Toward User Interface Virtualization: Legacy Applications and Innovative Interaction Systems

Guillaume Besacier[1, 2]
guillaume.besacier@limsi.fr

Frédéric Vernier[2]
frederic.vernier@limsi.fr

[1] LRI - Université Paris-Sud & CNRS
Bâtiment 490, Université Paris-Sud
91405 Orsay Cedex, France

[2] LIMSI-CNRS
BP 133
91403 Orsay Cedex, France

ABSTRACT

Single-user, desktop-based computer applications are pervasive in our daily lives and work. The prospect of using these applications with innovative interaction systems, like multi-touch tabletops, tangible user interfaces, large displays or public/private displays, would enable large scale field studies of these technologies, and has the potential to significantly improve their usefulness and, in turn, their availability. This paper focuses on the architectural requirements, design, and implementation of such a technology. First, we review various software technologies for using a single-user desktop application with a different model of user inputs and graphical output. We then present a generic technique for using any closed-source or open-source application with different input and output devices. In our approach, the application is separated from the user input and graphical output subsystem. The core part of the application runs in a system-specific virtual environment. This virtual environment exposes the same API as the removed standard subsystems. This eliminates the need to rewrite the "legacy" application and provides high performances by using the application native way to communicate with the system.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design.

Keywords: Novel Interaction Systems, Toolkit, Legacy Applications

1. INTRODUCTION

Single-user, desktop-based computer applications are pervasive in our daily lives and work. Being able to use these applications with novel interactive systems, like multi-touch tabletops, tangible user interfaces, large displays or public/private displays, is a requirement for a large deployment of these systems in a production environment. Indeed, these applications, whether they are widely used or specialized business applications, are essential to their users.

Widely used applications (word processor, Internet browser, etc.) could be rewritten from scratch for each new system. Some projects have created clones of these applications for their research prototypes. It would require a large amount of work for each application and each system and is, in our opinion, out of the scope of most research projects.

Specialized business applications, on the other end, don't even have this possibility. It seams businesses will not massively embrace, for example, tabletop computing if it doesn't interface with its existing applications for workflow, modeling, productivity, asset management, accounting, etc. These applications may be specific to a company, even developed in-house: it would be unreasonable to consider rewriting each one of them for each new platform.

On the other hand, novel interactive systems use new kinds of hardware, beyond the desktop keyboard, mouse and screen, that introduce new possibilities and new challenges for user interface. For example, a high-resolution widescreen display wall [7] is not well suited for vertical scrolling in a word processor: displaying several full-height pages side by side seems better than full-width pages on top of each other. User input may be fundamentally different, yet the actions offered by each application are the same than on the desktop. With a tangible user interface, one could use a tangible interactor to represent a folder [23], and put the interactor on a document window to trigger the action "save this document to that folder", instead of clicking on a toolbar "save" button and selecting a folder with a dialog box.

On a shorter-term scale, existing applications on novel interactive systems would enable large scale field studies. Some novel interactions or systems may not fully realize their potential until they are used by real users with real applications and data. This practice of long-term field studies has long been part of the scientific procedure, and has been used with success by HCI researchers (e.g. [10, 3] studies lasted several months). A technology such as the ones reviewed or proposed in this article, acting like "glue" between the research prototype and the users' activities, would be a breakthrough in bringing this tool to the novel interactive system researcher.

The purpose of this article is to discuss potential technologies to run existing desktop applications on the novel interactive systems that use a different model of user inputs and graphical output. The technology should be able to smoothly integrate the legacy applications in the novel interactive system, by modifying the application processing of its user inputs and the application graphical user output rendering.

It presents a complex challenge of balancing the functionalities, performances and amount of this "glue" between the existing applications and the novel interactive system. We think such a technology would indeed be a useful tool available to the interactive systems researcher, but also presents some very interesting problems in its own right. A parallel could be drawn with the physicist toolbox which conception requires large amounts of research, before it is even used for actual physics researches. For example, in the search for the "God particle", a bubble chamber is used. This tool was invented in the 50's by Donald A. Glaser, for which he was awarded a Nobel Prize in Physics.

We then present in details the user interface toolkit rewriting technology. It emulates the standard toolkit the application knows how to "talk" to and adapts the API calls for the innovative interaction system toolkit. We implemented two proof-of-concept implementations with this technology. In the first one, we extended Microsoft Windows window management with innovative window management techniques like rotating windows, peeling back, stacking, zooming, etc. The second test removes the pull-down and popup menus of an application and sends them to another graphical environment. The control environment displays the application menus in a suitable way and sends back menu events to the applications.

2. SOFTWARE TECHNOLOGIES FOR USING A SINGLE-USER DESKTOP APPLICATION WITH A DIFFERENT MODEL OF USER INPUTS AND GRAPHICAL OUTPUT

We identified several technologies to achieve our goal of using existing desktop applications with the novel interactive systems. Note that we call "technology" a set of software constructs that would allow us to achieve our goal, while we call the implementation details and the individual software construct "technique". These technologies purpose is to link the application with the environment provided by the innovative system. All these technologies interface with the legacy applications in some original ways not envisioned by the applications developers. The position of this interface in the input processing chain and the graphical rendering chain is the primary difference between the technologies. To compare them, we identified several metrics.

2.1 Metrics

2.1.1 Data Structure

The first metric is the amount of structure in the data exchanged with the application. It can range from completely unstructured data to fully structured data. Structured output data may be a description of each component of the interface and their relations to each other, or a functional description of the application internal data and the available actions. Unstructured data, on the other hand, may be a bitmap of the application windows, as rendered on the desktop computer screen. For input, structured data revolves around sending high level commands, either interaction oriented (e.g. the technology inserts a "menu item have been selected" event in the application event queue, without justifying its source in a mouse click, a keyboard shortcut, a vocal command, or the novel input device) or action oriented (e.g. "load a file" event). In both cases, it requires some sort of information about the application interface or actions: it needs to know what actions the application supports, what format the application expects, etc. Sufficiently structured output data provide this information. Unstructured input data is achieved by simulating the physical devices the application expects (most likely a mouse and a keyboard).

2.1.2 Flexibility

The second metric is the flexibility in changing the internal mechanisms of the default user input and graphical output subsystems. Examples of such changes include: drawing each window in its own buffer to transform a desktop situation where occlusion between windows would have occurred to a large display situation of the same windows without occlusion; modifying the UI components state machines to transform a single focus desktop situation to a multi-user multi-touch multi-focus situation; changing modal dialog box to block only the user who invoked it; etc. While the application being use may influence this metric (e.g. an application may support receiving two "mouse button down" events at different locations, even it is not supposed to happen), achieving a high level of flexibility requires a high level of structure.

2.1.3 Performance

The third metric is the performance of the interaction. We are designing interactive systems; we need them to be responsive in both registering an event and rendering the graphical output [11]. It is influenced by the amount of processing remaining to be done by the technology and the amount of unneeded or irrelevant processing already done by the application (or the default user input and graphical output subsystems).

2.1.4 Availability and Reusability

The fourth metric is the availability and the reusability. The availability determines the global usefulness of the technology (will it work with any application?), while the reusability determines the immediate usefulness of a prototype (does it work with the same code for all applications or does each application require its own supporting plug-in?).

2.1.5 Difficulty

The fifth and last metric is the difficulty to implement a prototype. Some technologies require few works to have a working prototype, and gradually more work to add functionalities; and some require a lot of work before having a first prototype, but then it will almost be a fully functional system. Of course, some others also require very little work to have a finished product.

We presented five metrics for comparing technologies that would enable single-user, desktop-based computer applications to run on novel interactive systems. This is not a definitive list, but we think it covers the most important points of interfacing an existing, non-cooperative application with a user input and graphical output system it was not conceived for. We will now review six technologies that could achieve our goal, and rate them on a five-value scale ("very low", "low", "middle", "high", "very high") against each of our metrics ("data structure", "flexibility", "availability and reusability", "performance" and "difficulty").

2.2 Technologies

2.2.1 Screenshots

The simplest technology to get the graphical output of an application is beyond any doubt to take screenshots at a regular interval. It copies the image displayed on the desktop screen to a

Table 1: Technologies to use an existing application with a different model of user input and graphical output.

	Data Structure	Flexibility	Availability and Reusability	Performance	Difficulty
Screenshots (output)	Low	Very low	Very high	Low	Easy
Virtual Graphic Card (output)	Low	Low	Very high	High	Middle
Virtual Keyboard and Mouse (input)	Low	Very low to low	Very high	High	Easy
Scripting (both)	Very high	High	Low	Very high	Middle
Accessibility API (both)	Output: Middle Input: High	Middle	Middle	Middle	Middle
UI Toolkit Rewriting (both)	High	Output: Middle Input: High	High	Very high	Hard

technology-controlled memory buffer. With additional information about the size and position of each window (information which is provided by all graphical output systems), it can segment the screenshot in images of each window and thus manage them separately. This technology has a low level of data structure. It is also limited in the ways it can rearrange windows placements or sizes: when two windows overlap on the desktop screen, a part of one of the windows is hidden and cannot be reconstructed. Thus, it can't place this window in positions where it would be fully visible. A similar approach was used to interact with a desktop computer from a handheld device [12]. A full screen screenshot of the desktop is displayed with a controllable zoom on the handheld, and additional information about the menus and the text fields are used to adapt their format for the handheld constraints (multi-column menu and shorter lines of text).

2.2.2 Virtual Graphic Card

The second technology is an improved version of the previous one, using a virtual graphic card or a virtual screen. It is the approach used by most remote desktop software (e.g. VNC [18] uses a virtual "VNC hook" graphic card device driver for Windows). Metisse [4], a "meta window-manager" for implementing novel desktop interaction techniques like rotation or peeling of windows [2], is a modified X-server [19] which renders windows off screen and recomposes their images in a 3D space. While this technology still has a low level of data structure, it is more flexible than screenshots. One can create a virtual screen with a resolution so high that no windows will ever overlap, or create several dozens of virtual screens and open a single application per screen, thus grouping all the application windows together. Moreover, this technology allows for high frame rate with typical desktop content.

2.2.3 Virtual Keyboard and Mouse

Both screenshots and virtual graphic card technologies capture the graphical output of an application, but don't allow sending back events to it. The third technology is their user input counterpart: a virtual keyboard and a virtual mouse. It is integrated in the Metisse software, which also allows specifying which window should receive the event. This gives some flexibility: the virtual mouse cursor can be on two unrelated windows at the same time to allow concurrent interaction by two users (but two users still can't interact with the same window without turn taking). Some other platforms are limited to a single keyboard and a single mouse shared by all applications. The level of structure of these events is low. Most likely, the user of the innovative system will have to use a mouse-like interaction (with notions of current cursor position, click, etc.), and the processing will be limited to transforming the coordinates of the event from the large display, tabletop,… coordinates to desktop screen coordinates.

These first three technologies share an advantage: they are compatible with all applications and highly generic. While some additional rules or processing may be added for a particular application to integrate it more consistently with the innovative interactive system, any application can have at least their basic functionalities without specific programming.

2.2.4 Scripting

The fourth technology uses a completely different philosophy. Some applications expose a scripting interface. Its purpose is to facilitate tasks automation (without using a "macro" of raw mouse and keyboard events and replaying it) and applications interactions. For example, an application might send a specially formatted message to the web browser application to navigate to a certain web page, a word processing application might ask a spreadsheet application for some data to create a mailing, or a script might communicate with the file manager to backup some folders. For our purpose, scripting allows to access an application data without formatting for a particular type of viewport. A part (depending on the application implementation of scripting) of the burden of presenting the data in a visual way will be upon the technology. While it enables interaction system dependent way of presenting the data, it means a very low reusability, as each application has different data. For user input, scripting also has a high level of structure and flexibility. Some operations are standardized among applications (printing the data, saving to a file,…). One could use a personalized "select a file" dialog box, even drop the file and folder metaphor, and invoke the "load a file" command with the selected file name. Nevertheless, most operations are application-dependent, which make scripting a technology with low reusability. The communications with the application are minimal (structured data is lighter than images, and only need to be refreshed when the data changes, and not when the viewport changes) and the user interface and the interactions are managed by the novel interactive system natively, performances are thus maximal. However, this technology is only available for applications that do implement scripting. Most Mac OS applications tend to have scripting capabilities and use a standard scripting language [1] introduced in 1992, while other OS applications use incompatible scripting languages, even when they do have scripting.

2.2.5 Accessibility API

The fifth technology uses the accessibility API [16, 22] to get a description of the user interface and adapt it to the novel interactive system. The accessibility API is designed to provide alternative user interfaces adapted to disabled users. It can be used to plug a user interface for a novel interactive system. The API is consistent among different OS, probably due to its recent conception, meaning a good reusability. Most recent applications support accessibility, and several countries enacted laws to make it mandatory for future applications, but older applications cannot use this technology. An interesting use of this technology is Facades [21], which allows end-users to recompose the interfaces of their applications. The accessibility API is used to query the current position, size, type, and state of the visible widgets of an application. On the input side, all possible widget actions can be activated via this API (e.g. one can cause selection events, trigger buttons, etc.). While it is entirely appropriate to plug a voice command interface to select among the visible buttons, it lacks a bit of flexibility when it comes to switching to a different metaphor than the desktop, due to its close links with it. Indeed, it aims at providing access to traditional desktop computer environment to more people rather than trying to offer an alternative access to its data and processing capabilities.

2.2.6 User Interface Toolkit Rewriting

The last technology is to rewrite parts of the user interface toolkit. Virtually all graphical applications use a toolkit to build their user interfaces. Example of toolkits include GTK+, Qt, Xaw, Motif for X Window based operating systems; Cocoa, Carbon, Toolbox for Mac OS; and win32 for Microsoft Windows. These toolkits are an abstraction layer between the application and the operating system: they manage an event queue, translate raw mouse and keyboard events in high level commands (e.g. whether the user clicked on the "print" menu item or used its keyboard shortcut, the application receive a "print" event), provide a library of user interface widgets, with their own interaction state machines (e.g. when the user click on a window title bar, the application does not need to track the mouse moves, compute the offset between two successive cursor positions and move the window accordingly) and rendering routines, and more. By rewriting parts of the user interface toolkit used by the application, one can selectively extends or changes the behavior of individual services provided by the toolkit. This approach was used as early as 1992 in Mercator [14] to translate calls to the X Window GUI API to an audio interface.

This technology can achieve a high level of output data structure, in both ways given in the metric description. Indeed, by rewriting the functions related to UI components creation, it can have a structured description of the user interface; while by rewriting the functions related to common actions (clipboard, open/save/print standard dialog boxes, drag and drop and data exchange, etc.), it can have a limited functional description of the application internal actions and data representation. This technology also has the potential for a high flexibility. For example, it could be used to rewrite the event processing functions related to text boxes to allow two points of insertion for collaborative editing of the same text; or to change the state machine associated with checkboxes to activate them with crossing instead of clicking. This technology can achieve high level of performances, because it communicates with the application through the toolkit API the application was design for. This also eliminates the need to rewrite the application, as long as it uses the right toolkit. However, it is the most difficult technology to implement, and needs a large volume of code before producing any results. Several toolkit functions would need to be rewritten for each desired change, and great care should be given to potential far-reaching consequences in others areas of the toolkit.

Table 1 provides a synthesis of the technologies and how they rate against our metrics.

2.3 Choice of a Technology

We previously created with the DiamondSpin [20] toolkit for multi-touch tabletop applications. It is designed as an extension of the Java-SWING user interface toolkit: for each standard class (JFrame, JMenuBar, JComboBox, etc.) we created a new class inheriting from the base class, and added and/or redefined some of its methods. Internally, we completely redeveloped the event dispatching system to allow concurrent threaded user interaction, and the display system to use hardware accelerated 3D graphics.

A similar approach is used by the subArctic toolkit [7], using the underlying programming language concepts of subclassing of drawable objects (in conjunction with wrapping). The application programmer is not required to know about these subclasses to write his application [5].

DiamondSpin was used to create dozens of applications, and we noticed a similar trend: we almost never directly called the DiamondSpin methods we had added. We just used the existing methods, with the extended semantics we gave them (for example, we use the existing setFocus to raise a window to the front, knowing that it wouldn't remove the focus from the windows owned by the other users of the tabletop). The few additional public methods we created were for toolkit and hardware initialization and for setting up the user environment (sharing policy, windows orientation, and others tabletop-specific application-global properties).

This approach considers tabletop computing as an evolution, not a revolution, and our experience seems to fit this philosophy quite well. In the same way, desktop operating systems evolution obsoletes some functions (e.g. in Microsoft Windows, the LimitEmsPages API function now does nothing at all because modern computers don't have active 64K memory segments anymore), extends some functions (e.g. CreateWindowEx take more arguments than CreateWindow, the later calling the former with default values for the new arguments) or changes the semantics of some functions (e.g. the DS_SYSMODAL flag doesn't create a system-modal window, because there in no longer the concept of system modal window (it contradict the concept of multitasking); instead it sets the WS_EX_TOPMOST style, to create a window that always stay on top but is not system-modal).

We think we can generalize this successful previous approach to a larger code base and more novel interactive systems, by rewriting parts of a user interface toolkit.

3. TOOLKIT REWRITING WITH BINARY INTERCEPTION

We aim at rewriting parts of a user interface toolkit, with the ultimate goal to use the existing desktop applications with novel interactive systems that expose a different model of user inputs and graphical output. We also strive to really integrate the application in the interactive system, by modifying the application processing of its user input and the application graphical user output rendering. These modifications shall be driven by the large body of research into interaction techniques and visualizations for each interactive system.

In order to implement these modifications, we must first choose which user interface toolkit to modify. We reckon the various toolkits will expose different API, prompting for firm choice of a single toolkit, and offer support for different applications.

The way to plug our modified toolkit into applications will also influence the availability of our solution. We investigated several ways to link the applications with the modified toolkit.

User interface toolkits comprise thousands of functions; we don't propose to rewrite them all. We see three levels of functionality that could be achieve by modifying various subsets of the toolkit in goal-dependent ways. We ran experiments to identify the implication of various types of modifications, and understand functions interdependency.

3.1 Which Toolkit?

Toolkits are an abstraction layer between the application and the operating system: they manage an event queue, translate raw mouse and keyboard events in high-level commands, provide a library of user interface widgets, with their own interaction state machines and rendering routines, and more. Yet, the different toolkits can vary greatly in their implementation: some toolkits are object-oriented (e.g. Cocoa for Mac OS; Qt and Xaw for X Window) while others are procedural (Carbon and Toolbox for Mac OS; win32 for Microsoft Windows; GTK+ for X Window,…), some toolkits are native (i.e. embedded in the operating system, Carbon, Toolbox, and win32 are native to their respective operating systems; there is no native toolkit for X Window based operating systems) while others are separate layers, often high level, built on top of the native toolkits (Cocoa, MacApp, MacZoop and others for Mac OS; MFC, WTL, VCL, .NET, WPF,… for Microsoft Windows are all built on top of the procedural win32; dozens of toolkits for X Window, built from scratch or from lower-level toolkits, and sometime ported to other operating systems native toolkits). In order to achieve our goal of using existing applications with novel interactive systems, we need to find which toolkit we want to address and partially rewrite. We think the most important point to consider is the number of closed-source applications using each toolkit. From a difficulty of implementation point of view, we should also consider choosing a well documented or open-source toolkit, to minimize the potential far-reaching consequences of our modifications in others areas of the toolkit.

We choose the win32 toolkit. All the applications running under Microsoft Windows (which include the most wanted applications, according to our informal user study) use this toolkit, whether directly or via another higher level toolkit. All the public functions of the win32 API are documented, and Microsoft should disclose (as required by the European Commission) the documentation of its private API and protocols. A subset of the API has been standardized by the European Computer Manufacturers Association as ECMA-234 [6], any application written against this standard is guaranteed to run with any ECMA-234 implementation. Furthermore, we can use the open-source Wine/Winelib [24] and PEACE [17] projects, which aim at reimplementing the win32 API from scratch to run Windows applications with others operating systems.

3.2 Linking the Applications with the Modified Toolkit

The API functions we wish to extend are not part of the applications. They live in external libraries. We have several options to extend them:

- If both the application and the library are open-source, we can modify the library sources and link the application (without modifications) against the new library. In our case, the libraries are not open-source, tough we could use the Wine or PEACE implementation of win32 API as a base.
- If only the application is open-source, we can write a library containing the functions we rewrote and proxy functions for the others. We link our library against the original library (the proxy functions are one-line functions calling the original functions), and the application against our library.
- If the application is not open-source, we need to dynamically intercept its API functions calls and reroute them to our own functions at run-time: we need a binary interception package.

The Center for Bioinformatics and Computational Biology of the University of Maryland uses this last option to achieve a goal similar to ours [13]. They have a large number of bioinformatics applications they wished to use with the distributed computing platform BOINC (SETI@home, the World Community Grid,…). Instead of rewriting the application, and even instead of recompiling them, they wrote an adaptation layer between the legacy applications and the operating system (in their case, mostly file and folder related functionalities, which BOINC needs to handle) and dynamically intercepted the API calls. The adaptation layer functions need to have the exact same arguments and return types as the original API functions, but are free to call others functions (e.g. BOINC functions), the original API (with or without modifying the arguments first), or do any processing they need.

A similar approach is used by Chromium: it intercepts the 3D rendering functions of OpenGL and modifies their behavior on the fly, e.g. to add clipping plane to produce exploded views of 3D architectural environments such as multi-story buildings [15].

As the University of Maryland software includes Microsoft Windows, Mac OS X and UNIX software, they reviewed the binary interception techniques on these three operating systems. In our case, tough, we'll only use the Windows version: the Detours package from Microsoft Research [9].

3.3 Which Kind of Modifications?

During our design sessions, we identified three distinct goals toolkit modifications would enable, which turn out to be three ways of modifying the functions of the toolkit.

We could just want to obtain information about the application. Technologies like the accessibility API already allow to programmatically retrieve information about an application user interface, but we can get more detailed information by intercepting the API calls to the functions which create and/or update the interface. For this goal, very simple functions, just storing and processing the arguments (and/or the return value) of the API calls the application makes, and calling the actual win32 function, are sufficient.

We could want to add behaviors to the application. For example, introducing rotating windows or relocatable popup menus. This goal requires associating additional data to the existing objects, and extending functions that should use our data. As in the previous case, the extended functions call the actual win32 functions to retrieve the "traditional" data associated with an object, while a hash-map can be used to store the additional data. For rotating windows, it would require storing a rotation point and angle for each window, and applying a rotation matrix to the graphical device context before the window gets rendered. Relocatable menus require adding a dragging control to each popup menu, and updating the existing x and y coordinates and calling the menu rendering function when the control is dragged.

Finally, we could want to modify or remove behaviors of the application. For example, activating buttons by crossing instead of clicking, or allowing several users to each have a separate focus. These goals are incompatible with some current data structures used by the win32 API: it has a single variable to store the identifier of the currently focus widget, for example, and it is impossible to store more than one identifier in this space. It requires creating our own data structures, which mean rewriting all the functions that create, destroy or access to these data structures. The new functions would not call the win32 functions they replace, and the original win32 data structure would not be created. In consequence, it is the most difficult type of modification to implement. One must be sure to have rewritten all the functions which use these data structures (a discussion of how to identify these functions is in the following section).

Our first example, activating buttons by crossing instead of clicking, would require a new state machine, new state variables (storing from which side of the button the cursor entered instead of storing if the mouse button is currently pressed) and new events processing functions. The different states a checkbox can have would be changed. These states (detailed in Table 2) are both highly dependant of a mouse cursor styled interaction, and tightly integrated in the existing data structure and state machine of a checkbox.

Table 2: The different states involved in activating a Windows checkbox with the mouse cursor.

☐	☑	When the cursor is not over the checkbox and the checkbox has not been clicked on
☐	☑	When the cursor hover over the checkbox, or when the user clicked on the checkbox and then dragged away from the checkbox with the mouse button still pressed
☐	☑	While the mouse button is pressed and the cursor is over the checkbox

Our second example, allowing several users to each have a separate focus, requires creating a global array of focused widgets, and rewriting the functions that query whether a certain widget is focused (search in the array), gets the focused widget (it returns a single widget, so figure out which user action triggers this function call and return this user focused widget), changes the focused widget (remove the focus from the widget returned by the function that gets the focused widget, and focus the new), and several more. New global variables would need to be created, for example to know which user action triggers this function call, we could create a global "current user" variable. When the application process an event from the multi-user input device, we update this variable, we update this variable with the input device supplied user ID. As the events are always processed sequentially [6], all the API calls the application would make could query this variable to know, e.g., which user focus to change. Should the application start a new thread in response to a user event, this thread would be tagged with the current ID for the rest of its execution.

3.4 Which Functions of the Toolkit?

We conducted an experiment to check the level of functions interdependency and redundancy in the Microsoft implementation of win32. We choose a set of win32 UI functions that is as independent as possible of the other functions: the pull-down and popup menu API. We wrote a menu library that overloads each and every menu API function with a custom function that simply print its name, its arguments and call the original function.

We ran several common applications with this library and found that several functions in the win32 menu API are in fact shortcuts or preprocessors for a few core functions.

For example, the function *LoadMenu(ResourceID)* calls *FindResource* and *LoadResource* with the resource id to load the menu descriptor in memory. It then calls *LoadMenuIndirect(MenuDescriptor)* on that descriptor.

The *LoadMenuIndirect* function parses the in-memory menu descriptor, calls *CreateMenu()* to create an empty menu, *InsertMenuItem(Menu, ItemName,…)* to add the menu items (with the appropriate arguments from the menu descriptor) and calls itself recursively if the descriptor includes a submenu. Thus, if we want to intercept the creation of an application menus (e.g. because we want to use a pie menus library), we just need to rewrite the *CreateMenu* and *InsertMenuItem* functions, without mangling the menu description parser nor having to check whether the application uses *LoadMenu*, *LoadMenuIndirect* or manual *CreateMenu* and *InsertMenuItem* to create its menus.

We took a look at the Wine [24] and PEACE [17] win32 reimplementation projects, and confirmed the calling tree we observed with the instrumented Microsoft implementation is consistent with the calling tree that would be produced by Wine and PEACE code. For example, the code below for *LoadMenu* comes from the Wine project:

```
HMENU WINAPI LoadMenu(HINSTANCE instance, LPCSTR name){
  HRSRC hrsrc=FindResource(instance,name,(LPSTR)RT_MENU);
  if (!hrsrc) return 0;
  return LoadMenuIndirect(
           (LPCVOID)LoadResource(instance, hrsrc));
}
```

And here is the code from the PEACE project:

```
HMENU WINAPI LoadMenu(HINSTANCE h, LPCWSTR name){
  HRSRC r;
  HGLOBAL rh;
  MENUTEMPLATE *tmpl;
  if ((r = FindResource(h, name, (LPCWSTR)RT_MENU)) == 0)
    return 0;
  if ((rh = LoadResource(h, r)) == 0)
    return 0;
  tmpl = LockResource(rh);
  return LoadMenuIndirect(tmpl);
}
```

This confirms both the usefulness of these sources as a tool to understand win32 internals, and the independence of the shortcut and preprocessing functions from the actual data structure of the MENUTEMPLATE and HMENU types. We can change the internal representation of a menu: the functions we have not modified and the applications will use this new data structure through the opaque pointer HMENU.

Among the 39 menu-related functions, we identified 20 "core" functions, including 5 trivial getters.

4. IMPLEMENTATION

In order to validate our approach, we created two proof-of-concept implementations: replacing the pull-down and popup menus of an application by custom menus, and extending the windows management with rotation, zooming, peeling-back, and stacking of windows.

4.1 Replacing Pull-down Menus

Our first proof-of-concept implementation replaces the pull-down menus by a custom implementation. All menus-related activity is sent to another computer via a network link. This computer displays the menus of all connected applications, and allows for selecting menu items.

The library for this example intercepts calls to the menu "core" functions identified previously (detailed in Table 3), but does not call the original win32 function when processing an intercepted call. The library maintained its own data structure of the menus. As the function calls are not forwarded to the original functions, win32 does not create the "real" menus. In consequence, we also rewrote the getter functions to fetch the data from the library data structure.

Table 3: The menu-related functions of the win32 API and how they relate to each other.

API function	Is a shortcut/preprocessor for
CreateMenu DestroyMenu InsertMenuItem	None
GetMenuItemCount GetMenuItemID GetMenuState GetMenuString GetSubMenu	None: access menu structure, but are trivial
GetMenuInfo GetMenuBarInfo GetMenuItemInfo	None: convert the opaque structure in a documented read-only structure
CheckMenuItem CheckMenuRadioItem GetMenuDefaultItem HiliteMenuItem ModifyMenu RemoveMenu SetMenuDefaultItem	None: implement an algorithm which accesses menu structure directly
EnableMenuItem	None: calls several menu functions, and implements a post-processing algorithm
DrawMenuBar	None: calls non-menu-related drawing functions
LoadMenuIndirect	CreateMenu, InsertMenuItem, LoadMenuIndirect (recursive)
LoadMenu	LoadMenuIndirect
CreatePopupMenu	CreateMenu with MF_POPUP flag
DeleteMenu	DestroyMenu and RemoveMenu
InsertMenu	InsertMenuItem with default values for additional parameters
AppendMenu	InsertMenu with position = -1
TrackPopupMenu TrackPopupMenuEx EndMenu	Shortcuts for event queue messaging functions
GetMenu	Calls a generic win32 function to access opaque data structure
SetMenu	IsMenu, generic win32 function to access opaque data structure
IsMenu	Try to GetMenu, and clean-up and return an error if it failled
GetSystemMenu	GetMenu
GetMenuCheckMark-Dimensions GetMenuItemRect	Call generic win32 functions for screen coordinates conversion
MenuItemFromPoint	GetMenuItemRect
SetMenuInfo SetMenuItemBitmaps SetMenuItemInfo	ModifyMenu

The main highlights of this implementation are the functions DrawMenuBar, CreateMenu, DestroyMenu, InsertMenuItem, and SetMenu.

DrawMenuBar is called when a window menu bar needs to be drawn. The library implementation does nothing, as the whole menu system is separated from the win32 windowing system.

CreateMenu, DestroyMenu and InsertMenuItem are used, directly or via the preprocessing and shortcut functions, to manipulate the menus content. The library implementation updates its own menu data structure. The various getters defined in the win32 API are also rewritten to fetch their data from this structure.

The SetMenu function is used to associate a menu with a window. As the menu has never been created by win32, our implementation should not call the original SetMenu function. Instead, it stores the window handle in the menu data. When a menu item is selected, the appropriate message is sent to this window handle.

This set of functions provides enough data to maintain an updated description of every menu used by the applications. This description is sent over the network to the control computer. It displays the menus in a relevant form (for our test, as a textual hierarchy of menus items) and managed the interactions with it. It sends back regular win32 menu messages to the relevant application, which performs the corresponding action.

The library code written for this example comprises 21 functions (the 20 core menu functions and SetMenu), for a total of 500 lines of C code. The control computer software, which receives the menu descriptions over a network link and handles the interactions with the user, consists of 200 lines of Java.

4.2 Extending the Window Management

We added rotation, peeling-back, stacking, zooming, and duplication capabilities to regular windows. We used the DiamondSpin [20] tabletop toolkit to provide the interactions and rendering functions, and a custom library to link the applications with it.

The library intercepts calls to the CreateWindow and DestroyWindow functions. Each time CreateWindow is called to create a user interface window[1], the library calls the equivalent DiamondSpin function, and then calls the original win32 function. It registers the link between the win32 data structure, identified by its opaque HWND pointer, and the DiamondSpin data structure, identified by a DSFrame instance, in a hash-table. Similarly, it intercepts calls to DestroyWindow, decrements the DSFrame instance use count (DiamondSpin being written in Java, the instance will be freed by the garbage collector) and removes the corresponding hash-table entry.

As we are using two independent, separately developed window management toolkits (the win32 API and DiamondSpin), both toolkits store the basic window attributes (position, title,…). During window creation, DiamondSpin attributes are initialized with the parameters passed to win32 by the application. To keep these common attributes synchronized, the library intercepts calls

[1] CreateWindow is also used in win32 to create message only windows, Dynamic Data Exchange (DDE) windows, and others non-visible non-interactive windows.

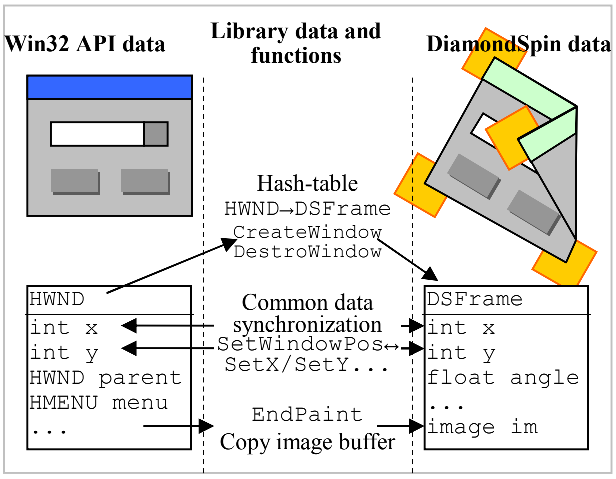

Figure 1: Extending the window management

to win32 functions modifying them, and calls the equivalent DiamondSpin function. Likewise, DiamondSpin initiated modifications (i.e. user interaction results) are monitored (using the listener facilities built in DiamondSpin), and propagated to the win32 data structure.

Modifications to unshared attributes, on either side (e.g. window angle in DiamondSpin, window content in win32), are managed natively by the relevant toolkit.

Finally, the library intercepts calls to the EndPaint win32 function. This function is called when a window visual content has been updated. The content is copied, as an image, to DiamondSpin to fill the corresponding DSFrame. In the current form of this proof of concept, the users cannot interact with this content image from the tabletop. They are limited to the window managements interactors provided by DiamondSpin (moving windows, resizing, zooming, peeling back, stacking,…).

All in all, we extended some thirty win32 functions in this library, a summary of which is provided in Figure 1. This amounts to 5000 lines of code, a large part of which was auto generated by a script. Indeed, most functions just need to convert their arguments from C type to Java type or Java type to C type and call the equivalent function in the other toolkit.

5. CONCLUSION

Single-user, desktop-based computer applications are pervasive in our daily lives and work. These applications are not compatible with new interactive systems. We think they should be: this ability to run existing desktop applications on novel interactive systems is needed for a large deployment of these systems in a production environment.

In this article, we reviewed several approaches to enable this user interface compatibility. We rated these approaches against five metrics: data structure, flexibility, availability and reusability, performance, and difficulty.

We think user interface toolkit rewriting is the most sensible approach. This approach enables communication with the legacy applications in a highly structured way, and provides a reasonable amount of flexibility in changing user input and graphical output internal mechanisms. It also offers native-like performances, thanks to its channel of communication with the applications. Indeed, the approach is to rewrite key functions of a user interface toolkit. The rewritten functions can do anything (including calling the original function, or any other functions) but must have the same arguments list and return type than the original toolkit function. As several such user interface toolkits exist, we choose to extend to win32 toolkit used in Microsoft Windows.

Our two proof-of-concept implementations show it is a viable approach with a great potential. However, it will require a large amount of work to build a working prototype. Writing a full implementation is out of the scope of a research study, tough such a technology would be a revolution for the novel user interfaces and interactive systems community. It would enable creation of high-fidelity prototypes using real applications and data, and opens the door to long term user studies and field studies.

We call this approach user interface virtualization. Indeed, running applications designed for a certain environment in a different environment which emulates the capabilities of the application native environment is reminiscent of CPU virtualization (e.g. running PowerPC applications on an i386 CPU with software emulating PowerPC instructions with a sequence of i383 instructions). User interface virtualization may well become a research field of its own, as it presents some new and very interesting problems of software design and engineering.

We would be delighted to work with the community on an effort to shape the outline of, and implement, a hardware-independent library for using exiting applications with novel interaction systems and interaction styles.

6. ACKNOWLEDGMENTS

We whole heartily thank Olivier Chapuis and Nicolas Roussel from the LRI for their help with the Metisse and the Façade sections.

This work was supported by a grant from the Région Île-de-France (Digitéo Labs).

7. REFERENCES

1. Apple Inc. AppleScript: The Language of Automation. http://www.apple.com/applescript/
2. Beaudouin-Lafon, M. Novel Interaction Techniques for Overlapping Windows. *Proc. UIST 2001*, pp. 153-154
3. Chapuis, O., Blanch, R., Beaudouin-Lafon, M. Fitts' Law in the Wild: A Field Study of Aimed Movements. *LRI Technical Report 1480*, Laboratoire de Recherche en Informatique (2007).
4. Chapuis, O., Roussel, N. Metisse is not a 3D desktop! *Proc. UIST'05*, pp. 13-22.
5. Edwards, W. K., Hudson, S., Rodenstein, R., Smith, I., Rodrigues, T. Systematic output modification in a 2D UI Toolkit. *Proc. UIST'97*, pp. 151-158.
6. European Computer Manufacturers Association. Standard ECMA-234, Application Programming Interface for Windows (APIW), December 1995. http://www.ecma-international.org/publications/standards/Ecma-234.htm
7. Guimbretiere, F., Stone, M., Winograd, T. Fluid Interaction with High-Resolution Wall-Size Displays. *Proc. UIST 2001*, pp. 21-30.
8. Hudson, S. E., Smith, I. SubArctic UI toolkit user's manual. Technical report, College of Computing, Georgia Institute of Technology, 1996.
9. Hunt, G., Brubacher, D. Detours: Binary Interception of Win32 Functions. *Proc. 3rd USENIX Windows NT Symposium*, pp. 135–143.
10. Hutchings, D. R., Smith, G., Meyers, B., Czerwinski, M., Robertson, G. Display space usage and window management operation comparisons between single monitor and multiple monitor users. *Proc. AVI'04*, pp. 32–39.
11. MacKenzie, I. S., Ware, C. Lag as a determinant of human performance in interactive systems. *Proc. INTERACT'93 and CHI'93*, pp. 488-493.
12. Myers, B.A., Peck, C.H., Nichols, J., Kong, D., Miller, R. Interacting At a Distance Using Semantic Snarfing. *Proc. UbiComp'2001* pp. 305-314.
13. Myers, D. S., Bazinet, A. L., Intercepting arbitrary functions on Windows, UNIX, and Macintosh OS X platforms. *Technical Report CS-TR-4585, UMIACS-TR-2004-28*, University of Maryland (2004).
14. Mynatt, E. D., Edwards, W. K. Mapping GUIs to auditory interfaces. *Proc. UIST'92*, pp. 61-70.
15. Niederauer, C., Houston, M., Agrawala, M., Humphreys, G. Non-invasive interactive visualization of dynamic architectural enviuronments. *Proc. I3D'03*, pp 55-58.
16. Parente, P., Clippingdale, B. Linux screen reader: extensible assistive technology. *Proc. Assets'06*, pp. 261-262.
17. PEACE project. http://chiharu.haun.org/peace/status.html
18. Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, Vol. 2, No. 1 (1998), pp. 33-38.
19. Scheifler, R. W., Gettys, J. The X window system. *ACM Transactions on Graphics (TOG)*, Vol. 5, Issue 2 (April 1986), pp. 79-109.
20. Shen, C., Vernier, F., Forlines, C., Ringel, M. DiamondSpin: An Extensible Toolkit for Around-the-Table Interaction. *Proc. CHI 2004*, pp. 167-174.
21. Stuerzlinger, W., Chapuis, O., Phillips, D., Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *Proc. UIST'06*, pp. 309-318.
22. Thatcher, J. Screen reader/2: access to OS/2 and the graphical user interface. *Proc. Assets'94*, pp. 39-46.
23. Ullmer, B., Ishii, H. Emerging Frameworks for Tangible User Interfaces. *IBM Systems Journal*, Vol 39, Issue 3-4 (July 2000), pp. 915-931.
24. "Wine Is Not an Emulator" project. http://www.winehq.org/about/