Everything is a Window: Utilizing the Window Manager for Multi-Touch Interaction

Raphael Wimmer
University of Munich
Amalienstr. 17, 80333 Munich, Germany
raphael.wimmer@ifi.lmu.de

Fabian Hennecke
University of Munich
Amalienstr. 17, 80333 Munich, Germany
fabian.hennecke@ifi.lmu.de

ABSTRACT
Interactive surfaces are becoming more and more common in research and everyday use. Recent research has focused on hardware technologies and basic interaction techniques. While the WIMP paradigm is the prevalent standard for desktop computers, there is no consensus yet on similar paradigms for interactive surfaces. In order to explore novel interaction paradigms for interactive surfaces, researchers commonly use specialized frameworks like the Microsoft Surface SDK or write their own middleware, forfeiting many advantages of existing operating systems and closing out legacy applications. This paper proposes an approach for building multitouch-enabled desktop environments based on common UNIX paradigms and components. By extending the system's window manager researchers can transparently control user input and graphical output on interactive surfaces, simultaneously supporting both multitouch-capable and legacy applications.

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces

General Terms
Human Factors

Author Keywords
window manager, unix, compiz, x11, multi-touch, curve

INTRODUCTION
In the context of our research project *Curve* we are exploring the use of interactive surfaces in office environments. We believe that this is an area where interactive surfaces can provide significant benefits, once ubiquitous large interactive surfaces become economically feasible. The Curve desk employs a large interactive surface that consists of a vertical and an horizontal segment connected by a curved segment (Figure 1). This allows for seamless touch interaction across the whole surface while retaining the respective advantages of a physical desktop and a computer screen [6]. A major goal of this project is to explore how different input modalities like mouse, direct touch, or pen input are used in real work scenarios, and how the different segments of our interactive surface affect the interaction.

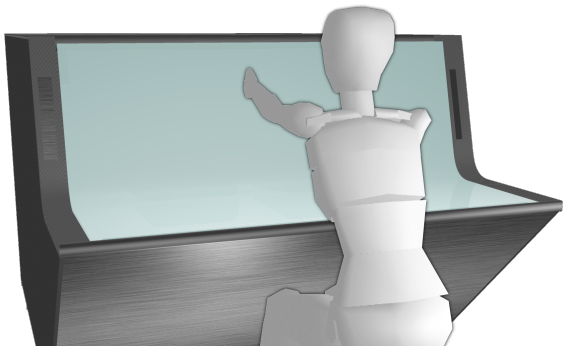

Figure 1. Curve is a desktop environment that combines a horizontal and a vertical surface. This gives the user the opportunity to choose the surface that is best suited for a particular task.

The experience we collected so far in this project leads us to the following assumptions about real-world scenarios for interactive surfaces:

- Office workplaces are important use cases for interactive surfaces. Many people sit in front of their office PC for several hours a day. Enhancing their productivity will have significant and quantifiable impact. Long-term studies are needed for investigating productivity gains.
- Single full-screen applications are not well suited for handling office tasks. Like with current WIMP interfaces, users will need to view multiple applications at the same time and exchange data between them.
- Legacy, single-pointer applications are here to stay for a while. While it is technically possible to augment them with rudimentary multi-touch support [3], the effort needed is significant. Therefore, users will need to run multi-touch applications and legacy applications simultaneously, side-by-side.

Current multi-touch frameworks offer a multi-touch-capable canvas which is embedded into a window that is managed by the operating system. Usually, multi-touch applications are run in a full-screen mode. As widgets for multi-touch UIs are usually larger than WIMP UI widgets, running sev-

Submitted to workshop "Engineering Patterns for Multi-Touch Interfaces" in conjunction with EICS 2010.

eral multi-touch applications side-by-side is In the following, we argue that multi-touch systems should support multiple simultaneously running applications, including applications written using different toolkits and legacy applications. As window managers already implement many features needed for multi-touch systems, we suggest, using a window manager for offering basic multi-touch interactions while supporting legacy applications.

SUPPORTING LEGACY APPLICATIONS

In the following we use the term *legacy application* for software that recognizes only single pointer events, compared to *novel applications* which recognize and use richer input like multiple touch points or contact area. A typical *novel application* is the prevalent "photo sorting" demo offered by many multi-touch SDKs. *Legacy applications* include e.g. word processors, web browsers, or file managers.

Benko et al. [2] conducted a survey of researchers, developers, and users working with multi-touch systems. For long-term individual use, support for legacy applications was considered very important. Such legacy applications are prevalent on existing computers and have evolved over several years of maintenance. They applications provide powerful and essential features for many users. Banning them from interactive surfaces will inevitably lessen the productivity of such a system's users. However, it is unfeasible to port all those applications to a new input model. Additionally, it is not clear, how multi-touch interaction for such applications might look like. On the other hand, confining novel applications to single windows similar to legacy applications, would certainly reduce their expressiveness. It might also hinder adoption of novel applications, as multi-touch input would need to happen in certain confined screen areas - which are not always at ergonomic positions.

These problems can be addressed by giving each type of application its preferred space: Novel applications should use the whole screen space while legacy applications are presented as windows within the same screen space (Figure 2).

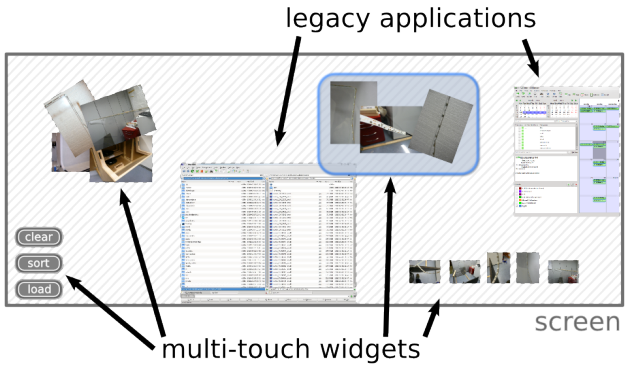

Figure 2. Novel multi-touch applications and legacy applications should share the same screen space. Legacy applications are rendered within a window - the way they were designed. Novel applications may place their UI widgets across the whole screen.

Implementing such a system using current multi-touch toolkits poses some challenges:

- Embedding: Graphical output of legacy applications has to be redirected into an image that is displayed within the toolkit drawing area. Toolkit touch events have to be translated into the operating system's pointer events before sending them to the legacy application.
- Cross-Application Interaction: How can *drag and drop* of an object from a legacy to a novel application and vice versa be implemented?
- Flexibility: While legacy applications can be written in any programming language, novel applications have to use the SDK's API.

We propose a flexible solution for these combined systems: Instead of re-implementing an operating system within the multi-touch toolkit, we use the operating system's window manager as a middleware that combines legacy and novel applications. In the following we explain this concept and discuss its benefits and drawbacks. While the following description uses UNIX concepts and software, the general idea holds true for other operating systems.

UTILIZING THE WINDOW MANAGER

One of the fundamental principles of UNIX systems is *Everything is a File* - be it system settings, external devices, pipes, or internal components of a computer. While this concept has some exceptions it allows for transparent reading and manipulation of a multitude of digital objects. Demonstrating the power of this concept, Ballesteros et al. [1] have implemented a distributed pervasive computing framework on top of file system operations.

X11, the default graphical I/O subsystem for UNIX has a similar concept: *Everything is a Window*, be it the desktop, application windows, menus, or other widgets. While applications can manage their canvas on their own, toolkits like QT or GTK use embedded X11 windows for representing certain UI widgets. They therefore expose these widgets to the windowing system. Hutterer et al. [5] have implemented native multi-pointer support for X11.

In order to display (and interact with) legacy applications alongside novel multi-touch UIs, the graphical output of both must be combined. Additionally, multi-touch input events have to be translated int mouse events for legacy applications. A window manager is actually doing exactly these things - it can modify input events before they reach the application, and modify graphics output before it is handed to the graphics card. The window manager captures all events that are sent to the root window by the X server. It subsequently propagates events to the appropriate windows. The window manager can therefore filter and manipulate input events, and generate new events. For example, the window manager can filter multi-touch events, so that legacy applications receive only single-pointer events. Compositing window managers ,like Compiz[1], redirect all window contents

[1] `http://www.compiz.org`

to an off-screen buffer before combining them and transferring them to the GPU. Therefore the window manager is also able to filter and manipulate windows, and augment them with additional graphics like window decorations. Current compositing window managers use OpenGL acceleration for mosts tasks. For example, the window manager can prewarp the screen content, so it appears undistorted when projected onto non-planar surfaces.

Compiz is an open-source compositing window manager for X11. Like the window managers in Windows 7 and Mac OS X, Compiz offers multiple virtual desktops, smooth window scaling, 2D and 3D effects, and many other features. Compiz consists of a rendering core and a sophisticated plugin system. Almost all components (virtual desktops, application switcher, wallpaper, etc.) are realized as plugins that can be configured, enabled and disabled at runtime. Such plugins also offer multi-pointer input and freely rotatable windows. For our Curve system we have implemented a plugin that uses a distortion map to pre-warp the output of two projectors. The plugin can be configured and enabled/disabled at runtime. However, the window manager does not only act as a gatekeeper for user input and graphical output. It also runs alongside applications. In the following section we describe how the window manager can therefore be used as a flexible middleware for multitouch interaction.

EVERYTHING IS A WINDOW

In many cases, interactive surfaces are used to create, move, and/or manipulate virtual objects on the screen. A window manager can handle such interactions with minor modifications. In the following, we describe an exemplary configuration of such a system. Legacy applications are handled just the same way as before: The window manager receives input events and either manipulates the window accordingly or propagates the event to the application. This concept of the window manager handling multi-touch input has also been described by Cheng et al. [4]. In order to give the window manager control over a novel application, all multi-touch UI widgets are implemented as X11 windows. For example, when media files shall be displayed, a launcher application starts a simple viewer application for every file. This application creates an undecorated window (without title bar, borders) into which a preview of the file is rendered. The window also gets assigned custom properties (X11 atoms) that contain the corresponding file name and process id. Thus, the window manager and X11 applications like *xprop* can access this information. The viewer application is responsible for generating a scaled preview when its window is resized. Using an own process for every media file, generates a memory overhead. However, a single malformed media file may only crash its own viewer application, not the whole system. Additionally, this approach automatically makes use of multiple CPU cores. Buttons, sliders, and similar widgets can be implemented as a window, too. A generic launcher could generate UIs based on stored definitions. Additionally, standalone applications can be written, that do not depend on a media file. Using common touch gestures the user can move all widgets around the screen and interact with them.

APPLICATIONS

Interaction concepts can be implemented either as a plugin to the window manager or as a standalone application that polls window positions and acts upon changes. Usually, the plugin approach is both faster and more flexible. Simple systems can be even prototyped with shell scripts. For example, one might want to implement a photo viewing and sorting application. This application should display a set of images on the desktop that can be moved, scaled, and rotated with multitouch gestures. When dragging such an image onto the window of an image editor this image should be opened in the editor. As described above, a simple launcher application would generate a window for every image file. Users can drag them around the screen, arrange them in piles, make them transparent, or manipulate them any way the window manager allows for. A shell script continuously scans the window tree for the editor window and the image windows. Once the distance bewtween an image window and the editor window gets smaller than a certain threshold, the shell script retrieves the file name from the image window and passes it to the editor application. This approach requires no knowledge of window manager internals.

Further applications of this concept include:

- custom window decorations, that arrange tool or action buttons around an object.
- automated arrangement of objects, e.g. spreading all objects across the screen.
- drawing onto the desktop can be rendered into a transparent window on top.

CONCLUSION

In this paper we propose using the system's window manager for managing directly manipulable widgets as well as legacy applications. Using windows as widget containers eases the development of such widgets, as any programming language can be used. New interaction and visualization concepts can be implemented as a window manager plugin. Many ideas can also be implemented by manipulating window properties from within a shell or python script. A legacy application may not even be aware of multi-touch events, receiving only single-pointer events. Nevertheless, the application's window can be manipulated by multi-touch gestures. Thus, the gap between novel and legacy application gets smaller. While our approach will surely not fit all needs it seems very flexible, robust, and easy to learn. It should be noted that only some of the ideas presented here have been implemented yet.

ABOUT THE AUTHORS

Raphael Wimmer is a PhD student in the Media Informatics Group at the University of Munich, Germany. His research focuses on technologies for touch-sensitive surfaces. **Fabian Hennecke** is a PhD student in the Media Informatics Group at the University of Munich, Germany. His research focuses on non-planar interactive surfaces.

REFERENCES

1. F. Ballesteros, E. Soriano, G. Guardiola, and K. Leal. Plan B: Using Files instead of Middleware. *IEEE Pervasive Computing*, pages 58–65, 2007.

2. H. Benko, M. R. Morris, A. J. B. Brush, and A. D. Wilson. Insights on interactive tabletops: A survey of researchers and developers, 2009.

3. G. Besacier and F. Vernier. Toward user interface virtualization: legacy applications and innovative interaction systems. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 157–166. ACM, 2009.

4. K. Cheng, B. Itzstein, P. Sztajer, and M. Rittenbruch. A unified multi-touch and multi-pointer software architecture for supporting collocated work on the desktop. Technical Report ATP-2247, NICTA, Sydney, Australia, September 2009.

5. P. Hutterer and B. Thomas. Groupware support in the windowing system. In *Proceedings of the eight Australasian conference on User interface-Volume 64*, page 46. Australian Computer Society, Inc., 2007.

6. F. H. S. B. H. H. Raphael Wimmer, Florian Schulz. Curve: Blending Horizontal and Vertical Interactive Surfaces. In *Adjunct Proceedings of the 4th IEEE Workshop on Tabletops and Interactive Surfaces (IEEE Tabletop 2009)*, Nov. 2009.