XICE Windowing Toolkit: Seamless Display Annexation

RICHARD ARTHUR and DAN R. OLSEN, Jr., Brigham Young University

Users are increasingly nomadic, carrying computing power with them. To gain rich input and output, users could annex displays and input devices when available, but annexing via VGA cable is insufficient. This article introduces XICE, which uses wireless networks to connect portable devices to display servers. Network connections eliminate cables, allow multiple people to share a display, and ease input annexation. XICE mitigates potentially malicious input, and facilitates comfortable viewing on a variety of displays via view-independent coordinates. The XICE-distributed graphics model greatly reduces portable device CPU usage and extends portable device battery life.

Categories and Subject Descriptors: H5.2 [**Information Interfaces and Presentation**]: User Interfaces—*graphical user interfaces (GUI)*

General Terms: Human Factors, Design, Management

Additional Key Words and Phrases: Nomadic interaction, annex screens, security

ACM Reference Format:
Arthur, R., and Olsen, Jr., D. R. 2011. XICE windowing toolkit: Seamless display annexation. ACM Trans. Comput.-Hum. Interact. 18, 3, Article 14 (July 2011), 46 pages.
DOI = 10.1145/1993060.1993064 http://doi.acm.org/10.1145/1993060.1993064

1. INTRODUCTION

Computing becomes increasingly nomadic as Moore's law continues to push more computing power into smaller devices. Portable devices now have significant computing resources and are becoming primary personal computing devices.

The interactive input and output of personal portable devices are limited by their size. A user could carry around a larger screen, keyboard, and mouse for richer interaction, but not in a handheld form-factor. Keyboards have shrunk to fit small devices but are difficult or slow to use. The screens on small devices cannot provide enough visual output to meet a typical user's needs, even with a lot of panning and zooming.

A richer nomadic interactive experience would be to carry data and applications in a personal device, then annex display servers such as screens and input devices when necessary [Olsen et al. 2001; Paek et al. 2004; Pierce and Mahaney 2004]. Screens are accessible in many places such as offices, kiosks, home entertainment systems (e.g. televisions, projectors, etc.), and conference rooms (Figure 1). Users do not need to carry screens if they can effectively use available screens. Keyboards and mice are inexpensive, so they too can be readily available for use.

This article discusses the XICE (eXtending Interactive Computing Everywhere—pronounced "zice") Windowing Toolkit, which seamlessly distributes the user interface (UI) of applications to annexed displays and directs input from annexed devices to

Authors' present addresses: R. Arthur, Neumont University, 10701 South River Front Parkway, Suite 300, South Jordan, UT 84095, e-mail: startether@startether.com; D. R. Olsen, Jr., Brigham Young University, Computer Science Department, 3336 TMCB, Provo, UT 84602, e-mail: olsen@cs.byu.edu.

DOI 10.1145/1993060.1993064 http://doi.acm.org/10.1145/1993060.1993064

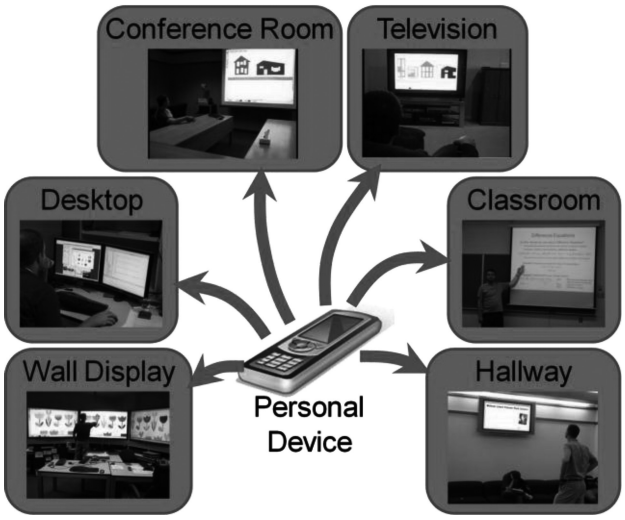

Fig. 1. Display servers a nomadic user could annex.

Fig. 2. Mobile interaction in a stopped car. All application processing happens within the handheld (highlighted).

applications. XICE operates by letting personal devices annex display devices through wireless networks instead of through direct cable connections like Video Graphics Array (VGA) and Universal Serial Bus (USB). The user experience becomes much more flexible when the wireless network is used. This architecture requires one or more *display servers*, each of which includes a processor and software to transmit user input to and show visual output from applications running on a personal device.

In a nomadic environment, users should be able to easily annex screens and input from display servers. These display servers should have a stable annexing protocol. Users should also be protected from malicious display servers. In particular, the protection should address attacks specific to distributing input and output. Software for nomadic users should require minimal overhead for annexing display servers to maximize the personal device's battery life. Display servers vary according to viewing distance and screen size, and applications must adapt to these differences. In addition, multiple people should be able to annex a display server simultaneously.

2. NOMADIC COMPUTING

A nomadic user goes various places and uses a variety of display servers. She may experience three *nomadic situations*: personal device alone, annexed screen and input, and annexed screen alone.

The *personal device alone* situation involves using the personal device (e.g., a smartphone) directly as seen in Figure 2. This setup provides the nomadic user with the

Fig. 3. Desktop interaction. All application processing happens within the handheld (to the right of the user).

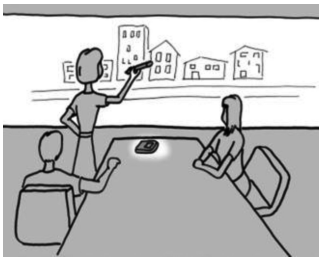

Fig. 4. Collaboration on a large screen. All application processing happens in the handheld.

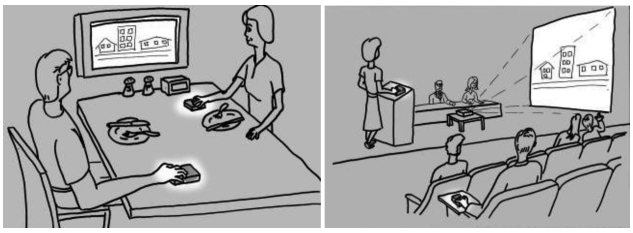

Fig. 5. Collaboration in a restaurant or a public forum. Multiple people can interact, but personal processing happens in the handheld.

requisite mobility but poses serious interaction problems with regard to the personal device's small form factor.

The *annexed screen and input* situation is illustrated in Figure 3. When a nomadic user arrives at her personal workspace, the personal device should annex a mouse, a keyboard, multiple screens, and other available interactive resources. Though applications run on the personal device, her interaction with them should feel like she is using a desktop.

A user may move from her private workspace to a collaborative, company-controlled space that includes display servers like the wall screen in Figure 4. In this environment, screens may be larger and farther away and input devices may differ, but all of the inputs can be fully trusted. Although the environment is different from the personal workspace setup, this scenario is also an annexed screen and input situation.

The *annexed screen only* situation applies when annexed input cannot be trusted. A user may move to a collaborative setting, such as the restaurant or public meeting in Figure 5. Other institutions control the display servers and input devices. In these scenarios, the personal device accepts input only from itself so that annexed input devices cannot be used as attack vectors.

Not only are input threats present in such public situations, but the screen will also be visible to multiple people. Users may want to annex these screens, but will likely

not want certain pieces of sensitive data (e.g. emails, financial data, file names) to be shown so publicly. Either people in the room or the display server itself could steal that data.

In many collaborative situations, multiple people may annex a display server simultaneously. Each user may want to interact with his or her applications independently on the same display (i.e. each person can bring up various single-user applications). Within these multi-user configurations, each user may choose a different nomadic situation. For instance, if a display server only has one mouse, then only one user may annex that mouse's input (the annexed screen and input situation); other users can annex the screen (the annexed screen only situation) and opt to use the input on their personal devices instead. The display server must accommodate these independent requests.

2.1. Solution Requirements

The nomadic situations have seven important challenges that must be addressed by a user interface toolkit for nomadic computing.

—Wireless display connectivity and seamless distribution of application output;
—Disparate display resolutions, sizes, and viewing conditions;
—Input acceptance from a variety of sources;
—Avoidance of attack from annexed input devices;
—Privacy sensitivity for application output;
—Myriad software installations;
—Limited battery life and processing power;
—Shared public display space among multiple users.

2.1.1. Wireless Display Connection. Using a cable to physically tie a personal device to a display is severely limiting. In many situations, including business meetings, such a setup is infeasible. Besides unnecessarily tethering the user to a particular display, other problems with using a cable persist. First, large collaborative displays can easily have more pixels (e.g. 7000×2000) than a single VGA or Digital Visual Interface (DVI) cable can handle. Second, in small handheld form-factors, the size of the monitor connector itself becomes prohibitive. Third, if a display is located some distance away from the user (as with a podium and a projector), the user must find the display's physical cable, or the display's owner must make the connector obvious and useable. Fourth, multiple people cannot use a single display without an awkward and time-consuming cable exchange. In addition, multiple people cannot show and interact with information on the display simultaneously. Fifth, with multiple people frequently connecting and disconnecting, the connector is susceptible to fatigue and failure.

XICE connects to display servers using Wi-Fi and the Internet. This capability provides a highly flexible software-defined display mechanism that can readily adapt to a variety of display services. Because XICE uses the network, it can handle large displays with an arbitrary number of pixels and dimensions, and users are not tethered to a specific location. A network-based protocol also enables multiple personal devices to simultaneously annex a single display server, and a single personal device to simultaneously annex multiple display servers.

2.1.2. Varying Displays. Nomadic users will interact with displays of different sizes, ranging from a smartphone's screen to large wall displays. Nomadic users also experience multiple viewing conditions. A personal device's screen is typically viewed from about 12 inches. For the display at a workstation, the viewing distance is about 24 inches. Conference room viewing can easily range from 6 to 50 feet. Typical 10-pixel-tall

text on a desktop will look miniscule on a high-resolution wall-sized screen in a conference room.

In XICE, each display server is configured with the pixels-per-inch resolution and the normal viewing distance. These two pieces of information describe a new rendering and input coordinate space called *view-independent coordinates* (VIC). VIC enables the application and the toolkit to communicate in terms of user-perceived sizes rather than physical or pixel dimensions. VIC provides the necessary adaptation to make text and application UIs readable in each viewing situation. If the user has poor vision, she can configure her device to automatically scale up the application's presentation.

2.1.3. Accepting a Display's Input. An annexed display server may provide input devices, such as a mouse or a keyboard. The display server offers a full QWERTY keyboard and a typical mouse for a personal device in the desktop situation (Figure 3). If the personal device just uses VGA to connect to a screen, the personal device will not be able to accept input through the same cable; to accept input the personal device must have another connection port, and the user will have to find and connect that cable as well. Because XICE uses Wi-Fi, sending input from the display server to the personal device is straightforward and adaptive.

Applications on a personal device must smoothly and rapidly adjust to each nomadic situation and to dynamically shift input sources. To facilitate quick and consistent annexation, the toolkit—not the application—must manage the input bindings. Some display servers will not have input. Others cannot be trusted, as described in the next section, and must be protected against.

2.1.4. Protecting Nomadic Users. Annexed display servers may be in places that are obviously public, such as large conference halls or mall kiosks, but they may also be in places that seem private, such as hotel rooms. Display servers in all of these locations may be malicious because their owners are malevolent or because previous users have infected the servers with malware. When distributing a UI, any combination of four serious threats can surface: stolen input, false input, stolen output, and false output.

Stealing input or output is an effort to capture the user's sensitive data. Falsifying occurs when the display server either alters user data or otherwise coaxes a user into situations where the display server can steal her sensitive data. XICE takes a large step toward mitigating these threats by preventing the personal device from accepting input from a distrusted display server. In such situations, the personal device becomes the only source of input to applications. To protect the user, the default setting is to always reject display server input. Using an options dialog on the personal device, the user can inform her personal device that the annexed input and/or output of a particular display server is trusted. The UI toolkit (XICE) must explicitly inform applications when devices are distrusted. Applications must also inform the toolkit about sensitive input and output so that interaction can be appropriately redirected based on the trust settings. This cooperative communication about sensitive data and trust does not exist in current toolkits.

2.1.5. Differing Software Installations. For a nomadic architecture to be widely adopted, the display services must be standardized. If servers are embedded in screens or smart keyboards, behavior and protocols must be static. Requiring users to check software compatibility before annexing a server is unacceptable. Many proposed display annexing solutions do not support a lightweight service.

When a user annexes a display, she wants familiar window management behavior. To provide consistency, standards for window management need to be developed. Standards built around human-consumable data types, such as drawings, are more likely

to remain constant than those built around programming environments [Olsen 1999]. The XICE solution is a lightweight service that distributes drawings, not code.

2.1.6. Limited Battery Life and Processing Power. When users operate nomadically, their personal devices are used to annex display servers using a wireless network. To properly annex a display server, the personal device's software must translate rendering commands for the network and transmit them. In addition, the personal device must accept similar transmissions containing input from the display server and translate the commands into input applications accept. The consequent radio and CPU overhead has the potential to greatly reduce the battery life of the personal device.

Radio power requirements are related to distance and usage. In the illustrated scenarios, the distances are short (less than 10 meters), so network latency and radio power requirements are both low. As this article later demonstrates, a protocol that distributes scene-graph changes will further reduce bandwidth demands. In addition, using scene-graphs to push graphics rendering off the personal device onto display servers sharply reduces CPU demands and consequently reduces battery requirements.

The user could plug their personal device into a power source within a room. Such an action would obviate the need for low-power consumption. However, this approach requires that rooms prepare accessible power sources and that users find those sources. Regardless, the standard for nomadic computing should minimize power consumption for situations when devices operate on battery power.

Obviously, many other power management issues affect development for and use of personal devices. This article is concerned solely with the load imposed by the user interface architecture.

2.1.7. Support Multiple Simultaneous Users. Part of why people are nomadic is for collaboration with other people. People need to be able to discuss data while using a shared display space. A display space must be able to support multiple people showing data on it at the same time, and it must support independent input for each user. For instance, with three people at a display space, each user would need an independent pointing device so that he or she may interact independently with his or her own applications.

3. PRIOR UI DISTRIBUTION TECHNOLOGIES

Wireless connectivity necessitates a protocol for transmitting visual information from the personal device to a display. Several existing technologies could be used to distribute a user interface across a network. These solutions fall into three broad categories: distributing data, distributing code, and distributing graphics.

3.1. Distributing Data

One way to interact with data on an annexed screen is to send that data to the display server and have an application there interact with that data. This approach is employed manually via thumb drive in the Dynamo [Izadi et al. 2003] multi-user environment and automatically via central server in iRoom [Johanson et al. 2002]. Sending data to the display server greatly reduces the computing requirements of personal devices by offloading all but the long-term storage requirements. Unfortunately, people often have custom-produced software they want to use on the displays. Expecting all display servers to have exactly the same software or equivalent substitute software is unreasonable. And, any software incompatibilities between the personal device and the display server disrupt the user experience. To function effectively, display servers executing application code must have regular software updates and be compatible with all applications users want to run. This requirement creates a tremendous deployment burden.

3.2. Distributing Code

Another option is that the user could transfer her custom software to the display server for execution. Applications could be transmitted in whole (like Equalizer [2008] or blue-c [Naef et al. 2003]), or in part (like Migratory Applications [Bharat and Cardelli 1997]). This workaround would theoretically allow her to use any display server to interact with data. However, display servers may have incompatible hardware or operating systems for executing her software. In addition, the display server owner may not want anonymous users arbitrarily installing software on his display server.

The user's custom software could be compiled to an intermediate language to become hardware-independent and possibly OS-independent. This is how Flash [Adobe Systems 1996], Java [Gosling et al. 2000], and Silverlight [Microsoft 2011e] operate. But, many applications are simply too large to dynamically transmit to the display server for execution. By comparing installation folder sizes with data file (not video) sizes for most applications, one can see that typical code consumes significantly more hard drive space than individual data files. Transmitting all that application code just to view and edit some data requires a heavy bandwidth load. The excessive startup time for many Flash and Java applets demonstrates this deficiency, which is why Flash, Java, and Silverlight applets are typically designed as small, quickly downloadable applications or are distributed via physical media or long downloads.

Distributing code also introduces major security risks. The display server is given both user data and the code to process it. A malicious display server could easily infect, damage, or propagate that data. In addition, if the platform-independent code is not properly sandboxed, then software transmitted from a client machine could infect the display server.

Alternatively, an application could send only UI-related code to the display server, leaving the core of the application on the personal device. NeWS [Gosling et al. 1989] sends such PostScript [Gosling et al. 1989] commands to the display server. For NeWS to effectively reduce bandwidth and computing on the personal device, the display server must be dynamically programmed by the application. This dynamic programming reintroduces the distributed code problems that enable the display server to control the UI and the user's data. In addition, importing foreign code exposes display servers to potential infection by malicious clients.

Shipping code, data, or credentials to distrusted devices, and then accepting data back, does not offer safe interaction. In addition, such approaches are limited by network speeds because of the volume of data transmitted, so interaction may not be quick.

3.3. Distributing Graphics

Instead of distributing code, data, or credentials and receiving processed data, applications could distribute graphics to the display server and directly receive user input. Then, user data and credentials can be safely retained on the personal device. Two existing methods for distributing UIs to display servers include *rendering-based protocols* and *web-based protocols*.

3.3.1. Rendering-Based Protocols. In a rendering-based protocol, personal devices send either raw pixels or rendering commands to the display server. Systems such as X-Windows (X11) [Scheifler and Gettys 1986], PostScript [Gosling et al. 1989], Virtual Network Computing (VNC) [Richardson et al. 1998], and Remote Desktop Protocol (RDP) [Tritsch 2003] utilize rendering-based presentation. X11, VNC, and RDP execute an application on one computer and render it on another. Remote rendering is feasible as long as compatible, standard server software is installed on the display server. X11, PostScript, and VNC provide useful architecture examples for nomadic computing and

have been stable standards for some time. These protocols are stable because each is designed around a simple graphics model.

The core issue with existing graphics distribution technologies is that they assume the machine running applications has plentiful resources and that the display server has limited resources—only enough to show pixels and do simple processing. The situation is actually the reverse for nomadic users—the personal device has limited resources and the display server has much greater resources. These technologies all assume a single user interacting at the display server but nomadic computing frequently involves multiple users interacting simultaneously. These assumptions generate challenges for nomadic computing in four areas: bandwidth, CPU power, trust, and multi-user support.

X-Windows and RDP convert drawing calls into network messages that are sent to the display server and rendered as pixels. However, as this article shows in Section 5.4, X11 and RDP can generate a high volume of network traffic. A simple scroll operation, for example, causes the draw commands for an entire window to be resent. This function not only incurs bandwidth and radio power costs, but also imposes a large computational burden on the personal device, which executes draw commands to rerender the presentation for each scroll movement.

On Windows Vista, RDP allows users to connect to Network Projectors [Microsoft 2011d], which are RDP-specific display servers. Users annex Network Projectors which then becomes an extension of the user's desktop, where she can use her mouse to drag windows to the projector. However, the projector can only support a single user, and RDP imposes a rendering load similar to X11 on the personal device.

3.3.2. Pixel-Based Protocols. VNC copies pixels from the personal device to the display server. VNC's pixel-based model is a highly stable protocol because a purely pixel data structure does not change. But VNC requires much more bandwidth than X11 in many cases (as will be shown in Section 5.4), and the personal device incurs the rendering costs. Mechanisms in the VNC protocol mitigate net traffic but impose additional computational burdens on the personal device. Pixel-based solutions like VNC also react poorly to the varying screen sizes and resolutions encountered in nomadic computing. If the user annexes a large screen via a smartphone, then copying the pixels from a smartphone's screen to the annexed screen is not satisfying. Copying the window from an off-screen buffer can alleviate this issue, but can consume significant RAM and processing resources on the phone, especially when rendering to large display spaces. The application shown on the shared screen needs to render differently to take advantage of the added screen space.

X11, VNC, and RDP always accept input from the display server, rendering the personal device vulnerable to malicious exploitation. X11 and VNC are known for introducing security holes that must be carefully protected. One such protection mechanism that was developed to address such vulnerabilities is the "xhosts" command which limits the machines that may initiate connections to an X11 display server. To increase interactive security, the devices providing application input must be restricted.

X11, VNC, and RDP do not support multiple users simultaneously interacting on the display space: neither of the situations illustrated in Figure 5 are possible. In particular, the desired system must support multiple people interacting with the display server, with each user supplying output and input from their personal devices. If one of these technologies were chosen, only one user could interact with applications shown on a display server at any one time, there may only be only one mouse cursor on the display, and all conflicts must be handled socially.

Some technologies use RDP or a custom RDP-Like protocol, such as WebEx [Cisco 1997], GoToMyPC, GoToMeeting [Citrix 1997] and MaxiVista [Bartels Media 2011].

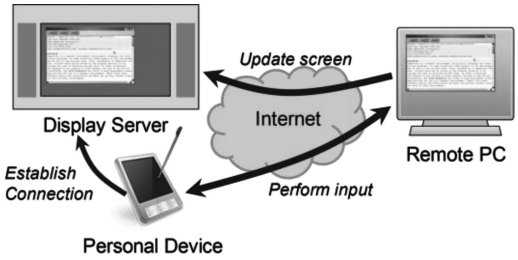

Fig. 6. Oprea et al. annexing configuration. The personal device brokers the connection between the remote PC and the display server. The personal device provides pointing input.

These technologies may add some performance enhancements to RDP, but their annexing model is identical to RDP and is insecure because credentials are entered into foreign machines and the user's PC is under the complete control of the foreign machine.

There are a number of other technologies that use VNC, some variation thereof, or a custom protocol that transmits frame buffers, to implement their interactive experiences, including IMPROMPTU [Biehl et al. 2008]—this implementation detail was discovered via discussion with IMPROMPTU's authors—LivOlay [Jiang et al. 2008], WinCuts [Tan et al. 2004], Lacome [Liu 2007], and Reflect [Argue 2007]. These technologies use VNC because it is a simple, stable, cross-platform protocol, with source code available for any platform. Unfortunately, because VNC is a passive process separated from the applications it captures, it limits a window's size and an application's expressiveness. Because VNC renders on the client machine before transmission, the size of a window and the amount of information shown is limited to the size of the screen buffer. Annexing a large display server provides no real benefit to users with small screens on their personal devices. Because VNC is passive, applications cannot have two windows shown on two separate screens. For example, if a spreadsheet application running on a handheld device could participate more fully in the UI transmission process, then the spreadsheet could show a UI on the handheld which is tuned for that handheld's screen while simultaneously showing a much larger, fuller version of the spreadsheet on an annexed screen. This problem surfaces in reverse with Sharp et al. [2006]: only a portion of the shared screen can be seen on the personal device, creating a limited experience.

Oprea et al. [2004] has all the same problems discussed relative to using VNC. However, it does lead toward a better nomadic interaction model. In this case, some computing and input is provided by the personal device. The personal device brokers a connection between a remote computer and the display server, as illustrated in Figure 6, and then provides text and pointing input to the remote computer. The personal device provides a soft keyboard and is enhanced with an optical mouse to provide pointing input. The user retains control over input to his applications. XICE incorporates the input separation and application control models.

3.3.3. Web-Based Protocols. In a web-based protocol, Hypertext Transfer Protocol (HTTP) and Hypertext Mark-up Language (HTML) are used for application transmission and presentation. A web server sends an HTML version of application output to a web browser. The web browser then renders the HTML to the screen. HTML is an attractive option, because web technologies are so well-known and prevalent. However, web technologies have several drawbacks, including the following five. First, not all web browsers render identically. Second, HTML versions 1 through 4 do not have general drawing capacity while HTML 5 requires JavaScript [Flanagan 2006] to support generalized drawing. Regardless of support for generalized drawing, HTML does

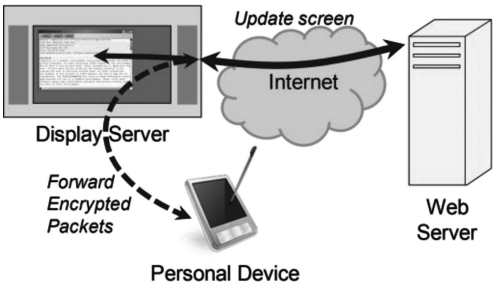

Fig. 7. Mobile Composition web-based display annexation. The personal device provides temporary credentials the display server uses to connect to a web server.

not handle varying display sizes, resolutions, and viewing distances. Without generalized drawing and support for varying display setups, many interactive techniques become difficult to implement. Third, JavaScript implementations are not consistent between browsers. Fourth, JavaScript requires a programmable display and reintroduces many of the security problems related to distributing code (see 3.2). AJAX is becoming more popular as a development technology, but it relies on JavaScript and frequently distributes user data to the browser for manipulation. Fifth, web technologies are difficult to program, because they split code across languages to distribute the UI between the display server and the web server.

Despite the five problems introduced by using HTML, many researchers are exploring how HTML would be used in nomadic environments. Each of these alternatives creates additional obstacles.

There are two basic approaches to using HTML as a network UI distribution technique: the web server and personal server models. The web server model is a familiar paradigm of just logging in to websites via a display server. The personal server model treats the personal device as a web server, so all applications on that device expose their UI as HTML, which may be transmitted to a display server for rendering.

3.3.3.1 Web server HTML model. A frequently suggested option would be to simply use websites for all processing instead of carrying a personal device. To manage personal data, the user would login to her favorite websites at a display server. In this situation, the web browser would play the role of a display server and the websites would be analogous to the personal device. Relying exclusively on the web is colloquially referred to as "living in the cloud". The current interaction model for living in the cloud requires display servers to have an acceptable web browser installed. However, just using a web browser for personal computing on display servers introduces several problems. The user must use a browser that is not configured per her preferences (e.g. the display server may have plug-ins or JavaScript disabled, reducing the user's preferred experience), and she must remember all her favorite sites (which she usually just bookmarks) and usernames and passwords (which are usually stored by the browser). Most importantly, she would have to provide her credentials and interactive input to a foreign device, rendering her vulnerable to identity theft [Sharp et al. 2008]. Display servers would be required to supply input devices which they may not have or which may not be convenient for the environment.

One approach to dealing with security is Mobile Composition [Sharp et al. 2008]. As illustrated in Figure 7, the personal device annexes a display server to view specially-crafted web pages. The web pages may contain special encrypted sections which only the personal device may decrypt and view. This approach helps protect the user's data but requires significant changes to all websites which process sensitive information.

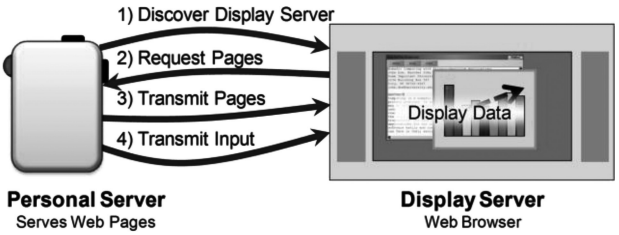

Fig. 8. Personal Server connections. The hand-held personal device supplies web pages to a discovered display server. It also provides simple scrolling and selection input while the display server provides no input.

Such an approach requires significant changes to large portions of the Internet and is likely unworkable.

It is clearly not suitable to rely on display servers having a web browser, the correct plug-ins, being virus-free, and safe for sensitive input. XICE can provide a more effective interaction model for living in the cloud by merging the browser with a network UI distribution framework. If the user's web browser is implemented in XICE, then she can always use her favorite browser wherever she is, with her plug-ins, even if the browser or plug-in is not installed on the display server. This user always has her browser settings available and can safely and securely enter input (particularly credentials) via her portable device even if she is browsing at a distrusted, public display.

3.3.3.2 Personal server HTML model. Alternatively, the personal device could act as the web server and produce a UI in HTML for an annexed display server's web browser. Handheld devices can easily host a web server, as demonstrated by the Personal Server [Want et al. 2002], which is illustrated in Figure 8. The handheld Personal Server wirelessly discovers display servers which are automatically annexed. The Personal Server transmits web pages to the display server, and provides a jog wheel for scrolling through links on the page, and two buttons for navigating via selected links. However, the personal-device-as-web-server design still suffers from the HTML-induced deficits including insufficiently rich interaction and several other issues.

Furthermore, for nomadic computing to use HTTP, connections must be established in a direction that is counter to HTTP's design: currently, a device rendering HTML may request that HTML from a server, but the web server (i.e., the personal device) cannot forcibly push that HTML to a device for rendering. Consequently, the display server must initiate the connection to the personal device to request HTML. As a result, either the connection happens automatically or manually. If it happens automatically, as the Personal Server implements, then the user is exposed to significant security risks because the choice to share information is no longer under the user's control: it is under the display server's control. On the other hand, if the user must establish the connection manually, then she must physically interact with the display server to supply it with the personal device's IP address or URL. This second option creates confusion for the user because her device's IP address is different at each location because that portable device is physically mobile and is likely on a different network with a different IP address.

In addition, the Personal Server automatically trusts input from the display server, so the display server could select any pages from the personal device to be shown or input any data to the personal device. Website developers must proactively work to protect their applications from malicious input [Howard and LeBlanc 2003], creating a heavy developer burden. What is preferred is a system that more proactively protects software from malicious input.

An interesting approach is SessionMagnifier [Yue and Wang 2009] which is a combination of Mobile Composition [Sharp et al. 2008] and Personal Server: the personal device serves up web pages that have been sanitized for the display server. The sanitization process ensures that input happens on the personal device, and that the display server only receives URLs that point to the personal device: no cookies, no URLs to the pages the user is browsing to, etc. Even though this approach still suffers from the five problems of HTML (especially because SessionMagnifier requires JavaScript to be enabled), it does approach data sensitivity in an important way: provide applications with a way to filter out sensitive data. XICE incorporates this sanitation in its privacy implementation: an application can produce a UI using widgets that automatically sanitize the output for a distrusted display server.

Although HTML needs some changes to its standards to reduce the five problems, there are three key observations about the web and nomadic interaction that encourage a new, simple rendering standard: distance, styling, and layout. First, the web is designed for large distances—both in physical distance and in the number of network hops—between the server and the browser and high latency. But, nomadic interaction has short distances—physical and hops—and low latency. A large reason to have JavaScript or Applets is to prevent round-trips to the display server, thus reducing latency and increasing interaction time. With short distances and low latency, preventing round-trips is no longer necessary. Second, CSS is designed to add styling to web pages so that people in different situations can view the same content differently. However, with applications that show on a single display space, such styling is not necessary. Third, the layout problem addressed by CSS can be handled by the "server" (personal device) instead of the "browser" (display server). With a sufficiently consistent rendering framework and low network latency, users can have a richer interaction so these techniques for dealing with high latency are no longer necessary and complicate the programming model.

4. THE XICE PROTOCOL

No existing UI distribution technology sufficiently meets nomadic computing needs. Each system that has some promise is missing several other key pieces. With all of these issues, the authors decided to build a new framework from scratch. For the framework to be successful, applications must be re-implemented, but many systems require applications to be rewritten, such as the iPhone [Apple 2010a] and Android [Google 2011] platforms.

A new architecture is needed that puts the personal device in control. A user needs to carry "his world" with him: his data, his applications, his settings, all in his own way. This architecture needs to expand applications from the tiny screen space on the personal device to the large space afforded by the screens he encounters. This expansion needs to take place in the context of being efficient on computational and network burdens. The framework must also allow multiple people to interact simultaneously in the same display space. The framework must also address the issues of trust as users more-frequently interact with foreign devices. For example, the iPhone addresses trust by locking up everything: each application lives in its own isolated partition. However, this is a limiting approach to trust so a broader mechanism is needed.

XICE was developed specifically to minimize the overhead of network traffic and CPU usage while distributing the UI so that smaller devices may be better supported in nomadic experiences. The theory raised and confirmed in this paper is that a scenegraph (described in the Section 5) drastically reduces CPU and network loads, which opens up avenues for smaller devices to present large interfaces on annexed display servers.

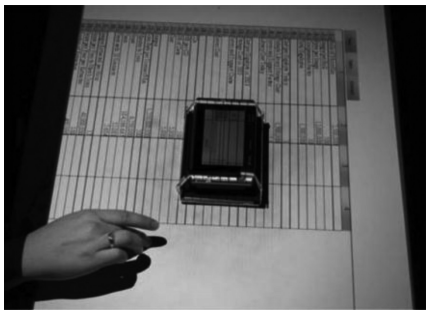

Fig. 9. Handheld spilling: A UI is shown both on a handheld and tabletop computer. The user may interact directly with data through the handheld, and may scroll via the tabletop.

XICE is intended as an experimental framework to test the integration of several existing technologies with some newer technologies, as well as to design new interaction techniques. One such new technique is Handheld Spilling [Olsen et al. 2007] where a UI from a handheld device is shown on both a table-top screen and a handheld. As shown in Figure 9, the UI is synchronized between the handheld and tabletop so that the portion shown on the handheld aligns with the window shown on the tabletop. The user interacts directly with data via the handheld's screen and navigates by scrolling the UI via the table-top (often with her nondominant hand). The handheld presents a spatially limited interaction area, but the tabletop may be an untrustworthy device. Spilling is beneficial because the user can see more information via the tabletop while the user's software is protected from potentially malicious input by interpreting all incoming input as panning motions rather than input that could alter user's data (potentially injecting malicious data). Another new technique is the MousePuter, which is described in 6.2.1.2.

There are two large facets to XICE's implementation: the underlying protocol that communicates between a personal device and a display server, and the windowing toolkit that performs window management and supplies common UI components such as window decorations, buttons, labels, text boxes, and system-supplied dialogs. Although several examples and figures within this article discuss aspects of the windowing toolkit, this article focuses on the underlying protocol. The windowing toolkit is not as important as the protocol: the protocol has stayed fairly stable across experiments while the look and feel of the toolkit has changed, sometimes drastically. Consequently, this article evaluates the protocol, not the look and feel.

Concentrating on the protocol elucidates the goal of integrating a wide variety of devices into the nomadic experience. Expecting all devices to conform to a new windowing toolkit is unreasonable: each device's toolkit has its own strengths that its users prefer. But, expecting a large subset of devices to be able to support a simple protocol is reasonable, especially because so many devices support networking protocols such as HTTP and HTML.

The XICE protocol has two major pieces which provide a rich and safe nomadic experience: the graphics engine and the input framework. The graphics engine provides a large decrease in network usage when distributing a UI. The input framework adds safety via *input redirection*: the personal device can supply all input to windows the user shares on annexed displays instead of accepting input from the annexed display's input hardware. A detailed discussion of the graphics engine is in Section 5, and the input framework is discussed in Section 6.

The XICE protocol is implemented in Java 6. Java was chosen primarily because garbage collected platforms facilitate rapid software development, but Java was also

chosen because of its cross-platform abilities. In addition, version 6 brings developer tools such as annotations and generics (from version 5) and tighter integration with a platform's UI-toolkit (e.g., window transparency).

Although the protocol is currently implemented in Java, the protocol is designed to be implemented by other languages on other platform. The public interface exposed by the UI transmission protocol is platform independent. To demonstrate this, XICE has also been implemented in C# [Microsoft 2000].

5. RENDERING IMPLEMENTATION

The following specific principles guided development of the graphics engine within XICE. First, graphical presentations should be simpler to program than the traditional damage-repaint cycle, reducing development time for an application. Next, the rendering system should be seamless across devices, so developers need not consider whether a window is rendered on the personal device or on an annexed screen. Third, the UI should scale to be easily readable regardless of the display's pixels per inch (PPI) resolution and the user's viewing distance, and the developer should not need to consider either of these. Finally, the rendering system should handle large screens containing numerous pixels, but only incur a minimal load on the personal device. The scene-graph architecture was selected for its superior ability to meet these criteria.

5.1. Automatic Rendering

At the core of XICE is the scene-graph or *presentation tree*. (Those familiar with scene-graphs may skip ahead to Section 5.2.) The presentation tree is modeled after technologies such as Graphical Kernel System (GKS [ANSI 1984]), Programmer's Hierarchical Interactive Graphics System (PHIGS) [Shuey et al. 1986], Jazz and Piccolo [Bederson et al. 2004], and Windows Presentation Foundation (WPF) [Petzold 2006]. In these technologies, an application builds a graph of draw commands that the toolkit renders onto a display. Scene-graphs are basically display lists structured in graph format. Scene-graphs were first developed with GKS and PHIGS to render 2D and 3D scenes across a network. HTML—especially when represented as a Document Object Model within a browser—is a more popular, but less consistent, scene-graph.

With the scene-graphs in WPF, Jazz, and Piccolo, interactive *widgets*—such as buttons or text boxes—are embedded in the scene-graph. These widgets may contain other widgets as part of the graph, and may supply rendering primitives. So, for these technologies, the scene-graph provides structure for rendering, widget containment, and input dispatching. XICE also follows this approach.

Traditionally widgets are rendered using the damage-repaint cycle. Each widget possesses a widget model, produces visual output, handles input, and raises events. For example, a button widget would have Boolean variables to track its enabled and up states and a string variable for the label. To change its visual output, the widget must inform the windowing toolkit (damage), and after all immediate events that might affect the widget's visual output have been processed, the windowing toolkit sends a redraw (repaint) event to the button. Continuing our button example, as a mouse down occurs on the button, the button changes to the down state and then executes the damage-repaint cycle. Next, when a mouse up is received, the button changes back to the up state, raises a "clicked" event, and executes the damage-repaint cycle again.

A significant benefit of scene-graphs is that the toolkit automatically renders pixels from the instructions in the graph, and changes to the graph are automatically rerendered. Figure 10 illustrates a scene-graph and the output that is rendered from it. When the scene-graph is altered—in this case, by changing the circle's fill color to yellow—the rendering engine tracks those changes and rerenders the altered portions of the UI.

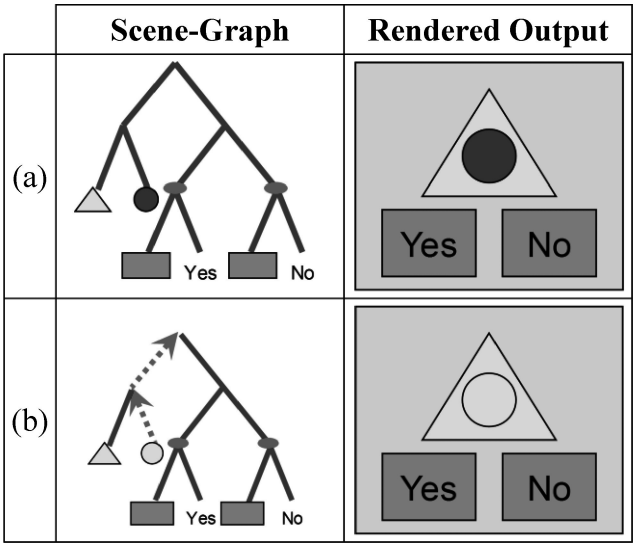

Fig. 10. Scene-graph rendering: (a) shows the original graph, while (b) shows how changing the circle's fill color to yellow causes notifications to travel up the graph and affects the rendered output.

By automatically rerendering the UI's scene-graph changes, code can be minimized, and end developers can be removed from the damage-repaint cycle. Rather than recreating the UI through draw calls each time the paint portion of the cycle happens, developers can create the UI once with the presentation tree, and then update the tree with changes. Piccolo allows developers to perform custom rendering, improving performance in some places; but, custom rendering using Piccolo necessitates the damage-repaint cycle. On the other hand, Jazz removes the damage-repaint cycle, simplifying rendering for developers. XICE does what Jazz does: applications maintain a tree of draw commands instead of calling rendering methods.

The presentation tree represents draw commands—such as draw a line, draw a rectangle, or draw an ellipse—as *nodes* within the tree. Leaf nodes represent drawing primitives, while interior nodes can apply simple effects to those drawing primitives. For instance, the Transformer node applies an affine transform to all child nodes and the Clipper node clips the rendering of its child nodes. Interior nodes can show or hide their children. Nested interior nodes accumulate their effects. For example, embedding a Transformer node with a rotate transform inside a Transformer node with a scale transform causes all leaf nodes within the inner Transformer node to be both scaled and rotated. Embedding a Clipper node inside a Clipper node causes the clipping effects to be accumulated on the child nodes of the inner Clipper node.

The XICE rendering frameworks supports the graphical primitives supplied by the Java Graphics2D object. These primitives include lines, rectangles, ellipses, curves, gradients (linear and radial), images, audio and text. In addition to clipping and transforming, XICE supports transparency. XICE does not support video, but that is because the Java Media Framework (JMF) [Oracle 2011b] does not support video well. If the rendering engine were implemented in WPF, it would easily support video.

Transforming and clipping support not only scrolling, but also rotated windows in DiamondSpin [Shen et al. 2004] and skewed windows in Metisse [Chapuis and Roussel 2005]. XICE does not support the extended zooming architecture of Pad [Perlin and Fox 1993], Jazz, or Piccolo.

5.2. Seamless UI Distribution

Seamless UI propagation to an annexed display server is critical to a simple nomadic architecture. Scene-graphs within XICE are seamlessly propagated over a network without involving the application. When the presentation tree is rendered on an annexed display the entire tree is serialized to the display server, and the display server maintains a copy of that tree. As the application changes nodes within the presentation tree on the personal device, the altered portions of the tree are serialized to the display server, and the display server's copy of the tree is updated.

To accomplish this seamless UI distribution, the serialization protocol has four key aspects: primitives, messages, graph nodes, and widgets.

5.2.1. Serialization Primitives. XICE uses a standard TCP/IP connection to transmit data between a client and a display server. The protocol for serializing and transmitting that data is simple and easily implemented in other languages and frameworks. A display server was built in C# and easily processes such data.

Within XICE there are seven different serializable primitives: integer, double, byte-array, string, start, end, and message. Integer, double, byte-array, and string use standard high-byte first or UTF encodings to transmit the data. Start, end, and message are XICE-specific identifiers.

Before transmitting any primitive, a single-character prefix is transmitted as a 2-byte encoding. Then the data follows that prefix. In the case of the byte-array, the length of the byte array follows the prefix as an integer before the contents of the byte array are transmitted. In the Java implementation, all values are transmitted using functions on the DataOutputStream [Oracle 2011a] object. A C# decoder was a straightforward, simple implementation.

5.2.2. XICE Messages. The primitives are available so that messages may be easily transmitted between machines. When writing out a message, XICE first writes out the identifier for the "message" primitive. Then, the message contents follow as a sequence of primitives.

XICE messages are predefined for the framework and are not expected to be created by end developers. Message objects are designed with both a serialize and a deserialize method, and the framework developers kept the two methods coordinated such that each primitive written has a matching read.

5.2.3. Graph Nodes. Most messages have a fixed number of properties, so coordinating the serialize and deserialize methods is a matter of reading in data in the exact same sequence and type it was written out in. However, graph nodes may contain an arbitrary number of properties of arbitrary types and an arbitrary number of children.

The properties and children are written out as two separate groups. The serialization mechanism uses the start and end primitives to track the beginning and end of the groups of properties and the groups of children, similarly to how Java, C, and C# developers use curly braces to delimit a group of code.

The scene-graph nodes are transmitted at specific times relative to user interaction. The entire scene-graph is transmitted the first time that presentation tree must be rendered by a display server. After that initial transmission, changes to the scene-graph are transmitted in batches.

To transmit changes in batches XICE integrates with the application's event dispatching loop. Every time the event queue is emptied (i.e., all the pending events are processed by the application) XICE inspects each scene-graph for changes. If a scene-graph has changed then those changes are collated into a single message which is transmitted to the display server.

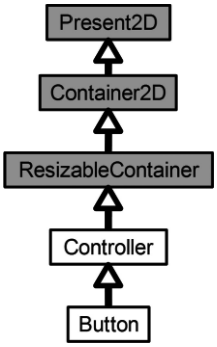

Fig. 11. Type hierarchy of the XICE-supplied Button class. The gray boxes are the widgets types that a client and server are both aware of.

After the changes are collated and transmitted, XICE clears the presentation tree of all change tracking flags.

5.2.4. Serializing Widgets. In XICE, as with Piccolo, interactive widgets are implemented as subclasses of interior nodes in the presentation tree. Embedding widgets within the scene-graph merges the geometry of presentation rendering with the geometry of input handling.

XICE offers nothing novel with regard to the basics of the serialization mechanism, so a description of that process is omitted from this paper. However, because widgets are embedded within the tree the widget serialization process needs to be detailed.

Standard serialization techniques (e.g., Java and .NET [Microsoft 2011c] serialization) require the same class structure to exist on both the sending and receiving ends of serialization. In addition, all private fields of a class are serialized. But, the private fields of interactive widgets can contain sensitive data (e.g. email addresses, usernames, or the password a user enters into the password box) that should not be sent to a display server. The display server renders widget output, but does not execute widget code, so serializing the entire widget is inefficient. Developers may also create novel widgets unknown to the display server, which necessitates shipping novel class structures to the display server before serializing the widget. To minimize security risks and to simplify the architecture, display server implementations should not adapt to class structures of new applications.

The XICE serialization protocol minimizes risk by transmitting *display serialized nodes*: a safe subset of presentation node classes that both the personal device and display server understand. When serializing a widget, its type hierarchy is searched until a display serialized node is found. Then, the properties from that node are serialized to the display server. Consequently, a display server can support the graphical output of any widget without receiving its class definition, and the widget's private data will not be automatically transmitted to the display server. Consider the simplified type hierarchy of the Button widget shown in Figure 11. The gray boxes represent types within that hierarchy that are display serialized nodes. When the serialization mechanism encounters the Button in a scene-graph, it first looks to see if Button is a display serialized node. Because Button is not display serialized, the parent type, Controller, is inspected. It also is not display serialized. However, ResizableContainer is, so the display server is informed of the presence of a ResizableContainer, and all the properties from the ResizableContainer type are serialized to the server.

Notice that even though the XICE windowing toolkit provides the Button type, the ResizableContainer is the type actually transmitted to the display server. The Button

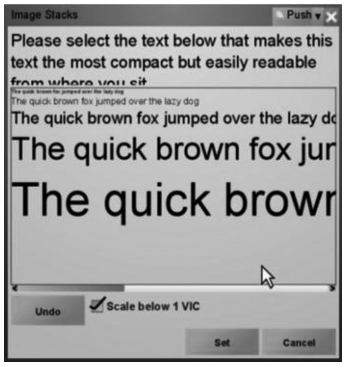

Fig. 12. VIC configuration program. A display server's owner uses this program to configure the display server for typical viewing by iteratively selecting the ideal text size for viewers.

type also performs no rendering on its own: it has child nodes which represents all the graphical elements that represent the output of that button. After serializing the ResizableContainer type, all the child nodes from the Button are also serialized. Consequently, the Button renders identically on the display server. From the display server's perspective the Button appears simply as a ResizableContainer (which is a container with Bounds) with several rendering primitive nodes as children, but none of the button functionality. Not transmitting the Button type is an intentional design decision which keeps the XICE protocol simple, and allows the protocol to be implemented in other languages and on other platforms.

5.3. View Independent Coordinates

Nomadic users will encounter different display and viewing situations. Regardless of the situation, information the user shows on an annexed display must adapt to the display's size, resolution, and position, and produce a readable, interactive interface.

XICE solves the screen adaptation problem by having all applications use view independent coordinates (VIC). Pixels per inch (PPI) have no real perceptual meaning. Instead of PPI, visual perception is defined in degrees of visual arc (DVA). For projectors, PPI, zoom lens, and viewing distance must all be taken into account.

Employment of VIC is not merely an adaptation to the resolution of a display. VIC is determined by what the user sees rather than just what the hardware can show. 10 VIC is defined as the size of normal, comfortably read text. When viewed from 24 inches, a 10-point font (which is 0.1 inches high) is comfortable to read for most people. This font viewed at this distance is approximately 0.3 DVA. So, 10 VIC equals 0.3 DVA. This model is simple for programmers to understand because 10 VIC is conceptually similar to 10-point font. Originally, distances in XICE were defined using DVA, but programmers were confused about how large 1 DVA would be, so they guessed. The VIC system is much easier to explain and to use effectively.

People annexing the display server do not need to configure it. Every XICE display server has the VIC configuration program shown in Figure 12.

An owner runs the VIC configuration program once to configure the display server for use. This program shows text of various sizes on the display and asks the owner to pick the smallest text he can comfortably read from a typical viewing distance. As the owner chooses text, a different set of text with sizes near the selected text are shown. The owner iteratively picks text until he likes the selected size. The program then calculates the appropriate VIC for the display based on the chosen text.

Instead of picking a comfortable font size using the VIC configuration program, the display server owner could rely on the VIC measurement program to calculate a reasonable estimate. The application displays a 200-pixel by 200-pixel square. The owner is asked to measure and enter the width W and the height H of the square on the screen, and the user's typical viewing distance D. The units of measurement are unimportant as long as the same units are used for all three values. In places like conference halls, viewers are located at different viewing distances, so a judgment call must be made regarding the appropriate view distance for common uses of the display. Given W, H, and D, and the fact that 10 VIC is 0.3 DVA, the program can compute scale factors for X and Y.

$$s_x = \frac{0.3}{10} \times \tan(1) \times D \times \frac{200}{W},$$
$$s_y = \frac{0.3}{10} \times \tan(1) \times D \times \frac{200}{H}.$$

Two separate scale factors are used, because displays do not always use square pixels.

When a user connects to a display server, his personal device is informed of the display dimensions in VIC. The toolkit software on personal device then knows the bounds of the screen. Applications can use this toolkit information to adapt themselves to the available display space. XICE does not address the UI adaptation problem [Nichols 2006]; XICE merely informs the application of what the visual parameters are. The application can then adapt using any algorithms or techniques developers may choose.

In some situations, the viewing distance could depend on the distance the user is connecting from. For instance, a user at one end of the room may want to interact directly with the screen, while another user may want to interact with his data from a distance. The two viewing distances could be configured dynamically. Tools that can measure distance such as the Wii [Nintendo 2011] or the Kinect [Microsoft 2011a] could easily measure user distance and configure the VIC automatically. How to best approach this is left as an area for future research.

5.4. CPU/Network Load Evaluation

A major issue with distributing the UI from a personal device to a display server is the battery drain. Battery drain comes primarily from the backlight, CPU, and the radio. An interactive architecture can affect the CPU load and the radio, but not the backlight. The process of generating and transmitting a UI to a different machine influences the CPU and radio usage. Although a user could potentially plug their device into a wall outlet (or other power source) to gain the requisite processing power, such an approach then tethers the user to a limited range of motion. An ideal approach would be to operate for longer periods of time exclusively on batter power. A rendering engine that minimizes both CPU and radio usage will help extend battery life when annexing a display server. This paper contends that the average interactive application spends the majority of its CPU cycles rendering pixels and that using a scene-graph can greatly reduce processing time and network usage by offloading pixel rendering to another device, especially when compared to other network UI distribution technologies such as X11, RDP, and VNC.

Consider a word processor displayed in a 600- by 400-pixel window. Typing the letter "y" will cost at most 10,000 instructions to make room in the document and insert the letter "y". If the "y" is placed in the middle of the document, half of the pixels in the window may need to be repainted for word wrapping and new lines. Suppose each pixel takes a minimum of 12 instructions to render for a total of 1.44 million instructions to repaint half the window. Less than 1% of the CPU cost is in actual document modification.

Now consider a 10-column by 30-row spreadsheet displayed in the same 600 × 400 window. If a change in one cell causes all other cells to be recalculated at a cost of 100 instructions per cell, a total of 30,000 instructions must be processed for that one cell's value change. With 2.88 million instructions necessary to repaint the entire window, the cell recalculations amount to approximately 1% of the cost of repainting the window.

These two simple examples show that in modern graphical applications, the CPU cost of updating the application's model is far overshadowed by the cost of painting pixels. In the examples, only 600 × 400 windows were used. If the personal device distributes these windows' pixels via VNC or VGA, rendering on the personal device may be reasonable. However, with wall-sized displays, the personal device may manage several million pixels, greatly increasing the personal device's effort to calculate pixel values.

XICE addresses the multi-million-pixel rendering problem by imposing only presentation tree manipulations on an application rather than constantly rendering UIs on the personal device. Experiments were conducted to validate the advantage of XICE's distributed scene-graph architecture. The tests compare the UI distribution techniques of TightVNC [TightVNC 2011] (Tight), RealVNC [RealVNC 2011] (Real), RDP, and XICE on Windows XP, and X11, the system-supplied VNC, and XICE on Linux. With each UI distribution technique, the tool was distribution technique was installed and used as-is. In the case of TightVNC, it was used with the "high-bandwidth" option selected.

XICE, X11, and RDP each render only on the display server, while all versions of VNC render on both the client machine and the display server. One may wonder why XICE should be compared to VNC because the fundamental difference in rendering styles. VNC was chosen because the render-locally-and-transmit-frame-buffers approach is used by several potentially nomadic environments such as IMPROMPTU [Biehl et al. 2008], LivOlay [Jiang et al. 2005], WinCuts [Tan et al. 2004], Lacome [Liu 2007], and Reflect [Argue 2007].

For all experiments, the personal device and display server are each a 3.4-GHz, hyper-threaded Intel Pentium 4 processor with 1 GB of RAM. The network is a single gigabit Ethernet switch. For all windows experiments, Windows XP SP3 is used, with all updates current to December 15, 2010. For all Linux experiments, Ubuntu 10.10 [Canonical 2011] is the operating system used, with all updates similarly current. In both cases, the Sun-supplied JRE is used: the OpenJDK [Oracle 2011c] supplied with Ubuntu did not perform as well when rendering. Data was collected using the logman [Microsoft 2011b] tool in Windows and nmon [IBM <cmr rid="cm1" label="IBM"> [RBA1]</cmr><cmt id="cm1">Should this be an acronym or the full spelled-out company name? 2011] in Ubuntu. In all tests, the application that produces the UI is a XICE application. The application is started, after a few seconds the monitoring tool is started and allowed to run for 4 minutes, collecting samples once every second. Then the first 15 seconds and last 15 seconds are discarded and the averages collated.

In the first experiment—the *small rotation task*—a spreadsheet is displayed in a small 50 × 50 window. Every 100 milliseconds, a separate thread rotates that window back and forth by 1.2 degrees, testing transformation with few overall drawing updates. Results are compared in Figure 13.

The None column shows the percent of total processing power the personal device uses to execute without distributing the UI. The client row shows the percent CPU usage on the client device while the Display row shows the percent CPU usage on the display server. The values in the Display row are not consequential, because the display server can be expected to have the resources necessary to process rendering commands. Processor usage includes the application, the UI distribution technology, and the operating system. The Bytes/s row shows the number of bytes transmitted per

Win	None	Tight	Real	RPD	XICE
Client	0.44%	2.02%	12.91%	0.64%	0.42%
Display		0.57%	0.48%	1.95%	0.47%
Bytes/s		1611	5044	1106	3128
Pack/s		3.74	6.35	14.21	14.2
Linux	**None**		**VNC**	**X11**	**XICE**
Client	3.94%		4.97%	6.86%	0.29%
Display			0.40%	3.30%	4.28%
Bytes/s			18408	2470160	3895
Pack/s			87.61	3494	20.13

Fig. 13. Performance on a small rotation task.

Win	None	Tight	Real	RPD	XICE
Client	50.09%	51.72%	50.03%	49.89%	0.44%
Display		2.68%	0.51%	5.46%	49.34%
Byte/s		96933	27240	589899	3111
Packet/s		158	40.5	686	14.1
Linux	**None**		**VNC**	**X11**	**XICE**
Client	48.54%		71.67%	16.98%	0.29%
Display			12.12%	14.51%	48.48%
Byte/s			1229522	28653027	3895
Packet/s			2870	27926	20.14

Fig. 14. Performance on a large rotation task.

second on the LAN while Pack/s shows the number of packets transmitted per second. Network usage is important, because radio transmissions can consume considerable power. Decreasing network traffic helps minimize radio utilization.

The most salient result in Figure 13 is that the CPU usage drops for both Windows and Linux, although Linux drops 3.94% to 0.29% when using XICE. That decrease translates to a 92.6% drop in processor usage on the personal device and includes the time required to serialize the presentation tree. By contrast, all other distribution techniques increased processor usage, and in the case of RealVNC, by 29 fold. And the network usage—of secondary import for preserving battery life—shows that XICE performs worse in half of the cases. However, it performs much better than VNC or X11 on Linux, and performs a little better than RealVNC on Windows.

In the second experiment—the *large rotation task*—the same rotations are performed with a 600 $\times$ 400 pixel window. Nearly 100 times as many pixels are rendered. The results of this experiment are shown in Figure 14. On the personal device, XICE drops CPU usage from about 50% to less than 0.5% (a 99% drop), while RDP does not drop usage at all. In terms of network usage on this second task, XICE greatly outperforms the other options in both Bytes per second and packets per second. In fact, network usage stays roughly the same as the small rotate task.

For the final experiment—the *scrolling task*—a separate thread scrolls a spreadsheet vertically one movement every 100 milliseconds. The spreadsheet is presented in a

Win	None	Tight	Real	RPD	XICE
Client	24.86%	27.61%	25.76%	30.53%	0.44%
Display		1.04%	0.55%	2.79%	36.19%
Byte/s		11268	9642	200265	12645
Packet/s		22.5	22.2	237	14.3
Linux	**None**		**VNC**	**X11**	**XICE**
Client	35.84%		70.34%	16.96%	0.39%
Display			15.74%	14.52%	43.17%
Byte/s			1269737	30686223	14232
Packet/s			3615	29470	20.1

Fig. 15. Performance on a scrolling task.

	Tight	Real	RDP	X11	VNC	XICE
Rotate Small	Fair	Fair	Good	Fair	Excellent	Excellent
Rotate Large	Fair	Fair	Fair	Poor	Excellent	Excellent
Scroll	Fair	Fair	Fair	Poor	Excellent	Excellent

Fig. 16. Interactive leg performance.

600 × 400 pixel window. While the prior two tasks are designed to force the UI to completely repaint, this last task more closely matches the common user technique of scrolling a window. The act of scrolling within the application causes many pixels to be rerendered without changing all pixels.

The results of the scrolling task experiment are shown in Figure 15. XICE drops processor usage from 25% to 0.44% (a 98% drop), while X11 can only reduce usage from 36% to 17% (a 53% drop). All other distribution techniques increase the client's CPU usage. In terms of network usage, XICE significantly outperforms the other options.

A major complaint about distributed interaction is the lag imposed by the distribution mechanism. To address this issue, the interactive response of the various distribution techniques is rated according to the following human observation of the interactive behavior of the application.

Excellent—no noticeable lag: less than 0.1 seconds
Good—slightly noticeable lag: between 0.1 and 0.25 seconds
Fair—noticeable lag: between 0.25 seconds and 1 second
Poor—excessive lag: more than 1 second

The times presented are a rough estimate of the lag between when a change is performed on the client machine and how long it takes before that change is rendered on the display server. Figure 16 lists the overall performance of each distribution technique relative to each task. Surprisingly, Ubuntu's version of VNC performed excellently for all three tests. However, that performance came at a large CPU and network cost. In terms of interactive responsiveness, XICE clearly performed better than any other technology examined.

The implementation of the Java Graphics rendering pipeline on Windows may affect these results. RDP intercepts Windows Graphics Device Interface (GDI) [Yuan 2000] calls and forwards them to the rendering display. However, Java Graphics2D uses DirectDraw [Yuan 2000] and not GDI. DirectDraw is meant for high-performance gaming with rapidly changing UIs, while GDI is meant for applications in which the UI does not change as frequently. Consequently, RDP does not receive notifications of each draw call within DirectDraw. Instead, RDP periodically sends the DirectDraw frame buffer to the display server. For Java applications, RDP pushes frame buffers just like VNC does, so RDPs true performance characteristics may not be reflected in the collected data. RDP would be expected to perform closer to X-Windows' characteristics.

X-Windows provides a more accurate representation of a competitive distributed UI technology, because X11 intercepts Java rendering calls and forwards them to the display server. The comparison between X11 and XICE is a better indicator of XICE's performance against a tightly integrated UI distribution technology. Even X11, however, could only achieve a 65% reduction in processor usage (48% to 17%), compared with XICE's 99% reduction. And, X11 imposes a gigantic network load—one to two thousand times what XICE uses during the same task. Even though the network protocol for XICE is not optimized for highly efficient binary throughput, it still yields a huge performance boost with minimal effort.

In the two larger experiments, using XICE greatly reduces both CPU load and network usage, while the smaller experiment shows that XICE performs a little worse. However, XICE's interactive experience excels for all experiments. Combining both the interactive experience and the observation that XICE's client-side network and CPU loads stay constant for all three distributed tests, shows XICE to clearly be preferable. The scene-graph architecture greatly reduces processing time, and can greatly reduce network time as well. Nomadic computing with XICE's incremental update of scene-graphs extends battery life and helps ensure that personal devices can handle the required computation and rendering loads on large shared displays while providing a responsive user experience.

6. INPUT IN NOMADIC SITUATIONS

Users must be able to provide both text and pointing input to applications regardless of which of the three nomadic situations applies. The applications must be able to operate without considering the input's source or if that source changes. XICE's scene-graph architecture simplifies input dispatching, compared to input handling in the damage-repaint cycle, and allows applications to operate regardless of the nomadic situation.

6.1. Input Handling

In each nomadic situation, users provide input to interact with a widget. Traditionally, input is handled by widgets that use the damage-repaint cycle, and presentation geometry is created in the repaint method. To select an object, pointing input is hit-tested against the object's presentation geometry. If a geometric transformation is applied to an object's presentation, the inverse of that transformation must be applied to properly hit-test the object. Hit-testing in traditional architectures frequently requires the widget to reproduce the presentation geometry. Applying a transform in the repaint method, for example, necessitates calculating and inserting the inverse transform into hit-testing code. Nested geometric transformations further complicate this process.

In scene-graph architectures, the presentation geometry, including affine transforms, is stored in the scene-graph. In architectures like XICE, Piccolo, Jazz, and WPF, input is dispatched top-down through the scene-graph. For example, Figure 17 shows pointing input dispatched to the "Yes" button, represented by the left red ellipse in the tree.

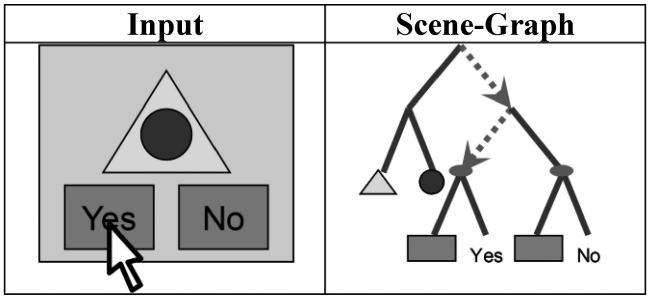

Fig. 17. Dispatching input within a scene-graph. The mouse click starts at the tree root and passes through each node down the tree until it arrives at the button widget. The button widget hit-tests the click against the rectangle's bounds.

The button widget performs a hit-test against the button's background rectangle to determine if the input is within the rectangle's bounds.

When using a scene-graph, developers create the tree once, and any transforms that need to be applied are embedded within the tree. As pointing input passes through a Transformer node, that input is transformed according to the transform at that node. Nested transformations result in nested transforms of the input. By transforming the pointing input at each Transformer node, the input is in the proper form for each node's coordinate space, removing input handling complications from the damage-repaint cycle. Developers can hit-test the existing tree instead of recreating the presentation geometry and remembering to properly apply inverse transforms to input.

Keyboard input is not affected by affine transforms and is therefore dispatched using standard text dispatching mechanisms.

6.2. Input In Nomadic Situations

In nomadic situations, different devices can supply user input to local or remote windows. *Local windows* are pieces of software on the personal device that send their UIs to the personal device's screen. *Remote windows* are pieces of software on the personal device that send their UI's to a *proxy window* on an annexed device's screen. In the personal device alone situation (Figure 2), local windows receive input from the personal device's input hardware. In the annexed screen and input situation (Figure 3), remote windows receive input from annexed devices. And, in the annexed screen only situation (Figure 5), remote windows receive input from the personal device. When supplying input, users need to be able to smoothly transition among nomadic situations.

If the user is in the personal device alone situation, no transitioning is necessary, because all interaction occurs on personal device. If she is in the annexed screen and input situation, remote windows receive input from the annexed input devices. Suddenly, an intimate note from her spouse arrives on her personal device. When the message arrives, she transitions by physically using her personal device instead of the annexed input devices.

The annexed screen only situation, however, requires special consideration for redirecting input. A user in the annexed screen only situation interacts with applications shown on an annexed screen. Her personal device supplies pointing and text input. Pointing input is supplied by translating input on the personal device into mouse clicks and cursor movements reflected on the annexed screen. Text is entered through a soft keyboard on her personal device. When a potentially embarrassing note arrives from her spouse, the note is shown on the personal display where she can view it discreetly (Figure 18). If she ignores the message, personal device input continues to

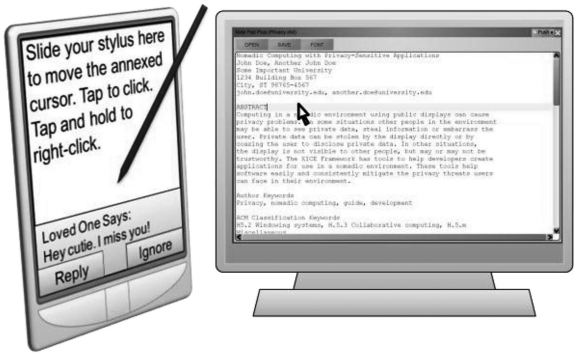

Fig. 18. A user may supply input on her personal device for a document in a remote window. If a message arrives on her personal device, she must be able to smoothly transition to supply input for the local window.

be sent to remote windows. However, if she elects to write a reply, input should be redirected to local windows. She must be able to smoothly transition between local and remote windows while using only the personal device for pointing and text input.

6.2.1. Pointing Input. The user should not need to look at the personal device to supply pointing input for remote windows. On the personal device, the UI for entering pointing input should not provide mechanisms whereby the user can inadvertently switch from remote to local windows. When she is focused on the display server's UI, looking back at the personal device to ensure she is providing input to the correct windows can annoy her and distract her from accomplishing her tasks.

Existing personal devices are designed to only provide pointing input for local windows. When annexing display servers, that input must be redirected to remote windows. As annexing becomes more prevalent, personal devices may supply a second input source dedicated to remote windows. In the meantime, existing devices need to provide pointing input to remote windows.

6.2.1.1 Redirected pointing input. Each personal device needs a device-specific way to redirect pointing input to remote windows. Personal devices that accept stylus interaction would require interactive techniques like those used in Pebbles [Myers 2001]. Personal devices that have other sources of pointing input (e.g., touchpad, mouse, arrow keys, or fingers) would need device-specific solutions for redirecting their input.

One way to redirect input to remote windows is to show a dialog that receives all input and then redirects that input, similar to Pebbles. For instance, on a laptop with only touchpad input, a small dialog could capture cursor movements and translate them into remote cursor movements. After each cursor move, the cursor would be recentered in the dialog so that remote cursor movement is not constrained by the bounds of the laptop screen.

Another way to share input between a personal device and a display server is to treat the personal device and display server as a unified display space. For instance, Synergy [2011], MaxiVista [Bartels Media 2011], iRoom [Johanson 2002], and many other toolkits allow the user to slide the mouse off of one screen onto another screen. Related is the approach of Mouse Ether [Baudisch et al. 2004] which allows the mouse cursor to travel in the space between monitors so that the mouse does not jarringly jump to a destination monitor. However, such approaches may allow a window to overlap screens from two separate devices which may interfere with the privacy features that will be discussed in section 8.

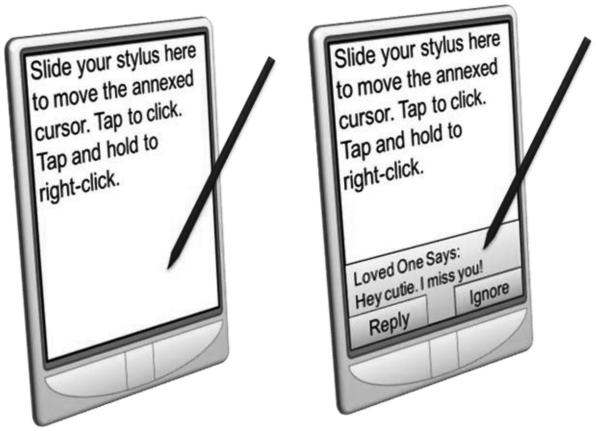

Fig. 19. A full-screen window can redirect stylus input as mouse input (left). A message may arrive at any time and inadvertently interrupt stylus input.

To capture stylus input from the personal device without shifting attention from the annexed screen requires a full-screen dialog on the personal device. A simple example of this dialog is shown on the left of Figure 19. If the dialog is not full-screen, the user may unintentionally tap widgets external to that dialog.

Suppose a message—like the one on the right of Figure 19—arrives while the user is looking at the annexed screen. If she accidentally taps within the message, she may respond to or ignore the message without realizing. She needs to be notified of the message without closing the capturing dialog or inadvertently interrupting input redirection. She also needs the ability to relinquish pointing input redirection, so she can interact with the message.

Instead of completely obscuring all window UIs on the personal device, the UI for a capturing window could be transparent. A transparent window UI allows the user to see her messages and any other local window UIs. Instructions on the transparent window's UI would let her know how to access local windows. With XICE, when pointing input is sent to remote windows, the UI for a transparent *blocking window* is shown on the personal device (Figure 20). The blocking window's UI is shown on handhelds, laptops, and any other personal devices that have a single source for pointing input.

6.2.1.2 Independent pointing input. A straightforward way of supplying pointing input to local and remote windows is to provide a dedicated source of input on the personal device for each type of window. For example, similar to Oprea et al.'s [2004] device, a compact, inexpensive optical mouse sensor could be mounted on the back of a personal device. A prototype, called the MousePuter, has been created using a Sony VAIO UX and the core hardware from an optical mouse (Figure 21).

The personal device can then be used like a mouse. The optical mouse on the underside of the personal device provides pointing input for remote windows; direct interaction with the personal device's screen supplies pointing input to local windows. Input redirection becomes unnecessary, eliminating the need for a blocking window. This solution works well for handheld personal devices, and the user can interact naturally with any display server that does not supply input or that the user does not trust.

6.2.2. Text Input. In addition to providing pointing input to remote windows, the personal device should provide text input. The personal device could provide a full physical keyboard, a small physical keyboard, a soft keyboard shown on the personal device, or a soft keyboard shown on the annexed display.

Fig. 20. The UI for blocking windows informs the user that input has been redirected from local windows to remote windows, and offers instructions for returning interaction to local windows on the personal device. The software buttons along the bottom cannot be interacted with directly: the user must use the physical buttons below them.

Fig. 21. MousePuter prototype.

6.2.2.1 Full physical keyboard on the personal device. A personal device such as a laptop may provide a physical keyboard that requires little visual attention. A user may type with the keyboard while watching the annexed display. The interaction with this particular device setup is natural, because people use tactile and visual feedback when entering text.

6.2.2.2 Small keyboard or soft keyboard on the personal device. If the keyboard is small or soft, the user will typically spend most of her time looking at the keyboard while entering text. Without looking, touch-typing accurately is difficult in the case of the small keyboard, and nearly impossible with the soft keyboard. Consequently, she will want to regularly look at the screen to ensure she is entering text correctly. In such circumstances, she must frequently switch visual attention between the personal device and the display server to confirm text entered.

If, however, the text input area on the remote window is duplicated on the personal device, the user can confidently enter text by watching only the personal device. Sharp et al. [2006] implement a form of this type of text entry. Unfortunately, in their solution, the local window shows a small portion of a pixel-by-pixel copy of the remote window's UI that does not properly fit the personal device's screen. The shown pixels are from the area of the window's UI directly surrounding the mouse cursor. As a result, the user must regularly move the mouse to keep the entered text within the personal device's screen. To compensate for this, the local window must be able to adjust its UI for the

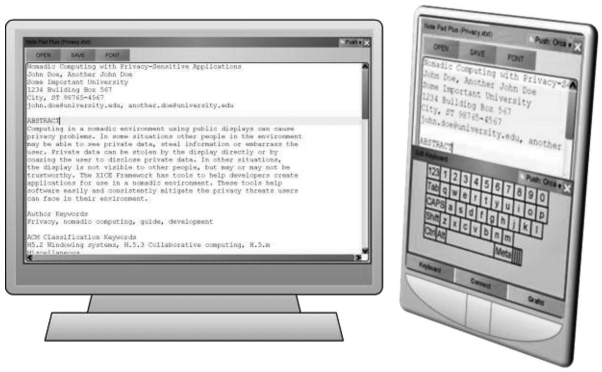

Fig. 22. Using the personal device's soft keyboard to provide text input for a remote window. The remote window is copied to a local window so the user can immediately see the entered text on the rendered UI.

personal device's screen. If just the text entry widget were replicated, it would have to adjust as well.

XICE allows windows to adjust their UI to the personal device's screen. XICE clones the focused remote window's presentation tree to a new local window, as shown in Figure 22.

The clone is created when a widget on the remote window requests text focus. Through model-view-controller (MVC) design, both the local window's presentation tree and the remote window's presentation tree share the same model. The XICE architecture makes cloning presentation trees simple and efficient without necessitating application intervention. XICE automatically ties each cloned widget to the original widget's model, rendering the cloning process trivial. Each clone may adapt its appearance according to the screen's parameters and the input available on the personal device.

6.2.2.3 Soft keyboard on the annexed display. The personal device could show a soft keyboard on the annexed display. The soft keyboard is easy to create and position near where the user is already looking. However, the display server can perform malicious acts with the soft keyboard. For instance, if the display server is blocked from supplying text input directly, the display server may still commandeer the keyboard and surreptitiously move it around the screen or rearrange the characters on the keyboard; as a user attempts to enter standard text, she might inadvertently send malicious text to the application. Consequently, XICE does not show a soft keyboard on an annexed screen unless the user completely trusts the display.

7. XICE TOOLKIT AND NOMADIC EXPERIENCE

The XICE windowing toolkit has several major components that enable users to annex display servers safely and confidently. XICE is designed around the idea that users push UIs to display servers. The toolkit facilitates annexing devices and enables developers to seamlessly write code that operates in all three nomadic situations. This section will describe how XICE implements each of the nomadic situations: personal device alone, annexed screen only, and annexed screen and input. Regardless of the nomadic situation, applications must be able to build presentation trees, create windows, render each presentation tree on a window, and receive user input.

An application must request that a "space" create a window. A *space* is software on the personal device that represents the display area on which the window's UI will be rendered. A space tracks the screens, their sizes in VIC, and their relative

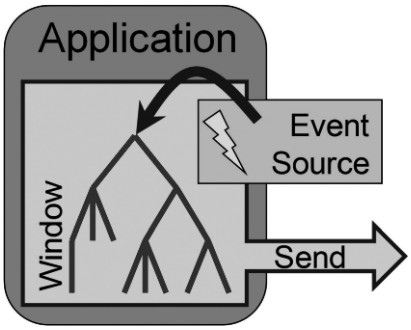

Fig. 23. Generic software organization for any application window. The window stores the presentation tree and dispatches events from the event source to the tree. The window serializes the tree or renders the UI.

locations. The space manages the size, position, and z-order of any windows rendered on that space. Spaces also perform the work necessary to show a window's UI on a screen. A *local space* creates local windows and represents the personal device's screen resources, whereas a *remote space* creates remote windows and represents an annexed display server's screen resources. Local spaces direct windows to render their UI's on the personal device, while remote spaces direct windows to serialize their UIs to the display server.

To receive user input, a window must have an event source. An *event source* is software that tracks and directs input from a machine. This input could be from a single hardware device, from multiple hardware devices, or from multiple users on a single machine. (With multiple devices, each device is identified using a unique ID, similar to Multi-Pointer X [Hutterer and Thomas 2007]). The space automatically registers each new window with an appropriate event source. The event source uses the space to determine which window receives dispatched input (based on the size, position, z-order, and text focus of the windows' UIs). After the event source sends input to a window, the window dispatches the input to the presentation tree.

A *local event source* is an event source that dispatches user input from the personal device's hardware. A *remote event source* dispatches user input from a display server's hardware. A space may supply either type of event source; a window is assigned a single event source that is either local or remote. As XICE connects to a display server, its space is assigned a default event source based on user preferences: a local event source is assigned when the user does not want the annexed device to supply input, and a remote event source is assigned when the user does want the annexed device to supply input. When the space creates a window, the default event source is attached to the window.

Applications always execute on the personal device; they are never transmitted to a display server. Only the UI is serialized to a display server. As the user changes an application's nomadic situation, specific components used by the application may change (e.g., an application may swap a local event source for a remote event source or vice-versa), but the overall UI architecture remains the same. The general software organization for all windows within an application is shown in Figure 23.

All pointing input within XICE is stored and transmitted in VICs. For instance, a mouse location is stored in VIC's relative to the display space, and dispatched according to which window is directly under that mouse location. Button-click states are transmitted with the mouse movements and whenever the button-click state changes. Keyboard input is not related to VICs so keyboard input is dispatched to whichever widget currently has input focus.

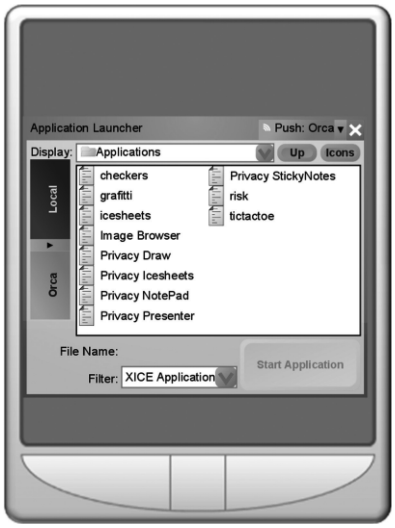

Fig. 24. Application launch dialog rendered on the personal device.

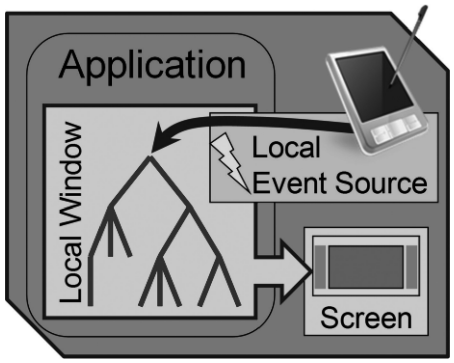

Fig. 25. Personal device alone software organization.

Developers do not need to consider whether a space, a window, or an event source is local or remote. Developers only need to create presentation trees and assign them to windows.

7.1. Personal Device Alone

With XICE, users point, select, and execute as usual. When all input and output is on the personal device, self-contained interaction does not differ on personal devices such as laptops and PDAs. However, core application design and application startup processes diverge from existing systems.

A user must be able to launch an application on her personal device. The XICE framework, as implemented, provides an *application launch dialog* (Figure 24), and users select an application from its UI.

After the user chooses an application on the personal device, XICE starts the application and provides it the local space, so the application can show local window UIs. The software is organized in accordance with the representation in Figure 25.

Fig. 26. The "Push" button initiates the process of annexing a display server. More options are available through a drop-down menu (inverted triangle).

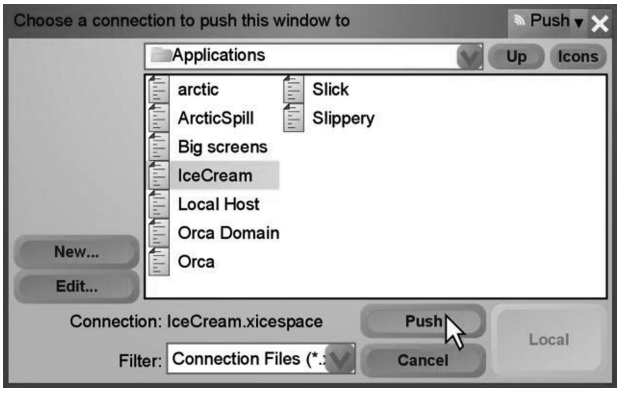

Fig. 27. XICE connection dialog UI. The user is pushing a window's UI to the computer she named IceCream.

In the software organization for the personal device alone, the local space creates a local window and assigns the local event source to that window. The window then renders its presentation tree.

7.2. Annexing Display Servers

The process of annexing a display server should be as straightforward as possible, particularly for frequently used display servers, such as the personal desktop. In windowing toolkits, common window management functions are typically available in the corner of the title bar. XICE employs a similar approach and adds new functions for pushing a window to another screen. To reduce clutter and increase understandability, two buttons are presented to the user: the standard "close" button with one other (a drop-down provides access to other common functions). By default, a "Push" button is shown in the upper right corner of window UIs on the personal device (Figure 26). The user simply clicks the "Push" button to establish a connection with a display server and push that window's UI to the display server.

More specifically, after a user clicks the "Push" button, a connection dialog's UI is shown (Figure 27). The dialog lists display servers which the personal device has previously annexed. A configuration file is associated with each of those display servers. The user simply selects a display server's configuration file, and the current window's UI is pushed to that display server.

Each configuration file contains the following information.

—The name of the display server (as assigned by the user)
—The domain name or IP address of the appropriate XICE display server
—Whether the display is trusted to show sensitive data (default: distrusted)
—Whether the display server's input devices are trusted (default: distrusted)
—Where input comes from (default: local event source)

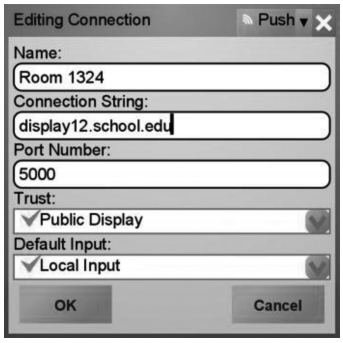

Fig. 28. XICE connection properties dialog. The "Default Input" drop-down box is used to select which device input comes from.

Fig. 29. An option on the personal device for changing the hardware input source.

These configuration files are intended to identify display servers the user connects to regularly, like a desktop or a home television. In such situations, the user has a specific trust setting meant for each respective display server. Saving these settings in a file simplifies the annexation process for subsequent connections to a particular display server.

People may also annex display servers they have never used before. To annex a new display server, the user clicks the "Push" button, requests a new connection, and enters the domain name or IP address of the display server. The connection dialog clearly shows a "New..." button (Figure 27) users can click to initiate a connection with a new display server. The user must be made aware of the display server's domain name or IP address; otherwise, she cannot connect to the display server. XICE can show that information via a dialog on the display server, or the display's owner can post that information on a physical sign near the display server.

Editing a configuration file is inconvenient for users. XICE provides the dialog in Figure 28 for editing the properties stored in the display's configuration file. The green check marks represent the default, safe configuration for those two properties.

When the user connects to a display server, the XICE toolkit provides an additional dialog on the personal device to allow the user to manage the annexed screen. This dialog lists the windows on the annexed screen and provides configuration options for those windows and for the display server. One such option (shown in Figure 29) allows the user to change the hardware input source.

The connection process can be accelerated with broadcast techniques such as Bonjour [Apple 2010b] and location services [Thota 2005]. These techniques filter results based on the personal device's wireless access point or Global Positioning System (GPS) location, enabling a user to easily and accurately select a nearby display server. Such connection facilitators could be used by XICE but are not essential to this discussion.

Once a display server has been selected, the scene-graph on the current local window is attached to a remote window. Remote windows' UIs show the "Pull Back" option (Figure 30) in place of the original "Push" button. The "Pull Back" button allows a

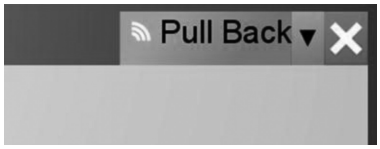

Fig. 30. A window's UI that has been pushed from a personal device to a display server has a "Pull Back" button that enables the user to easily remove that window's UI.

Fig. 31. After a connection is made, local windows prepare to push their UIs to the annexed display server.

user to quickly remove individual scene-graphs from the display server and closes the respective remote window.

After establishing a connection, pushing scene-graphs to the annexed display server is simple. The "Push" buttons for all local windows change to "Push: *Connection*", where "*Connection*" is the name of the most recently connected display server. For example, clicking the "Push: Orca" button in Figure 31 sends the scene-graph to the machine the user named Orca.

After a scene-graph is attached to a remote window, the scene-graph is serialized to the annexed display server. Then, the application executes normally.

Some applications may need to know if a user has pushed their scene-graphs to an annexed display. In particular, widgets may need that information so they can adapt their appearance based on the display server or context. For example, to protect email addresses in a scene-graph from being shown on a public display—where other people or the display server might steal them—the *To* and *From* address boxes would remove those addresses from the scene-graph (described in more detail in the Protecting Distributed Applications section). To protect sensitive data, applications must be informed that a scene-graph has been pushed to a distrusted display server.

XICE notifies each presentation tree when it has been moved to a different window. When the user pushes a scene-graph, XICE creates the remote window, moves the presentation tree to the remote window, closes the local window, and then serializes the scene-graph to the annexed display server. Before serialization, all widgets within the tree are sent a *recontext event* to notify them that the scene-graph has been attached to a new window. The recontext event provides each widget an opportunity to change its appearance according to the window's context. Most widgets ignore the event and pass it to all child nodes. However, widgets that adapt to context can alter the presentation tree. After all widgets have processed the recontext event, serialization proceeds normally.

In addition to pushing a window's UI from the personal device to the annexed screen, a user may launch an application on the personal device directly from the display server. The user clicks on the display server's desktop and the personal device shows the application launch dialog's UI (Figure 24) on the display server. The dialog's UI lists all the applications installed on the personal device; any applications that might be installed on the display server are not listed. The user selects an application that launches on his personal device, and XICE supplies the application with the remote space for creating windows. However, launching applications directly from a display server has security and privacy implications (Protecting Distributed Applications: section 8).

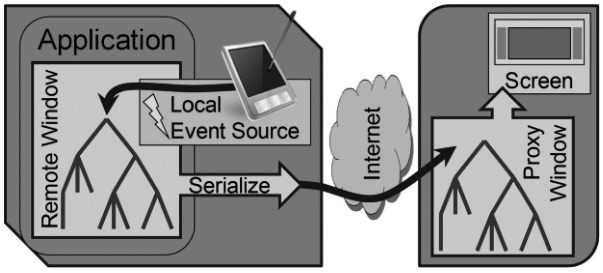

Fig. 32. Annexed screen only software organization.

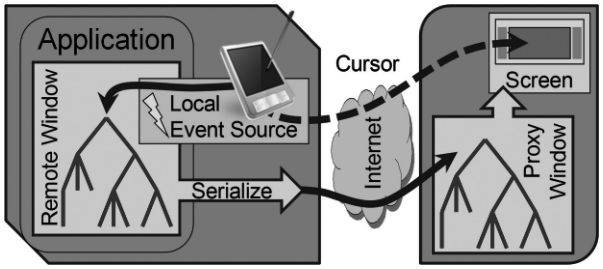

Fig. 33. Remote cursor echo.

7.2.1. Annexed Screen Only. If a display server does not provide input devices, or if a user distrusts the display server's input devices, then the user is in the annexed screen only situation. Figure 32 illustrates the software organization for a remote window with input from the personal device.

The user needs to provide pointing and text input to remote windows using the personal device. The remote space supplies a local event source to attach to new remote windows. The local event source tracks the cursor's position and shape, and transmits that information to the display server for presentation to the user (Figure 33).

7.2.2. Annexed Screen and Input. In the richest nomadic situation, a user interacts with data directly using the display server's input devices. The user could annex a display server, then set the personal device down or place it in her pocket. Although the personal device would continue to run her applications, she would interact exclusively via annexed devices.

To annex a screen and its input devices, the user changes the configuration settings to accept annexed input (Figures 28 and 29). After this change, a remote event source is attached to each new remote window on that remote space. Remote windows render on the annexed display, input is processed on the personal device, and scene-graph changes are serialized to the display server. The annexed screen and input software organization is illustrated in Figure 34.

7.2.3. Annexed Input. Another potential situation is if the user just wants to accept richer input for their personal device. For example, the user may want to annex a keyboard and/or mouse so that he can have a richer typing experience or finer pointing ability within his applications. In this configuration, the display server's input is routed across the network to the personal device and then transmitted to the user's applications.

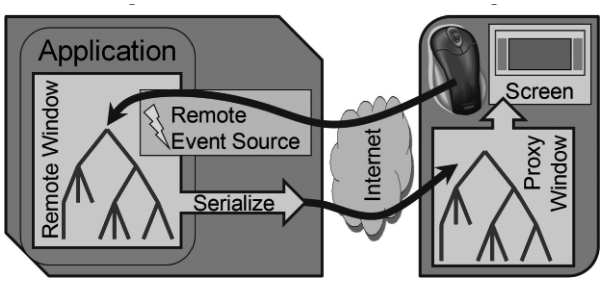

Fig. 34. Annexed screen and input software organization.

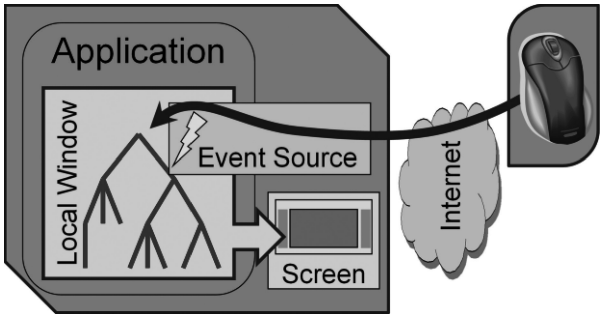

Fig. 35. Annexed input software organization.

To establish such a connection, the user is no longer pushing a window to a display server. Instead, he is grabbing input from a display server. He must establish such a connection through a channel other than the "Push" option from Figure 26, but XICE's software can easily handle such an organization. This software organization is illustrated in Figure 35.

8. PROTECTING DISTRIBUTED APPLICATIONS

One major reason a user may enter into the annexed screen only situation—and then have to provide all input from the personal device—is that she may not trust the annexed display server to provide input. As mentioned earlier, some significant security risks are introduced by distributing a UI to annexed screens. Four threats specific to UI distribution are: stolen input, false input, stolen output, and false output.

With XICE, the personal device performs application processing to protect a user's sensitive code and data; code and data are never distributed to the display server, so the display cannot directly steal that data. User input from the local event source is never routed through the display server, so the display server cannot redirect user input to different widgets or change the input the user supplies.

XICE has built-in provisions for privacy. The details of the levels of privacy protection XICE supplies are unimportant for this discussion. What is salient is that display servers may be assigned a *privacy state* of *public* (distrusted) or *private* (trusted). The user specifies whether the display server is distrusted or trusted in the display server configuration file. Input from a display server may also be considered distrusted or trusted, and this setting is independent of the privacy state.

Most distributed UI solutions do not provide applications with information about the privacy state of prospective displays. If an application does not have access to the privacy state of a display, the application cannot take appropriate protective action.

Consequently, each privacy-aware application must have its own way of detecting the privacy state of a display, usually by allowing the user to input that state directly to the application. Symbiotic Displays [Berger et al. 2005] takes this approach, allowing the user to specify the privacy state of a display server directly to the application.

Instead of having a per-application privacy-aware solution, the windowing toolkit should track the privacy state of the display, provide that information to applications, and inform applications of any privacy state changes. Applications, on the other hand, must inform the toolkit of sensitive input and output. By making the privacy state available and identifying sensitive input and output, the toolkit and applications can proactively protect sensitive data.

XICE provides the privacy state in the device context passed to each widget when its presentation tree is attached to a window or when an event is dispatched to the presentation tree. The most important event for privacy is the recontext event. To protect sensitive data, applications that respond to the recontext event may change their UI before distributing it to a public display server. In addition, applications may tag presentation trees or sections of presentation trees as sensitive, and the XICE toolkit can protect those trees or sub-trees from potentially malicious display servers.

8.1. Stolen and False Input

Stolen input occurs when the display server overtly steals sensitive input (e.g., usernames and passwords) directly from the user. *False input* happens when the display server impersonates user input to applications. False input usually happens in an attempt to get the user to expose sensitive data (e.g., email addresses) to the display server so that it can steal that data.

To protect against stolen input and false input, XICE automatically configures any new connection to block input from a display server. If user input from the display server is blocked, the display server cannot supply false input to the user's applications. Also, the display server is less likely to obtain sensitive input directly from the user.

When the user elects to accept display server input (the annexed screen and input situation), XICE encourages her to distrust that input. So, a display server may supply input that the personal device dispatches, but the input is tagged—per her settings—as either distrusted or trusted. This tagging allows an application to identify what the user considers unsafe input and take appropriate action with respect to that input.

8.1.1. Distrusted Input. Most input a user enters is not sensitive and does not affect sensitive data. In these situations, the user can seamlessly supply input using annexed input devices.

When annexed input affects data the software knows is sensitive, the software should ensure that the user explicitly confirms that action on the personal device before implementing changes. For example, XICE supplies the dropdown menu in Figure 36 on all remote window UIs on public display servers. If the user clicks the "Show Private Data" option, the window could expose all of the user's sensitive data on that UI. However, because input to the "Show Private Data" option is distrusted, XICE explicitly confirms the action on the personal device prior to exposure of any sensitive data. XICE's event handler for that menu option checks the trust tag on the mouse input, discovers that the input is distrusted, and shows a confirmation dialog on the personal device. The user must confirm the menu click on her personal device before the window will explicitly show the user's sensitive data on that public screen.

The confirmation dialog also alerts the user when the display server surreptitiously attempts something malicious. In addition, when a widget expects sensitive input (e.g., usernames and passwords), the widget can check the trust level of the input source; the sensitive input can then be requested on the personal device, preventing the display

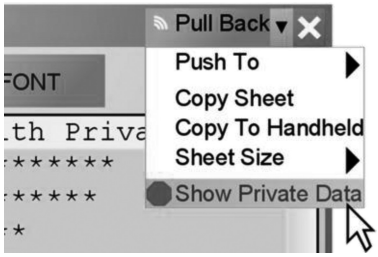

Fig. 36. Options such as "Show Private Data" can be exploited by malicious display servers. XICE mitigates such acts by explicitly confirming privacy-affected commands on the personal device.

server from accessing sensitive data. Both of these features further protect the user against stolen and false input.

If an application supplies an option similar to the "Show Private Data" option, the application must also enable the user to confirm that option on the personal device; applications can easily detect when input is distrusted, so they can properly confirm such actions.

The ability to launch applications on the personal device directly from the display server is a known security threat. Unbeknownst to the user, the display server might launch a susceptible application to exploit its vulnerability. On the personal device, XICE automatically requires the user to confirm application launch requests originating from distrusted display servers.

8.1.2. Trusted Input. If a user completely trusts a display server (like she would at a personal desktop), then none of XICE's security measures are enforced. Users have a seamless experience with fully trusted annexed display servers.

8.2. Stolen Output

Any data shown on a display server—including sensitive data like usernames, email addresses, and phone numbers—can be stolen by that display server and potentially used maliciously. The easiest solution is for users to never annex any display servers, but that would unnecessarily limit users to the resources on their respective personal devices. In addition, a trustworthy screen may be available in a public place. For instance, a screen in a conference room might be trusted because the company properly maintains the display server, but a presenter may not want sensitive company data to be divulged to an audience of vendors.

To protect sensitive data, applications must be aware of public environments. The user should account for the public nature of an environment by setting the display server's privacy state to public. When the application is aware the display server is public, the application can appropriately protect the user's sensitive data. XICE stores a device's privacy state as a property of the device's space. The local space is always private because it represents the personal device, but a remote space may be public or private.

In XICE, the user can change a remote space's privacy state at any time. For instance, if a user is at her private desktop and a coworker approaches to discuss some work, the user may change the desktop to public, protecting her sensitive data from the visiting coworker. When the coworker leaves, she can change her desktop back to private. XICE sends a recontext event when the privacy state of a display changes, informing all windows and widgets of the change.

Applications that use sensitive data may need to proactively protect that data. The developer of an email application may design it to ensure that each email is initially

shown on a private display. To implement this functionality, the email application checks the space's privacy state before creating an email window; if the space is public, the application shows the email window on the personal device instead of on the public display server. If the user chooses to push an email to a public display, the presentation tree receives a recontext event that contains the privacy state for the new remote window. The "To" and "From" widgets receive the recontext event and can then, like Symbiotic Displays [Berger et al. 2005], protect embedded email addresses.

A simple way to protect sensitive data is to overlay black rectangles. Privacy Blinders [Tarasewich et al. 2006] uses this approach to protect sensitive data. Even if viewers cannot see the sensitive data, the display server still has access to the information, since both the sensitive data and the overlaid rectangles are sent to the display server. Widgets that protect sensitive data must scrub that data from the presentation tree before the tree is serialized to a public display server. The recontext event affords widgets opportunity to remove sensitive data from the presentation tree and ensures that no sensitive data is inadvertently sent to a public display.

XICE provides some automatic systems to help developers protect sensitive data. For instance, file dialogs may reveal sensitive data (e.g. file names or file system structure) outside of an application's control. Therefore, file dialogs should not show on a public display. However, expecting each developer using the file dialog to properly implement this restriction is not practical. Instead, the file dialog widget should automatically ensure that the file dialog only shows on private displays.

To guarantee that a particular dialog only appears on private displays, XICE requires that the dialog be tagged with the *private only* Java annotation. The UI for the dialog is represented in a presentation tree that is rendered on a display, so the root of the presentation tree must have the private only tag. XICE is designed so that public spaces check for the private only tag and automatically redirect a tagged presentation tree to a new window on the personal device. The private only annotation is only effective on the root widgets in a presentation tree. Applying that annotation anywhere else is not protected.

If developers create a subclass of a dialog that is tagged private only, the dialog remains private only. Because XICE automatically redirects these dialogs to the personal device, developers can use the original or subclass dialogs without implementing redirection code or explicitly protecting those dialogs within the application. So the user's private file system is protected consistently across applications.

When a file dialog is opened on a public space and consequently redirected to the personal device, the user needs to shift her visual focus from the annexed screen to the personal device. However, if she expects the dialog to be rendered on the public display she may be confused. To minimize this confusion, XICE automatically renders a *heads-up dialog* on the public display whenever a private only dialog is redirected. The UI for the heads-up dialog instructs the user to look at her personal device (Figure 37).

In the annexed screen only situation, input on the personal device is typically directed at remote windows. If a user performs an action on a remote window's UI which creates a dialog that is redirected to a local window, then input is also automatically redirected to local windows. When the dialog closes, XICE automatically redirects input back to the remote windows.

The heads-up dialog is likely to work best in situations where a user's action caused a redirected dialog to appear. If an application attempts to show a private window independently of user action and the heads-up dialog appears, then the other users at the display space may become confused as each tries to figure out if he is the owner of that dialog. Such software design decisions are discouraged.

Fig. 37. File dialogs are not sent to public displays. Instead, a *heads-up* dialog rendered on the public screen directs users to the file dialog's UI on the personal device.

8.3. False Output

Display servers could overtly falsify the user's application output. *Falsifying output* includes any effort to coax users to expose sensitive data that the display server can then steal. For example, selecting the "Show Private Data" menu option in Figure 36 causes an application to expose the user's sensitive data. A malicious display server might expect the user to click the "Copy Sheet" option, but wants the user to click the "Show Private Data" option. To accomplish this, the display server would swap the text of the two menu options, so the user thinks she is clicking "Copy Sheet" when, in reality, the personal device interprets that click as "Show Private Data".

A similar situation could occur when a user interacts with the application launch dialog on a malicious display server. The display server might rearrange application names or reposition the mouse cursor to coax the user into launching applications that the display server could then exploit.

To mitigate the false output problem, critical inputs or outputs are shown only on private devices. If the annexed device is public, XICE renders a confirmation dialog on the personal device, and a heads-up dialog rendered on the display server lets the user know she needs to enter input on the personal device. By double-checking critical actions, XICE discourages a display server from falsifying output and prevents the display server from surreptitiously stealing the user's sensitive data.

9. BENEFITS OF THE XICE PROTOCOL

XICE is a windowing toolkit that offers a seamless nomadic experience and helps mitigate or solve a wide array of problems related to creating a nomadic computing environment. In particular, XICE is network-based to provide easy connection to a variety of display servers and to accept input from them. The network-based protocol also allows multiple users to annex a display server simultaneously. Unlike with VGA connections, XICE users may show their personal applications on the same shared display to compare information.

Scene-graphs defined in VIC allow XICE to distribute UIs to display servers in myriad viewing situations that include multiple sizes, resolutions, and viewing distances. Using the scene-graph increases the UI's ability to adapt to window size, screen dimensions, and screen resolutions; reduces personal device CPU requirements; and lengthens battery life on the personal device by offloading the rendering burden to the display server.

XICE encourages development and implementation of a stable display server platform. This stability results from a simple communication and rendering framework.

One key benefit of a stable platform is that users and developers can trust that their application UIs will have the same appearance, regardless of the device annexed. Another benefit is that users do not need to reconfigure their personal devices to deal with each new display server's installation.

If a user chooses to accept input from a display server, she will have an interactive experience similar to her desktop. However, display servers might not supply input devices, or the user may distrust input from the display server, so XICE encourages users to annex only a display server's output. Consequently, the personal device should be used for all input to the user's applications. When the user provides input on the personal device, XICE can smoothly transition between local windows and remote windows. Particular devices, such as the MousePuter, can leverage XICE to naturally transition between local and remote windows by supplying dedicated input hardware for each type of window.

XICE has several features that protect sensitive data from malicious displays. First, core application processing is on the user's personal device, so a malicious display server cannot steal or alter that data. Second, annexing input is discouraged. Blocking display server input ensures that the input comes from a trusted source: the personal device. Third, accepting but distrusting input from the display server allows applications to require the user to confirm actions that include or affect sensitive data. Fourth, XICE sends applications the display server's privacy state so that private data in the application's output can be protected. Fifth, XICE automatically prevents sensitive dialogs, such as file dialogs, from rendering on public displays.

10. DEVELOPER EXPERIENCE

A key factor that influenced the design of XICE is how effective it is for developers to learn and use. For example, in Section 5.3, this article mentioned the developer confusion in relation to DVA.

There have been roughly a dozen developers who have used XICE to create applications. The developers in the lab pick up XICE relatively quickly. The overall scene-graph structure takes little time to understand and start using. It takes only a few hours to master the overall usage of XICE, from building scene graphs to showing new windows, especially since developers do not have to deal with a distributed application unless they choose to.

There have been several dozen applications built using the XICE windowing toolkit and framework. Some of these test specific techniques while others pursue research avenues independently of XICE. Some of the key programs implemented in XICE are a spreadsheet application (called IceSheets), a text editor, a simple drawing application, and presentation software. These applications have varying levels of sophistication, as can be observed in the screen shots shown in Figure 38. IceSheets is designed for research into new avenues with spreadsheets and represents more than six hundred hours of developer effort. Most of that effort was spent implementing the various mathematical functions for the spreadsheet. Little effort went into UI development. In Figure 9, the user has "spilled" the IceSheets application and is panning it to another position. Other programs are games, such as Checkers, tic-tac-toe, or Risk, which are for training developers or pushing the graphical bounds of the toolkit. Checkers represents about 40 hours of developer effort; tic-tac-toe about 8; and Risk—which includes some automated game-playing capabilities—about 100.

When the privacy toolkit was added to XICE, IceSheets was already complete, but an experienced developer retrofitted IceSheets to support privacy by hiding private columns and rows, and graying out private cells. This retrofitting consumed about 8 hours of developer time, including the time it took to alter the data format to store

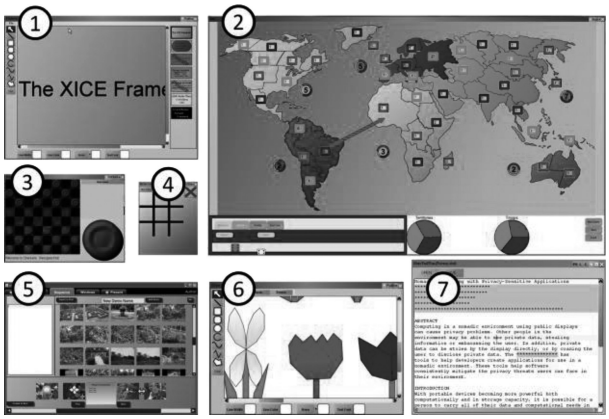

Fig. 38. Various applications implemented with XICE. 1) A presentation tool. 2) Risk^{TM}. 3) Checkers. 4) Tic-Tac-Toe. 5) A more sophisticated presentation tool. 6) Drawing application. 7) Privacy-aware text editor.

the custom values necessary to track which columns, rows, and cells contained private data. The data issues are not XICE related.

11. LIMITATIONS OF THE XICE PROTOCOL

There are some limitations to the XICE implementation. Some of these are limitations of developer resources; others are a limitation of the technology itself. This section will discuss several of these important limitations.

11.1. Backward Compatibility

A key limitation of XICE is that all applications must be rewritten for the XICE platform. When developing this framework, the authors considered this limitation and chose to forgo backward compatibility for several reasons.

First, although RDP supports network transmission of WPF scene-graphs, this fact is not widely known or easily discoverable. In addition, the source code for RDP is not available for augmentation to handle the aspects of multiple input devices, multiple focused windows (one per connected user), input redirection, and privacy that the authors need.

Second, coercing RDP or X11 into Java and then to support multiple input devices, multiple focused windows, input redirection, scene-graphs, and privacy is beyond the resources of the authors, especially when trying to provide timely research. It was easier to build the framework from scratch.

Third, separating the display spaces made for a natural addition of privacy because privacy may be attached to a display server.

Fourth, the paradigm for nomadic application development is fundamentally different from how legacy applications were built. With this new paradigm, a single application may simultaneously show separate windows on different displays and even display servers, with each window getting input from different sources. Legacy applications do not understand operating in this kind of environment. These applications are designed under the assumption of a single, unified display space where a single user interacts. Although backward compatibility would be nice for these applications, the authors think that many of these applications need to be rewritten simply because of the paradigm shift to a more flexible environment.

For example, consider the desktop and Windows Mobile versions of Microsoft Outlook: neither version handles swapping between a tiny screen and a large screen, let

alone rendering on both screens at once. The desktop version is only usable on an annexed large screen, while the mobile version can only show small windows on an annexed screen. The desktop version of outlook is pointless if it useless on a small screen. The mobile version gains no benefit from annexing a screen. A new version or Outlook that can simultaneously handle both a large and a small screen should be implemented. This new version should incorporate privacy-aware interaction: for example, show a window for privately selecting contacts via the smartphone while allowing the user to type up a new email on an annexed screen.

Fifth, smartphones released in recent years (e.g., the iPhone [Apple 2011a] and Android [Google 2011]) have shown that people are willing to rewrite applications for different devices, especially if rich toolkits are available for these devices. Although Android uses the Java syntax and a byte language, it is not backward compatible with existing Java applications, especially because the Android has a different rendering framework. Other technologies have also shown that developers are willing, if not eager, to rewrite their applications to adopt the new technologies. This happened with the advent of HTML and again with AJAX, especially when rich toolkits became available for these technologies.

But, regardless of these reasons to not be backward compatible, backward compatibility should be incorporated in the future, especially because users have existing applications on laptops that should also fit into the ecosystem afforded by XICE.

11.2. Rendering Performance/Capabilities

In some situations, damage/repaint is faster than a scene-graph. This happens when the scene-graph takes longer to serialize than to render the pixels and serialize the frame buffer. For instance, scatter plots with hundreds of thousands of points or complicated CAD diagrams may exhibit this property.

Pixel painting and photo editing are also not efficiently supported by XICE. To implement these, for each change made to the image, the client device must render to a new pixel buffer and serialize that buffer to the display server.

If a damage/repaint option is integrated into XICE, both of these problems may be addressed.

11.3. Collaboration

Copy & Paste between application windows supplied by different personal devices is not supported. For example, when two people share a display space, the first person cannot copy an appointment to the other person's calendar.

Part of this limitation is caused by the perspective of the authors that XICE should inherently distrust display servers. Consequently, a display server might not be trustworthy to either broker a connection between the two users or to transmit data from the first user to the second. The authors have not focused on this problem, but hope to spur thought in this direction as users need more seamless collaboration via shared display spaces.

11.4. Interaction Distance

XICE is not intended for large network distances: hosting an application in New York and interacting with it in California is not likely to produce a favorable user experience. Technologies such as HTML, AJAX, etc. provide a better experience over large distances because they transmit code to the web browser for execution. With code executing at the browser, the UI is more responsive.

However, with XICE, the user's personal device is in the room with the display server. So, the physical and network distances are short. Transmitting code is not necessary to gain the needed interactive richness and responsiveness.

11.5. Animating Graphical Primitives

A common feature of modern scene-graph toolkits is animation of graphical primitives. Unfortunately, XICE does not support such animation except through application-level implementations (i.e., an application developer must create an animation loop to update the discrete position of shapes). So far, with a low latency in the network, these animations are nearly seamless. However, serializing animations to the display server may have great performance benefits.

Animations are a natural extension of scene-graphs, as WPF has shown. An animation engine could easily be incorporated into the XICE scene-graph architecture and animation instructions serialized to the display server for execution. Many animation frameworks exist that could be used as a model for a XICE animation framework.

11.6. Video

As mentioned in Section 5.1, XICE has no video support. This support was not incorporated because the XICE rendering engine is implemented in Java, and JMF [Oracle 2011b] has poor video support, especially transforms, transparency, and graphical overlays to the rendered video.

If the XICE rendering engine were a platform-specific rendering engine (e.g., WPF or Cocoa [Apple 2010c]) which supports deep integration of video with other graphical primitives, then support for simple video rendering and manipulation (e.g., play, pause, fast forward) is straightforward. The XICE scene-graph would be transmitted to the platform-specific rendering engine, which would interpret that scene-graph accordingly.

11.7. 3D Graphical Primitives

Additionally, XICE does not provide facilities for 3D graphics, such as might be used in gaming or visualization scenarios. The framework was designed with 3D graphics in mind as a potential addition, but this was not included in the prototype. Incorporating work similar to blue-c's distributed 3D scene-graph [Naef et al. 2003] appears to be straightforward.

12. FUTURE WORK

In the immediate future, researchers and developers can design or re-implement widgets to be compatible with the XICE windowing toolkit, integrate and test multiple personal devices and display technologies, and examine numerous collaborative work environments.

Through XICE, input and output can be easily annexed, so widgets will need to be redesigned to handle varying input sources and output sizes. Input devices will range from the common mouse and keyboard to pens, styluses, fingers, lasers, game controllers, gestures, and eye tracking. Hard-coding widgets to meet every possible combination of input device and output device is excessively complicated; therefore, widgets and software need to be designed to handle a more abstracted version of input that can be stored and transmitted in a common way. The exact form of input/output storage and the precise mechanism of input/output transfer also need to be determined.

XICE does not transmit any data other than scene graphs and device input, but interaction with devices in the environment may require transmitting other kinds of data. For example, an environment may have a printer with advanced features that the user wants to print through. The user should be able to annex that printer and send documents to it (probably using PostScript [Gosling et al. 1989]) and use that printers advanced features using a UI that is familiar to the user (i.e., matches the user's look and feel on her personal device, but supplied by the printer, possibly using

XICE). Speakeasy [Edwards et al. 2002] is one such approach for combining nearby resources in flexible, powerful ways. Device Ensembles [Schilit and Sengupta 2004] lays out a good taxonomy of where these design decisions need to take placed. XICE is an example of a technology that fits in the "Application" (input and output redirection) and "Data Format" (standardized scene-graph and hardware input) groups in the Device Ensembles taxonomy. Such research should be explored in conjunction with the XICE interaction model.

XICE is designed to support multiple users with a variety of input devices and display server configurations. Exploration of scenarios and implications in the collaborative realm, with well-designed user studies, would be appropriate.

REFERENCES

ADOBE SYSTEMS. 1996. Adobe Flash. http://get.adobe.com/flashplayer/ (accessed 6/11).

ANSI. 1984. *GKS*. ANSIX3.124-1985.

APPLE COMPUTER. 2010a. iPhone. http://www.apple.com/iphone/ (accessed 6/10).

APPLE COMPUTER. 2010b. Bonjour. http://www.apple.com/support/bonjour/ (accessed 6/10).

APPLE COMPUTER. 2010c. Cocoa. http://developer.apple.com/technologies/mac/cocoa.htmh (accessed 7/10).

ARGUE, R. 2007. Advanced multi-display configuration and connectivity. MS dissertation. Dalhousie Univ.

BARTELS MEDIA GMBH. 2011. MaxiVista. http://www.maxivista.com/ (accessed 1/11).

BAUDISCH, P., CUTRELL, E., HINCKLEY, K., AND GRUEN, R. 2004. Mouse ether: Accelerating the acquisition of targets across multi-monitor displays. In *Proceedings of Extended Abstracts on Human Factors in Computing Systems (CHI'04)*. ACM press, 1379–1382.

BEDERSON, B. B., GROSJEAN, J., AND MEYER, J. 2004. Toolkit design for interactive structured graphics. *IEEE Softw. Engin.* 535–546.

BERGER, S., KJELDSEN, R., NARAYANASWAMI, C., PINHANEZ, C., PODLASECK, M., AND RAGHUNATH, M. 2005. Using symbiotic displays to view sensitive information in public. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*. IEEE Computer Society, 139–148.

BHARAT, K. AND CARDELLI, L. 1997. Migratory applications. In Lecture Notes in Computer Science. Springer Berlin, 131–148.

BIEHL, J. T., BAKER, W. T., BAILEY, B. P., TAN, D. S., INKPEN, K. M., AND CZERWINSKI, M. 2008. Impromptu: A new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. In *Proceeding of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. ACM, New York, 939–948.

CANONICAL, LTD., 2011. Ubuntu 10.10, http://www.ubuntu.com/. (accessed 1/11).

CHAPUIS, O. AND ROUSSEL, N. 2005. Metisse is not a 3D desktop! In *Proceedings of the User Interface Software and Conference Technology (UIST'05)*. ACM, 13–22.

CISCO SYSTEMS INC. 1997. WebEx, http://www.webex.com/. (accessed 1/11).

CITRIX SYSTEMS, INC. 1997. Citrix Online, http://www.citrixonline.com/. (accessed 1/11).

EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J., SMITH, T., AND IZADI, S. 2002. Challenge: Recombinant computing and the speakeasy approach. In *Proceedings of the 8th Annual international Conference on Mobile Computing and Networking (MobiCom'02)*. ACM, New York, 279–286.

EQUALIZER GRAPHICS. 2008. http://www.equalizergraphics.com/ (accessed 1/11).

FLANAGAN, D. 2006. *JavaScript: The Definitive Guide.* O'Reilly Media, Inc.

GOOGLE, INC. 2011. Android, http://www.android.com/. (accessed 8/11).

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2011. *Java Language Specification*, 2nd Ed. The Java Series. Addison-Wesley Longman Publishing Co., Inc.

GOSLING, J., ROSENTHAL, D., AND ARDEN, M. 1989. *The NeWS Book: An Introduction to the Networked Extensible Window System*, Sun Microsystems.

HOWARD, M. AND LEBLANC, D. 2003. *Writing Secure Code*, 2nd Ed. Microsoft Press.

HUTTERER, P., AND THOMAS, B. H. 2007. 'Groupware support in the windowing system. In *Proceedings of the 8th Australasian Conference on User Interface (AUIC'07)*. Australian Computer Society, Inc., 39–46.

INTERNATIONAL BUSINESS MACHINES CORP., 2011. NMON performance: Nigel's Monitor. http://www.ibm.com/developerworks/aix/library/au-analyze_aix/. (accessed 1/11).

IZADI, S., BRIGNULL, H., RODDEN, T., ROGERS, Y., AND UNDERWOOD, M. 2003. Dynamo: A public interactive surface supporting the cooperative sharing and exchange of media. In *Proceedings of the User Interface Software and Technology Conference (UIST'03)*. ACM, 159–168.

JIANG, H., WIGDOR, D., FORLINES, C., BORKIN, M., KAUFFMANN, J., AND SHEN, C. 2008. LivOlay: Interactive ad-hoc registration and overlapping of applications for collaborative visual exploration. In *Proceeding of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. ACM, 1357–1360.

JOHANSON, B., FOX, A., AND WINOGRAD, T. 2002. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Comput. 1*, 2, 67–74.

LIU, Z. 2007. Lacome: A cross-platform multi-user collaboration system for a shared large display. Computer Science, University of British Columbia. http://hdl.handle.net/2429/378.

MICROSOFT CORPORATION. 2011a. Kinect, http://www.xbox.com/en-US/kinect. (accessed 1/11).

MICROSOFT CORPORATION. 2011b. logman, http://technet.microsoft.com/en-us/library/bb490956.aspx. (accessed 1/11).

MICROSOFT CORPORATION. 2011c. NET Framework http://www.microsoft.com/net/. (accessed 6/11).

MICROSOFT CORPORATION. 2011d. Network Projectors. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/aa934598.aspx. (accessed 6/11).

MICROSOFT CORPORATION. 2011e. Silverlight, http://www.microsoft.com/silverlight/. (accessed 6/11).

MICROSOFT CORPORATION. 2011f. Windows Phone 7 Series, http://www.windowsphone7.com/. (accessed 6/11).

MICROSOFT CORPORATION. 2011g. Visual C#, http://msdn.microsoft.com/en-us/vcsharp/default.aspx, 2000.

MYERS, B. A. 2001. Using handhelds and PCs together. *Comm. ACM 44*, 11, 34–41.

NAEF, M., LAMBORAY, E., STAADT, O., AND GROSS, M. 2003. The blue-c distributed scene graph. In *Proceedings of the Workshop on Virtual Environments (EGVE'03)*. ACM, 125–133.

NICHOLS, J., MYERS, B. A., AND ROTHROCK, B. 2006. UNIFORM: Automatically generating consistent remote control user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'06)*. ACM, 611–620.

NINTENDO CO., LTD. 2011. Wii, http://www.wii.com/. (accessed 1/11).

OLSEN, D. R. 1999. Interacting in chaos. *Interactions*, 42–54.

OLSEN, D. R., CLEMENT, J., AND PACE, A. 2007. Spilling: Expanding hand held interaction to touch table displays. In *Proceedings of TABLETOP '07*. IEEE Computer Society. 163–170.

OLSEN, D. R., HUDSON, S. E., VERRATTI, T., HEINER, J. M., AND PHELPS, M. 1999. Implementing interface attachments based on surface representations. In *Proceedings of the Symposium on Human Factors in Computing Systems (CHI'99)*. ACM, 191–198.

OLSEN, D. R., NIELSEN, S. T., AND PARSLOW, D. 2001. Join and capture: A model for nomadic interaction. In *Proceedings of the User Interface Software and Technology Conference (UIST'01)*. ACM, 131–140.

OPREA, A., BALFANZ, D., DURFEE, G., AND SMETTERS, D. K. 2004. "Securing a remote terminal application with a mobile trusted device. In *Proceedings of the 20th Annual Computer Security Applications Conference.* 438–447.

ORACLE CORPORATION. 2011a. Java documents on DataOutputStream, Oracle Corporation, http://download.oracle.com/javase/6/docs/api/java/io/DataOutputStream.html. (accessed 1/11).

ORACLE CORPORATION. 2011b. *Java media framework*, Oracle Corporation, http://java.sun.com/javase/technologies/desktop/media/jmf/. (accessed 1/11).

ORACLE CORPORATION. 2011c. OpenJDK, Oracle Corporation, http://openjdk.java.net/. (accessed 1/11).

PAEK, T., AGRAWALA, M., BASU, S., DRUCKER, S., KRISTJANSSON, T., LOGAN, R., TOYAMA, K., AND WILSON, A. 2004. Toward universal mobile interaction for shared displays. In *Proceedings of the Computer Supported Cooperative Work Conference (CSCW'04)*, ACM, 266–269.

PERLIN, K. AND FOX, D. 1993. Pad: An alternative approach to the computer interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'93)*. ACM, 57–64.

PETZOLD, C. 2006. *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation*, Microsoft Press.

PIERCE, J. S. AND MAHANEY, H. E. 2004. Opportunistic annexing for handheld devices: Opportunities and challenges. In *Proceedings of HCIC (HCIC'04)*.

REALVNC LTD. 2011. RealVNC, http://realvnc.com/. (accessed 1/11).

RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. 1998. Virtual Network Computing. *IEEE Internet Comput. 2*, 1.

SCHEIFLER, R. W. AND GETTYS, J. 1986. The X window system. *ACM Trans. Graph. 5*, 2, 79–109.

SHARP, R., MADHAVAPEDDY, A., WANT, R., AND PERING, T. 2008. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys'08)*. ACM, 94–105.

SHARP, R., SCOTT, J., AND BERESFORD, A. R. 2006. Secure mobile computing via public terminals. In *Proceedings of the International Conference on Pervasive Computing (PerCom'06)*. IEEE Computer Society, 238–253.

SHEN, C., VERNIER, F. D., FORLINES, C., AND RINGEL, R. 2004. DiamondSpin: An extensible toolkit for around-the-table interaction. In *Proceedings of the Symposium on Human Factors in Computing Systems (CHI'04)*. ACM, 167–174.

SCHILIT, B. N. AND SENGUPTA, U. 2004. Device ensembles. *Computer 37*, 12, 56–64.

SHUEY, D., BAILEY, D., AND MORRISSEY, T. P. 1986. PHIGS: A standard, dynamic, interactive graphics interface. *Comput. Graph. Appl. 6*, 8, 50–57.

SYNERGY. 2011. http://synergy-foss.org/. (accessed 1/11).

TARASEWICH, P., GONG, J., AND CONLAN, R. 2006. Protecting private data in public. In *Proceedings of CHI'06 Extended Abstracts on Human Factors in Computing Systems*. ACM, 1409–1414.

TAN, D. S., MEYERS, B., AND CZERWINSKI, M. 2004. WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In *Proceedings of CHI'04 Extended Abstracts on Human Factors in Computing Systems*. ACM, 1525–1528.

TIGHTVNC GROUP. 2011. TightVNC, http://tightvnc.com/. (accessed 1/11).

THOTA, C. 2005. *Programming MapPoint in .NET*, O'Reilly Media, Inc.

TRITSCH, B. 2003. *Microsoft Windows Server 2003 Terminal Services*, Microsoft Press.

WANT, R., PERINS, T., DANNEELS, G., KUMAR, M., SUNDAR, M., AND LIGHT, J. 2002. The personal server: Changing the way we think about ubiquitous computing. In *Proceedings of the Ubiquitous Computing Conference (UbiComp'02)*.

YUAN, F. 2000. *Windows Graphics Programming: Win32 GDI and DirectDraw.* Prentice-Hall.

YUE, C. AND WANG, H. 2009. SessionMagnifier: A simple approach to secure and convenient kiosk browsing. In *Proceedings of the 11th International Conference on Ubiquitous Computing (Ubicomp '09)*. ACM, 125–134.

Received August 2010; revised February 2011; accepted March 2011