize. In the case that the `CuRTClient` program terminates, the hotkey is freed using the UnregisterHotKey function, also from `user32.dll`.

5.2.4 Mouse/Keyboard Hooks

The file `user32.dll` provides functionality to create global hooks, or listeners, for all mouse and keyboard activity that happens on the local machine. These hooks are initialized as soon as the `CuRTClient` is initialized, and they remain active until the `CuRTClient` is closed. The mouse/keyboard hooks serve to pass any cursor/keyboard activity to the machine where the local input is currently being remoted. The mouse hook is also used to perform edge detection of the local cursor on the local machine.

5.2.5 The `VirtualCursor` Class and Its Subclasses

The `VirtualCursor` class is the parent class of the instantiated `VirtualCursor`s of various colors that are maintained by the `CuRTClient` class. The parent class provides some member variables that are used by the CuRT system, mainly for use with edge detection. There are ten `VirtualCursor` subclasses, one for each color (aqua, blue, brown, gray, green, magenta, purple, red, white, and yellow): for example, there is a `VirtualCursor`Aqua class. There is also a class corresponding to each color that represents a "blocked" cursor, such as `VirtualCursor`AquaBlocked, meaning that the cursor has a red "X" through it (see Section 5.4.2). Using the ten `VirtualCursor` subclasses and the ten "blocked" `VirtualCursor` subclasses, setting the color of a virtual cursor on the local machine, as well as temporarily "blocking" the virtual cursor, is as simple as instantiating an object of the correct `VirtualCursor` subclass during the `CuRTClient` initialization process, and then calling the `Show()` and `Hide()`

commands as necessary. There is an array in the `CursorModule` that stores the ten (potentially visible) `VirtualCursor` subclassed objects, of varying colors, and there is a corresponding array of "blocked" `VirtualCursor` objects. The indices of these `VirtualCursor`s (and "blocked" `VirtualCursor`s) correspond to the indices of the clients that are potentially connected to the server. The `CuRTClient` uses the `Visible` property of an individual `VirtualCursor` to determine if that cursor is currently being remoted to the local machine: if a `VirtualCursor` form is visible, it is currently being remoted to the local machine, and if it is invisible, it is currently is not being remoted to the local machine.

To make the `VirtualCursor`, which is simply a Windows form, appear to be a real cursor, several steps are taken. First, the transparency property of the Windows form has to be set. For the transparency property, a color is indicated (using RGB values), and any part of the background of the Windows form that matches those RBG values is set to "transparent," meaning that whatever is "behind" the form on the Windows desktop is shown, instead of that color. This is necessary because cursors are not rectangular, but "cursor-shaped," and the Windows form itself must also be "cursor-shaped." Then, the background image of the Windows form is set to be a bitmap image of a cursor of the appropriate color. However, since this is the background, the image repeats, so two opaque panels are placed in the correct positions so as to "hide" the repeating cursors from view. The end result is one cursor image that is transparent outside the borders of the cursor itself, which gives the illusion of an actual cursor. Note: CuRT currently has a minor issue regarding the transparency of the cursors; please see "Color depth and Windows form invisibility" in Section 6.1.

5.2.6 Network Message Codes

As mentioned in Section 5.2.1, the `CuRTServer` acts as the switchboard for all messages; there is no direct client-to-client communication. Generally, a client generates a message and sends the message to the server. The server's `ClientCommunicator` object for that specific client receives the message. Using the index numbers of the client generating the message and the client that will eventually receive the message, the `ClientCommunicator` may modify some of the information stored in the server about the individual clients, such as which clients are currently redirecting their input to other clients. It then sends the pertinent information to the client that is to receive the information.

This section contains two sets of messages used in communication between the `CuRTClient` and the `CuRTServer`. The first set is client-to-server communication, and the second set is server-to-client communication. Currently, the first part of the message is the message code itself, and then any needed parameters are concatenated at specific indices in the message string (character array). One

possible improvement to the CuRT system is to use an XML message structure as a cleaner way to specify message codes and associated parameters (see "XML messaging protocol" in Section 6.1).

5.2.6.1 Client-to-Server Messages

- *"MR"+ X coordinate change + Y coordinate change*: this message stands for "move (relative)," and it indicates that a local cursor, currently being remoted to another machine, has moved, and the virtual cursor should be moved as well. The move is relative because only the relative X and Y coordinate change since the last recorded cursor position, in pixels, is sent in the message.
- *"KEYPRESS" + value of key pressed*: this message indicates that a key was pressed on a local machine, and that keypress should be passed to the machine where the local machine's cursor/keyboard input is currently being remoted.
- *"SC" + cursor index + machine index + X coordinate starting position + Y coordinate starting position*: this message indicates that the machine at the specified machine index should create a virtual cursor at the specified cursor index. This new virtual cursor should immediately be relocated to the specified X and Y coordinates.
- *"RC"*: this message indicates that the cursor of the machine that sent the message should no longer be remoted. The ClientCommunicator finds the index of the machine where the current local cursor is being remoted, and it sends the appropriate message to that machine.
- *"BRC"*: this message indicates that the virtual cursor owned by the machine that sent the message should be temporarily "blocked," meaning that the "blocked" `VirtualCursor` object should be temporarily displayed in place of the normal `VirtualCursor` object. This takes place on the machine to which the cursor is currently being remoted.
- *"URC"*: this message "unblocks" the virtual cursor that was blocked by the previous message.
- *"LFTCLKDN"*: this message indicates that the local machine that sent the message has just enacted a left click down, and that this click should be passed along to the machine to where the local machine's cursor/keyboard input is currently being remoted.
- *"LFTCLKUP"*: this message indicates that the local machine that sent the message has just enacted a left click up, and that this click should be passed along to the machine to where the local machine's cursor/keyboard input is currently being remoted.

- *"RGTCLKDN"*: this message indicates that the local machine that sent the message has just enacted a right click down, and that this click should be passed along to the machine to where the local machine's cursor/keyboard input is currently being remoted.

- *"RGTCLKUP"*: this message indicates that the local machine that sent the message has just enacted a right click up, and that this click should be passed along to the machine to where the local machine's cursor/keyboard input is currently being remoted.

5.2.6.2 Server-to-Client Messages

- *"MR" + cursor index + X coordinate change + Y coordinate change*: this message is the same as the "MR" message in the previous section, except that the cursor index is included so that the client knows which of its virtual cursors needs to be moved.

- *"KEYPRESS" + originating client index + value of key pressed*: this message is the same as the "KEYPRESS" message in the previous section, except that the originating client index (the client that pressed the key) is included. This index is not currently used, but it may be interesting for future improvements to the CuRT system (see "Simultaneous keyboard input in two separate windows on the same machine" in Section 6.2).

- *"SC" + cursor index + X coordinate starting position + Y coordinate starting position*: this message is the same as the "SC" message in the previous section, except that only the cursor index is indicated. This cursor index indicates the index of the virtual cursor that should be displayed on the local machine, at the specified coordinates. Note that if the cursor index parameter is the same as the client's local index, this means that the client should activate its own cursor, which had previously been remoted to another machine (presumably the same machine that is sending the message).

- *"RC" + cursor index*: this message is the same as the "RC" message in the previous section, except that the index of the virtual cursor to be removed is included. The local machine that receives the message removes the virtual cursor.

- *"BRC"*: this message is the same as the "BRC" message in the previous section, except that the index of the virtual cursor to be blocked is included. The local machine that receives the message blocks the virtual cursor.

- *"URC"*: this message "unblocks" the virtual cursor that was blocked by the previous message.

- *"LFTCLKDN"*: this message is the same as the "LFTCLKDN" message in the previous section, except that the index of the virtual cursor to be clicked is included. The local machine that receives the message enacts a left click down at the position of the indicated virtual cursor.
- *"LFTCLKUP"*: this message is the same as the "LFTCLKUP" message in the previous section, except that the index of the virtual cursor to be clicked is included. The local machine that receives the message enacts a left click up at the position of the indicated virtual cursor.
- *"RGTCLKDN"*: this message is the same as the "RGTCLKDN" message in the previous section, except that the index of the virtual cursor to be clicked is included. The local machine that receives the message enacts a right click down at the position of the indicated virtual cursor.
- *"RGTCLKUP"*: this message is the same as the "RGTCLKUP" message in the previous section, except that the index of the virtual cursor to be clicked is included. The local machine that receives the message enacts a right click up at the position of the indicated virtual cursor.

5.3 Extended Functionality

5.3.1 Cursor Update Resolution

The CuRT system generally performs well in practice, with minimal lag as cursors are remoted from one machine to other. However, when several cursors are remoted to the same machine at the same time, and several cursors are moving at the same time on the same machine, lag can occur as the machine tries to perform edge detection on all of the cursors. The CuRT system provides a feature known as cursor update resolution in order to improve performance in situations such as this. Cursor update resolution is an integer parameter that is set when the `CuRTClient` constructor is called (note that cursor update resolution has a default value of 3 and cannot be changed in the standalone application mode). The cursor update resolution parameter represents how many coordinate updates are skipped before a relative coordinate change is sent from the local machine to the remote machine where the cursor is being remoted. For example, a cursor update resolution of 3 would mean that only every third coordinate update is actually communicated to the machine where the cursor is being remoted. Since the coordinates are communicated relatively, when the cursor update resolution is greater than 1, the relative change communicated is the change since the last communication, not since the last coordinate change. In practice, cursor update resolutions such as 3 or 4 have improved performance significantly without noticeably affecting the smoothness of the cursor's motion.

5.3.2 Edge Detection

As mentioned in Section 5.2.2.2, an obvious extension of CuRT is a system that manages an MDE (multiple-display environment) and that uses CuRT as an API to allow the user to send his local machine's cursor to any machine within a set of allowable, public displays in the MDE. There are several ways to allow this, but the most common is to allow the cursor to "jump" from the edge of one screen to the corresponding edge of another screen (*** sources here or in the next few sentences-MDE room mapping, edge jumping). There are several challenges involved. One such challenge is how to maintain a spatial map of the various and potentially mobile displays in the room so that the "jump" from the edge of one display to the edge of another display is logical according to the spatial arrangement of the displays. This challenge is left up to the MDE system to solve. A second challenge is to differentiate between the case when the user is trying to perform a "jump" to another screen and the case when the user is simply working on the edge of his own screen's desktop space or has otherwise inadvertently moved his cursor to the edge of the screen.

CuRT includes two mechanisms to detect the user's intention to "jump" his cursor to another screen. When activated, either or both of these mechanisms can fire an event for which the developer can listen in his code. The event indicates that the user is attempting to "jump" his cursor at the top, bottom, left, or right edge of his screen. Note that the screen on the user's local machine is defined as the entire desktop, so in the case of an extended desktop, since the rectangular regions of the multiple displays can be of different sizes and can be connected to each other in infinitely many ways, the edge of a screen is defined as an individual screen border past which no adjoining screen exists.

Also note that both edge-detection mechanisms work for the actual local cursor as well as any virtual cursors on a given machine. The event, when fired, provides the cursor's client index, the cursor's X and Y coordinates on the screen when the jump occurs, the part of the screen where the jump occurs (top, bottom, left, or right), the screen object where cursor is located (for use in systems using multiple monitors), and the source of the event (0 for loiter detection, and 1 for speed detection).

5.3.2.1 Speed Detection

The first mechanism detects the cursor's "speed" as it moves toward the edge of the screen, where "speed" is defined as the number of pixels (either horizontally or vertically, depending on which edge is being approached) that the cursor travels between each detected change of coordinates in the mouse listener (see Figure 5.2 for an illustration of the speed detection process). For example, when the user moves the cursor extremely slowly, the mouse listener will detect every

single coordinate change, both horizontally and vertically. An example of the series of detected coordinates (X, Y) might be (100,100), (101, 100), (102, 101), etc. In the case where the user moves the cursor extremely quickly, several pixels will be skipped between each coordinate change. An example of the series of detected coordinates (X, Y) might be (100,100), (105, 110), (111, 109). In this case, the cursor is redrawn on the screen only at those discrete coordinates, but our perception of the motion is that is smooth and connected.

When the CuRT system is used as an API, the developer has the option to define three parameters to detect the speed of the cursor as it approaches the edge. These three parameters are: zone size, minijump size, and minijump threshold. The zone size, defined in number of pixels, defines two horizontal bands of screen space (starting from the top and the bottom edges of the screen and extending toward the center) and two vertical bands of screen space (starting from the left and right edges of the screen and extending toward the center). The minijump size is also defined in pixels. When the cursor is within one of the four zones, any coordinate change toward the edge of the screen (respective to the current zone), where the number of pixels changed is greater than or equal to the minijump size, is counted as a minijump. Whenever a coordinate change does not create a minijump for a given zone, the minijump zone is reset for that zone. If the minijump count for a given zone reaches minijump threshold, then a "jump" event is fired, to be detected and acted upon by the developer using the CuRT API. Note that the zones overlap, forming squares at the four corners of the screen. This causes no problem, as the minijump counts are kept separate for each of the four zones.

5.3.2.2 Loiter Detection

The second mechanism, loiter detection, detects when the cursor has "loitered" along one of the edges of the screen sufficiently long enough to fire a jump event (see Figure 5.2 for an illustration of the loiter detection process.). Note that when the cursor "loiters" at the edge of the screen, one of the two coordinates necessarily remains constant: the X coordinate remains at 0 when loitering at the left edge; the X coordinate remains at the value of the width of the screen when loitering at the right edge; the Y coordinate remains at 0 when loitering at the top edge; and the Y coordinate remains at the value of the height of the screen when loitering at the bottom edge. In each case, movement of the non-constant coordinate is required. In other words, moving the cursor to the edge of the screen and leaving it there, with no further movement, will not trigger a loiter jump. The loiter detection mechanism takes one parameter, the loiter threshold. This represents the number of coordinate changes in the non-constant coordinate that must be detected in a row. If the constant coordinate changes, the loiter count is reset. For example, if the loiter threshold is 3, and the cursor

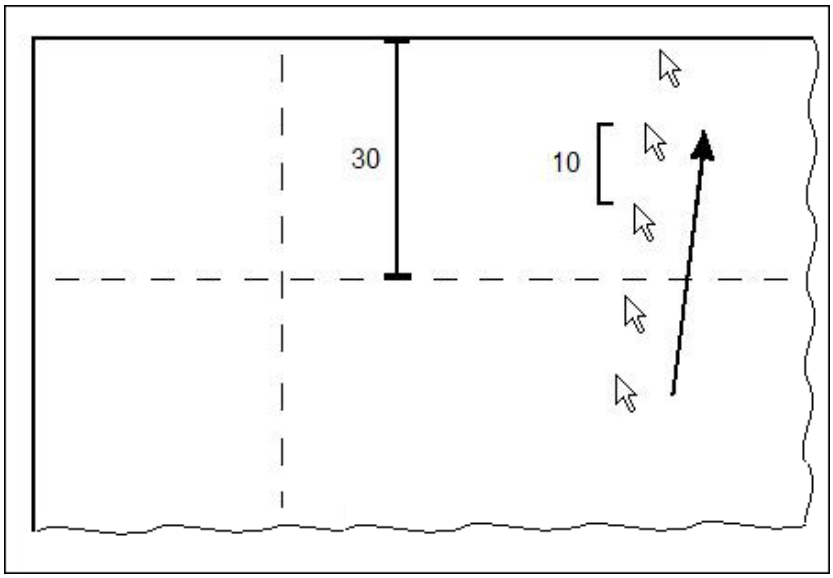

Figure 5.2: This diagram of the upper-left corner of the display illustrates the cursor movement that takes place during speed detection. The zone size in this case is 30 pixels, as indicated, and the minijump size is 10, as indicated. Two 10-pixel minijumps are registered within the zone in this example. If this is greater or equal to the minijump threshold, then a speed detection event is triggered.

is loitering on the left edge of the screen, a possible series of coordinates (x, y) might be (0,100), (0, 103), (0, 104). After the third coordinate set, the loiter event is fired, since there are three coordinate sets in a row with X=0. In contrast, with the coordinate series (0,100), (0, 103), (1, 104), (0, 106), no loiter event is fired, since the largest number of coordinate sets in a row with X=0 is two. See the "Edge detection studies" in Section 6.2 for further discussion of the two jump detection mechanisms.

5.4 Notable Challenges

5.4.1 When to Recognize Input, and When to Block It

An important challenge in cursor/keyboard redirection arises on the local machine when trying to redirect cursor/keyboard input from a local machine to a remote machine. The local machine must continue to listen for all mouse and keyboard activity, but without allowing it to take place on the local machine. Even more complicated is the situation where the local cursor/keyboard are being remoted to another machine, but another machine is currently remoting its mouse and keyboard activity to the local machine. In that case, no local mouse and keyboard activity should affect the local machine, but input coming from the other machine should affect the local machine.

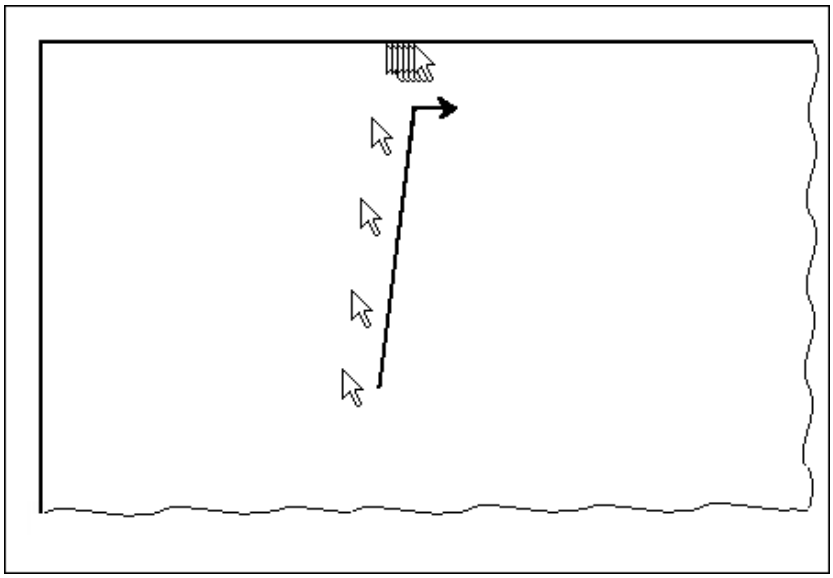

Figure 5.3: This diagram of the upper-left corner of the display illustrates the cursor movement that takes place during loiter detection. In this case, the cursor loiters along the top edge of the screen during six coordinate changes. If this is greater or equal to the loiter threshold, then a loiter detection event is triggered.

To help overcome this challenge, the CuRT system includes a Windows form subclass called `ClickGuard`. The purpose of the `ClickGuard` is that when it is enabled and brought to the front of the Z-order of the windows that are currently open, it effectively disables all clicks and keypresses on the local machine, but it still allows the clicks and keypresses to be recognized by the mouse and keyboard listeners that are part of the `CuRTClient`. `ClickGuard` is a Windows form that is automatically set to the dimensions of the primary screen on the local machine. Its background color is set to white, and its opacity is set to 2% (see "ClickGuard opacity" in Section 6.1). The `ClickGuard` form is hidden when the `CuRTClient` starts, but whenever the local input is to be redirected to another machine, the `ClickGuard` is shown. The local mouse cursor is hidden from view at this point. From this point on, no clicks or keypresses are registered on the local machine, until a command is made to send the local input back to the local machine; at that point, the `ClickGuard` is again hidden. However, the mouse and keyboard listeners remain active, and all mouse and keyboard activity is recognized, even though the local machine is not affected.

The other challenge occurs when the local input is being redirected to a remote machine, but another remote machine is redirecting its input to the local machine. In this case, whenever a mouse down action is to be simulated on the local machine, the `ClickGuard` is temporarily disabled. As soon as the mouse up action occurs, the `ClickGuard` is enabled again.

5.4.2 How Clicks and Drags Affect Other Cursors

Because of the inherent operating-system-level limitation that only one cursor can perform a click or drag at a given time (see Chapter 1), it is often the case that other cursors must be "locked out" while one cursor is performing a click or drag (see Section 5.2.5). The result of this is that no two cursors can perform a drag at the same time, meaning that no two menus can be open at the same time, no two sections of text can be selected at the same time, etc. The CuRT system uses a first come, first served policy when determining which cursor has priority, regardless of which cursor is local to the particular machine. The following sections describe specific situations where one or more cursors must be "locked out" while another cursor performs a click or drag.

5.4.2.1 Virtual Cursor Click/Drag with Local Cursor Present

In this case, a cursor is being remoted (i.e. the cursor local to machine A is invisible and disabled, and its input is being sent to machine B as a virtual cursor). See Figure 5.4. The user intends to perform a (virtual) cursor click, starting with a `MOUSEDOWN` event (for either the left or the right mouse button) on machine B. The `MOUSEDOWN` event actually takes place on machine A, but the `ClickGuard` form prevents the click from affecting machine A. However, the mouse/keyboard listener on machine A detects the `MOUSEDOWN` event. The `localCursorIsRemote` flag in the `CursorModule` determines whether machine A's local cursor is currently being remoted. The flag is `true` in this case, so a `MOUSEDOWN` message is sent by machine A to the CuRT server. The CuRT server then relays the `MOUSEDOWN` message to the appropriate remote machine, machine B (note: in the interest of simplicity, throughout the rest of this and the following sections, the description of the messages passing through the server will be omitted). In order for machine B to be able to enact a click/drag action for the virtual cursor, it must temporarily "borrow" the clicking/dragging ability of the local cursor. As mentioned in chapter 1, this is a grave and inherent limitation of current operating systems such as Windows. Upon receiving the `MOUSEDOWN` message, the `localCursorIsRemote` flag is checked, this time on machine B, to determine whether the local cursor is currently being remoted. If it is, the procedure follows as described here. If it is not, then the procedure in the next section is followed, as this means that the local cursor is being remoted. In the case where the local cursor is present on the local machine (machine B in this case), the machine B's local cursor is relocated to the coordinates of the virtual cursor that is attempting to click (actually, to one pixel above and one pixel to the left of the virtual cursor, to avoid enacting a click directly on top of the Windows form that simulates the virtual cursor), and the `MOUSEDOWN` action is effected on machine B at that location. Machine B's local cursor's previous location is recorded, and the `BlockInput` function, imported from `user32.dll`

is called on machine B, which disables all local mouse movement on machine B. Also, a "blocked" virtual cursor (with a red "X" through it) is displayed in the previous location of machine B's local cursor, indicating to machine B's user that his cursor is currently disabled (see Section 5.2.5). The `drag` flag in the `VirtualCursor` class is set to `true`, indicating that a drag is currently taking place.

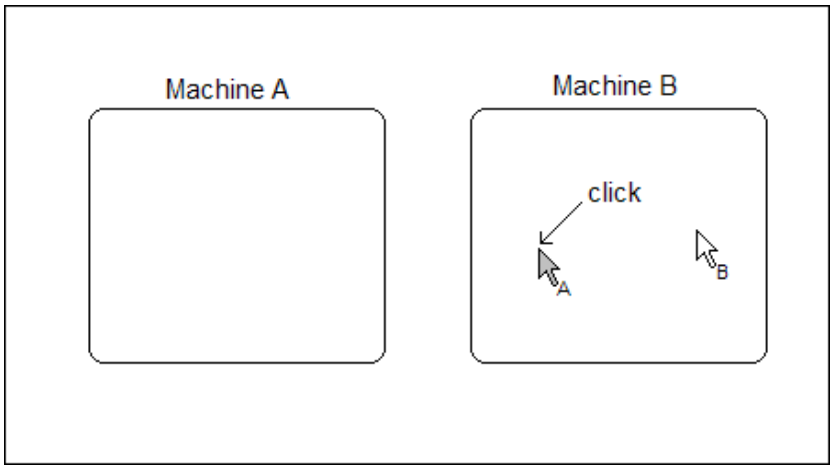

Figure 5.4: This diagram illustrates a virtual cursor click (on machine B, from machine A) when the local cursor is present (on machine B). Note that the letters on the cursors are simply labels representing the machine that owns the cursor.

In the case where there are other virtual cursors (besides the one enacting the click) on machine B, the `BlockAllVirtualCursorsExceptOne` method in the `CursorModule` class uses a `for` loop to find all active virtual cursors (when the `Visibility` property of a virtual cursor is set to `true`, this indicates that the cursor is currently active on the local machine), excluding the one that is actually performing the click. For each active virtual cursor, it is set to invisible, and its corresponding blocked virtual cursor (again, with a red "X" through it) is moved to the same location as the active virtual cursor and set to visible. When a message attempting to move or click a virtual cursor arrives on the local machine, the local machine first checks to see if a cursor is currently enacting a drag. If so, the movement or click message is ignored. In this way, all cursors, both local and virtual, except for the one enacting the click/drag, are effectively disabled.

In the case of a drag, there is cursor movement after the `MOUSEDOWN` event and before the `MOUSEUP` event. Messages for machine A's mouse movement are sent to machine B, just as they would be sent if the virtual cursor were not currently in a drag state. The only difference is that a `MOUSEDOWN` action has been effected on machine B, so any movement of machine A's virtual cursor on machine B will move the virtual cursor as well as machine B's local cursor.

All `MOUSEDOWN` events must be followed by a `MOUSEUP` event. This `MOUSEUP` event is detected on machine A, and a message is sent to machine B. When the message is received by machine B, all blocked cursors are returned to their normal state, and the "borrowed" local cursor is returned to its previous location.

In the case of a single, rapid click, the `MOUSEDOWN` and `MOUSEUP` events happen in quick succession. In the case of a double click, the `MOUSEDOWN`, `MOUSEUP`, `MOUSEDOWN`, and `MOUSEUP` events happen in quick succession. Both the simulated single click and simulated double click work well in practice when following the above procedure. Even though most of the above actions are in preparation for a prolonged drag, they must still happen in the case of a single click or double click. This is because after the `MOUSEDOWN` event occurs, it is impossible to know whether the user is going to begin a drag motion or just immediately release the mouse for a single click.

5.4.2.2 Virtual Cursor Click/Drag with Local Cursor Remoted

The procedure for this case is the same as the above case (Virtual Cursor Click/Drag with Local Cursor Present), with a few exceptions. In the previous case, the local cursor in machine B is not being remoted, so to "borrow" it to enact the click/drag is just a matter of relocating it on the same screen and then moving it back afterward. But in the case when the local cursor in machine B is being remoted, the process is more complicated. In this case, machine A's cursor is being remoted to machine B, and machine B's cursor is being remoted to another machine (call it machine C). See Figure 5.5. First, a message ("BRC": "block remote cursor") must be sent to the machine where machine B's local cursor is currently being remoted, machine C. Upon receiving the message, machine C calls the `BlockVirtualCursor` method in the `CursorModule`. This method disables machine B's virtual cursor on machine C; the cursor is set to invisible, the corresponding blocked cursor is relocated to the same position, and the blocked cursor is set to visible. Note that in this case, only one virtual cursor is disabled.

At the same time, on machine B, the local cursor (which has been hidden, as the cursor is being remoted to machine C) is sent to the location where the click needs to be made. The `ClickGuard`, which is used to prevent clicks from occurring on the local machine B while its cursor is being remoted, is temporarily set to invisible so that the click/drag can occur. Also, local mouse movement on machine B is temporarily disabled (using the `BlockInput` function imported from `user32.dll`) so that machine B's mouse activity doesn't interfere with either machine B's actual cursor on machine B, or machine B's virtual cursor on machine C. As machine A's local mouse moves, movement messages are sent to machine B to move both machine A's virtual cursor on machine B and machine B's local cursor, just as described in the previous section. Then, once

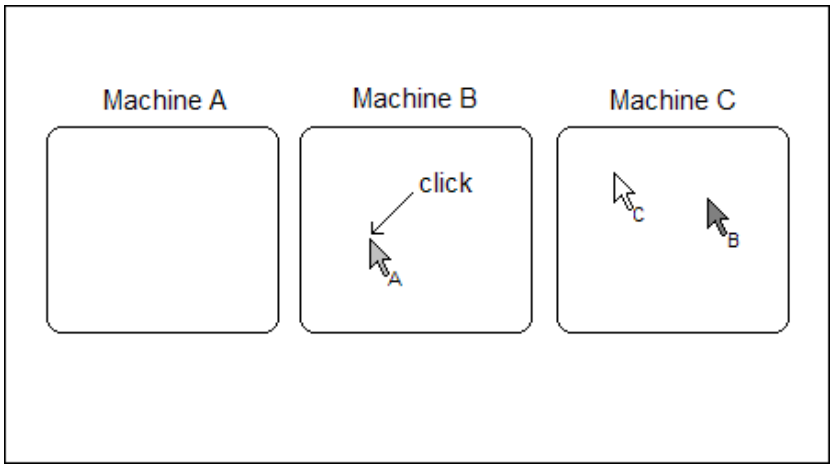

Figure 5.5: This diagram illustrates a virtual cursor click (on machine B, from machine A) when the local cursor is remoted (on machine C). Note that the letters on the cursors are simply labels representing the machine that owns the cursor.

the `MOUSEUP` event occurs on machine A, the message is sent to machine B. All blocked cursors are unblocked, and the `ClickGuard` on machine B is activated again to block machine B's mouse input on its own machine.

5.4.2.3 Local Cursor Click/Drag

In this case, a click or drag is being performed by the local cursor on machine A, and virtual cursors may or may not be present on machine A. See Figure 5.6. The mouse/keyboard listener is always active as long as the CuRT client is running, regardless of whether the local cursor is currently being remoted. When the local CuRT client mouse listener detects a `MOUSEDOWN` event, it checks whether the machine's local cursor is currently being remoted. If it is not being remoted, this indicates that the user is trying to execute a `MOUSEDOWN` on the local machine. As a reaction to this event, for each virtual cursor that is currently visible (and thus active) on the local machine, the virtual cursor is made invisible, and the corresponding blocked cursor is made visible. Any cursor movement messages for the virtual cursors that are received by this same client first check to see if the machine's local cursor is enacting a drag. If so, the cursor movement messages are ignored. In this way, all virtual cursor movement is disabled while the local cursor is enacting a click/drag. Then, once the local mouse listener detects a `MOUSEUP` event, all virtual cursors on the local machine are set to visible again, their corresponding blocked cursors are set to invisible, and their cursor movement messages are no longer ignored.

Interestingly, in the case where virtual cursors are present, it would be possible for them to continue to move around the screen while the local cursor is in a `MOUSEDOWN` state. However, in the situation described in the previous two

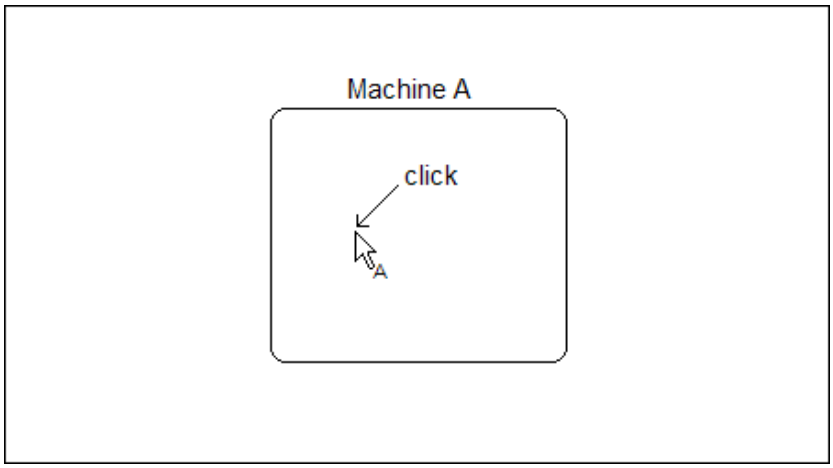

Figure 5.6: This diagram illustrates a local cursor click (on machine A). Note that the letter on the cursor is simply a label representing the machine that owns the cursor.

sections, when a virtual cursor is in a `MOUSEDOWN` state, the local cursor must necessarily be disabled to be able to simulate the virtual cursor's click and/or drag action. To ensure consistency regardless of which cursors are being remoted on which machines, all virtual cursors are disabled when any cursor on a machine, local or virtual, in a `MOUSEDOWN` state.

5.4.3 Cursor Looping and Pixel Mapping

Another challenge arises with the fact that the local machine and the machine to where the local input is redirected will often have different resolutions. This is a problem when the local cursor reaches the edge of the local screen, but the remote screen (to where the local input is being redirected) still has more space before the edge is reached. This problem is exacerbated when one or more of the machines in a system have an extended desktop with two or more monitors, meaning that the effective width of a machine's screen can be two- or three-times the width of a normal screen. To resolve this issue, a cursor looping system was developed for CuRT. When local input is redirected to a remote machine, the local cursor is confined to the primary screen (this applies to systems with extended desktops). The `ClickGuard`, as described in the previous section, is activated, and the mouse cursor becomes invisible. Then, any time the local cursor, which is now invisible, comes to within ten pixels of one of the edges of the primary screen, it is automatically relocated to the center of the screen. This works because all mouse movements are sent from the local screen to the remote screen using relative coordinates; that is, the coordinates represent the distance that the cursor has traveled since the last time that the coordinates were sent. This way, the difference in resolution between the local machine and the remote machine is immaterial. The user does not realize that

the looping is happening on the local machine; it appears that cursor is moving in one continuous motion on the remote machine. Effectively, this provides a one-to-one pixel mapping from the local machine to the remote machine, so no factor needs to be determined to convert from the resolution of one machine to the resolution of another.

6 Future Work

The future work section is divided into two parts. The CuRT system was designed as a prototype tool; while all basic functionality is provided, some features are currently lacking that would need to be included for CuRT to be considered a complete and robust input redirection toolkit. These basic features are included in the first list. The second list, of extended features, includes more advanced features, including those that would require the development of a larger application, using CuRT as an API. Some of these features also include the possibility of performing user studies to learn more about how users would interact with and best benefit from a system such as CuRT in a collaborative environment.

6.1 Basic Features

- *Maximum number of users*: currently, the maximum number of users that can be connected to the CuRT system at the same time is ten. If an eleventh user tries to connect, he simply gets a "connection failed" error message. This could be extended without much difficulty.
- *Reusing client indices*: currently, if a user connects to the CuRT system and then disconnects, his client index is not reused, nor is his username removed from the master list of client usernames. This would be a relatively easy change to implement that would allow for more efficient use of the ten possible spots for clients.
- *Username conflicts*: currently, there is no feature in place to detect username conflicts when users connect to the CuRT system. When a CuRTClient method call has a username as a parameter, it simply chooses the first matching username. This ambiguity would be avoided if duplicate usernames were prevented at the time the CuRTClient is initialized.
- *Redirecting ALT/CTRL/WINKEY key commands*: currently, when a machine's input is being redirected to another machine, the ALT/CTRL/WINKEY key commands are not redirected. This limitation prevents keyboard actions such as cutting, copying, and pasting from taking place virtually.

- *Preventing ALT/CTRL/WINKEY key commands from executing locally*: currently, when a machine's input is being redirected to another machine, the ALT/CTRL/WINKEY key commands are executed on the local machine. The best solution would probably be to remote all ALT/CTRL/WINKEY key commands except for the hotkey (used to return the local input back the local input device) and CTRL-ALT-DEL, which would create a security concern if it could not be executed on the local machine with the local keyboard.

- *Mouse wheel*: currently, mouse wheel activity and third mouse button activity are not redirected when a machine's input is redirected to another machine. This would just be a matter of setting up some additional mouse hooks and creating a few more messages for client/server communication.

- *Color depth and Windows form invisibility*: to achieve invisibility on Windows forms, which allows the virtual cursors to be cursor-shaped (see Section 5.2.5), the color depth of the local display sometimes needs to be set to 16-bit (most current displays are set to 32-bit. This seems to be a problem that is video-card-dependent, as some video cards do provide invisibility at 32-bit. The easiest solution would be to programmatically set the color depth of the local machine to 16-bit when the CuRTClient starts, and then to change it back to its previous setting (normally 32-bit) when the CuRTClient terminates.

- *Connection lost exception*: currently, the CuRTClient does not throw an exception when it loses the connection with the server. This would be a useful feature for an application using the CuRT system as an API so that it can react appropriately when input redirection is no longer possible due to a dropped connection.

- *ClickGuard opacity*: currently, in order for the ClickGuard to be able to intercept clicks properly, its opacity must be set to a minimum of 2%. For some reason, when the opacity is set to 1% or 0%, clicks are no longer intercepted by the ClickGuard, but instead are applied to any window that is directly behind the ClickGuard window on the desktop. The ClickGuard's background color is set to white, and with a 2% opacity, a very slight "dimming" of the screen takes place whenever the ClickGuard is activated. Future work would include some sort of workaround, either by somehow achieving complete transparency but still preventing clicks, or by developing another method to block mouse activity on the local machine while retaining the ability to listen for it with the mouse hook.

- *XML messaging protocol*: currently, the CuRT system uses a consistent but unstructured protocol for sending messages between the server and the client. An XML messaging protocol would improve the readability of the messages.

6.2 Extended features

- *A more advanced MDE system using CuRT for input redirection*: this extended feature is the reason for which CuRT was written. As explained throughout this paper, CuRT was designed primarily as a toolkit, not as an application. The idea is that other developers can come along later and develop full systems to drive MDEs, using CuRT to add the functionality of cursor/keyboard redirection.

- *Edge detection studies*: several studies could be performed related to the two edge detection systems (speed detection and loiter detection) that are included in the CuRT system. The general questions to be answered with these studies are whether each edge detection system is effective and "natural" to the user, and whether one or both systems should be used in future systems that include cursor/keyboard redirection.

- *Simultaneous keyboard input in two separate windows on the same machine*: though it borders on CuRT losing its universality and application independence, one possible additional feature is to allow simultaneous keyboard input in two separate windows on the same machine using keyboard redirection. The reason that application independence is lost is that there would need to be a direct pipeline from the machine that is sending the keyboard input remotely to the local machine (if not, then there is no way to distinguish the keypresses from the local machine and the keypresses from the remote machine).

- *Customizable cursor icons*: currently, the CuRT system provides the ability to set a virtual cursor icon to one of ten possible colors. The cursor is always the shape of a standard cursor, only the color can be set when the CuRT client is initialized, and the color cannot be changed during runtime. One possible addition to the system would be to allow arbitrary icons to be set as the cursor's icon at runtime. One example of this is to allow a user's photo or some other identifying image to represent the user's cursor in an MDE system. Another possible addition to the system would be to allow all cursors present on a machine at a given time to change based on the currently selected tool (e.g. a pencil tool in a paint program). The problem with this is that a cursor's icon on a machine changes based on one of two events. First, the cursor may change if a tool is selected, as mentioned above. Second, the cursor may change based on where the mouse is currently located (hovering) on the screen. It may be the case that not all cursors should change to the selected tool cursor, as some may be currently hovering on an area where it would make more sense for their cursor to be a different icon. It may be possible to achieve the same hovering for virtual cursors as well, but this would probably require

significant resources. The simplest addition would be to allow arbitrary image changes for any cursor at runtime.

7 Conclusion

The most exciting and promising aspect of the CuRT system is not the functionality provided by the toolkit itself, but rather the potential of future projects, using the CuRT system as an API, to develop MDE systems that take collaboration with multiple people and multiple machines to a new level. Much research has been done on input redirection, and it seems to be one of the necessary components of the collaborative MDE systems of the future. CuRT is simply an imperfect approximation of how many-to-one input redirection would work in a future operating system that truly supports it. The one-cursor-per-machine limitation of current operating systems carries over and limits the CuRT system in several aspects, as described in several chapters of this thesis. Systems cannot be designed effectively without first testing them on real users in simulated environments, and systems using many-to-one input direction would be no exception. Using CuRT and its imperfect many-to-one input redirection capability, future studies might conclude that many-to-one input redirection is important enough to include as a low-level feature of an operating system. This might mean that multiple input devices, representing multiple cursors, can be connected to the same machine, or it might mean that one machine can send its cursor to another machine, across a network, allowing both cursors to interact with the same machine at the same time, with simultaneous clicks, drags, menu selections, etc. The end goal of the CuRT system is for it to have played a part in a push toward allowing multiple cursors on the same machine at the same time, at the operating system level.

8 Appendix

8.1 CuRTClient Class Reference

8.1.1 Detailed Description

The **CuRTClient** class represents one client connected to a CuRTServer. The **CuRTClient** has two main modes of operation, represented by the two different **CuRTClient** constructors. One is the standalone application mode. In this mode, the user manually specifies the connection parameters, and a GUI allows the user to send his cursor/keyboard input to other machines in the system and then return his cursor/keyboard input back to his own machine. To execute the **CuRTClient** in this mode, just run this compiled code (the main method of this class starts the **CuRTClient** in the standalone application mode). The other mode is to use the **CuRTClient** as an API. In this mode, a developer specifies all connection parameters in the **CuRTClient** constructor. Then, method calls in this class can be called to manipulate the client and to interact with other clients.

8.1.2 Constructor & Destructor Documentation

8.1.2.1 CuRTClient ()

This is one of two **CuRTClient** constructors. This one is used in standalone application mode. This one simply starts the CuRTGUI, which then starts the ConnectionBox. The ConnectionBox allows the user to specify the connection parameters manually.

8.1.2.2 CuRTClient (String *newClientName*, String *newIP*, KeyModifiers *newModkey*, Keys *newHotkey*, CursorColors[] *newColorScheme*, int *newCursorMovementResolution*, EdgeDetectionParams *newEdgeDetectionParams*)

This is one of two **CuRTClient** constructors. This one is used the API mode. This one takes in the same parameters that the user specifies manually in the standalone application version. The GUI is started here as well, but the

ConnectionBox is bypassed.

8.1.3 Public Method Documentation

8.1.3.1 static bool AnotherInstanceRunning () [static]

This method counts the number of processes running with the same name as the current one. It then returns true if this result is 1 (meaning that this process is the only one running), or false otherwise. This is used to prevent two instances of the **CuRTClient** from running at the same time.

8.1.3.2 int GetClientIndex (String *name*)

This is an API method that returns the index of the client with the specified username. If there is no client with the specified username, -1 is returned. Note: in the case of multiple clients with the same username, the first matching username is returned.

8.1.3.3 bool [] GetClientIndicesCurrentlyRemotedToLocalMachine ()

This is an API method that returns a boolean array indicating which clients are currently remoting their cursors to the local machine. A value of true at a given index indicates that the client at that index is currently remoting to the local machine, and a value of false indicates that the client at that index is either not currently remoting to the local machine, or is not connected to the system.

8.1.3.4 String [] GetClientNameList ()

This is an API method that returns a String array containing the usernames of all clients currently connected to the CuRT system. For every index where no client is connected, the String "[EMPTY]" is returned.

8.1.3.5 String [] GetClientNamesCurrentlyRemotedToLocalMachine ()

This is an API method that a String array containing the usernames of all clients whose cursors are currently being remoted to the local machine. For every index where the client is either not being remoted to the local machine or is not connected to the system, the String "[EMPTY]" is returned.

8.1.3.6 String GetClientUsername (int *index*)

This is an API method that returns the username of the client at the specified index. If there is no client at that specified index, the String "[EMPTY]" is returned.

8.1.3.7 CursorColors GetCursorColor (String *name*)

This is an API method that returns an enumerated CursorColors value indicating the color of the cursor with the specified client name.

8.1.3.8 CursorColors GetCursorColor (int *index*)

This is an API method that returns an enumerated CursorColors value indicating the cursor's color at the specified index.

8.1.3.9 Point GetCursorPosition (String *name*)

This is an API method that returns a Point value indicating the coordinate position on the local machine of the cursor with the speficied client username. If the client with the specified username is not currently being remoted to the local machine, or there is no such client with the speficied username, an empty Point object is returned.

8.1.3.10 Point GetCursorPosition (int *index*)

This is an API method that returns a Point value indicating the coordinate position on the local machine of the cursor with the speficied client index. If the client at the specified index is not currently being remoted to the local machine, or is not connected to the system at all, an empty Point object is returned.

8.1.3.11 Screen GetCursorScreen (String *name*)

This is an API method that returns a Screen object that represents the screen (for use in multi-screen setups) where the cursor with whose client has the specified username is currently located. If there is no client with that username, or if the client with that username is not currently remoting its cursor to the local machine, a null object is returned.

8.1.3.12 Screen GetCursorScreen (int *index*)

This is an API method that returns a Screen object that represents the screen (for use in multi-screen setups) where the cursor with the specified index is currently located. If there is no cursor at that index, or if that cursors is not currently being remoted to the local machine, a null object is returned.

8.1.3.13 int GetLocalClientIndex ()

This is an API method that returns index of the local client.

8.1.3.14 String GetLocalClientUsername ()

This is an API method that returns username of the local client.

8.1.3.15 CursorColors GetLocalCursorColor ()

This is an API method that returns an enumerated CursorColors value indicating the local cursor's color.

8.1.3.16 Point GetLocalCursorPosition ()

This is an API method that returns a Point value indicating the local cursor's coordinate position on the local machine. If the local cursor is currently being remoted, an empty Point object is returned.

8.1.3.17 int GetLocalCursorRemoteIndexLocation ()

This is an API method that returns an integer indicating the index of the client where the local cursor is currently being remoted. If the local cursor is not currently being remoted, a value of -1 is returned.

8.1.3.18 String GetLocalCursorRemoteUsernameLocation ()

This is an API method that returns a String representing the username of the of the client where the local cursor is currently being remoted. If the local cursor is not currently being remoted, the String "[Empty]" is returned.

8.1.3.19 Screen GetLocalCursorScreen ()

This is an API method that returns a Screen object that represents the screen (for use in multi-screen setups) where the local cursor is currently located. If

the local cursor is currently being remoted, a null object is returned.

8.1.3.20 bool IsLocalCursorCurrentlyRemote ()

This is an API method that returns a boolean indicating whether the local machine's cursor is currently being remoted to another machine in the CuRT system.

8.1.3.21 static void Main () [static]

This is the **CuRTClient** Main method. It simply starts the **CuRTClient** in standalone application mode. To use the **CuRTClient** in the API mode, do not execute this Main method. Rather, create an instance of a **CuRTClient** object and pass it the appropriate parameters.

8.1.3.22 bool RemoveLocalCursorFromRemote (Point *newLocation*)

This is an API method that attempts to remove the local cursor from a remote client. If the local cursor is currently being remoted on a remote client, then it is returned to the local machine and is moved to the coordinates of the new-Location parameter, and true is returned. If it is not currently beind remoted, then false is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.23 Point SendLocalCursorToRemoteMachine (String *username*, Point *startingLocation*)

This is an API method that sends the local cursor to a remote machine, at the specified X and Y coordinates. The current location of the local cursor on the local machine is returned as a Point object. If the local cursor is already being remoted, it is first removed from the remote machine, and then sent to the new remote machine (an Empty point is returned in that case). If the index parameter is the same as the local index, an Empty point is returned.

8.1.3.24 Point SendLocalCursorToRemoteMachine (int *index*, Point *startingLocation*)

This is an API method that sends the local cursor to a remote machine, at the specified X and Y coordinates. The current location of the local cursor on the local machine is returned as a Point object. If the local cursor is already being remoted, it is first removed from the remote machine, and then sent to

the new remote machine (an Empty point is returned in that case). If the index parameter is the same as the local index, an Empty point is returned.

8.1.3.25 Point SendRemoteCursorToOwnMachine (String *cursorName*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine "back home" to its local machine. If the virtual cursor with the specified username is not currently remoted to the local machine, or if no client with the specified username exists in the CuRT system, an empty point object is returned. Otherwise, the virtual cursor is sent back to its own machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.26 Point SendRemoteCursorToOwnMachine (int *cursorIndex*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine "back home" to its local machine. If the virtual cursor at the specified index is not currently remoted to the local machine, or if the client at the specified index is not currently connected to the CuRT system, an empty point object is returned. Otherwise, the virtual cursor is sent back to its own machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.27 Point SendRemoteCursorToRemoteMachine (String *cursorName*, String *machineName*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine to a remote machine. If there is no virtual cursor on the local machine with the specified index, or if there is no client connected at the indicated destination index, or if the specified cursor index is the local index, or if the specified destination machine index is the local index, an empty point object is returned. Otherwise, the virtual cursor is passed to the indicated remote machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.28 Point SendRemoteCursorToRemoteMachine (int *cursorIndex*, String *machineName*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine to a remote machine. If there is no virtual cursor on the local machine with the specified index, or if there is no client connected at the indicated destination index, or if the specified cursor index is the local index, or if the specified destination machine index is the local index, an empty point object is returned. Otherwise, the virtual cursor is passed to the indicated remote machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.29 Point SendRemoteCursorToRemoteMachine (String *cursorName*, int *machineIndex*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine to a remote machine. If there is no virtual cursor on the local machine with the specified index, or if there is no client connected at the indicated destination index, or if the specified cursor index is the local index, or if the specified destination machine index is the local index, an empty point object is returned. Otherwise, the virtual cursor is passed to the indicated remote machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.1.3.30 Point SendRemoteCursorToRemoteMachine (int *cursorIndex*, int *machineIndex*, Point *newLocation*)

This is an API method that attempts to send a virtual cursor from the local machine to a remote machine. If there is no virtual cursor on the local machine with the specified index, or if there is no client connected at the indicated destination index, or if the specified cursor index is the local index, or if the specified destination machine index is the local index, an empty point object is returned. Otherwise, the virtual cursor is passed to the indicated remote machine, and a point object with the virtual cursor's final position on the local machine is returned. If the point parameter is an empty point object, the cursor's new location is set to (0,0).

8.2 CursorOnEdgeEventArgs Class Reference

8.2.1 Detailed Description

This class encapsulates the data members that are set when an edge detection event is fired. This information is returned to the member that receives the event so that an appropriate action (such as moving the cursor to another screen) can be taken.

8.2.2 Public Member Variables

- readonly int **cursorIndex**

  the index (0-9) of the machine that is controlling the cursor

- **readonly String cursorName**

  the user-specified name of the machine that is controlling the cursor

- **readonly Point cursorLocation**

  a point that contains the coordinates (X,Y) of the cursor at the moment when the event is fired

- **readonly ScreenEdges edge**

  an enumerated ScreenEdges object representing the particular edge (top, bottom, left, or right) where the edge detection occurred

- **readonly Screen screen**

  a Screen object representing the particular screen where the edge detection occurred (for use with multi-screen setups)

- **readonly int source**

  an integer representing the type of edge detection that occurred (0 = Speed Detection; 1 = Loiter Detection)

8.2.3 Constructor & Destructor Documentation

8.2.3.1 CursorOnEdgeEventArgs.CursorOnEdgeEventArgs (int *cursorIndex*, String *cursorName*, Point *cursorLocation*, ScreenEdges *edge*, Screen *screen*, int *source*)

CursorOnEdgeEventArgs constructor.

This constructor just takes the abovementioned six values as parameters and assigns their values to the appropriate member variables.

8.3 CursorOnEdgeListener Class Reference

CursorOnEdgeListener.

8.3.1 Detailed Description

CursorOnEdgeListener.

This is a method that is called whenever the edge detection event is fired. This method currently just displays all pertinent information to the console window, but it can be redefined to take more interesting actions.

8.4 EdgeDetectionParams Class Reference

8.4.1 Detailed Description

This class encapsulates the parameters needed for both edge detection systems, Speed Detection and Loiter Detection.

8.4.2 Public Member Variables

8.4.2.1 int EdgeDetectionParams.loiterMovementsNeeded

loiterMovementsNeeded (Loiter Detection) A loiter movement occurs when all of the following are true: 1) either the X or the Y coordinate is one of the edges of the screen (top, bottom, left, or right) 2) a change in cursor coordinate is detected 3) the coordinate that was on one of the edges of the screen remains on the edge (i.e. only the other coordinate changes)

Whenever a cursor movement is detected that is NOT a loiterMovement, the loiterMovement count is reset to 0. If the loiterMovement count reaches the loiterMovementsNeeded parameter, a Edge Detection event is fired.

8.4.2.2 int EdgeDetectionParams.miniJumpSize

the minimum size (in pixels) of one mini jump

8.4.2.3 int EdgeDetectionParams.miniJumpsNeeded

This is the number of mini jumps in a row that are needed to trigger an edge detection. A mini jump occurs when all of the following are true: 1) a change in cursor coordinate is detected 2) the cursor is within one of the zones specified by the zoneSize parameter (see below) 3) the cursor is moving toward the edge of the screen that corresponds to the particular zone 4) the change of position, in pixels, of the cursor's position (the X coordinate in the case of movement toward the left or right edge of the screen, and the Y coordinate in the case of movement toward the top or bottom edge of the screen) is greater than or equal to the miniJumpSize parameter

Whenever a cursor movement is detected that does NOT trigger a minijump, the minijump count is reset to 0. If the miniJump count reaches the miniJumps-Needed parameter, a Edge Detection event is fired.

8.4.2.4 int EdgeDetectionParams.zoneSize

the size (in pixels) of the zone at the top, bottom, left, and right edge of the screen where mini jumps can occur

8.4.3 Constructor & Destructor Documentation

8.4.3.1 EdgeDetectionParams.EdgeDetectionParams (int *miniJumpsNeeded*, int *zoneSize*, int *miniJumpSize*, int *loiterMovementsNeeded*)

The **EdgeDetectionParams** constructor.

This constructor just copies values over from the parameters to the object's member variables. It also makes appropriate adjustments to the values if the user has signified that he wants to disable one or both of the systems (by passing -1 as the parameter). This constructor is to be used by developers using the CuRT system API in order to specify the parameters of the Edge Detection system. The newly created **EdgeDetectionParams** object is then passed into the **CuRTClient** constuctor.

References

[1] P. Baudisch, E. Cutrell, D. Robbins, M. Czerwinski, P. Tandler, B. Bederson, and Z. Zierlinger. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch- and pen-operated systems.

[2] Lior Berry, Lyn Bartram, and Kellogg S. Booth. Role-based control of shared application views. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 23–32, New York, NY, USA, 2005. ACM Press.

[3] Jacob T. Biehl and Brian P. Bailey. Aris: an interface for application relocation in an interactive space. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 107–116, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[4] Jacob T. Biehl and Brian P. Bailey. Improving scalability and awareness in iconic interfaces for multiple-device environments. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 91–94, New York, NY, USA, 2006. ACM Press.

[5] Kellogg S. Booth, Brian D. Fisher, Chi Jui Raymond Lin, and Ritchie Argue. The "mighty mouse" multi-screen collaboration tool. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 209–212, New York, NY, USA, 2002. ACM Press.

[6] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. Easyliving: Technologies for intelligent environments. In *HUC '00: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, pages 12–29, London, UK, 2000. Springer-Verlag.

[7] Olivier Chapuis and Nicolas Roussel. Metisse is not a 3d desktop! In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 13–22, New York, NY, USA, 2005. ACM Press.

[8] Maxime Collomb, Mountaz Hascoët, Patrick Baudisch, and Brian Lee. Improving drag-and-drop on wall-size displays. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 25–32, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[9] Jr. D. Austin Henderson and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, 1986

.

[10] Scott Elrod, Richard Bruce, Rich Gold, David Goldberg, Frank Halasz, William Janssen, David Lee, Kim McCall, Elin Pedersen, Ken Pier, John Tang, and Brent Welch. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 599–607, New York, NY, USA, 1992. ACM Press.

[11] Armando Fox, Brad Johanson, Pat Hanrahan, and Terry Winograd. Integrating information appliances into an interactive workspace. *IEEE Comput. Graph. Appl.*, 20(3):54–65, 2000.

[12] S. Greenberg. Sharing views and interactions with single-user applications. In *Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, pages 227–237, New York, NY, USA, 1990. ACM Press.

[13] Franois Guimbretière, Maureen Stone, and Terry Winograd. Fluid interaction with high-resolution wall-size displays. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 21–30, New York, NY, USA, 2001. ACM Press.

[14] Fredrik Hubinette. x2vnc. http://fredrik.hubbe.net/x2vnc.html.

[15] Dugald Ralph Hutchings, Greg Smith, Brian Meyers, Mary Czerwinski, and George Robertson. Display space usage and window management operation comparisons between single monitor and multiple monitor users. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 32–39, New York, NY, USA, 2004. ACM Press.

[16] Dugald Ralph Hutchings and John Stasko. Revisiting display space management: understanding current practice to inform next-generation design. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 127–134, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[17] Brad Johanson and Armando Fox. The event heap: A coordination infrastructure for interactive workspaces. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 83, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. Pointright: experience with flexible input redirection in interactive workspaces. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 227–234, New York, NY, USA, 2002. ACM Press.

[19] Azam Khan, George Fitzmaurice, Don Almeida, Nicolas Burtnyk, and Gordon Kurtenbach. A remote control interface for large displays. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 127–136, New York, NY, USA, 2004. ACM Press.

[20] Azam Khan, Justin Matejka, George Fitzmaurice, and Gordon Kurtenbach. Spotlight: directing users' attention on large displays. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 791–798, New York, NY, USA, 2005. ACM Press.

[21] J. Chris Lauwers and Keith A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[22] Mary D. P. Leland, Robert S. Fish, and Robert E. Kraut. Collaborative document production using quilt. In *CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, pages 206–215, New York, NY, USA, 1988. ACM Press.

[23] Du Li and Rui Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 246–255, New York, NY, USA, 2002. ACM Press.

[24] Masood Masoodian, Sam McKoy, Bill Rogers, and David Ware. Deepdocument: use of a multi-layered display to provide context awareness in text editing. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 235–239, New York, NY, USA, 2004. ACM Press.

[25] Brad A. Myers. The pebbles project: using pcs and hand-held computers together. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, pages 14–15, New York, NY, USA, 2000. ACM Press.

[26] Brad A. Myers, Rishi Bhatnagar, Jeffrey Nichols, Choon Hong Peck, Dave Kong, Robert Miller, and A. Chris Long. Interacting at a distance: measuring the performance of laser pointers and other devices. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–40, New York, NY, USA, 2002. ACM Press.

[27] Miguel A. Nacenta, Dzmitry Aliakseyeu, Sriram Subramanian, and Carl Gutwin. A comparison of techniques for multi-display reaching. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 371–380, New York, NY, USA, 2005. ACM Press.

[28] J. Karen Parker, Regan L. Mandryk, and Kori M. Inkpen. Tractorbeam: seamless integration of local and remote pointing for tabletop displays. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 33–40, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[29] Elin Ronby Pedersen, Kim McCall, Thomas P. Moran, and Frank G. Halasz. Tivoli: an electronic whiteboard for informal workgroup meetings. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 391–398, New York, NY, USA, 1993. ACM Press.

[30] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. *Lecture Notes in Computer Science*, 2201, 2001.

[31] Jun Rekimoto. Pick-and-drop: a direct manipulation technique for multiple computer environments. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 31–39, New York, NY, USA, 1997. ACM Press.

[32] Jun Rekimoto and Masanori Saitoh. Augmented surfaces: a spatially continuous work space for hybrid computing environments. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 378–385, New York, NY, USA, 1999. ACM Press.

[33] George Robertson, Eric Horvitz, Mary Czerwinski, Patrick Baudisch, Dugald Ralph Hutchings, Brian Meyers, Daniel Robbins, and Greg Smith. Scalable fabric: flexible task management. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 85–89, New York, NY, USA, 2004. ACM Press.

[34] George Robertson, Maarten van Dantzich, Daniel Robbins, Mary Czerwinski, Ken Hinckley, Kirsten Risden, David Thiel, and Vadim Gorokhovsky. The task gallery: a 3d window manager. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 494–501, New York, NY, USA, 2000. ACM Press.

[35] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, 2002.

[36] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, 1986.

[37] Ramona E. Su and Brian P. Bailey. Put them where? towards guidelines for positioning large displays in interactive workspaces. *Interact*, pages 337–349, 2005.

[38] Desney S. Tan, Brian Meyers, and Mary Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1525–1528, New York, NY, USA, 2004. ACM Press.

[39] Masayuki Tani, Masato Horita, Kimiya Yamaashi, Koichiro Tanikoshi, and Masayasu Futakawa. Courtyard: integrating shared overview on a large screen and per-user detail on individual screens. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 44–50, New York, NY, USA, 1994. ACM Press.

[40] Edward Tse, Jonathan Histon, Stacey D. Scott, and Saul Greenberg. Avoiding interference: how people use spatial separation and partitioning in sdg workspaces. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 252–261, New York, NY, USA, 2004. ACM Press.

[41] VNC. http://www.realvnc.com/.

[42] Grant Wallace, Peng Bi, Kai Li, and Otto Anshus. A multi-cursor x window manager supporting control room collaboration. Princeton University, Computer Science, Technical Report TR-707-04, July 2004.

[43] x2x. http://ftp.digital.com/pub/DEC/SRC/x2x.

[44] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 162–171, New York, NY, USA, 2004. ACM Press.