Customizable Physical Interfaces for Interacting with Conventional Applications

Saul Greenberg and Michael Boyle
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
Tel: +1 403 220 6087
saul *or* boylem@cpsc.ucalgary.ca

ABSTRACT

Most of today's complex software products rely solely on graphical controls (GUI widgets) for user interaction. However, GUI widgets can be difficult to find and use. Physical controls are often simpler to manipulate and arrange sensibly about one's workspace. Thus, we wish to link a physical interface to existing commercial applications, e.g., an office productivity suite. To do so, we must tap in to its functionality in ways that do not require access to its source code. We present our *widget picker/taps* package. It gives developers access to the functionality of an existing application via the semantics of its GUI widgets. This approach works well with many present-day commercial applications, unlike two other common approaches: hooking into application-specific APIs, and simulating raw user input. We present examples that illustrate how this package is used to link existing application widgets to physical controls. Our implementation prompts a number of issues relevant to makers of windowing systems and GUI toolkits.

INTRODUCTION

Almost all desktop computer interaction is done using keyboards and pointers (e.g., mouse, light pen) to interact with graphical UI components (e.g., buttons, menus, sliders). The typical application may present the user with hundreds of different functions, each behind its own GUI control. All systems make a tacit assumption that, with the exception of typing and keyboard shortcuts, these controls should be on-screen and should be accessed with the mouse. Also, it is assumed that invoking functionality this way is an efficient interaction method.

We challenge these assumptions. First, it can be hard to find the GUI control needed for a given application function because the controls are hierarchically nested in dialog boxes and menus. This is done to optimize use of screen real estate. Second, to minimize the effort associated with this hierarchical navigation, some GUI controls are brought to the top-most level by clustering them into tool palettes. These tool windows remain visible on the display at all times, and thus compete with the application itself not only for display space, but also for the user's attention. Third, while nearly all GUI controls rely on the mouse and display for input and output, these are not necessarily the best devices for any given control task. They provide few cues as to their behaviour beyond mouse pointer shape and the use of shading and highlighting in the control's graphical appearance. Also, although the mouse offers (virtually) unlimited motion in a 2D plane, many GUI widgets do not need both degrees of freedom. Consider, for example, a trackbar (slider) control: it can only be moved along a straight line. Thus, the unconstrained 2D nature of mouse movement does not match the constrained 1D nature of a trackbar.

By contrast, physical interface components—e.g., push button switches, rheostats (dials), sliders, and light or pressure sensors for input, and LEDs, servo motors, and DC power supplies for output—have a number of properties that complement graphical controls.

- *Screen real estate is saved*, leaving more room for applications and diminishing competition for the user's attention. Navigation poses fewer demands as all physical controls are 'top-level.'
- *More efficient input* is possible, since a physical control's form factor can more closely match the needs of the interaction. Consider, for example, that a rheostat makes a better volume control than a using a mouse to control a GUI trackbar because it constrains the user's actions along just one dimension.
- *Two-handed input* uses are possible when the dominant hand controls a mouse while the other hand controls the (better constrained) physical device.
- *Can be brought 'ready to hand'* as needed and pushed to the periphery when not needed [18].
- *Spatial memory is better used.* Physical controls do not move about the workspace of their own accord. By contrast, GUI controls are often repositioned as an unexpected consequence of some unrelated user action.
- *All of a person's abilities are used.* Consider an electric fan instead of a GUI progress bar to illustrate the progress

made on a lengthy operation. The fan blows harder as the process nears completion. While a GUI progress bar relies solely on the visual sense, the fan's output is perceived by many senses: sight, hearing, and touch.

Given these advantages, why aren't physical controls more prevalent in modern interfaces? Besides cost-related factors, physical interfaces scale poorly. Having hundreds of devices—one for each application function—is simply impractical. Also, physical controls are not as malleable as graphical controls and are quickly rendered useless when one updates his/her software or switches to a competitor's product. Consequently, there is substantial pressure to keep the number of physical controls small.

Although it has been repeatedly shown that people use only a small subset of the large number of functions available in most productivity applications [2], this subset differs considerably from user to user [13] (beyond a few universal functions like cut and paste). Thus, it is not possible to determine beforehand which functions should be mapped onto physical devices.

Despite these problems, the advantages of physical controls motivate our desire to re-introduce them into the interface. We believe this can be accomplished through *customizable physical user interfaces*, the main idea of which is:

> …to allow a person to easily bind a function from an application to a physical device, and invoke the function through that device or see its state displayed on it.

We also believe that customizable physical user interfaces will be realistic only if they work with existing unaltered applications. These could include not only well known applications (e.g., Microsoft Office) but also niche products. In either case, source code modifications should not be needed to customize them with physical devices.

In this paper, we describe a software package for customizing existing applications with physical interfaces. Our approach is to 'tap in' to functions exposed by graphical controls, and to bind the widget semantics to physical controls with similar properties.

To explain, we first describe what we have built, as seen from an end-user's perspective, using various example physical interface customizations. We then transpose the examples to show them from an end-programmer's perspective. We follow this with a description of our package's internals, which raises issues relevant to makers of windowing systems and GUI toolkits. We conclude by describing a small representative sample of the interface design possibilities afforded by our architecture and providing an historical overview of related work.

WHAT WE BUILT: AN END-USER'S PERSPECTIVE

Our architecture allows one to craft many kinds of physical interface customizations. In this section, we show by examples what an end-user may see and what they must do to customize a particular set of controls.

a) a physical button
b) its on-screen counter-part
c) dialog to select a menu item
d) modified on-screen button and the invoked application

Figure 1: Storyboard interaction showing how one customizes and uses a physical button.

Example 1. The button

Our first simple example illustrates a single customizable push button. Figure 1a shows the physical button. Figure 1b displays the on-screen controls that a person would use to customize the button: the annotated control on the left is called a *widget picker*, while the button on its right is a standard GUI button. We consider a scenario where the end-user wants to customize both the physical and GUI buttons to open a new Microsoft Outlook e-mail message.

1. The end-user drags the widget picker over the Outlook menu bar (see Figure 1b annotation). This particular picker recognizes 'command' widgets that invoke a single function, such as buttons, menus and toolbars. As the picker passes over a widget of this type it indicates the widget is selectable by highlighting it in blue and changing the cursor shape.
2. Because menu bars contain many items (commands), the drop action raises a dialog box listing all items (Figure 1c). The person selects the 'File / New / Mail Message' menu item. The dialog box disappears, and the on-screen button is automatically relabeled with the name of the menu item, i.e., 'Mail Message' (Figure 1d).
3. When the person presses either the physical button or GUI button (Figure 1d, top), a new Outlook mail message window appears (Figure 1d, bottom). Pressing

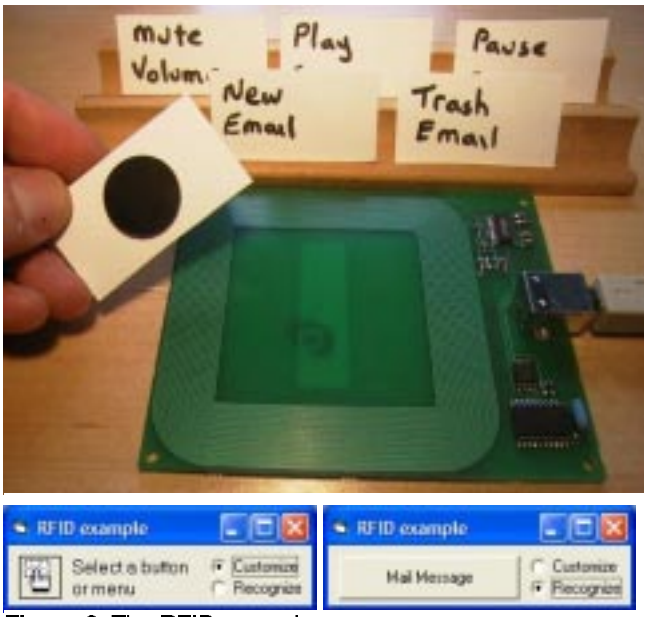

Figure 2. The RFID example.

either button is equivalent to selecting the 'File / New / Mail Message' menu command.

Example 2. RFID tags to invoke functions

Although we could extend our previous example to include many buttons, this example instead shows how one can quickly assign RFID tags to different functions. Figure 2 (top) shows an RFID reader and several RFID tags taped onto the backs of small pieces of stiff paper (one is shown turned around, with the round tag visible). At the bottom of Figure 2 is the on-screen interface. While in the 'customize' mode (Figure 2, bottom-left), a user assigns a function to a tag by first bringing it near the reader, and by then dragging and dropping the widget picker as in Example 1. To differentiate among the many tags, the user simply writes the action (in his or her own words) on the stiff paper, as shown in the figure. While in the 'recognize' mode, user invokes the function assigned to the tag by moving the tag near the reader. This displays the assigned function's name and then executes it (Figure 2, bottom-right). We see in Figure 2 the end-user has assigned RFID tags to invoke several e-mail and media player functions and has organized them on wooden trays.

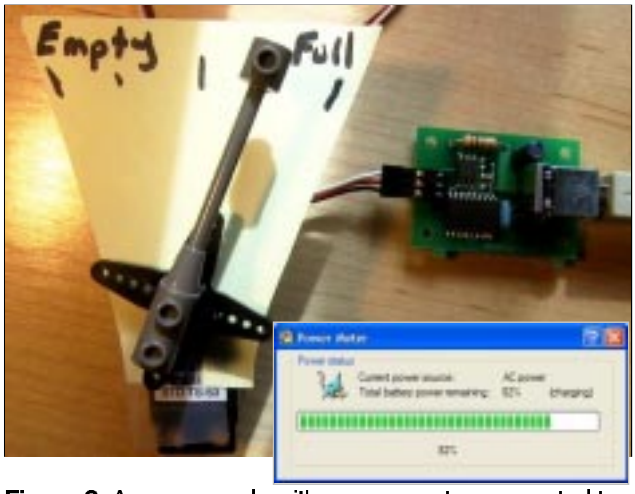

Figure 3. A gauge made with a servo motor connected to a laptop's power meter.

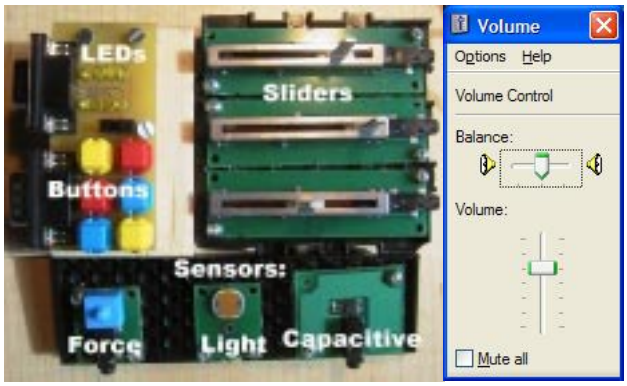

Figure 4. A console made up of many inputs and outputs used to adjust a software volume control.

Example 3. A physical gauge as a progress bar

One can also display output on a physical device. Figure 3 shows one example, where a gauge was constructed using a servo motor. The end-user has dropped a widget picker onto the progress bar that displays the charge state of his/her laptop battery (Figure 3, bottom). As a result, the motor automatically tracks the progress bar's value and rotates to a position relative to it.

Example 4. A console containing many devices

The previous examples illustrate a few types of single-device customizations. Of course, many other devices are available and we can combine them to create consoles. Figure 4 annotates an unassembled console. It includes:

- three slider potentiometers that one can attach to any on-screen trackbar (slider) or scrollbar;
- eight LEDs that one can attach to a boolean control (such as a checkbox) to monitor its state;
- six push buttons and one toggle that one can attach to any button, menu item or checkbox;
- force, light and capacitive sensors that can be attached to any widget that recognizes a continuous range of values, for example, a trackbar or scrollbar

For example, we can use this console to create a physical interface to the volume control pictured on the right. We attach the first physical slider to the volume slider, the second to the balance, and a button (or toggle) to the mute. We can also attach an LED to the mute checkbox so its light is on when the volume has been muted.

WHAT WE BUILT: A PROGRAMMER'S PERSPECTIVE

Our customizable physical interfaces architecture contains three main parts. First, *phidgets*™ are *physical widgets* used to construct a myriad of physical controls (buttons, dials, sliders, switches) and displays (gauges, lights) [3].

Second, the *widget picker* (seen by both user and programmer) and *taps* (object visible only to programmers) expose an application's functionality as controlled by its GUI widgets so that one can send directives to the function and/or get the state of the function. That is, a widget tap 'taps in' to the functionality exposed by a widget. Finally, *connector software* lets a user rapidly connect a physical device to a function exposed by the widget tap.

Physical widgets

Our physical interfaces are made with *phidgets*™ [3]. A phidget comprises a device, a software architecture for communication and connection management, a well-defined API for device programming, a simulation capability, and an optional on-screen component for interacting with the device. Phidgets are ideally suited for this project, for it means one can quickly prototype various customizations without spending effort developing special hardware, firmware, or software. Several phidgets in our toolkit are particularly well suited to this project.

- *PhidgetInterfaceKit* lets one plug in a combination of off-the-shelf controls such as those used in Figures 1 and 4. Specifically, a programmer can control through software up to eight digital output devices (e.g., LEDs and solenoids); retrieve the state of up to eight digital input devices (e.g., various types of push buttons and throw switches); and, inspect the state of various analog sensors connected to it (e.g., potentiometers, heat, force, capacitive plates and light sensors, as shown in Figure 4).
- *PhidgetRFID* is an RFID tag reader (Figure 2), where a program is notified whenever an RFID tag passes over an antenna. The notification includes the unique ID of that tag.
- *PhidgetServo* comprises one or more servo motors (one is illustrated in Figure 3) where a motor's position is easily set through software.
- *PhidgetPower* varies the amount of power sent to an attached DC device such as a motor or light.

From a coding perspective, detecting a change in phidget status is easy. We illustrate this with the physical button in Figure 1. It is connected to a PhidgetInterfaceKit represented by the `phidgetIK` programmer object. When the button is pressed, an `OnInputChange` software event is raised. We check which digital input signaled the change and its new state (True for pushed) and then take the desired action.

```
Sub phidgetIK_OnInputChange (Index, State)
   If Index = 1 and State = True Then
      'do something
   End If
End Sub
```

Similarly, PhidgetRFID raises an `OnTag` event when an RFID tag is detected near its antenna. From this the programmer can easily identify which tag was read. Example 2 from above would dynamically track these tag identifiers in an array and search the array whenever it sees a tag.

```
Sub phidgetRFID_OnTag(TagNumber)
   Select Case TagNumber
      Case TagNumber = "00041135a0" 'one tag
         'do something
      Case TagNumber = "00053343a5" 'another tag
         'do something else
    End Select
End Sub
```

Similarly, a programmer can change the state of any physical output device. The source code below illustrates how to turn on the 2nd LED in a bank of LEDs attached to the PhidgetInterfaceKit, and rotate the first servo motor controlled by a PhidgetServo to the 90° position.

```
phidgetIK.Output(2) = True
phidgetServo.MotorPosition(1) = 90
```

Using these phidgets, we can quickly create quite different control consoles. For example, we constructed the push button in Figure 1 in minutes: we cut off the top of a plastic bottle, drilled a hole in the cap, and embedded a switch in it. We gained access to the switch's state by plugging it into the PhidgetInterfaceKit. The more complex console in Figure 4 uses sliders, buttons and rocker switches all connected to a PhidgetInterfaceKit. The RFID tags (Figure 3) are read with a PhidgetRFID, and the mechanical gauge in Figure 2 is actually a PhidgetServo.

Widget picker and tap

Accessing existing system and application functions is difficult. From a technical perspective, neither operating systems nor applications offer a convenient and standardized way to access their functionality without a large amount of programming effort. Thus, our solution must abstract away from individual application or GUI toolkit differences and shield the programmer from the details of the lower-level window manager/operating system interfaces. Furthermore, it is not even clear what we mean by 'functionality' as it can be defined a number of different ways: we discuss this aspect later in this paper.

Our *widget picker* and *taps* package makes it easy for an end-user to select a particular GUI widget and for a programmer to access the semantic functions of that widget. Specifically, our *widget picker* is an interactive ActiveX control that lets an end-user select a graphical widget already on the display (e.g., as illustrated by the left control in Figure 1b). When a widget is picked, the programmer is provided with a *widget tap* object that exposes the interface of that particular widget. The widget tap can invoke the function controlled by the selected widget and retrieve information about the widget's state.

Currently, we have implemented three classes of widget taps. Each widget tap may be used with a number of different types of widgets, each with distinct visual appearances and interaction paradigms yet all sharing the same logical operation.

CommandWidgetTap for GUI widgets that invoke a single application function, e.g., push buttons, menu items, and toolbar buttons.

```
'This object will expose a command-type widget
Dim wTap As CommandWidgetTap

Sub Form_Load()
   'Initialize the picker to select only this type of widget
   wPicker.TapClass=New CommandWidgetTap
End Sub

'A person selected a widget
Sub wPicker_Pick (Tap As Object)
   Set wTap = Tap
   Button.Caption = wTap.Name
End Sub

'A person pressed the button
Sub Button_Click()
   wTap.Click
End Sub
```

Figure 5. The code behind Example 1's Button.

RangeWidgetTap for GUI widgets that are used to select a discrete value between a minimum/maximum range: track bars (sliders), scroll bars, and progress bars.

ToggleWidgetTap for GUI widgets such as checkboxes that toggle between yes/no values.

We show how this works by walking through the complete code in Figure 5 that sits behind the example customizable button illustrated in Figure 1. For now, we will only show how we connect the on-screen button at the right of Figure 1b to any command widget picked by the end-user.

1. The programmer constructs the window in Figure 1b by dropping in: a widget picker named `wPicker` and a button named `Button` with its caption set to the string "Select a button or a menu".
2. In code, the programmer declares a widget tap called `wTap`. Both the `wTap` and `wPicker` are initialized to understand the semantics of a 'command' widget. The user can now drag and drop the `wPicker` over any supported widget (e.g., buttons, menus, and tool bars).
3. When the end-user selects a widget (such as the "New Mail Message" menu item), the `wPicker_Pick` callback is automatically invoked, providing a handle to a widget tap object (`Tap`) that is now connected to the selected widget. To use it, the programmer assigns this handle to the declared `wTap` variable, thereby exposing properties and methods specific to command widget taps. For example, in Figure 5 we see that the programmer has retrieved the name of the command widget (i.e., the text label of the menu item) via the `wTap.Name` property and has assigned it to the button's caption property.
4. When the end-user presses the GUI button, the standard `Button_Click` event is raised and the programmer calls the `wTap.Click` method. This invokes the equivalent semantic operation on the corresponding widget, i.e., it invokes the application function that would arise out of clicking the widget. In this case, a new mail message will appear.

Of course, this is just a simple example, and far more interesting ones can be constructed. Other (quite different) widget classes, such as sliders and checkboxes, can be accessed and controlled in a very similar manner. What is important is that our picker control and tap objects give the programmer access to the semantics of *any* recognized widget in *any* application. The programmer needs no access to the application source, nor does he or she need any prior knowledge of that application.

Connector software

The final step is to connect the widget(s) exposed by the widget tap to the phidget(s). This connector software is implemented by programmers using our picker/tap package and the phidget library. Programmers build the physical interfaces, deciding which and how many devices to use, how they are packaged, and how they may be represented on the screen (e.g., as a mimic diagram).

To completely implement the push button example in Figure 1, one adds the following code (as well as some phidget initialization code) to Figure 5:

```
Sub phidgetIK_OnInputChange (Index, State)
   If Index = 1 and State = True Then
      wTap.Click
   End If
End Sub
```

As before, `phidgetIK` is the PhidgetInterfaceKit object declared by the programmer. Connecting an analog physical device (e.g., a physical slider) to a 'range' widget (e.g., a GUI slider) is just as easy. Assuming the physical slider is the first sensor:

```
Sub phidgetIK_OnSensorChange (Index, Value)
   If Index = 1 Then
      wTap.Value = Value
   End If
End Sub
```

Finally, connecting an analog physical output device such as a servo motor to (say) a 'range' widget such as the battery recharge progress bar on a laptop computer is almost as easy. The `RangeWidgetTap` provides a means to query the current progress bar value. One can set up a timer to poll this value, and convert it to an angle between 0° and 180° used to set the position of a servo motor.

```
Sub tmrPoll_Timer()
   ratio = (wTap.Value - wTap.Min) / _
               (wTap.Max - wTap.Min)
    Servo.MotorPosition(1) = 180 * ratio
End Sub
```

WHAT WE BUILT: INTERNALS

Our architecture provides two fundamental components: the phidget hardware and software [3], and the widget picker/tap software. Programmers apply the API of each to build software that bridges physical and GUI controls. Thus our discussion of the internal workings of our architecture will concentrate on the widget picker/tap component.

As mentioned, the WidgetPicker control allows the end-user to interactively choose a widget: it is responsible for enumerating widgets on the display and getting the user to select just one. WidgetTap objects allow programmers to inspect and manipulate the selected widget. The picker and taps work together to identify which on-screen widgets are of a suitable class.

The previous discussions also illustrated that a given widget tap class can support several logically similar yet visually and interactively dissimilar widget types. In a sense, widget taps are "meta-classes" of widgets. With our architecture, these meta-classes are easy to create, and existing ones are easily extended to include new GUI widgets that fit the class semantics.

We implemented the picker/tap package on Windows XP as follows. As the user drags the WidgetPicker about the display, the picker uses standard Windows APIs to discover the widget beneath the mouse pointer. It passes a handle to that widget to the programmer-specified WidgetTap class, which in turn uses windowing system APIs to discover the class name (and, in some cases, the class-specific styles) of that widget. If the widget meets the criteria set by the WidgetTap class, the WidgetPicker highlights the widget on the display as being a suitable drop target.

A lot of effort is spent deciding if a widget is of a supported class. This is because many widget classes with similar looks and feels are known to the windowing system by very different class names. Often, GUI toolkits will make it possible for the programmer to create new widget classes that override and extend existing ones. This is *inheritance* and it is widely used, e.g., in the Java Swing toolkit and the gtkmm toolkit for GTK+. Some toolkits provide equivalents of system-supplied widget classes that have been tweaked to work within the toolkit framework (e.g., Microsoft .NET Windows Forms).

The problem is that the underlying windowing system is ignorant of these inheritance relationships. Thus, while the system-supplied push button widget class is named BUTTON, the virtually identical Microsoft .NET counterpart is called WindowsForms10.BUTTON.app1. The relationship between base and derived widget classes is unknown to the windowing system and our widget taps (except through trial and error experimentation). Worse, in some toolkits (e.g., GTK+), all widgets are known to the windowing system by the exact same class name. Thus, to the windowing system, a GTK+ text box is the same as a GTK+ push button.

Beyond participating in the selection of a widget, the WidgetTap must also invoke the functionality provided by the widget. System-supplied widgets communicate with their containers (i.e., parent windows) through the exchange of well-documented messages. The messages are exchanged using the same underlying mechanisms the windowing system uses to deliver mouse/keyboard input events to a widget. For example, clicking a button widget sends (loosely speaking) a `BN_CLICKED` notification message to its container; the container then decides what action to take in response to this notification. Thus, when tapping in to a system-supplied button widget, the `Click` method of CommandWidgetTap mirrors this process by sending a `BN_CLICKED` message to the button widget's container, as though it had come from the widget itself.

While this sounds simple, it is difficult to do in practice. For example, if application- or widget-specific messages carry pointers as arguments then they cannot be marshaled between processes as the windowing system is largely ignorant of the format of these widget-specific messages. Messages have no meta-data to describe parameter types and formats. Careful use of various system-supplied hooks, however, will circumvent process boundaries.

Each application/toolkit may implement its own widgets with idiosyncratic class names; these widgets are ignored if unknown to a WidgetTap class. Thus, the user may not be able to pick whole sets of widgets, even if the widget 'looks' and 'feels' like a system-supplied one. Moreover, some toolkits use *lightweight widgets*: those that (as a resource-use optimization) have no representation known to the windowing system. Because our widget picker relies solely on information available from the windowing system, it is thus ignorant of these lightweight widgets.

One promising way around these issues draws upon the *accessibility* facilities of the operating system. These features allow applications to support assistive technologies that help users with mobility, hearing, or visual impairments by giving them alternate interfaces. This is very similar to what we wish to accomplish with our package. For example, these accessibility facilities provide a standard way to access *command bars*: the menu bars and tool bars used in Microsoft Office products. The individual buttons on command bars are lightweight widgets, and each is registered with the system's accessibility facilities. Enumerating the accessible widgets is precisely how the dialog of menu items shown in Figure 1c is made.

Sadly, there is no means to automatically register all widgets with the accessibility facilities. Furthermore, the accessibility APIs themselves have poor support for output (i.e., user feedback). Worse, a modest amount of programmer effort is needed to incorporate accessibility features into applications, and developers are often unaware of these facilities and their value. Not surprisingly, only a handful of existing applications presently leverage accessibility features (e.g., Microsoft Office). The U.S. government's strong commitment to the development and deployment of assistive technology [16] does offer hope that this situation will improve with time.

We should stress that this implementation is far from ideal. For example, most windowing systems do not use instance-invariant widget identifiers, and so if a widget is destroyed

and then subsequently recreated there is no convenient means to readily derive the new widget instance given the old one. Consequently, with our package, if a user selects a widget and then restarts the application, the widget tap chosen is rendered useless. Although some toolkits provide instance-invariant widget names (e.g., widget paths in Tcl/Tk) there is little consistency between them and the underlying windowing system remains ignorant of these names.

In a subsequent section we summarize the implications to windowing system and GUI toolkit makers prompted by issues related to our package implementation.

IMPLEMENTATION DESIGN SPECTRUM

Our goal was to customize a broad base of applications already in wide use to work with physical controls. As this in turn implies no modifications may be made to the existing applications, we designed our package to expose existing application functionality to programmers so that they can write software to customize these applications with physical interfaces. Our solution represents only one point on a spectrum of possible solutions, each with its own set of advantages and issues. In this section, we present this spectrum of approaches to the problem of accessing and exposing application functionality.

In general, we can access an application's functionality using a range of syntactic to semantic methods.

At one extreme, *syntactic access* simulates raw user input: that is, the syntax of the interaction. Syntactic access closely mirrors the low-level motor operations performed by users as they interact with the application, e.g., click the mouse at display coordinates (*x*,*y*), press the Ctrl+S key combination. Programmable function keys of older character terminals illustrate this. Pressing a function key inserts a user-programmable character stream into the terminal. An advantage of syntactic access is that it exactly mirrors how a user expresses a function, i.e., 'I did *X* to get *Y*; the physical device just has to do the same thing'. The problem is that syntactic access is difficult to implement reliably. It is ignorant of the modes that the application may be in, and provides little opportunity to assess an application's feedback. It also fails in GUIs when things do not appear in constant locations or when the interface is rearranged from one invocation to the next. Also, while syntactic access lets us emulate user input, it does not give easy access to an application's output, e.g., the name of a button visible on the display or the invisible minimum and maximum properties of a trackbar (slider).

At the other extreme, *semantic access* uses high-level abstractions and hooks that expose access to the components and operations performed by an application, i.e., the semantics of the interaction. Hooks, automation APIs and scriptable object models used by macros are examples of this kind of access. The advantage of semantic access is that it provides reliable access to an application's functionality, and provides the ability to respond to (significant) application events. Ideally, these functions and events will mirror the high-level cognitive operations performed by users, e.g., save the document, delete the e-mail message.

Unfortunately, semantic access has disadvantages. First, the only functions available are those that the application programmer *a priori* decided to supply through the application's high-level object models and automation interfaces. Second, the provided semantic functions may not match how the end-user actually thinks of their application. For example, a user's view of a 'function' may actually be a chain of automation calls and logic, e.g., as captured in a macro. Lastly, these API calls are hidden from the user—they do not have a graphical, on-screen presence—and are not written in the user's language (they are written in the programmer's language). Given this, how can the end-user specify which function he/she wishes to invoke?

Straddling these two extremes is an application's graphical interface, e.g., the types of widgets, their positions, styles, and properties. *Widgets simultaneously reveal both the semantics of their functionality and the syntax of their invocation.* Information about available functions normally provided intentionally through hooks and automation APIs are also expressed as a consequence of the GUI widgets presented to the user. For example, we can infer that a menu item that displays the text "Save" invokes the application's Save functionality. A very important advantage of using widgets to access functions is that it is in the language of the end-user. This makes it very easy for the end-user to specify a desired action simply by selecting the widget that invokes that action. Another advantage is that the invoked function may actually execute a complex bit of application code; it is not limited to invoking only primitive automation calls.

While a good solution, widget disadvantages concern implementation. When an application's widgets are of known standard types, we can use well-documented system-supplied hooks and APIs to access them on a level that is more semantic than syntactic e.g., invoke the 'Save' menu item *vs.* 'click the second item of the first sub-menu of an application's main menu bar.' This approach, sadly, cannot be applied universally as custom widgets used in an application may have no publicly viewable documentation. Given the diversity in application implementation, it is quite likely that no matter the technique used to gain access to functionality of one application, it will eventually fail for some other application.

DISCUSSION

Implications to application framework designers

We summarize here the issues raised in the previous discussions that are most relevant to windowing system and GUI toolkit makers.

1. *Lack of instance-invariant widget identifiers* makes it hard to find a widget after it has been recreated.
2. *Implementation inheritance relationships are not known* to the windowing system. GUI toolkits do not make each derived widget class look distinct to the windowing system.
3. *Accessibility features are not automatically applied* to every widget, neither by the windowing system nor GUI toolkits. Accessibility features are often poor at communicating application feedback.
4. *Lightweight widgets are inaccessible* using standard windowing system APIs.
5. *Widgets are not self-describing*, i.e., they do not reveal the semantic operations they support. This is especially true of undocumented custom widgets.
6. *Windowing-system events carry no metadata* about parameter types and formats. There is no reliable means to learn of application-specific events.

Design possibilities

While the basic idea of a customizable physical interface is fairly simple, it opens the door to many design possibilities. A few are listed below, although we believe that many more compelling examples remain as yet undiscovered.

Construction kits. Instead of giving end-users pre-assembled physical consoles, one can give them a construction kit that, for example, includes a PhidgetInterfaceKit and a multitude of switches and sensors mounted on Lego™-like blocks. End-users can then assemble their own custom panels using whatever controls they wish. On the software side, we can easily create movable controls representing the eight digital inputs and outputs, and the four sensor inputs. Users can match the type of input with what they actually attached to the PhidgetInterfaceKit through a shortcut menu, e.g., a particular sensor input could be set to look like a slider or a force sensor. Finally, users can position these movable controls on the display so they match the arrangement of physical controls thus creating a mimic diagram.

Interfaces for people with special needs. While many people suggest that computers should help those with special needs, most of today's computers tend to have built-in help for only particular types of disabilities e.g., low vision. One of the problems is cost: unless many people have a particular type of disability, it is just too expensive to build in accessibility features. Customizable controls can lower this cost, as it would be fairly easy to create a custom physical control panel that (say) gives people with fine motor control problems easier access to their applications. Similarly, we can map an application's state onto output devices to make them more perceivable (e.g., mapping a progress bar to a fan, as mentioned earlier, benefits those with visual and/or aural impairments).

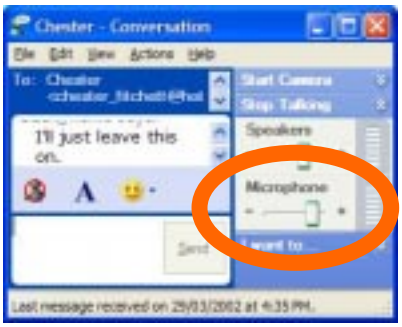

Figure 6. Controlling Windows™ Messenger.

It is important to point out here that assistive technology (AT) product makers often use approaches similar to what we describe here. However, our widget picker/taps package is not a product focused on one or a few particular kinds of ability impairments, nor is it wholly restricted to the AT domain.

Customizable reactive environments. A reactive environment is one where computers sense the environment and take action depending on what is sensed. There are now many examples of reactive environments. e.g., those reported in the ubiquitous and context aware computing literature. However, most are hard-wired to particular environments and situations. In contrast, customizable physical controls would make it simple for a person to 'build' his/her own—albeit limited—reactive environment.

For example, let us say two people have created a voice connection through an instant messenger client (Figure 6) and wanted to leave it running. Because of privacy concerns, both parties only want the microphones to be enabled when people are close to their computers. They can easily achieve this by using a capacitive sensor instead of a physical slider, and embedding that into the chair (this would set the microphone to maximum when someone sits in the chair and off otherwise). Alternatively, one can plug in an ultrasonic proximity sensor and place it atop the computer. In this case, the microphone is sensitive when one is nearby, but decreases in sensitivity as one moves away (e.g., one may only receive a slight murmur of conversations occurring away from the computer).

Linking smart appliances to applications. Looking ahead, as our appliances get smarter, there is nothing to stop them from becoming wireless physical devices that control similar applications on desktop PCs. For example, the controls of a physical MP3 player could be linked to a media player application running on a traditional PC desktop. If one presses a mode button, all the physical player controls (e.g., volume control, play, seek) could then be used to operate the media player application. If the MP3 player device was Bluetooth™-enabled, it could become a kind of wireless remote control for the PC.

Figure 7. Ivan Sutherland interacting with Sketchpad on the TX-2 computer console.

Figure 8. Engelbart's mouse-keyset combination, including a one-handed chorded keyboard (from www.bootstrap.org)

HISTORY AND RELATED WORK

In 1963, Ivan Sutherland [9] demonstrated Sketchpad, the very first interactive graphical user interface. Films of Sketchpad [7] highlight how people use a light pen to manipulate drawings, which foreshadowed the widespread use of pointing devices for graphical interaction. What is often overlooked in these old demonstrations is that almost all user actions involved two hands—a person would simultaneously manipulate large banks of physical controls as they used the light pen. These physical controls had dedicated functions that modified the light pen actions, for example, to specify start and end points of lines, to make lines parallel or co-linear, to delete existing lines, to indicate centers of circles, to store drawing objects, and so on [8]. Physical controls were also used for other interactions, such as zooming and rotation of objects. As seen in Figure 7, these controls surround and dwarf the 7-inch display containing the Sketchpad interface, and comprise physical knobs, push-buttons and toggle switches.

Sketchpad's use of physical interaction techniques was not atypical, as many computers of the 1960's and earlier often came with consoles packed with physical controls. For example, the operator console of the IBM Stretch machine, built in 1961, was immersed in a myriad of dials, lights, meters and switches.

In 1967, Douglas Engelbart introduced a new way of interacting with technologies, where almost all physical controls were replaced by the mouse and the two keyboards pictured in Figure 8. Similar to Sketchpad, keyboard 'commands' (instead of physical button presses) would modify mouse actions [1].

While Engelbart's system did away with most special purpose physical controls, they appeared again as special purpose function keys in the Xerox Star ([6]). Because there were relatively few function keys on the keyboard and a fairly large repertoire of system commands, the Star inventors came up with the notion of 'generic commands:' a small set of commands, mapped onto the function keys in Figure 9 that applied to all types of data. The active selected object interpreted these function key presses in a semantically reasonable way.

Later desktop computers, as popularized by the Apple Macintosh in the early 1980's, reduced even these special-purpose keys by replacing them with the now-familiar on-screen graphical user interface widgets. From this point on, graphical user interface controls reigned supreme on desktop computers. While most keyboards do allow some keys to be reprogrammed (including function keys), they are no longer a dominant part of interaction. In the last decade, the only other physical devices prevalent on desktop computers were games controls. Typically a generic input device (such as a joystick or steering wheel) controls a broad class of gaming applications, although one can also buy dedicated controls for particular games.

Recent research in human computer interaction has reintroduced physical controls. Many of the examples involve controls for new classes of computers e.g., tilting and panning actions for scrolling through items on a PDA [4]. Others try to bridge physical world objects with computer objects through tagging and tracking [17]. Perhaps the closest to our work is tangible media [10,11], which describes how physical media can be attached to digital information and controls. An excellent example is Ullmer, Ishii and Glas's mediaBlocks [10]. Similar to our Example 2, their mediaBlocks (electronically tagged blocks of wood) can be assigned to particular functions and bits of information, further depending upon the location of the block reader.

Figure 9. Star's left function key cluster.

There are many more exciting examples of how new technology can use physical devices (e.g., [15]). Almost all of them, however, interact with special purpose software rather than commonly used

applications, thus limiting their immediate use in daily life. Overcoming this serious limitation was one of the motivations behind our work

SUMMARY

Customizing existing applications with physical interfaces allows us to immediately realize very diverse design opportunities for accessible, tangible, and context-aware computing. This technique certainly does not apply to all of the capabilities afforded by an application. However, we feel that the pendulum has been swung too far, and applications are now so dependent in GUI widgets that we have lost the benefits of judicious application of physical controls.

We presented in this paper our notion of customized physical interfaces to existing applications. We described our widget picker/taps package that allows programmers to seize upon this design idea, and offered examples demonstrating its use. In implementing this package we have identified a number of impediments that could be addressed by windowing system and GUI toolkit makers.

Our future work in this area will focus on finding solutions to the architectural problems presented here, in particular the problem that widget taps are rendered useless if the corresponding GUI widget is destroyed. We will also focus on exposing application feedback so that it may be rendered using physical devices and techniques borrowed from calm computing.

Software and hardware availability. Phidgets hardware and software is available through www.phidgets.com. The widget picker/taps package and examples will be available summer 2002 at http://www.cpsc.ucalgary.ca/grouplab/.

Acknowledgements. The Microsoft Research Collaboration and Multimedia Group, the National Sciences and Engineering Research Council of Canada, and the Alberta Software Engineering Research Consortium partially funded this work.

REFERENCES

1. Engelbart, D. and English, W. A Research Center for Augmenting Human Intellect, *AFIPS Conference Proc Fall Joint Computer Conference* (33), 395-410, 1968.
2. Greenberg, S. *The computer user as toolsmith: The use, reuse, and organization of computer-based tools.* Cambridge University Press, 1993.
3. Greenberg, S. and Fitchett, C. (2001) Phidgets: Easy Development of Physical Interfaces through Physical Widgets. Proc ACM UIST 2001, 209-218, ACM Press.
4. Harrison, B., Fishkin, K., Gujar, A., Mochon, C. and Want, R. Squeeze Me, Hold Me, Tilt Me! An Exploration of Manipulative User Interfaces. *Proc ACM CHI'98*, 17-24,1998
5. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc. ACM CHI'97*, 234-241, 1997.
6. Johnson, J., Roberts, T., Verplank, W., Smith, D., Irby, C., Beard, M. and Mackey, K. The Xerox Star: A Retrospective. *IEEE Computer* 22(9), 11-29, 1989.
7. Kay, A. (1987) Doing with images makes symbols. Distinguished lecture series, Apple Computer.
8. MIT. Sketchpad. ACM CHI'83 Video Program in SIGGRAPH Video Review Issue 13. 1983.
9. Sutherland, I. Sketchpad: A man-machine graphical communications systems, Proceedings of the Spring Joint Computer Conference, 329-346, Baltimore, MD: Spartan Books, 1963.
10. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc. ACM CHI'97,* 234-241, 1997.
11. Ishii, H., Mazalek, A., Lee, J. Bottles as a minimal interface to access digital information. *Extended Abstracts of ACM CHI*, 2001.
12. Kaminsky, M., Dourish, P., Edwards, K. LaMarca, A., Salisbury, M. and Smith, I. SWEETPEA: Software tools for programmable embodied agents. *Proc. ACM CHI,* 144-151, 1999.
13. McGrenere, J., Baecker, R. and Booth, K. An evaluation of a multiple interface design solution for bloated software. ACM CHI 2002 [CHI Letters 4(1)], 163-170.
14. Resnick, M. Behavior construction kits. *Communications of the ACM* 36(7), 64-71.
15. Ullmer, B., Ishii, H. and Glas, D. mediaBlocks: Physical Containers, Transports, and Controls for Online Media. ACM SIGGRAPH'98, 379-386, 1998.
16. Government of the United States of America. Section 508 of the Rehabilitation Act (29 US.C. 794d, Public Law 10-24). http://www.section508.gov.
17. Want, R., Fishkin, K., Gujar, A. and Harrison, B. Bridging Physical and Virtual Worlds with Electronic Tags. *Proc ACM CHI'99*, 370-377, 1999.
18. Winograd, T., and Flores, F. Understanding Computers and Cognition: A New Foundation for Design, Ablex, 1986.